AGGIORNAMENTO A JAVA 17 DE IL NUOVO JAVA

(www.nuovojava.it)

[Versione 1.0 09/2022]

EDITORE ULRICO HOEPLI MILANO

Capitolo extra "Aggiornamento a Java 17"

Introduzione

In questo capitolo supplementare, verranno descritte le più interessanti caratteristiche introdotte nelle versioni 16 e 17 di Java. Ricordiamo che "Il nuovo Java" è aggiornato alla versione 15, ma con lo studio di questo documento possiamo aggiornare le nostre conoscenze alla versione 17 in breve tempo. In realtà, era prevista la pubblicazione da parte di Hoepli di un secondo volume per aggiornare il libro alla versione 17. Tale versione ha però introdotto una quantità di novità non sufficiente a giustificare la pubblicazione di un nuovo volume. Quindi, di comune accordo con l'editore, abbiamo deciso di realizzare questo capitolo supplementare gratuito in formato PDF, liberamente scaricabile dal sito ufficiale https://www.nuovojava.it.

Questo documento sarà probabilmente aggiornato in futuro. Questa filosofia di rilascio incrementale, ci permetterà di eseguire ulteriori verifiche sulla qualità e la correttezza del contenuto. L'autore si riserva di notificare qualsiasi novità o informazione, tramite i suoi canali social ed in particolare sul canale ufficiale Telegram dedicato alle news su "Il nuovo Java", all'indirizzo https://t.me/nuovojava.
PS: non è un canale interattivo, riceverete solo notifiche da parte dell'autore.

La stesura di questo documento ha richiesto diversi giorno di lavoro da parte dell'autore che si è impegnato al massimo delle sue possibilità. Tuttavia, non ha potuto usufruire del supporto dell'editore, in particolare sulla formattazione del documento e sulle revisioni. Nel caso trovaste refusi, o per qualsiasi altro tipo di segnalazione, è possibile scrivere direttamente all'autore all'indirizzo: claudio@claudiodesio.com. È anche possibile contattarlo tramite i più importanti social network e sul sito personale dove è presente un blog ed è possibile iscriversi alla newsletter per essere sempre aggiornati su novità, corsi, iniziative, articoli e tanto altro:

Internet (blog, corsi, newsletter, etc.): https://www.claudiodesio.com
Facebook: https://www.facebook.com/claudiodesiocesari
Twitter: https://twitter.com/cdesio
LinkedIn: https://www.linkedin.com/in/claudiodesio
YouTube: https://www.youtube.com/claudiodesiocesari
Instagram: https://www.instagram.com/nuovojava
https://www.instagram.com/javaforaliens
https://www.instagram.com/cdesio

Buon lavoro!

Claudio De Sio Cesari

Capitolo 21 (Extra)

Aggiornamento a Java 17

Obiettivi:
Al termine di questo capitolo il lettore dovrebbe avere le informazioni necessarie per aggiornare le proprie conoscenze alla versione 17 di Java. In particolare dovrebbe:

- ✓ Conoscere le novità delle caratteristiche che sono state ufficializzate nelle versioni 16 e 17 (unità 21.1).
- ✓ Saper creare un file di installazione per applicazioni Java tramite il nuovo tool jpackage (unità 21.2).
- ✓ Comprendere la nuova caratteristica in anteprima introdotta in Java 17 nota come pattern matching per `switch` (unità 21.3).
- ✓ Conoscere le novità secondarie introdotte nelle versioni 16 e 17 (unità 21.4).

In questo capitolo extra, aggiorniamo "Il nuovo Java" con le novità più rilevanti delle versioni 16 e 17. Ricordiamo che la versione 17 è la terza versione **LTS** (acronimo di **Long Term Support**, ovvero **supporto a lungo termine**) che succede alle versioni 8 e 11. Le versioni LTS sono caratterizzate dal fatto che Oracle offre un supporto garantito e a pagamento per applicazioni commerciali per queste versioni. Ciò non interessa alle persone che vogliono studiare Java, ma semmai alle aziende che intendono utilizzare il JDK di Oracle ed hanno intenzione di rimanere con la stessa versione LTS in produzione per alcuni anni. Tuttavia, Oracle proprio con questa versione ha introdotto una nuova licenza gratuita, ma dato che l'aggiornamento delle licenze è in continua evoluzione, come già fatto nell'appendice A, vi rimandiamo a questo link per tutti gli aggiornamenti futuri: https://www.oracle.com/java/technologies/java-se-support-roadmap.html.
In qualsiasi caso, utilizzando l'OpenJDK non avremo mai nessun dubbio sulla gratuità della licenza. Infatti l'Oracle JDK e l'OpenJDK sono del tutto equivalenti tranne che per pochi trascurabili dettagli.

Un'altra novità riguarda la prossima versione LTS, che sarà la versione 21 la cui data di pubblicazione è prevista per settembre 2023, e non più come previsto in precedenza la versione 23 che dovrebbe essere disponibile a partire da settembre 2024. Da Java 17 in poi quindi, avremo una versione LTS ogni due anni invece che ogni tre.

In realtà le novità davvero interessanti rispetto alla versione 15 non sono tante. Sono state ufficializzate alcune caratteristiche in anteprima come i tipi record, il pattern matching for `instanceof` (dove ci accenneremo fugacemente anche a Java 19) e i tipi sealed con qualche piccola modifica. La maggior parte delle novità riguardano il porting di Java su piattaforme come Alpine Linux e le versioni a 64 bit del JDK per Windows e Mac, la migrazione del progetto OpenJDK a GitHub, la rimozione di alcuni moduli poco utilizzati dagli utenti relativi a GraalVM che rimane un progetto a parte, l'introduzione di progetti ancora sperimentali (incubazione) che migliorano la gestione della memoria, la robustezza del linguaggio e la sua interazione con linguaggi nativi, ed altre novità che poco impattano sullo studio del linguaggio.

Le novità più interessanti riguardano sicuramente le ufficializzazioni delle caratteristiche che erano in anteprima in Java 15, la possibilità di creare dei programmi di installazione per applicazioni Java per le piattaforme supportate con `jpackage`, l'introduzione come caratteristica in anteprima del pattern matching per il costrutto `switch` e poco altro.

> **Questo documento è stato progettato per aggiornare i contenuti del libro "Il nuovo Java" alla versione 17. Per tale ragione richiama spesso argomenti già spiegati nei paragrafi del libro. Per alcuni argomenti verranno linkati anche alcuni articoli presenti sul blog dell'autore claudiodesio.com per facilitare la consultazione di questo documento, soprattutto per chi non ha la possibilità di consultare il libro cartaceo durante lo studio di questo capitolo extra.**

21.1 Ufficializzazioni delle feature preview

Nelle versioni 16 e 17, sono state ufficializzate alcune caratteristiche in anteprima già discusse all'interno de "Il nuovo Java". In particolare nella versione 16 sono state formalizzati il *pattern matching per `instanceof`* e i tipi *record*. Nella versione 17 invece i tipi sealed (classi ed interfacce sealed) sono considerati ufficialmente una nuova caratteristica di Java. Tali ufficializzazioni rendono quindi inutile specificare opzioni supplementari per abilitare le caratteristiche in anteprima nelle fasi di compilazione ed esecuzione dell'applicazione.

> **Per maggiori informazioni sulle caratteristiche in anteprima è possibile consultare l'articolo relativo su claudiodesio.com.**

21.1.1 Novità relative al pattern matching for `instanceof`

Il pattern matching per `instanceof` (cfr. paragrafo 7.3.3) è stato ufficializzato con tre semplici modifiche rispetto alla versione in anteprima della versione 15.

> **Se non avete confidenza con questo argomento, consigliamo di rileggere velocemente il paragrafo 7.3.3 o in alternativa l'articolo di approfondimento sul blog dell'autore chiamato Pattern Matching per `instanceof`.**

Di seguito descriviamo le novità che ha portato con sé Java 16.

21.1.1.1 Terminologia

La prima modifica riguarda la terminologia. Nel libro avevamo definito un nuovo tipo di variabili locali che avevamo definito come **variabili di binding** (o **variabili di vincolo**), la cui visibilità era definita nel blocco di codice laddove era verificato il *predicato* che caratterizza il pattern matching. Gli autori di questa caratteristica sembrano aver abbandonato questa terminologia sostituendola in favore di una nuova che definisce tali variabili come **variabili di pattern** (**pattern variables**), e di conseguenza la relativa visibilità viene ora definita **visibilità di pattern** (**pattern scope**).

21.1.1.2 Variabili di pattern

Nel paragrafo 7.3.3.3 (pagina 320) avevamo specificato che le variabili di pattern, erano implicitamente costanti. In particolare avevamo visto un esempio dove provavamo a modificare una variabile di pattern nel seguente modo:

```
if (dip instanceof Programmatore pro) {
  pro = new Programmatore();
  //...
}
```

ottenendo il seguente errore di compilazione:

```
error: pattern binding pro may not be assigned
             pro = new Programmatore();
             ^
1 error
```

Questa limitazione è stata completamente rimossa nella versione 16 per ridurre le differenze tra variabili locali e variabili di pattern.

21.1.1.3 Compilatore più efficiente

Nella versione 16 il compilatore non compilerà codice dove il secondo operando dell'`instanceof` è un sottotipo del tipo del primo operando, in quanto l'espressione dell'`instanceof` restituirà sempre `true` e quindi è del tutto inutile eseguire. Questo significa che il seguente codice:

```
    public void metodoInutile(String s) throws Exception {
        if (s instanceof Object o) {
            System.out.println(o);
        } else {
            System.out.println(s);
        }
    }
```

Compilerà con Java 15, ma restituirà il seguente errore a partire da Java 16:

```
PMInstanceof16.java:3: error: expression type String is a subtype of
pattern type Object
        if (s instanceof Object o) {
              ^
1 error
```

21.1.2 Ufficializzazione dei tipi record in Java 16

I tipi record sono stati trattati in vari punti del libro ed in particolare nella seconda parte del capitolo 9. In alternativa potete consultare anche l'articolo su claudiodesio.com chiamato Tipi record per avere una panoramica sull'argomento.

L'unica novità che è stata introdotta in Java 16 non riguarda direttamente l'uso dei record, bensì è una conseguenza della loro introduzione. Questa novità sarà per la maggior parte dei lettori poco interessante e riguarda la definizione delle classi innestate (cfr. paragrafo 13.1), in particolare le classi interne, che da Java 16 possono anche dichiarare membri (variabili e metodi) statici. Prima di Java 16, solo *classi innestate statiche* potevano dichiarare membri statici, mentre le *classi interne* (ovvero classi innestate non statiche) potevano dichiarare solo costanti statiche. Questa limitazione è ora quindi caduta grazie alla definizione dei record. Infatti questi ultimi quando sono innestati sono implicitamente dichiarati `static`.

Quindi il seguente codice è valido se compilato con Java 16:

```
public class Record16 {
    public class Inner {
        record InnerRecord(String n) {}
        static int i;
        static int m() {}
    }
}
```

mentre se compiliamo il codice precedente con Java 15 verranno evidenziati i seguenti errori in compilazione:

```
Record16.java:3: error: static declarations not allowed in inner classes
        record InnerRecord(String n) {}
        ^
Record16.java:4: error: Illegal static declaration in inner class
Record16.Inner
        static int i;
                   ^
  modifier 'static' is only allowed in constant variable declarations
Record16.java:5: error: Illegal static declaration in inner class
Record16.Inner
        static int m() {}
                   ^
  modifier 'static' is only allowed in constant variable declarations
3 errors
```

21.1.3 Ufficializzazione dei tipi Sealed in Java 17

I tipi sealed sono stati ufficializzati nella versione 17 senza ulteriori modifiche rispetto alle modifiche che sono state apportate con la versione 16 che era ancora una versione in anteprima. Tali modifiche sono solamente tre e vengono dettagliate nei prossimi sottoparagrafi.

21.1.3.1 Terminologia

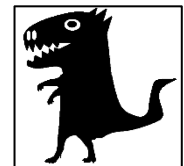
Da Java 9 in poi, non sono state più introdotte le keyword tradizionali come nelle passate versioni, bensì particolari keyword che non soffrono delle restrizioni delle keyword tradizionali, ovvero di non poter essere usate come identificatori di variabili, metodi, tipi, package, moduli.

In particolare, nel capitolo 3 abbiamo visto che le parole introdotte in Java 9 con i moduli, ovvero `module`, `open`, `opens`, `provides`, `requires`, `exports`, `to`, `transitive`, `uses` e `with` sono state definite come **parole a utilizzo limitato** (**restricted word**), perché sono considerate keyword solo all'interno del file descrittore di modulo module-info.java (cfr. capitolo 16). Quindi potremmo dire che *sono keyword solo nel caso di dichiarazione di moduli*.

La parola `var` invece, è stata inizialmente definita come **nome di tipo riservato** (in inglese **reserved type name** o relativo articolo online sul blog claudiodesio.com). È stata introdotta con la versione 10 di Java (cfr. paragrafo 3.7). Tuttavia, anche per `var`, l'unico vincolo rimane quello di non poter utilizzare questa parola come identificatore per un tipo Java (una classe, un'interfaccia, un'enumerazione, un'annotazione o un record).

Abbiamo definito come *parola ad utilizzo limitato* anche `yield`, che viene utilizzata solo nel nuovo costrutto dell'espressione `switch` per restituire un valore (cfr. paragrafo 4.4.3 o articolo online). Anche per `yield`, l'unico vincolo esistente è quello di non poter essere utilizzato come identificatore di un tipo di Java. Possiamo però usare `yield` come identificatore per variabili, metodi, package, etc. senza problemi. Stesso di discorso vale per la restricted word `record` introdotta in Java 14, e per le restricted word `sealed` e `permits`, introdotte nella versione 15, che hanno lo stesso vincolo sui nomi dei tipi Java. Invece, `non-sealed`, contiene il simbolo *trattino* - non utilizzabile all'interno di un identificatore Java.

Per evitare nuove future definizioni ed ulteriore caos, in Java 16 tutte le varie terminologie precedentemente utilizzate sono state sostituite con il termine **keyword contestuale** (in inglese **contextual keyword**), a sottolineare il loro limitato impatto sulla programmazione Java. Quindi l'introduzione dei tipi sealed ha causato una semplificazione terminologica che riguarda l'intero linguaggio che siamo sicuri sarà sicuramente apprezzata.



21.1.3.2 Classi locali

Nelle ultime righe del paragrafo 6.2.4 avevamo scritto quanto segue:

"*Un altro vincolo per le classi dichiarate `sealed` è l'obbligo di dichiarare la clausola `permits`, a meno che le sottoclassi siano dichiarate nello stesso file della superclasse. In tal caso il compilatore automaticamente eleggerà le sottoclassi contenute nello stesso file, come uniche sottoclassi possibili della classe `sealed`.*"

Ciò significa che se per esempio scriviamo all'interno dello stesso file il seguente codice:

```
public sealed class SealedSuperClass {}

final class SubClass1 extends SealedSuperClass {}

final class SubClass2 extends SealedSuperClass {}
```

Le sottoclassi `SubClass1` e `SubClass2`, saranno considerate implicitamente le uniche due sottoclassi della classe sigillata `SealedSuperClass`, senza che sia specificata la clausola `permits`. Potendo evitare di inserire tale clausola, si potrebbe pensare che eventuali classi locali (ovvero classi innestate definite all'interno di un metodo, cfr. paragrafo 13.1.1) che estendono una classe sealed, possano essere implicitamente considerate tra le uniche sottoclassi della classe `SealedSuperClass`. Ciò non è possibile. Infatti il seguente codice:

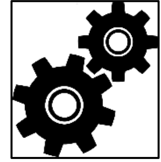
```
public sealed class SealedSuperClass {
    public void method() {
        final class LocalSubClass extends SealedSuperClass {}
    }
}

final class SubClass1 extends SealedSuperClass {}

final class SubClass2 extends SealedSuperClass {}
```

produrrà il seguente messaggio di errore:

```
SealedSuperClass.java:3: error: local classes must not extend sealed
classes
        final class LocalSubClass extends SealedSuperClass {}
              ^
1 error
```

Notare che una classe locale non è visibile all'esterno del metodo in cui è dichiarata, e quindi anche una clausola `permits` esplicita, non potrebbe elencarla tra le sottoclassi della classe.

21.1.3.3 Miglioramento dei controlli sulle gerarchie sigillate



Questo miglioramento permette alle gerarchie basate su tipi dichiarati `sealed` di essere controllate in maniera accurata già a livello di compilazione, migliorando così la robustezza del linguaggio.
Partiamo da un esempio, consideriamo il seguente codice:

```
interface Interface {}

class Class {}

public class TestHierarchy {
    public static void method(Class c) {
        if (c instanceof Interface) {
            System.out.println("c instanceof Interface -> true");
        }
    }
}
```

Abbiamo dichiarato un'interfaccia chiamata `Interface` e una classe `Class` che non implementa `Interface`. Nella classe `TestHierarchy` un metodo statico chiamato `method` utilizza l'operatore `instanceof` per controllare se il parametro `c` di tipo `Class` è un'istanza di tipo `Interface`. Anche se è evidente che `c` non possa essere di tipo `Interface` siccome la classe `Class` non implementa `Interface` questo codice viene compilato correttamente. Questo perché il compilatore considera un'applicazione Java aperta alle modifiche e nella attuale situazione è possibile che possa essere introdotta una classe come la seguente:

```
class ClassInterface extends Class implements Interface {}
```

Questa classe estende la classe `Class` e contemporaneamente implementa l'interfaccia `Interface`. Quindi se ora chiamassimo il metodo statico `method` passandogli un'istanza della classe `ClassInterface`:

```
method(new ClassInterface());
```

allora otterremo il seguente output:

```
c instanceof Interface -> true
```

Questa è la ragione per la quale il compilatore aveva compilato il codice correttamente.
Se dichiarassimo `final` la classe `Class`:

```
final class Class {}
```

allora il compilatore non compilerebbe la classe `TestHierarchy`. Infatti, non c'è possibilità di estendere la classe `Class` e quindi non potremmo mai creare una sottoclasse di `Class` che implementi anche l'interfaccia `Interface`. In questo caso otterremmo quindi il seguente errore in compilazione:

```
TestHierarchy.java:9: error: incompatible types: Class cannot be
converted to Interface
        if (c instanceof Interface) {
            ^
1 error
```

Dalla versione 16, controlli simili a quelli appena visti vengono eseguiti anche nel caso di gerarchie sigillate. Vediamo come, partendo da un esempio. Modifichiamo il codice precedente dichiarando `sealed` la classe `Class`:

```
interface Interface {}

sealed class Class permits SubClass {}
```

Inoltre definiamo la sottoclasse `final` di `Class` che chiamiamo `SubClass`:

```
final class SubClass extends Class {}
```

Infine creiamo una classe `TestSealedHierachy` con un metodo statico chiamato `method` identico a quello creato nella classe `TestHierarchy` dell'esempio precedente:

```
public class TestSealedHierarchy {
    public static void method(Class c) {
        if (c instanceof Interface) {
            System.out.println("c instanceof Interface -> true");
        }
    }
}
```

A partire dalla versione 16 il compilatore non compilerà questo codice restituendo questo errore:

```
TestSealedHierarchy.java:9: error: incompatible types: Class cannot be
converted to Interface
        if (c instanceof Interface) {
            ^
1 error
```

perché non può esistere una sottoclasse di `Class` oltre a `SubClass`, che implementa `Interface`. Se però `SubClass` fosse dichiarata per esempio `non-sealed`:

```
non-sealed class SubClass extends Class {}
```

il codice verrebbe compilato correttamente, perché potremmo estendere `SubClass` con una sottoclasse che implementa `Interface` come la seguente:

```
class SubClassInterface extends SubClass implements Interface {}
```

e quindi potremmo potenzialmente invocare il metodo `method` passandogli un'istanza della classe `SubClassInterface`:

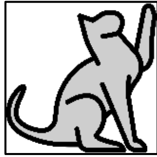
```
method(new SubClassInterface());
```

21.2 Packaging tool: jpackage

Nella versione 16 del Java Development Kit è stato introdotto un nuovo tool chiamato **jpackage** che ci permette di creare file di installazione per applicazioni Java per i principali sistemi operativi. Per file di installazione intendiamo quella tipologia di applicazione che a volte chiamiamo **installer**, che tramite una procedura guidata installa l'applicazione vera e propria sul sistema operativo.
Ma perché avere un tool del genere se Java è un linguaggio multipiattaforma? Secondo Oracle ciò permetterebbe all'utente di avere un'esperienza di installazione delle applicazioni Java *più naturale* sulle varie piattaforme di destinazione. Per esempio su Windows, dove con jpackage è possibile creare file di installazione con formato .exe o .msi.

I formati supportati per i file di installazione su MacOS sono invece .dmg e .pkg, mentre su piattaforma Linux sono supportati i formati .rpm e .deb.

In pratica questo tool potrebbe in futuro dare un nuovo impulso alla creazione di applicazioni desktop basate su Java. Con un file di installazione non avremo più bisogno di distribuire anche un runtime personalizzato con jlink (cfr. paragrafo 16.6), perché di default jpackage ne creerà uno automaticamente che ci permetta di eseguire l'applicazione sulla piattaforma dove sarà installata.



In realtà jpackage non è proprio una novità. Si tratta del risultato dell'evoluzione di uno strumento creato per installare applicazioni JavaFX chiamato javapackager, presente dalla versione 8 alla 11 del JDK. Tuttavia jpackage non supporta nessuna delle caratteristiche relative alla applicazioni JavaFX presenti in javapackager. Non supporta nemmeno Java Web Start (come faceva javapackager), visto che si tratta di una tecnologia deprecata nella versione 9 e rimossa dalla versione 11.

21.2.1 Cosa si può fare

Con jpackage possiamo quindi creare file di installazione (installer) per applicazioni Java modulari (costituite da uno o più moduli) o non modulari (costituite da uno o più file JAR). Come quasi tutti i tool del JDK, jpackage non fornisce un'interfaccia grafica, bensì si utilizza da riga di comando specificando diverse opzioni. Queste opzioni sono numerose, ed è possibile anche scriverle all'interno di un file di testo da richiamare direttamente da riga di comando per riutilizzarle.
Il tool jpackage non supporta la *cross-compilation*, che potremmo tradurre come *compilazione multipiattaforma*. Ciò significa che se per esempio vogliamo ottenere un file di installazione per la nostra applicazione Windows con suffisso .exe, dobbiamo obbligatoriamente crearlo su piattaforma Windows. Se vogliamo ottenere un file di installazione per la nostra applicazione Linux con suffisso .rpm, dobbiamo obbligatoriamente crearlo su piattaforma Linux e così via.
È anche possibile creare un'associazione sul sistema operativo per i file con un determinato suffisso, associandogli anche alcune proprietà come un'icona. Questo significa che se per esempio associassimo i file con suffisso .jpac alla nostra applicazione installato con jpackage, allora quando apriremo un file con tale suffisso, esso verrà caricato all'interno della applicazione.

21.2.2 Come si usa

Trattandosi di un tool e non di una caratteristica del linguaggio, ci limiteremo di seguito a fare un esempio pratico di utilizzo e a descriverne le caratteristiche più importanti. Creeremo quindi un file di installazione per piattaforma Windows per l'applicazione SwingMixExample.java, presentata nel paragrafo 19.5.4 (leggermente modificata per supportare il caricamento delle risorse all'interno di un file JAR).
Prima di ottenere il nostro file di installazione utilizzando il comando jpackage, eseguiremo due passi preliminari: la creazione del file JAR dell'applicazione SwingMixExample tramite il comando jar e la creazione tramite il comando jlink dell'ambiente di runtime contente solo i moduli strettamente necessari per l'esecuzione dell'applicazione.

Come sempre, tutti i comandi e codici descritti di seguito si possono trovare online nel file che contiene il codice del libro nella cartella capitolo_21, da scaricare sul sito ufficiale nuovojava.it. Un file leggimi.txt spiega come utilizzare i file batch forniti per realizzare i passi qui sotto descritti.

21.2.2.1 Creazione del file JAR

Per prima cosa compiliamo il nostro codice sorgente contenuto interamente nella file SwingMixExample.java con il seguente comando:

```
javac -d . SwingMixExample.java
```

Siccome all'interno del file sorgente è stato dichiarato il package jpackage.test, allora tutti i file .class verranno generati all'interno della cartella jpackage.test (cfr. paragrafo 2.3.2).
Andiamo poi a creare un file JAR nella cartella swingmix contenente la nostra applicazione lanciando la seguente istruzione da riga di comando:

```
jar cmvf MANIFEST.MF swingmix\SwingMix.jar resources jpackage
```

dove abbiamo usato il tool jar del JDK, specificando:

1) un file MANIFEST.MF che abbiamo creato appositamente per specificare il nome della classe che contiene il metodo `main` per rendere il file JAR eseguibile (cfr. paragrafo 19.6.3.2);
2) di creare il file SwingMix.jar all'interno della cartella swingmix;
3) di impacchettate al suo interno tutto ciò che è contenuto nelle cartelle resources e jpackage (rispettivamente le immagini che usa l'applicazione e i file .class dell'applicazione stessa).

Per assicurarci che il nostro comando abbia fatto quello che ci aspettiamo, lanciamo il file SwingMix.jar che si trova nella cartella swingmix e verifichiamo che l'applicazione parta correttamente. Da riga di comando basterà scrivere:

```
java -jar swingmix\SwingMix.jar
```

In alternativa, se avete installato correttamente il JDK (cfr. appendice B), potrete lanciare il file JAR eseguibile con un classico doppio click del mouse.

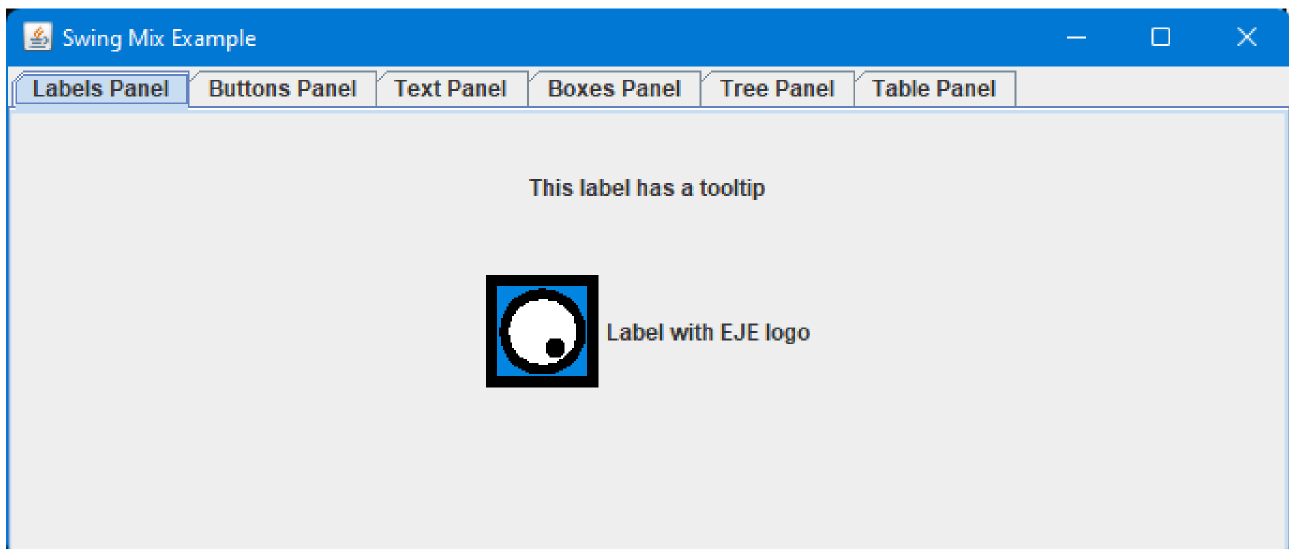


Figura 21.1 – L'applicazione SwingMix.

21.2.2.2 Creazione dell'ambiente di runtime con jlink

Questo passo è opzionale, infatti jpackage di default crea automaticamente un runtime standard per la nostra applicazione e lo installa insieme ad essa. Tuttavia possiamo ottimizzare il nostro runtime utilizzando solo i moduli che l'applicazione davvero utilizza. Nel nostro caso abbiamo deciso di creare un runtime chiamato myjre appositamente personalizzato per l'applicazione SwingMix, che contiene solo i moduli java.base e java.desktop (gli unici indispensabili). Basta lanciare il seguente comando basato sul tool jlink (cfr. paragrafo 16.6):

```
jlink --add-modules java.base,java.desktop --output myjre
```

A questo punto nella cartella corrente dovrebbe essere stata creata la cartella myjre contenente il nostro ambiente di runtime personalizzato.

21.2.2.3 Creazione dell'installer con jpackage

Solo per la piattaforma Windows, per poter utilizzare con profitto il tool jpackage, è necessario installare l'applicativo Wix Toolset, che è possibile scaricare dal sito https://wixtoolset.org/releases/, ed installare molto semplicemente.

L'ultimo passo consiste nell'utilizzare il tool jpackage per generare finalmente il file d'installazione per la nostra applicazione SwingMix. Il comando che utilizzeremo è il seguente:

```
jpackage --input swingmix --name SwingMix --main-jar SwingMix.jar
  --runtime-image myjre --icon resources\eje.ico --win-dir-chooser
  --win-menu --win-shortcut
```

dove abbiamo utilizzato diverse opzioni che descriviamo brevemente nella seguente tabella:

Notare che nella colonna OPZIONE tra parentesi abbiamo anche specificato le possibili *forme abbreviate* delle opzioni stesse (dove applicabile). Tali forme abbreviate sono del tutto equivalenti alle rispettive forme estese. Non le usiamo nei nostri esempi per questioni di leggibilità.

OPZIONE	DESCRIZIONE
--input (-i)	Specifica la cartella swingmix che contiene i file JAR da installare.
--name (-n)	Assegna un nome all'applicazione. Dopo aver eseguito l'installazione quindi troveremo l'applicazione che si chiama SwingMix tra le applicazioni installate sul sistema operativo (vedi figura 21.3).
--main-jar	Specifica quale file JAR deve considerarsi quello eseguibile, ovvero quello che contiene la classe del metodo `main` per lanciare l'applicazione. Infatti la nostra applicazione potrebbe essere costituita da diversi file JAR. Questa opzione è utilizzabile solo nel caso l'applicazione non sia modulare (infatti esiste anche l'opzione --module che verrà spiegata nella prossima tabella).
--runtime-image	Specifica la cartella contenente il runtime personalizzato. Se assente, jpackage creerà un runtime automaticamente con il set standard di moduli che usa jlink.
--icon	Specifica quale deve essere l'icona che rappresenta l'applicazione.
--win-dir-chooser	Questa opzione, se specificata, permetterà di scegliere la cartella dove installare l'applicazione durante la fase di installazione. Se non specificata, l'applicazione verrà installata nella cartella dove di solito vengono installate le applicazioni. Questa opzione è specifica per sistemi Windows.
--win-menu	Questa opzione, se specificata, aggiungerà nel menu del sistema operativo una shortcut per lanciare l'applicazione. Questa opzione è specifica per sistemi Windows.
--win-shortcut	Questa opzione, se specificata, aggiungerà sul desktop del sistema operativo una shortcut per lanciare l'applicazione. Questa opzione è specifica per sistemi Windows.

Altre opzioni interessanti e semplici da utilizzare vengono riassunte in quest'altra tabella:

OPZIONE	DESCRIZIONE
--java-options	Specifica opzioni da passare alla JVM e può essere utilizzata più volte.
--arguments	Specifica opzioni da passare al metodo `main` (gli elementi dell'array `args`) e può essere utilizzata più volte.
--module-path (-p)	Specifica una lista i cui elementi sono separati dal simbolo *punto e virgola* `;` di percorsi (relativi o assoluti) a directory contenenti moduli o a Jar modulari. Anche questa opzione si può specificare più volte.
--module (-m)	Specifica il modulo principale che contiene la classe del metodo `main`. Si usa in alternativa ad --main-jar per installare applicazioni modulari. Il modulo specificato deve essere presente nel modulepath.
--add-modules	Specifica una lista i cui elementi sono separati dal simbolo *virgola* `,` dei moduli da aggiungere all'applicazione, in modo tale da creare un runtime personalizzato senza usare jlink. Se questa opzione non è specificata, in caso di applicazione modulare verrà usato solo il modulo specificato con --module (vedi voce precedente), in caso di applicazione non modulare invece verrà usato il gruppo standard di moduli che usa jlink.
--main-class	Specifica il fully qualified name (nome completo di package) della classe contenente il metodo `main`. Può essere specificato solo se presente l'opzione --main-jar.
--type	Specifica il tipo di applicazione da utilizzare per l'installatore. Gli unici valori validi sono: exe, msi o app-image. Abbiamo già detto che su Windows è possibile utilizzare le opzioni per creare file di installazione con suffisso .exe o .msi. La specifica del valore app-image invece, provoca la creazione dell'*immagine dell'applicazione* (*app image*) ovvero la struttura dei file e delle cartelle, così come verrebbe creata dopo aver eseguito l'installazione. Volendo possiamo testarla e modificarla a nostro piacimento per poi chiedere a jpackage di impacchettarla con l'opzione --app-image (vedi voce successiva). La specifica del valore app-image per l'opzione --type non è compatibile con la specifica di alcune opzioni che caratterizzano la creazione dell'installer (visto che non sarà creato) come --icon, --win-dir-chooser, --win-shorcut e --win-menu.
--app-image	Specifica il percorso (relativo o assoluto) della directory contenente l'immagine dell'applicazione creata molto probabilmente con jpackage tramite la specifica dell'opzione --type app-image (vedi voce precedente).
--app-version	Specifica la versione dell'applicazione. Il nome dell'installer sarà composto dal nome dell'applicazione specificato con --name e la versione dell'applicazione.
@filename	Specifica un file dove il comando jpackage può leggere la lista delle opzioni specificate al suo interno. Questo facilita il riuso dei lunghi comandi di jpackage, ed è possibile anche utilizzare più file specificando più volte questa opzione.

Esistono tante altre opzioni da poter utilizzare, che sono descritte nelle pagine del manuale dei tool del JDK a questo indirizzo: https://docs.oracle.com/en/java/javase/17/jpackage. È anche possibile digitare da riga di comando la seguente istruzione per ottenere una sinossi di tutte le opzioni:

```
jpackage --help
```

21.2.3 Installazione dell'applicazione

Una volta completati i passi precedenti, avremo ottenuto un file di installazione chiamato SwingMix-1.0.exe, dove 1.0 rappresenta la versione dell'applicazione che di default ha impostato jpackage. In pratica il nome del file sarà formato con quanto specificato dalle opzioni:

1. --name: che rappresenta il nome del file;
2. --app-version: che rappresenta la versione dell'applicazione (di default è 1.0)
3. --type: che imposta il tipo di file caratterizzato dal suffisso (di default è .exe sui sistemi operativi Windows).

Con un doppio click possiamo lanciare l'installer, ed apparirà la prima schermata che ci guiderà con un classico wizard che presenterà varie schermate nell'installazione come mostrato nella figura 21.2.

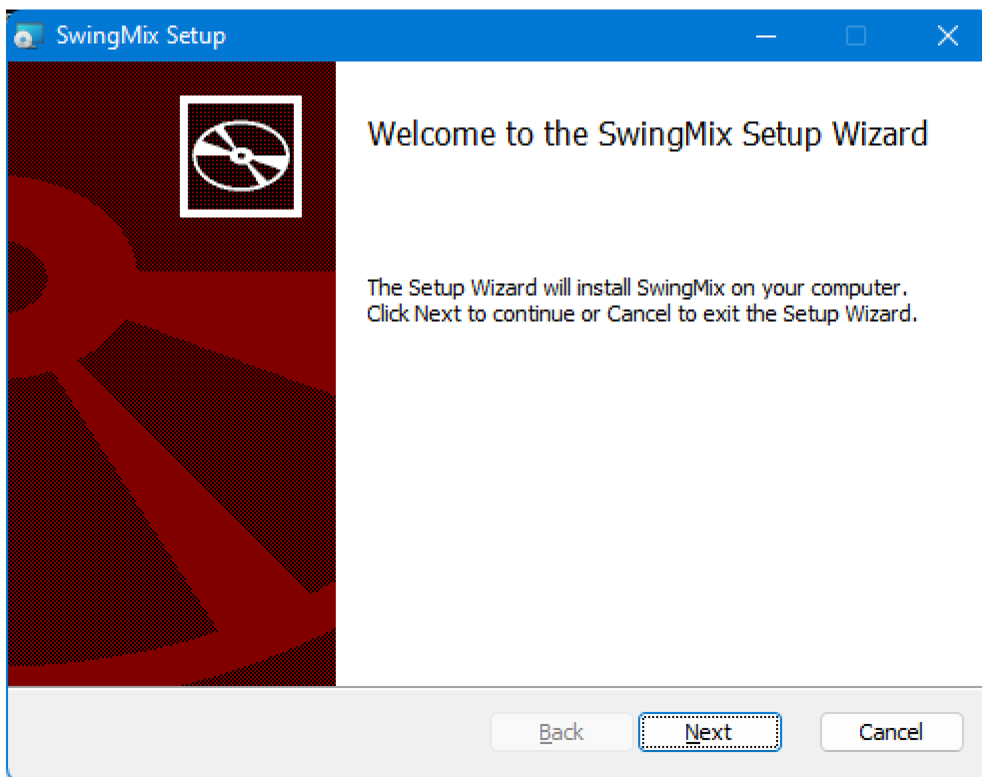


Figura 21.2 – L'applicazione SwingMix.

Il wizard come sempre chiederà di poter installare l'applicazione con i privilegi di amministratore, e ci permetterà di scegliere la cartella di installazione visto che in fase di creazione dell'installer, abbiamo specificato l'opzione --win-dir-chooser.
Una volta installata, potete lanciare l'applicazione tramite la shortcut creata sul desktop, o quella creata nel menu di Windows. Inoltre troverete l'applicazione anche tra le applicazioni installate come mostrato in figura 21.3.

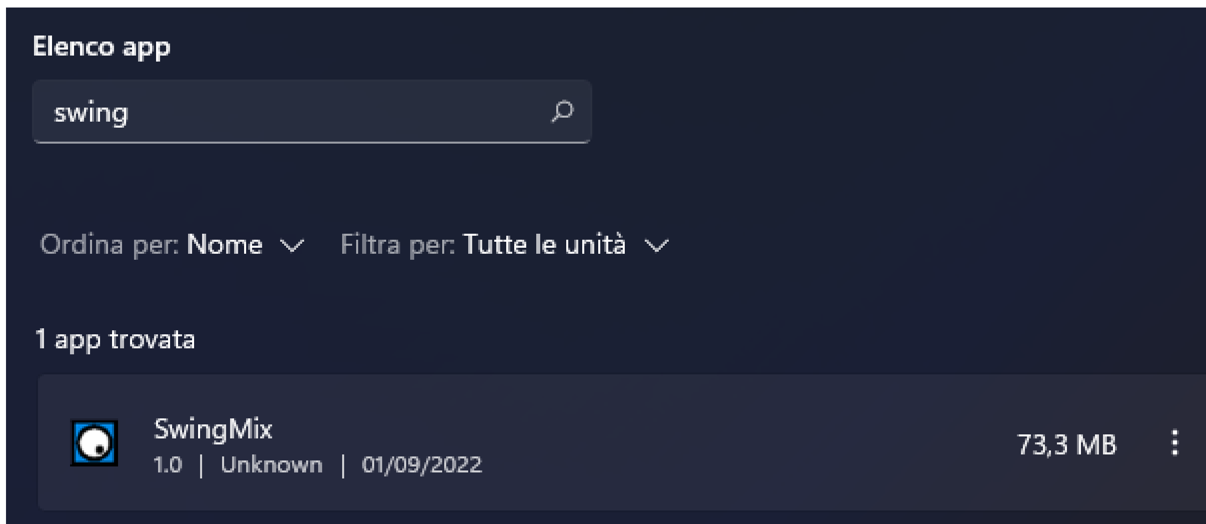


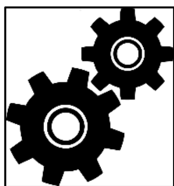
Figura 21.3 – L'applicazione SwingMix nel menu delle applicazioni installate di Windows.

Ovviamente, se volete disinstallare l'applicazione dovete utilizzare sempre la schermata delle applicazioni installate di Windows mostrata nella figura 21.3.

21.3 Pattern matching for switch (feature preview)

Questo paragrafo coincide (tranne che in pochi trascurabili dettagli) con l'articolo relativo al pattern matching per `switch` presente sul blog dell'autore claudiodesio.com.

La caratteristica probabilmente più interessante introdotta in Java 17 è l'introduzione del **pattern matching per il costrutto `switch`**. Si tratta solo di una caratteristica in anteprima (feature preview, cfr. relativo articolo) che probabilmente verrà ufficializzata nella versione 20 (con o senza modifiche all'attuale implementazione). Si tratta della seconda parte dell'articolata caratteristica nota come **pattern matching**, che ha già cambiato il modo in cui utilizziamo l'operatore `instanceof` (cfr. paragrafo 7.3.3.3 o articolo dedicato). Questa seconda parte del pattern matching migliora il costrutto `switch`, in realtà già migliorato nella versione 14 con l'introduzione di una nuova sintassi e la possibilità di utilizzarlo come espressione (cfr. paragrafo 4.4 o articolo dedicato).



Questo paragrafo è abbastanza tecnico e richiede la conoscenza di alcune caratteristiche recentemente aggiunte al linguaggio. Se non ricordate gli argomenti qui sotto e non avete la possibilità di consultare il libro cartaceo durante lo studio di questo paragrafo, consigliamo di leggere prima gli articoli sul pattern matching per `instanceof`, il nuovo `switch`, le feature preview, ed i tipi sealed, che risultano essere propedeutici alla piena comprensione di quanto riportato di seguito.

21.3.1 Pattern matching

Con l'introduzione del **pattern matching** per **instanceof** in Java 16, abbiamo definito un ***pattern*** come composto da:

- un *predicato*: un test che cerca la corrispondenza (`matching`) di un input con un suo operando. Come vedremo tale operando è un tipo (infatti si parla di *type pattern*)
- una o più *variabili di pattern* (dette anche *variabili di vincolo* o *variabili di binding*): queste vengono estratte dall'operando a seconda del risultato del test. Con il pattern matching è stato infatti introdotto un nuovo scope per le variabili, lo *scope di pattern* (o *di binding*), che garantisce la visibilità della variabile solo dove il predicato è verificato.

Un pattern è quindi un modo sintetico di esprimere una soluzione complessa.

Il concetto è lo stesso che troviamo alla base delle *regular expression*. (cfr. paragrafo G.1.5 nel file delle appendici del libro). In questo caso però il pattern è basato sul riconoscimento di un certo tipo mediante l'operatore `instanceof`, e non su una determinata sequenza di caratteri da trovare in una stringa.

21.3.2 Il nuovo `switch`

Il costrutto `switch` è stato rivisto nella versione 12 come feature preview ed ufficializzato nella versione 14. La rivisitazione del costrutto ha introdotto una sintassi meno verbosa e più robusta basata sull'*operatore freccia* `->`, ed inoltre è possibile utilizzare `switch` come espressione (in particolare si parla di *poli-espressione*). Potete approfondire tutti dettagli nel paragrafo 4.4 o nell'articolo dedicato. Il costrutto quindi è diventato più potente, utile ed elegante. È rimasto però il vincolo di poter passare in input ad uno `switch` solo determinati tipi:

1) I tipi primitivi `byte`, `short`, `int` e `char`.
2) I corrispondenti tipi wrapper `Byte`, `Short`, `Integer` e `Character`.
3) Il tipo `String`.
4) Un qualsiasi tipo enumerazione.

In futuro si prevede di rendere il costrutto ancora più utile aggiungendo nuovi tipi alla lista di cui sopra, come i tipi primitivi `float`, `double` e `boolean`. In generale si punta ad arrivare ad uno `switch` che ci permetta di realizzare la *decostruzione degli oggetti*. Attualmente è ancora presto per parlarne, ma intanto Java avanza velocemente passo dopo passo e dalla versione 17 è possibile provare in anteprima una nuova versione del costrutto `switch`, che permette di passare come input un oggetto di qualsiasi tipo. Per entrare in una certa clausola `case` verrà utilizzato il pattern matching per *instanceof*.

21.3.3 Il nuov(issim)o `switch`

Passiamo subito ad un esempio, consideriamo il seguente metodo:

```
public static String getInfo(Object object) {
  if (object instanceof Integer integer) {
    return (integer < 0 ? "Negative" : "Positive") + " integer";
  }
  if (object instanceof String string) {
    return "String of length " + string.length();
  }
  return "Unknown";
}
```

Esso prende in input un parametro polimorfo di tipo `Object` quindi accetta qualsiasi tipo, e sfruttando l'operatore `instanceof` ritorna una particolare stringa descrittiva. Nonostante il pattern matching per `instanceof` ci abbia permesso di risparmiare il tipico passaggio che comprendeva la dichiarazione di una reference ed il relativo cast, il codice è comunque ancora poco leggibile, inelegante e soggetto ad errori. Riscriviamo quindi il metodo precedente utilizzando il pattern matching applicato ad una `switch` expression:

```
public static String getInfo(Object object) {
  return switch (object) {
    case Integer i -> (i < 0 ? "Negative" : "Positive")+ " integer";
    case String s -> "String of length " + s.length();
    default -> "Unknown";
  };
}
```

Il codice è ora più conciso, leggibile, applicabile, funzionale ed elegante, analizziamolo con calma.
Notiamo che, diversamente dal costrutto `switch` che abbiamo sempre utilizzato, nell'esempio precedente la validazione di un certo `case` non sarà basata sull'operatore di uguaglianza il cui secondo operando è una costante, ma sull'operatore `instanceof` il cui secondo operando è un tipo.
In pratica verrà eseguito il codice che segue l'operatore freccia `->` del `case Integer i`, se il parametro `object` è di tipo `Integer`. All'interno di questo codice la variabile di binding `i` di tipo `Integer` punterà allo stesso oggetto a cui punta il reference `object`.
Invece entreremo nel codice che segue l'operatore freccia `->` del `case String s` se il parametro `object` è di tipo `String`. All'interno di questo codice la variabile di binding `s` di tipo `String` punterà allo stesso oggetto a cui punta il reference `object`.
Infine entreremo nella clausola `default`, nel caso il parametro `object` non sia né di tipo `String` né di tipo `Integer`.
Per poter padroneggiare il pattern matching per `switch` però, bisogna anche conoscere una serie di proprietà che verranno presentati nei prossimi paragrafi.

Ricordiamo che nelle versioni 17, 18 e 19, questa caratteristica è ancora in anteprima. Questo significa che per compilare un'applicazione che fa uso del pattern matching per `switch`, bisogna specificare determinati flag come descritto nell'articolo dedicato alle caratteristiche in anteprima.

21.3.4 Exhaustiveness/Completeness

Notare che la clausola `default` è necessaria per non ottenere un errore in compilazione. Infatti il costrutto `switch` con il pattern matching annovera tra le sue proprietà la ***exhaustiveness*** (nota anche come ***completeness***), ovvero la *completezza della copertura di tutte le possibili opzioni*. In questo modo il costrutto è più robusto e meno soggetto ad errori.

Per la retrocompatibilità che da sempre caratterizza Java, non è stato possibile modificare il compilatore in modo tale che pretenda la exhaustiveness anche con il costrutto `switch` originale. Una tale modifica infatti impedirebbe la compilazione a tantissimi progetti preesistenti. È però prevista per le prossime versioni di Java che il compilatore stampi un warning nel caso di implementazioni del "vecchio" `switch` che non coprono tutti i casi possibili. Tutto sommato gli IDE più importanti già avvertono i programmatori in queste situazioni.

Notare che in teoria potremmo anche sostituire la clausola:

```
default -> "Unknown";
```

con l'equivalente:

```
case Object o -> "Unknown";
```

Infatti anche in questo caso avremmo coperto tutti le possibili opzioni. Di conseguenza il compilatore non permetterà di inserire entrambe queste clausole nello stesso costrutto.

21.3.5 Dominance

Se provassimo a spostare la clausola `case Object o` prima delle clausole relative ai tipi `Integer` e `String`:

```
public static String getInfo(Object object) {
  return switch (object) {
    case Object o -> "Unknown";
    case Integer i -> (i < 0 ? "Negative" : "Positive")+ " integer";
    case String s -> "String of length " + s.length();
  };
}
```

otterremo i seguenti errori in compilazione:

```
error: this case label is dominated by a preceding case label
       case Integer i -> (i < 0 ? "Negative" : "Positive")+ " integer";
            ^
error: this case label is dominated by a preceding case label
       case String s -> "String of length " + s.length();
            ^
```

Infatti, un'altra proprietà del pattern matching per `switch` nota come **dominance**, fa sì che il compilatore consideri irraggiungibili i `case Integer i` e `case String s`, perché "dominati" dal `case Object o`. In pratica quest'ultima clausola comprende le condizioni delle successive due che quindi non verrebbero mai raggiunte.

> **Questo comportamento è molto simile a quello che già conosciamo sussistere quando specifichiamo più clausole `catch` per gestire le nostre eccezioni. Una clausola `catch` più generica, potrebbe dominare le clausole `catch` successive, causando un errore del compilatore. Anche in quel caso è necessario posizionare la clausola dominante dopo le altre.**

21.3.6 Dominance e clausola `default`

A differenza delle clausole `case` ordinarie, la clausola `default` invece non deve per forza essere inserita come ultima clausola. Infatti è perfettamente legale inserire la clausola `default` come prima istruzione di un costrutto `switch` senza alterarne il funzionamento. Il seguente codice viene compilato senza errori:

```
public static String getInfo(Object object) {
  return switch (object) {
    default -> "Unknown";
    case Integer i -> (i < 0 ? "Negative" : "Positive")+ " integer";
    case String s -> "String of length " + s.length();
  };
}
```

Ciò in realtà vale anche per il costrutto `switch` classico, ed è anche la ragione per cui si consiglia di inserire un'istruzione `break` anche nella clausola `default`. Infatti aggiungendo inavvertitamente una nuova clausola dopo il `default` e senza il `break` potremmo provocare un *fall-through* indesiderato.

21.3.7 Guarded pattern

Possiamo anche specificare dei pattern composti con delle espressioni booleane tramite l'operatore `&&` ed in questo caso si parla di **guarded pattern** (e l'espressione booleana viene detta **guard**). Per esempio possiamo riscrivere il codice precedente in modo più leggibile e intuitivo:

```
public static String getInfo(Object object) {
  return switch (object) {
    case Integer i && i < 0 -> "Negative integer"; // guarded pattern
    case Integer i -> "Positive integer";
    case String s -> "String of length " + s.length();
    default -> "Unknown";
  };
}
```

Nella versione 19 (terza preview) in base ai feedback degli sviluppatori, l'operatore `&&` è stato sostituito dalla clausola **`when`** (nuova parola chiave contestuale). Quindi il precedente codice dalla versione 19 in poi deve essere riscritto in questo modo:

```
public static String getInfo(Object object) {
  return switch (object) {
    case Integer i when i < 0 -> "Negative integer"; // guarded pattern
    case Integer i -> "Positive integer";
    case String s -> "String of length " + s.length();
    default -> "Unknown";
  };
}
```

Notare che se invertissimo le clausole riguardanti gli interi nel seguente modo:

```
   case Integer i -> "Positive integer"; //questo pattern "domina" il successivo
   case Integer i when i < 0 -> "Negative integer";
```

otterremo un errore di dominance:

```
error: this case label is dominated by a preceding case label
            case Integer i when i < 0 -> "Negative integer";
                 ^
```

21.3.8 Fall-through

Abbiamo già visto nel paragrafo 4.4.2.3 e nell'[articolo dedicato al nuovo `switch`], come la nuova sintassi basata sull'operatore freccia `->` ci permetta di utilizzare una sola clausola `case` per gestire `case` diversi allo stesso modo. In pratica simuliamo l'utilizzo di un operatore OR `||` evitando di utilizzare una tecnica tanto controversa come quella del *fall-through*. Per esempio possiamo scrivere:

```
Month month = getMonth();
String season = switch(month) {
    case DECEMBER, JANUARY, FEBRUARY -> "winter";
    case MARCH, APRIL, MAY -> "spring";
    case JUNE, JULY, AUGUST -> "summer";
    case SEPTEMBER, OCTOBER, NOVEMBER -> "autumn";
};
```

La sintassi è notevolmente più sintetica, elegante e robusta.
Quando utilizziamo il pattern matching però la situazione cambia. Nelle clausole del costrutto `switch` non è possibile utilizzare pattern multipli per gestire tipi diversi allo stesso modo. Per esempio, il seguente metodo:

```
public static String getInfo(Object object) {
  return switch (object) {
    case Integer i, Float f -> "This is a number";
    case String s -> "String of length " + s.length();
    default -> "Unknown";
  };
}
```

produrrebbe un errore in compilazione:

```
error: illegal fall-through to a pattern
            case Integer i, Float f -> "This is a number";
                            ^
```

Infatti, per il concetto di variabili di pattern, il codice dopo l'operatore freccia potrebbe utilizzare sia la variabile `i` che la variabile `f`, ma una di esse sicuramente non sarà inizializzata. È stato scelto quindi di non rendere questo codice compilabile per avere un costrutto più robusto.
Notare che il messaggio di errore ci evidenzia che questo codice non è valido perché definisce un *fall-through illegale*. Infatti il codice precedente è equivalente al seguente (anch'esso non compilabile) che non usando la sintassi basata sull'operatore freccia `->`, fa uso del fall-through:

```
public static String getInfo(Object object) {
  return switch (object) {
    case Integer i: // manca il break: fall-through
    case Float f:
      yield "This is a number";
    break;
    case String s:
      yield "String of length " + s.length();
    break;
    default:
      yield "Unknown";
    break;
  };
}
```

21.3.9 Controllo di nullità

Nel momento in cui è stata introdotta la possibilità di passare un qualsiasi tipo ad un costrutto `switch`, dovremmo prima controllare che il reference in input non sia `null`. Invece di far precedere il costrutto `switch` dal classico controllo di nullità:

```
  if (object == null) {
    return "Null!";
  }
```

possiamo utilizzare una nuova elegante clausola da aggiungere nello `switch` per gestire il caso in cui il parametro `object` sia `null`.

```
public static String getInfo(Object object) {
  return switch (object) {
    case null -> "Null!"; // controllo di nullità
    case Integer i -> (i < 0 ? "Negative" : "Positive")+ " integer";
    case String s -> "String of length " + s.length();
    default -> "Unknown";
  };
}
```

La clausola `case null` ci permette di evitare il solito noioso controllo a cui siamo abituati. Questa clausola è opzionale, ma siccome è comunque sempre possibile passare un reference `null` ad uno `switch` che fa uso del pattern matching, nel caso non ne inserissimo una esplicitamente, il compilatore ne inserirà una per noi il cui codice lancerà una `NullPointerException`.

```
public static String getInfo(Object object) {
  return switch (object) {
    case null -> throw new NullPointerException(); // inserita dal compilatore
    case Integer i -> (i < 0 ? "Negative" : "Positive")+ " integer";
    case String s -> "String of length " + s.length();
    default -> "Unknown";
  };
}
```

21.3.10 Dominance e `case null`

Notare che, come la clausola `default` non deve per forza essere l'ultima delle clausole di uno `switch`, non è necessario che la clausola `case null` si trovi in cima al costrutto. Di conseguenza anche per tale clausola la regola della dominance non è applicabile. È del tutto legale spostare il `case null` come ultima riga dello `switch`, come è legale avere come prima clausola il `case default` senza influire sulla funzionalità del costrutto:

```
public static String getInfo(Object object) {
  return switch (object) {
    default -> "Unknown";
    case Integer i -> (i < 0 ? "Negative" : "Positive")+ " integer";
    case String s -> "String of length " + s.length();
    case null -> "Null!";
  };
}
```

Tuttavia questa pratica non è consigliata: meglio mantenere la leggibilità del costrutto seguendo il buon senso e lasciare le varie clausole nelle posizioni in cui ci aspettiamo di trovarle.

21.3.11 Fall-through con `case null`

Il `case null` è l'unico `case` che può essere utilizzato in un clausola che raggruppa più pattern. Per esempio la seguente clausola è legale:

```
case null, Integer i -> "This is a number or null";
```

Più probabilmente accoppieremo il `case null` con la clausola `default`:

```
case null, default -> "Unknown or null";
```

In questo caso, c'è il vincolo che il `case null` sia specificato prima della clausola `default`. Il seguente codice infatti produrrà un errore in compilazione:

```
default, case null -> "Unknown or null";
```

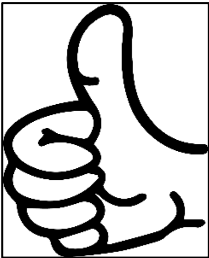
21.3.12 Exhaustiveness con tipi sigillati

Il concetto di exhaustiveness/completeness di cui abbiamo già accennato precedentemente, deve essere rivisto nel caso di gerarchie di tipi sigillati (classi ed interfacce sealed, cfr. paragrafi 6.2 e 6.4 o articolo dedicato). Come al solito cerchiamo di semplificare il discorso introducendo un esempio. Consideriamo una gerarchia sigillata, dove la classe `DiscoOttico` è dichiarata `sealed`, ed ha come uniche due sottoclassi le due classi `CD` e `DVD`: dichiarate `final` e che quindi non possono essere estese:

```
public sealed abstract class DiscoOttico permits CD, DVD {
  // codice omesso
}

public final class CD extends DiscoOttico {
  // codice omesso
}

public final class DVD extends DiscoOttico {
  // codice omesso
}
```

Il seguente codice compila senza errori nonostante non sia stata specificata la clausola `default`:

```
public class LettoreOttico {
  public void inserisci(DiscoOttico discoOttico) {
    switch(discoOttico) {
      case CD cd -> suonaDisco(cd);
      case DVD dvd -> caricaMenu(dvd);
    }
  }
  // resto del codice omesso
}
```



Notare che non è necessario utilizzare la clausola `default` in casi come questi. Infatti l'utilizzo della classe astratta `sealed DiscoOttico` ci garantisce che come input questo `switch` può accettare solamente oggetti di tipo `CD` e `DVD`, e che quindi non è necessario aggiungere una clausola `default` perché tutti i casi sono già stati coperti. In caso di utilizzo di gerarchie sigillate, è quindi sconsigliato utilizzare la clausola `default`. Infatti la sua assenza permetterebbe di segnalare eventuali modifiche della gerarchia in fase di compilazione.

Per esempio, proviamo ora a modificare la classe `DiscoOttico` aggiungendo nella clausola `permits` la seguente classe `BluRay`:

```
public sealed abstract class DiscoOttico permits CD, DVD, BluRay {
  // codice omesso
}

public final BluRay implements DiscoOttico {
  // codice omesso
}
```

Se ora proviamo a compilare la classe `LettoreOttico` otterremo un errore:

```
.\LettoreOttico.java:3: error: the switch statement does not cover all
possible input values
        switch(discoOttico) {
        ^
```

che evidenzia che il costrutto non rispetta la regola della exhaustiveness.
Se invece avessimo inserito anche la clausola `default` il compilatore non avrebbe segnalato nessun errore.

Notare che se la classe `DiscoOttico` non fosse stata dichiarata `abstract`, avremmo potuto passare in input allo `switch` oggetti di tipo `DiscoOttico`. Di conseguenza, per la regola della exhaustiveness avremmo dovuto aggiungere anche una clausola per gli oggetti di tipo `DiscoOttico`. Per la regola della dominance inoltre tale clausola avrebbe dovuto essere posizionata come ultima. L'alternativa sarebbe stata quella di aggiungere una clausola `default`.

21.3.13 Gestione della compilazione migliorata

Java 17 implementa un sistema di compilazione migliorato per prevenire eventuali problemi di compilazione parziale del codice. Se nell'esempio precedente avessimo compilato solo la classe `DiscoOttico` e la classe `BluRay` senza ricompilare la classe `LettoreOttico`, allora il compilatore avrebbe aggiunto implicitamente una clausola `default` al costrutto `switch` di `LettoreOttico`, il cui codice lancerà una `IncompatibleClassChangeError`. In pratica il codice sarebbe stato modificato nel seguente modo:

```
public class LettoreOttico {
  public void inserisci(DiscoOttico discoOttico) {
    switch(discoOttico) {
      case CD cd -> suonaDisco(cd);
      case DVD dvd -> caricaMenu(dvd);
      default -> throw new IncompatibleClassChangeError(); // codice implicito
    }
  }
  // resto del codice omesso
}
```

Quindi in casi come questo il compilatore renderà automaticamente il nostro codice più robusto.

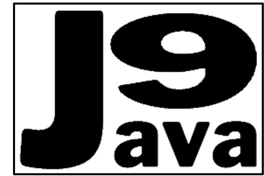
21.4 Altre novità

In questo paragrafo e nei suoi sottoparagrafi, descriviamo brevemente altre novità secondarie introdotte nelle versioni 16 e 17. Si tratta più che altro di novità riguardanti il comportamento della piattaforma Java, e non di caratteristiche del linguaggio. Abbiamo deciso inoltre di non occuparci delle caratteristiche in incubazione, visto che potrebbero subire grosse modifiche o anche essere eliminate completamente in futuro. Ce ne occuperemo, se possibile, nelle prossime versioni.

21.4.1 Forte incapsulamento di default sulla JDK Internal API

Molte librerie, framework e strumenti di terze parti accedono tramite reflection (cfr. paragrafo 11.1) alla cosiddetta **JDK Internal API** (detta anche semplicemente **Internal API**), che contiene package destinati all'utilizzo del JDK detti *package interni* (in inglese *internal packages*). Appartengono alla JDK Internal API alcune classi, metodi e variabili non pubbliche dei package che iniziano con `java.*`, la maggior parte delle classi, metodi e variabili dei package che iniziano con `org.sun.*`, `jdk.*` ed `org.*`, e tutte le classi di `sun.*`. C'è un potenziale problema di sicurezza riguardo l'utilizzo di questi package da parte dei programmi ordinari, e già dalla versione 9 di Java, con l'introduzione del sistema modulare (cfr. capitolo 16), l'utilizzo di tali package è stato limitato. Infatti, la modularizzazione ha portato con sé il *forte incapsulamento* (in inglese *strong encapsulation*, cfr. paragrafo 16.2.3.3). Ricordiamo che, se il classico incapsulamento impedisce l'utilizzo di membri non accessibili in fase di compilazione, il forte incapsulamento non permette neanche di accedere tramite reflection al codice compilato in fase di esecuzione. Possiamo dire che il forte incapsulamento evita che l'incapsulamento venga violato al runtime.

21.4.1.1 Relaxed Strong Encaspulation



Nella versione 9 sono stati fortemente incapsulati, tutte i nuovi elementi della Internal API, ma non quelli che esistevano prima nella versione 8. Questo per favorire una migrazione graduale e sostenibile per i programmi in essere. Negli ultimi anni Oracle sta rilasciando librerie alternative e sicure per permettere agli sviluppatori di abbandonare le librerie interne gradualmente. Per esempio la libreria `java.util.Base64` (cfr. paragrafo 15.4.3.8) ha sostituito la libreria interna basata sulle classi `sun.misc.Base64Encoder` e `sun.misc.Base64Decoder`.

La lista di tutte le alternative allo stato attuale la trovate a questo link.

Quindi dalla versione 9 alla 15, è stato possibile utilizzare tramite reflection le librerie interne che esistevano prima di Java 9 liberamente. Questa caratteristica è nota come *relaxed strong encapsulation*, che potremmo tradurre in italiano *forte incapsulamento leggero*.

Nella versione 16 il forte incapsulamento è stato esteso di default a tutti gli elementi interni del JDK ad eccezione delle API interne critiche (al cui utilizzo non c'è alternativa) come `sun.misc.Unsafe`. Anche nella versione 16 è possibile usare il *forte incapsulamento leggero* per accedere agli elementi interni che esistevano prima della versione 9, grazie all'opzione da specificare durante il lancio dell'applicazione `--illegal-access`.

21.4.1.2 L'opzione `--illegal-access`

Per gestire il forte incapsulamento leggero esiste l'opzione `--illegal-access` da specificare in fase di lancio dell'applicazione.

Notare che il nome di questa opzione, parlando di *accesso illegale*, è stato scelto provocatoriamente allo scopo di scoraggiarne l'uso.

A tale opzione si può assegnare uno tra i valori `permit`, `warn`, `debug` e `deny`:

- `--illegal-access=permit` : specificando questa opzione ogni elemento della Internal API che esisteva prima della versione 9 non è fortemente incapsulato. Questo significa che è possibile utilizzare la reflection per accedere a questi elementi. Durante il runtime solo al primo accesso *illegale* ad uno di questi elementi, verrà visualizzato un warning.

Dalla versione 9 alla versione 15, questa opzione era implicita quando lanciavamo un'applicazione Java.

- `--illegal-access=warn` : questa opzione funziona come `--illegal-access=permit` tranne per il fatto che ad ogni accesso tramite reflection *illegale* verrà visualizzato un warning.

- `--illegal-access=debug` : questa opzione funziona come `--illegal-access=warn` tranne per il fatto che ad ogni accesso tramite reflection *illegale* verrà visualizzato un warning accompagnato dallo stack trace che ha provocato l'accesso.

- `--illegal-access=deny` : disabilita tutte le operazioni di *accesso illegale* ad eccezione di quelle abilitate da altre opzioni della riga di comando, come ad esempio `--add-opens` (cfr. 16.2.3.3).

> **Nella versione 16 questa opzione era implicita quando lanciavamo un'applicazione Java. Come già asserito nel paragrafo precedente, è sempre possibile accedere alle API interne critiche come `sun.misc.Unsafe` tramite reflection.**

21.4.1.3 Obsolescenza dell'opzione `--illegal-access`

Nella versione 17 invece, Oracle ha reputato i tempi maturi per avanzare di un altro passo verso il miglioramento della sicurezza e della manutenibilità di Java. L'opzione `--illegal-access` è ora stata dichiarata *obsoleta* ed il suo uso non produrrà altro che il messaggio di warning seguente:

```
Java HotSpot(TM) 64-Bit Server VM warning:
  Ignoring option --illegal-access=deny; support was removed in 17.0
```

Nelle prossime versioni l'opzione sarà completamente rimossa.
Ovviamente le API interne critiche come `sun.misc.Unsafe`, possono ancora essere usate tramite reflection, e continueranno ad esserlo sino a quando Oracle non avrà fornito delle alternative equivalenti sicure. In particolare:

- I package `sun.misc` e `sun.reflect` verranno comunque esportati dal modulo `jdk.unsupported` e quindi sarà possibile accedervi via Reflection API. Nessun altro package sarà *aperto* (cfr. paragrafo 16.2.3.3) dal modulo `jdk.unsupported`.
- La maggior parte dei package che iniziano con `com.sun.*` vengono considerati di solo uso interno, ma alcuni sono supportati per l'uso esterno, nel senso che si tratta di classi, metodi e campi pubblici e quindi possono essere usate normalmente dagli sviluppatori. Questi package non saranno accessibili al runtime tramite reflection, infatti sono *esportati* ma non dichiarati *aperti* dai rispettivi moduli. Alcuni di questi sono:
  - l'API *Compiler Tree* nel modulo `jdk.compiler`;
  - l'API del *HTTP Server* nel modulo `jdk.httpserver`;
  - l'API *SCTP* nel modulo `jdk.sctp`;
  - alcune estensioni specifiche del JDK all'API *NIO* (cfr. paragrafo 15.5), contenute all'interno del package `com.sun.nio.file` del modulo `jdk.unsupported`.

Sarà comunque possibile utilizzare l'opzione della riga di comando `--add-opens` o l'attributo manifest del file JAR `Add-Opens` per aprire specifici package.

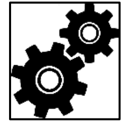
> **La lista completa delle API interne esistenti prima di Java 9 non più disponibili per l'utilizzo in Java 17 è disponibile a questo indirizzo.**

21.4.2 Warnings per le Value-Based Classes

Sono stati introdotti dei warning sia a livello di compilazione che a livello di runtime, nel caso di utilizzo in maniera sincronizzata di istanze delle cosiddette **value-based classes** (*classi basate su*

valori). La definizione di value-based class è stata modificata più volte nelle ultime versioni, e prima di scendere nei dettagli conviene prima formalizzarne la definizione allo stato attuale.

21.4.2.1 Cosa sono le value-based class



Viene definita **value based class** (*classe basata sul valore*), una classe che ha le seguenti caratteristiche:

- dichiara solo variabili di istanza `final` (sebbene possa contenere reference ad oggetti mutabili);
- definisce i metodi `equals`, `hashCode`, e `toString` in modo tale che restituiscano risultati basati solo sulle variabili di istanza (ed eventuali campi degli oggetti che referenziano), ma non sulla propria ***identità*** (in inglese ***identity***). In pratica questi metodi non si devono basare su controlli che confrontino il reference `this` con altri reference dello stesso tipo.

> **Possiamo dire che due oggetti hanno la stessa identità se confrontati con l'operatore == restituiscono `true` (anche se in realtà è possibile verificare che due oggetti hanno la stessa identità utilizzando altre tecniche come la serializzazione).**

- quando due istanze della classe sono uguali in base al metodo `equals`, esse si possono considerare *intercambiabili*. Questo significa che i metodi di due istanze della classe che sono uguali per il metodo `equals` si comporteranno allo stesso modo;
- la classe non esegue alcuna sincronizzazione utilizzando il monitor di un'istanza (cfr. paragrafo 12.4.2);
- la classe non dichiara costruttori accessibili, o se li dichiara, questi sono deprecati;
- la classe non fornisce nessun metodo di creazione di istanze (come un costruttore pubblico) che garantisce un'identità univoca su ogni chiamata al metodo. In pratica gli oggetti verranno creati tramite metodi factory. Se due istanze create da due diverse chiamate ad un metodo factory sono uguali secondo il metodo `equals`, potrebbero anche essere considerati uguali secondo l'operatore `==`;
- la classe è dichiarata `final` ed estende `Object` o una gerarchia di classi astratte che non dichiarano campi di istanza o inizializzatori di istanza (cfr. paragrafo 5.4.3.1) e i cui costruttori sono vuoti.

La definizione sembra quindi piuttosto complicata, ma potremmo semplificarla sottolineando le caratteristiche più importanti descritte sopra.

21.4.2.2 Value-based class in pratica

In pratica si tratta di classi immutabili che <u>non definiscono costruttori pubblici</u>, quindi verranno istanziate mediante <u>metodi factory</u>. Le istanze di una value-based class che risultano uguali secondo il metodo `equals` dovrebbero considerarsi <u>equivalenti ed intercambiabili</u>. Infine sussiste un vincolo molto particolare: non bisogna <u>sincronizzare il codice sul monitor di un istanza di una value-based class</u>. Infatti il programmatore non può garantire la proprietà esclusiva del monitor associato ad un oggetto, vista la proprietà delle istanze di essere intercambiali.

Vedremo nel prossimo paragrafo, perché è così importante non sincronizzare il codice sul monitor di una istanza di una value-based class.

Per esempio, tenendo presente che la class `Integer` è una value-based class, il seguente codice:

```
public class TestValueBasedClass {
    private Integer integer = 2;

    public void synchIncrement() {
        synchronized(integer) {
            System.out.println(++integer);
        }
    }

    public static void main(String args[]) {
        var tvbc = new TestValueBasedClass();
        tvbc.synchIncrement();
    }
}
```

che con Java 15 compilava correttamente senza produrre alcun output, a partire da Java 16 restituisce il seguente warning:

```
TestValueBasedClass.java:6: warning: [synchronization] attempt to
synchronize on an instance of a value-based class
        synchronized(integer) {
        ^
1 warning
```

Inoltre è possibile anche visualizzare warning in fase di esecuzione dell'applicazione, o persino fare in modo che l'applicazione si blocchi in fase di esecuzione al verificarsi di una sincronizzazione su una istanza di una value-based class. Infatti possiamo lanciare l'applicazione usando le seguente coppia di opzioni da riga di comando:

- `-XX:+UnlockDiagnosticVMOptions`: che abiliterà le opzioni diagnostiche della JVM
- `-XX:DiagnoseSyncOnValueBasedClasses`: che abiliterà l'opzione di diagnostica della JVM che riguarda la sincronizzazione sulle value-based class. A questa dobbiamo assegnare il valore `1` o `2`. In caso di utilizzo della sincronizzazione su un'istanza di una value-based class, se vogliamo che l'applicazione si interrompa con un errore al runtime dobbiamo specificare il valore `1`. Se vogliamo che venga stampato un warning al runtime dalla JVM dobbiamo usare specificare il valore `2`.

Quindi, lanciando l'applicazione specificando il valore `2` con il seguente comando:

```
java -XX:+UnlockDiagnosticVMOptions -XX:DiagnoseSyncOnValueBasedClasses=2
TestValueBasedClass
```

otterremo il seguente log che ci avverte del problema.

```
[0.040s][info][valuebasedclasses] Synchronizing on object
0x00000006220124e0 of klass java.lang.Integer
[0.040s][info][valuebasedclasses]       at
TestValueBasedClass.synchIncrement(TestValueBasedClass.java:5)
[0.041s][info][valuebasedclasses]       - locked <0x00000006220124e0> (a
java.lang.Integer)
[0.041s][info][valuebasedclasses]       at
TestValueBasedClass.main(TestValueBasedClass.java:12)
```

Invece, lanciando l'applicazione specificando il valore `1` con il seguente comando:

```
java -XX:+UnlockDiagnosticVMOptions -XX:DiagnoseSyncOnValueBasedClasses=1
TestValueBasedClass
```

otterremo un errore al runtime che produce il seguente output:

```
#
# A fatal error has been detected by the Java Runtime Environment:
#
#  Internal Error (synchronizer.cpp:397), pid=25604, tid=1832
#  fatal error: Synchronizing on object 0x00000006220124e0 of klass
java.lang.Integer at
TestValueBasedClass.synchIncrement(TestValueBasedClass.java:5)
#
# JRE version: Java(TM) SE Runtime Environment (17.0.1+12) (build
17.0.1+12-LTS-39)
# Java VM: Java HotSpot(TM) 64-Bit Server VM (17.0.1+12-LTS-39, mixed
mode, sharing, tiered, compressed oops, compressed class ptrs, g1 gc,
windows-amd64)
# No core dump will be written. Minidumps are not enabled by default on
client versions of Windows
#
# An error report file with more information is saved as:
# codice\capitolo_21\esempi\paragrafo_21.4\hs_err_pid25604.log
#
# If you would like to submit a bug report, please visit:
#   https://bugreport.java.com/bugreport/crash.jsp
#
```

In futuro questo codice potrebbe anche produrre di default un errore in compilazione o un errore al runtime. È importante quindi iniziare ad utilizzare le value-based class seguendo le indicazioni di cui sopra.

21.4.2.3 Perché è importante?

È previsto per le prossime versioni di Java una importante modifica alla base stessa del linguaggio: l'introduzione di un nuovo tipologia di dato che si affiancherà ai tipi primitivi e ai tipi reference: le **classi primitive** (**primitive classes**). Tali classi dichiarano istanze prive di identità e capaci di essere considerate in memoria come fossero tipi di dati primitivi e codificate utilizzando esclusivamente i valori dei campi delle istanze. Il concetto è simile a quello delle classi wrapper che usufruiscono della caratteristica dell'autoboxing-unboxing: potremo creare classi che si possono usare come i tipi primitivi, migliorando le performance e semplificando il nostro codice.
In pratica le value-based class sono candidate a diventare nelle prossime versioni classi primitive. Ciò implica che le loro istanze siano immutabili e che le identità di tali istanze non siano importanti per il comportamento della classe. Inoltre queste classi non definiscono costruttori pubblici che invece garantirebbero un'identità univoca con ogni chiamata.

21.4.2.4 L'annotazione `jdk.internal.ValueBased`

Da Java 16 una serie di classi è stata annotata con l'annotazione `ValueBased` del package `jdk.internal`. Possiamo dire che una classe è value-based se è annotata con `@ValueBased`. Segue un estratto della dichiarazione della classe value-based `Optional` (cfr. paragrafo 14.3):

```
package java.util;
//imports...
@jdk.internal.ValueBased
public final class Optional<T> {
```

```
// . . .
}
```

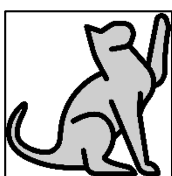
21.4.2.5 Quali sono le value-based class

Nella versione 17 sono state annotate con l'annotazione `ValueBased` 35 classi come è possibile vedere nella figura 21.4. Essa mostra un filtro fatto sui sorgenti delle classi della libreria Java che contengono tale annotazione. Tutte appartengono al modulo `java.base`, ed essenzialmente appartengono ai package `java.lang`, `java.time`, `java.time.chrono` e `java.util`. Tra queste spiccano le classi wrapper (`Integer`, `Double`, `Character`, etc., cfr. paragrafo F.5 dell'appendice F), le classi della Optional API (`Optional`, `OptionalDouble`, `OptionalInt` ed `OptionalLong`, cfr. paragrafo 14.3)), la maggior parte delle classi importanti della Java Date-Time API come `LocalDate`, `LocalDateTime`, `ZonedDateTime`, `Period`, `Duration`, `Instant`, etc. (cfr. appendice I) e le collezioni immutabili prodotte dai metodi di convenienza per le collection (cfr. H.7.1.3 dell'appendice H).

java.base\java\lang\Boolean.java
java.base\java\lang\Byte.java
java.base\java\lang\Character.java
java.base\java\lang\Double.java
java.base\java\lang\Float.java
java.base\java\lang\Integer.java
java.base\java\lang\Long.java
java.base\java\lang\ProcessHandle.java
java.base\java\lang\ProcessHandleImpl.java
java.base\java\lang\Runtime.java
java.base\java\lang\Short.java
java.base\java\time\chrono\HijrahDate.java
java.base\java\time\chrono\JapaneseDate.java
java.base\java\time\chrono\MinguoDate.java
java.base\java\time\chrono\ThaiBuddhistDate.java
java.base\java\time\Duration.java
java.base\java\time\Instant.java
java.base\java\time\LocalDate.java
java.base\java\time\LocalDateTime.java
java.base\java\time\LocalTime.java
java.base\java\time\MonthDay.java
java.base\java\time\OffsetDateTime.java
java.base\java\time\OffsetTime.java
java.base\java\time\Period.java
java.base\java\time\Year.java
java.base\java\time\YearMonth.java
java.base\java\time\ZonedDateTime.java
java.base\java\time\ZoneId.java
java.base\java\time\ZoneOffset.java
java.base\java\util\ImmutableCollections.java
java.base\java\util\KeyValueHolder.java
java.base\java\util\Optional.java
java.base\java\util\OptionalDouble.java
java.base\java\util\OptionalInt.java
java.base\java\util\OptionalLong.java

Figura 21.4 – Java 17: value-based classes.

Le uniche classi value-based dell'elenco precedente che hanno costruttori pubblici (che però sono deprecati *for removal*) sono le classi wrapper.



Notare anche che in realtà la classe `Runtime` elencata nella figura 21.4 non è una value-based class. Infatti la figura mostra il risultato di un filtro fatto sui sorgenti delle classi della libreria Java che contengono l'annotazione `ValueBased`, ma in questo caso questa annota la classe interna statica `Runtime.Version`, che è la vera classe value-based.

21.4.3 Ripristino della semantica relativa a `strictfp`

Abbiamo introdotto il modificatore `strictfp` nel paragrafo 3.3.2.1. Essenzialmente abbiamo asserito che questo modificatore permette di troncare i bit che fanno variare la rappresentazione approssimata dei numeri floating point (sia di tipo `float` che `double`) su piattaforme diverse, garantendo così l'uniformità dei risultati su tutte le piattaforme (ma non garantendone la correttezza). Inoltre abbiamo sottolineato che il modificatore può essere applicato a classi, metodi e a variabili di tipo `float` o `double`.
In realtà, prima della versione 1.2 di Java, tutti i calcoli che coinvolgevano numeri decimali avevano già questa caratteristica, ma essa causava un problema di surriscaldamento su processori che supportavano l'architettura x87. Quindi con Java 1.2, venne introdotto `strictfp` che evitava questo tipo di problema. Con i progressi hardware compiuti dalle più importanti case produttrici di processori negli ultimi anni, questo problema di surriscaldamento è stato risolto grazie al supporto dell'estensione SSE2 (Streaming SIMD Extension 2).
Java 17 ha quindi ripristinato il modo di calcolare le operazioni che coinvolgono i numeri decimali come avvenivano prima di Java 1.2 di default, il che rende l'utilizzo del modificatore `strictfp` di fatto inutile.

21.4.5 Deprecazione di SecurityManager e Applet API

Java 17 ha deprecato per la rimozione (*for removal* cfr. paragrafo 11.3.3) due storiche librerie come *Security Manager* e *Applet API*. Questo significa che nelle prossime versioni di Java queste API saranno eliminate del tutto, e che quindi gli sviluppatori devono iniziare a rimuovere dal loro codice riferimenti a queste librerie. Segue un estratto del sorgente della classe `SecurityManager`:

```
package java.lang;
//imports...
/**
 * . . .
 * @since   1.0
 * @deprecated The Security Manager is deprecated and subject to removal in a
 *       future release. There is no replacement for the Security Manager.
 *       See <a href="https://openjdk.java.net/jeps/411">JEP 411</a> for
 *       discussion and alternatives.
 */
@Deprecated(since="17", forRemoval=true)
public class SecurityManager {
  //...
}
```

Di seguito invece trovate un estratto della codice sorgente della classe `Applet`:

```
package java.applet;
//imports...
/**
 * . . .
 * @since 1.0
 * @deprecated The Applet API is deprecated, no replacement.
 */
@Deprecated(since = "9", forRemoval = true)
@SuppressWarnings("removal")
public class Applet extends Panel {
  //...
}
```

Riepilogo

La versione 16 di Java ha ufficializzato i **tipi record** e il **pattern matching per `instanceof`**, che ora non sono più caratteristiche in anteprima. L'ingresso ufficiale dei tipi record ha portato con sé la possibilità di dichiarare classi interne che dichiarano membri statici, mentre il pattern matching per `instanceof` ha raffinato il concetto di **variabili di pattern** (ex variabili di binding) che ora sono più simili alle variabili locali e ha migliorato l'efficienza nei controlli del compilatore.
L'introduzione del tool **jpackage** dovrebbe cambiare il modo in cui le applicazioni Java vengono distribuite, creando file di installazione come avviene per altri linguaggi di programmazione.
Abbiamo definito le **value-based class** come le future candidate di un nuovo tipologia di dato in Java: le *classi primitive*. Abbiamo sottolineato come diversi tipi di warning vengono visualizzati nel caso di utilizzo scorretto di tali classi.
Inoltre in Java 16 **il forte incapsulamento per la JDK Internal API è stato impostato di default**, mentre la seconda preview dei tipi sealed ha portato con sé la definizione di **keyword contestuale**, oltre a miglioramenti dei controlli del compilatore sulle gerarchie sealed e sulle classi locali.

Nella versione 17 **i tipi sealed sono stati ufficializzati** senza ulteriori modifiche, mentre **il forte incapsulamento per la JDK Internal API è stato rafforzato** con la obsolescenza dell'opzione `--illegal-access` che rende l'utilizzo degli *internal package* più complicato.
Il **pattern matching per il costrutto `switch`** è stato introdotto come caratteristica in anteprima (feature preview). Questa nuova manifestazione del pattern matching ha aggiunto una nuova sintassi per il costrutto `switch`, sicuramente più elegante e funzionale. Tuttavia la complessità del linguaggio si incrementa di pari passo con l'aumento delle nuove caratteristiche. Infatti, questa feature preview è costruita sul nuovo `switch`, il pattern matching per `instanceof` ed i tipi sealed
La conoscenza di questi argomenti è propedeutica alla piena comprensione del pattern matching per *switch*. Inoltre insieme ad esso sono stati introdotti diverse nuove definizioni come la **dominance**, i **guarded pattern**, i **controlli di nullità**, la **completeness**, etc.
Infine, in Java 17 il modificatore **strictfp è stato reso obsoleto** e sono state **deprecate *for removal* le storiche Security Manager e Applet API**.

Esercizi, approfondimenti ed altro materiale didattico sono disponibili all'indirizzo http://www.nuovojava.it.

Tabella di autovalutazione

Sono stati raggiunti i seguenti obiettivi?

Obiettivo	Raggiunto	In Data
Conoscere le novità delle caratteristiche che sono state ufficializzate nelle versioni 16 e 17 (unità 21.1).	□	
Saper creare un file di installazione per applicazioni Java tramite il nuovo tool jpackage (unità 21.2).	□	
Comprendere la nuova caratteristica in anteprima introdotta in Java 17 nota come pattern matching per `switch` (unità 21.3).	□	
Conoscere le novità secondarie introdotte nelle versioni 16 e 17 (unità 21.4).	□	
Conoscere le novità delle caratteristiche che sono state ufficializzate nelle versioni 16 e 17 (unità 21.1).	□	