UNIVERSITÀ
DEGLI STUDI
FIRENZE

Scuola di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea in Informatica

Tesi di Laurea

EVOLUZIONE DI JAVA: LE PIÙ RECENTI INNOVAZIONI

EVOLUTION OF JAVA: THE MOST RECENT FEATURES

GABRIELE BERTINI

Relatore: *Battistina Venneri*

Anno Accademico 2018-2019

INDICE

ELENCO DELLE FIGURE

"There is a lot of ways known to do it wrong
and which one is right is not clear."
— James Gosling

INTRODUZIONE

Dalla sua nascita, il 23 maggio 1995, il linguaggio Java ha subito molti sviluppi, di cui alcuni particolarmente significativi, in una linea continua di evoluzione. In questo processo possiamo affermare che, per la comunità di sviluppatori Java, il rilascio di Java 8 è stato un punto di svolta fondamentale, con l'introduzione delle lambda-expressions e l'innesto della programmazione funzionale in un rigido impianto orientato agli oggetti.
Lo scopo di questa tesi è quello di prendere in esame le novità introdotte più recentemente. A tale scopo partiamo proprio dall'introduzione delle lambda come novità acquisita, su cui esiste ormai anche una larga letteratura e un'ampia esperienza di programmazione.

La domanda che ci poniamo è la seguente: quali sono, dopo le lambda, le più importanti novità introdotte nelle ultime versioni di Java, fino a Java 12?

Abbiamo individuato tre argomenti principali in questa evoluzione.

Il primo argomento, trattato nel Capitolo 1, riguarda tutte le nuove caratteristiche di cui si è arricchita la nozione di *Interface*. Mentre l'Interfaccia originale era legata al tipo di dato astratto, quindi rappresentava l'astrazione pura dei contratti con le classi clienti, la sua recente definizione si è arricchita di altri aspetti riguardanti le implementazioni dei servizi offerti. In primo luogo, sono stati introdotti i metodi con codice di default, poi i metodi statici e quelli privati. Nel Capitolo 1, quindi, analizziamo non solo le specificità linguistiche di questi aspetti, ma soprattutto l'uso di un concetto di Interfaccia così radicalmente modificato, l'impatto sullo sviluppo del codice, i limiti e i benefici.

Il secondo argomento, trattato nel Capitolo 2, riguarda invece il sistema di tipo di Java, mostrando come il controllo dei tipi da semplice *typechecking* sia progressivamente evoluto in vera e propria *type inference*. Ovviamente l'introduzione delle lambda-expressions è stato il primo passo in questa direzione. Ma la Local-Variable Type Inference di Java 10

e l'uso sempre più esteso dei tipi intersezione sono gli elementi aggiunti che permettono allo sviluppatore di scrivere codice meno verboso e meno pedante sui tipi, affidando il resto del lavoro a un'inferenza di tipo più sofisticata. Nel Capitolo 2, perciò, esaminiamo l'impatto che questi aspetti hanno soprattutto sul refactoring del codice e le linee-guida che si devono adottare nel programmare con queste nuove caratteristiche.

Il terzo argomento, trattato nel Capitolo 3, è quello che riteniamo più importante, ovvero l'introduzione di una programmazione modulare "proprietaria" per il linguaggio Java. In un linguaggio come Java, in cui i goal principali riguardano la riusabilità del codice, si sentiva la mancanza di uno strumento nativo per gestire la suddivisione in moduli. Esaminate le potenzialità di questo strumento, i moduli Java sono confrontati con altri strumenti per gestire programmi in moduli quali Maven o OSGi. Lo scopo del Capitolo 3 è quello di mostrare come si usa la modularizzazione in Java, le potenzialità di questo nuovo strumento e, soprattutto, far vedere come esso sia un effettivo passo avanti rispetto ai moduli Maven o all'uso di altri framework.

Infine, nel Capitolo 4, facciamo un breve accenno al potenziamento delle espressioni switch: questo aspetto non ci sembra particolarmente interessante, ma è dovuto un richiamo essendo esso l'unico aspetto linguistico di Java 12.

La metodologia usata per studiare gli argomenti suddetti, sui quali non esiste ancora una vera e propria letteratura, è stata basata sulla sperimentazione del loro uso sia nello sviluppo di nuovo codice che nel refactoring di codice preesistente (ossia, che non usa le nuove caratteristiche).

1

EVOLUZIONE DEL CONCETTO DI INTERFACCIA

Il concetto di *Interface* in Java è presente sin dalla nascita del linguaggio e rappresenta uno strumento fondamentale di astrazione e, quindi, di strutturazione del codice. Possiamo dire che il suo significato originario è legato alla nozione di "tipo di dato astratto", che è alla base dei linguaggi orientati agli oggetti basati su classi.
La classe definisce un tipo di oggetti con comportamento comune, dove l'interfaccia è la parte pubblica dei servizi offerti con i relativi contratti di uso (le signature dei metodi pubblici), mentre l'implementazione sia degli oggetti che dei comportamenti è nascosta alle classi client.
In questo senso poter astrarre l'interfaccia di una classe in una struttura del linguaggio, l'*Interface*, rappresenta l'uso più corretto di questa astrazione sui dati. Inoltre, permette di evidenziare l'uniformità di tipi (classi) di oggetti che offrono gli stessi servizi seppur con implementazioni diverse.
Quindi, per rispondere a questa finalità, una definizione di Interfaccia può contenere solo due tipi di elementi, costanti statiche e metodi astratti:

```
public interface SomeInterface {
   int SOME_CONSTANT = 35; // variable declaration
   int abstractMethod(int x, int y); // method declaration
}
```

Inoltre, proprio perché una Interfaccia non contiene implementazioni, è possibile ammettere l'ereditarietà multipla, che è uno strumento molto utile nella progettazione del codice.
Chiaramente, se una classe potesse ereditare implementazioni dalle

superclassi, sarebbero necessari ulteriori controlli in fase statica, per essere sicuri che lo stesso metodo non sia ereditato con implementazioni diverse da due diverse gerarchie. Invece, si può concedere che una classe erediti da più Interfacce, senza bisogno di questi controlli, quindi senza complicare il typechecker, proprio per il suo carattere completamente astratto.

Quanto detto sopra, però, è il concetto originario di *Interface* in Java, con i suoi limiti ma anche con la sua chiara connotazione di semplice contratto sui servizi offerti.
Possiamo affermare, invece, che la successiva evoluzione della definizione di interface, a partire da Java 8, rompe con questo punto di vista tradizionale, offrendo ulteriori vantaggi allo sviluppatore del codice ma perdendo di vista la differenza basilare fra classe e interfaccia. Il punto di svolta di questo cambiamento avviene in Java 8 con l'introduzione dei *metodi default* nelle interfacce; le evoluzioni successive non fanno che portare avanti questa linea diventando una vera e propria frattura con il concetto originario.

Per queste ragioni, nel presente capitolo partiamo proprio dall'introduzione dei metodi *default* (e *static*) in Java 8 (parr. 1 e 2). Quindi passeremo ad analizzare l'introduzione dei *metodi privati* nelle interfacce (par. 3). Per ognuno di questi passaggi, mostreremo le motivazioni, ossia i vantaggi offerti dalle nuove possiblità, ma anche gli aspetti negativi. Infine trarremo le nostre conclusioni su questo processo evolutivo della nozione di *Interface* nel paragrafo conclusivo.

1.1 LE NUOVE INTERFACCE IN JAVA 8

Fino a Java 7 le interfacce sono molto semplici, perché possono contenere sostanzialmente solo metodi *public abstract* e variabili costanti. Questi metodi dell'interfaccia devono essere implementati dalle classi che scelgono di implementare l'interfaccia.
Con Java 8 si introduce in Java il concetto di programmazione funzionale, attraverso l'uso della lambda-expression, che nel seguito chiameremo "lambda" per semplicità. Una nuova caratterizzazione del concetto di interfaccia, quella di *Interfaccia Funzionale* viene introdotta proprio per dare tipo alle lambda.
Un'interfaccia funzionale è un'interfaccia che prevede la definizione di

un solo metodo astratto.
Esempio:

```
@FunctionalInterface
public interface IAction {
  public void doWork();
}
```

Il ruolo di una tale interfaccia è quello di poter essere assegnata come tipo a una qualunque lambda, che concordi con la signatura del metodo astratto, sia nel tipo degli argomenti che nel tipo del risultato. Infatti, la lambda a cui viene assegnata un'interfaccia funzionale rappresenta, nel suo *body*, l'implementazione del relativo metodo astratto. In tal modo, anche il codice Java precedentemente sviluppato acquista maggiore flessibilità. Un metodo che richiede un parametro attuale il cui tipo è un'Interfaccia Funzionale può essere usato su qualunque valore, sia oggetti che lambda.

In secondo luogo, Java 8 introduce i metodi *default* e *static* nelle interfacce, dichiarati obbligatoriamente con il modificatore di accesso *public*. Questa caratteristica ci consente di estendere la gerarchia di sottoclassi di un'interfaccia, con nuove classi che hanno un comportamento aggiuntivo; ciò significa aggiungere nuovi servizi al contratto di un'interfaccia: introducendo un metodo default per questo nuovo comportamento, non si presentano possibili errori nelle sottoclassi preesistenti, ossia si preserva il contratto con le sottoclassi.
Esempio:

```
public interface CustomInterface {
   public abstract void method1();

   public default void method2() {
      System.out.println("default method");
   }

   public static void method3() {
      System.out.println("static method");
   }
}

public class CustomClass implements CustomInterface {
   @Override
   public void method1() {
```

```
        System.out.println("abstract method");
    }

    public static void main(String[] args) {
        CustomInterface instance = new CustomClass();
        instance.method1();
        instance.method2();
        CustomInterface.method3();
    }
}
```

Certamente, la motivazione di base per l'introduzione dei metodi default è stata quella suddetta, ossia la possibilità di estendere i servizi di un'interfaccia senza introdurre errori di compilazione. Ma possiamo affermare che anche le lambda hanno beneficiato di questa novità e probabilmente sono stati un motivo di spinta in questa direzione.
Adesso analizziamo in dettaglio l'introduzione dei modificatori *static* e *default* per la definizione di metodi all'interno di un'interfaccia.

1.1.1 *Metodi Static*

L'utilizzo più calzante per un metodo statico in un'interfaccia riguarda il *pattern creazionale* **Factory method**, il quale indirizza il problema della creazione di oggetti senza specificarne l'esatta classe. Questo pattern raggiunge il suo scopo fornendo un'interfaccia per creare un oggetto, ma lascia che le sottoclassi decidano quale oggetto istanziare.
Prendiamo per esempio un'interfaccia definita come segue:

```
public interface IntSequence {
    ...
    static IntSequence digitsOf(int n) {
        return new DigitSequence(n);
    }
}
```

La dichiarazione seguente mostra l'utilizzo del metodo, cioè la creazione statica dell'oggetto `digits`:
`IntSequence digits = IntSequence.digitsOf(1729);`
Il metodo statico produce un'istanza di alcune classi che implementano l'interfaccia data, ma al chiamante non importa quale classe sia.

1.1.2 *Metodi Default*

A partire da Java 8 è possibile fornire un'implementazione predefinita per qualsiasi metodo di un'interfaccia; per fare ciò è necessario utilizzare il modificatore *default* in testa alla dichiarazione del metodo.
Se associati a metodi di interfacce, permettono (al contrario dei metodi senza questo modificatore) non solo di dichiarare, ma anche di implementare, in modo differente gli stessi metodi nelle interfacce. È comunque d'obbligo l'override dei metodi nelle sottoclassi che implementano entrambe le interfacce, ma tramite un'istruzione ben definita si può invocare un preciso metodo implementato di una specifica interfaccia genitrice. Così nel caso del problema del diamante la classe D effettua l'override del metodo in comune a B e C definito in A e invoca lo specifico metodo di B o C.
Mostriamo un esempio di interfaccia con l'implementazione di un metodo default:

```
public interface IntSequence {
   default boolean hasNext() { return true; }
      // By default, sequences are infinite
   int next();
}
```

Una classe che implementa questa interfaccia può scegliere di sovrascrivere il metodo `hasNext` o ereditare l'implementazione predefinita. Per esempio:

```
public class FloatSequence implements IntSequence {
   ...
   public boolean hasNext() { return false; } // method override
}
```

Se una classe implementa due interfacce, una delle quali ha un metodo default e l'altra un metodo (default o meno) con la stessa *signature* (stesso nome e tipi di parametri), è necessario risolvere il conflitto. Per esempio, date due interfacce e una classe che le implementa entrambe come segue:

```
public interface InterfaceA {
 void performA();
 default boolean canPerform() { // return true if I can perform the
    action
```

```
  }
}

public interface InterfaceB {
  void performB();
  default boolean canPerform() { // return true if I can perform the
      action
  }
}

public class ConcreteC implements InterfaceA, InterfaceB {
}
```

Il codice soprastante fallirà con il seguente errore di compilazione:

```
error: unrelated defaults for canPerform()
from InterfaceA and InterfaceB
```

Per superare questo problema, è necessario fare un *override* del metodo predefinito:

```
public class ConcreteC implements InterfaceA, InterfaceB {
  public boolean canPerform() { } // method override
}
```

Supponiamo, però, di non voler fornire l'implementazione del metodo predefinito sovrascritto, ma di voler riutilizzare quello esistente. Possiamo fare ciò con la seguente sintassi:

```
public class ConcreteC implements InterfaceA, InterfaceB {
   public boolean canPerform() {
     return InterfaceA.super.canPerform();
   } // method override
}
```

Riassumendo:

- È possibile ereditare il metodo *default*;
- È possibile ridichiarare il metodo *default* essenzialmente rendendolo *abstract*;
- È possibile ridefinire il metodo *default* (equivalente a *override*).

Pertanto, da Java 8 in poi, gli elementi di un'interfaccia possono essere: costanti statiche, metodi astratti, metodi default, metodi statici, classi nidificate, interfacce nidificate, enumerazioni nidificate e annotazioni nidificate.

Di seguito un esempio in cui sono utilizzate tutte le novità sopra citate.

```
// generic interface with one type parameter T
public interface SomeInterface<T> {
   int SOME_CONSTANT = 35; // variable declaration
   int abstractMethod(int x, int y); // method declaration
   T abstractMethodUsingGenericType(T[] array, int i); // method
       using type parameter
   default int defaultMethod(int x, int y) {
      // implementation of method
   }
   static void main(String[] args) {
      // any static method, including main can be included in
          interface
   }
    // nested class definition
   class NestedClass {
      // members of a class
   }
   // nested interface definition
   interface NestedInterface {
      // member of an interface
   }
  // nested enum definition
   enum NestedEnum {
      OBJECT1,
      OBJECT2,
      ;
      // methods, variables and constructors
   }
    // nested annotation definition
   @interface NestedAnnotation {
      String attrib1();
   }
}
```

Ora, dato che abbiamo metodi default, i quali sono implementazioni, significa che abbiamo anche ereditarietà multipla di comportamento e

non solo di tipi. E sorge di nuovo il *diamond problem*, stavolta relativo alle interfacce. Poiché possiamo implementare il comportamento tramite metodi default, possiamo ora avere un codice comune ripetitivo, che può essere duplicato in più metodi default all'interno della stessa interfaccia. Per evitare ciò, normalmente rompiamo l'implementazione di un metodo in metodi più piccoli; e, poiché questi metodi possono non essere richiesti come visibili al di fuori dell'interfaccia, essi dovrebbero idealmente essere dichiarati con il modificatore *private*.

Mostriamo adesso un esempio di utilizzo dove possiamo notare alcuni limiti delle interfacce presenti in Java 8. In tale esempio abbiamo tre metodi default, all'interno dell'interfaccia, i quali svolgono tre operazioni identiche:

```
public interface DBLogging {
     String MONGO_DB_NAME = "ABC_Mongo_Datastore";
     String NEO4J_DB_NAME = "ABC_Neo4J_Datastore";
     String CASSANDRA_DB_NAME = "ABC_Cassandra_Datastore";

     // abstract method example
     void logInfo(String message);

     // default method example
     default void logWarn(String message){
       // Step 1: Connect to DataStore
       // Step 2: Log Warn Message
       // Step 3: Close the DataStore connection
     }
     default void logError(String message){
       // Step 1: Connect to DataStore
       // Step 2: Log Error Message
       // Step 3: Close the DataStore connection
     }
     default void logFatal(String message){
       // Step 1: Connect to DataStore
       // Step 2: Log Fatal Message
       // Step 3: Close the DataStore connection
     }
     // static method example
     static boolean isNull(String str) {
   System.out.println("Interface Null Check");
   return str == null ? true : "".equals(str) ? true : false;
     }
```

```
    // Any other abstract, default, static methods
}
```

È possibile notare la ridondanza di codice comune in tutti e tre i metodi `log`, poiché ognuno sta aprendo e chiudendo una connessione in modo autonomo.
Se volessimo avere un codice più pulito, dovremmo quindi spostare questa parte comune in un metodo `public`, ma così facendo sarebbe accessibile a tutte le altre classi del programma. Per evitare ciò potremmo allora implementare una classe astratta con un metodo privato per il codice comune ripetuto nei metodi `log`.

Oracle ha fatto fronte a questa evenienza nella versione 9 di Java, con l'introduzione dei metodi privati nelle interfacce, presentati nel paragrafo seguente.

1.2 METODI PRIVATI NELLE INTERFACCE

Una volta aggiunti i metodi *default* nelle interfacce, mancavano ancora i metodi *private*, impedendo così la suddivisione del codice in metodi più piccoli all'interno di un'interfaccia per far sì che si abbia *clean code*, cioè **refactoring**.
Fino a Java 8, il problema si verifica nel caso si abbia un metodo *default* troppo lungo, quindi impossibile da snellire estrapolando da esso un altro metodo.
Java 9 supera la problematica attraverso la possibilità di poter applicare il modificatore di accesso *private* (ed anche *private static*) ad implementazioni di metodi. Per esempio:

```
public interface MyInterface {
   default void defaultMethod() {
      privateMethod("Hello from the default method!");
   }
   private void privateMethod(final String string) {
      System.out.println(string);
   }
   void normalMethod();
}
```

Questa porzione di codice mostra come sia possibile rifattorizzare un metodo default, utilizzando un metodo private richiamato al suo interno, cosa che fino a questa versione del linguaggio non era possibile fare. Questa novità è stata introdotta a partire da Java 9 a Luglio 2017, argomentata in **JEP 213: Milling Project Coin** e creata da Joseph D. Darcy. Questi metodi privati migliorano la riutilizzabilità del codice all'interno delle interfacce. Ad esempio, se due metodi default necessitano di condividere parte di codice, un metodo private dell'interfaccia consente loro di farlo, ma senza esporlo alle sue classi di implementazione.

Riprendiamo adesso l'esempio fatto in precedenza per mostrare come abbiamo snellito il codice attraverso il refactoring, utilizzando un metodo private all'interno dell'interfaccia:

```
public interface DBLogging {
  String MONGO_DB_NAME = "ABC_Mongo_Datastore";
  String NEO4J_DB_NAME = "ABC_Neo4J_Datastore";
  String CASSANDRA_DB_NAME = "ABC_Cassandra_Datastore";

  default void logInfo(String message) {
    log(message, "INFO");
  }

  default void logWarn(String message) {
    log(message, "WARN");
  }

  default void logError(String message) {
    log(message, "ERROR");
  }

  default void logFatal(String message) {
    log(message, "FATAL");
  }

  private void log(String message, String msgPrefix) {
    // Step 1: Connect to DataStore
    // Step 2: Log Message with Prefix and styles etc.
    // Step 3: Close the DataStore connection
  }
  // Any other abstract, static, default methods
}
```

Ma quali sono i limiti e le regole da seguire per tale strumento?

1.2.1 *Limiti e regole di utilizzo*

L'utilizzo di metodi privati nelle interfacce prevede quattro regole, di cui tre di esse sono più adeguatamente inquadrabili come limiti:

1. Il metodo *private* dell'interfaccia non può essere *abstract*.
2. Il metodo *private* può essere utilizzato solo all'interno dell'interfaccia.
3. Il metodo *private static* può essere utilizzato all'interno di altri metodi static e non static dell'interfaccia.
4. I metodi *private non static* non possono essere utilizzati all'interno di metodi private static.

Mentre altrettante sono le regole per la scrittura di tale funzionalità:

1. Per definire questi metodi si usa il modificatore *private*.
2. I metodi *private* devono contenere un corpo.
3. Nessuna accessibilità minore rispetto al modificatore *private*.
    - In Java *private* è il modificatore di accesso meno visibile. Quindi non possiamo ridurre la sua visibilità da private a nessun altro modificatore.
4. Non è possibile utilizzare i modificatori *private* e *abstract* insieme perché hanno significati diversi e risulterebbe in un errore di compilazione.
    - Il metodo "privato" significa metodo completamente implementato perché le sottoclassi non possono ereditare e sovrascrivere questo metodo.
    - Il metodo "astratto" significa metodo di non implementazione. Quindi le sottoclassi devono ereditare e sovrascrivere questo metodo.

Riportiamo un esempio di applicazione di tali regole:

```
public interface CustomInterface {
   public abstract void method1();

   public default void method2() {
      method4(); //private method inside default method
      method5(); //static method inside other non-static method
      System.out.println("default method");
   }

   public static void method3() {
      method5(); //static method inside other static method
      System.out.println("static method");
   }

   private void method4() {
      System.out.println("private method");
   }

   private static void method5() {
      System.out.println("private static method");
   }
}

public class CustomClass implements CustomInterface {
   @Override
   public void method1() {
      System.out.println("abstract method");
   }

   public static void main(String[] args){
      CustomInterface instance = new CustomClass();
      instance.method1();
      instance.method2();
      CustomInterface.method3();
   }
}
```

Gli elementi supportati in un'interfaccia da Java 9 in poi sono: costanti statiche, metodi astratti, metodi default, metodi statici, metodi privati, classi nidificate, interfacce nidificate, enumerazioni nidificate e annotazioni nidificate.
Di seguito un esempio in cui sono presenti tutti gli elementi sopracitati:

```
// generic interface with one type parameter T
public interface SomeInterface<T> {
   int SOME_CONSTANT = 35; // variable declaration
   int abstractMethod(int x, int y); // method declaration
   T abstractMethodUsingGenericType(T[] array, int i); // method
       using type parameter
   default int defaultMethod(int x, int y) {
      // implementation of method
      // can call the privateMethod and privateStaticMethod here
   }
   static void main(String[] args) {
      // any static method, including main can be included in
          interface
      // can call privateStatic method here
   }
   private int privateMethod(int x, int y) {
      // private method implementation
   }
   private static void privateStaticMethod(int x, int y) {
      // private method implementation
   }
// nested class definition
   class NestedClass {
      // members of a class
   }
  // nested interface definition
   interface NestedInterface {
      // member of an interface
   }
// nested enum definition
   enum NestedEnum {
      OBJECT1,
      OBJECT2,
      ;
      // methods, variables and constructors
   }
  // nested annotation definition
```

```
   @interface NestedAnnotation {
      String attrib1();
   }
}
```

Riassumendo, i metodi *private* dell'interfaccia in Java 9 possono essere *static* o *di istanza*. In entrambi i casi, il metodo private non è ereditato da sotto-interfacce o implementazioni.
Essi sono principalmente utili per migliorare la riutilizzabilità del codice all'interno dell'interfaccia, migliorando così l'*incapsulamento*, e forniscono la possibilità di esporre ai clients solo le implementazioni dei metodi previsti.
Rivediamo tutti i tipi di metodi consentiti in Java 9:

TIPO METODO	**INTRODUZIONE**
`public abstract`	Java 7
`public default`	Java 8
`public static`	Java 8
`private`	Java 9
`private static`	Java 9

1.3 CONCLUSIONI

È interessante notare come la natura di un'interfaccia si sia evoluta, pur mantenendo la *retrocompatibilità* con le versioni precedenti. Prima di Java 8, un principio fondamentale di un'interfaccia era la sua completa connotazione come puro tipo astratto. Invece, da Java 8, un'interfaccia può avere al suo interno anche metodi non abstract, e da Java 9 in poi può anche contenere metodi *private*.
Con Java 9 la diversificazione tra interfaccia e classe astratta si è quasi perduta; la differenza principale è individuabile nello stato della classe (rappresentato dall'insieme delle variabili di stato o *attributi*) che caratterizza una classe Java, e quindi anche una classe astratta.

Di seguito riportiamo una tabella che mette a confronto i due strumenti, *Interface* e *Abstract class*, in modo da esplicitarne le differenze:

	INTERFACCIA	**CLASSE ASTRATTA**
Istanziabile	no	no
Fields	solo `static final`	sì
Costruttore	no	sì
Metodi statici	da Java 8	sì
Dichiarazione metodi virtual	no	sì
Implementazione metodi	da Java 8 con modificatore `default`	sì

Come si evince dalla tabella soprastante, non ci sono particolari differenze degne di nota tra interfacce e classi astratte.
Perciò possiamo concludere che, anche se tale novità porta in dote un notevole refactoring del codice all'interno di un'interfaccia, la sua evoluzione rappresenta anche la perdita di quella struttura del linguaggio che era destinata a descrivere un tipo di dato come puro contratto con le altre classi.

2

ESTENSIONE DELLA TYPE INFERENCE

In questo capitolo vogliamo illustrare come il linguaggio Java stia evolvendo, facendo fronte all'esigenza degli sviluppatori di avere meno verbosità nel codice. La strada intrapresa per fare ciò, è quella di aumentare le potenzialità di uso dell'inferenza di tipo.
Partiamo con una definizione generale di *typechecking* e *type inference*, soprattutto per evidenziarne la differenza.
Il typechecking è quell'algoritmo che esamina codice in cui i tipi delle variabili sono esplicitamente fornite nel codice stesso: utilizzando queste annotazioni di tipo, esso controlla che siano rispettati i vincoli derivati dal contesto, per concludere che l'espressione è ben tipata, assegnandogli il suo (unico) tipo oppure concludendo che l'espressione non ha un tipo.
Si parla, invece, di type inference quando il codice non contiene annotazioni esplicite di tipo, quindi il tipo dell'espressione, se essa ne ha uno, deve essere ricostruito deducendo i tipi opportuni per le variabili.
In tal caso, molto spesso, un'espressione ben tipata può avere diversi tipi, ossia essere "polimorfa"; quindi, è necessario poter definire un tipo principale, che rappresenta tutti gli altri e che viene inferito dall'algoritmo di controllo dei tipi. [1]

Possiamo dire che ogni linguaggio si caratterizza, in primo luogo, per la scelta basilare di uno dei due suddetti approcci, insieme con la scelta di altri aspetti correlati quali

i. la definizione del sistema di tipi utilizzati;

ii. la caratteristica di essere o meno *"strongly typed"* (ossia, ogni sottoespressione di un'espressione ben tipata ha un tipo);

iii. la scelta sul momento in cui il controllo dei tipi viene effettuato, ossia in fase statica (linguaggi *statically-typed*) o a run-time (linguaggi *dynamically-typed*).

Chiaramente ciascuno dei due approcci, typechecking e type inference, ha i suoi vantaggi e svantaggi: il primo è più espressivo, permette di ridurre molti errori concettuali, ha minore complessità ed è molto importante per la modularità; mentre il secondo è sicuramente più complesso ma riduce notevolmente il tempo di scrittura del codice e diminuisce la sua verbosità.

Ritornando a Java, possiamo affermare che la base del suo sistema di tipo è il fatto che le variabili devono essere esplicitamente annotate con il loro tipo e, quindi, l'algoritmo di controllo dei tipi è essenzialmente un algoritmo di typechecking. Basti pensare al fatto che i parametri formali di un metodo devono essere dichiarati insieme con il loro tipo e che ogni nuovo identificatore viene introdotto con il suo tipo esplicito. È vero che Java, come tutti i linguaggi orientati agli oggetti, adotta il polimorfismo per sottotipo: ma questo è realizzato dal typechecker assegnando ad ogni espressione il suo unico tipo, derivato dalla sua struttura sintattica, assumendo, però, che la stessa espressione sia utilizzabile in ogni punto in cui sia richiesta dal contesto un'espressione di un supertipo (*principio di Liskov*).

Tuttavia, come tutti i linguaggi più avanzati, l'evoluzione di Java negli ultimi anni è sempre andata nella direzione di potenziare la parte di type inference. La prima apparizione di inferenza di tipo vera e propria è collocabile in Java con l'introduzione dei *Generics*. Ad esempio, i metodi generici permettono di essere invocati come un qualsiasi altro metodo, senza dover specificare il tipo concreto che deve essere rimpiazzato al parametro di tipo: questo sarà inferito dal typechecker usando le informazioni fornite dal contesto. Per illustrare questo ultimo punto, si consideri il caso seguente:

```
static <T> T pick(T a1, T a2) { return a2; }
Serializable s = pick("d", new ArrayList<String>());
```

Qui l'inferenza di tipo deduce che il secondo argomento passato al metodo `pick` è di tipo `Serializable`.

In questo capitolo analizziamo alcuni casi specifici di inferenza di tipo in Java, seguendone la progressiva evoluzione fino ad arrivare all'ultima

importante novità, in cui tramite la parola chiave **var**, possiamo dichiarare nuove variabili senza scriverne esplicitamente il tipo. Lo scopo finale è quello di mettere in rilievo non solo le nuove potenzialità introdotte, ma soprattutto i limiti di uso e, quindi, le possibili direzioni di sviluppo futuro del linguaggio.

2.1 I PRIMI CASI DI INFERENZA DI TIPO

La prima apparizione dell'inferenza di tipo è avvenuta in Java 5 con l'introduzione di metodi *generici*.
Per esempio, invece di

```
List<String> cs = Collections.<String>emptyList();
```

possiamo scrivere semplicemente

```
List<String> cs = Collections.emptyList();
```

Da Java 7 è possibile omettere i parametri di tipo dei Generics in un'espressione, quando il contesto permette di dedurli.

Per esempio,

```
Map<String, List<String>> myMap =
  new HashMap<String,List<String>>();
```

può essere abbreviato nel modo seguente, utilizzando l'operatore diamond <>:

```
Map<User, List<String>> userChannels = new HashMap<>();
```

L'idea generale è che il compilatore possa inferire il tipo basato sul contesto. In questo caso, la `HashMap` contiene un elenco di stringhe come indicato sul lato sinistro della dichiarazione.

L'ambito dell'inferenza di tipo è stato significativamente ampliato in Java 8, inclusa l'inferenza estesa per le chiamate nidificate (metodo che richiama un altro metodo nella stessa classe) e concatenate (es. `Stream`) di un metodo generico e l'inferenza per i parametri formali delle *lambda expressions*. Ciò ha reso molto più facile la creazione di API progettate per il concatenamento delle chiamate e tali API (come gli `Stream`) sono abbastanza popolari, dimostrando che gli sviluppatori sono già a loro agio con i tipi intermedi inferiti.

Consideriamo la seguente catena di chiamate:

```
int maxWeight = blocks.stream()
     .filter(b -> b.getColor() == BLUE)
     .mapToInt(Block::getWeight)
     .max();
```

nessuno si preoccupa (o nemmeno nota) che i tipi intermedi `Stream` e `IntStream`, così come il tipo di parametro formale `b` della lambda, non compaiono esplicitamente nel codice sorgente.
In particolare, vogliamo soffermarci sul fatto che nelle *lambda expressions*, introdotte in Java 8, è possibile omettere i tipi dei parametri formali e lasciarli dedurre all'algoritmo di tipo.

Per esempio, lambda expressions come

```
Predicate<String> nameValidation =
                (String x) -> x.length() > 0;
```

possono essere scritte più semplicemente come

```
Predicate<String> nameValidation = x -> x.length() > 0
```

omettendo di esplicitare il tipo del parametro `x`.

In questi casi, il contesto deve offrire, però, il *target type* della lambda, ossia il tipo che il typechecker deve assegnare alla lambda se questa può avere quel tipo; il target type prescrive sia il tipo dei parametri formali che il tipo del risultato.
Nell'esempio sopra, il target type di `x -> x.length() > 0` è `Predicate<String>`, che nella signature del suo metodo astratto precisa che il tipo del parametro `x` deve essere `String`. In qualche modo, quindi, è come se la lambda avesse la variabile annotata con tipo esplicito.
Dunque, non dobbiamo pensare che scrivere una lambda senza annotazioni esplicite di tipo sulle variabili significhi che per la lambda l'algoritmo deduca tutti i suoi possibili tipi o un loro rappresentante.
Le lambda di Java sono *polytyped* non *polimorfe*. La stessa espressione lambda può avere anche tipi diversi (non correlati da alcuna relazione di sottotipo), ma nel momento in cui viene introdotta essa ha assegnato il suo tipo unico e con questo si confronta il typechecker quando tipa le espressioni in cui la lambda è usata.

Notiamo, però, che sebbene l'inferenza di tipo sia migliorata molto in Java 8 con l'introduzione delle lambda expressions, method references e

Streams, le variabili locali in Java 8 devono ancora essere associate al loro tipo.

Man mano che i tipi crescono di dimensioni, come Generics parametrizzati da altri tipi Generics, l'inferenza di tipo può facilitare la leggibilità del codice.
I linguaggi Scala e C# consentono di sostituire il tipo in una dichiarazione di variabile locale con la keyword var e il compilatore *compila* il tipo appropriato dall'inizializzatore di variabile. Ad esempio, la dichiarazione di `userChannels` mostrata in precedenza potrebbe essere scritta in questo modo:

```
var userChannels = new HashMap<User, List<String>>();
```

o restituita da un metodo che restituisce una lista:

```
var channels = lookupUserChannels("Tom");
channels.forEach(System.out::println);
```

Per il linguaggio Java questa importante caratteristica è stata introdotta solo a partire dalla versione 10, che porta in dote un nuovo oggetto per l'inferenza di tipo, ovvero la *Local-Variable Type Inference*.

2.2 INFERENZA DI TIPO PER VARIABILI LOCALI

La *Local-Variable Type Inference* è una novità del linguaggio Java introdotta a partire da Java 10 (Marzo 2018), discussa e argomentata in **JEP 286: Local-Variable Type Inference** [2] (Java Enhancement Proposal) e creata da Brian Goetz, autore di Java Concurrency in Practice, uno dei libri più popolari per gli sviluppatori Java.
Con tale strumento Java viene dotato di una nuova importante funzionalità, che estende l'inferenza dei tipi anche alla dichiarazione di variabili locali con inizializzatore.
La parola *var* permette di dichiarare una variabile senza specificarne il tipo. Per esempio, invece di scrivere `String str = "Java"`, adesso possiamo semplicemente scrivere `var str = "Java"`. Il tipo di riferimento viene dedotto al momento della compilazione, basandosi sul tipo di valore ad essa associato nell'assegnamento.

L'identificatore var non è una *keyword* in Java, ma è un *reserved type name*; ciò significa che il codice che utilizza var come variabile, metodo o nome del pacchetto non sarà interessato, mentre il codice che utilizza var come classe o nome dell'interfaccia sarà interessato (ma questi nomi sono rari nella pratica, poiché violano le convenzioni di denominazione usuali). Quindi var può essere ancora usata in altri punti del codice, ad esempio come variabile o nome di classe; ciò consente a Java di rimanere retro-compatibile con il codice pre-Java 10 in cui un programmatore può aver utilizzato tale parola per nominare una variabile.

Gli sviluppatori Java hanno a lungo lamentato il *boilerplate code* e le formalità necessarie durante la scrittura del codice. Molte cose che richiedono solo 5 minuti in linguaggi come Python o JavaScript possono richiedere più di 30 minuti in Java a causa della sua verbosità.
Il termine "boilerplate code" si riferisce a sezioni di codice che devono essere incluse in molte parti con modifiche minime o nulle. Viene spesso utilizzato quando si fa riferimento a linguaggi considerati verbosi, cioè in cui il programmatore deve scrivere molto codice per eseguire lavori minimi.

Un noto blog sul linguaggio Java ha effettuato un'intervista [3] ai più importanti influencers in questo campo, riguardante le loro aspettative per le novità attese in Java 10. Alla domanda "cosa ti piacerebbe vedere in Java 10", colpisce particolarmente la risposta di Trisha Gee, sviluppatore Java ed esperta del linguaggio:
"Mi piacerebbe vedere introdotta la Local-Variable Type Inference. Per molto tempo, sono stata uno di quei programmatori Java più longevi a cui non importava di tutto il boilerplate perché nel tempo diventa invisibile a chi scrive il codice. Ma più diminuiamo il boilerplate (ad esempio con lambda expressions, riferimenti al metodo e i nuovi *factory methods* della classe `Collections`) più mi rendo conto che una lingua può essere succinta senza perdere ogni significato." [4]

In effetti, possiamo tranquillamente affermare che la nuova caratteristica del **var** semplifica notevolmente la verbosità del codice.

Chi scrive codice in Scala, Kotlin, Go, C# o in qualsiasi altro linguaggio che opera su JVM (Java Virtual Machine), sa certamente che tutti questi linguaggi hanno una sorta di inferenza di tipo di variabile locale già integrata. Ad esempio, JavaScript ha *let* e *var*, Scala e Kotlin hanno *var* e *val*, C++ ha *auto*, C# ha *var*, e Go ha il supporto tramite dichiarazione con

l'operatore := e la keyword *var*.
Mentre linguaggi come Scala utilizzano la keyword *val* per dichiarare una variabile immutabile, per Java possiamo aggiungere il modificatore *final* a una dichiarazione di variabile con la keyword *var* :
`final var person = new Person()`.

Ad esempio, la dichiarazione

```
HashMap<String, String> esempio = new HashMap<>();
```

può essere abbreviata in Java 10 come

```
var esempio = new HashMap<String, String>();
```

Ad ogni modo, l'inferenza del tipo di variabile locale con la keyword var di Java 10 può essere utilizzata solo per dichiarare variabili locali, come per esempio qualsiasi variabile all'interno di un corpo del metodo o di un blocco di codice di inizializzazione, per gli indici di un "ciclo `for` potenziato" (es. `i = i+2`), per le lambda expressions, e per le variabili locali dichiarate all'interno di un ciclo `for` tradizionale.
Non è possibile però utilizzare var per dichiarare variabili globali della classe, variabili formali di un costruttore, parametri formali, per restituire il tipo di metodi, o qualsiasi altro tipo di dichiarazione di variabile. D'altra parte, non è neanche possibile dichiarare con `var` una variabile a cui si assegna una lambda expression senza annotazioni di tipo sui parametri: questo è evidente da quanto detto nella sezione precedente, perché in tal caso il contesto non può fornire il target type, ossia la signature, della lambda.

Infine, seguendo le dichiarazioni dello stesso autore Brian Goetz, vogliamo brevemente ricordare le osservazioni che stanno alla base della scelta sintattica. Ci sono diverse opinioni al riguardo e diverse scelte in altri linguaggi. I due principali gradi di libertà qui sono le parole chiave da usare (var, auto, ecc.) e se avere un nuovo modulo separato per le variabili locali immutabili (val, let).
Sono state prese in considerazione le seguenti opzioni sintattiche:

- `var x = expr` (come C#)
- `var`, più `val` per le variabili locali immutabili (come Scala, Kotlin)
- `var`, più `let` per le variabili locali immutabili (come Swift)
- `auto x = expr` (come C++)

- `const x = expr` (già reserved word in Java)
- `final x = expr` (già reserved word in Java)
- `let x = expr`
- `def x = expr` (come Groovy)
- `x := expr` (come Go)

Dopo aver raccolto input sostanziali, `var` è stato preferito rispetto agli approcci di Groovy, C++ o Go.

2.2.1 *Limiti e problemi della Local-Variable Type Inference*

Si consideri l'esempio seguente con variabile globale:

```
public void aMethod() {
   var name = "Java 10";
}
// But the following is NOT OK
class aClass {
   var list; // compile time error
}
```

Si consideri ora un esempio con variabile locale:

```
public long countNumberofFiles(var fileList);
// Compilation error because compiler cannot
// infer type of local variable fileList;
// cannot use 'var' on variable without initializer
```

Dunque, la prima osservazione è che non è possibile inizializzare una variabile `var` a `null`. Assegnando `null`, non è chiaro quale dovrebbe essere il tipo, poiché in Java qualsiasi riferimento a un oggetto può essere nullo.

Esempio:

```
var count = null; // Compilation error because compiler
                  // cannot infer type for local variable count
```

Ciò è possibile solo esplicitando il tipo attraverso un cast, come ad esempio `Object` o un qualsiasi altro tipo:

```
var cast = (Object) null;
System.out.println(cast); // Prints "null"
```

Abbiamo però rilevato una incongruenza nel comportamento di *var* quando sul lato destro della dichiarazione abbiamo un'espressione condizionale contenente il valore *null*.

Prendiamo per esempio questa dichiarazione:

```
Object goodObj = (true ? null : new A()); // compiles and assigns
    null
```

Essa compila ed assegna un valore `null` alla variabile di tipo `Object`.
Se però scriviamo la seguente dichiarazione (in cui `A` può essere un tipo qualsiasi)

```
var test = (true ? null : new A()); // compiles but throws null
    pointer exception at runtime
```

ci aspetteremmo di avere un errore di compilazione.

Ciò è dato dal fatto che, in seguito al calcolo della condizione, potrebbe venire assegnato il valore `null` alla variabile `test` e ciò è in netta contraddizione con la regola dettata dal linguaggio e citata in precedenza all'inizio del paragrafo. In realtà la JVM (Java Virtual Machine) lancerà una `Null Pointer Exception` a runtime e questo porta ad una **ambiguità di comportamento**.
Purtroppo nella documentazione Java non è stato fatto alcun riferimento a questo caso, perciò non sappiamo se tale ambiguità è intenzionale oppure semplicemente una situazione non gestita.

Il secondo limite riguarda l'inizializzazione di una variabile con una lambda expression. Abbiamo già parlato di questo aspetto nel paragrafo precedente: non è possibile utilizzare l'inferenza del tipo di variabile locale con le lambda expressions, poiché queste richiedono un *target type* (tipo di destinazione) esplicito. Esempio:

```
var z = () -> {} // Compilation error because compiler
                 // cannot infer type for local variable z
```

Ogni istruzione contenente la keyword `var` deve avere un tipo, e un unico tipo, che rappresenta il tipo di espressione associata. Ricordiamo che

Java è un linguaggio staticamente tipato (diversamente da JavaScript) e, quindi, devono esserci sufficienti e univoche informazioni di tipo per poter dedurre il tipo di una variabile locale. Se non c'è, il typechecker fallisce, ad esempio:

```
var id = 0; // At this moment, compiler interprets
            // variable id as integer
id = "34"; // Compilation error because of incompatible types:
           // java.lang.String can't be converted to int
```

Ora mostriamo uno scenario d'uso con l'ereditarietà. Supponiamo che ci siano due sottoclassi (`Doctor`, `Engineer`) estese dalla classe padre `Person`. Di seguito un esempio di creazione di un oggetto `Doctor`:
`var p=new Doctor();`

Una variabile dichiarata con var è sempre il tipo dell'inizializzatore (`Doctor`, in questo caso), e var non può essere usato quando non c'è alcun inizializzatore. Se proviamo a riassegnare la variabile la compilazione fallisce:

```
p = new Engineer(); // Compilation error because of
                    // incompatible types
```

Così possiamo concludere che il *polimorfismo per sottotipo* non funziona con la keyword var.

Stranamente la seguente assegnazione è valida data la presenza di un inizializzatore esplicito sul lato destro: `var list = new ArrayList<>();` Il tipo della variabile inferita è `ArrayList`, ma non è particolarmente utile in quanto non si beneficia dei generics; quindi questa assegnazione è legale ma si consiglia di evitarla.

Ci sono casi in cui il codice può essere difficile da leggere con l'utilizzo di questa funzionalità. Per esempio con la seguente dichiarazione
`var x = someFunction();`
non sappiamo a priori il tipo di ritorno restituito dalla funzione; infatti per capire il tipo della variabile `x` dobbiamo rintracciare la funzione `someFunction()` e il suo tipo di ritorno.

Per il momento abbiamo visto come si comporta questo strumento quando il target type è un tipo definito (o `Denotable`, per es. `int`, `String` o `Integer`), ma vediamo cosa succede se il tipo è `Non Denotable`.

2.2.2 *Inferenza di tipo con Non Denotable Types*

Espressioni il cui tipo non è deducibile, in Java sono chiamate *Non Denotable Types*, vale a dire tipi che possono esistere all'interno del programma, ma per i quali non c'è modo di scriverne esplicitamente il nome.
A volte il tipo di inizializzatore è un *Non Denotable*, come un tipo di variabile catturato (una lambda expression è in grado di accedere a variabili dichiarate al di fuori del corpo della funzione lambda in determinate circostanze), un'intersezione di tipo o un tipo di classe anonimo.
In tali casi, abbiamo la possibilità di scegliere se

i. inferire il tipo,

ii. rifiutare l'espressione,

iii. dedurre un supertipo denotabile.

Un buon esempio di tipo `Non Denotable` è una *classe anonima*: è possibile aggiungere campi e metodi, ma non scriverne il nome. L'operatore diamond non può essere utilizzato con classi anonime.
Il `var` è meno ristretto e può essere usato per supportare alcuni tipi Non Denotable, in particolare *classi anonime* e *intersezione di tipi* (forma di generics come per esempio `<T extends A & B>`, dove `T` è un parametro di tipo mentre `A` e `B` sono tipi).

Riportiamo un esempio di intersezione di tipi, in cui trasformiamo una lista di `double` in una lista di `int`:

```
var numbers = List.of(1.1, 2.2, 3.3, 4.4, 5.5);
var integers = toIntegerList(numbers);
System.out.println(integers); // prints [1, 2, 3, 4, 5]
...
static <T extends Number & Serializable> List<Integer>
    toIntegerList(List<T> numbers) {
  var integers =
   numbers.stream().map(Number::intValue)
     .collect(Collectors.toList());
  return integers;
}
```

Per fare ciò abbiamo utilizzato un metodo statico, il quale restituisce una intersezione di tipi, che è stato assegnato ad una variabile dichiarata con l'utilizzo di `var`.

La keyword var ci consente inoltre di utilizzare le classi anonime in modo più efficace e di fare riferimento a tipi che sarebbero altrimenti impossibili da descrivere. Normalmente se si crea una classe anonima è possibile aggiungervi dei campi, ma non è possibile fare riferimento a quei campi altrove perché è necessario assegnarli a un tipo denominato.
Ad esempio il seguente frammento di codice non verrà compilato poiché il tipo di `productInfo` è `Object` e non è possibile accedere ai campi `nome` e `total` da un oggetto di tipo `Object`.

```
Object productInfo = new Object() {
   String name = "Apple";
   int total = 30;
};
System.out.println("name = " + productInfo.name + ", total = " +
    productInfo.total);
```

Con var possiamo superare questi limiti. Quando assegniamo una classe anonima a una variabile locale di tipo `var`, viene inferito il tipo della classe anonima, piuttosto che la sua classe padre. Ciò significa che si può fare riferimento ai campi dichiarati nella classe anonima.

```
var productInfo = new Object() {
  String name = "Apple";
  int total = 30;
};
System.out.println("name = " + productInfo.name + ", total = " +
productInfo.total);
```

Per comprendere al meglio qual è il funzionamento di var con i tipi intersezione e per capirne gli eventuali limiti, ci siamo posti la seguente domanda: è possibile assegnare a una variabile dichiarata con `var` un'espressione condizionale (o, più comunemente, operatore ternario)?
Proviamo a costruirci un esempio per verificare ciò. Definiamo quattro interfacce legate tra loro attraverso l'ereditarietà:

```
public interface I1 {}
public interface I2 {}
public interface I3 extends I1, I2 {}
public interface I4 extends I1, I2 {}
```

E due classi `A` e `B` che implementano `I3` e `I4` rispettivamente:

```
public class A implements I3 {}
public class B implements I4 {}
```

Vogliamo verificare se la dichiarazione
`var test = (true ? new A() : new B());` è ben tipata.

Questo esempio è correlato all'introduzione dei tipi intersezione in Java. Quindi, nel caso di un'espressione condizionale in cui i due rami hanno tipi diversi, il *typechecker* inferisce come tipo dell'espressione il *LUB* (Least Upper Bound) dei tipi dei due rami, ovvero il minimo supertipo comune. Questo tipo può anche essere un tipo non nominale, quando non esiste un nome di *Interfaccia* o *Classe* che rappresenti il minimo supertipo comune. In tal caso il LUB è `lub(A, B) = I1&I2` che è il tipo intersezione, e soprattutto questo tipo è il tipo assegnato a var.
Successivamente, viene calcolata a runtime la condizione dell'espressione e l'operando scelto sarà assegnato a var, come possiamo osservare mostrandone il tipo dopo aver eseguito il programma:
`System.out.println(test.getClass()); // Prints "Class A"`

A sostegno di questo fatto riportiamo un frammento di testo esplicativo: "Il secondo e il terzo operando sono di tipi S1 e S2 rispettivamente. Sia T1 il tipo risultante dall'applicazione della conversione di boxing in S1, e sia T2 il tipo risultante dall'applicazione della conversione di boxing in S2. Il tipo della espressione condizionale è il risultato dell'applicazione della conversione di cattura a lub(T1, T2)." [5]

Riportiamo adesso un esempio concreto, in cui i due valori dell'operatore sono di tipi diversi (per es. `Integer` e `String`):

```
var y = 1 > 0 ? 10 : "Less than zero";
System.out.println(y.getClass()); // Integer
var z = 1 < 0 ? 10 : "Less than zero";
System.out.println(z.getClass()); // String
```

Questo è ovviamente confinato nei limiti del typechecking statico. Per esempio,

```
Serializable x = 1 < 0 ? 10 : "Less than zero";
System.out.println(x.getClass()); // Prints Serializable
```

`Serializable` è un tipo comune compatibile e il più specializzato per i due diversi operandi (il meno specializzato è `Object`).
Sia `String` che `Integer` implementano `Serializable` (il tipo `Integer`

viene dall'auto-box da `int`). In altre parole, `Serializable` è il LUB dei due operandi, perciò questo esempio mostra che il tipo `var` è anche `Serializable`.

Non tutti i tipi `Non Denotable` possono essere usati con var, sono supportate solo classi anonime e tipi intersezione. I tipi catturati da un *Wildcard* non vengono dedotti in modo da evitare che messaggi di errore relativi a wildcard ancor più criptici vengano esposti ai programmatori Java.
L'obiettivo di supportare tipi Non Denotable è sempre quello di conservare quante più informazioni possibili nel tipo inferito e consentire l'inferenza di variabili locali per il refactoring del codice.

Possiamo riassumere brevemente le considerazioni che hanno portato gli autori ad utilizzare la keyword var con i tipi Non Denotable nei punti seguenti.

- Una variabile di tipo `null` è praticamente inutile e non esiste una buona alternativa per un tipo inferito, quindi sono escluse.
- Consentire alle variabili catturate di scorrere nelle dichiarazioni successive aggiunge nuova espressività al linguaggio, ma non è un obiettivo di questa funzionalità. Invece, l'operazione di proiezione proposta è quella che dobbiamo comunque utilizzare per risolvere vari bug nel sistema di tipi, ed è ragionevole applicarlo qui.
- Le intersezioni di tipi sono particolarmente difficili da mappare a un supertipo: non sono ordinati, quindi un elemento dell'intersezione non è intrinsecamente "migliore" degli altri. La scelta stabile per un supertipo è il lub di tutti gli elementi, ma spesso sarà `Object` o qualcosa di altrettanto inutile. Quindi sono permesse.
- I tipi di classi anonime non possono essere nominati, ma sono facilmente comprensibili: sono semplicemente classi. Consentire alle variabili di avere tipi di classi anonime introduce una utile "shorthand" per dichiarare un'istanza **singleton** (design pattern creazionale il cui scopo è garantire che per una determinata classe venga creata una e una sola istanza, e di fornire un punto di accesso globale a tale istanza) di una classe locale. Perciò sono permessi.

Le ultime versioni di Java sembrano continuare la costante evoluzione del linguaggio in questa direzione. Ad esempio in Java 11 l'uso della keyword var sarà consentito all'interno dei parametri di una lambda expression.

Questo è utile perché consente di avere un parametro formale di cui viene inferito il tipo, ma su cui si può ancora aggiungere annotazioni Java, ad esempio: `(@Nonnull var x, var y) -> x.process(y)`.

2.3 INFERENZA E PARAMETRI DELLE LAMBDA EXPRESSIONS

Come abbiamo visto nella sezione precedente, a proposito dell'inferenza di tipo per variabili locali, non è possibile utilizzare var con identificatori per lambda expressions, poiché le lambda richiedono un *target type* esplicito.
Quindi, avendo una lambda expression come `(int x) -> x + 1`, non è possibile assegnarla a un'espressione come
`var foo = (int x) -> x + 1;`
la quale risulterebbe in un errore di compilazione dato che la variabile `foo`, avendo come identificatore `var`, non ha un tipo di destinazione esplicito.

In Java 10, l'uso della keyword var per l'inferenza di tipo è, inoltre, proibito quando si dichiara l'elenco dei parametri delle lambda expressions implicitamente tipizzate, perciò non è possibile scrivere codice come
`int x = (var foo, var bar) -> foo + bar`.
Vediamo ora come questo limite sia stato superato, potenziando così le funzionalità delle lambda expressions, in Java 11.

Con l'introduzione di Java 11, la Local-Variable Type Inference è stata estesa ai parametri delle lambda expressions. L'obiettivo è quello di allineare la sintassi di una dichiarazione di parametro formale in una lambda implicitamente tipizzata con la sintassi di una dichiarazione di variabile locale. Tutto ciò è argomentato in **JEP 323: Local-Variable Syntax for Lambda Parameters**.

Adesso possiamo quindi scrivere dichiarazioni come
`ITest divide = (var x, var y) -> x / y;`
quando il tipo dei parametri `x` e `y` può essere inferito dal loro uso nel corpo della funzione.
L'obiettivo di questa funzione è quello di consentire a var di essere utilizzato per dichiarare i parametri formali di un'espressione lambda implicitamente tipizzata.

A questo punto ci domandiamo quale sia il senso o il vantaggio di usare var per i parametri delle lambda. Prima di Java 11 abbiamo due modi di dichiarare i parametri delle lambda expressions:

- con tipo esplicito:
  `ITest exp = (double x, double y) -> Math.pow(x, y);`
- con tipo implicito: `ITest multiply = (x, y) -> x * y;`

Nel primo esempio dichiariamo esplicitamente che le variabili `x` e `y` sono di tipo `double`. Nel secondo esempio, incarichiamo la type inference di dedurre il tipo di `x` e `y` sulla base del loro uso. Questo sembra del tutto equivalente all'uso della var per i parametri della lambda:
`ITest divide = (var x, var y) -> x / y;`

Una ragione per introdurre questa funzionalità potrebbe essere solo la volontà di uniformare la sintassi per la Local Variable Type-Inference. Rendere coerente la sintassi di var tra le variabili locali offre il vantaggio della produttività in quanto gli sviluppatori faranno meno errori dovuti all'utilizzo della keyword in posizioni sbagliate.
A nostro parere, questa è comunque una ragione insignificante per aggiungere tale caratteristica. Gli sviluppatori in genere hanno già il compito di imparare e ricordare molte cose; dover ricordare che non è possibile utilizzare var in una dichiarazione formale lambda probabilmente non sarebbe un impegno così grande.

È stata inoltre data la possibilità agli sviluppatori di aggiungere annotazioni alle variabili locali di una lambda expression, le quali hanno tipo esplicito.

Quindi possiamo scrivere dichiarazioni come
`ITest divide = (@ATest var x, final var y) -> x / y;`
mentre prima di Java 11 queste dichiarazioni erano illegali.

Un'annotation è un elemento del linguaggio di programmazione utilizzato per fornire metadati in un'applicazione Java. Le annotazioni non possono cambiare direttamente il comportamento del codice, in quanto esse sono dati sui dati.
È tuttavia possibile, mediante la *reflection*, prendere decisioni nel codice in base ai metadati memorizzati in un'annotazione runtime. Sono, per lo più, meccanismi del codice sorgente. In genere, le annotazioni vengono lette dai plug-in del compilatore, da altri sviluppatori che leggono il codice e dagli strumenti utilizzati nei test.

Per esempio:

- `@Override (java.lang.Override)` è una comune annotazione marcatore utilizzata per consentire ad altri sviluppatori che lavorano sullo stesso codice di essere consapevoli del fatto che è stato fatto l'override di un metodo.
- `@Test (org.junit.Test)` è invece utilizzata per indicare al framework di testing **JUnit**, che un metodo segnato con tale annotation deve essere eseguito come test case.

Si noti che, ora, l'utilizzo delle annotazioni ci consente di cogliere i seguenti vantaggi:

- diminuire la verbosità dovuta all'inferenza di tipo;
- utilizzare i modificatori su variabili di tipo inferito.

Precedentemente a Java 11 avremmo potuto scegliere solo l'una o l'altra opzione, mentre ora possiamo averle entrambe. Se avessimo voluto usare i modificatori sui parametri delle lambda, avremmo dovuto utilizzare la sintassi esplicita.

Immaginate di dover scegliere tra le seguenti dichiarazioni:

```
// Pre Java 11
ITest op = (@ATest ClassName x, final ClassName y,
                        final ClassName z) -> ....;
```

Chiaramente la nuova scrittura

```
// Post Java 11
ITest op = (@ATest var x, final y, final z) -> .... ;
```

è molto più leggibile.

Dopo aver analizzato le funzionalità di var estesa ai parametri delle lambda expressions, passiamo ad analizzare, attraverso degli esempi, quali sono i limiti che questo strumento porta con sé.

È illegale mescolare stili espliciti e impliciti. I parametri formali della lambda expressions devono essere tutti impliciti o tutti espliciti.
Esempio:
`Itest subtract = (var x, double y) -> x-y; // Compilation error`

Analogamente è illegale mescolare due tipi impliciti, con e senza var, per i parametri formali della lambda expression. I parametri della lambda implicitamente tipizzati dovrebbero assumere una o l'altra sintassi, ma non entrambe. Esempio:

```
Itest subtract = (var x, y) -> x-y; // Compilation error
```

var è trattato come se fosse tipizzato esplicitamente. È quindi necessario includere una parentesi che racchiuda un singolo argomento. Esempio:

- Illegale
  ```
  ITest2 upper = var x -> { return x.toUpperCase(); };
  ```
- Legale
  ```
  ITest2 upper = x -> { return x.toUpperCase(); };
  ```
- Legale
  ```
  ITest2 upper = (var x) -> { return x.toUpperCase(); };
  ```

2.4 LINEE GUIDA STILISTICHE PER LA LOCAL-VARIABLE TYPE INFERENCE

In questa sezione vogliamo discutere come la caratteristica del linguaggio detta Local-Variable Type Inference possa influenzare anche il design del codice. Per comprendere al meglio l'uso di var, ci ispiriamo alle linee guida sullo stile [6] fornite da Stuart Marks, sviluppatore Java/JDK/OpenJDK di Oracle.
Le dichiarazioni delle variabili locali non esistono in modo isolato; il codice circostante può influire o addirittura sopraffare gli effetti dell'uso di var.
L'obiettivo di questo paragrafo è esaminare l'impatto che il codice circostante ha sulle dichiarazioni var, spiegare alcuni dei compromessi e fornire linee guida per un uso efficace della funzionalità.

G1. Scegliere nomi di variabile che forniscano informazioni utili

Questa pratica è molto buona e diffusa, ma nel contesto di var diventa ancor più importante. In una dichiarazione var, le informazioni sul significato e l'uso della variabile possono essere trasmesse attraverso il

nome di quest'ultima, perciò sostituire un tipo esplicito con var dovrebbe sempre essere accompagnato dal miglioramento del nome della variabile.

Per esempio:

```
// ORIGINAL
List<Customer> x = dbconn.executeQuery(query);

// GOOD
var custList = dbconn.executeQuery(query);
```

Questo esempio riassume perfettamente il concetto, perché un nome di variabile pressoché inutile è stato sostituito con un nome che è evocativo del tipo di variabile, ora implicito nella dichiarazione var. Questo metodo di codifica del tipo di variabile nel suo nome è chiamato "**Notazione ungara**" (o Notazione ungherese, cioè una convenzione di denominazione in cui il nome dell'oggetto indica il suo tipo e il suo scopo d'uso).
In questo esempio il nome `custList` implica che viene restituita una `List`, che potrebbe non essere significativo.

Invece del tipo esatto, a volte è preferibile, per il nome di una variabile, esprimerne il ruolo o la natura di essa, come ad esempio `customers`:

```
// ORIGINAL
try (Stream<Customer> result = dbconn.executeQuery(query)) {
    return result.map(...)
                 .filter(...)
                 .findAny();
}

// GOOD
try (var customers = dbconn.executeQuery(query)) {
    return customers.map(...)
                    .filter(...)
                    .findAny();
}
```

G2. Minimizzare l'ambito delle variabili locali

Limitare l'ambito delle variabili locali è buona pratica in generale ed è descritta in *Effective Java (3rd edition), Articolo 57*. Si applica moltissimo

con l'uso di var. "Riducendo al minimo l'ambito delle variabili locali, si aumenta la leggibilità e la manutenibilità del codice e si riduce la probabilità di errore". [7]

Nell'esempio seguente, il metodo `add` aggiunge l'elemento `MUST_BE_PROCESSED_LAST` come ultimo elemento della lista, quindi viene elaborato per ultimo.

```
var items = new ArrayList<Item>(...);
items.add(MUST_BE_PROCESSED_LAST);
for (var item : items) ...
```

Supponiamo ora che per rimuovere gli elementi duplicati, si debba modificare questo codice per utilizzare un `HashSet` invece di un `ArrayList`:

```
var items = new HashSet<Item>(...);
items.add(MUST_BE_PROCESSED_LAST);
for (var item : items) ...
```

Dal momento che gli insiemi non hanno un ordine di iterazione definito, questo codice ora ha un bug. Tuttavia, è molto probabile che il programmatore corregga questo bug velocemente, poiché gli usi della variabile `items` sono nelle righe adiacenti alla sua dichiarazione.

Supponiamo ora che questo codice sia parte di un metodo di grandi dimensioni, con un più ampio ambito per la variabile `items`:

```
var items = new HashSet<Item>(...);
// ... 100 lines of code ...
items.add(MUST_BE_PROCESSED_LAST);
for (var item : items) ...
```

Essendo `items` usato molto lontano dalla sua dichiarazione è più difficile capire che stiamo utilizzando un `HashSet` piuttosto che un `ArrayList` e quindi questo semplice problema potrebbe essere meno facile da risolvere.

Dichiarando `items` esplicitamente come `List<String>`, la modifica dell'inizializzatore richiederebbe anche la modifica del tipo in `Set<String>`. Ciò potrebbe richiedere al programmatore di ispezionare il resto del metodo alla ricerca di codice che potrebbe essere influenzato da tale modifica (ma potrebbe anche non esserlo). L'uso di var rimuoverebbe ovviamente questo compito, aumentando però il rischio che accada una situazione come quella appena descritta.

A primo impatto, questa potrebbe sembrare un'argomentazione contro l'uso di var, ma in realtà non lo è. L'esempio iniziale che mostra l'utilizzo di var è perfettamente corretto, ma il problema si verifica quando l'ambito della variabile è ampio.
Invece di evitare semplicemente di usare var in questi casi, è più indicato modificare il codice per ridurre l'ambito delle variabili locali e solo successivamente dichiararle con var.

G3. Considerare var quando l'inizializzatore fornisce informazioni sufficienti al lettore

Le variabili locali sono spesso inizializzate con costruttori.
Il nome della classe che stiamo costruendo viene spesso ripetuto come il tipo esplicito sul lato sinistro della dichiarazione. Se il nome del tipo è lungo, l'uso di var fornisce la concisione senza perdita di informazioni, quindi una riduzione della verbosità nel codice:

```
// ORIGINAL
ByteArrayOutputStream outputStream =
                        new ByteArrayOutputStream();
// GOOD
var outputStream = new ByteArrayOutputStream();
```

Altro caso in cui utilizzare var è quando l'inizializzatore è una chiamata ad un metodo anziché un costruttore, ad esempio un metodo factory statico, e quando il suo nome contiene abbastanza informazioni sul tipo:

```
// ORIGINAL
BufferedReader reader = Files.newBufferedReader(...);
List<String> stringList = List.of("a", "b", "c");

// GOOD
var reader = Files.newBufferedReader(...);
var stringList = List.of("a", "b", "c");
```

In questi casi, i nomi dei metodi implicano fortemente un particolare tipo di ritorno, che viene quindi utilizzato per inferire il tipo di variabile.

G4. Utilizzare var per suddividere espressioni concatenate o nidificate con variabili locali

Prendiamo per esempio un frammento di codice che prende una raccolta di stringhe e trova la stringa con il maggior numero di occorrenze. Come il seguente:

```
return strings.stream()
              .collect(groupingBy(s -> s, counting()))
              .entrySet()
              .stream() // secondo stream nidificato
              .max(Map.Entry.comparingByValue())
              .map(Map.Entry::getKey);
```

Questo codice è corretto ma confuso, in quanto sembra una pipeline di `stream`.
Si tratta invece di un breve `stream` seguito da un secondo `stream` sul risultato del primo, seguito da una mappatura del risultato `Optional` del secondo `stream`.

Ci sarebbe un modo più leggibile per esprimere questo codice e sarebbe come due o tre *statements*; per prima cosa dobbiamo raggruppare le voci in una `map`, quindi ridurre su quella `map` ed infine estrarre la chiave dal risultato (se presente), come mostrato di seguito:

```
Map<String, Long> freqMap =
     strings.stream()
            .collect(groupingBy(s -> s, counting()));

Optional<Map.Entry<String, Long>> maxEntryOpt =
                freqMap.entrySet()
                       .stream()
                       .max(Map.Entry.comparingByValue());
return maxEntryOpt.map(Map.Entry::getKey);
```

Il motivo per cui abbiamo un'unica dichiarazione risiede nel fatto che scrivere i tipi delle variabili intermedie può sembrare troppo oneroso, ma così facendo è stato distorto il flusso di controllo dell'esecuzione.

Vediamo un modo alternativo per scrivere il precedente frammento di codice:

```
var freqMap = strings.stream()
                     .collect(groupingBy(s -> s, counting()));

var maxEntryOpt = freqMap.entrySet()
                         .stream()
                         .max(Map.Entry.comparingByValue());
return maxEntryOpt.map(Map.Entry::getKey);
```

Utilizzare var ci consente di esprimere il codice in modo più naturale senza dover dichiarare esplicitamente i tipi delle variabili intermedie, cosa che volevamo evitare nel primo esempio.

Si potrebbe legittimamente preferire il primo pezzo di codice con la sua unica lunga catena di chiamate al metodo, anche se in alcuni casi è meglio suddividere le catene di metodi lunghe per rendere più leggibile ciò che stiamo facendo.
In questi casi l'utilizzo di var è un buon approccio, mentre l'utilizzo di dichiarazioni complete delle variabili intermedie, come nel secondo frammento di codice, aumenta la verbosità del codice.
Come in molte altre situazioni, l'uso corretto di var potrebbe implicare sia il prelievo di qualcosa (tipi espliciti) sia l'aggiunta di qualcosa (migliori nomi di variabili e una migliore strutturazione del codice).

G5. Non preoccuparsi troppo della "programmazione all'interfaccia" con le variabili locali

È uso comune, nella programmazione Java, costruire un'istanza di un tipo concreto e assegnarlo a una variabile di un tipo che è interfaccia.
Questa è una buona pratica perché vincola il codice all'astrazione anziché all'implementazione e ciò preserva la flessibilità per una manutenzione futura del codice. Per esempio:

```
// ORIGINAL
List<String> list = new ArrayList<>();
```

Utilizzando var, tuttavia, il tipo concreto viene inferito al posto dell'interfaccia:

```
// Inferred type of list is ArrayList<String>.
var list = new ArrayList<String>();
```

Dobbiamo comunque ribadire qui che var può essere utilizzato solo per variabili locali. Non può essere utilizzato per inferire il tipo dei campi di una classe, il tipo dei parametri di un metodo o il tipo di ritorno di un metodo. Il principio della "programmazione orientata all'interfaccia" è ancora più importante che mai in quei contesti.

Il problema principale che può verificarsi è che il codice che utilizza la variabile possa formare dipendenze dall'implementazione concreta. Se l'inizializzatore della variabile dovesse cambiare, questo potrebbe causare la modifica del tipo inferito, causando errori in parti di codice che utilizzano la variabile.
Come raccomandato nella linea guida **G2**, conviene che l'ambito della variabile locale sia piccolo per limitare i rischi di "fuoriuscita" dell'implementazione concreta, i quali possono influire sul codice.
In questo caso particolare, `ArrayList` contiene solo un paio di metodi non presenti nella classe `List`, ovvero `ensureCapacity` e `trimToSize`.

Tali metodi non influiscono sul contenuto della lista, quindi chiamate a questi metodi non influiscono sulla correttezza del programma. Ciò riduce ulteriormente l'impatto del tipo inferito essendo un'implementazione concreta piuttosto che un'interfaccia.

G6. Fare attenzione nell'utilizzo di var con l'operatore diamond o con metodi generic

Sia var che la funzione "diamond" consentono di omettere informazioni di tipo esplicito quando possono essere derivate da informazioni già presenti.
La domanda che ci poniamo è: si possono usare entrambi nella stessa dichiarazione? Consideriamo quanto segue:

```
PriorityQueue<Item> itemQueue = new PriorityQueue<Item>();
```

Questo può essere riscritto usando sia l'operatore diamond sia var, senza perdere le informazioni sul tipo:

```
// OK: both declare variables of type PriorityQueue<Item>
PriorityQueue<Item> itemQueue = new PriorityQueue<>();
var itemQueue = new PriorityQueue<Item>();
```

è legale utilizzare entrambe, ma il tipo inferito cambierà:

```
// DANGEROUS: infers as PriorityQueue<Object>
var itemQueue = new PriorityQueue<>();
```

Per inferire il tipo, l'operatore diamond può usare il target type (tipicamente, il lato sinistro di una dichiarazione) o i tipi degli argomenti del costruttore.
Se nessuno dei due è presente, allora ricade nel tipo più ampio applicabile, che solitamente è `Object`. Ma sicuramente non è ciò che il programmatore vuole.

L'inferenza per i metodi generic si basa sul target type se non ci sono parametri attuali del metodo che forniscono sufficienti informazioni di tipo. In una dichiarazione var, non esiste un target type, perciò accade lo stesso che con l'operatore diamond. Per esempio,

```
// DANGEROUS: infers as List<Object>
var list = List.of();
```

Con l'operatore diamond e i metodi generic, ulteriori informazioni sul tipo possono essere fornite da parametri attuali al costruttore o al metodo, permettendo di inferire il tipo previsto. Così come segue:

```
// OK: itemQueue infers as PriorityQueue<String>
Comparator<String> comp = ... ;
var itemQueue = new PriorityQueue<>(comp);

// OK: infers as List<BigInteger>
var list = List.of(BigInteger.ZERO);
```

Se si decide di utilizzare var con l'operatore diamond o un metodo generic, è necessario assicurarsi che gli argomenti del metodo o del costruttore forniscano abbastanza informazioni sul tipo in modo che il tipo inferito corrisponda a ciò di cui abbiamo bisogno. Altrimenti, è meglio evitare di utilizzarli nella stessa dichiarazione.

G7. Fare attenzione all'utilizzo di var con i letterali

I letterali primitivi possono essere usati come inizializzatori per le dichiarazioni var ma in questi casi non fornisce molti vantaggi, in quanto i nomi dei tipi sono solitamente brevi. Non c'è alcun problema con letterali di tipo `boolean`, `char`, `long` e `String`. Il tipo inferito da questi letterali è definito, quindi il significato di var non è ambiguo:

```
// ORIGINAL
boolean ready = true;
```

```
char ch = '\ufffd';
long sum = 0L;
String label = "wombat";

// GOOD
var ready = true;
var ch = '\ufffd';
var sum = 0L;
var label = "wombat";
```

Quando l'inizializzatore è un valore numerico, in particolare un letterale intero, è necessario prestare particolare attenzione. Con un tipo esplicito sul lato sinistro, il valore numerico può essere ampliato o ristretto a tipi diversi da `int`. Con var, il valore sarà dedotto come `int`, che potrebbe non essere nelle nostre intenzioni.

```
// ORIGINAL
byte flags = 0;
short mask = 0x7fff;
long base = 17;

// DANGEROUS: all infer as int
var flags = 0;
var mask = 0x7fff;
var base = 17;
```

I letterali in virgola mobile invece sono inequivocabili:

```
// ORIGINAL
float f = 1.0f;
double d = 2.0;

// GOOD
var f = 1.0f;
var d = 2.0;
```

Si noti che i letterali `float` possono essere ampliati a `double`. È insensato inizializzare una variabile `double` usando un letterale `float` esplicito come ad esempio `3.0f`, tuttavia, possono verificarsi casi in cui una variabile `double` viene inizializzata da un campo `float`. In questi casi è consigliato prestare attenzione:

```
// ORIGINAL
static final float INITIAL = 3.0f;
...
double temp = INITIAL;

// DANGEROUS: now infers as float
var temp = INITIAL;
```

In ogni caso, questo esempio viola la linea guida **G3**, perché non c'è abbastanza informazione nell'inizializzatore affinché un lettore possa vedere il tipo inferito.

2.4.1 *Esempi*

Concludiamo con alcuni esempi, in cui l'uso di var è indirizzato ad ottenere il massimo beneficio sulla qualità del codice.

Il seguente codice rimuove al massimo `max` voci corrispondenti da un oggetto di tipo `Map`. I *bounds* del tipo con *wildcard* vengono utilizzati per migliorare la flessibilità del metodo, determinando una notevole verbosità. Sfortunatamente, ciò richiede che il tipo di `Iterator` sia un wildcard nidificato, rendendo la sua dichiarazione più verbosa.
Questa dichiarazione è così lunga che l'intestazione del ciclo `for` non si adatta più a una singola riga, rendendo il codice ancora più difficile da leggere.

```
// ORIGINAL
void removeMatches(Map<? extends String, ? extends Number> map, int
    max) {
   for (Iterator<? extends Map.Entry<? extends String, ? extends
       Number>> iterator =
           map.entrySet().iterator(); iterator.hasNext();) {
      Map.Entry<? extends String, ? extends Number> entry =
          iterator.next();
      if (max > 0 && matches(entry)) {
         iterator.remove();
         max--;
      }
   }
}
```

L'uso di var in questo caso rimuove le lunghe dichiarazioni di tipo per le variabili locali. Avere tipi espliciti per `Iterator` e `Map.Entry` locali in questo ciclo sono in gran parte inutili. Ciò consente anche al controllo del ciclo di adattarsi su una singola riga, migliorando ulteriormente la leggibilità.

```
// GOOD
void removeMatches(Map<? extends String, ? extends Number> map, int
    max) {
   for (var iterator = map.entrySet().iterator();
       iterator.hasNext();) {
      var entry = iterator.next();
      if (max > 0 && matches(entry)) {
         iterator.remove();
         max--;
      }
   }
}
```

Dopo aver mostrato qualche caso d'uso, cerchiamo di trarre alcune conclusioni sulla novità appena descritta.

2.5 CONCLUSIONI

Abbiamo visto che l'inferenza di tipo, che viene introdotta in Java 10 con var, è una forma di inferenza molto interessante in termini di produttività e leggibilità del codice, ma anche limitata dalle condizioni in cui il typechecking di Java lavora sulle espressioni.

Sintetizzando, possiamo dire che l'approccio molto semplice garantisce che eventuali errori di compilazione, relativi alle dichiarazioni var, siano limitati a una singola istruzione, poiché l'algoritmo di inferenza var esamina solo l'espressione assegnata alla variabile per dedurne il tipo. Quindi l'inferenza di tipo è lecita solo quando è supportata da un typechecker che ha successo sull'espressione assegnata alla variabile.

Ci sono diverse correnti di pensiero su questa nuova funzionalità. Alcuni accolgono con favore la concisione che essa permette, mentre altri temono che privi i lettori di importanti informazioni sul tipo, compromettendo la leggibilità del codice.
Entrambe le correnti di pensiero hanno validi motivi: l'uso di questa fun-

zionalità può rendere il codice più leggibile, eliminando le informazioni ridondanti, ma può anche rendere il codice meno leggibile, eliminando informazioni utili. Altri sviluppatori si preoccupano del fatto che sarà sovrautilizzato, con conseguente scrittura di peggior codice Java. Anche questo è vero, ma è anche probabile che porti a scrivere buon codice.
Come tutte le caratteristiche del linguaggio, anche questo strumento deve essere usato secondo precise linee guida, come ad esempio quelle riportate nel paragrafo precedente.

In ogni caso, la nostra conclusione è che esso rappresenti un vero e proprio potenziamento effettivo della parte di inferenza nel controllo dei tipi. A sostegno di questo, riprendiamo l'esempio considerato precedentemente: nell'espressione `var test = (true ? new A() : new B());`
i tipi `A` e `B` hanno come supertipi comuni interfacce differenti, ma non un minimo. Essendo `I1` e `I2` le due interfacce comuni allo stesso livello, allora viene inferito il tipo intersezione `I1&I2`, che non è neanche un tipo nominale ma è costruito completamente dalla type inference.
Ebbene, questo tipo è assegnato alla variabile `test`, su cui saranno invocabili sia i metodi dichiarati nell'interfaccia I1 che i metodi dichiarati nell'interfaccia `I2`.

Crediamo, pertanto, che l'evoluzione generale di Java sarà proprio in un sempre maggiore potenziamento della parte di inferenza di tipo rispetto al semplice typechecking. Questo significherà aumentare l'inferenza dei tipi intersezione per dare tipo a diverse espressioni, incluse probabilmente le lambda expressions per le quali attualmente il tipo intersezione è assegnabile solo con un cast esplicito.
Come conseguenza, anche l'uso di **var** sarà esteso. In tal modo, l'inferenza di tipo non sarà solo un modo per diminuire la verbosità del codice ma diventerà sempre di più uno strumento di maggiore espressività.

3

PROGRAMMAZIONE MODULARE: JAVA 9

L'introduzione della programmazione modulare in Java 9 è considerata da molti la novità finora più significativa nell'evoluzione del linguaggio. Un obiettivo di Java è fin da sempre quello di permettere e incoraggiare la scrittura di codice facilmente mantenibile e, soprattutto, riusabile. Tale approccio è ispirato al **Single Responsibility Principle (SRP)**, principio fondamentale nella programmazione orientata agli oggetti, che prescrive di *identificare chiaramente la singola responsabilità* assegnata a ciascun oggetto o componente del sistema.
Una classe, in generale una porzione di codice, deve avere una, e solo una, ragione per essere cambiata. [8]

Uno degli strumenti fondamentali per realizzare questo principio è proprio quello di dividere le responsabilità del programma in più parti, che siano queste metodi, classi o moduli.
La modularità è un modo di scrivere e implementare un programma come un numero di moduli unici. Esso evita il design monolitico e aiuta a ridurre la complessità di un sistema, minimizzando l'accoppiamento fra componenti.
In poche parole, la modularità è un principio di progettazione che ci aiuta a raggiungere:

- disaccoppiamento tra componenti;
- contratti chiari e dipendenze tra componenti;
- implementazione nascosta, usando un forte incapsulamento.

In questo capitolo analizziamo il concetto di modularità, introdotto da Java 9 come strumento nativo del linguaggio, osservando il modo in cui i

moduli cambiano la scrittura del codice ma anche il loro utilizzo radicato nell'intera struttura del JDK (Java Development Kit). Un modulo può essere inteso come un'aggregazione di packages Java (e di risorse statiche di natura diversa come immagini, file XML, ecc.).
Ogni modulo ha un nome univoco, può dichiarare dipendenze da altri moduli, deve dichiarare esplicitamente quali packages sono resi disponibili ad altri moduli, i servizi offerti e quelli consumati.

Per trattare questo argomento, consideriamo i seguenti aspetti: concetto di modulo precedente a Java 9 (par. 1), progetto Jigsaw (par. 2), Java Platform Modules System (par. 3), differenze tra moduli Java e Maven (par. 4), vantaggi della modularità (par. 5), applicazione dei moduli al JDK (par. 6), interviste ad esperti (par. 7) ed infine alcune riflessioni conclusive (par. 8).

3.1 CONCETTO DI MODULO PRIMA DI JAVA 9

Prima di Java 9, i principali strumenti che permettevano la modularità (anche se in due concetti diversi fra loro) erano *Maven* (sviluppato dalla Apache Software Foundation) e *OSGi* (sviluppato da OSGi Alliance).

Apache Maven è uno strumento di gestione di progetti che permette la divisione di un programma in moduli, chiamati appunto moduli Maven; usa un costrutto conosciuto come *Project Object Model (POM)*, cioè un file XML che descrive le dipendenze fra il progetto e le varie versioni di librerie necessarie nonché le dipendenze fra di esse. In questo modo si separano le librerie dalla directory di progetto utilizzando questo file descrittivo per definirne le relazioni.
Maven effettua automaticamente il download di librerie Java e plug-in Maven dai vari repository definiti scaricandoli in locale o in un repository centralizzato lato sviluppo (ad esempio Git Repository). Questo permette di recuperare in modo uniforme i vari file JAR e di poter spostare il progetto indipendentemente da un ambiente all'altro avendo la sicurezza di utilizzare sempre le stesse versioni delle librerie.

OSGi (Open Service Gateway initiative) è un framework che permette di costruire applicazioni modulari a componenti (i Bundle) e che introduce una programmazione Service Oriented, permettendo una separazione tra interfaccia ed implementazione molto più rigorosa di quella nativa Java.

Nel mondo Java, specialmente con OSGi, i JAR (Java Archive) erano considerati l'unità di modularità. I JAR hanno aiutato a raggruppare insieme i componenti correlati, ma hanno alcune limitazioni:

- contratti espliciti e dipendenze tra JAR;
- debole incapsulamento di elementi all'interno dei JAR.

Un altro problema con i JAR risiede nel *JAR Hell*. Più versioni dei JAR che si trovano sul *classpath*, portano il *ClassLoader* a caricare la prima classe trovata dal JAR, con risultati molto inaspettati. L'altro problema con JVM che utilizzava classpath era che la compilazione dell'applicazione avrebbe avuto successo, ma l'applicazione non avrebbe funzionato a runtime lanciando l'eccezione `ClassNotFoundException`, a causa dei JAR mancanti sul classpath in fase di esecuzione.

Viste tutte le limitazioni suddette nell'utilizzo di JAR come unità di modularità, è stato introdotto il concetto di **Java Module** e, insieme con esso, un nuovo sistema modulare progettato per Java, ovvero il **Project Jigsaw**.

3.2 PROJECT JIGSAW

Le motivazioni principali che hanno portato allo sviluppo di questo progetto sono:

- creare un sistema di moduli per il linguaggio - *implementato in JEP 261*;
- applicare tale sistema al sorgente JDK - *implementato in JEP 201*;
- modularizzare le librerie JDK - *implementate in JEP 200*;
- aggiornare il JRE (Java Runtime Environment) per supportare la modularità - *implementato in JEP 220*;
- essere in grado di creare un JRE più piccolo con un sottoinsieme di moduli dal JDK - *implementato in JEP 282*.

Un'altra importante caratteristica è quella di incapsulare le API interne nel JDK, quelle sotto i pacchetti `sun.*` e altre API non standard. Queste API non sono mai state pensate per essere utilizzate dal pubblico e

non sono mai state programmate per essere mantenute. Ma la potenza di queste API ha fatto sì che gli sviluppatori Java le sfruttassero nello sviluppo di librerie, framework e strumenti diversi. Sono state introdotte sostituzioni per poche API interne e le altre sono state spostate in moduli interni.

Inoltre sono stati rilasciati nuovi strumenti per la modularità:

- **jdeps** - aiuta ad analizzare il codebase (ovvero l'intera collezione di codice sorgente usata per costruire una particolare applicazione o un particolare componente) per identificare le dipendenze sulle API JDK e sui JAR di terze parti. Indica anche il nome del modulo in cui è possibile trovare l'API JDK. Ciò semplifica la modularizzazione del codebase;
- **jdeprscan** - aiuta ad analizzare il codebase per l'utilizzo di qualsiasi API deprecata;
- **jlink** - aiuta a creare un JRE più piccolo combinando i moduli dell'applicazione e di JDK;
- **jmod** - aiuta a lavorare con i file jmod; esso è un nuovo formato per il packaging dei moduli. Questo formato consente di includere codice nativo, file di configurazione e altri dati che non si adattano ai file JAR.

Componente fondamentale del Project Jigsaw è il *Java Platform Modules System (JPMS)*.

3.3 JAVA PLATFORM MODULES SYSTEM

Il **JPMS** è una novità introdotta a partire da Java 9, presentata in dettaglio in **JSR 376** (Java Specification Request) e creata da un gruppo di esperti del linguaggio Java.
Sua base è il concetto di modulo con cui dar vita ad applicazioni modulari, che offre benefici in termini di produttività e manutenibilità.
La modularità utilizza il concetto base di package aggiungendo un più alto livello di aggregazione basato sul loro uso.
Un modulo è un insieme di codice e dati auto-descrittivo. Il suo codice è organizzato come un'aggregazione di package relazionati contenenti tipi, cioè classi e interfacce Java; i suoi dati includono risorse statiche e altri

tipi di informazioni.

Ogni modulo è dotato di un descrittore, denominato `module-info.java`, che specifica:

- nome del modulo;
- moduli dai quali dipende (dipendenze);
- package resi esplicitamente disponibili (gli altri packages nel modulo sono implicitamente non disponibili ad altri moduli);
- servizi che offre e servizi che consuma;
- a quali altri moduli rende possibile la reflection.

Con l'introduzione di questa novità, Java sta lentamente eliminando il concetto di *classpath* e invece introduce il concetto di **module path**, cioè il percorso in cui i moduli possono essere trovati.

Lo stesso JDK è stato diviso in un insieme di moduli, come sopra già citato.

Cerchiamo però di capire quali sono le keywords da utilizzare con questo strumento e qual è il loro funzionamento.

3.3.1 *Funzionamento e Keywords*

Il descrittore risiede nel file `module-info` che contiene la dichiarazione del modulo.

- Ogni dichiarazione inizia con la keyword `module`, seguita dal nome del modulo e da parentesi graffe.

```
module com.foo.bar { }
```

Nel corpo del modulo possiamo inserire direttive per agire sugli aspetti prima elencati.

- Con `requires`, seguita dal nome di un modulo, specifichiamo le sue dipendenze. Essa consente di utilizzare la keyword `static` per indicare che la dipendenza è richiesta in compilazione e non a runtime.

```
module com.foo.bar {
    requires org.baz.qux;
}
```

Un aspetto molto interessante legato a tale direttiva è la *implied readability* (o dipendenza transitiva).

- Possiamo specificarla facendo seguire a `requires` la keyword `transitive` seguita dal nome di un modulo. Con essa intendiamo dire che un modulo `A` che richieda l'uso del modulo `B` diventa dipendente anche dal modulo `transitive C` dichiarato in `B`.

Cosa significa però "implied readability"?
Quando un modulo dipende direttamente da un altro nel grafico dei moduli, il codice nel primo modulo sarà in grado di riferirsi ai tipi nel secondo modulo. Diciamo quindi che il primo modulo legge il secondo o, equivalentemente, che il secondo modulo è leggibile dal primo. [9] Estendiamo quindi le dichiarazioni dei moduli in modo che un modulo possa garantire la leggibilità a moduli aggiuntivi, da cui dipende, ed a qualsiasi modulo che dipende da esso. Tale leggibilità implicita è espressa includendo il modificatore `public` in una clausola `requires`. [10]

In un modulo, in genere, non si vuole rendere pubblici tutti i package, ci possono essere infatti package destinati solo ad uso interno.

- La keyword `exports` permette di specificare quali package sono visibili ad altri moduli. Quando un package di `A` è visibile ad un altro modulo `B`, il codice di `B` può accedere a quanto dichiarato `public` nel package di `A`.
- La keyword `exports...to` permette invece di specificare la visibilità di un modulo a un altro specifico modulo.

```
module com.foo.bar {
    requires org.baz.qux;
    exports com.foo.bar.alpha;
    exports com.foo.bar.beta;
}
```

Se la dichiarazione di un modulo non contiene clausole `exports`, non esporterà alcun tipo a nessun altro modulo.

Le direttive `opens`, `opens...to` ed `open` sono utili invece per l'accessibilità dei package verso altri moduli.

- Utilizzando `opens`, seguita dal nome di un package del modulo, stiamo dicendo che le classi `public` (ed i loro campi e metodi `public` e `protected`) all'interno del package sono accessibili in altri moduli soltanto a runtime, consentendo l'accesso con reflection a tutti i tipi all'interno delle classi del package.

- `opens...to` consente di specificare i moduli per i quali il package è accessibile. In un modulo potremmo dichiarare qualcosa come
  `opens mypackage to module1, module2`
  per indicare che il package è accessibile con le modalità indicate da open ai soli `module1` e `module2`.

- Se invece tutti i package di un modulo devono essere accessibili a runtime e via reflection agli altri moduli, è possibile aprire totalmente un modulo con `open` a livello di dichiarazione di modulo.

- Un modulo può utilizzare specifici servizi divenendo un *service consumer*, cioè un oggetto di classe che implementa l'interfaccia, o estende la classe astratta, specificata con la direttiva `uses`.

L'uso di un servizio da parte di un modulo avviene tramite reflection.

- Utilizzando la direttiva `provides...with` un modulo può diventare fornitore di servizi (*service provider*). Con essa specifichiamo un'interfaccia, o una classe astratta, nella sezione `provides`, indicando la sua implementazione nella parte `with`.

La **Reflection** è una funzionalità del linguaggio di programmazione Java. Consente a un programma Java in esecuzione di esaminare o "introspettarsi" su se stesso e manipolare le proprietà interne del programma. Ad esempio, è possibile per una classe Java ottenere i nomi di tutti i suoi membri e visualizzarli. [11]
Java ha supportato a lungo i servizi tramite la classe `java.util.ServiceLoader`, che individua i provider di servizi in fase di esecuzione cercando nel classpath. Per i service provider definiti nei moduli dobbiamo considerare come individuare tali moduli tra l'insieme di moduli osservabili, risolvere le loro dipendenze e renderli disponibili al codice che utilizza i servizi corrispondenti.

In Java, come abbiamo anticipato prima, esistono diversi tipi di moduli, ognuno con il proprio scopo. Quali sono però le differenze principali?

3.4 DIFFERENZE TRA CONCETTO DI MODULO MAVEN E JAVA

I moduli Maven sono uno strumento per organizzare il progetto in più sottoprogetti (moduli). Con Maven, si possono controllare le versioni di questi moduli e le dipendenze tra essi. Ogni modulo produrrà un *artifact*.
I moduli Java sono un modo per incapsulare fortemente le proprie classi, ma non forniscono alcun mezzo per controllare la versione di un artifact che si sta utilizzando.
Quindi, i moduli Maven e i moduli Java hanno uno scopo diverso, nel senso che viene usato ambiguamente lo stesso termine "modulo" per rappresentare concetti diversi; per aumentare l'ambiguità, anche IntelliJ utilizza il termine modulo per raggruppare i file.

Essi possono però essere utilizzati congiuntamente, come vedremo nell'esempio seguente, per fornire un maggiore controllo sull'esposizione del codice e aumentare l'encapsulation.

3.4.1 *Esempio di utilizzo dei moduli*

Durante la mia attuale esperienza lavorativa mi trovo ad utilizzare i moduli Maven per gestire un progetto che presenta una divisione in più componenti, ciascuno descritto da un POM in cui sono elencate le dipendenze necessarie al funzionamento del componente stesso.
Ho scelto così di introdurre in tale progetto anche i moduli Java e ciò è risultato in un maggiore controllo sull'esposizione del codice di ciascun modulo verso gli altri.

I moduli Java devono essere dichiarati con un "punto" come separatore (es. `vpostel.common`) e per convenienza abbiamo denominato i moduli Maven con un "tratto" come separatore (es. `vpostel-common`) così da non dover specificare quando parliamo degli uni o degli altri.
Di seguito riportiamo, a titolo esemplificativo, il contenuto dei file `module-info.java` e la definizione del modulo Maven nel `pom.xml` per ciascun modulo, nonché una breve spiegazione di quale compito svolga il componente trattato all'interno del programma.

Figura 1: **modulo Maven padre**

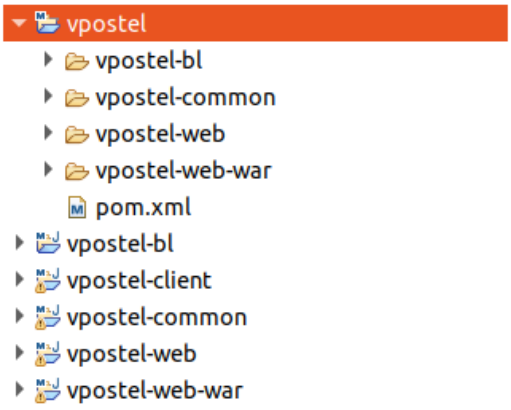
```
5   <modelVersion>4.0.0</modelVersion>
6   <groupId>TesiBertini</groupId>
7   <artifactId>vpostel</artifactId>
8   <version>0.0.1-SNAPSHOT</version>
9   <packaging>pom</packaging>
10  <name>vpostel</name>
```

Figura 2: **struttura moduli Maven**

```
226  <modules>
227      <module>vpostel-bl</module>
228      <module>vpostel-common</module>
229      <module>vpostel-web</module>
230      <module>vpostel-web-war</module>
231  </modules>
```

Come possiamo osservare da queste due prime immagini, il progetto è diviso in 4 moduli (`bl`, `common`, `web` e `web-war`) e il modulo padre è denominato `vpostel`.

Figura 3: **struttura progetto**



In questa immagine è mostrata la struttura del progetto in Eclipse ed è possibile notare anche un quinto modulo, cioè `vpostel-client`, che è però in realtà un progetto separato, utilizzato per effettuare dei test sul progetto `vpostel`.

Figura 4: **modulo Maven bl**

```
4     <modelVersion>4.0.0</modelVersion>
5     <parent>
6         <groupId>TesiBertini</groupId>
7         <artifactId>vpostel</artifactId>
8         <version>0.0.1-SNAPSHOT</version>
9     </parent>
10    <artifactId>vpostel-bl</artifactId>
```

Figura 5: **modulo Java bl**

```
1 module vpostel.bl {
2     requires vpostel.common;
3     requires spring.core;
4 }
```

Il modulo `vpostel.bl` rappresenta la logica del progetto e necessita del modulo `vpostel.common` e del modulo `spring.core`, la cui libreria è dichiarata tra le dipendenze del modulo Maven `vpostel-common`.

Figura 6: **modulo Maven common**

```
4     <modelVersion>4.0.0</modelVersion>
5     <parent>
6         <groupId>TesiBertini</groupId>
7         <artifactId>vpostel</artifactId>
8         <version>0.0.1-SNAPSHOT</version>
9     </parent>
10    <artifactId>vpostel-common</artifactId>
```

Figura 7: **modulo Java common**

```
1  import org.apache.commons.codec.binary.Hex;
2
3  module vpostel.common {
4      requires transitive spring.beans;
5      requires transitive spring.web;
6      requires transitive spring.context;
7      requires transitive slf4j.api;
8      requires commons.lang3;
9      requires org.apache.commons.codec;
10
11     exports vpostel.common.intf.orchestrator;
12     exports vpostel.common.request;
13     exports vpostel.common.response;
14     exports vpostel.common.util;
15 }
```

Come si può chiaramente notare da questa immagine, il modulo `vpostel.common` è quello più corposo, perché contiene elementi utili al funzionamento dell'intero progetto; questo è il motivo per cui abbiamo inserito la keyword `exports` per esportare i pacchetti indicati all'esterno del componente, i quali sarebbero altrimenti non visibili dagli altri componenti.
Con `requires` abbiamo invece dichiarato i moduli necessari solo per questo componente. Abbiamo inoltre aggiunto la keyword `transitive` ad essa per evitare ripetizioni nei moduli che dipendono dal modulo `vpostel.common`.

Figura 8: **modulo Maven web**

```
6     <modelVersion>4.0.0</modelVersion>
7     <parent>
8         <groupId>TesiBertini</groupId>
9         <artifactId>vpostel</artifactId>
10        <version>0.0.1-SNAPSHOT</version>
11    </parent>
12    <artifactId>vpostel-web</artifactId>
```

Figura 9: **modulo Java web**

```
1 module vpostel.web {
2     requires vpostel.common;
3     requires spring.webmvc;
4     requires gson;
5 }
```

Il componente `vpostel-web` si occupa della gestione delle chiamate **HTTP**, principalmente di metodi *get* e *post*.
Il modulo `vpostel.web` necessita dei moduli dichiarati con la keyword `requires`.

Figura 10: **modulo Maven web-war**

```
6     <modelVersion>4.0.0</modelVersion>
7     <parent>
8         <groupId>TesiBertini</groupId>
9         <artifactId>vpostel</artifactId>
10        <version>0.0.1-SNAPSHOT</version>
11    </parent>
12    <artifactId>vpostel-web-war</artifactId>
13    <packaging>war</packaging>
```

Figura 11: **modulo Java web-war**

```
1 module vpostel.web.war {
2     requires vpostel.web;
3     requires vpostel.bl;
4 }
```

Il modulo `vpostel-web-war` contiene le risorse statiche utilizzate nel progetto, cioè file di configurazione e pagine web.

Figura 12: **modulo Maven client**

```
4   <modelVersion>4.0.0</modelVersion>
5   <groupId>TesiBertini</groupId>
6   <artifactId>vpostel-client</artifactId>
7   <version>0.0.1-SNAPSHOT</version>
8   <packaging>war</packaging>
```

Figura 13: **modulo Java client**

```
1 module vpostel.client {
2     requires spring.web;
3     requires spring.webmvc;
4     requires spring.context;
5 }
```

Come già anticipato, il modulo `vpostel-client` rappresenta il progetto `vpostel-client`, ed è utilizzato per effettuare dei test attraverso una pagina web in cui inserire i valori in un form html.
Esso necessita di tre moduli del framework Spring fondamentali al funzionamento del progetto.

Con questo esempio pratico abbiamo dimostrato come è possibile utilizzare congiuntamente i moduli Java e i moduli Maven, proprio perché le loro funzionalità sono differenti nonostante la stessa denominazione.
L'utilità di usare i due strumenti in modo congiunto è la possibilità di scaricare librerie esterne tramite dipendenze con i moduli Maven e utilizzarle con maggiore controllo con l'uso dei moduli Java, a dimostrazione del più *forte incapsulamento* che porta in dote la programmazione modulare introdotta con Java 9.

È possibile utilizzare questo strumento anche attraverso il terminale del proprio sistema operativo, come riportiamo nel seguente esempio.
Le proprietà del modulo si trovano nel file `module-info.java`.
Per visualizzare la descrizione del modulo definito in questo file, è possibile utilizzare il seguente comando: `java --describe-module java.sql`.
Ciò produrrà il seguente output:

```
java.sql@9
exports java.sql
exports javax.sql
exports javax.transaction.xa
requires java.logging transitive
requires java.xml transitive
requires java.base mandated
uses java.sql.Driver
```

Dove si ritrovano le keyword spiegate in precedenza.

Osservando la *Javadoc* di Java 9 si può notare che ora è divisa in moduli anziché in pacchetti. All'interno della Javadoc di un modulo, si trova un grafico del modulo e i pacchetti. In realtà, la stessa struttura di cui sopra può essere trovata con il comando `--describe-module`.

Ma il modulo Java è di un unico tipo o si differenzia a seconda del suo utilizzo?

3.4.2 *Tipi differenti di modulo*

Esistono diversi tipi di moduli, categorizzati in base al ruolo che assumono all'interno della piattaforma:

- **Moduli a livello applicazione**: i moduli che creiamo per ottenere funzionalità. Tutte le dipendenze di terze parti rientrano in questa categoria.
- **Moduli automatizzati**: i JAR posizionati nel module path senza il descrittore del modulo sono chiamati moduli automatizzati. Essi esportano implicitamente tutti i pacchetti e leggono tutti gli altri moduli. Lo scopo principale di questi moduli è utilizzare i JAR creati prima di Java 9.

- **Moduli senza nome**: tutti i JAR e le classi sul classpath saranno categorizzati come moduli senza nome (questo perché il classpath non è stato completamente rimosso). Un modulo di questo tipo non ha alcun nome e quindi può leggere ed esportare tutti i moduli.
- **Moduli della piattaforma**: lo stesso JDK è stato migrato in una struttura modulare, perciò tali moduli sono chiamati moduli della piattaforma. Esempio: `java.se, java.xml.ws`

Abbiamo analizzato finora il concetto di modularità: ora vediamo i vantaggi che essa offre nello sviluppo di applicazioni in Java.

3.5 VANTAGGI DELLA MODULARITÀ

Potrebbe sembrare contraddittorio per le regole del linguaggio ma, per rimuovere i difetti nell'incapsulamento, il modificatore `public`, con l'utilizzo dei moduli non è più visibile all'intero programma, a meno che non venga esportato nel file `module-info.java`. Esso controlla le dipendenze - sia in fase di compilazione che a runtime - e anche con la reflection non possiamo accedervi quando non viene esportato.
La modularità permette di nascondere l'implementazione e migliorare sicurezza, manutenibilità e prestazioni del codice. I principali vantaggi che offre sono la diversificazione dei ruoli tra i moduli stessi, l'astrazione del modulo identificandone solo il compito che svolge e non la sua implementazione, il riutilizzo del modulo in diversi contesti e la facilità di manutenzione.

I vantaggi conseguenti all'utilizzo dei moduli Java, in termini di scrittura del codice, sono riassunti nel seguito.

Configurazione affidabile. Gli sviluppatori hanno sofferto a lungo il fragile meccanismo del classpath soggetto a errori per la configurazione dei componenti del programma. Il classpath non può esprimere relazioni tra componenti, quindi se manca un componente necessario, quest'ultimo non verrà scoperto fino a quando non verrà effettuato un tentativo di usarlo. Il classpath consente anche il caricamento di classi nello stesso pacchetto da componenti diversi, portando a comportamenti imprevedibili e errori difficili da diagnosticare. La specifica proposta consentirà a un componente di dichiarare che dipende da altri componenti, dal momento che altri componenti dipendono da esso.

Forte incapsulamento (strong encapsulation). Il meccanismo di controllo degli accessi del linguaggio di programmazione Java e della JVM non consente in alcun modo a un componente di impedire ad altri componenti di accedere ai suoi pacchetti interni. La specifica proposta consentirà a un componente di dichiarare quali dei suoi pacchetti sono accessibili da altri componenti e quali no.

Una piattaforma scalabile. Le dimensioni sempre crescenti della piattaforma Java SE hanno reso sempre più difficile l'utilizzo in dispositivi di piccole dimensioni, nonostante il fatto che molti di questi dispositivi siano in grado di eseguire una JVM di classe SE. I Compact Profiles introdotti in Java SE 8 (JSR 337) aiutano in questo senso, ma non sono abbastanza flessibili. La specifica proposta permetterà alla piattaforma Java SE, e alle sue implementazioni, di essere scomposte in un insieme di componenti che possono essere assemblati dagli sviluppatori in configurazioni personalizzate che contengono solo la funzionalità effettivamente richiesta da un'applicazione.

Maggiore integrità della piattaforma. L'uso occasionale delle API interne alle implementazioni della piattaforma Java SE rappresenta un rischio per la sicurezza e un onere di manutenzione. Il forte incapsulamento fornito dalla specifica proposta consentirà ai componenti che implementano la piattaforma Java SE di impedire l'accesso alle loro API interne.

Prestazioni migliorate. Molte tecniche *ahead-of-time* di ottimizzazione dell'intero programma possono essere più efficaci quando è noto che una classe può fare riferimento solo alle classi in pochi altri componenti specifici piuttosto che a qualsiasi classe caricata in fase di esecuzione. Le prestazioni migliorano particolarmente quando i componenti di un'applicazione possono essere ottimizzati insieme ai componenti che implementano la piattaforma Java SE.
La compilazione ahead-of-time esegue la compilazione una volta per tutte, prima dell'esecuzione, tipicamente durante l'installazione del programma. Quindi durante l'installazione del programma il codice in linguaggio intermedio viene compilato in codice binario nativo della piattaforma. Questo permette di eseguire il programma usando codice binario nativo ed evitando la compilazione durante l'esecuzione del programma; in generale ciò migliora le prestazioni e la reattività dei programmi.

Vediamo adesso quali sono i moduli appartenenti al JDK e come è cambiata la sua struttura a seguito dell'introduzione di questo strumento.

3.6 MODULE SYSTEM APPLICATO AL JDK

Ogni installazione JDK è fornita con un file src.zip. Questo archivio contiene il codebase per le API Java JDK. Se si estrae l'archivio, si troveranno più cartelle, alcune iniziano con "`java`", alcune con "`javafx`" e il resto con "`jdk`". Ogni cartella rappresenta un modulo.
I moduli che iniziano con "`java`" sono i moduli JDK, quelli che iniziano con "`javafx`" sono i moduli `JavaFX` e quelli che iniziano con "`jdk`" sono i moduli degli strumenti JDK. Tutti i moduli JDK e tutti i moduli definiti dall'utente dipendono implicitamente dal modulo `java.base`. Il modulo java.base contiene le API JDK comunemente utilizzate come `Utils`, `Collections`, `IO`, `Concurrency` tra gli altri.

Figura 14: **sorgente JDK9**

Il grafico delle dipendenze dei moduli JDK, chiamato *module graph* è:

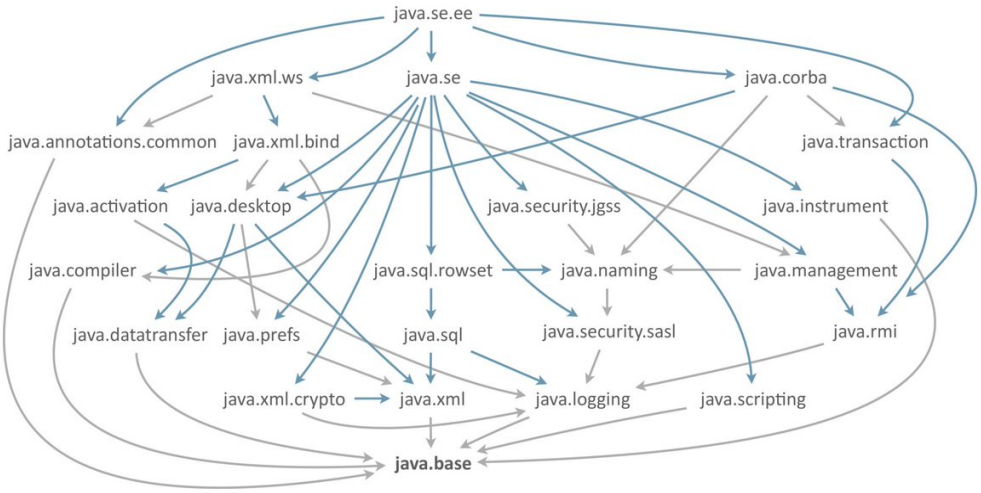


Figura 15: **grafico dei moduli JDK**

Per vedere quali moduli sono disponibili in Java, possiamo digitare il seguente comando nel terminale del sistema operativo:
`java --list-modules`.
Viene visualizzato un elenco con i moduli disponibili. Di seguito è riportato un estratto dalla lista:

```
java.activation@9
java.base@9
java.compiler@9
...
```

Dall'elenco, possiamo vedere che i moduli sono suddivisi in quattro categorie: quelli che iniziano con `java` (moduli Java standard), quelli che iniziano con `javafx` (moduli JavaFX), quelli che iniziano con `jdk` (moduli specifici di JDK) e quelli che iniziano con `oracle` (moduli specifici di Oracle). Ogni modulo termina con `@9`, a indicare che il modulo appartiene a Java 9.

3.7 I MODULI JAVA: INTERVISTE AD ESPERTI

Marcus Biel

"Vedo un enorme potenziale in Jigsaw per la creazione di codice facilmente mantenibile, riutilizzabile e ben strutturato. Tuttavia, dubito che Java 9 sarà interessante come Java 8 per gli sviluppatori". [12]

JAXenter: Project Jigsaw è finalmente arrivato. Quali sensazioni hai riguardo al sistema modulare? Inizierai con o senza moduli?

Marcus Biel: L'approccio modulare di Jigsaw è un grande passo nella giusta direzione. Ma la migrazione verso un sistema modulare farà male all'inizio. Non è facile modularizzare correttamente un progetto grande e costa molto tempo. Attualmente sto lavorando per un cliente che sta passando a Java 8 per la prima volta, quindi Jigsaw dovrà aspettare un po'!

Lukas Eder

"Java 9 non è importante quanto Java 8, è più una versione di manutenzione". [13]

JAXenter: Project Jigsaw avrebbe dovuto essere una delle caratteristiche

principali di Java 8, ma Oracle decise di rimandarlo. Invece, è diventato la caratteristica chiave di Java 9 - è valsa la pena aspettare?

Lukas Eder: Sì. Java 8 è stata una versione importante. Jigsaw avrebbe ritardato gli streams, le funzioni lambda, i metodi default, JSR-310 e molte altre belle funzionalità senza aggiungere troppo valore all'ecosistema. Non fraintendermi: Jigsaw è importante per Oracle e il futuro di JDK, ma per i consumatori non sarà importante per un po' di tempo - abbiamo già Maven e OSGi, e non migreremo immediatamente a Jigsaw. Quindi posticipare Jigsaw per aiutare Java 8 a essere rilasciato è stata sicuramente una buona decisione.

Alex Buckley

"Public su una dichiarazione di classe non significa più accessibile a tutti". [14]

Che cosa significa strong encapsulation in JDK 9? Come si esegue la migrazione da un'applicazione ad una con l'utilizzo di moduli?
Alex Buckley spiega in grande dettaglio i concetti chiave dietro i moduli. Descrive come il sistema dei moduli può migliorare la struttura del proprio codice, come è possibile migrare progressivamente l'applicazione combinando codice modulare e non modulare, e come un JDK modulare consente una migliore compatibilità.

Con JDK 9, è ancora possibile eseguire le proprie applicazioni sul classpath. Il nuovo sistema dei moduli è integrato nel linguaggio Java e nella *virtual machine*. Le proprie applicazioni e le librerie che si utilizza possono essere pacchettizzate, testate e implementate come moduli gestiti dal sistema dei moduli. Un modulo è essenzialmente un insieme di pacchetti che hanno senso essere raggruppati ed è progettato per il riutilizzo. Poiché la completa piattaforma Java è modulare, il sistema modulare è più affidabile, più facile da mantenere e sicuro.

Gli obiettivi principali del sistema dei moduli sono di migliorare la sicurezza con una *strong encapsulation* e stabilità con dipendenze affidabili. Prima di JDK 9, non era possibile regolare le restrizioni di accesso pubblico e molte API venivano esposte. I moduli hanno pacchetti nascosti per uso interno e pacchetti esportati per codice condiviso con altri moduli. Si può anche specificare quali classi possono essere condivise con quali moduli o si può impostarli per essere accessibili a qualsiasi modulo. *Public* su una dichiarazione di classe *non significa più* accessibile a tutti.

Come si esegue la migrazione di un'applicazione esistente? Probabilmente non si migrerà un'intera applicazione al sistema modulare tutto in una volta. Il codice nei moduli e i jar tradizionali sul classpath coesisteranno nelle proprie applicazioni. Java 9 ha strumenti per facilitare la migrazione. Ad esempio, jdeps ti aiuta a trovare le dipendenze delle classi Java - basta eseguire lo strumento jdeps sui jar dell'applicazione. Esegue la scansione di file di classe o di file jar e dice quale codice dipende da altri file jar: non è necessario modificare i tradizionali file jar per eseguirli sul module path.

3.8 CONCLUSIONI

Nonostante la parola *Module* fosse già stata usata in relazione a Java, il concetto di *modulo* che viene introdotto con Java 9 è differente da quello precedente, ad esempio quello di modulo Maven e OSGi.
Il fatto che sia stato introdotto il module path in Java non vuol dire che il classpath non è più presente, e questo è il motivo per cui non abbiamo bisogno di modularizzare dei progetti già esistenti. Il mantenimento del classpath permetterà agli sviluppatori di poter utilizzare anche strumenti quali Maven per la gestione dei propri progetti; probabilmente però, scegliere la modularità offerta da Java per la creazione di nuovi progetti potrebbe essere la scelta giusta, visto il forte incapsulamento che essa offre, senza dover rinunciare a Maven per l'aggiunta e la gestione di librerie esterne tramite dipendenze.
Con i moduli Java abbiamo la possibilità di suddividere le responsabilità del codice senza esporre le classi definite con il modificatore *public* all'esterno di un modulo, se ciò non è necessario ai fini del nostro programma. Questo può essere utile, per esempio, quando non si vuol rendere visibili alcune API.

L'introduzione della modularità ha reso Java 9 un aggiornamento del linguaggio molto sostanziale; infatti, i piani di rilascio prevedevano che l'intero progetto Jigsaw fosse rilasciato con Java 8, ma successivamente è stato preferito dare maggior evidenza a funzionalità quali Lambda expressions e Stream, che altrimenti avrebbero avuto minor visibilità.
A nostro parere, questa novità dei moduli Java è sicuramente una rivoluzione per gli sviluppatori Java. Riteniamo, però, che essa sarà difficilmente utilizzata a breve in ambito lavorativo, perché la migrazione di un programma da Java 8 (o inferiore) a Java 9 sarebbe comunque un costo non

indifferente per un'azienda, sia economico che, principalmente, come fattore temporale.
Project Jigsaw ha comunque cercato di ovviare a questo con il contemporaneo rilascio di *jdeps*, strumento di cui abbiamo già parlato nel paragrafo 2, che permette una migrazione più rapida al sistema modulare tramite l'identificazione delle dipendenze sulle API JDK e sui JAR di terze parti.
Infine, con la versione 9, Java ha puntato a semplificare la manutenzione del codice sorgente del JDK attraverso la sua modularizzazione: ciò ha portato, secondo la nostra opinione, alla semplificazione della piattaforma Java, tanto che da Java 10 il ciclo di rilascio degli aggiornamenti è passato da dodici a sei mesi.

4

POTENZIAMENTO DELLA STRUTTURA SWITCH

Vogliamo fare un breve accenno all'unica novità inerente al linguaggio presente in Java 12, cioè il potenziamento della struttura *Switch*. Nonostante essa non sia, a nostro parere, una novità di grande rilievo, ci sembra un atto dovuto menzionarla.
Tale potenziamento prevede l'estensione di utilizzo dello switch in modo che possa essere utilizzata come un'espressione oltre che come un'istruzione.

Riportiamo un esempio di switch tradizionale:

```
public static void traditionalSwitchStatement() {
  System.out.println("Traditional Switch Statement:");
  final int integer = 3;
  String numericString;
  switch (integer) {
    case 1 :
      numericString = "one";
      break;
    case 2 :
      numericString = "two";
      break;
    case 3:
      numericString = "three";
      break;
    default:
      numericString = "N/A";
  }
  System.out.println("\t" + integer + " ==> " + numericString);
}
```

Questo particolare esempio mostra un uso comune dell'istruzione switch tradizionale per impostare il valore di una variabile locale.

Con questa novità viene introdotta una nuova sintassi che prevede l'utilizzo del simbolo freccia "->", che può essere utilizzata con un'istruzione switch o con un'espressione switch come riportato in *JEP 325: Switch Expressions*. Analogamente, la tradizionale sintassi con il simbolo due punti ":" può anche essere utilizzato con un'espressione switch o con un'istruzione switch. In altre parole, la presenza dei "due punti" (:) non implica necessariamente un'istruzione switch e la presenza di una "freccia" (->) non implica necessariamente un'espressione switch.

Ai fini di questa discussione, è utile riportare due osservazioni importanti:

- Un'*espressione* è un costrutto costituito da variabili, operatori e invocazioni di metodi che valuta un singolo valore.
- Il linguaggio di programmazione Java consente di costruire espressioni composte da varie espressioni più piccole purché il tipo di dato richiesto da una parte dell'espressione corrisponda al tipo di dato dell'altro.

Vediamo adesso come poter riscrivere il frammento di codice precedente con l'utilizzo dell'istruzione switch potenziata, utilizzando la sintassi con la freccia:

```
public static void enhancedSwitchStatement() {
  System.out.println("Enhanced Switch Statement:");
  final int integer = 2;
  String numericString;
  switch (integer) {
       case 1 -> numericString = "one";
       case 2 -> numericString = "two";
       case 3 -> numericString = "three";
       default -> numericString = "N/A";
  }
  System.out.println("\t" + integer + " ==> " + numericString);
}
```

Questo esempio mostra lo switch ancora utilizzato come istruzione, ma in questo caso sfrutta la sintassi "freccia" per eseguire il suo switching senza specificare esplicitamente un *break*. Questo non porta solo una

riduzione del codice, ma ha soprattutto il vantaggio di non consentire il **fall-through** della struttura switch, cioè il proseguire del flusso di controllo dopo l'esecuzione del case che soddisfa la condizione.

Oltre a migliorare l'attuale istruzione switch per consentire la specifica di un'istruzione senza il rischio di fall-through, Java 12 introduce anche il concetto di utilizzo della keyword *switch* in un'espressione switch.

Si può sostituire il codice mostrato sopra, per assegnare il valore del risultato alla variabile locale in una singola istruzione:

```
public static void switchExpressionWithBreaks() {
  final int integer = 1;
  System.out.println("Switch Expression with Colons/Breaks:");
  final String numericString =
    switch (integer) {
      case 1 :
           break "uno";
      case 2 :
           break "dos";
      case 3 :
           break "tres";
      default :
           break "N/A";
  };
  System.out.println("\t" + integer + " ==> " + numericString);
}
```

A differenza dello switch tradizionale le clausole *break* ora hanno ciascuna il valore da restituire per il caso rilevante immediatamente specificato dopo la keyword break. In sostanza, break nell'espressione switch si comporta come il *return* di un metodo Java.

Per restituire un valore da un'espressione switch possiamo inoltre utilizzare la sintassi "freccia" discussa in precedenza, evitando così la necessità della keyword break:

```
public static void switchExpressionWithArrows() {
  final int integer = 4;
  System.out.println("Switch Expression with Arrows:");
  final String numericString =
    switch (integer) {
      case 1 -> "uno";
```

```
        case 2 -> "dos";
        case 3 -> "tres";
        case 4 -> "quatro";
        default -> "N/A";
    };
  System.out.println("\t" + integer + " ==> " + numericString);
}
```

È inoltre possibile utilizzare più costanti per un singolo *case*, con entrambe le sintassi sopra citate, andando a ridurre ancor più la lunghezza del codice.

- `case 1, 3, 5, 7, 9 -> "odd";`
- ```
  case 1, 3, 5, 7, 9:
      numericString = "odd";
      break;
  ```

Come tutte le funzionalità, oltre a portare dei vantaggi in termini di refactoring, ha dei limiti che ne regolano l'utilizzo:

- non è possibile utilizzare le due notazioni contemporaneamente, quindi una struttura switch come la seguente restituirà un errore di compilazione

  ```
  switch(integer) {
      case 1 :
          numericString = "one";
          break;
      case 2 -> numericString = "two";
      default -> numericString = "N/A";
  }
  ```

- attribuire un valore di ritorno a un *break* risulterà in un errore di compilazione, perché ciò non è permesso in un'istruzione switch tradizionale

  ```
  switch (integer) {
      case 1:
          break "one";
  ```
  ```

```
        case 2:
            break "two";
        default:
            break "N/A";
    };
```

tale limite può essere semplicemente superato assegnando lo switch a una variabile locale, che risulterà in un passaggio da istruzione ad espressione switch

```
final String numericString =
 switch (integer) {
    case 1:
        break "one";
    case 2:
        break "two";
    default:
        break "N/A";
 };
```

- in un'istruzione switch che usa la sintassi freccia, ciascun *case* deve presentare una valida dichiarazione Java, altrimenti risulterà in un errore di compilazione

```
 switch (integer) {
    case 1 -> "one";
    case 2 -> "two";
 };
```

- dato che un'espressione switch richiede un valore di ritorno non-`void`, è necessario esplicitare un case per ogni possibilità, facilmente realizzabile con una label *default*, pertanto l'esempio seguente non compilerà

```
final int integer = 5;
String numericString =
    switch (integer) {
        case 1 -> "one";
```

```
            case 2 -> "two";
        };
```

Il significato dell'introduzione della struttura switch potenziata è però legata ai numerosi problemi presenti nell'istruzione switch tradizionale, quali: il problema del fall-through, l'ambito predefinito dei blocchi switch (il blocco viene considerato come un unico ambito) e il fatto che lo switch funziona solo come una dichiarazione, anche se è comunemente più naturale esprimere condizionali a più vie (es. struttura `if-else`) come delle espressioni.
Introducendo tale novità, gli sviluppatori del linguaggio hanno fatto fronte ai problemi sopra citati, andando anche a migliorare il refactoring del codice all'interno di una struttura switch, che adesso potrà essere un'istruzione (come nella versione tradizionale) o un'espressione alla quale attribuire un valore di ritorno.

CONCLUSIONI

In questa tesi abbiamo passato in rassegna le recenti novità introdotte nelle ultime versioni di Java, da Java 9 a Java 12. Per ciascuna di queste nuove caratteristiche, abbiamo analizzato le motivazioni sottostanti alla sua introduzione, i casi di applicazione, i vantaggi conseguenti al suo uso e i limiti. In particolare, abbiamo cercato di individuare, attraverso le nuove funzionalità, il filo conduttore comune e, soprattutto, un tracciato presumibile per le prossime evoluzioni del linguaggio.

Cercando di tirare le fila delle nostre analisi, possiamo trarre alcune conclusioni riassuntive.

In primo luogo, si nota un'evoluzione del sistema di tipo verso un potenziamento della *type inference* rispetto ai tipi espliciti, al puro *typechecking* ed ai tipi nominali, che sono state le caratteristiche basilari iniziali del controllo dei tipi in Java.
Anche se l'inferenza di tipo era già stata potenziata con l'introduzione delle lambda-expressions, Java era rimasto essenzialmente un linguaggio con tipi espliciti, associati ad ogni *identificatore*, e con tipi nominali. Invece con la Local-Variable Type Inference di Java 10, è finalmente possibile introdurre identificatori senza il tipo esplicito, attraverso l'uso della keyword `var`.
Ad esempio in Java 11 l'uso di `var` è consentito anche all'interno dei parametri di una lambda-expression. Questo risponde ad un'esigenza degli sviluppatori Java, che da tempo si lamentano per il "boilerplate code" e le tediose formalità talvolta necessarie durante la scrittura del codice. Molte cose che richiedono solo 5 minuti in linguaggi come Python o JavaScript possono richiedere più di 30 minuti in Java a causa di una certa verbosità del linguaggio.
Fra gli elementi di questa verbosità la dichiarazione esplicita del tipo di ogni variabile è certamente un punto rilevante. Nella stessa linea si colloca un progressivo affermarsi dell'uso di *Non Denotable Types*, vale a dire tipi che possono esistere all'interno del programma, inferiti dal controllo dei tipi, ma non associati ad un nome di tipo introdotto dal programmatore. Pensiamo, ad esempio, all'uso sempre più consistente dei tipi intersezione.

Ovviamente l'affermarsi di queste caratteristiche implica algoritmi di *type inference* più sofisticati, rispetto al semplice controllo di consistenza dei tipi usati, detto *typechecking*.
Dunque, come prima conclusione, alla luce di quanto discusso nel Capitolo 2, possiamo dire che una linea di tendenza nell'evoluzione di Java sembra essere quella di potenziare algoritmi di inferenza di tipi, anche non-nominali, invece di lasciare al programmatore l'onere completo di esplicitare e definire tutti i tipi necessari al programma.

In secondo luogo, possiamo affermare che l'introduzione dei *Moduli* in Java 9 è stata la novità più importante, a lungo attesa, per gli sviluppatori Java. Come evidenziato nel Capitolo 3, questo strumento è ancora più efficace dell'uso di altri strumenti esterni quali i moduli Maven, per realizzare una vera modularizzazione del codice, con grande affidabilità, maggiore sicurezza e forte incapsulamento.

Un altro concetto che è mutato molto nelle ultime versioni di Java è certamente la definizione di *Interfaccia*. La *interface* non è più un contratto astratto dei servizi offerti, ma può contenere codice, ossia implementazione, a diversi livelli, ad esempio metodi *default* e *private*. Abbiamo visto, nel Capitolo 1, le motivazioni che hanno portato a questi cambiamenti: per esempio, la necessità di fare evolvere un'interfaccia senza introdurre errori nelle gerarchie di sottoclassi già esistenti. Certamente, questo nuovo concetto di Interfaccia può essere utile in molti casi, soprattutto quando trattiamo Interfacce Funzionali per le lambda-expressions. Ma, al di là di queste utilità, non ci pare che questo sia il caso di una evoluzione significativa del linguaggio; anzi, diverse critiche da noi condivise, sono state portate contro questo arricchimento di codice nel concetto di interfaccia.

BIBLIOGRAFIA

[1] Luca Cardelli - *Type Systems in The Computer Science and Engineering Handbook 1997* - 2208-2236 (Cited on page 25.)

[2] Brian Goetz - *JEP 286: Local-Variable Type Inference* (Cited on page 29.)

[3] Gabriela Motroc - *Java 9 misconceptions & Java 10 wish list: A tell-it-all interview with Java influencers* - 2017, `https://jaxenter.com/java-influencers-interview-3-135847.html` (Cited on page 30.)

[4] Trisha Gee - *A tell-it-all interview with Java influencers* - 2017, `https://jaxenter.com/java-influencers-interview-3-135847.html` (Cited on page 30.)

[5] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley - *Reference Conditional Expressions* - 2014, The Java Language Specification (Java SE 8 Edition), par. 15.25.3 (Cited on page 37.)

[6] Stuart W. Marks - *Style Guidelines for Local Variable Type Inference in Java* - 2018, `https://openjdk.java.net/projects/amber/LVTIstyle.html` (Cited on page 42.)

[7] Joshua Bloch - *Minimize the scope of local variables* - 2017, Effective Java (3rd edition), Item 57 (Cited on page 44.)

[8] Robert C. Martin - *The Single Responsibility Principle* - 2014, The Clean Code Blog (Cited on page 55.)

[9] Mark Reinhold -*Readability* - 2016, The State of the Module System, Par. 2.3 (Cited on page 60.)

[10] Mark Reinhold -*Implied Readability* - 2016, The State of the Module System, Par. 2.5 (Cited on page 60.)

[11] Glen McCluskey - *Using Java Reflection* - 1998, `https://www.oracle.com/technetwork/articles/java/javareflection-1536171.html` (Cited on page 61.)

[12] Marcus Biel - *Interview with Marcus Biel, CleanCode Evangelist and JCP member* - 2017, https://jaxenter.com/java-9-interview-marcus-biel-137836.html (Cited on page 71.)

[13] Lukas Eder - *Interview with Java Champion Lukas Eder* - 2017, https://jaxenter.com/java-9-jigsaw-interview-eder-137726.html (Cited on page 71.)

[14] Alex Buckley - *Modular Development with JDK 9* - 2018, https://blogs.oracle.com/java/modular-development (Cited on page 72.)