ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Scienze
Corso di Laurea in Ingegneria e Scienze Informatiche

IL LINGUAGGIO CEYLON

Elaborato in
PROGRAMMAZIONE AD OGGETTI

Relatore
Prof. MIRKO VIROLI

Co-relatore
Ing. DANILO PIANINI

Presentata da
LUCA CASAMENTI

Seconda Sessione di Laurea
Anno Accademico 2018 – 2019

PAROLE CHIAVE

Language

JVM

Java

Ceylon

Features

A mia sorella Elisa

Indice

Introduzione

Il linguaggio Ceylon è un linguaggio di programmazione orientato agli oggetti, a tipizzazione statica e con un enfasi sull'immutabilità, progettato da Gavin King (Red Hat) nel 2011. Questo elaborato nasce come progetto di studio del linguaggio e delle sue caratteristiche principali confrontandolo con il linguaggio Java, a cui Ceylon si ispira in molti aspetti, a partire dalla struttura. In questo documento analizzo il linguaggio Ceylon, a partire da Java, dalla sua evoluzione nel tempo e dal suo ecosistema, per poi analizzare nello specifico Ceylon, mostrandone type system, features principali e le differenze con il linguaggio Java.

Sommario

La tesi è suddivisa in due macro capitoli più un terzo e quarto capitolo in aggiunta con il setup e qualche esempio d'uso. Il primo capitolo si occupa di analizzare come il linguaggio Java si è evoluto negli anni, mostrando le features principali introdotte durante gli aggiornamenti della piattaforma. Nel secondo invece, viene mostrato nello specifico il linguaggio Ceylon, esponendone le principali caratteristiche e facendo qualche paragone con altri linguaggi della JVM.

Capitolo 1 Il primo capitolo affronta brevemente la Java Virtual Machine. Dopodiché si occupa di analizzare le caratteristiche principali delle più importanti release del linguaggio Java. Nello specifico il capitolo è a sua volta suddiviso in sezioni contenenti le principali versioni del linguaggio e le features introdotte con esse.

Capitolo 2 Il secondo capitolo esplora il linguaggio Ceylon partendo da una panoramica sulla sintassi di base, come la costruzione di variabili e classi, per passare poi ad un'analisi più nel dettaglio di alcune delle più importanti funzionalità del linguaggio.

Capitolo 3 Il terzo capitolo mostra il setup di Ceylon sull'IDE Eclipse.

Capitolo 4 Il quarto capitolo mostra qualche esempio d'uso del linguaggio Ceylon.

Capitolo 1

Java e la JVM

1.1 L'ecosistema Java e la Java virtual machine

Java è una piattaforma software e linguaggio di programmazione originariamente sviluppato da James Gosling presso Sun Microsystem e infine acquisita nel Gennaio 2010 dalla Oracle Corporation. Tale piattaforma ha come caratteristica principale il fatto di rendere possibile scrittura ed esecuzione di applicazioni indipendentemente dall'hardware sul quale vengono eseguiti. Tutto ciò è permesso da un componente chiamato Java Virtual Machine (JVM) che interpreta il bytecode, ovvero un linguaggio intermedio più astratto tra il linguaggio macchina e il linguaggio di programmazione, generato a seguito di una prima compilazione del codice. Esistono implementazioni software della JVM per praticamente tutti i sistemi operativi; oltre a implementazioni speciali per particolari ambienti hardware/software, come telefoni cellulari o palmari. Questa è la chiave della cosiddetta "portabilità" di Java, ovvero il processo di trasposizione, volto a consentire l'uso un componente software in un ambiente di esecuzione diverso da quello originale, proclamata nello slogan "write once, run everywhere". Un'implementazione della piattaforma Java è la Java Runtime Environment (JRE), contenente la Java Virtual Machine e le librerie standard (API Java). Per lo sviluppo dei programmi in Java a partire dal codice sorgente è necessario invece il Java Development Kit (JDK), che include anche il JRE. In supporto alla programmazione, in linguaggio Java e non solo, esistono anche ambienti di sviluppo integrato (in lingua inglese Integrated Development Environment ovvero IDE), ovvero dei software che, in fase di programmazione, aiutano i programmatori nello sviluppo del codice sorgente di un programma segnalando errori di sintassi del codice direttamente in fase di scrittura, oltre a tutta una serie di strumenti e funzionalità di supporto alla

fase di sviluppo e debugging. Gli ambienti di sviluppo integrato più famosi e utilizzati per la programmazione in linguaggio Java sono Eclipse, Netbeans e IntelliJ Idea.

1.2 Il linguaggio Java: evoluzione

1.2.1 Da Java 1 a Java 4

Sebbene Java sia conosciuto a molti a partire dal 1996, quando iniziò a circolare la versione 1.0, il progetto iniziale ebbe inizio originariamente nel 1990-91, quando un'elite di 14 professionisti della Sun Microsystem capitanata da James Gosling, considerato "il padre" di Java, si riunì ai piedi di una quercia situata a Sand Hill Road, Menlo Park, California, per ideare il linguaggio che avrebbe dovuto rimpiazzare C++. Nacque Oak (in italiano "quercia"), un linguaggio di programmazione che aveva l'obiettivo di sfruttare il meglio dei linguaggi OO esistenti e risolvere i problemi che più frustravano i programmatori C++, ad esempio la mancanza di un dispositivo automatico di allocazione/-deallocazione della memoria; problema considerato cardine e foriero di bug, o la mancanza di abilità native di networking, threading, e soprattutto di trasporto su diverse piattaforme. Il team, che prese il nome di "Green Project", aveva come obiettivo iniziale quello di creare un linguaggio per la logica di controllo degli elettrodomestici di largo consumo; successivamente si pensò di utilizzarlo per la TV interattiva via cavo. L'avvento dei browser web grafici, come Mosaic, permisero a Oak (Java) di segnare una rivoluzione nel mondo di Internet. Grazie infatti alle Applet, le pagine web diventarono interattive a livello client e gli utenti poterono per esempio utilizzare giochi direttamente sulle pagine web ed usufruire di chat dinamiche e interattive. Java si è evoluto nel tempo con un ritmo lento e le versioni principali del linguaggio vennero rilasciate ogni due o tre anni, spesso impiegando anche più tempo del dovuto; per questo motivo Mark Reinhold, capo architetto della piattaforma Java, nel Settembre 2017 propose di cambiare il ritmo di aggiornamenti delle features, il cosiddetto "Java release train", rilasciando un aggiornamento ogni sei mesi[1]. La prima versione stabile disponibile (Java 1.0.2) fu rilasciata il 23 gennaio 1996, completa del Java Development Kit per i soli ambienti Windows e Solaris. A quei tempi, il JDK includeva tutte le varie componenti, incluso l'ambiente di run-time (JRE, Java Runtime Environmnet) apparso come entità a se stante solo a partire dalla versione 1.1. Questa prima versione del linguaggio includeva due package allora particolarmente apprezzati ovvero `AWT`, utilizzabile per la creazione di interfacce utente grafiche portabili e `Applet` che

[1] https://mreinhold.org/blog/forward-faster

consentiva la creazione delle stesse. Altri package fondamentali introdotti in questa versione sono stati:

- `java.lang`: Contiene classi fondamentali come ad esempio `Byte`, `Double`, `Long`, `String`, `Runtime`, `Thread`, etc... ;
- `java.io`: Package che fornisce classi per gestire stream di input e di output per leggere e scrivere dati su file, stringhe ed altre fonti
- `java.io`: Package che fornisce le classi per gestire stream di input e di output per leggere e scrivere dati su file, stringhe ed altre fonti.
- `java.util`: Conteneva classi di utilità varie, comprese le strutture di dati generiche (`Vector`, `Hastable`, etc.), insiemi di bit, ora, data, manipolazione di stringhe, la generazione di numeri casuali, proprietà di sistema, la notifica (`Observer`, `Observable`, etc) ed enumerazione di strutture di dati (come l'interfaccia `Enumeration`);
- `java.net`: Fornisce classi per il supporto del networking, inclusi "Universal Resource Locator" (URL, localizzatore universale di risorse), socket Transmission Control Protocol (TCP, protocollo di controllo di trasmissione) e User Datagram Protocol (UDP, protocollo datagrammi utente), indirizzi Internet Protocol (IP, protocollo Internet), e un convertitore di binario/test;
- `java.awt.image`: Fornisce le classi per la gestione di immagini, compresi i modelli di colore, ritaglio, filtro colore, e l'impostazione dei valori dei pixel e PixelGrabber;
- `java.awt.peer`: Package di collegamento tra le componenti AWT alle correspondenti componenti specifiche della piattaforma di esecuzione (esempio: "widget Motif" e controlli Microsoft Windows).

Questa prima release poteva contare in tutto 8 package e 212 classi, considerate ai tempi un ricco supporto di librerie. Sebbene la Java 1.0.2 presentasse molte interessanti potenzialità, si trattava comunque di una prima versione ancora acerba. Il controllo della "user interface" era limitato e le corrispondenti interfacce peccavano di scarsa stabilità; non c'era un supporto per la connessione a database, non erano ancora state incluse feature per internazionalizzazione e localizzazione, non esisteva il framework Remote Method Invocation (RMI, invocazione di metodo remoto), né la serializzazione, e il "compilatore just-in-time(JIT)" era più un'idea che una vera implementazione.

La successiva release, Java 1.1, fu rilasciata il 19 febbraio 1997. Secondo molti programmatori questa era la prima vera e propria versione di Java

utilizzabile professionalmente per la costruzione di sistemi complessi. La sua formulazione fu il risultato di un'estesa rivisitazione del linguaggio, di cui le feature principali includevano:

- AWT rinnovato. Ci fu una completa rivisitazione dei package AWT. La parte del modello degli eventi (event model) fu interamente re ingegnerizzata;

- Le classi annidate, ovvero costrutti utili per la corretta implementazioni di framework/classi di utilità come le collezioni. Per esempio, l'utilizzatissima classe `java.util.HashMap` (introdotta con Java 1.2) dichiara una serie di classi annidate, di cui le più importanti sono: `Entry` (utilizzata per incapsulare gli oggetti memorizzati nella map in modo da poter costruire delle linked list), `HashIterator` (necessaria per implementare la funzionalità di iterazione degli elementi della mappa) e gli altri iterator (`ValueIterator`, `KeyIterator`, `EntryIterator`), `KeySet` (serve per costruire una view dell'insieme delle chiavi presenti nella mappa), `Values` (serve per costruire una view dell'insieme delle chiavi presenti nella mappa) ed `EntrySet` (serve per costruire una view delle `Entry` presenti nella mappa);

- "JavaBeans" ovvero classi utilizzate per incapsulare più oggetti in un oggetto singolo;

- il supporto ai database, grazie al JDBC (Java DataBAse Connectivity). JDBC è l'architettura e la corrispondente API che permettono alle applicazioni Java di interagire con i DBMS per eseguire comandi SQL. L'architettura JDBC è stata disegnata prendendo spunto da Microsoft Open Database Connectivity (ODBC, connettività aperta al database) utilizzato per l'accesso e la gestione di database con la differenza che mentre JDBC è Java specifico, ODBC è indipendente dai specifici linguaggi di programmazione;

- La definizione di una prima versione del framework "RMI" mancante del supporto del protocollo IIOP (Internet Inter Orb Protocol), protocollo pensato in modo da garantire la comunicazione sullo strato TCP (Internet) e di connettere ORB differenti fra loro (IOP); L'integrazione del mondo Java RMI e di CORBA è resa possibile grazie alla introduzione di RMI-IIOP, sviluppato congiuntamente da IBM e SUN nel 1999[2]. Si tratta di un meccanismo che permette di eseguire un metodo esposto

[2]http://www.mokabyte.it/mokaold/2003/01/rmi-iiop.htm

da un oggetto remoto (Remote Method Invocation) tipicamente presente in una diversa JVM. La versione iniziale, denominata Java Remote Method Protocol (JRMP, protocollo Java specifico di invocazione dei metodi remoti), era specifica Java (non IIOP);

- Una versione iniziale della "Reflection" basata sull'introspezione (introspection). In particolare, queste feature permettono di ottenere informazioni relative alle classi a run-time come costruttori, metodi e attributi. L'introspection (e la reflection più in generale) sono un meccanismo fondamentale per poter implementare architetture basate su "plug-in";

- L'introduzione dell'interfaccia Java per il codice nativo (Java Native Interface, JNI). Si tratta di una che permette alle applicazioni di incorporare codice nativo scritto in linguaggi di programmazione come C e C++.

L'8 dicembre del 1998 fu rilasciata la nuova major release di Java, che fu inizialmente battezzata "Java 2" (a discapito di chi si aspettava la versione "Java 1.2"). Si trattò però di un nome che finì per creare non poca confusione, al punto che "Java 2" fu cambiato retrospettivamente nella sigla "J2" facendo nascere J2SE (Java 2 Platform, Standard Edition) ma in versione 1.2. Questa nomenclatura è stata poi mantenuta fino a Java 6 in cui si assiste al nuovo cambiamento. Si decise di sostituire la dicitura "JDK" con "J2" per poter distinguere le tre piattaforme base:

- edizione standard, J2SE

- edizione enterprise, J2EE (Java 2 Platform, Enterprise Edition)

- edizione micro, J2ME (Java 2 Platform, Micro Edition)

"Java 1.2" fu una versione java molto importante in quanto triplicò le dimensioni della piattaforma Java (a 1520 classi in 59 pacchetti). Le feature principali introducevano importanti novità:

- "JCF": Fu operata una completa rielaborazione del framework delle collezioni JCF (Java Collections Framework). Si tratta di un insieme di classi e interfacce che implementano strutture dati di utilizzo frequente come per esempio array e mappe. Le versioni precedenti di Java già includevano concetti equivalenti (per esempio `Vector` e `HashMap`), tuttavia queste non erano organizzate in un framework e non si basavano su un insieme di interfacce standard, perciò il loro uso non era immediato.

- "Swing": Java 2 vide l'inclusione dell'API grafica Swing. La versione primordiale Swing si deve alle Internet Foundation Classes (IFC, classi fondamentali internet) per Java sviluppate originariamente dalla Netscape Communications Corporation e rilasciate per la prima volta nel dicembre del 1996. "AWT" presentava una serie di limitazioni, e gli sviluppatori dovevano avere una buona consapevolezza delle differenze di comportamento della GUI nelle varie piattaforme al fine di prendere le dovute contromisure per far sì che le interfacce utente fossero veramente "portabili"

- Fu introdotta la capacità di Java di invocare e di esporre servizi compatibili CORBA[3], permettendo un'interoperabilità run-time con componenti implementati con diversi linguaggi di programmazione. Si trattò di fondere insieme le feature di Java RMI con quella di CORBA. Per questo lavoro la Sun Microsystem si avvalse della collaborazione di IBM.

- Fu introdotta la parola chiave *strictfp* (strict floating point, aritmetica a virgola mobile rigida) che limita i calcoli in virgola mobile per garantire la portabilità del codice.

- "JIT": La JVM (Java Virtual Machine) fu equipaggiata per la prima volta con un componente Just In Time (JIT compilation, in italiano "compilazione appena in tempo"). Si tratta di una strategia atta ad accelerare le performance a tempo di esecuzione del bytecode. Ciò è ottenuto attraverso un'analisi basata su due direttive principali:

  - compilazione di blocchi di codice in anticipo rispetto alla reale richiesta;
  - esecuzione di ottimizzazioni basati su pattern pre-definiti sul bytecode stesso.

La versione "J2SE 1.3" fu rilasciata l'8 maggio 2000 con il nome di "Kestrel" (si tratta di un piccolo falco, comunemente noto come gheppio). Non si tratta di una versione rivoluzionaria, tuttavia introdusse alcune significative caratteristiche, tra le quali le più importanti sono:

- "JNDI"(Java Naming and Directory Interface): JNDI è una API Java pensata per integrare servizi di "naming" e "directory". Si tratta di software che permettono di memorizzare e organizzare informazioni, tipicamente secondo un ordine strettamente gerarchico, e di accedervi per mezzo di una chiave, tipicamente mnemonica;

[3]https://it.wikipedia.org/wiki/Common_Object_Request_Broker_Architecture

- "Java Sound": L'API del suono Java è una API di basso livello che permette alle applicazioni Java di controllare l'input e l'output di media audio, incluso il formato Musical Instrument Digital Interface (MIDI[4], interfaccia digitale dei strumenti musicali). La API Java Sound fornisce un controllo esplicito delle feature normalmente richieste per l'input e l'output dei media audio, in un framework che promuove l'estensibilità e la flessibilità;

- "JPDA" (Java Platform Debugger Architecture): Una collezione di API che permettono di eseguire debug sul codice Java. In particolare, questa architettura include le seguenti API:

  - Java Debugger Interface (JDI, interfaccia del debugger Java) che definisce un'interfaccia di alto livello per l'implementazione di strumenti per il debugger remoto;
  - Java Virtual Machine Tools Interface (JVMTI, interfaccia per i tools della JVM): Si tratta di un'interfaccia nativa che permette di eseguire l'ispezione dello stato e del controllo di esecuzione dell'applicazione in corso di esecuzione sulla JVM;
  - Java Virtual Machine Debug Interface;
  - Java Debug Wire Protocol (JDWP, protocollo di rete per il degub Java) che definisce il protocollo di comunicazione tra l'applicazione oggetto di debug e l'applicazione debugger;

- "HotSpot": Si tratta di una JVM gestita e distribuita da Oracle Corporation. Le sue feature più interessanti ai tempi erano la presenza del compilatore Just-In-Time e un ottimizzatore adattivo (si tratta di una componente che esegue compilazioni dinamiche di porzioni di programma in funzione del corrente profilo di esecuzione) pensato per migliorare le performance. Il nome è dovuto al fatto che mentre esegue il byte code, la JVM analizza continuamente le performance del programma per individuarne i punti caldi (hot spots) eseguiti più frequente o ripetutamente che, come tali, costituiscono i target ideali per possibili ottimizzazioni;

- "Synthetic proxy classes / dynamic proxy": I proxy sintetici, successivamente rinominati in proxy dinamici, (dynamic proxy) sono stati introdotti in Java 1.3 tramite l'introduzione della classe `Proxy` (`java.lang.reflect.Proxy`) e dell'interfaccia `InvocationHandler` (`java.lang.reflect.InvocationHandler`) al package `java.lang.reflect`. Questa coppia consente la creazione di

[4]https://docs.oracle.com/javase/7/docs/api/javax/sound/midi/package-summary.html

"oggetti proxy sintetici": una classe dynamic proxy è una classe che implementa un elenco di interfacce specificate a run-time quando la classe viene creata. Una chiamata di un metodo di un'istanza proxy attraverso una delle sue interfacce proxy viene passata al metodo invoke del gestore di invocazione dell'istanza, passando all'istanza proxy tutti le informazioni della chiamata. Il gestore dell'invocazione esegue le previste elaborazioni e a seconda dei casi il risultato viene restituito come risultato della chiamata del metodo sull'istanza di proxy. Si tratta di un importante meccanismo che consente di intercettare le chiamate a specifici metodi in modo da poter interporre un comportamento aggiuntivo tra il chiamante di un oggetto e l'oggetto stesso. Si tratta di un ottimo meccanismo che permette di implementare aspetti importanti per l'applicazione, ma ortogonali al compito principale delle singole classi. I proxy dinamici sono alla base di molti importanti framework, tra cui Spring[5].

La versione J2SE 1.4 fu rilasciata il 6 febbraio del 2002 con il nome di Merlin (anche questo è un piccolo falco, lo smeriglio). Si tratta di una versione che ha fatto la storia di Java in quanto è stata la prima major release sviluppata attraverso il Java Community Process (JCP, processo della Java Community). Il JCP permette alla parti interessate di lavorare allo sviluppo delle specifiche standard Java e quindi di collaborare alla definizione del futuro della tecnologia Java. J2SE 1.4 è parte della Java Specification Request 59 (JSR, richiesta di specificazione Java, cfr). I JSR sono i documenti formali che descrivono le proposte di modifiche e relativa evoluzione. La J2SE versione 1.4 introdusse una serie di cambiamenti; di seguito i più significativi:

- `Assert`: Venne introdotta la struttura assert. Si tratta di un costrutto utilizzato per specificare delle assunzioni del programmatore in merito a determinate condizioni che devono essere soddisfatte in specifici punti del codice. L'obiettivo consiste nel migliorare la robustezza dei programmi. Ogni assert contiene espressioni booleane che devono essere soddisfatte durante l'esecuzione(invarianti): in caso contrario il sistema genera un errore runtime incluso nella classe `java.lang.AssertionError`. Di default le *Assertion* sono disabilitate a runtime, per abilitarle è necessario il comando "-enableassertions" o "-ea";

- "NIO": Fu introdotta una nuova API di Input/Output (New Input/Output, NIO) specificata per mezzo della JSR51. Le principali feature introdotte sono: possibilità di eseguire il mapping in memoria dei files; operazioni orientate ai blocchi che sfruttano i servizi messi a disposizione dalle piattaforme di esecuzione al posto del tradizione flusso di byte;

[5]https://en.wikipedia.org/wiki/Spring_Framework

operazioni di I/O asincrone e quindi non bloccanti e lock dei file o di opportune sezioni. Sebbene il package `java.nio` includa molti elementi interessanti, il framework è basato su tre concetti fondamentali: `Buffer`, che come atteso sono contenitori di dati; `Charset`, con i corrispondenti decoder ed encoder, eseguono la traduzione tra byte e caratteri Unicode; channels ossia canali di vario tipo che rappresentano le connessioni con dispositivi di vario tipo in grado di eseguire operazioni I/O; selectors (selettori), che insieme ai canali permettono di definire operazioni multiplex e non-bloccanti;

- "IPv6": Supporto per Internet Protocol versione 6. Si tratta di un aggiornamento del formato dell'indirizzo dei pacchetti di reti resosi necessario per fronteggiare il previsto esaurimento delle configurazioni specificate dalla versione precedente (da IPv4 a 32 bit si è passati a IPv6 a 128 bit);

- "JAXP": Altra caratteristica degna di nota fu l'integrazione JAXP (Java API for XML Processing, API Java per l'elaborazione XML) che include il parser XML e il processore XSLT. JAXP consente alle applicazioni di eseguire il parsing, trasformare, validare e interrogare (query) documenti XML utilizzando una API indipendente da una particolare implementazione del processore XML;

- Security: Nella versione 1.4 fu data particolare importanza al tema dell'integrazione della sicurezza e delle estensioni per il supporto della crittografia. Ciò include una serie di API quali:

  – JAAS (Java Authentication and Authorization Service, servizi Java di autenticazione e autorizzazione). Inizialmente definito come package opzionale (estensione) per J2SE 1.3 è stato poi integrato definitivamente in J2SE 1.4. Questa API definisce il framework per implementare servizi di autenticazione, il cui obiettivo è di accertarsi in maniera affidabili che l'utente (o il componente software) sia effettivamente chi dichiara di essere, e di autorizzazione, il cui obiettivo è di verificare che l'utente (o il sistema) precedentemente autenticato abbiamo i permessi necessari per eseguire una determinata azione;

  – JSSE (Java Secure Socket Extension, estensione per la sicurezza dei socket Java). Anche in questo caso si tratta di un package che era opzionale per le versione J2SE 1.2 e 1.3, e che viene completamente integrato con la versione J2SE 1.4. JSSE è un insieme di package che permettono di effettuare comunicazione Internet sicure. In particolare, vi è una versione Java dei protocolli Secure Sockets Layer (SSL, strato socket sicuri) e del Transport Layer Security (TLS,

strato di sicurezza del trasporto). Inoltre, include funzionalità per la crittografia dei dai, l'integrità dei messaggi e l'autenticazione dei server;

  - JCE (Java Cryptography Extension, estensione Java per la crittografia). Come visto prima, anche JCE era un package opzionale per le versioni J2SE 1.2 e 1.3 ed è stato poi integrato al J2SE versione 1.4. Si tratta di un framework con relativa implementazione che fornisce servizi di crittografia, generazione di chiavi con relativa certificazione e algoritmi Message Authentication Code (MAC, codice di autenticazione del messaggio). I servizi di crittografia, a loro volta, includono cifrature simmetriche, asimmetriche, a cipher a blocchi e stream. Come lecito attendersi, JCE si basa sugli stessi principi di progettazione del JCA basati sull'indipendenza dalla specifica implementazione e, ove possibile, anche da specifici algoritmi. Pertanto, il funzionamento richiede la presenza di plug-in in grado di fornire i servizi specifici.

- "Java WS": Integrazione di Java Web Start (JavaWS o JAWS, avvio Web di Java). La prima versione di Java Web Start fu rilasciata nel marzo 2001 come estensione per J2SE 1.3. Si tratta di un framework originariamente sviluppato dalla Sun Microsystem che permette agli utenti di eseguire applicazioni Java scaricate direttamente da Internet attraverso un browser commerciale: un po' come avviene con le Applet, con la differenza che queste applicazioni non funzionano all'interno di un browser (non sono parte della pagina da cui sono scaricate). Tuttavia, di default, funzionano in un ambiente ristretto simile al sandbox previsto per le Applet, con qualche piccola concessione.

1.2.2 Java 5 e i Generici

A questo punto dell'evoluzione di Java, appariva chiaro che nella versione successiva ci sarebbero state alcune novità di rilievo. Lo sviluppo delle tecnologie Internet, l'adozione globale del linguaggio Java come piattaforma di sviluppo per progetti importanti, la base sempre più ampia di sistemi e dispositivi che facevano uso di Java erano il presupposto per una piccola rivoluzione. Il primo cambiamento netto, comunque, si ebbe ancora una volta nella denominazione della versione. Laddove ci si aspettava la versione 1.5 di Java 2 (il che non è già di per se il massimo della linearità, ma rientrava comunque nello standard di denominazione adottato fin qui), ossia una J2SE 1.5, assistemmo invece al rilascio della versione 5.0, con una mutazione nella progressione dei numeri che poi sarà meglio definita nelle versioni successive. Nel 1998, Gilad

Bracha, Martin Odersky, David Stoutamire e Philip Wadler crearono *Generic Java* (*GJ*), un'estensione del linguaggio Java per supportare tipi generici [3]. GJ è stato incorporato in Java con l'aggiunta di wildcard, basate sul lavoro di Igarashi e Viroli [5]. La versione J2SE 5.0, rilasciata il 30 settembre del 2004 con la JSR 176, nome in codice Tiger, ha rappresentato una piccola rivoluzione del linguaggio Java per via degli importanti cambiamenti apportati al linguaggio.

I generici sono una funzionalità introdotta per estendere il type system di Java al fine di consentire a "un tipo o un metodo di operare su oggetti di vario tipo fornendo al contempo sicurezza dello stesso in fase di compilazione". La complessità aggiunta è considerevole, tanto che nel 2016 è stato dimostrato che il type system con generici e null è inconsistente[1].

I tipi parametrizzati rivestono particolare importanza perché consentono di creare classi, interfacce e metodi per i quali il tipo di dato sul quale si opera può essere specificato come parametro. Parliamo quindi di classi, interfacce e metodi generici. Attraverso un codice generico possiamo realizzare un algoritmo che astrae dal tipo di dato specifico sul quale opera. Ad esempio un algoritmo per l'ordinamento di entità numeriche è lo stesso qualunque sia il tipo numerico che si utilizza; il tipo numerico utilizzato rappresenta quindi un parametro. Una classe generica rappresenta un type constructor, e specifica l'uso di un parametro di tipo con la notazione "simbolo minore – carattere – simbolo maggiore" chiamato "Diamond Operator"; una convezione sintattica prevede di usare singole lettere maiuscole per il tipo generico. Non siamo poi limitati all'uso di un solo parametro, possiamo infatti definire classi generiche con più parametri. Per garantire compatibilità con le versioni precedenti alla J2SE 5, l'implementazione in Java dei generici viene realizzata attraverso la tecnica nota con il nome di *erasure*[7]. Quando viene compilato il codice sorgente, tutti i parametri di tipo vengono sostituiti dal loro tipo limite che in mancanza di una specifica diversa è `Object`. Avviene quindi il cast appropriato da parte del compilatore che si preoccupa di gestire la compatibilità del tipo. In sostanza i generics sono semplicemente un meccanismo del codice sorgente e non esistono parametri di tipi in fase di esecuzione.

I tipi generici sono in generale invarianti. Se si è alla ricerca della compatibilità, bisogna prendere in considerazione casi specifici e tipi di parametri di singoli metodi. Quindi, alla normale notazione dei tipi generici `List<T>`, usata per creare oggetti, si affianca una nuova notazione, pensata per esprimere i tipi accettabili come parametri in singoli metodi. Si parla quindi di tipi parametrici varianti, in Java detti wildcard [8]. Posto che la notazione `List<T>` denota

il normale tipo generico, si introducono le seguenti notazioni wildcard:

- tipo covariante `List< ? extends T >`: cattura le proprietà dei `List<X>` in cui X estende T; si usa per specificare tipi che possono essere solo letti.
- tipo controvariante `List< ? super T >`: cattura le proprietà dei `List<X>` in cui X è esteso da T; si usa per specificare tipi che possono essere solo scritti.
- tipo bivariante `List<?>`: cattura tutti i `List<T>` senza distinzioni; si usa per specificare i tipi che non consentono né letture né scritture.

L'introduzione nel codice Java dei generici ha portato con sé qualche limitazione, come ad esempio:

- Il tipo formale di un parametro generico come ad esempio "T" non può essere usato per istanziare un oggetto, poiché al momento che avviene l'*erasure* il compilatore non conosce il tipo "T";
- Non è consentita la creazione di *Array* o *Enumerazioni* generiche;

La ricercatrice Julia Belyakova, a fronte di precedenti studi comparativi sul supporto linguistico per la programmazione generica (GP), che hanno dimostrato che i linguaggi mainstream orientati agli oggetti (OO) come C # e Java forniscono un supporto più debole per GP rispetto ai linguaggi funzionali come Haskell o SML, ha pubblicato nel 2016 un libro nel quale confronta le differenti tecniche di supporto alla programmazione generica su 7 linguaggi di programmazione orientati agli oggetti [2].

Un'area di applicazione standard dei generici sono le *Collection*, ovvero le raccolte di dati. Ad esempio, il tipo `List[A]` rappresenta un elenco di un dato elemento di tipo A, che può essere scelto liberamente. Sebbene il polimorfismo parametrico consenta di astrarre più tipi, come ad esempio il *type costructor* `List`, questo costrutto non può essere astratto a sua volta. Ad esempio, non si può passare questo *type costructor* come argomento di un altro *type costructor*. Questa restrizione può portare a una duplicazione del codice [6].

1.2.3 Java 6 e Java 7

Con il nome in codice "Mustang", nel Dicembre 2006 la Sun Microsystem rilasciò Java SE 6, portatrice di piccole migliorie; passata un po' inosservata per via della notevole evoluzione già avvenuta due anni prima con Java 5. Java 6 ha portato con sé un significativo miglioramento delle performance, sia del

core Java, sia della API Swing. Sebbene non esistano stime oggettive, molti test informali mostrano per Mustang un aumento delle performance dell'ordine del 50-60 per cento rispetto alle precedente versione *Tiger*. Java 6 segna un miglioramento delle prestazioni della JVM, tra i quali i più interessanti sono l'ottimizzazione delle prestazioni dell'interprete (accelerazione del metodo `System.arraycopy()`, un nuovo Linear Scan Register Allocation Algorithm) e della gestione della sincronizzazione (biased locking, lock coarsening, adaptive spinning) a cui si affianca l'introduzione di nuovi algoritmi per la garbage collection (Parallel Compaction Collector, Concurrent Low Pause Collector) e il miglioramento di quelli esistenti.

Altro punto importante ad esempio è il supporto per la JDBC API versione 4.0. Interessanti novità riguardano il fatto che non è più necessario caricare esplicitamente il driver JDBC utilizzando codice come `Class.forName()` per registrare il driver JDBC. La classe `DriverManager` si fa carico di questa attività localizzando automaticamente un driver adatto all'atto dell'esecuzione del metodo `DriverManager.getConnection()`. JDBC 4.0 fornisce anche un migliore ambiente di sviluppo riducendo al minimo il codice "infrastrutturale" (boiler plate) necessario per accedere a database relazionali. Inoltre sono state aggiunte classi di utilità per migliorare la registrazione del driver JDBC, per la gestione dei data source, e per una migliore gestione della connessione. Con JDBC 4.0, inoltre, gli sviluppatori possono specificare query SQL mediante opportune annotazioni. La versione Java 6 inoltre include nel bundle Apache Derby: si tratta di un database relazionale molto leggero, sviluppato interamente in Java.

Dopo cinque anni, nel Luglio 2011, venne rilasciato Java SE 7. Il nome in codice di questa versione è Dolphin, e la sua gestione è stata caratterizzata da due eventi molto importanti:

- Il primo è avvenuto nel maggio 2007 con il rilascio di OpenJDK kit, implementazione aperta e gratuita del linguaggio di programmazione Java. L'implementazione è stata rilasciata sotto licenza GNU General Public License (GPL), con un'eccezione di collegamento (linking exception), che esenta Java Class Library dai termini della licenza GPL[6];

- Il secondo è l'acquisizione da parte di Oracle Corporation della Sun Microsystem nel Gennaio 2010

Java 7 adottò una serie di variazioni minori del linguaggio, raggruppate nel progetto Coin. È possibile, ad esempio, realizzare costrutti switch con

[6]https://www.gnu.org/licenses/gpl.html

stringhe. È stata poi adottata una gestione automatica delle risorse attraverso la nuova versione del costrutto "try-with-resources". L'istruzione try-with-resources è un blocco `try` che dichiara una o più risorse, ossia oggetti che devono essere opportunamente chiusi dopo che il programma ne ha terminato l'utilizzo. L'istruzione try-with-resources assicura che ogni risorsa venga chiusa correttamente alla fine della dichiarazione. Qualsiasi classe che implementa `java.lang.AutoCloseable`, che comprende tutti gli oggetti che implementano `java.io.Closeable`, può essere come risorsa del costrutto try-with-resource.

C'è stato anche un miglioramento dell'inferenza di tipo per la creazione di istanze generiche. Questa nuova feature rende possibile sostituire gli argomenti di tipo necessari per invocare il costruttore di una classe generica con un insieme vuoto di parametri di tipo (<>) fintanto che il compilatore possa dedurre gli argomenti di tipo dal contesto. La coppia di parentesi angolari aperta-chiusa viene informalmente chiamata "diamante" (<>) chiaramente per la sua forma. Ad esempio la seguente dichiarazione:

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

con la nuova feature può essere sostituita con un insieme vuoto di parametri di tipo (<>):

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

Un altro miglioramento è quello dei warnings e degli errori generati dal compilatore quando si utilizzano parametri formali non reifiable con i metodi varargs. La maggior parte dei tipi parametrizzati, come `ArrayList` e `List`, sono detti tipi non-reifiable per indicare che il tipo non è completamente disponibile a runtime. Al momento della compilazione, il tipo di oggetti non-reifiable subisce un processo chiamato cancellazione di tipo (erasure) durante il quale il compilatore rimuove le informazioni relative ai parametri di tipo e gli argomenti di tipo per assicurare la compatibilità binaria con le versioni Java antecedenti ai generici. Altri piccoli miglioramenti sono stati attuati ad esempio al package `java.util.concurrent` e a Java 2D, ma i cambiamenti più sostanziosi, come i progetti Lambda e Jigsaw, vengono fatti slittare alla versione Java SE 8.

1.2.4 Java 8 e le Lambda-Expressions

Il rilascio della versione 8 è avvenuto nel marzo 2014 con l'obiettivo di spingere il linguaggio Java il più possibile verso la programmazione funzionale. Si tratta principalmente del progetto *Lambda*. Con il loro uso è possibile definire e chiamare un metodo, senza necessariamente dichiarare una classe

che lo contiene. La definizione di un'espressione lambda è formata da tre parti:

- elenco dei parametri formali, input, tra (), separati da virgola. Se c'è un solo input, le parentesi possono essere omesse;
- freccia `->`;
- corpo dell'espressione, costituito da una o più istruzioni, racchiuse tra {}.

Le espressioni Lambda si possono dichiarare in modi diversi:

```
(a,b) -> a+b;

x -> {
       x += 5;
       return x;
     }

Predicate<String> notBlank = s -> s != null && s.length > 0;
```

La prima ha due input e un corpo con una sola istruzione. In questo caso, le parentesi { } vengono omesse e il valore calcolato dall'istruzione viene restituito implicitamente senza bisogno di usare *return*.

La seconda rappresenta un'espressione con un corpo formato da istruzioni multiple, rispetto al caso precedente va notato l'uso delle parentesi { } e del return per indicare quale valore verrà effettivamente restituito.

La terza rappresenta l'uso dell'espressione lambda per definire un predicato. Predicate è una delle nuove interfacce funzionali definite in Java 8. Nel caso riportato sopra viene definita la proprietà di stringa non vuota: se l'input s è una stringa diversa da null e con lunghezza maggiore di 0, il predicato restituirà true, in caso contrario false.

Le espressioni lambda possono essere usate per creare metodi, detti *anonymous method*, oppure possono eseguire chiamate ad altri metodi già definiti. Un'espressione lambda può anche essere formata da un'unica istruzione che esegue la chiamata ad un metodo esistente.

Java 8 segna anche l'introduzione delle *interfacce funzionali*. Le interfacce funzionali sono delle interfacce che definiscono un solo metodo astratto (SAM,

Single Abstract Method) e zero o più metodi di default o metodi statici. Grazie a questa particolarità possono essere implementate tramite un'espressione lambda. Sono definite usando l'annotazione `@FunctionalInterface` , per indicare il carattere particolare dell'interfaccia, ma soprattutto per permettere al compilatore di generare errori se l'interfaccia non soddisfa i requisiti funzionali, ad esempio se contiene più di un metodo astratto. Il metodo delle interfacce funzionali è detto *functional method*. Alcune delle interfacce funzionali introdotte in Java 8 nel package `java.util.functions` sono :

- `Function<T,R>`
- `Supplier<T>`
- `Consumer<T>`
- `Predicate<T,U>`

Alle Collection in Java 8 sono stati aggiunti alcuni metodi tra `stream()`, che restituisce la collezione e permette l'elaborazione della stessa. Uno Stream offre diverse funzioni di ordine superiore, tra cui ad esempio `map` e `reduce`, il primo restituisce un nuovo stream basato sul risultato dell'applicazione della funzione map e su cui poter continuare l'elaborazione. L'operazione di reduce invece combina i risultati della collezione in input in un singolo valore opzionale, in quanto cominciando da un insieme vuoto, l'operazione di reduce non fornirebbe nessun risultato.

1.2.5 Java 9 e il progetto Jigsaw

Java 9 ha introdotto nel codice la completa modularizzazione del codice e dell'ecosistema Java, inizialmente pensata tra gli scopi principali di Java 7 ma rilasciata nella versione Java 9 con il progetto *Jigsaw*. Lo scopo del progetto Jigsaw è progettare e realizzare un sistema modulare standard per Java SE da applicare sia al linguaggio, sia al suo ecosistema in generale a partire dal JDK. Si tratta di implementare nativamente una ristrutturazione del linguaggio e dell'ambiente che integri feature simili a quelle di OSGi (Open Services Gateway initiative)[7].

Il Framework OSGi è un sistema modulare e una piattaforma di servizi per il linguaggio di programmazione Java che implementa un modello a componenti. OSGi, pertanto, si occupa degli aspetti di modularizzazione a tempo di esecuzione: per essere precisi, interviene a partire dal packaging includendo il

[7]https://it.wikipedia.org/wiki/OSGi

deployment e l'esecuzione.

L'obiettivo fondamentale del progetto Jigsaw consiste nell'introdurre una serie di nuovi meccanismi, al livello di linguaggio di programmazione, da utilizzarsi per la modularizzazione di grandi sistemi, applicati a partire dallo stesso JDK. Poiché si tratta di nuove caratteristiche al livello di linguaggio di programmazione, queste sono disponibili ai programmatori per organizzare i propri progetti. Si tratta di un cambiamento, per molti versi radicale, che potrà avere una notevole influenza sul processo di sviluppo e di deployment dei progetti Java, specie per quelli di medio-grandi dimensioni. L'implementazione del progetto Jigsaw permette di scomporre sia la piattaforma Java SE, sia le sue implementazioni in una serie di componenti indipendenti che possono essere assemblati in un'installazione personalizzata contenente esclusivamente le funzionalità richieste dall'applicazione. Questa feature va a risolvere una lacuna sempre più evidente del mondo Java: il continuo crescere del Java Runtime. Prima di Java 9 non era possibile creare delle configurazioni su misura dell'ambiente JRE. La distribuzione includeva necessariamente tutte le librerie, indipendentemente dal loro utilizzo. Si tratta di una lacuna che ha un impatto in ambienti di piccole dimensioni, caratterizzati dalla presenza ridotta di risorse. Con Java 9 i vari moduli sono dotati di una configurazione che permette di dichiarare esplicitamente le dipendenze da altri moduli. Queste configurazioni giocano un ruolo importante nell'arco dell'intero processo di sviluppo del software, a partire dalla compilazione, passando per il build e terminando al tempo di esecuzione. Pertanto, si finisce per avere a disposizione un meccanismo di controllo in grado di far fallire immediatamente un sistema qualora una o più dipendenze non siano soddisfatte o risultino in conflitto. Un obiettivo fondamentale del progetto Jigsaw consiste nell'introdurre un forte incapsulamento al livello di modulo. In particolare, si ha ora la possibilità, per ogni modulo, di suddividere i package in due gruppi: quelli visibili da parte di altri moduli e quelli privati, visibili solo al suo interno. I modificatori di visibilità Java si sono dimostrati un ottimo supporto per l'implementazione dell'incapsulamento tra classi dello stesso package. Tuttavia, l'unica visibilità permessa per superare i confini dei package è quella pubblica. Questo ha finito per indebolire il concetto di incapsulamento. Il forte incapsulamento dei moduli ha anche importanti ripercussioni positive sia sulla sicurezza, sia sulla manutenibilità del codice. Per quanto concerne il primo aspetto, è ora possibile far sì che il codice critico sia veramente nascosto/protetto in modo tale da non essere assolutamente disponibile ad altro codice che non dovrebbe utilizzarlo. Inoltre, anche la manutenibilità del codice migliora, perché le API pubbliche dei moduli possono essere mantenute leggere e concise.

Nello specifico, le caratteristiche principali di un modulo sono:

- Un modulo può contenere file di classi Java, risorse, librerie native e file di configurazione;
- Un modulo deve avere un nome univoco;
- Un modulo può dipendere da altri moduli attraverso il suo nome: l'unico modulo che non dipende da nessuno è il modulo base, mentre tutti gli altri devono dipendere almeno da questo;
- Un modulo può esportare (tutti) i tipi pubblici presenti in uno o più package, definiti *package API*, che li contengono: in questo modo, tali tipi diventano disponibili per l'uso da parte del codice presente in altri moduli che dipendono da esso e anche da codice non organizzato in un modulo, ossia non ancora modularizzato.
- Attraverso il nome, un modulo può limitare l'insieme dei moduli a cui vengono esportati i tipi pubblici definiti nei propri package API;
- Un modulo può anche riesportare a sua volta tutti i tipi pubblici esportati da uno o più moduli da cui esso dipende.

Un esempio del modulo `java.sql`:

```
<module>
  <!-- The name of this module -->
  <name>java.sql</name>

  <!-- Every module depends upon java.base -->
  <depend>java.base</depend>

  <!-- This module depends upon the java.logging and java.xml
       modules, and re-exports their exported API packages -->
  <depend re-exports=\true">java.logging</depend>
  <depend re-exports=\true">java.xml</depend>

  <!-- This module exports the java.sql, javax.sql, and
       javax.transaction.xa packages to any other module -->
  <export><name>java.sql</name></export>
  <export><name>javax.sql</name></export>
  <export><name>javax.transaction.xa</name></export>
</module>
```

le parti di un modulo sono:

- `name`: la dichiarazione del nome univoco del modulo;
- `depend`: Permette di specificare la lista dei moduli da cui dipende quello oggetto di definizione. Tutti i moduli, come illustrato poco sopra, dipendono almeno da quello base: `java.base`;
- `Depend/re-export`: Il tag depend permette anche di riesportare l'API di qualsiasi altro modulo da cui quello attuale dipende: a tal fine è utilizzato l'attributo XML re-export. Ciò al fine di supportare processi di refactoring e, in particolare, fornisce la possibilità di poter dividere e unire i moduli senza rompere le dipendenze giacché quelli originali possono continuare a essere esportati;
- `export`: Permette di definire i package esportati dal modulo in questione, quindi solo i tipi pubblici presenti in tali package saranno visibili ad altri moduli; Per poter poi accedere a tali tipi, eventuali moduli "client" dovranno dichiarare esplicitamente la dipendenza dal modulo.

1.3 Altri linguaggi per JVM

La Java Virtual Machine, componente essenziale per l'ambiente Java che ne assicura la portabilità, funziona traducendo il bytecode. Quindi un linguaggio, per essere virtualizzato e tradotto dalla JVM, deve solamente trasformare il suo codice in bytecode in fase di compilazione. Java non è l'unico in ambito dei linguaggi di programmazione che produce bytecode; tra i più famosi abbiamo Scala, Clojure, Kotlin e Groovy

1.3.1 Scala

Scala è stato progettato e sviluppato a partire dal 2001 da Martin Odersky come linguaggio per interoperare con la piattaforma Java 2 Runtime Environment (JRE). Scala utilizza lo stesso modello di compilazione utilizzato da Java, permettendo così l'accesso a molte librerie sviluppate in questo ambiente. Scala è un linguaggio a pardigma ibrido funzionale. Il tipo e il comportamento degli oggetti è descritto da classi e *traits*. Scala supporta l'ereditarietà multipla attraverso traits e mixin. In Scala ogni funzione è un valore; Fornisce una sintassi leggera per la definizione di funzioni anonime (anonymous functions), supporta le funzioni di alto livello (higher-order functions), permette di nidificare le funzioni e supporta la tecnica di *currying*. Il type system di Scala supporta:

- generic classes,

- overload e definizione di operatori,
- variance annotations,
- type-level programming,
- upper and lower type bounds,
- impliciti
- inner classes and abstract types as object members,
- tipi algebrici,
- compound types,
- pattern matching
- explicitly typed self references,
- repl,
- views,
- argomenti di default e named,
- polymorphic methods e
- higher kinded types[8].

1.3.2 Clojure

Clojure[9] è un dialetto del linguaggio di programmazione Lisp sviluppato da Rich Hickey allo scopo di ottenere un moderno linguaggio Lisp in grado di supportare il paradigma di programmazione funzionale, di sfruttare una piattaforma software già esistente e di gestire facilmente la concorrenza. Clojure è interoperabile con Java. Come la maggior parte degli altri linguaggi Lisp, la sintassi di Clojure è basata su *S-expressions* che vengono prima trasformate in strutture dati da un lettore (reader) prima di essere compilate. Il *reader* di Clojure supporta la *literal syntax* per le strutture `map`, `sets` e `vectors` oltre che alle liste, che sono compilate direttamente dalle strutture appena citate. Clojure è un *Lisp-1* e non è pensato per avere codice compatibile con altri linguaggi Lisp. Clojure supporta le funzioni come *first-class objects*, un

[8] https://www.atlassian.com/blog/archives/scala-types-of-a-higher-kind
[9] https://clojure.org/

read-eval-print loop (REPL) e un *macro system*. Clojure inoltre supporta *multimethods*[10] e per le astrazioni di tipo interfaccia ha un protocollo basato sul polimorfismo e un type-system che utilizza i record, fornendo prestazioni e polimorfismo dinamico. Come linguaggio funzionale, Clojure pone enfasi sulle ricorsioni e sulle *high-order functions* invece del *side-effect-based looping*.

1.3.3 Kotlin

Kotlin è un linguaggio di programmazione general purpose, multi-paradigma, open source sviluppato nel 2011 dall'azienda di software JetBrains ed ispirato ad altri linguaggi di programmazione tra i quali Scala e lo stesso Java, mentre ulteriori spunti sintattici sono stati presi da linguaggi classici, come ad esempio il Pascal. Kotlin è un linguaggio a tipizzazione statica e forte, è orientato ad oggetti pur permettendo di esprimere alcune computazioni con uno stile funzionale. Tra le principali features di Kotlin ci sono:

- extension functions,
- multi-piattaforma target (Java, JS, Native),
- high-order functions,
- interoperability with Java,
- null safe-types,
- smart cast,
- overload degli operatori,
- default and named arguments,
- tipi algebrici,
- data classes.

1.3.4 Groovy

Groovy è un linguaggio di programmazione ad oggetti per la Piattaforma Java alternativo al linguaggio Java creato da James Strachan nel 2003. Può essere visto come linguaggio di scripting per la Piattaforma Java, presenta caratteristiche simili a quelle di Python, Ruby, Perl, e Smalltalk. Groovy usa una sintassi simile a quella di Java, viene compilato dinamicamente in

[10] https://clojure.org/reference/multimethods

bytecode per la Java Virtual Machine, ed interagisce in modo trasparente con altro codice Java e con le librerie esistenti. Groovy può anche venire usato come linguaggio di scripting dinamico. Tra le funzionalità principali di Groovy ci sono:

- static and dynamic typing (con la keyword `def`)
- overloading degli operatori
- sintassi nativa per le liste e gli array associativi (map)
- supporto nativo per le *regular expressions*
- polymorphic iteration

Groovy fornisce supporto nativo per vari linguaggi di markup, come *XML* e *HTML* realizzati attraverso una sintassi *DOM* (Document Object Model sintax). A differenza di Java, un file di codice sorgente Groovy può essere eseguito come uno script (non compilato), se contiene codice al di fuori di qualsiasi definizione di classe, se si tratta di una classe con un metodo principale (`main()`) o se è `Runnable` o `GroovyTestCase`.

Capitolo 2

Il linguaggio Ceylon

Ceylon è un linguaggio di programmazione orientato agli oggetti, general-purpose, a forte tipizzazione statica, sviluppato da Gavin King (Red Hat),che può essere anche compilato in *Javascript* che fece la sua prima apparizione nel Gennaio 2011. King sviluppò Ceylon ispirandosi a molti linguaggi di programmazione operanti sulla Java Virtual Machine (JVM), anche se tuttavia il principale modello di ispirazione, soprattutto a livello sintattico, è stato il linguaggio Java.

2.1 Motivazioni

Quando si sparse la voce che King stava lavorando allo sviluppo di un nuovo linguaggio di programmazione orientato agli oggetti, i media si affrettarono a saltare sull'idea che Red Hat stesse preparando un "killer Java". King, a una presentazione del progetto su InfoQ ad aprile 2011[1] si preoccupò subito di smentire questi pensieri spiegando che Ceylon è un linguaggio profondamente influenzato da Java, progettato da utenti Java e che quindi non nasce con l'intento di sostituirlo, ma con l'intento di migliorarlo; Ceylon, come spiega il suo creatore, cerca di eliminare la verbosità e la complessità della sintassi del codice Java e cerca di offrirne una versione più leggibile e flessibile, pur mantenendone simile la struttura. King sostiene che molti elementi della sintassi Java, come ad esempio i `getter/setter`, i *costruttori*, il *cast* oppure l'`istance of` abbiano sintassi verbose e inclini agli errori. Per questo motivo il principale obiettivo del progetto Ceylon è realizzare un linguaggio leggibile, semplice da programmare e da capire, eliminando tutto ciò che rende il codice meno comprensibile. Caratteristiche di Ceylon, come la robusta modularità interna o le

[1] https://www.infoq.com/news/2011/04/ceylon

funzioni di alto livello, le cosiddette *high-order function*, erano caratteristiche uniche di Ceylon prima dell'introduzione di Java 8.

2.2 Elementi base del linguaggio

2.2.1 Classi

Nel linguaggio ceylon una classe è letteralmente un "tipo" (type); le caratteristiche più importanti di un tipo sono:

- I metodi (member functions),
- gli attributi (member values), e
- le classi (member classes).

Metodi, attributi e classi sono polimorfici, ovvero le loro definizioni possono essere raffinate da un sottotipo. Il primo carattere di un identificatore è significativo:

- I nomi dei tipi (interfaccia, classe e parametro) devono iniziare con una lettera maiuscola;
- I nomi di funzioni e valori iniziano con una lettera minuscola iniziale o underscore.

Queste due dichiarazioni ad esempio generano un errore:

```
class hello() { ... }

String Name = ....
```

Nel codice seguente c'è l'esempio di una semplice classe rappresentante le coordinate polari:

```
class Polar(Float angle, Float radius) {

    shared Polar rotate(Float rotation)
           => Polar(angle+rotation, radius);

    shared Polar dilate(Float dilation)
           => Polar(angle, radius*dilation);

    shared String description
```

```
            = "(``radius``,``angle``)";

}
```

La classe precedente possiede due parametri, due metodi e un attributo. I parametri utilizzati per creare un'istanza di una classe vengono specificati come parte della dichiarazione della classe stessa, subito dopo il nome della classe.

I parametri di una classe possono essere usati ovunque all'interno del corpo della classe. In Ceylon, non è necessario definire esplicitamente i membri della classe per mantenere i valori dei parametri. Invece, come nell'esempio, possiamo accedere ai due parametri della classe (`angle` e `radius`) direttamente dai metodi `rotate()` e `dilate()`, e anche dall'espressione che specifica il valore dell'attributo `description`. In Ceylon non è presente la parola chiave *new*, ma è necessario "invocare la classe" scrivendo `Polar(angle, radius)`. La parola chiave *shared* specifica il livello di accessibilità del *type*. Ceylon non fa distinzione tra visibilità *public*, *protected* o di "default" come fa il linguaggio Java, distingue invece:

- elementi del programma che sono visibili solo all'interno dell'ambito in cui sono definiti, e
- elementi del programma che sono visibili ovunque sia visibile l'elemento a cui appartengono (un *type*, un *package* o un modulo)

Di default, i membri di una classe sono nascosti dal codice al di fuori del corpo della classe. Dichiarando un membro *shared*, esso viene reso visibile in qualsiasi parte del codice a cui la classe stessa è visibile. Naturalmente, una classe stessa potrebbe essere nascosta da un altro codice. Per impostazione predefinita, una classe è nascosta dal codice esterno al package in cui è definita la classe. Dichiarare una classe *shared* la rende visibile al codice in cui è visibile il package che contiene la classe. I packages, infine, di default sono nascosti al codice esterno al modulo a cui il package appartiene. Solo i package esplicitamente definiti *shared* sono visibili ad altri moduli.

Un attributo è un membro di una classe che ne rappresenta lo stato. Molto spesso, lo stato di una classe deriva dagli argomenti usati per istanziarla. Pertanto, esiste una stretta relazione tra parametri di classe e attributi. Se vogliamo fare in modo che i parametri di una classe siano visibili al resto del codice dovremo definirli con la keyword `shared`. Essendo una pratica molto comune, Ceylon fornisce una sintassi semplificata per questo:

```
class Polar(angle, radius) {
```

```
    shared Float angle;
    shared Float radius;

    shared Polar rotate(Float rotation)
            => Polar(angle+rotation, radius);

    shared Polar dilate(Float dilation)
            => Polar(angle, radius*dilation);

    shared String description
            = "(``radius``,``angle``)";

}
```

2.2.2 Variabili

In Ceylon di default le variabili (values) sono assegnabili una sola volta e per creare variabili mutabili, ovvero variabili alle quali è possibile riassegnare il valore, è necessaria la keyword `variable`.

```
String bye = "Adios";      //a value
variable Integer count = 0; //a variable

bye = "Adieu"; //compile error
count = 1;     //allowed
```

Anche un valore che non è una variabile in questo senso, può ancora essere "variabile" nel senso che il suo valore può variare tra diverse esecuzioni del programma o tra contesti all'interno di una singola esecuzione del programma:

```
String name { return firstName + " " + lastName; }
```

Se il valore di `firstName` o `lastName` varia, anche il valore di `name` varia. In Java, un campo di una classe si distingue abbastanza facilmente da una variabile locale o da un parametro di un costruttore. I campi sono dichiarati direttamente nel corpo di una classe Java, mentre le variabili locali sono dichiarate all'interno del corpo del costruttore. Una variabile locale di un costruttore non "sopravvive" al di fuori della chiamata di esso. Un campo invece, "vive" fino a quando l'oggetto a cui appartiene non è distrutto. Questa distinzione è molto meno marcata in Ceylon.

```
class Counter() {
```

```
    variable Integer count = 0;
}
```

In questo esempio `count` è una variabile locale al "blocco di codice" dell'inizializzazione di `Counter`.

```
class Counter() {
    shared variable Integer count = 0;
}
```

In questo esempio invece, `count` è dichiarato *shared* perciò è visibile a tutta la classe e anche all'esterno di essa.

2.2.3 Funzioni

```
void trivialBlock() {
    /* method block: statements */
}
```

Questo è un esempio di come si dichiara una funzione. Alternativamente a questo, si può dichiarare la funzione usando un operatore denominato *fat arrow*: `=>` in questo modo:

```
void trivialSpecifier() => anotherMethod();
```

Il *fat arrow* ci permette di abbreviare la sintassi e avere un codice più compatto. Una funzione deve sempre specificare il valore di ritorno o in assenza di esso, la keyword `void` per le funzioni che non hanno uno specifico valore di ritorno. Il type system considera una funzione di tipo *void* allo stesso modo di una funzione che ha come valore di ritorno il tipo `Anything`. La classe `Anything` è il supertipo astratto di ogni classe e può definire un valore di tipo `Object` oppure di tipo *null*. Perciò come nel seguente esempio, una funzione di tipo *void* restituisce sempre un *null*:

```
class Top() {
    shared default void m() {}
}
class Sub() extends Top() {
    shared actual Boolean m() => true
}
void example() {
    Anything topM = Top().m(); // topM is null
    Boolean subM = Sub().m(); //subM is true
```

```
}
```

Grazie alla tecnica chiamata *type inference* non è necessario sempre specificare il tipo di una funzione, il compilatore ne assegna il tipo attraverso la keyword `function`:

```
class C() {
    function f() => 0; //inferred type Integer
}
```

Una dichiarazione di funzione può avere subito dopo il nome, una lista di "tipi di parametri"(type parameters) racchiusi in parentesi angolari (<>):

```
void generic<Foo, Bar>(){
    /* method block: statements
       type parameter Foo and Bar are treated as a type */
}
```

Il "blocco" di codice nel quale è racchiusa una funzione si presenta come in Java:

```
String fun(Boolean bool) {
    if (bool) {
        return "hello";
    }
    // error: missing return
}
```

2.2.4 Interfacce

Le interfacce sono strutture che, al contrario delle classi:

- non possono contenere riferimenti ad altri oggetti,
- non definiscono la logica di inizializzazione, e
- non possono essere istanziate direttamente.

Le interfacce supportano l'ereditarietà multipla.

```
interface Trivial {
    /* declarations of interface members */
}
```

Questo è l'esempio di una dichiarazione di un'interfaccia. Un'interfaccia può anche avere una lista di *type parameters* racchiusi nelle parentesi angolari (<>) subito dopo il nome dell'interfaccia; In quel caso si parla di *generic interface*:

```
interface Generic<Foo, Bar> {
    /* declarations of interface members
       type parameters Foo and Bar are treated as a types */
}
```

Questo tipo di interfaccia può anche avere clausole per i *type parameters* che possiede , come nel caso seguente:

```
interface Constrained<Foo, Bar>
        given Foo satisfies Baz1&Baz2
        given Bar of Gee1|Gee2 {
    /* declarations of interface members
       type parameters Foo and Bar treated as a types */
}
```

Per fare in modo che una classe o anche un'altra interfaccia implementino un'interfaccia è necessaria la keyword `satisfied`, al contrario di Java che usa la keyword `implements`

```
interface Constrained<Foo, Bar>
        given Foo satisfies Baz1&Baz2
        given Bar of Gee1|Gee2 {
    /* declarations of interface members
       type parameters Foo and Bar treated as a types */
}
```

Anche le interfacce possono essere dichiarate *shared* per essere viste ovunque il *package* che le contiene è visibile.

2.2.5 Top type e bottom type

In Ceylon la classe `Anything` è la classe considerata il "supertipo" di ogni tipo. Questo vuol dire che ogni tipo in Ceylon ha `Anything` come supertipo; perciò ogni riferimento è assegnabile a `Anything`. Si può pensare a `Anything` come l'unione di tutti i tipi; un'istanza di `Anything` non si può usare a meno che non venga ristretto a un tipo più specifico.

La classe invece che risulta "sottotipo" di tutte le classi in Ceylon è `Nothing`, perciò ogni tipo in Ceylon ha `Nothing` come sottotipo. Si può pensare a `Nothing` come l'intersezione di tutti i tipi, essendo ciò, non può avere istanze.

Qualsiasi valore che cerca di restituire `Nothing` non può restituirlo normalmente ma deve:

- lanciare un'eccezione o
- non ritornarlo, ad esempio cadendo in un loop infinito o stoppando la virtual machine.

2.3 Features principali

2.3.1 Inheritance and refinement

Ereditarietà

L'ereditarietà è uno dei due modi, insieme ai generici, con cui Ceylon permette di astrarre i tipi. La feature di Ceylon che permette l'ereditarietà multipla si chiama "Mixin Inheritance". Il modello di ereditarietà di Java non supporta l'ereditarietà multipla, dato che un'interfaccia non può definire la concreta implementazione di un membro, ma può farlo a partire da Java 8 attraverso i *default methods*. Le interfacce in Ceylon sono più flessibili:

- Un'interfaccia può definire metodi concreti, *getters*, *setters*, ma
- non può definire dipendenze o logica di inizializzazione.

Proibire i riferimenti e la logica di inizializzazione rende le interfacce completamente prive di stato. Ecco un esempio dell'interfaccia `Writer` per Ceylon:

```
interface Writer {

    shared formal Formatter formatter;

    shared formal void write(String string);

    shared void writeLine(String string) {
        write(string);
        write("\n");
    }

    shared void writeFormattedLine(String format, Object* args) {
        writeLine( formatter.format(format, args) );
    }

}
```

Nell'esempio il membro `formatter` e il metodo `write` vengono dichiarati con la keyword `formal`. *Formal* è l'equivalente in Java di *abstract* e dichiara un elemento che non ha la concreta implementazione. Poiché deve essere visibile al di fuori del codice nel quale è stato dichiarato, un membro definito *formal*, deve essere anche *shared*. Solo le classi astratte e le interfacce possono avere membri di tipo *formal*. Questa invece una concreta possibile implementazione dell'interfaccia:

```
class ConsoleWriter() satisfies Writer {
    formatter = StringFormatter();
    write(String string) => print(string);
}
```

La keyword per fare in modo che un'interfaccia estenda un'altra interfaccia oppure che una classe estenda un'interfaccia è *satisfies*. In java invece è *implements*. Al contrario di *implements*, la dichiarazione *satisfies* non specifica argomenti, poiché le interfacce non hanno parametri o logica di inizializzazione. Inoltre, la keyword `satisfies` può includere più di un'interfaccia. L'approccio di Ceylon alle interfacce elimina un modello comune in Java in cui una classe astratta separata definisce un'implementazione predefinita di alcuni membri di un'interfaccia. In Ceylon, le implementazioni predefinite possono essere specificate dall'interfaccia stessa. Inoltre è possibile aggiungere un nuovo membro a un'interfaccia senza rompere le implementazioni esistenti dell'interfaccia.

Refinement

Un'altra tecnica in Ceylon è quella chiamata *refinement*, ovvero l'equivalente in Java della tecnica di *override*. Ecco un esempio del *refactoring* della classe `Polar` in due classi , con due differenti implementazioni del membro `description`. Il seguente codice rappresenta la superclasse:

```
"A polar coordinate"
class Polar(Float angle, Float radius) {

    shared Polar rotate(Float rotation)
           => Polar(angle+rotation, radius);

    shared Polar dilate(Float dilation)
           => Polar(angle, radius*dilation);

    "The default description"
    shared default String description
           => "(``radius``,``angle``)";
```

```
}
```

In Ceylon un attributo o metodo sul quale potrebbe avvenire la tecnica di override deve essere definito con la keyword `default` come suggerito in effective Java "design for inheritance or prohibit it". Una possibile sottoclasse che estende la classe polar potrebbe essere questa:

```
"A polar coordinate with a label"
class LabeledPolar(Float angle, Float radius, String label)
        extends Polar(angle, radius) {

    "The labeled description"
    shared actual String description
            => label + "-" + super.description;

}
```

Ceylon necessita che un attributo o metodo che applica l'override di una superclasse lo dichiari con la keyword `actual`.

2.3.2 Generics

Ceylon supporta genericità a livello di classi, interfacce, *type alias*, e funzioni. I *type parameters*, ovvero i parametri generici, come in Java, vengono racchiusi fra parentesi angolari e sono dichiarati prima dei parametri ordinari. Ogni type parameter ha un nome e una *varianza*:

- un type parameter *covariante* è indicato attraverso la keyword `out` come negli esempi precedenti
- un type parameter *controvariante* è indicato attraverso la keyword `in`
- se nessun modificatore è specificato, il type parameter è invariante

```
shared interface Iterator<out Element> { ... }
```

```
shared class Singleton<out Element>(Element element)
        extends Object(){...}
```

Per convenzione, in Ceylon vengono usati nomi più significativi per i type parameters, anziché le singole lettere come in Java. Al contrario di Java, in Ceylon è necessario specificare sempre il tipo degli argomenti in una dichiarazione. Ceylon non possiede *raw types* perciò il seguente esempio non viene compilato:

```
value strings = {"hello", "world"};
Iterator it = strings.iterator(); //error: missing type argument to
    parameter Element of Iterable
```

Se invece viene specificato il tipo, "`String`" nel precedente esempio, il codice verrà compilato:

```
value strings = {"hello", "world"};
Iterator<String> it = strings.iterator();
```

Nella maggior parte di invocazioni di metodi o istanziazioni di classi non è necessario esplicitare il tipo di argomento come in questo caso:

```
Array<String> strings = Array<String> { "Hello", "World" };
```

Ceylon riconosce il tipo di argomento a partire dal tipo degli argomenti ordinari:

```
value strings = Array { "Hello", "World" }; // type Array<String>
value things = interleave(strings, 0..2); // type {String|Integer*}
```

Ceylon cerca di eliminare i "caratteri jolly", o *wildcard*. La covarianza in Ceylon funziona in modo diverso, come mostrato nei seguenti esempi:

```
interface Collection<Element> {
    shared formal Iterator<Element> iterator();
    shared formal void add(Element x);
}
```

Questo è un esempio di una possibile implementazione dell'interfaccia `Collection`, e viene considerato che `Geeks` sia una collezione di `Person` e che sia anche un sottotipo di `Person`. Date per vere queste due considerazioni il seguenti esempi di codice dovrebbero funzionare:

```
Collection<Geek> geeks = ... ;
Collection<Person> people = geeks; //compiler error
for (person in people) { ... }
```

```
Collection<Geek> geeks = ... ;
Collection<Person> people = geeks; //compiler error
people.add( Person("Fonzie") );
```

Anche se il codice mostrato, a livello nominale, sembra perfettamente ragionevole, sia in Java che in Ceylon, produce un errore a tempo di compilazione

nella seconda riga, dove la collezione `geeks` è assegnata alla collezione `people`. Se andasse a buon fine, verrebbe permesso a `people` di aggiungere un elemento sbagliato alla collezione. Nello specifico, `Collection` è *invariante* in `Element`. Oppure si può affermare che entrambe le Collezioni:

- "producono" `Element` con il metodo `iterator()` e
- "consumano" `Element` con il metodo `add()`

In Java, si potrebbero usare le wildcard per ottenere:

- un tipo covariante, per esempio `Collection<? extends Person>` che "produce" solamente `Element` oppure
- un tipo controvariante, per esempio `Collection<? super Geek>` che "consuma" solamente `Element`

Ceylon possiede anche le wildcard, ma sono principalmente usati per l'interoperabilità con Java; nel codice puro Ceylon viene usato un approccio differente per la covarianza e controvarianza. Il codice di `Collection` potrebbe essere rifattorizzato nelle due interfacce pure `Consumer` e `Producer`.

- L'annotazione `out` specifica che `Producer` è covariante in `Output`; produce istanze di `Output` ma non consuma mai istanze di `Output`.
- L'annotazione `in` specifica che `Consumer` è controvariante in `Input`; consuma istanze di `Input` ma non produce mai istanze di `Input`.

A fronte di queste due nuove interfacce, il compilatore Ceylon si assicura che le regole di varianza siano soddisfatte; Se si tenta di dichiarare un metodo `add()` in `Producer` oppure di dichiarare un metodo `iterate()` in `Consumer` viene generato un errore di compilazione. Inoltre:

- Poiché `Producer` è covariante nel suo parametro `Output`, e poiché `Geek` è un sottotipo di `Person`, Ceylon consente di assegnare `Producer <Geek>` a `Producer <Person>`.
- Inoltre, poiché `Consumer` è controvariante nel suo parametro `Input`, e poiché `Geek` è un sottotipo di `Person`, Ceylon permette di assegnare `<Person>` a `Consumer <Geek>`.

Possiamo definire l'interfaccia `Collection` come mix di `Producer` e `Consumer` in questo modo:

```
interface Collection<Element>
       satisfies Producer<Element> & Consumer<Element> {}
```

Così facendo `Collection` rimane invariante rispetto ad `Element`; Se si provasse ad aggiungere un'annotazione di varianza ad `Element` risulterebbe un errore di compilazione, perché l'annotazione andrebbe in contrasto con le annotazioni di varianza in `Producer` e `Consumer`. Ora il seguente codice compila senza errori:

```
Collection<Geek> geeks = ... ;
Producer<Person> people = geeks;
for (person in people) { ... }
```

In ceylon è possibile utilizzare le wildcard di Java, attraverso una sintassi di questo genere:

```
void java(Map<in String, out Widget> map) { ... }
```

Java Generics

```
package it.html.java.generics;

public class Bottle<T> {

    private T content;

    public Bottle(T t){
        content=t;
    }

    public T getContent() {
        return content;
    }

}
```

Il precedente è un esempio di una classe generica in Java `Bottle` che rappresenta una bottiglia nella quale il contenuto è un generico. Per rappresentare il parametro generico usiamo l'annotazione "simbolo minore – carattere – simbolo maggiore". E' possibile specificare più parametri generici nella dichiarazione della classe come questo esempio:

```
public class GenericClass<T,E,K> {...}
```

Per garantire compatibilità con le versioni precedenti alla J2SE 5, l'implementazione in Java dei generici viene realizzata attraverso la tecnica nota con il nome di erasure. Quando viene compilato il codice sorgente, tutti i para-

metri di tipo vengono sostituiti dal loro tipo limite, che in mancanza di una specifica diversa è `Object`. Avviene quindi il cast appropriato da parte del compilatore che si preoccupa di gestire la compatibilità del tipo. In sostanza i generici sono semplicemente un meccanismo del codice sorgente e non esistono parametri di tipi in fase di esecuzione.

Scala Generics

Anche Scala possiede i parametri generici, particolarmente utili per le `Collection`. una classe generica racchiude il parametro dentro le parentesi quadre. Una convenzione di Scala è usare il parametro `A` come tipo di parametro, sebbene sia possibile utilizzare qualsiasi nome

```
class Stack[A] {
  private var elements: List[A] = Nil
  }
}
```

Nell'esempio, l'implementazione di `Stack` può avere qualsiasi parametro di tipo `A`. La lista in nella seconda riga di codice: `var elements: List[A] = Nil` può contenere solamente elementi di tipo `A`

```
val stack = new Stack[Int]
stack.push(1)
stack.push(2)
println(stack.pop) // prints 2
println(stack.pop) // prints 1
```

In questo esempio, l'istanza `stack` può contenere solo oggetti di tipo `Int`, tuttavia, se l'argomento possiede dei sottotipi, possono essere passati in ingresso:

```
class Fruit
class Apple extends Fruit
class Banana extends Fruit

val stack = new Stack[Fruit]
val apple = new Apple
val banana = new Banana

stack.push(apple)
stack.push(banana)
```

Le classi `Apple` e `Banana` estendono entrambe `Fruit` perciò, istanze come `apple` e `banana` possono essere inserite all'interno di `stack` della classe `Fruit`.

Scala ha co(ntro)varianza a livello di tipo come Ceylon.

2.3.3 Type alias and type inference

Le features *type aliases* e *type inference* aiutano a ridurre la verbosità del codice di un linguaggio "a tipizzazione statica". Ceylon fornisce un nome più breve o più significativo a una classe esistente o un tipo interfaccia, soprattutto se la classe o l'interfaccia è un tipo parametrizzato. Per definire un *alias* per una classe o un'interfaccia usiamo la freccia, chiamata *fat arrow*, ad esempio:

```
interface People => Set<Person>;
```

Un *alias* di una classe deve dichiarare i suoi parametri formali:

```
class People({Person*} people) => ArrayList<Person>(people);
```

Per creare invece un alias per un tipo di "Unione" o "Intersezione" è necessaria la keyword `alias`:

```
alias Num => Float|Integer;
```

Un *type alias* riduce la verbosità del codice, perché invece di riscrivere sempre lo stesso tipo generico , come ad esempio `Set<Person>`, possiamo usare un *alias*, come `People`. Un *type alias* di "alto livello" o un *type alias* appartenente a una classe o interfaccia può essere dichiarato *shared*:

```
shared interface People => Set<Person>;
```

Un *type alias* può essere nidificato all'interno di una classe o interfaccia. Nel caso di un *alias* di una classe, viene considerato un membro a tutti gli effetti:

```
class BufferedReader(Reader reader)
        satisfies Reader {
    shared default class Buffer()
        => MutableList<Character>();
}
```

Se viene dichiarato con la keyword `default` è possibile anche applicare l'*override* su di esso, sia da un *alias* interno di una sottoclasse della classe originale, come nel seguente caso:

```
class BufferedFileReader(File file)
        extends BufferedReader(FileReader(file)) {
    shared actual class Buffer()
        => MutableLinkedList<Character>();
```

```
}
```

sia da una sottoclasse interna della classe originale:

```
class BufferedFileReader(File file)
        extends BufferedReader(FileReader(file)) {
    shared actual class Buffer()
            extends super.Buffer() {
        ...
    }
    ...
}
```

Il "tipo" di ogni dichiarazione, se specificato, in generale rende il codice più facile da leggere e capire. Tuttavia Ceylon permette di inferire il tipo di una variabile locale o il tipo di ritorno di un metodo locale. E' necessario scrivere le keyword `value` per un valore locale, o *function* per una funzione locale all'inizio della dichiarazione:

```
value polar = Polar(pi, 2.0);
value operators = { "+", "-", "*", "/" };
function add(Integer x, Integer y) => x+y;
```

Però ci sono delle restrizioni su questa feature. Non si può usare *value* o *function*:

- per le dichiarazioni *shared*,
- per le dichiarazioni *formal*,
- per dichiarare un parametro.

Le regole di inferenza per Ceylon sono quindi abbastanza semplici:

- Il tipo inferito di un valore dichiarato attraverso *value* è il tipo di espressione assegnatagli dopo l'operatore di assegnamento (=).
- Il tipo di ritorno inferito di un metodo dichiarato attraverso *function* è l'unione dei tipi di ritorno delle espressioni che appaiono nel metodo, con la keyword `return` (oppure `Nothing` se il metodo non ha tipi di ritorno)

In questo esempio:

```
function parseIntegerOrFloat(String text) {
    if ('.' in text) {
        return parseFloat(text);
```

```
    }
    else {
        return parseInteger(text);
    }
}
```

il tipo inferito del metodo `parseIntegerOrFloat()` è `Integer|Float?` perchè `parseInteger()` ritorna un valore di tipo `Integer?` e `parseFloat()` ritorna un valore di tipo `Float?`

Ereditarietà in Scala

Le classi in Scala sono estendibili. Il meccanismo delle sottoclassi permette di specializzare una classe dalla quale verranno ereditati tutti i membri ed è possibile anche definire metodi aggiuntivi

```
class Point(xc: Int, yc: Int) {
  val x: Int = xc
  val y: Int = yc
  def move(dx: Int, dy: Int): Point =
    new Point(x + dx, y + dy)
}
class ColorPoint(u: Int, v: Int, c: String) extends Point(u, v) {
  val color: String = c
  def compareWith(pt: ColorPoint): Boolean =
    (pt.x == x) && (pt.y == y) && (pt.color == color)
  override def move(dx: Int, dy: Int): ColorPoint =
    new ColorPoint(x + dy, y + dy, color)
}
```

In questo esempio viene definita la classe `Point` e successivamente la classe `ColorPoint` che estende `Point`. Le conseguenze sono:

- La classe `ColorPoint` eredita tutti i membri dalla superclasse `Point`, nello specifico eredità i valori `x,y` e il metodo `move`
- La sottoclasse `ColorPoint` aggiunge un nuovo metodo `compareWith` in aggiunta ai metodi ereditati
- Scala permette l'`override`; In questo caso il metodo `move` della classe `point` subisce l'override nella sottoclasse `ColorPoint`. Questo rende il metodo `move` inaccessibile dagli oggetti di `ColorPoint`. All'interno della classe `ColorPoint` però possiamo accedere al metodo `move` con

una "super chiamata": `super.move()`. Nell'esempio precedente utilizziamo questa funzione lasciando che il metodo `move` restituisca un oggetto di `ColorPoint` invece che un oggetto di `Point` come specificato nella superclasse `Point`

- Possiamo usare gli oggetti della classe `ColorPoint` dove ci si aspettano oggetti della classe `Point`

Inheritance su Kotlin

Tutte le classi in Kotlin, come anche in Scala hanno una superclasse `Any`, che è la superclasse di default. Per dichiarare un supertipo è necessario specificarlo dopo l'operatore ":" nell'intestazione della classe:

```
open class Base(p: Int)

class Derived(p: Int) : Base(p)
```

L'operatore *open* è il contrario su Java dell'operatore *final*. Per default tutte le classi in Kotlin sono final. Se la classe derivata ha un costruttore principale, la classe base può (e deve) essere inizializzata all'interno di esso, usando i parametri del costruttore principale. Se la classe non ha un costruttore principale, allora ogni costruttore secondario deve inizializzare il tipo base usando la parola keyword `super` o delegare il compito a un altro costruttore. Si noti che in questo caso i diversi costruttori secondari possono chiamare i diversi costruttori del tipo base:

```
class MyView : View {
    constructor(ctx: Context) : super(ctx)

    constructor(ctx: Context, attrs: AttributeSet) : super(ctx,
        attrs)
}
```

A differenza di Java, Kotlin richiede una specifica keyword sia per i membri che subiscono override, sia per i nuovi metodi della sottoclasse:

```
open class Base {
    open fun v() { ... }
    fun nv() { ... }
}
class Derived() : Base() {
    override fun v() { ... }
}
```

Nell'esempio soprastante la keyword `override` è richiesta dal metodo `v()` della classe `Derived`. Se non venisse messa il compilatore segnalerebbe un errore. Se invece non è presente la keyword `open` in un metodo, come ad esempio il metodo `nv()` della classe `Base`, allora dichiarare un metodo con la stessa signature in una sottoclasse è illegale in Kotlin, anche se presente o no la keyword `override`. In una classe *final*, cioè una classe dichiarata non *open* i membri definiti *open* sono proibiti. I membri contrassegnati *override* sono anch'essi *open*, cioè possono essere sovrascritti da una sottoclasse. Se si vuole proibire l'override è necessario usare *final*:

```
open class Base {
    open fun v() { ... }
    fun nv() { ... }
}
class Derived() : Base() {
    override fun v() { ... }
}
```

2.3.4 Smart Cast

In qualsiasi linguaggio a tipizzazione statica, potrebbe essere necessario eseguire delle "conversioni" del tipo di un oggetto. Per esempio in Java, questo avviene in due passaggi; Prima si capisce il tipo dell'oggetto in questione attraverso l'operatore di `istance of` e poi si esegue il cosiddetto *downcast*. Ceylon, essendo anch'esso un linguaggio con enfasi sulla tipizzazione statica, ha una sintassi un pò diversa da Java. Ceylon non possiede *typecast* "in stile C", ma è necessario invece testare e "restringere" il tipo di un oggetto in un solo step, usando il costrutto `if(is...)`.

```
void printIfPrintable(Object obj) {
    if (is Printable obj) {
        obj.printObject();
    }
}
```

In questo esempio, attraverso il costrutto appena citato, viene stampato l'oggetto solo se è considerato un elemento "stampabile". Un relativo costrutto `switch` potrebbe essere questo:

```
void switchingPrint(Object obj) {
    switch (obj)
    case (is Hello) {
```

```
        obj.printMsg();
    }
    case (is Person) {
        print(obj.firstName);
    }
    else {
        print(obj.string);
    }
}
```

In questo modo Ceylon protegge il codice dal causare inavvertitamente `ClassCastException`, grazie a `if(exist...)` che "protegge" dallo scrivere codice che causerebbe una `NullPointerException`. Si può anche eseguire un test in stile Java usando l'operatore `assert` in questo modo:

```
void printIfPrintable(Object obj) {
    assert (is Printable obj);
    obj.printObject();
}
```

2.3.5 Union, intersection and enumerated types

Due features molto importanti di Ceylon e non presenti nel type System di Java sono i tipi di Unione e Intersezione. Un'espressione è assegnabile al tipo di intersezione (Intersection type), scritto nel seguente modo: `X & Y`, se è assegnabile sia al tipo X che al tipo Y. Ad esempio, poiché `Tuple` è un sottotipo di `Iterable` e di `Correspondence`, la tupla di tipo `[String,String]` è anche un sottotipo dell'intersezione `String* & Correspondence<Integer,String>`. I "supertipi" di un 'intersezione includono tutti i supertipi di ogni tipo presente nell'espressione dell'intersezione. Il tipo `String*` rappresenta uno stream che potrebbe non produrre alcun valore quando è iterato. Pertanto il seguente codice è corretto:

```
{String*} & Correspondence<Integer,String> strings
        = ["hello", "world"];
String? str = strings.get(0); //call get() of Correspondence
Integer size = strings.size; //call size of Iterable
```

Nel seguente esempio invece viene usato il costrutto `if(is...)`:

```
{String*} strings = ["hello", "world"];
if (is Correspondence<Integer,String> strings) {
    //here strings has type
```

```
    //{String*} & Correspondence<Integer,String>
    String? str = strings.get(0);
    Integer size = strings.size;
}
```

Un'espressione invece assegnabile al tipo di unione (union type), scritta in questo modo: `X|Y` è assegnabile o al tipo X o al tipo Y. Il tipo `X|Y` è sempre supertipo di entrambi i tipi X e Y. Pertanto il seguente codice è corretto:

```
void printType(String|Integer|Float val) { ... }

printType("hello");
printType(69);
printType(-1.0);
```

Perciò T è un supertipo di `X|Y` se e solo se è un supertipo di entrambi i tipi X e Y. Il compilatore di Ceylon lo determina in automatico. Il seguente codice non crea errori al compilatore:

```
Integer|Float x = -1;
Number<Anything> num = x; // Number is a supertype of both Integer
    and Float
String|Integer|Float val = x; // String|Integer|Float is a supertype
    of Integer|Float
Object obj = val; // Object is a supertype of String, Integer, and
    Float
```

Il seguente esempio invece è errato poichè `Number` non è un supertipo di `String`:

```
String|Integer|Float x = -1;
Number<Anything> num = x; //compile error: String is not a subtype
    of Number<Anything>
```

E' molto comune "restringere" un espressione di tipo unione attraverso un costrutto `switch`. Il compilatore di Ceylon forza a mettere una clausola `else` per ricordare che ci sono casi che potremmo non aver considerato; Ma se li consideriamo tutti, il compilatore ci permetterà di omettere la clausola `else`:

```
void printType(String|Integer|Float val) {
    switch (val)
    case (is String) { print("String: ``val``"); }
    case (is Integer) { print("Integer: ``val``"); }
    case (is Float) { print("Float: ``val``"); }
}
```

A volte potrebbe essere utile fare lo stesso tipo di operazioni con i sottotipi di classi o interfacce. Prima di tutto è necessario usare la clausola `of`:

```
abstract class Point()
        of Polar | Cartesian {
        ...
}
```

Questo rende la classe `Point`, all'interno di Ceylon, allo stesso modo dei tipi che la comunità dei programmi funzionali definisce "algebraic types" o "sum types". Ora il compilatore non permetterà di dichiarare altre sottoclassi di `Point`, e quindi il tipo di unione `Polar|Cartesian` è esattamente lo stesso tipo di `Point`. pertanto, si può scrivere un costrutto `switch` senza la clausola `else`:

```
void printPoint(Point point) {
    switch (point)
    case (is Polar) {
        print("r = " + point.radius.string);
        print("theta = " + point.angle.string);
    }
    case (is Cartesian) {
        print("x = " + point.x.string);
        print("y = " + point.y.string);
    }
}
```

Quando Ceylon riconosce un type parameter di tipo covariante lo sostituisce con l'unione dei corrispondenti tipi degli argomenti, mentre se deduce un type parameter di tipo controvariante lo sostituisce con la relativa intersezione come in questo esempio:

```
value points = Array { Polar(pi/4, 0.5), Cartesian(-1.0, 2.5) }; //
    type Array<Polar|Cartesian>
value entries = zipEntries(1..points.size, points); // type
    {<Integer->Polar|Cartesian>*}
```

2.3.6 Streams, sequences, and tuples

Streams

Uno *stream* è un oggetto "iterable" che produce una serie di valori; il termine "iterabile" significa che attraverso alcuni costrutti possiamo scorrere i

valori prodotti dallo stream. Gli streams implementano l'interfaccia `Iterable`. Ceylon fornisce qualche scorciatoia per lavorare con gli stream:

- il tipo *`Iterable<X,Null>`* rappresenta uno stream che potrebbe non produrre alcun valore quando è "iterato", si può abbreviare con `{X*}`
- il tipo *`Iterable<X,Nothing>`* rappresenta uno stream che quando è iterato produce sempre almeno un valore, si può abbreviare con `{X+}`

Possiamo creare un'istanza di `Iterable` usando le parentesi graffe:

```
{String+} words = { "hello", "world" };
{String+} moreWords = { "hola", "mundo", *words };
```

Il prefisso * si chiama *spread operator* e ha il compito di "diffondere" i valori di uno stream. Perciò, riferendosi al precedente codice, `moreWords` produce i valori `hola`, `mundo`, `hello`, `world` quando è iterato. Per iterare uno stream possiamo usare un semplice "ciclo `for`":

```
for (word in moreWords) {
    print(word);
}
```

Se, per qualsiasi motivo, si necessita l'indice degli elementi prodotti da uno stream su Ceylon, possiamo usare il seguente costrutto, per iterare uno stream di `Entrys`:

```
for (i -> word in moreWords.indexed) {
    print("``i``: ``word``");
}
```

`Entrys` è una coppia contenete una chiave e il valore associato, E' utilizzato principalmente per gli elementi di una mappa. Il tipo `Entry <Key, Item>` può essere abbreviato `Key-> Item`. Un'istanza di `Entry` può essere costruita usando l'operatore `->` : `String->Person entry = person.name->person;` Nell'esempio, l'attributo `indexed` restituisce uno stream di `entries` contenente gli elementi dello stream originale con l'indice. A volte è utile iterare due sequenze contemporaneamente; La funzione `zipEntries()` ci permette di farlo:

```
for (name -> place in zipEntries(names,places)) {
    print(name + " @ " + place);
}
```

Gli stream creati attraverso le parentesi graffe possono creare errori perchè gli elementi al suo interno vengono "rivalutati" ogni volta che lo stream è iterato. Il seguente esempio:

```
variable value counter = 0;
value stream = { for (i in 0:5) counter++ };
print(stream); //evaluate elements
print(stream); //reevaluate elements
```

stampa:

```
{ 0, 1, 2, 3, 4 }
{ 5, 6, 7, 8, 9 }
```

Cosa che non succede usando le parentesi quadre:

```
variable value counter = 0;
value stream = [ for (i in 0:5) counter++ ];
print(stream); //elements already evaluated
print(stream); //elements already evaluated
```

Il precedente codice stampa invece:

```
[0, 1, 2, 3, 4]
[0, 1, 2, 3, 4]
```

Sequences

I moduli del linguaggio di Ceylon definiscono il supporto ai tipi sequenza detti "sequence types" attraverso le interfacce `Sequential, Sequence e Empty`. Scorciatoie associate sono:

- il tipo `Sequential<X>` rappresenta una sequenza che potrebbe essere vuota, e può essere abbreviata con `[X*]` o `X[]`
- il tipo `Sequence<X>` rappresenta una sequenza non vuota, e può essere abbreviata con `[X+]` e
- il tipo `Empty` rappresenta una sequenza vuota e si abbrevia con `[]`

Alcune operazioni di `Sequence` non sono definite da `Sequential`, quindi non possono essere chiamate se si ha `X[]`. Perciò per accedervi è necessario usare il costrutto `if (nonempty ... )` in questo modo:

```
void printBounds(String[] strings) {
```

```
    if (nonempty strings) {
        //strings is of type [String+] here
        print(strings.first + ".." + strings.last);
    }
    else {
        print("Empty");
    }
}
```

Con le sequenze possiamo usare una sintassi simile a Java:

```
String[] operators = [ "+", "-", "*", "/" ];
String? plus = operators[0];
String[] multiplicative = operators[2..3];
```

Il tipo `String?` raffigurato sopra, come verrà descritto più avanti, rappresenta il tipo unione tra il tipo `String` e il tipo `Null` ovvero `String|Null`. Tuttavia, questi costrutti sono pure abbreviazioni; Il seguente codice è esattamente equivalente all'esempio precedente:

```
Sequential<String> operators = [ "+", "-", "*", "/" ];
Null|String plus = operators.get(0);
Sequential<String> multiplicative = operators.span(2,3);
```

L'interfaccia `Sequential` estende `Iterable` perciò possiamo iterare una sequenza del tipo `Sequential` usando un ciclo `for`:

```
for (op in operators) {
    print(op);
}
```

I *Range* sono un tipo di `Sequence`. La funzione `span` crea un *Range*:

```
Sequential<Character> uppercaseLetters = span('A','Z');
Sequential<Integer> countDown = span(10,0);
```

Tuple

Una Tupla è una *linked list* che cattura il tipo statico di ogni singolo elemento nella lista, ad esempio :

```
[Float,Float,String] point = [0.0, 0.0, "origin"];
```

Questa tupla contiene due `Float` seguiti da una `String`. Questa informazione è catturata njel tipo statico `[Float,Float,String]`. `Tuple` estende `Sequence`, quindi si possono fare tutte le operazioni eseguibili sulle sequence, ad esempio iterarle. Come le sequenze, possiamo accedere a un elemento di una tupla attraverso un indice. Ma, nel caso di una tupla, Ceylon è in grado di determinare il tipo di un elemento quando l'indice è un numero intero:

```
Float x = point[0];
Float y = point[1];
String label = point[2];
Null zippo = point[3];
```

2.3.7 Null Type-Safety

I tipi di Unione e Intersezione sono usati per fornire sicurezza ai valori `null`. La super classe di tutte le altre in Ceylon è la classe `Anything`; essa ha due sottoclassi dirette: `Object`, ovvero la superclasse di tutte le classi e di tutte le interfacce, e `Null`, una classe *singleton*. Poiché `Object` e `Null` sono tipi *disgiunti*, molti tipi classici come `Integer` o `List<String>` non consentono di poterli definire opzionali. Un tipo opzionale su Ceylon è composto dall'unione `Intero | Nullo`, che si può abbreviare con `Integer?`. Segue un esempio di un piccolo programma nel quale viene passato una variabile `String?` da linea di comando. Ceylon tiene conto anche del fatto che potrebbe non essere passato niente in input; diversamente da Java, in Ceylon attraverso la keyword `exists` è possibile gestire i valori di tipo `Null`. In ceylon non esistono i *typecast* in "stile C"; Usare i costrutti "`if (is ... )`" e "`case (is ... )`" sono una possibile soluzione di Ceylon per testare il tipo di un elemento in un solo passo, senza causare una `ClassCastException`. Questa tecnica è chiamata *flow-sensitive typing*

```
String? name
       = process.arguments.first;
String greeting;
if (exists name) {
    greeting = "Hello, ``name``!";
}
else {
    greeting = "Hello, World!";
}
print(greeting);
```

```
String name(Organization|Person entity) {
    switch (entity)
    case (is Organization) {
        return entity.tradeName else entity.legalName;
    }
    case (is Person) {
        return entity.nickName else entity.firstName;
    }
}
```

Null Type-Safety in Scala

Confrontando con un altro linguaggio per JVM, ovvero Scala, i valori opzionali sono gestiti attraverso il tipo `Option`. In generale una `Option[A]` (valore opzionale di un tipo non specificato `A`) è un tipo algebrico che ammette solo 2 possibili istanze:

- `Some[A]` che contiene effettivamente un qualche valore
- `None` che indica un valore assente

Ad esempio, parlando di una potenziale rubrica di contatti, nella quale abbiamo oggetti di tipo `Contact` che al loro intorni hanno attributi come `phone`, `mobile`, ecc..., si potrebbe avere codice di questo genere che sfrutta `Option`:

```
contact.phone = Some("+39 000545110")
contact.mobile = None
contact.email = Some("miaemail@miodominio.it")
```

Analogamente al caso dei campi opzionali, `Option` è spesso usato per indicare una funzione il cui risultato non è garantito. Come esempio immaginiamo di avere una funzione della rubrica che permette di trovare un contatto per nome:

```
def findByName(name: String): Option[Contact] = ...
```

Già dalla signature del metodo ci si può rendere conto che si potrebbe non avere un risultato. Possiamo usare i tipi opzionali di Scala anche come parametri di una funzione, operazione semplificata grazie al supporto dei *parametri di default*

```
def addToFavorites(contact: Contact, category: Option[String] =
    None) = ...

addToFavorites(myGoodFriend)
```

```
addToFavorites(personalTrainer, "fitness")
```

In questo esempio viene definito il metodo `addToFavorites` specificando che se viene aggiunto un contatto senza specificare la relativa categoria, per default la categoria sarà `None`. Per la classe `Option`, gli strumenti forniti per estrarre un valore o testare se esiste sono i metodi `get` e `isDefined`:

```
class Option[A] {

    def get: A

    def isDefined: Boolean

}
```

Un modo intuitivo, senza rischi per il programma, per utilizzare un valore in un elemento option è utilizzare le capacità di *pattern matching* messe a disposizione da Scala. Analogamente a come le espressioni regolari permettono di verificare se un testo corrisponde ad uno specifico pattern, lo stesso si può fare in molti linguaggi funzionali verificando se un "oggetto" o "dato" è conforme ad un "pattern strutturale". La sintassi in Scala è simile ad uno "switch", e utilizza le istruzioni *match* e *case*

```
val optionalValue: Option[String] = Some("content")

val safeValue = optionalValue match {
    case Some(value) => value
    case None => "missing"
}
```

`safeValue` viene determinato in base al corrispondente ramo del match, ossia

- se `optionalValue` corrisponde a `Some(value)` allora restituisce quello che c'è in `value`
- se `optionalValue` corrisponde a `None` allora restituisce un valore predefinito, in questo caso "missing"

Nel nostro esempio `optionalValue` corrisponde a `Some("content")`, pertanto si ricade nel primo caso, dove `value` corrisponde a "content" per cui viene restituito il valore "content", appunto. In generale le conseguenze di ciascun case (ossia il codice definito a destra del `=>`) possono essere qualsiasi, e restituire qualunque valore, purché tutte siano consistenti rispetto al tipo di valore

restituito. Difatti è necessario che tutto il pattern match venga convertito in un risultato ben definito. potremmo anche eseguire un'operazione, come nel seguente esempio:

```
optionalValue match {
  case Some(value) => println(value)
  case None =>
}
```

Null Type-Safety in Java

Anche Java presenta il tipo `Optional<T>`, introdotto in Java 8. Esistono specializzazioni della classe utilizzate per rappresentare alcuni tipi primitivi, quali: `OptionalDouble` , `OptionalInt` e `OptionalLong`. In generale un oggetto `Optional` può trovarsi in due stati possibili:

- *present*: ovvero contiene un riferimento non nullo ad un oggetto di tipo `T`;
- *absent* o *empty*: in caso opposto.

Un oggetto `Optional` empty non è equivalente a un oggetto nullo. Per instanziare un nuovo oggetto di tipo `Optional` esistono diverse possibilità:

- `Optional.empty()`: metodo statico che restituisce un oggetto Optional empty.
- `Optional.of()`: metodo statico che crea un `Optional` per un oggetto non nullo, altrimenti solleva l'eccezione `NullPointerException`.
- `Optional.ofNullable()`: metodo statico che crea un `Optional` per un oggetto che può essere nullo, nel caso restituendo un oggetto `Optional.empty()`.

Altri metodi significativi della classe Optional sono:

- `isPresent()` che permette di controllare la presenza del valore nell'istanza `Optional` considerata, restituisce `true` se l'oggetto Optional non è vuoto altrimenti `false`
- `ifPresent(Consumer<? super T> consumer)` che invoca il `Consumer`, passandogli il valore, solo se l'oggetto `Optional` non è vuoto
- `get()` che restituisce il valore, se presente, altrimenti lancia `NoSuchElementException`

- `orElse(T other)` e `orElseGet(Supplier<? extends T> other)` usati per definire valori di default, il primo restituisce il valore, se presente, altrimenti restituisce l'istanza `other`, il secondo restituisce il valore, se presente, altrimenti invoca `other`, e restituisce il risultato dell'invocazione.

`orElse` richiede come argomento l'istanza di un oggetto, che viene sempre creato e usato solo se l'istanza `Optional` è vuota. `orElseGet`, invece, richiede l'interfaccia funzionale `Supplier`, che può essere implementata tramite espressione lambda, creata e passata al metodo, ma eseguita solo se l'istanza `Optional` è vuota. I controlli relativi a valori `null`, possono essere sostituiti da chiamate a metodi di `Optional`, riducendo la probabilità di incorrere in `NullPointerException`. Ad esempio, il seguente codice:

```
String str;
if(str!=null){
//operazioni da eseguire
}
```

con l'uso di `Optional` diventa:

```
String str;
Optional<String> optionalString = Optional.of(str);
if(optionalString.isPresent()){
//operazioni da eseguire
}
```

Con le annotazioni, sono stati introdotti strumenti che rilevano potenziali *null dereference*. Gli sviluppatori devono annotare le dichiarazioni appropiatamente. Uno studio empirico su 5 progetti open source riporta che, in media, 3/4 delle dichiarazioni devono essere non nulle, in base alla progettazione, al fine di migliorare gli errori causati dai valori nulli [4].

Null Type-Safety in Kotlin

Kotlin cerca di eliminare le `NullPointerException` con un approccio simile a Ceylon riducendo a due soli i casi in cui possono occorrere:

- Una chiamata esplicita a `throw NullPointerException()`
- L'uso dell'operatore "`!!`" su riferimenti `null`

In Kotlin, il type system distingue tra riferimenti che possono contenere valori `null` e quelli che non possono contenerne. Ad esempio, una variabile regolare

di tipo `String` non può contenere `null`. Per fare in modo che questo sia possibile, dobbiamo dichiararla in questo modo: `String?`

```
var a: String = "abc"
a = null // compilation error
```

```
var b: String? = "abc"
b = null
print(b)
```

Per testare la null condition è possibile verificare esplicitamente se è nullo e gestire le due opzioni separatamente:

```
val l = if (b != null) b.length else -1
```

Un altro modo è usare l'operatore "?" detto *safe call operator*:

```
val a = "Kotlin"
val b: String? = null
println(b?.length)
println(a?.length)
```

Esso restituisce `b.length` se `b` non è null, altrimenti restituisce `null`. L'operatore potrebbe essere usato in catena per verificare più condizioni di *nullabilità* una dopo l'altra. Una terza opzione è usare l'operatore `!!` che converte qualsiasi valore in un tipo non-nullo e lancia una exception se il valore è nullo.

2.3.8 Supporto integrato per la modularità

Il supporto integrato per la modularità è uno dei principali obiettivi del progetto Ceylon, che fornisce:

- Un supporto a livello di linguaggio per un'unità di visibilità più grande di un *package*, ma più piccola di "tutti i *packages*".
- Un archivio integrato di moduli e un *layout* del *repository* dei moduli compreso e condiviso da tutti gli strumenti del linguaggio, dal compilatore, all'ambiente di sviluppo, a tempo di esecuzione.
- Un ecosistema di repository di moduli remoti dove gli sviluppatori possono condividere codice con altri.
- Un runtime che presenta un caricamento di classi peer-to-peer(un *classloader* per modulo) e la possibilità di gestire più versioni dello stesso modulo.

In Ceylon la visibilità di un pacchetto (*package*) può essere condiviso (`shared`) o non condiviso. Un *package* non condiviso (di default) è visibile solo al modulo che contiene il *package*. Possiamo rendere il *package* condiviso, cioè `shared`, dichiarando ciò nel suo *descriptor*, ovvero la dichiarazione di esso, come nella figura sottostante:

```
Package package {
    name = 'org.hibernate.query';
    shared = true;
    doc = "The typesafe query API.";
}
```

Gli altri moduli possono accedere direttamente alle dichiarazioni condivise in un pacchetto dichiarato `shared`, ma un modulo deve specificare esplicitamente gli altri moduli da cui dipende. Questo però deve essere dichiarato nel *module descriptor*, cioè la dichiarazione del modulo:

```
Module module {
    name = 'org.hibernate';
    version = '3.0.0.beta';
    doc = "The best-ever ORM solution!";
    license = 'http://www.gnu.org/licenses/lgpl.html';
    Import {
        name = 'ceylon.language';
        version = '1.0.1';
        export = true;
    },
    Import {
        name = 'java.sql';
        version = '4.0';
    }
}
```

Un modulo può essere eseguibile, in tal caso deve specificare un metodo `run` nel module descriptor:

```
Module module {
    name = 'org.hibernate.test';
    version = '3.0.0.beta';
    doc = "The test suite for Hibernate";
    license = 'http://www.gnu.org/licenses/lgpl.html';
    void run() {
        TestSuite().run();
    }
```

```
    Import {
        name = 'org.hibernate'; version = '3.0.0.beta';
    }
}
```

Un *package* di moduli archiviati, può essere archiviato insieme ai binari e ai *module descriptor* in un archivio jar, proprio come in Java, ma con estensione `.car`. Il compilatore Ceylon di solito non produce singoli file con estensione `.class` in una directory. Invece, produce direttamente archivi di moduli. Gli archivi dei moduli si trovano nei repository dei moduli. Un repository di moduli è una struttura di directory ben definita con una posizione ben definita per ciascun modulo e può essere locale (sul filesystem) o remoto (su Internet). Dato un elenco di repository di moduli, il compilatore Ceylon può individuare automaticamente le dipendenze menzionate nel *module descriptor* che sta compilando in quel momento. Terminata la compilazione, l'archivio viene posizionato nel posto assegnato, in un repository di moduli locale. Il modulo *run time*, dato un elenco di repository di moduli, localizza automaticamente l'archivio e le sue dipendenze nei repository, scaricando archivi da remoto se necessario. Normalmente, il *runtime* di Ceylon viene richiamato specificando il nome di un modulo eseguibile sulla riga di comando, e tutte le dipendenze richieste vengono scaricate automaticamente secondo necessità:

```
ceylon run org.jboss.ceylon.demo -rep http://jboss.org/ceylon/modules
```

Red Hat gestisce un repository centrale di moduli pubblici in cui la community può contribuire con moduli riutilizzabili. Naturalmente, il formato del repository del modulo è uno standard aperto, quindi ogni organizzazione può mantenere il proprio repository di moduli pubblici.

2.3.9 Higher Order Language

Ceylon nasce come *Higher-Order-Language*; possiede quindi , tra le caratteristiche principali, la possibilità di scrivere le "high-order functions" ovvero le cosiddette funzioni di alto livello. Tale *feature* nel linguaggio Java venne introdotta insieme alle "Lambda-Expressions" con la versione *Java SE 8*. Una funzione è definita "di alto livello" se possiede almeno una delle seguenti caratteristiche:

- prende in input una o più funzioni come argomenti, oppure
- restituisce una funzione come risultato.

Ceylon non è un linguaggio funzionale puro, però la possibilità di avere *Higher-Order*, lo avvicina ai linguaggi di programmazione funzionali. In realtà, non

c'è nulla di nuovo nell'avere delle funzioni trattate come valori in un linguaggio orientato agli oggetti. Ad esempio, Smalltalk, uno dei primi, e ancora uno dei linguaggi orientati agli oggetti più puliti, è stato costruito attorno a questa idea. Ceylon, come Smalltalk, consente di trattare una funzione come un oggetto e passarla in ingresso a un'altra funzione, o anche a una variabile.

```
void repeat(Integer times,
        void perform(Integer n)) {
    for (i in 1..times) {
        perform { n=i; };
    }
}

void printNum(Integer n) => print(n);
repeat(10, printNum);
```

2.3.10 Interoperabilità con Java

Ceylon è progettato per l'esecuzione in un ambiente di macchina virtuale, ma non ha una propria macchina virtuale nativa. Invece, possiamo dire che "prende in prestito" una macchina virtuale progettata per un altro linguaggio. Può essere eseguito su:

- La Java Virtual Machine (JVM),
- Una macchina virtuale JavaScript.

Molti moduli Ceylon che richiamano codice Java nativo sono progettati per essere eseguiti solo sulla JVM. In questo caso, dichiariamo l'intero modulo Ceylon come un modulo JVM nativo usando l'annotazione `native`.

```
native ("jvm")
module ceylon.formatter "1.3.3" {
    shared import java.base "7";
    shared import com.redhat.ceylon.typechecker "1.3.3";
    import ceylon.collection "1.3.3";
    ...
}
```

Un modulo multi piattaforma può anche richiamare codice Java nativo, ma in questo caso è necessario applicare la keyword `native` sulle istruzioni di tipo `import` che dichiarano dipendenze su archivi jara.

```
module ceylon.formatter "1.3.3" {
```

```
    native ("jvm") shared import java.base "7";
    native ("jvm") shared import com.redhat.ceylon.typechecker
        "1.3.3";
    import ceylon.collection "1.3.3";
    ...
}
```

Il Java SE Development Kit (JDK) è rappresentato in Ceylon come un insieme di moduli, secondo la modularizzazione proposta dal progetto Jigsaw. Per usare i moduli del JDK dobbiamo importarli. Alcuni moduli importanti sono ad esempio:

- il modulo `java.base` che contiene pacchetti di base tra cui `java.lang`, `java.util` e `javax.security`,
- il modulo `java.desktop` che contiene i framework di interfaccia utente desktop AWT e Swing e
- `java.jdbc` che contiene l'API JDBC per la gestione dei Database.

Perciò, se abbiamo bisogno ad esempio di utilizzare il framework delle Collezioni Java nel nostro programma Ceylon, abbiamo bisogno di creare un modulo di Ceylon che dipenda da java.base. Fatto ciò, è possibile semplicemente importare la classe Java a cui siamo interessati e usarla come qualsiasi classe Ceylon ordinaria:

```
native ("jvm")
module org.jboss.example "1.0.0" {
    import java.base "7";
}

import java.util { HashMap }

void hashyFun() {
    value hashMap = HashMap<String,Object>();
}
```

Ci sono tre classi Java e un'interfaccia che non devono essere utilizzate nel codice di Ceylon, poiché Ceylon fornisce tipi che sono esattamente equivalenti nel pacchetto ceylon.language e sono :

- `java.lang.Object`, rappresentato da Ceylon attraverso la classe `Object`,
- `java.lang.Exception`, rappresentato da Ceylon attraverso la classe `Exception`,

- `java.lang.Throwable`, rappresentato da Ceylon attraverso la classe `throwable` e
- `java.lang.annotation.Annotation`, rappresentato da Ceylon attraverso l'interfaccia `Annotation`.

Tentare di importare uno di questi tipi Java nel codice di Ceylon produce un errore. Inoltre abbandona la distinzione fra primitivi e *wrapper types*, mappando i primitivi Java come segue:

- `boolean` è rappresentato in Ceylon come `Boolean`
- `char` è rappresentato in Ceylon come `Character`
- `long,int e short` sono rappresentati a Ceylon come `Integer`
- `byte` è rappresentato in Ceylon come `Byte`
- `double e float` sono rappresentati in Ceylon come `Float`
- `java.lang.String` è rappresentato in Ceylon come `String`

Tutte le conversioni da un tipo primitivo Java a un tipo Ceylon garantiscono successo a runtime, ma al contrario le conversioni da un tipo Ceylon a un tipo primitivo Java potrebbe coinvolgere una conversione implicita. Ad esempio se:

- Un tipo `Integer` di Ceylon è assegnato a un tipo Java `int` o `short`,
- un tipo Ceylon `Float` è assegnato a un tipo Java `float`, o se
- Un `Character` Ceylon, che usa UTF-32, è assegnato a un `char` Java(UTF-16).

Questi assegnamenti potrebbero causare overflow o perdita di precisione. Fornire avvertimenti sulle operazioni che potrebbero causare un overflow numerico non è tra gli obiettivi del linguaggio Ceylon; in generale, qualsiasi operazione su un tipo numerico, incluse somme o moltiplicazioni potrebbero causare overflow. Non esiste alcuna associazione tra le classi wrapper di Java, come ad esempio `java.lang.Integer` o `java.lang.Boolean`, e i tipi base di Ceylon, quindi queste conversioni devono essere eseguite esplicitamente chiamando ad esempio il metodo `longValue()` o `booleanValue()` oppure istanziando esplicitamente la classe wrapper, proprio come si farebbe in Java durante la conversione tra un tipo primitivo Java e la sua classe wrapper. Questo è importante soprattutto quando si lavora con le `Collection` Java. Ad esempio, in Java `List<Integer>` non contiene i tipi `Integers` di Ceylon.

```
import java.lang { JInteger=Integer }
import java.util { JList=List }

JList<JInteger> integers = ... ;
for (integer in integers) { //integer is a JInteger!
    Integer int = integer.longValue(); //convert to Ceylon Integer
    ...
}
```

Analoga situazione per le stringhe:

```
import java.lang { JString=String }
import java.util { JList=List }

JList<JString> strings = ... ;
for (string in strings) { //string is a JString!
    String str = string.string; //convert to Ceylon String
    ...
}
```

La conversione esplicita tra `String` e `java.lang.String` si esegue in questi modi:

- L'attributo `.string` di una stringa Java restituisce un tipo `string` di Ceylon, e
- uno dei costruttori di `java.lang.String` accetta il tipo `String` di Ceylon, o alternativamente
- la funzione `Types.nativeString()` nel package `java.lang` converte le stringhe di Ceylon nelle stringhe di Java senza necessitare dell'istanziazione dell'oggetto

Allo stesso modo, le conversioni tra tipi Ceylon e tipi wrapper primitivi Java si eseguono in questo modo:

- il metodo `.longValue()` di `java.lang.Long` e `java.lang.Integer` restituisce un `Integer` Ceylon, e
- i costruttori di `java.lang.Integer` e `java.lang.Long` accettano un `Integer` Ceylon

Se si necessità di un `int` o `float 32-bit` a livello di bytecode, anziché il tipo `long` o `double 64-bit` usato da Ceylon di default si può utilizzare l'annotazione `small`

```
small Integer int = string.hash;
```

Poiché in Ceylon non sono definiti gli array come tipi primitivi, essi sono rappresentati da calssi speciali, considerate appartenenti al package `java.lang`, a sua volta appartenente al modulo `java.base`. Perciò è necessario importare esplicitamente queste classi per poterle utilizzare.

- `boolean[]` è rappresentato in Ceylon con la classe `BooleanArray`,
- `char[]` è rappresentato in Ceylon con la classe `CharArray`,
- `long[]` è rappresentato in Ceylon con la classe `LongArray`,
- `int[]` è rappresentato in Ceylon con la classe `IntArray`,
- `short[]` è rappresentato in Ceylon con la classe `ShortArray`,
- `byte[]` è rappresentato in Ceylon con la classe `ByteArray`,
- `double[]` è rappresentato in Ceylon con la classe `DoubleArray`,
- `float[]` è rappresentato in Ceylon con la classe `FloatArray`, e, ultimo,
- `T[]` per qualsiasi oggetto di tipo `T` è rappresentato in Ceylon con la classe `ObjectArray<T>`.

Verifica a runtime di valori di ritorno `null`

I valori di ritorno di tipo null vengono controllati a runtime; i campi o metodi Java non specificano se un campo o un metodo produrranno un valore nullo, eccetto per i casi particolari dei tipi primitivi. Il compilatore Ceylon tratta quasi tutti i metodi e campi Java come aventi un tipo non-opzionale, però un metodo potrebbe ancora restituire `null` in fase di esecuzione. Pertanto il compilatore Ceylon deve inserire controlli sui valori nulli ovunque nel codice Ceylon venga invocata una funzione Java che restituisce un oggetto o venga testato il tipo di oggetto di un campo, e assegna il risultato a un tipo non-opzionale Ceylon. Nel seguente esempio, non viene eseguito alcun controllo del valore nullo a runtime, poiché il valore di ritorno di `System.getProperty()` è assegnato al tipo opzionale `String?`:

```
import java.lang { System }

void printUserHome() {
    String? home //optional type
```

```
            = System.getProperty("user.home");
    print(home);
}
```

Nel seguente codice invece, un controllo runtime viene eseguito quando il valore di ritorno di `System.getProperty()` viene assegnato al tipo non-opzionale `String`:

```
import java.lang { System }

void printUserHome() {
    String home //non-optional type, possible runtime exception
            = System.getProperty("user.home");
    print("home: " + home);
}
```

Costanti Java

Le costanti Java come `Integer.MAXVALUE` e i valori `enum` come `RetentionPolicy.RUNTIME`, sono per convenzione rappresentate in maiuscolo. Nel codice Ceylon per convenzione viene usato il *camel case* per rappresentarle, come in questo esempio:

```
Integer maxInteger = JInteger.maxValue;
RetentionPolicy policy = RetentionPolicy.runtime;
```

Java generic types

Un'istanziazione di un tipo generico Java come ad esempio `java.util.List<E>` è rappresentato senza nessuna regola speciale in Ceylon. Il tipo Java `List<String>` può essere scritto come `List<String>` in Ceylon, dopo aver importato `List` e `String` dai moduli `java.base`:

```
import java.lang { String }
import java.util { List, ArrayList }

List<String> strings = ArrayList<String>();
```

Wildcards e tipi raw

In Java, la covarianza e la controvarianza sono rappresentate attraverso i generici, usando i caratteri wildcard, mentre su Ceylon lo stesso risultato si

ottiene usando le keyword `in` e `out`:

- `List<out Object>` è covariante ed è l'equivalente Java di `List<? extends Object>`
- `Topic<in Object>` è controvariante ed è l'equivalente Java di `Topic<? super Object>`

I Java *raw types* sono rappresentati in Ceylon con la wildcard covariante `out`. Ad esempio il raw type `List` è rappresentato in Ceylon in questo modo: `List<out Object>`. Preso ad esempio un tipo generico invariante `Type<X>`, le istanziazioni `Type<out Anything>` e `Type<in Nothing>` sono esattamente equivalenti, e sono supertipi di ogni istanziazione di `Type`

Overloading di metodi e costruttori

Quando una classe o un'interfaccia è dichiarata nativa ("jvm"), può dichiarare metodi e costruttori *overloaded*. Sia i costruttori di default che i costruttori denominati possono subire l'overload. Un metodo o costruttore su cui si può eseguire l'overload deve essere contrassegnato con l'annotazione "overloaded", che è considerata appartenente al `package java.lang` nel modulo `java.base`.

```
import java.lang { overloaded }

native("jvm")
class Native {
    variable Integer int;
    shared overloaded new () {
        int = 0;
    }
    shared overloaded new (String string) {
        int = parseInteger(string);
    }
    shared overloaded new (Integer int) {
        this.int = int;
    }
    shared overloaded set(String string) {
        int = parseInteger(string);
    }
    shared overloaded set(Integer int) {
        this.int = int;
    }
    shared Integer get() => int;
```

Estensione di tipi Java

Una classe Ceylon può estendere una classe Java e/o implementare un'interfaccia java, anche se la classe o l'interfaccia presentano parametri generici. Come nell'esempio sottostante, i parametri dei metodi java sono trattati come valori opzionali; Tuttavia, quando viene ridefinito un metodo Java in una classe Ceylon, è possibile ridefinire i parametri come non-opzionali:

```
//java interface
public interface Stringifier<T> {
    public String stringify(T thing);
}
```

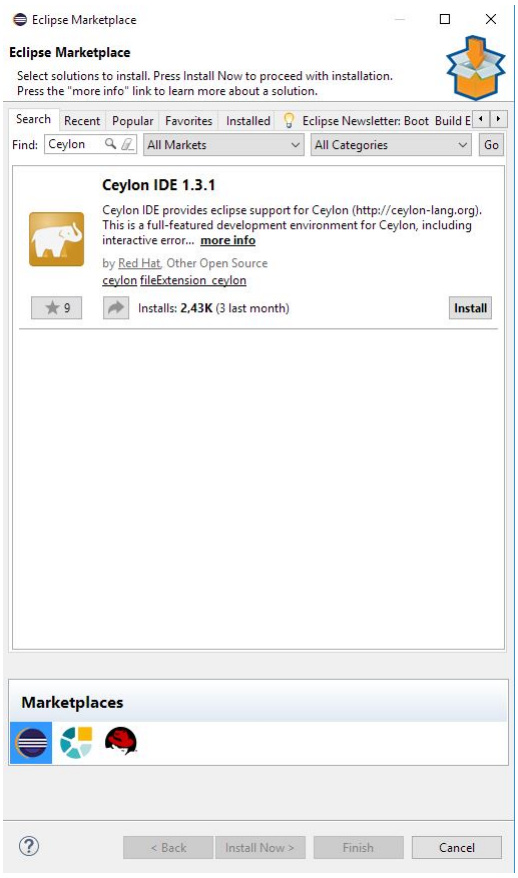
```
/ceylon implementation
class EntryStringifier<Key,Item>()
        satisfies Stringifier<Key->Item> {
    stringify(Key->Item entry)
            => "``entry.key``->``entry.item``";
}
```

Nell'esempio soprastante, non è necessario dichiarare `entry` come tipo `<Key->Item>?`, a meno che non ci si aspetti che il metodo possa essere chiamato con argomenti nulli

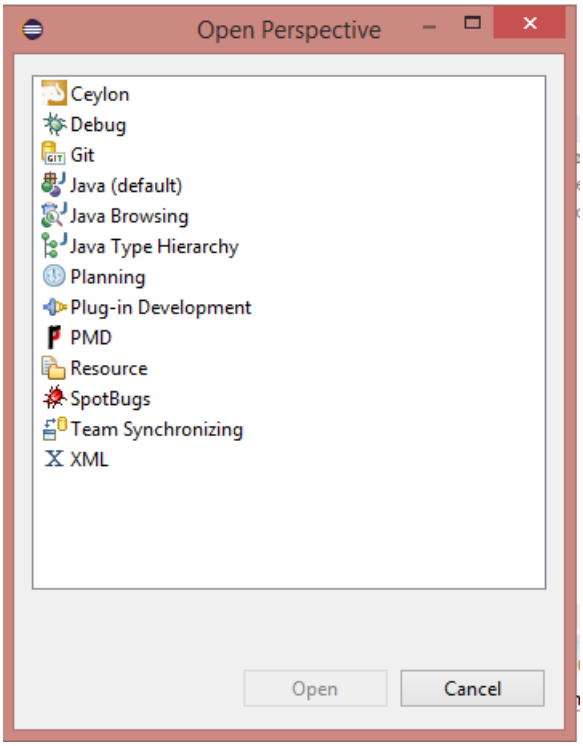
Capitolo 3

Setup

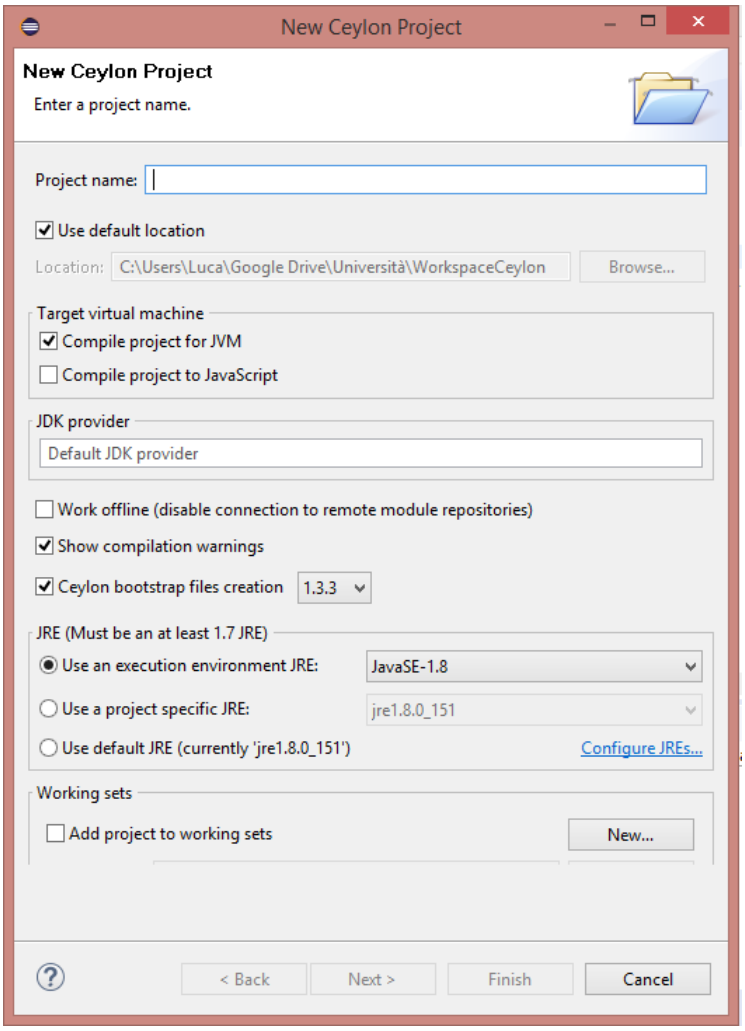
Per programmare con il linguaggio Ceylon sono disponibili due ambienti di sviluppo (IDE), per Eclipse e per IntelliJ. Quello più famoso è quello per Eclipse, chiamato "Ceylon IDE", il cui download è disponibile sul market di Eclipse nella sezione "Help" -> "Eclipse Marketplace" o direttamente sul sito ufficiale di Ceylon nella sezione "Download".



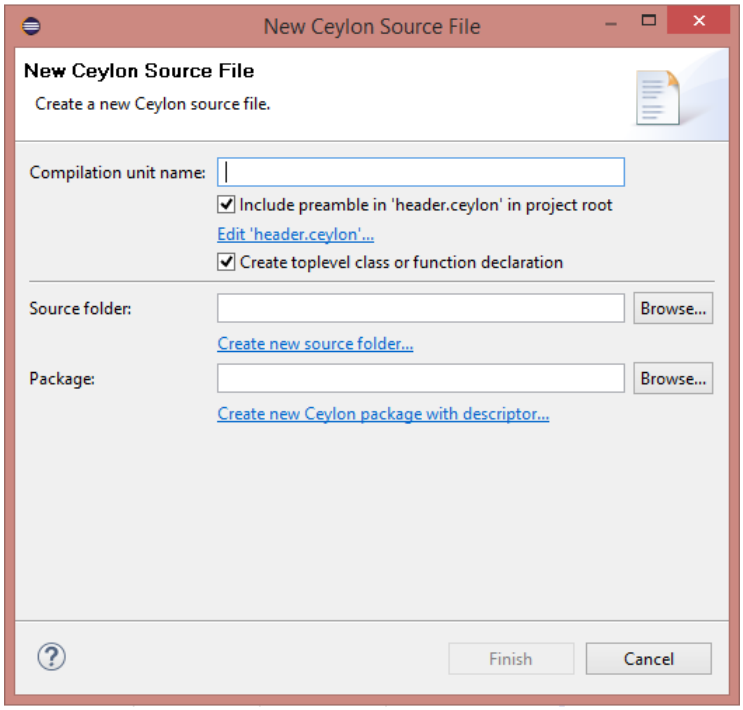
Per funzionare, il plug-in richiede la versione di Java 7 o superiore. Una volta installato, per entrare sulla *Ceylon Perspective*, dalla home di Eclipse è necessario cliccare su "Window" > "Open Perspective" > "Other" ... > Ceylon.



Per creare un nuovo progetto Ceylon è necessario cliccare su "File" > "New" > "Ceylon Project", dare un nome al progetto e infine cliccare su "Finish".



Successivamente è necessario creare un file sorgente cliccando su "File" > "New" > "Ceylon Source File", dare un nome al file, selezionare l'opzione "Create toplevel class or function declaration", scegliere la cartella che conterrà i file sorgente e infine cliccare su "Finish".



L'editor di Ceylon verrà aperto in automatico e si potrà iniziare a scrivere codice. Per eseguire il programma è necessario cliccare sulla cartella del progetto e cliccare su "Run" > "Run As" > "Ceylon Java Application"; dovrebbe essere ora possibile leggere i relativi messaggi o stampe sulla console.

Capitolo 4

Esempi d'uso

In questo capitolo si susseguono esempi di codice Ceylon che mostrano le funzionalità base del linguaggio.

4.1 Classes and functions

```
Float mean(Float x, Float y) {
    return ((x*x + y*y) / 2) ^ 0.5;
}
print("mean(11.0, 307.0) = ``mean(11.0, 307.0)``");

Float mean2(Float x, Float y) => ((x*x + y*y) / 2) ^ 0.5;

function average(Float x, Float y) {
    return (x + y) * 0.5;
}
print("average(11.0, 307.0) = ``average(11.0, 307.0)``");

class SimpleProduct(String name, Float price) {

    if (price < 0.0) {
        throw Exception("Invalid price: ``price``!");
    }

    variable value quantity = 10;

    shared Boolean inStock() => quantity > 0;

    "Determines if the product is free of charge"
    shared Boolean isFree() {
```

```
        return price == 0.0;
    }
}

value trompon = SimpleProduct("Elephant", 0.0);
print("free: ``trompon.isFree()``");


"Abstract base class for products"
by("superuser")
abstract class Product(name, price) {

    shared String name;
    shared Float price;

    if (price <= 0.0) {
        throw Exception("Invalid price: ``price``!");
    }

    shared formal variable Integer quantity;

    shared Boolean isFree = (price == 0.0);

    shared default Boolean inStock { return quantity > 0; }
}

"Concrete implementation of `Product`"
see (`class Product`)
class ProductInDatabase(String name, Float price)
           extends Product(name, price) {

    shared actual Integer quantity {
        return 10;
    }
    assign quantity {
        print("Set quantity to ``quantity``");
    }
}

value polly = ProductInDatabase("Parrot", 49.99);
print("Product '``polly.name``' costs ``polly.price``");

print("In stock: ``polly.inStock`` (quantity: ``polly.quantity``)");
polly.quantity = 12;
```

```
void order(Product|String product, Integer count=1,
          String* comments) {

    String name;
    switch (product)
    case (is Product) { name = product.name; }
    case (is String) { name = product; }

    print("Order product '``name``', count: ``count``");
    for (c in comments) {
        print(" Comment: '``c``'");
    }
}

order(polly, 28, "I like birds");
```

Nel precedente codice sono mostrati esempi del linguaggio Ceylon riguardanti funzioni e classi. Le funzioni usano una sintassi familiare a Java, come si nota dalla funzione `mean` in riga 1. L'operatore `=>` permette di scriverle in forma breve con una sola riga di codice. Se si utilizza la keyword `function` il compilatore sarà in grado di riconoscere il tipo. Le classi hanno una sintassi molto simile alle funzioni; i membri delle classi dichiarati shared come il metodo `inStock` della classe `SimpleProduct` è visibile ovunque la classe sia visibile, gli altri membri invece sono privati della classe. Successivamente è mostrata la classe astratta `Product` nella quale i parametri passati in ingresso `name` e `price` sono subito trasformati in attributi della classe. All'interno della classe appare il membro `quantity` dichiarato `formal` perchè implementato nella sottoclasse `ProductInDatabase` con la keyword `actual`. Nell'ultimo metodo, il metodo `order` si nota come l'utilizzo di *union type*, *default parameters* e *sequenced parameters* sostituisca il fatto che due funzioni con lo stesso nome non sono permesse. All'interno del metodo `order`, attraverso il costrutto `switch/case(is...)` è gestito il tipo di valore che può assumere il parametro `product`

4.2 Interfaces and inheritance

```
interface Customer {

    "The name of the customer."
    shared formal String name;
```

```
    shared default String description => "'``name``'";

}

class SimpleCustomer(name) satisfies Customer {
    shared actual String name;
}

value batman = SimpleCustomer("Batman");
print("Customer: ``batman.description``");

interface RegisteredCustomer satisfies Customer {

    "The unique customer ID"
    shared formal Integer id;

    shared actual default String description
            => "'``name``' (#``id``)";
}

interface CartOwner satisfies Customer {

    "Number of items in the shopping cart"
    shared formal Integer numberOfItems;

    shared actual default String description
            => "'``name``' (``numberOfItems`` items)";

    "The customer buys the items in the cart."
    shared default void checkOut() {
       print("'``name``' buys ``numberOfItems`` items.");
    }
}

class PetShopCustomer(id, name)
            satisfies RegisteredCustomer & CartOwner {

    shared actual Integer id;
    shared actual String name;

    shared actual variable Integer numberOfItems = 0;

    shared actual String description
```

```
            => "'``name``' (#``id``, ``numberOfItems`` items)";

    shared actual Boolean equals(Object other) {
        if (is PetShopCustomer other) {
            return other.id == id;
        }
        return false;
    }
    shared actual Integer hash = id;
    shared actual String string = description;
}

value catwoman = PetShopCustomer(1, "Catwoman");
print("Customer: ``catwoman``");
catwoman.numberOfItems = 3;
catwoman.checkOut();
```

In questo esempio viene mostrato il funzionamento delle interfacce e dell'ereditarietà. L'interfaccia `Customer` rappresenta un potenziale cliente di un negozio. `Customer` contiene la stringa rappresentante il nome del cliente e la stringa description che lo restituisce; description è dichiarata default per permettere l'override. La successiva classe `SimpleCustomer` implementa l'interfaccia `Customer`. Abbiamo poi due interfacce `RegisteredCustomer` e `CartOwner`, che implementano entrambe la prima interfaccia `Customer`. Queste due interfacce rappresentano due specializzazioni di clienti perciò aggiungono i loro campi personali come `id` o `numberOfItems` e fanno l'override del membro `description` di `Customer`. Infine abbiamo la classe `PetShopCustomer` che implementa sia `RegisteredCustomer` sia `CartOwner`. In quest'ultima classe viene fatto l'override del metodo `equals`, ereditato dalla classe `Basic`; perché se non viene specificato niente, ogni classe estende `Basic`.

4.3 Sequences

```
String[] animals = [ "elephant", "parrot", "giraffe" ];
String[] none = [];

print("First: ``animals.first else "none"``");
print("Second: ``animals[1] else "none"``");
print("Fourth: ``animals[3] else "none"``");

if (nonempty animals) {
```

```
    print("First: ``animals.first``");
    print("Last: ``animals.last``");
}

print("Is {} non-empty: ``none nonempty``");

if (nonempty animals) {
    process.write(animals.first);
    for (animal in animals.rest) {
        process.write(", ``animal``");
    }
    process.write(operatingSystem.newline);
}

for (Character c in "cat") {
    print(c);
}

void list(Object* items) {
    variable Integer index = 0;
    for (item in items) {
        print("``++index``.``item.string``");
    }
}
list("cat", "bat");

list(*animals);

print(", ".join(animals));

for (i->animal in animals.indexed) {
    print("``i+1``.``animal``");
}

class LengthsIterator(String* strings)
                  satisfies Iterator<Integer> {
    value it = strings.iterator();

    shared actual Integer|Finished next() {
        if (is String s = it.next()) { return s.size; }
        return finished;
    }
}
```

```
class Lengths(String* strings) satisfies Iterable<Integer> {
    shared actual Iterator<Integer> iterator()
            => LengthsIterator(*strings);
}
for (len in Lengths(*animals)) {
    print("length: ``len``");
}

{Integer*} lengths = { for (a in animals) a.size };
for (len in lengths) { print("length: ``len``"); }

list(for (a in animals) a+" egg");

list(for (i in 1..3) for (j in 1..3) if ((i+j)%2 == 0) i*j);

print("Nothing will be printed");
list(if (animals.size > 3) for (a in animals) a);
```

In questo esempio viene mostrato il funzionamento delle collezioni e delle sequenze. Vengono create due sequenze di valori, `animals` contenente stringhe con nomi di animali e **none**, sequenza vuota. Successivamente viene stampato il contenuto di `animals` oppure **none** se non presente. Attraverso `nonempty` vediamo se la sequenza è non-vuota, dopodiché con un ciclo `for` vengono stampati tutti gli elementi della sequenza. Un elemento dichiarato `String` è come se fosse una sequenza di caratteri perciò posso stamparne le lettere come nell'esempio. Viene poi creato il metodo `list` che stampa i valori passati in ingresso insieme ai rispettivi indici. La classe `LengthsIterator` che implementa `Iterator<Integer>` ha il metodo `next()` che restituisce la lunghezza dell'elemento oppure `finished`, oggetto di tipo `Finished`. Poco sotto, la classe `Lengths` rappresenta un semplice iteratore di valori di cui ne stampa di nuovo la lunghezza in termini di caratteri.

4.4 Null values and Union types

```
String? s1 = null;
String? s2 = "This is a String.";

print(s1);
print(s2);

if (exists s2) {
```

```
    print("'``s2``' has length ``s2.size``");
}


String safe = s2 else "NULL!";
Integer? len = s2?.size;
print("'``safe``' has length ``(len else 0)``");
print("'``s1 else "NULL!"``' has length ``s1?.size else 0``");

String? check = (10.0^0.5 > 3.0) then "ok";
Integer i = safe.startsWith("This") then 1 else -1;
print("check='``check else "NULL!"``', i=``i``");


Object o = s2 else "null";
if (is String o) {
    String s = o;
    print(s);
}


variable String|Integer x = "initial";
print("x has type ``className(x)``");
x = 5;
print("Now it has type ``className(x)``");
if (is Integer y = x) {
    print("y is an Integer with value ``y``");
}

String|Null optional = s1;
if (is Null optional) {
    print("optional is null");
}
```

In questo esempio di codice vengono mostrati esempi sui valori nulli e sul tipo di valore *Unione*. Come si può vedere nelle prime due righe di codice, dichiarare una variabile di tipo `String?` fa in modo che la variabile possa assumere sia il tipo `null` che il tipo `String`. Tuttavia non si può usare il tipo `String?` dove è richiesto il tipo `String`, ad esempio per la funzione `.size` che ne mostra la lunghezza, se prima non ne viene testato il tipo attraverso il costrutto `if(exists..)`. All'interno del costrutto `if(exists..)`, dopo che ne è stato testato il valore `null`, la variabile `s2` assume il tipo `String`, mentre fuori dal costrutto ha il tipo `String?`. Ci sono altri modi per operare coi

valori nulli; ad esempio con il codice: `String safe = s2 else "NULL!" safe` assume il valore di default se `s2` è null invece con la seguente dichiarazione: "Integer? len = s2?.size" `len` è `null` se `s2` è `null`. L'operatore `else` può essere combinato con `then`. Il tipo di Unione, come si può vedere con la variabile `x`, permette alla variabile di essere sia di tipo `String` che di tipo `Integer`; perciò è permesso assegnare ad essa una stringa e successivamente un intero se necessario. Il tipo `String?` è una shortcut per del tipo `String|Null`

Conclusioni

Il linguaggio Ceylon, è stato creato allo scopo di essere eseguito dalla Java Virtual Machine (JVM). A tal proposito, per la realizzazione di Ceylon sono stati analizzati altri linguaggi di programmazione operanti sulla stessa piattaforma, in particolare Java. Entrambi sono linguaggi di programmazione *ad alto livello*, *orientati agli oggetti*, *a tipizzazione statica*, progettati per essere il più possibile indipendenti dalla piattaforma di esecuzione. Il linguaggio Ceylon introduce caratteristiche nuove, mancanti in Java, come i tipi di unione, intersezione, il *bottom type* `Nothing` e un completo sistema di modularità (introdotte in Java dalla versione 9). In secondo luogo Red Hat ha cercato di migliorare elementi che, potessero essere inclini a errori in Java, come la gestione dei valori di tipo `Null`, migliorando la *type-safety* di Java; più in generale Ceylon migliora la sintassi di alcuni costrutti, definiti "goffi" o "verbosi" e ne presenta una versione più compatta; ma può comunque riusare le libreria Java esistenti. Ceylon purtroppo ha avuto una scarsa diffusione, a causa della piccola comunità di sviluppatori, ma anche a causa di linguaggi simili come Kotlin o Scala che hanno avuto maggior successo.

Ringraziamenti

Concludo il mio percorso triennale in questa facoltà con la consapevolezza che quello che ho imparato in questi lunghi anni mi ha cresciuto a livello di conoscenze e a livello interiore. Ringrazio tutte le persone che mi hanno accompagnato e supportato durante il triennio, dalla mia famiglia, che mi ha sempre aiutato, ai miei colleghi e amici, disponibili a ogni ora per chiarimenti, che hanno percorso questo viaggio accanto a me. Nello specifico ringrazio Nicola, Gianni e Orjada, colleghi, compagni e amici, sempre presenti nelle mie giornate, con i quali ho potuto svolgere gran parte dei progetti richiesti dal corso in questi anni, che hanno condiviso insieme a me gioie e dolori; e ringrazio in particolar modo Sara, per la pazienza, per avermi sempre spronato a dare il meglio e per non aver mai smesso di credere in me. Un pensiero va anche a tutti gli amici e colleghi non citati, che mi hanno tirato su il morale ogni volta che pensavo di non farcela. Un ringraziamento speciale va ai Professori Mirko Viroli e Danilo Pianini che mi hanno dato la possibilità di presentare questo elaborato e mi hanno aiutato nonostante il tempo fosse limitato.

Bibliografia

[1] Nada Amin and Ross Tate. Java and scala's type systems are unsound: The existential crisis of null pointers. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 838–848, New York, NY, USA, 2016. ACM.

[2] Julia Belyakova. Language support for generic programming in object-oriented languages: Peculiarities, drawbacks, ways of improvement. In Fernando Castor and Yu David Liu, editors, *Programming Languages*, pages 1–15, Cham, 2016. Springer International Publishing.

[3] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the java programming language. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '98, pages 183–200, New York, NY, USA, 1998. ACM.

[4] Patrice Chalin and Perry R. James. Non-null references by default in java: Alleviating the nullity annotation burden. In Erik Ernst, editor, *ECOOP 2007 – Object-Oriented Programming*, pages 227–247, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[5] Atsushi Igarashi and Mirko Viroli. On variance-based subtyping for parametric types. In *European Conference on Object-Oriented Programming*, pages 441–469. Springer, 2002.

[6] Adriaan Moors, Frank Piessens, and Martin Odersky. Generics of a higher kind. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA '08, pages 423–438, New York, NY, USA, 2008. ACM.

[7] Jaime Niño. The cost of erasure in java generics type system. 2006.

[8] Mads Torgersen, Christian Plesner Hansen, Erik Ernst, Peter von der Ahé, Gilad Bracha, and Neal M. Gafter. Adding wildcards to the java programming language. In *SAC*, 2004.