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

Tesi di Laurea

ANALISI COMPARATIVA DI RUST E JAVA: MEMORIA E POLIMORFISMO

NICOLA PAPINI

Relatore/Relatrice: Lorenzo Bettini

Anno Accademico 2024-2025

CONTENTS

LIST OF FIGURES

"Se avessi avuto più tempo, avrei scritto una lettera più breve."
— Blaise Pascal

1

INTRODUZIONE

The introduction is usually a short chapter that can be read in less than 10 minutes.

The goal of the Introduction is to engage the reader (why you should keep reading). You don't have to discuss anything in detail. Rather, the goal is to tell the reader:

- what is the problem dealt with and why the problem is a problem;
- what is the particular topic you're going to talk about in the thesis;
- what are your goals in doing so, and what methodology do you follow;
- what are the implications of your work.

The above points will be further expanded in the following chapters, so only a glimpse is essential.

It is customary to conclude the Introduction with a summary of the content of the rest of the thesis. One or two sentences are enough for each chapter.

2

CONCETTI FONDAMENTALI

In questo capitolo vengono introdotti i concetti fondamentali per la comprensione del lavoro svolto in questa tesi. In particolare, verrà fornita una panoramica sui tipi e sulla sintassi fondamentale di Rust, eseguendo un confronto con Java.

2.1 TIPI

Sia Java che Rust sono linguaggi *statically typed* (staticamente tipati), il che significa che ogni variabile ha un tipo conosciuto a compile time. Inoltre, sono anche *strongly typed* (fortemente tipati), ossia che il type system del linguaggio applica rigorosamente regole di tipo, impedendo operazioni tra tipi incompatibili e limitando le operazioni che possono essere eseguite su un dato tipo. Questi concetti sono fondamentali per prevenire errori di tipo a runtime.

In Java sono presenti i seguenti tipi di tipi:

- Tipi Primitivi: sono i tipi predefiniti dal linguaggio e includono:
  - Tipi Numerici: `byte`, `short`, `int`, `long`, `char`, `float`, `double`. I primi cinque sono *IntegralTypes* (interi), mentre gli ultimi due sono *FloatingPointTypes* (virgola mobile).
  - Tipo Booleano: `boolean`.

  Variabili di tipi primitivi contengono direttamente i valori e non possono essere `null`.
- Tipi Riferimento: sono memorizzati come riferimenti a oggetti nell' heap. Includono quattro diversi tipi di tipo riferimento:
  - Tipo Classe.
  - Tipo Interfaccia.

  - Variabili di tipo: parametro di tipo utilizzato nella programmazione generica.
  - Tipo Array: definito utilizzando il nome di un tipo seguito dalle parentesi quadre `[]`. Ad esempio `int[]`, `String[]`.

- Tipi parametrici: tipo Classe o Interfaccia nella forma `C<T1, T2, ..., Tn>`, dove `C` è il nome della classe o interfaccia e `T1, T2, ..., Tn` sono i parametri di tipo.

In Rust sono presenti i seguenti tipi di tipi:

- Tipi primitivi:
  - Tipo Booleano: `bool`.
  - Tipi Numerici: `u8`, `u16`, `u32`, `u64`, `u128`, `i8`, `i16`, `i32`, `i64`, `i128`, `f32`, `f64`. I primi sei sono interi, mentre gli ultimi due sono virgola mobile.
  - Tipo Testuale: `char` o `str`.
  - Tipo Never: `!`, rappresenta un tipo che non ha valori. Può essere utilizzato solo come tipo di ritorno di una funzione. Rappresenta il risultato di una computazione che non viene mai completata.

- Tipi Sequenza:
  - Tipo Tupla: raggrupa insieme un insieme di valori di tipo diverso in un unico tipo composito. Ha dimensione fissa che equivale alla dimensione nel momento in cui viene dichiarata. Ad esempio, `(i32, f64, char)` è una tupla che contiene un intero a 32 bit, un numero in virgola mobile a 64 bit e un carattere.
  - Tipo Array: tipo che rappresenta una collezione di elementi dello stesso tipo con una dimensione fissa. Ad esempio, `[i32; 5]` è un array di cinque interi a 32 bit.
  - Tipo Slice: tipo che rappresenta una vista dinamica di una sequenza di elementi di tipo `T`. La sintassi è `[T]`. Solitamente viene utilizzata dietro a un tipo Puntatore.

- Tipi definiti dall'utente: `struct`, `enum`, `union`. Trattati nel dettaglio nelle sezioni successive 2.2 e 2.3.

- Tipi Funzione: tipi che rappresentano funzioni e chiusure (*closure*). Una chiusura è una funzione anonima che può catturare variabili dall'ambiente circostante. La sintassi per definire una chiusura è `|parametri| espressione`.
- Tipi Puntatore: questo tipo comprende tre diversi tipi:
  1. Tipo Riferimento: vedi sezione 3.5.
  2. Tipo Raw Pointer: Per un tipo `T` si hanno `*const T` e `*mut T`. Sono puntatori senza garanzie di sicurezza e liveness.
  3. Tipo Smart Pointer: puntatori che forniscono funzionalità aggiuntive rispetto ai puntatori grezzi come la gestione automatica della memoria. I più comuni sono `Box<T>`, `Rc<T>` e `Arc<T>`.
  4. Tipo Puntatore a funzione: dichiarati come `fn(&T) -> U` per una funzione che prende un riferimento a `T` e restituisce un valore di tipo `U`.
- Tipi Trait (tratto, nel senso di caratteristica):
  - Tipi Trait Object (vedi sezione 4.5.1).
  - Tipi `impl Trait`: utilizzati in posizione di ritorno o come parametro di funzione per indicare che la funzione restituisce o accetta un tipo che implementa un certo trait. È una forma alternativa per indicare un tipo generico con un vincolo di trait (vedi sezione 4.4).

2.2 STRUCT

Una `struct` in Rust è un tipo di dato definito dall'utente che consente di raggruppare insieme più valori correlati in un'unica entità. In particolare, i valori all'interno di una struct possono essere di tipi diversi e, insieme al nome associato al valore, sono chiamati *campi*. Per definire una struct si utilizza la keyword `struct` come segue:

```
struct Book {
    title: String,
    pages: u32,
}
```

Per ottenere un comportamento analogo in Java si definisce una classe nel seguente modo:

```java
class Book {
    String title;
    int pages;
}
```

Come una struct Rust, una classe Java definisce un tipo di dato che può contenere più valori di tipo diverso, chiamati *campi*. Una struct può essere vista come una versione più leggera di una classe Java, in quanto non supporta l'ereditarietà e non può avere metodi direttamente associati ad essa (i metodi sono definiti separatamente in un blocco `impl`, come descritto più avanti).

Per creare un'istanza di una struct, si utilizza la seguente sintassi:

```rust
let lotr = Book {
    title: String::from("Il Signore degli Anelli"),
    pages: 1216
};
```

In Rust, il costruttore è implicito e si basa sulla sintassi di inizializzazione dei campi. In Java, invece, è necessario definire un costruttore esplicitamente come un metodo d'istanza all'interno della classe:

```java
class Book {
    String title;
    int pages;

    // Costruttore
    public Book(String title, int pages) {
        this.title = title;
        this.pages = pages;
    }
}
```

e successivamente si può creare un'istanza della classe utilizzando la keyword `new`:

```java
Book lotr = new Book("Il Signore degli Anelli", 1216);
```

In particolare, Rust non supporta l'inizializzazione di default dei campi di una struct, quindi è necessario fornire un valore per ogni campo al momento della creazione dell'istanza. Ad esempio:

```rust
let hp = Book {
    title: String::from("Harry Potter")
};
```

Questo codice genererebbe un errore di compilazione poiché il campo `pages` non è stato inizializzato. In Java, invece, se non si fornisce un valore per un campo, esso viene inizializzato con un valore di default (ad esempio, 0 per i tipi numerici e `null` per i tipi riferimento). Quindi il seguente codice Java sarebbe valido:

```
// assumendo che esista un costruttore
// public Book(String title)
Book book2 = new Book("Harry Potter");
```

In particolare, poiché Java supporta l'overloading, si possono avere costruttori in overload consentendo l'instanziazione di un oggetto in più modi. Questo è permesso anche grazie alla semantica di auto inizializzazione di Java. In generale, possiamo dire che l'approccio di Rust è molto più rigoroso e sicuro, infatti l'approccio Java può portare a errori di runtime se non si presta attenzione a come vengono inizializzati i campi.

Per accedere ai campi di una struct, e quindi potenzialmente cambiarne lo stato, si utilizza la notazione puntata come segue:

```
let lotr_title = lotr.title;
let lotr_pages = lotr.pages;
```

Da osservare che le variabili in Rust sono immutabili di default, quindi per modificare un campo di una è necessario dichiarare la variabile come mutabile utilizzando la parola chiave `mut`:

```
let mut lotr = Book {
    title: String::from("Il Signore degli Anelli"),
    pages: 1216
};
lotr.pages = 1300;
```

In Java, la modifica del valore di una variabile d'istanza di una classe avviene in modo simile, tramite notazione puntata, a patto che chi modifica lo stato dell'oggetto sia autorizzato a farlo: ad esempio il campo è dichiarato `public` (e non `final`) oppure viene fornite un metodo della classe che consente di modificare il campo.

Nel lavoro svolto in questa tesi, si fa anche ampio utilizzo delle *Unit-Like Structs*: strutture che non hanno campi ma possono essere utili quando si desidera definire un tipo che rappresenta un concetto o un'entità senza la necessità di memorizzare dati specifici. Un esempio di utilizzo di una unit-like struct è il seguente:

```rust
struct MyStruct;

// Per istanziare una unit-like struct
// non sono necessarie parentesi graffe
let instance = MyStruct;
```

2.2.1 *Funzioni e metodi*

In Rust, una funzione si definisce con la parola chiave `fn`, seguita dal nome, da una lista di parametri racchiusi tra parentesi tonde e, se presente, dal tipo di ritorno. Ad esempio:

```rust
fn add(a: i32, b: i32) -> i32 {
    a + b
    // L'ultima espressione senza punto
    // e virgola è il valore di ritorno
}
let result = add(5, 10); // Chiamata della funzione
println!("Result: {}", result); // Stampa: Result: 15
```

La funzione `add()` prende due parametri di tipo `i32` e restituisce la loro somma, anch'essa di tipo `i32`. Il tipo di ritorno è specificato dopo la freccia `->`.

Così come possiamo definire funzioni "libere", è possibile associare funzioni a una `struct` (o un `enum` come vedremo nella prossima sezione) tramite l'uso di un blocco `impl`. Queste funzioni prendono il nome di *funzioni associate* poiché sono legate a una specifica struct. Le funzioni associate che prendono come primo parametro `self` sono chiamate *metodi* e possono essere chiamati tramite notazione puntata. `Self` è una keyword Rust che rappresenta l'istanza della struct su cui il metodo viene chiamato. Ad esempio, per la struct `Book` definita in precedenza, un blocco `impl` potrebbe apparire come nel listato 1.

Il parametro `self` può essere passato in diversi modi, a seconda di come si desidera utilizzare l'istanza della struct all'interno del metodo, spesso è passato tramite riferimento per motivi di ownership e borrowing (vedi sezione 3.5). In Java, esiste un concetto simile a `self` che è `this`: un riferimento implicito all'istanza corrente della classe. A differenza di `self`, `this` è sempre disponibile e non è necessario dichiararlo esplicitamente. In generale, i metodi di una struct definiscono il comportamento associato a un'istanza di quella struct.

```rust
impl Book {
    fn describe(&self) {
        println!("\"{}\" has {} pages.", self.title, self.pages);
    }

    fn is_long(&self) -> bool {
        self.pages > 300
    }
}

fn main() {
    let lotr = Book {
        title: String::from("Il Signore degli Anelli"),
        pages: 1216
    };
    lotr.describe(); // Chiamata del metodo
}
```

Listato 1: Esempio di metodo in Rust.

Invece, le funzioni associate che non prendono `self` come primo parametro sono spesso usate come costruttori o funzioni di utilità legate alla struct ma non a una specifica istanza. Ad esempio:

```rust
impl Book {
    fn new(title: String, pages: u32) -> Self {
        Self { title, pages }
    }
}
```

Listato 2: Esempio di funzione associata in Rust.

`Self` in posizione di tipo di ritorno di una funzione associata si riferisce al tipo della struct stessa, in questo caso `Point`. Per chiamare questo tipo di funzioni associate, si utilizza la sintassi `NomeStruct::nome_funzione()`:

```rust
let book = Book::new(String::from("1984"), 328);
```

In particolare, nel listato 2, si è utilizzato un forma più breve e leggera per inizializzare i campi della struct. Questa sintassi è utilizzata quando i nomi dei parametri della funzione corrispondono ai nomi dei campi della struct. In questo caso, `Self { title, pages }` è equivalente a `Self`

```java
class Book {

    ...

    public void describe() {
        System.out.println("\""
                           + title
                           + "\" has "
                           + pages
                           + " pages.");
    }

    public boolean isLong() {
        return pages > 300;
    }
}

public class Main {
    public static void main(String[] args) {
        Book lotr = new Book("Il Signore degli Anelli", 1216);
        lotr.describe(); // Chiamata del metodo
    }
}
```

Listato 3: Listato 1 in Java.

`{ title: title, pages: pages }`. Java non mette a disposizione questo tipo di funzionalità, anzi in Java lo sviluppatore è tenuto a disambiguare esplicitamente i nomi dei parametri del costruttore dai nomi delle variabili d'istanza della classe utilizzando la parola chiave `this`.

In Java, il concetto di funzione associata è superfluo poiché le funzioni sono sempre definite, e quindi associate, all'interno di una classe. I metodi Rust sono analoghi ai metodi di istanza in Java, ossia metodi che appartengono a un'istanza specifica di una classe. Quindi, il listato 1 potrebbe essere tradotto in Java con il codice riportato nel listato 3.

Invece, le funzioni "libere" in Java sono implementate come metodi statici. I metodi statici appartengono alla classe stessa piuttosto che a un'istanza specifica della classe. Questi sono spesso utilizzati come costruttori alternativi, come è il caso del pattern creazionale *Static Factory Methods* [3], o per definire funzioni di utilità che non richiedono l'accesso ai dati di un'istanza specifica della classe.

2.3 ENUM

Un `enum` in Rust è un tipo di dato definito dall'utente che consente di rappresentare una variabile che può assumere uno tra un insieme finito di valori, chiamati *varianti*. Un enum Rust è dichiarato utilizzando la parola chiave `enum` come segue:

```
enum Shape {
    Circle,
    Rectangle,
    Triangle,
    Point,
}
```

In Java, un enum è dichiarato in maniera analoga:

```
enum Shape {
    CIRCLE,
    RECTANGLE,
    TRIANGLE,
    POINT;
}
```

In entrambi i linguaggi è possibile associare dati alle varianti di un enum. Tuttavia, c'è una differenza sostanziale tra la flessibilità con cui si può eseguire questa associazione.

- In Java, un enum è classe speciale in cui le varianti sono istanze singleton statiche e finali di quella classe. Questo significa che ogni variante deve essere inizializzata esplicitamente nel momento della sua dichiarazione o all'interno di un blocco `static` . Ad esempio, se volessimo associare un numero di lati a ogni variante dell'enum `Shape`, in Java potremmo farlo nel seguente modo:

```
enum Shape {
    CIRCLE(0), RECTANGLE(4), TRIANGLE(3), POINT(0);

    private int sides;

    private Shape(int sides) {
        this.sides = sides;
    }
}
```

La limitazione di questo approccio è che tutte le varianti dell'enum devono avere lo stesso insieme di dati associati, il che può essere restrittivo in alcuni casi. Ad esempio, se volessimo associare dati diversi a ogni variante, come il raggio per il cerchio, la larghezza e l'altezza per il rettangolo, e i lati per il triangolo, non sarebbe possibile farlo direttamente con gli enum di Java.

Inoltre, il valore associato a una variante di un enum in Java è immutabile, quindi non può essere cambiato dopo l'inizializzazione, che avviene nel momento della dichiarazione.

- Rust, invece, consente di associare dati diversi, sia in tipo che in numero, a ogni variante, rendendo gli enum Rust molto più flessibili e potenti. Ad esempio, questo codice è possibile in Rust:

```
enum Shape {
    Circle(f64),              // raggio
    Rectangle(f64, f64),      // larghezza, altezza
    Triangle(f64, f64, f64),  // lati
    Point,
}
```

Listato 4: Esempio di enum in Rust con varianti che contengono dati.

In particolare, l'esempio sopra può essere reso più chiaro utilizzando nomi per i campi tramite la seguente sintassi:

```
enum Shape {
    Circle { radius: f64 },
    Rectangle { width: f64, height: f64 },
    Triangle { a: f64, b: f64, c: f64 },
    Point,
}
```

Inoltre, a differenza di Java, è possibile definire il valore dei dati associati a una variante nel momento della sua istanziazione:

```
let circle = Shape::Circle { radius: 5.0 };
```

Un'altra differenza molto importante tra gli enum di Java e Rust è il modo in cui si esegue il pattern matching sulle varianti. Per pattern matching si intende la capacità di verificare quale variante di un enum è attualmente in uso e di estrarre i dati associati a quella variante. In Rust, il pattern matching è eseguito tramite la parola chiave `match`. Ad

esempio, considerando il listato 4, si potrebbe scrivere una funzione per calcolare l'area di una figura geometrica come segue:

```
fn area(shape: &Shape) -> f64 {
    match shape {
        Shape::Circle(radius) => 3.14 * radius * radius,
        Shape::Rectangle(width, height) => width * height,
        Shape::Triangle(a, b, c) => {
            // Compute area for triangle
        }
        Shape::Point => 0.0,
    }
}
```

In particolare, in Rust, il pattern matching è esaustivo, ossia il compilatore verifica che tutti i casi possibili siano coperti, garantendo così la sicurezza del codice. Ad esempio:

```
fn area(shape: &Shape) -> f64 {
    match shape {
        Shape::Circle(radius) => 3.14 * radius * radius,
        Shape::Rectangle(width, height) => width * height,
        Shape::Point => 0.0,
    }
}
// Questo codice non compila perché manca il caso per Triangle
```

In Java, il pattern matching sugli enum è meno potente e richiede l'uso di costrutti come `switch`. A differenza di Rust, Java non garantisce che tutti i casi siano coperti, il che può portare a errori di runtime se una variante non viene gestita. Ad esempio:

```
enum Status {
    SUCCESS,
    FAILURE,
    PENDING
}

public String getStatusMessage(Status status) {
    switch (status) {
        case SUCCESS:
            return "Operation was successful.";
        case FAILURE:
            return "Operation failed.";
```

```
            // Manca il caso per PENDING
            default:
                return "Unknown status.";
        }
    }
```

Questo codice compila, ma se venisse passato il valore `PENDING` si verificherebbe un comportamento inatteso: il messaggio di stato restituito sarà `"Unknown status."`, anche se lo stato in realtà è noto. Inoltre, in Java non è possibile estrarre dati associati a una variante di un enum all'interno di un costrutto `switch`, a meno di definire metodi aggiuntivi all'interno dell'`enum`.

Quindi, si può notare come gli enum di Rust offrano funzionalità molto più potenti e flessibili: è possibile associare dati diversi a ciascuna variante e accedervi direttamente durante il pattern matching. Per ottenere un comportamento simile in Java, sarebbe necessario ricorrere a soluzioni più complesse, ad esempio usando interfacce `sealed` e `record`:

```
sealed interface Shape permits Circle, Rectangle, Triangle, Point {}

record Circle(double radius) implements Shape {}
record Rectangle(double width, double height) implements Shape {}
record Triangle(double a, double b, double c) implements Shape {}
record Point() implements Shape {}

public double area(Shape shape) {
    return switch (shape) {
        case Circle c -> 3.14 * c.radius() * c.radius();
        case Rectangle r -> r.width() * r.height();
        case Triangle t -> {
            // Compute area for triangle
        }
        case Point p -> 0.0;
    };
}
```

In questo esempio, si definisce un'interfaccia `Shape` che è `sealed`, il che significa che solo le classi specificate (in questo caso, i record) possono implementarla. Ogni "variante" è rappresentata come un `record` che implementa l'interfaccia `Shape`. Il pattern matching viene eseguito utilizzando un costrutto `switch` che è in grado di estrarre i dati associati a ciascun record. Tuttavia, questo approccio è più verboso e complesso rispetto alla semplice definizione di un enum in Rust. Tuttavia, in questo

caso, Java garantisce l'esaustività del pattern matching grazie all'uso di interfacce `sealed`.

Infine, analogamente a quello che succede per le struct, anche gli enum in Rust possono avere metodi associati definiti all'interno di un blocco `impl`. Ad esempio:

```rust
impl Shape {
    fn area(&self) -> f64 {
        match self {
            /* ... */
        }
    }
}

fn main() {
    let circle = Shape::Circle { radius: 5.0 };
    println!("Area: {}", circle.area());
}
```

In Java, i metodi associati a un enum sono definiti all'interno della dichiarazione dell'enum stesso come avviene per una qualsiasi classe.

3

GESTIONE DELLA MEMORIA

Questo capitolo esplorerà nel dettaglio come Java e Rust affrontano il tema della gestione della memoria. Entrambi i linguaggi, a differenza di C/C++, sollevano il programmatore dalla responsabilità di gestire manualmente la deallocazione della memoria, adottando però filosofie e approcci differenti:

- Java affida al *Garbage Collector (GC)* il compito di liberare la memoria automaticamente, semplificando lo sviluppo ma introducendo overhead.
- Rust elimina il GC grazie a un sistema di *ownership* e *borrowing*, garantendo deallocazione deterministica e sicurezza a compile-time.

3.1 L'APPROCCIO DI JAVA

Java è un linguaggio di programmazione progettato per essere semplice, portabile e sicuro, con una gestione della memoria che mira a ridurre la complessità e prevenire errori comuni legati all'uso diretto delle risorse. Java raggiunge questi obiettivi affidando l'intera gestione della memoria alla Java Virtual Machine (JVM). Gli aspetti fondamentali dell'approccio Java sono:

1. L'allocazione automatica della memoria tramite JVM: gli oggetti vengono creati dinamicamente nell'heap senza che il programmatore debba preoccuparsi di allocare la memoria manualmente.
2. La deallocazione automatica della memoria tramite il garbage collector, una componente della JVM che si occupa di individuare e liberare la memoria occupata da oggetti non più raggiungibili, prevenendo così problemi comuni in altri linguaggi, come memory

leak e dangling pointer[1]. Questo, di fatto, toglie al programmatore la responsabilità collegata al dover gestire manualmente la deallocazione della memoria, riducendo la probabilità di errori nel codice.

3. La prevenzione di comportamenti indefiniti a run-time, attraverso un controllo rigoroso dell'accesso alla memoria a runtime: ad esempio, accessi a riferimenti `null` generano eccezioni gestibili da parte dello sviluppatore.

Tuttavia, l'approccio automatico della gestione della memoria porta alcuni svantaggi per quanto riguarda le prestazioni del programma. Il garbage collector, infatti, introduce un overhead significativo, poiché deve periodicamente eseguire la scansione della memoria per identificare gli oggetti non più raggiungibili e liberare la memoria occupata da essi. Questo processo può causare pause impreviste nell'esecuzione del programma, che possono essere problematiche quando si richiedono alte prestazioni. La JVM moderna implementa algoritmi di garbage collection avanzati [2] per cercare di ridurre al minimo le interruzioni e ottimizzare le prestazioni, ma il costo di queste operazioni rimane comunque un fattore da considerare.

È importante sottolineare, inoltre, che, nonostante il garbage collector riduca notevolmente il rischio di errori di memoria, non elimina completamente la possibilità di *memory leak* [1]. Un memory leak in Java si verifica quando un oggetto non più necessario continua a essere referenziato, impedendo al garbage collector di liberare la memoria da esso occupata. Alcuni casi più comuni di memory leak in Java includono:

- Memory leak causati da campi dichiarati come `static`. In Java, i campi `static` sono associati alla classe e non all'istanza, quindi rimangono in memoria finché la classe è caricata dalla JVM. Questo, solitamente, coincide con l'intero ciclo di vita dell'applicazione. Ad esempio:

```java
public class MemoryLeakExample {
    private static List<String> list = new ArrayList<>();

    public static void populateList() {
        for (int i = 0; i < 1000000; i++) {
            list.add("Item " + i);
```

1 Un dangling pointer è un riferimento a una variabile che è stata deallocata e quindi non più valido.

```
        }
    }

    public static void main(String[] args) {
        new MemoryLeakExample().populateList();
    }
}
```

In questo caso, una volta che il metodo `populateList()` termina la sua esecuzione, la memoria occupata da `list` non viene liberata, poiché è un campo `static`. Questo non si verificherebbe se `list` fosse una variabile d'istanza, poiché la memoria da essa occupata verrebbe liberata quando l'istanza della classe viene raccolta dal garbage collector.

- Memory leak causati da risorse non chiuse correttamente. In Java, oggetti che si riferiscono a risorse di sistema, come file o connessioni di rete, devono essere chiusi esplicitamente per liberare la memoria e le risorse associate. In caso contrario, possono causare memory leak. Ad esempio:

```
public class MemoryLeakExample {
    public static void main(String[] args) {
        try {
            Scanner sc = new Scanner(new File("file.txt"));
            // Logica di utilizzo dello Scanner
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
        // Memory leak causato dallo Scanner non chiuso
    }
}
```

3.2 L'APPROCCIO DI RUST

Rust è un linguaggio di programmazione moderno progettato per garantire sicurezza nella gestione della memoria senza fare uso di un garbage collector. Rust ha due obiettivi principali:

1. Garantire che il programma sia privo di comportamenti indefiniti, ovvero situazioni in cui il programma può agire in modo imprevedibile. Un esempio tipico è l'accesso a memoria non valida, che può

portare all'esecuzione di codice con dati non inizializzati o causare errori di memoria come segmentation fault.

2. Eseguire la prevenzione di comportamenti indefiniti a compile-time, piuttosto che a run-time. Questo significa che il compilatore di Rust è in grado di rilevare e segnalare errori di memoria prima che il programma venga eseguito, riducendo il rischio di bug e di errori durante l'esecuzione.

L'approccio di Rust consente di evitare interamente classi di errori comuni nei linguaggi tradizionali: buffer overflow, dangling pointer e race condition sui dati condivisi. Inoltre, poiché questi controlli vengono effettuati a compile-time, Rust riduce drasticamente la necessità di controlli a run-time, migliorando le prestazioni senza sacrificare la sicurezza.

Rust non può prevenire tutti i possibili bug relativi alla gestione della memoria ma le metodologie messe in atto rendono i programmi scritti in Rust significativamente più sicuri rispetto a quelli sviluppati in linguaggi con meno controlli. Un esempio concreto è fornito da Google [7], che ha introdotto il linguaggio nello sviluppo di Android 13. In particolare, circa il 21% del nuovo codice introdotto in Android 13 è stato scritto in Rust, e, alla data della pubblicazione dell'articolo, non sono state scoperte vulnerabilità di sicurezza legate alla memoria in questo codice. Questo è un risultato significativo che dimostra come gli obiettivi prefissati dagli sviluppatori di Rust siano stati raggiunti nella pratica.

Rust realizza questi obiettivi attraverso un sistema basato sui concetti di *ownership* (proprietà) e *borrowing* (prestito). Concetti fondamentali che verrano affrontati in dettaglio nelle prossime sezioni.

3.3 STACK E HEAP

Sia Java che Rust utilizzano due aree di memoria principali: lo stack e l'heap, ma la loro gestione è profondamente diversa, riflettendo i diversi modelli di memoria adottati dai due linguaggi.

Lo stack è un'area di memoria strutturata secondo una struttura dati stack LIFO (Last In, First Out). La memoria stack è contigua e i dati memorizzati al suo interno sono in posizione fissa rispetto allo stack pointer, un puntatore che punta all'ultimo elemento inserito. Questo permette un accesso rapido ai dati usando indirizzi di memoria calcolati in modo semplice tramite un offset rispetto allo stack pointer. Inoltre, allocazione e deallocazione della memoria stack sono molto veloci poiché avvengono spostando lo stack pointer, avanti o indietro, di un numero di

byte opportuno rispetto alla dimensione del dato e all'architettura della CPU.

L'heap, al contrario, è un'area di memoria in cui i dati possono essere allocati in qualsiasi sua posizione. L'allocazione e la deallocazione della memoria heap richiedono operazioni più complesse rispetto allo stack, poiché il sistema operativo deve tenere traccia degli spazi liberi e occupati. Questo può portare a un utilizzo meno efficiente della memoria (frammentazione) e a un accesso più lento ai dati rispetto allo stack.

In Java, l'allocazione della memoria è fortemente automatizzata. Ogni volta viene creato un oggetto, tramite la keyword `new`, viene allocata dinamicamente memoria heap nel quale sarà memorizzato l'oggetto. L'uso dello stack è limitato a variabili di tipo primitivo e variabili locali. Al contrario, Rust adotta un modello più esplicito e flessibile. In Rust, la variabili possono essere allocate sia nello stack che nell'heap, a seconda dalla conoscenza a compile time delle dimensioni del dato:

- Se la variabile ha una dimensione fissa nota a compile time, viene allocata nello stack. È possibile allocare nell'heap anche variabili di dimensione fissa attraverso `Box`[2].
- Se la variabile ha una dimensione variabile o non nota a compile time, viene allocata nell'heap.

Questa è una differenza fondamentale rispetto a Java, perchè permette allo sviluppatore di avere più controllo su dove vengono allocati i dati, permettendo ottimizzazioni specifiche per le esigenze del programma.

Ad esempio, sia in Java che in Rust, gli array hanno una dimensione fissa. Tuttavia, in Java, gli array sono allocati nell'heap e sono referenziati da variabili nello stack, mentre in Rust, poiché si conosce la loro dimensione a compile time, vengono allocati nello stack. Questo rende l'accesso agli elementi dell'array di Rust più veloce.

```java
int[] arr = {1, 2, 3, 4, 5}; // Array allocato nell'heap
System.out.println("Il primo elemento e': " + arr[0]);
```

```rust
let arr = [1, 2, 3, 4, 5]; // Array allocato nello stack
println!("Il primo elemento e': {}", arr[0]);
```

2 `Box<T>` è uno smart pointer fornito dalla standard library di Rust che consente di allocare un valore di tipo `T` sull'heap.

3.4 OWNERSHIP

L'ownership è un concetto fondamentale di Rust il quale può essere definito come un insieme di regole che il compilatore controlla per garantire una corretta gestione della memoria. Questo significa sia garantire che non ci siano errori di memoria a run-time, sia che la memoria inutilizzata venga rilasciata correttamente per non terminare lo spazio di memoria disponibile.

L'obiettivo principale dell'ownership è, quindi, quello di gestire la memoria heap tenendo traccia di quali parti di codice utilizzano valori contenuti in essa, minimizzare valori duplicati e garantire che la memoria venga rilasciata quando non è più necessaria.

L'ownership si basa su tre regole principali:

1. Ogni valore in Rust ha un *owner* (proprietario), ovvero una variabile che ne detiene la proprietà.
2. Un valore può avere un solo owner alla volta.
3. Quando l'owner di un valore esce dallo scope, il valore viene automaticamente rilasciato dalla memoria.

Consideriamo un caso banale in cui si crea una variabile all'interno di uno scope:

```
{
    let s = String::from("Hello");
}
```

In questo caso, secondo la regola 3, quando la variabile `s` esce dallo scope, il valore `"Hello"` viene automaticamente rilasciato dalla memoria. Questo avviene attraverso la funzione `drop` che viene chiamata automaticamente da rust nel momento in cui la variabile esce dallo scope. In Java questo non accade. Dato il seguente codice equivalente in Java:

```
{
    String s = new String("Hello");
}
```

La memoria occupata dalla stringa `"Hello"` non viene rilasciata automaticamente quando `s` esce dallo scope, ma solo quando il garbage collector esegue la raccolta dei valori non più raggiungibili. Già da questo semplice caso si può notare come l'ownership di Rust permetta di avere un controllo più preciso sulla memoria.

Un altro aspetto importante dell'ownership è che, quando si assegna un valore a un'altra variabile, l'ownership viene trasferita. Ad esempio, consideriamo il seguente codice:

```
let s1 = String::from("Hello");
let s2 = s1; // Ownership di s1 viene trasferita a s2
println!("{}", s1); // Errore di compilazione
println!("{}", s2);
```

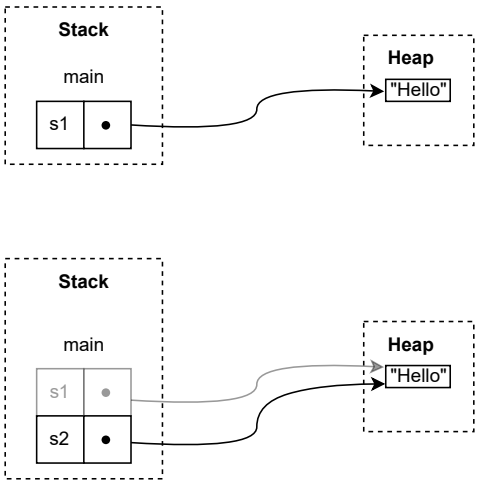


Figura 1: Visualizzazione memoria Rust dopo il trasferimento di ownership.

In questo caso, l'ownership della stringa `"Hello"` viene trasferita da `s1` a `s2`. Come si può vedere in figura 1, dopo il trasferimento, `s1` non è più valida e qualsiasi tentativo di accedervi causerà un errore di compilazione.

In Rust, per variabili il cui valore è contenuto in memoria heap, un'istruzione di copia esegue una *shallow copy* del valore e invalida la variabile originale. Questo comportamento prende il nome di *move*. L'operazione di move è a tutti gli effetti come un trasferimento di proprietà legale, dove il vecchio proprietario non può più accedere alla proprietà venduta. Rust non esegue mai una *deep copy* di una variabile: se il programmatore desidera duplicare effettivamente il contenuto, deve farlo in modo esplicito (ad esempio usando il metodo `clone()`). Quindi, l'operazione di copia base è poco costosa in termini di performance.

Questo non è consistente con quello che accade in Java, dove l' assegnazione di un oggetto a un'altra variabile non invalida quella originale, ma crea una nuova referenza all'oggetto esistente. Entrambe le variabili

possono accedere all'oggetto, condividendone lo stato. In Java, il codice equivalente sarebbe:

```
String s1 = "Hello";
String s2 = s1;
System.out.println(s1); // Valido
System.out.println(s2); // Valido
```

In questo caso, quindi, entrambe le stringhe verrebbero correttamente stampate. L'approccio adottato da Rust è decisamente più restrittivo, ma si tratta di una caratteristica desiderabile: consente infatti di evitare errori comuni legati alla gestione della memoria, come l'accesso a variabili non più valide o la modifica involontaria di dati condivisi.

Ownership e funzioni

Il meccanismo di passaggio degli argomenti a funzione in Rust è strettamente legato al concetto di ownership. Quando si passa una variabile a una funzione, l'ownership di quella variabile viene trasferita alla funzione, rendendo, quindi, la variabile originale non più utilizzabile dopo la chiamata. Questo avviene perché Rust utilizza il *pass-by-value*. Ad esempio, consideriamo il seguente codice:

```
fn main() {
    let s1 = String::from("Hello");
    // Ownership di s1 viene trasferita a print_string
    print_string(s1);
    println!("{}", s1); // Errore di compilazione
}

fn print_string(s: String) {
    println!("{}", s);
}
```

Ciò che accade è: la variabile `s1` viene passata alla funzione `print_string`, ossia `s1` viene copiata in `s` e, quindi, l'ownership dei dati di `s1` passa a `s`. Come risultato, `s1` non è più valida dopo la chiamata alla funzione, e qualsiasi tentativo di accedervi causerà un errore di compilazione.

Java, come Rust, utilizza il *pass-by-value*, ma il passaggio di un oggetto a una funzione non invalida la variabile originale, questo può portare a situazioni sgradevoli. Nel listato 5, la modifica del campo `name` di `p2` influisce anche `p1`, poiché entrambi i riferimenti puntano allo stesso oggetto in memoria. Questo comportamento può causare bug sottili e

```java
class Person {
    private String name;
    public Person(String name) {
        this.name = name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}

public class Main {
    public static void main(String[] args) {
        Person p1 = new Person("Alice");
        Person p2 = p1;
        p2.setName("Bob"); // Modifica il nome di p1 e p2
        System.out.println(p1.getName()); // Stampa "Bob"
    }
}
```

Listato 5: Modifica di un oggetto tramite un riferimento in Java.

difficili da individuare, specialmente in contesti complessi o concorrenti[3]. In Rust, invece, il trasferimento di ownership impedisce il comportamento appena descritto, poiché, una volta che l'ownership è stata trasferita, la variabile originale non può più essere utilizzata.

> Some people say that Java uses "call by reference" for objects. In a language that supports call by reference, a method can replace the contents of variables passed to it. In Java, all parameters—object references as well as primitive type values—are passed by value.
>
> — Cay S. Horstmann [6]

3 È utile notare che, in Java, la keyword `final` può essere utilizzata per dichiarare una variabile come immutabile. Tuttavia, `final` non rende immutabile l'oggetto a cui la variabile si riferisce: i campi dell'oggetto possono ancora essere modificati tramite metodi mutator.

È importante sottolineare che la semantica del *move* in Rust, così come descritta finora, si applica ai tipi di dati che allocano memoria sull'heap, come `String` o `Vec`. In questi casi, un'assegnazione o il passaggio a una funzione comporta il trasferimento dell'ownership, e quindi l'invalidazione del valore originale. Tuttavia, per tipi primitivi e a dimensione fissa, nota a compile time, come gli interi (`i32`, `u64`, etc.), Rust applica una semantica diversa: questi tipi implementano automaticamente il `trait`[4] `Copy`. Ciò significa che, in fase di assegnazione o di passaggio come parametro a una funzione, viene eseguita una copia bit a bit del valore, e l'ownership non viene trasferita.

Di conseguenza, entrambi i valori (l'originale e la copia) restano validi e utilizzabili, senza causare errori di compilazione. Ecco un esempio:

```
fn main() {
    let x = 42;
    print_value(x); // x viene copiato, non spostato
    println!("{}", x); // x è ancora valido
}

fn print_value(n: i32) {
    println!("{}", n);
}
```

Questa distinzione riflette chiaramente la filosofia di Rust nel garantire la sicurezza nell'accesso alla memoria:

- Per i tipi che contengono dati allocati dinamicamente o che possono essere modificati a runtime, Rust usa la semantica del *move*, che impedisce di accedere a un valore dopo che la sua *ownership* è stata trasferita altrove, evitando così potenziali problemi di accesso concorrente e modifiche inattese.

- Per i tipi con dimensione fissa e nota a compile time, che risiedono interamente nello stack, solitamente immutabili per definizione (come interi o booleani), Rust utilizza la semantica del *copy*, poiché la copia bit-a-bit è efficiente e non introduce rischi di inconsistenza o accessi errati.

L'ownership può essere trasferita anche con le funzioni che ritornano un valore. In questo caso, un assegnamento a una variabile di un valore

4 Un trait definisce un insieme di metodi che un tipo può implementare. È simile a un'interfaccia Java: stabilisce un contratto che i tipi devono rispettare. Questo concetto verrà approfondito nel dettaglio nella sezione 4.4.

restituito da una funzione comporta il trasferimento dell'ownership dalla funzione alla variabile. Ad esempio:

```
fn main() {
    let s1 = create_string();
    println!("{}", s1);
}

fn create_string() -> String {
    String s = String::from("Hello from function")
    s // Ownership di s viene trasferita a s1
}
```

L'ownership di una variabile segue sempre lo stesso principio: assegnare il valore a un'altra variabile trasferisce l'ownership. Quando una variabile che include dati nell'heap esce dallo scope, il compilatore chiama automaticamente la funzione `drop` per rilasciare la memoria occupata da quei dati, a meno che l'ownership non sia stata trasferita a un'altra variabile.

```
fn main() {
    let v1 = vec![10, 20, 30];
    let (v2, sum) = sum_vector(v1);
    println!("La somma degli elementi di {:?} è {}", v2, sum);
}

fn sum_vector(v: Vec<i32>) -> (Vec<i32>, i32) {
    let sum = v.iter().sum();
    (v, sum)
}
```

Listato 6: Trasferimento di ownership con ritorno di valore.

Quindi, se volessimo passare un valore a una funzione e riutilizzarlo dopo la chiamata, dovremmo ritornare quel valore al termine della funzione, eventualmente, in aggiunta ad altri valori che la funzione calcola[5] (vedi listato 6).

5 Questo può essere fatto tramite il tipo `Tuple`: un array di dimensione fissa in cui è possibile memorizzare dati di tipo diverso

3.5 BORROWING

È evidente come l'ownership sia un concetto potente per la gestione della memoria, ma che risulta troppo restrittivo e macchinoso in situazioni come quella riportata nel listato 6. Rust, per risolvere questo problema, introduce il concetto di *borrowing*, che consente di prendere in prestito un valore senza trasferirne l'ownership, consentendo una maggiore flessibilità. Il borrowing è realizzato attraverso il concetto di riferimento. Il riferimento in Rust non ha le stesse proprietà di un riferimento in Java:

- In Java un riferimento è l'indirizzo di memoria di un oggetto, e può essere utilizzato per accedere e modificare l'oggetto stesso.

  Inoltre, può assumere il valore `null`, ossia non puntare a nessun oggetto in memoria. Questo ha gravi ripercussioni sulla sicurezza del programma, poiché l'accesso a un riferimento `null` può causare un `NullPointerException` a run-time.

  Lo stesso creatore della nozione di riferimento `null`, Tony Hoare, lo ha definito "billion dollar mistake" [5], a causa dei costi che le aziende devono, e dovranno, sostenere per bug e vulnerabilità dovuti a `null`.

- In Rust, un riferimento è anch'esso l'indirizzo di memoria di un valore (mutabile o immutabile). Tuttavia, a differenza di Java, un riferimento in Rust non può essere `null`. Il compilatore di Rust garantisce che ogni riferimento punti sempre a un valore valido di un tipo specifico per tutta la durata della sua esistenza. Questo dà importanti garanzie di sicurezza, poiché elimina la possibilità di avere errori a run-time legati a riferimenti nulli.

I riferimenti in Rust sono ottenuti utilizzando l'operatore di referenziazione `&` che restituisce il riferimento alla variabile su cui viene applicato. Ad esempio:

```
fn main() {
    let s1 = String::from("Hello");
    print_string(&s1);
}
fn print_string(s2: &String) {
    println!("{}", s2);
}
```

In questo esempio, `s1` è una variabile che si trova nello stack e contiene un valore allocato nell'heap. Pertanto, quando si applica l'operatore `&` a `s1`, si ottiene un riferimento a una variabile sullo stack. Quindi, come mostrato in figura 2, si hanno due livelli di indirezione: `s2` per poter accedere a `"Hello"` deve prima seguire il riferimento a `s1` e poi accedere al valore allocato nell'heap.

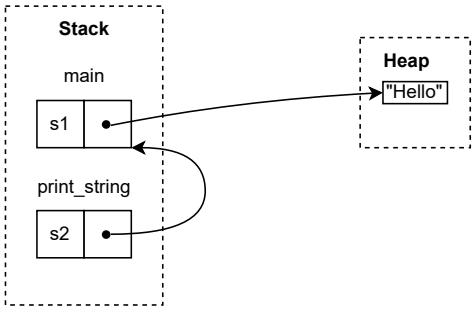


Figura 2: Visualizzazione memoria Rust durante la chiamata di `print_string()`.

In Java, non è possibile avere questo livello di indirezione, poiché i riferimenti puntano direttamente a oggetti in memoria. Infatti, il codice equivalente in Java sarebbe:

```
public class Main {
    public static void main(String[] args) {
        String s1 = "Hello";
        printString(s1);
    }
    public static void printString(String s2) {
        System.out.println(s2);
    }
}
```

In questo caso, `s1` e `s2` sono entrambi riferimenti all'oggetto `"Hello"` in memoria (vedi figura 3).

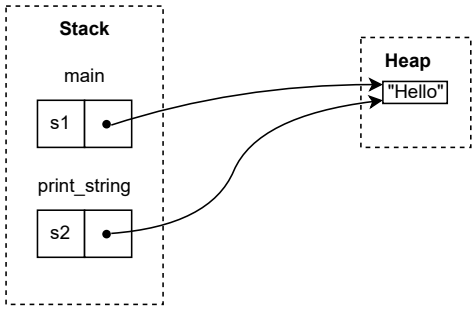


Figura 3: Visualizzazione memoria Java durante la chiamata di `printString()`.

Per eseguire l'operazione opposta, ossia ottenere il valore a cui punta un riferimento, si utilizza l'operatore di dereferenziazione $*$.

Abbiamo già visto come le variabili in Rust siano immutabili per definizione, ma è possibile dichiararle come mutabili utilizzando la keyword `mut`. I riferimenti in Rust hanno un comportamento simile: sono immutabili di default, ma possono essere dichiarati come mutabili utilizzando la keyword `mut`. Quindi, attraverso i riferimenti, è possibile accedere a un dato attraverso variabili diverse. Questo è utile per evitare la duplicazione di dati o per condividere dati tra diverse parti del programma. Tuttavia, quando si permette a più parti del codice di accedere allo stesso valore, è necessario garantire che non ci siano conflitti tra le operazioni di lettura e scrittura per evitare situazioni di errore come:

- Deallocazione di un dato mentre un'altra parte del codice lo sta ancora utilizzando.
- Modificazione di un dato mentre un'altra parte del codice lo sta leggendo o modificando.

In Java, i due casi sopra elencati sono permessi e la responsabilità di evitare conflitti ricade sul programmatore come già visto nell'esempio 5. Il problema è ancora più evidente in contesti concorrenti in cui più thread possono agire su una stessa variabile. Rust, invece, introduce un sistema di regole che impedisce questi conflitti a compile time:

- Se si ha un riferimento mutabile a un valore, allora quello sarà l'unica variabile che può accedere a quel valore. Questo impedisce che una parte di codice modifichi un valore mentre un'altra parte lo sta leggendo o modificando. Se non ci sono riferimenti mutabili, allora non ci sono restrizioni sul numero di riferimenti immutabili che possono esistere contemporaneamente.

- Lo scope dei riferimenti Rust inizia quando il riferimento viene creato e termina l'ultima volta che il riferimento viene utilizzato. In particolare, i riferimenti Rust non sono proprietari della variabile a cui si riferiscono, quindi quando escono dallo scope la variabile non viene deallocata.

Vediamo un esempio che mostra l'importanza di queste regole:

```
fn main() {
    let mut v = vec![1, 2, 3];
    let r1 = &v[1];
    v.push(4);
    // push prende in prestito v in modo mutabile
    // mentre r1 è un riferimento immutabile ancora attivo
    // Questo genera un errore di compilazione
    println!("Il secondo elemento e': {}", r1);
}
```

Un `vec` in Rust ha un comportamento simile a un `ArrayList` in Java, ossia è un array che cresce dinamicamente. Questo significa che quando si aggiunge un elemento, il `vec` potrebbe dover allocare nuova memoria e copiare i dati esistenti in essa. Quindi, il riferimento `r1` potrebbe non puntare più al secondo elemento del `vec` dopo l'operazione `push`. Se Rust permettesse questo codice, `r1` diventerebbe un dangling pointer, cioè un riferimento non più valido[6]. Tuttavia, il compilatore impedisce questa situazione, garantendo sicurezza a tempo di compilazione.

In Java il problema dei dangling pointer è quasi inesistente, poiché i riferimenti non possono essere invalidati in questo modo. L'unico modo di ottenere un dangling pointer in Java è quello di avere un riferimento a un oggetto non inizializzato o esplicitamente impostato a `null`. Il codice equivalente in Java sarebbe:

```
public class Main {
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<>();
        list.add(1);
        Integer r1 = list.get(0);
        list.add(2);
    }
}
```

6 In questo codice lo scope `r1` termina dopo la sua stampa. Se non ci fosse l'istruzione di stampa non si avrebbero errori di compilazione poiché lo scope di `r1` terminerebbe dopo la sua dichiarazione

Questo codice compila e non genera errori a run-time poiché in Java si ha un livello di indirezione in più: r1 in Rust è l'indirizzo di memoria del valore "2" nel `vec`, mentre in Java è l'indirizzo di memoria dell'oggetto `Integer` che contiene il valore "2". Quindi se l'`ArrayList` venisse ridimensionato e allocato in un area di memoria diversa, ciò che viene copiato sono i riferimenti agli oggetti, non gli oggetti stessi, quindi la variabile `r1` è ancora valida perché l'oggetto a cui si riferisce non ha cambiato posizione in memoria (vedi figura 4).

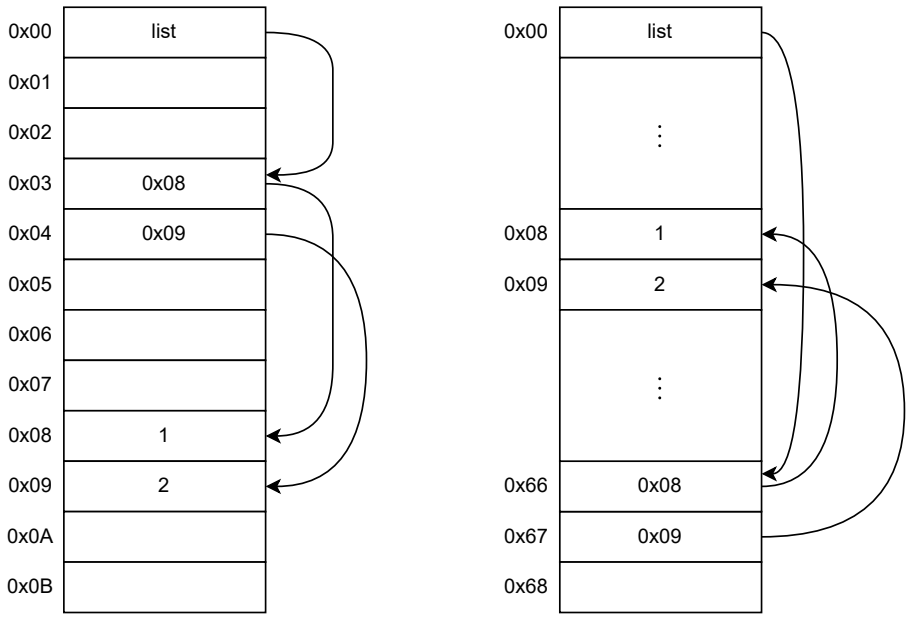


Figura 4: Memoria heap Java dopo il riallocamento dell' `ArrayList`.

3.6 L'ASSENZA DI VALORE IN RUST E JAVA

In Java, il valore `null` è un concetto fondamentale che rappresenta l'assenza di un valore o un riferimento non inizializzato. Ogni riferimento a un oggetto in Java può essere impostato a `null`, indicando che non punta a nessun oggetto valido. Questo approccio, sebbene flessibile e molto potente, introduce una serie di problemi:

- `NullPointerException`: l'accesso a un metodo o a un campo di un riferimento `null` genera un'eccezione a run-time, interrompendo l'esecuzione del programma.

- Ambiguità: non è chiaro se un riferimento `null` indica un errore, un valore mancante o, semplicemente, un'oggetto non ancora inizializzato. Inoltre, può non essere ovvio se un metodo può ritornare `null` o meno, richiedendo, quindi, documentazione aggiuntiva per chiarire il comportamento atteso.

- Errori a run-time: problemi legati a `null` vengono rilevati solo a run-time, non durante la compilazione.

Java cerca di mitigare questi problemi introducendo, a partire da Java 8, `Optional<T>`, un wrapper che può contenere un oggetto di tipo `T` o essere vuoto. Tramite `Optional<T>`, è possibile evitare il rischio di `NullPointerException` e rendere esplicito il fatto che un valore potrebbe non essere presente. È importante notare che, nonostante l'introduzione di `Optional<T>` e la sua crescente adozione, il problema di `null` non è completamente risolto:

- `Optional<T>` non è un sostituto diretto di `null`, ma piuttosto un modo per rappresentare l'assenza di un valore in modo più sicuro. Tuttavia, codice legacy e librerie esistenti continuano a utilizzare `null`, creando un sistema duale in cui entrambi i concetti coesistono.

- Il compilatore Java non impone l'utilizzo di `Optional<T>` al posto di `null`. Java mette a disposizione il costrutto ma non lo impone, lasciando la scelta al programmatore, che può essere più incline a utilizzare `null` per semplicità.

- Anche utilizzando `Optional<T>`, è possibile incorrere in errori se non si gestisce correttamente, o, ancora peggio, non si gestisce affatto il caso in cui l'oggetto è vuoto. Ad esempio, chiamare `get()` su un `Optional<T>` vuoto solleverà un'eccezione a run-time.

- Anche se un metodo Java richiede un parametro di tipo `Optional<T>`, si può sempre passare esplicitamente `null` invece di un'istanza di `Optional<T>`. Questo può portare a `NullPointerException`, vanificando i benefici di sicurezza offerti da `Optional`. Per ridurre questo rischio, esistono annotazioni sui parametri come `@NotNull` o `@Nullable`: un parametro marcato `@NotNull` segnala che non può essere null, mentre `@Nullable` indica che può esserlo. Alcuni IDE, come Eclipse, possono rilevare queste violazioni in fase di compilazione, aiutando a prevenire errori relativi a `null`.

Rust, invece, elimina del tutto l'uso `null`, prevenendo molti dei problemi sopra elencati. Tuttavia, il concetto rappresentato da `null` è sempre utile. Per questo motivo, Rust adotta un approccio simile a `Optional<T>` di Java tramite il tipo `Option<T>`, definito nella libreria standard come:

```
enum Option<T> {
    Some(T), // Contiene un valore di tipo T
    None,    // Non contiene alcun valore
}
```

L'idea alla base di `Option<T>` è analoga a quella di `Optional<T>`: fornire una rappresentazione sicura e non ambigua dell'assenza di un valore.

`Option<T>` è un `enum` dove `Some(T)` e `None` sono le sue due varianti. La differenza principale rispetto a Java è che, poiché in Rust una variabile non può mai assumere il valore `null`,per rappresentare l'assenza di valore il programmatore è costretto ad utilizzare `Option<T>`. Utilizzando la classe `Person`[7] del listato 5, consideriamo il seguente esempio per mostrare l'importanza di `Option<T>` in Rust:

```
public class PersonRepository{
    private List<Person> people = new ArrayList<>();

    public Person findPersonByName(String name) {
        Objects.requireNonNull(name,
                               "Il nome non può essere null");
        for (Person person : people) {
            if (person.getName().equals(name)) {
                return person; // Ritorna la persona trovata
            }
        }
        return null; // Ritorna null se non trovata
    }
}
```

Listato 7: Utilizzo di `null` in Java.

Il comportamento di `findPersonByName()` non risulta ovvio leggendo solo la firma del metodo e, quindi, va documentato adeguatamente. Inoltre, il chiamante deve ricordarsi di gestire il caso in cui il metodo ritorni `null`, altrimenti potrebbe incorrere in un `NullPointerException` a run-time.

7 Assumiamo che il campo `name` sia unico per ogni `Person`.

In Rust, invece, l'unico modo di avere un comportamento equivalente a quello del listato 7 è quello di utilizzare il tipo `Option<Person>`:

```rust
struct Person {
    name: String,
}

struct PersonRepository {
    people: Vec<Person>,
}

impl PersonRepository {
    pub fn find_person_by_name(&self, name: &str) -> Option<&Person> {
        for person in &self.people {
            if person.name == name {
                return Some(person);
            }
        }
        None
    }
}
```

Da questo esempio si capisce come l'utilizzo di `Option<T>` renda esplicito il fatto che il metodo può non trovare una persona con il nome specificato. Inoltre, non è necessario verificare se la stringa di input `name` è `null`, poiché una variabile Rust non può essere `null`. Se si prova a passare `None` al metodo, il compilatore segnalerà un errore.

Il codice del listato 7 può essere rifattorizzato per avere un comportamento analogo a quello di Rust, utilizzando `Optional<Person>`:

```java
public class PersonRepository {
    private List<Person> people = new ArrayList<>();

    public Optional<Person> findPersonByName(String name) {
        Objects.requireNonNull(name, "Il nome non può essere null");
        return people.stream()
                     .filter(person -> person.getName().equals(name))
                     .findFirst(); // Restituisce Optional<Person>
    }
}
```

Nonostante l'approccio di Rust risolva molti dei problemi legati a `null`, la responsabilità di utilizzare correttamente `Option<T>` ricade ancora sul programmatore. In Rust, come in Java, è necessario:

- Produrre un'alternativa valida quando il valore non è presente.
- Consumare il valore se è presente.
- Evitare l'accesso diretto e non controllato al valore (ad esempio `get()` in Java o, l'equivalente in Rust, `unwrap()`), poiché ciò sarebbe diverso da accedere a un valore `null`.

4

POLIMORFISMO

Il polimorfismo è un concetto fondamentale della programmazione. Derivato dal greco "molte forme", il polimorfismo consente a entità di assumere diverse forme o comportamenti in base al contesto, permettendo di scrivere codice più flessibile, estendibile e manutenibile. Grazie al polimorfismo, è possibile utilizzare un'interfaccia comune per manipolare elementi di tipi diversi, facilitando così l'implementazione di soluzioni generiche. Il vantaggio principale del polimorfismo è il miglioramento della qualità del codice:

- Favorisce l'astrazione, consentendo di trattare oggetti di tipi diversi in modo uniforme.
- Riduce la duplicazione di codice, poiché le operazioni comuni possono essere definite una sola volta e riutilizzate per diversi tipi.
- Semplifica la gestione delle estensioni future, poiché nuove funzionalità possono essere aggiunte senza modificare il codice esistente.

In un linguaggio come Java, orientato agli oggetti, il polimorfismo è una caratteristica centrale e largamente supportata tramite ereditarietà e interfacce. Rust, pur non essendo un linguaggio tradizionalmente orientato agli oggetti, offre un approccio alternativo al polimorfismo, basato su trait e tipi generici, che permette di ottenere astrazione e flessibilità.

In particolare, quando il codice coinvolge il polimorfismo, sono necessari meccanismi per determinare quale versione specifica di metodo sta venendo effettivamente eseguita. Questo processo prende il nome di *dispatch*. Esistono due forme di dispatch:

- Lo *Static Dispatch*, in cui la risoluzione della chiamata viene risolta a tempo di compilazione.
- Il *Dynamic Dispatch*, in cui la risoluzione della chiamata avviene a run-time.

In questo capitolo verrà fornita una panoramica delle modalità con cui Java e Rust forniscono supporto al polimorfismo, confrontando i due linguaggi.

4.1 POLIMORFISMO IN JAVA

Java supporta il polimorfismo attraverso due principali meccanismi: il *subtyping* (polimorfismo per inclusione) e il *parametric polymorphism* (polimorfismo parametrico).

Il subtyping si basa sul fatto che ci possa essere una relazione tra tipi chiamata *relazione di sottotipo*. Si dice che un tipo A è un sottotipo di un tipo B quando il contesto richiede un elemento di tipo B ma può accettare un elemento di tipo A. In Java, la relazione di sottotipo viene implementata attraverso l'ereditarietà e le interfacce. Le classi possono estendere altre classi e implementare interfacce, consentendo agli oggetti di essere trattati come istanze della loro classe base o interfaccia.

Il parametric polymorphism permette di assegnare a una parte di codice un tipo generico, utilizzando variabili di tipo al posto di tipi specifici, che poi possono essere istanziate con tipi concreti al momento dell'utilizzo. In particolare, Java supporta l'*F-bounded polymorphism* [4], che è la capacità di poter definire vincoli su un tipo generico che possono anche essere ricorsivi. Ad esempio `<T extends Comparable<T>>`.

4.2 POLIMORFISMO IN RUST

In Rust, il polimorfismo è implementato attraverso i concetti di *trait* e i *generics*. I trait sono simili alle interfacce in Java e definiscono un insieme di metodi che un tipo deve implementare per essere considerato conforme a quel trait. I generics consentono di scrivere funzioni e strutture dati che possono operare su tipi diversi senza dover specificare un tipo concreto. Tramite l'uso dei generics si ottiene il *parametric polymorphism*, mentre attraverso i trait si ottiene il *bounded parametric polymorphism*.[1]

4.3 GENERICS: MONOMORPHIZATION E TYPE ERASURE

Sia Rust che Java supportano la programmazione generica, che consente di scrivere codice che può operare su tipi diversi senza avere codice

1 Entrambe le forme di polimorfismo hanno la stessa definizione data in precedenza per Java.

duplicato per ogni tipo specifico. La sintassi per definire i generics in Rust e Java è simile: si utilizzano parentesi angolari per specificare i parametri di tipo. Tuttavia, ci sono differenze significative nella gestione dei generics tra i due linguaggi.

Il compilatore Java utilizza un processo chiamato *type erasure* per implementare i generics, questo include i seguenti fatti:

- Durante la compilazione i parametri di tipo vengono sostituiti con il tipo `Object` o con un tipo specifico se è stato definito un vincolo sul parametro di tipo.
- Vengono inseriti cast espliciti per mantenere la type safety.
- Generazione di *Bridge Methods* per preservare il polimorfismo dopo il processo di type erasure.

Ad esempio:

```
public class GenericClass<T> {
    T value;

    void setValue(T value) {
        this.value = value;
    }
}
```

Dopo la type erasure, il codice diventa:

```
public class GenericClass {
    Object value;

    void setValue(Object value) {
        this.value = value;
    }
}
```

Nel caso in cui, invece, si definisca un vincolo di tipo, ad esempio `<T extends Number>`, il compilatore Java sostituirà `T` con `Number` durante la type erasure, mantenendo la type safety:

```
public class GenericClass{
    Number value;

    void setValue(Number value) {
        this.value = value;
    }
}
```

Quando si combina la type erasure con l'overriding dei metodi, Java genera dei *Bridge Methods* per garantire che il polimorfismo funzioni correttamente. Uno scenario tipico è il seguente:

- Una classe generica o un'interfaccia ha un metodo che usa un tipo generico.
- Una sua sottoclasse sovrascrive quel metodo con un tipo concreto.
- Dopo la type erasure, le firme dei due metodi non corrispondono più. Questo romperebbe il polimorfismo.

Ad esempio:

```
class Parent<T> {
    T getValue() { return null; }
}

class Child extends Parent<String> {
    @Override
    String getValue() {
         return "Hello";
    }
}
```

Dopo la type erasure:

```
class Parent {
    Object getValue() {
        return null;
    }
}

class Child extends Parent {
    String getValue() {
        return "Hello";
    }

    // Bridge method generato dal compilatore:
    // Garantisce che la chiamata a Parent.getValue()
    // funzioni correttamente
    Object getValue() {
        return this.getValue(); // chiama String getValue()
    }
}
```

In Rust, invece, i generics sono implementati tramite un processo chiamato *monomorphization*. Questo è il processo tramite cui il compilatore Rust genera codice specifico per ogni tipo concreto utilizzato con i generics. Questo significa che Rust sostituisce i parametri di tipo generici con i tipi concreti utilizzati, e genera una versione specifica della funzione o della struttura per ogni tipo. Ad esempio:

```
struct Boxed<T> {
    value: T,
}

fn main() {
    let a = Boxed { value: 123 }; // T = i32
    let b = Boxed { value: "text" }; // T = &str
}
```

Dopo la monomorphization, il compilatore Rust genera due versioni della struttura `Boxed`:

```
struct Boxed_i32 {
    value: i32,
}

struct Boxed_str {
    value: &str,
}
```

È facile notare come i due approcci siano molto diversi e abbiano implicazioni diverse sul modo in cui il codice viene generato e sulla performance:

- Rispetto alla type erasure di Java, la monomorphization di Rust porta diversi vantaggi in termini di performance:
  - Nella type erasure, poiché i parametri di tipo vengono eliminati, il compilatore spesso deve inserire cast espliciti per garantire la type safety, il che può introdurre un overhead. Questo overhead è spesso trascurabile ma comunque presente.
  - I generics di Java non possono essere utilizzati con tipi primitivi, come `int` o `double`, ma solo con oggetti. Questo significa che quando si usano generics con tipi primitivi, Java deve utilizzare il boxing e l'unboxing, che introducono un ulteriore overhead.

  - Poiché Java usa lo stesso bytecode per tutte le istanziazioni di un generico, non può ottimizzare il codice per tipi specifici. Questo può essere rilevante per sistemi che richiedono un alto grado di performance.
  - Il monomorphization è un meccanismo statico che risolve i tipi al momento della compilazione, quindi una chiamata a un metodo generico è risolta a compile time e non ci sono overhead a runtime. Invece, la type erasure può coinvolgere il dynamic dispatch di Java che può essere più costoso in termini di performance.

- La monomorphization può portare ad un aumento della dimensione del codice del programma compilato poiché vengono generate diverse versioni della stessa funzione generica, una per ogni combinazione di tipi con cui viene chiamata.
- La monomorphization può incrementare notevolmente il tempo di compilazione del programma, specialmente se ci sono molte combinazioni di tipi concreti con cui viene chiamata una funzione generica.
- La monomorphization può fornire messaggi di errore più chiari e specifici poiché ogni versione specifica della funzione generica ha tipi concreti associati.

4.4 TRAITS

Un *trait* è un costrutto di Rust che consente di definire un insieme di funzionalità che un tipo deve implementare. I traits vengono utilizzati, quindi, per definire comportamenti comuni che possono essere condivisi tra diversi tipi. Questo significa che tutti i tipi che implementano un determinato trait condividono la stessa interfaccia di quel trait. Un trait viene dichiarato utilizzando la keyword `trait` come segue:

```
trait MyTrait {
    fn my_method(&self) -> ();
}
```

L'implementazione del trait avviene attraverso le keywords `impl` e `for` come segue:

```
struct MyStruct;
```

```rust
impl MyTrait for MyStruct {
    fn my_method(&self) -> () {
        println!("Hello from MyStruct");
    }
}
```

I traits possono essere utilizzati per realizzare il bounded parametric polymorphism in Rust, consentendo di specificare vincoli sui tipi generici (*trait bounds*). Ad esempio, si può definire una funzione che accetta un tipo generico che implementa un determinato trait nel seguente modo:

```rust
fn my_function<T: MyTrait>(item: T) {
    println!("{}", item.my_method());
}
```

Nel caso in cui siano presenti più vincoli, questi possono essere combinati utilizzando il simbolo + oppure attraverso l'uso di `where`:

```rust
fn my_function<T>(item: T)
where
    T: MyTrait + AnotherTrait
{
    println!("{}", item.my_method());
}
```

In Java, è possibile ottenere un comportamento simile ai trait bounds attraverso i vincoli di tipo (*type bounds*) nelle dichiarazioni generiche. Ad esempio, si può definire una classe generica che accetta un tipo che estende una classe base o implementa un'interfaccia:

```java
public class MyClass<T extends Bound> {
    /* ... */
}
```

In questo esempio, `T` è un tipo generico che deve implementare `Bound`. Questo consente di utilizzare i metodi definiti in `Bound` all'interno della classe `MyClass`. Anche in Java è possibile definire più vincoli di tipo[2] attraverso l'utilizzo dell'operatore &:

```java
public class MyClass<T extends Bound & AnotherBound> {
    /* ... */
}
```

2 A causa dell'ereditarietà singola di Java solo un vincolo può essere una classe: il primo nella lista.

Si può facilmente notare come il concetto di trait in Rust sia molto simile alle interfacce in Java. Entrambi servono a garantire che un valore o un oggetto possa essere utilizzato secondo un certo protocollo o insieme di regole, permettendo diverse implementazioni concrete senza essere vincolati a dettagli di implementazione, a differenza di quanto accade con una superclasse Java. Inoltre, in entrambi i costrutti è possibile definire metodi di default. Tuttavia, sono presenti alcune differenze significative:

- In Rust, un trait non è un tipo di dato, ma un insieme di metodi che un tipo può implementare. In Java, invece, un' interfaccia funge sia da contratto sia da tipo: quando una classe implementa un'interfaccia, è possibile trattare gli oggetti di quella classe come istanze del tipo dell'interfaccia stessa. La differenza principale è, quindi, che in Rust il trait è separato dal tipo concreto, mentre in Java l'interfaccia può essere usata direttamente come tipo del riferimento.

- In Java, le interfacce richiedono che la classe che le implementa abbia metodi con nomi specifici. Questo può creare conflitti: ad esempio, due interfacce potrebbero essere impossibili da implementare contemporaneamente se hanno metodi con lo stesso nome ma tipi di ritorno diversi. Ad esempio:

```java
public interface InterfaceA {
    String getValue();
}

public interface InterfaceB {
    int getValue();
}

public class MyClass implements InterfaceA, InterfaceB {
    // Errore: il compilatore non sa quale
    // metodo getValue() implementare
}
```

  In Rust, invece, ogni trait ha il proprio namespace separato. Quando si implementa un trait per un tipo, lo si fa in un blocco `impl` separato specificando il trait. In questo modo è sempre esplicito a quale trait appartiene ogni metodo, evitando conflitti tra trait diversi. Nel caso in cui entrambi i trait sono nello scope è necessario specificare il trait usando la sintassi `Trait::method(&obj)` invece della semplice chiamata con la notazione puntata.

- In Java, le interfacce possono avere parametri di tipo. Tuttavia, un oggetto può implementare un'interfaccia generica solo una volta:

```java
public interface anInterface<T> {
    void doSomething(T value);
}

public class MyClass implements anInterface<String>,
                                anInterface<Integer>
{
    @Override
    public void doSomething(String value) { /* ... */ }
    //Errore: il compilatore non sa quale metodo
    // doSomething() implementare
}
```

  In Rust, invece, un trait generico può essere implementato per molti tipi diversi, e ciascuna implementazione è considerata sostanzialmente un trait distinto:

```rust
trait MyTrait<T> {
    fn do_something(&self, value: T);
}

struct MyStruct;

impl MyTrait<String> for MyStruct {
    fn do_something(&self, value: String) { /* ... */ }
}

impl MyTrait<i32> for MyStruct {
    fn do_something(&self, value: i32) { /* ... */ }
}
```

- In Rust, è possibile implementare traits per tipi esterni, definiti in altri crate [3]. Questo consente di estendere il comportamento di tipi che non sono stati definiti nel proprio codice. Questo può essere fatto rimanendo conforme alla *Orphan Rule*: si può implementare un trait per un tipo solo se almeno uno dei due (trait o tipo) è definito nel proprio crate. Ad esempio, consideriamo il caso in cui si vuole estendere il tipo `String` in Rust. Non possiamo modificare direttamente `String` perché definito nella standard library, ma possiamo implementarci un trait come:

3 Un crate è un pacchetto di codice Rust. Simile a un modulo in altri linguaggi.

```rust
trait Shout {
    fn shout(&self) -> String;
}

// Implementiamo il trait per String (tipo esterno)
impl Shout for String {
    fn shout(&self) -> String {
        self.to_uppercase() + "!"
    }
}
```

Ora, posso chiamare direttamente il metodo `shout()` su una qualsiasi istanza di `String` :

```rust
fn main() {
    let s = String::from("ciao");
    println!("{}", s.shout());
}
```

In Java, invece, l'estensione di comportamento è basata su inheritance e method overriding. Questo funziona solo se si vuole estendere una classe esistente (non `final`). Invece, se si vuole implementare un'interfaccia su una classe esistente è necessario avere accesso al codice sorgente della classe per modificarla. Se questo non è possibile si deve ricorrere a metodi alternativi come l'utilizzo del pattern strutturale *Adapter* [3] che permette di ottenere lo stesso comportamento di Rust ma con molto più codice. Ad esempio:

```java
// Interfaccia che vogliamo usare
// nella nostra applicazione
interface Sensor {
    int getData();
}

// Classe esistente (non modificabile)
class LegacySensor {
    public int readValue() { /* ... */ }
}

// Adapter che incapsula LegacySensor
// e lo adatta all'interfaccia Sensor
class SensorAdapter implements Sensor {
    private LegacySensor legacy;

    public SensorAdapter(LegacySensor legacy) {
```

```
        this.legacy = legacy;
    }

    @Override
    public int getData() {
        return legacy.readValue();
    }
}
```

A differenza di Rust, in Java si deve definire un nuovo tipo che incapsula il tipo esistente e implementa l'interfaccia desiderata.

4.4.1 *Supertraits*

Per quanto riguarda i trait bounds, in Rust è possibile definire vincoli su trait: nella definizione di un trait è possibile specificare che esso estende un altro trait. Questo implica che qualsiasi tipo che implementa il trait figlio deve anche implementare il trait genitore. Ad esempio:

```
trait A {
    fn method_from_A(&self);
}

trait B: A {
    fn method_from_B(&self);
}
```

In questo esempio, il trait `B` estende il trait `A`, quindi qualsiasi tipo che implementa `B` deve anche implementare `A`. Si dice che `B` è *Supertrait* di `A`. In particolare, se sto lavorando un con tipo generico su cui ho un vincolo di tipo `B`, posso usare i metodi definiti in `A` senza dover specificare esplicitamente il vincolo su `A`:

```
fn my_function<T: B>(item: T) {
    item.method_from_A(); // Posso usare i metodi di A
}
```

Quindi, tramite supertraits, si possono creare gerarchie di trait, facilitando la definizione di comportamenti complessi. Questo comportamento può ricordare la costruzione della gerarchia dei tipi che si ottiene in Java tramite ereditarietà. Tuttavia, c'è una differenza sostanziale nel significato di estensione tra Java e Rust:

- In Rust, l'estensione di un Trait non implica l'ereditarietà di implementazioni concrete di metodi, ma solo l'ereditarietà di comportamenti. Scrivere `trait B: A` significa che se un tipo `T` implementa `B` allora deve implementare anche `A`. Infatti, se un tipo implementa `B`, deve fornire implementazioni per tutti i metodi di `A` e `B`.

- In Java, l'estensione di una classe implica l'ereditarietà di implementazioni concrete di metodi. Per ottenere un comportamento analogo ai supertrait, si utilizza l'estensione di interfacce tramite la keyword `extends`:

```java
interface A {
    void methodFromA();
}

interface B extends A {
    void methodFromB();
}

// MyClass deve implementare entrambi i metodi
class MyClass implements B {
    @Override
    public void methodFromA() { /* ... */ }

    @Override
    public void methodFromB() { /* ... */ }
}
// Un oggetto di tipo B può utilizzare anche methodFromA()
```

4.5 RISOLUZIONE DEI METODI: MECCANISMI STATICI E DINAMICI

Un aspetto fondamentale del polimorfismo è la risoluzione dei metodi, ossia il processo tramite cui il sistema determina quale versione specifica di un metodo deve essere eseguita al momento dell'invocazione. Questo processo può avvenire in due modi: staticamente (a compile-time) o dinamicamente (a run-time). Rust e Java utilizzano entrambi questi meccanismi, ma in modi diversi a causa delle loro differenze di progettazione e filosofia. In questa sezione viene trattato il funzionamento di una chiamata di metodo in entrambi i linguaggi.

Rust ha come filosofia principale la sicurezza e la performance, quest' ultima viene raggiunta attraverso l'eliminazione, per quanto possibile, di qualsiasi overhead a runtime. Questo concetto è riassunto nel credo

di Rust *zero-cost abstractions*: le astrazioni offerte dal linguaggio (come i generics) non comportano alcun costo a runtime, ma solo a compile time. Per questo motivo, Rust predilige il dispatch statico rispetto a quello dinamico.

Java, invece, adotta un approccio più orientato all'object-oriented tradizionale, dove, per supportare l'ereditarietà e il polimorfismo, si predilige il dynamic dispatch per determinare quale implementazione di metodo utilizzare per una specifica firma.

4.5.1 *Rust*

Il caso più semplice da considerare in Rust è quello in cui si chiama una funzione classica:

```rust
fn plus_one(x: i32) -> i32 {
    x + 1
}

fn main() {
    let result = plus_one(5);
    println!("The result is: {}", result);
}
```

Listato 8: Esempio di chiamata a funzione in Rust.

Il compilatore Rust è capace di determinare esattamente quale funzione chiamare, poiché non vi è alcuna ambiguità: esiste una sola funzione con quel nome e quella firma. In questo caso, la risoluzione del metodo avviene in modo statico, senza alcun overhead a runtime.

Il caso più interessante avviene quando viene eseguita una chiamata di metodo. Questa può riferirsi sia a un metodo intrinseco, ossia definito direttamente su una `struct` o un `enum`, sia a un metodo di trait. In entrambi i casi, il compilatore determina quale funzione chiamare eseguendo:

- Dispatch statico se il tipo esatto su cui il metodo è chiamato è conosciuto a compile-time.
- Dispatch dinamico se il tipo su cui il metodo è chiamato è un *trait object* indiretto (vedi sezione 4.5.1).

Quando vengono usati i generics e i trait bounds, il compilatore userà lo static dispatch per risolvere le chiamate ai metodi. In particolare, il

compilatore genera una versione specializzata della funzione per ogni tipo concreto utilizzato, e quindi la risoluzione avviene a compile time senza la necessità di eseguire alcuna scelta a runtime. Questo processo è noto come *monomorphization* (descritto nel dettaglio nella sezione 4.3). Per mostrare esplicitamente questo comportamento consideriamo il seguente codice [4]:

```
trait Drivable {
    fn drive(&self);
}

struct Car;
impl Drivable for Car {
    #[inline(never)]
    fn drive(&self) {
        println!("You are driving a car.");
    }
}

struct Motorcycle;
impl Drivable for Motorcycle {
    #[inline(never)]
    fn drive(&self) {
        println!("You are driving a motorcycle.");
    }
}

struct Boat;
impl Drivable for Boat {
    #[inline(never)]
    fn drive(&self) {
        println!("You are driving a boat");
    }
}
```

Definiamo la seguente funzione generica con vincolo di tipo:

```
fn start_driving<T: Drivable>(t: T) {
    t.drive();
}
```

4 L'attributo `#[inline(never)]` viene utilizzato per evitare che il compilatore effettui l'inlining delle funzioni durante la compilazione, in modo da poter osservare chiaramente le chiamate di funzione nel codice compilato.

Ora, all'interno del nostro `main()` chiamiamo `start_driving()` con uno dei tre tipi concreti precedentemente definiti:

```
fn main() {
    start_driving(Car{});
    start_driving(Motorcycle{});
    start_driving(Boat{});
}
```

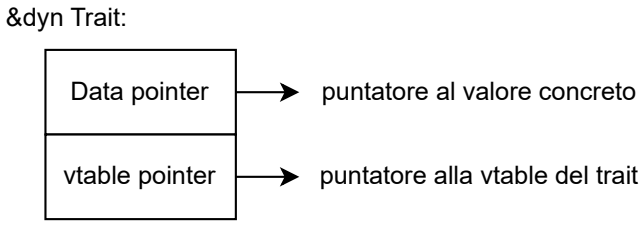
Durante la compilazione, tramite monomorphization, verranno create tre copie della funzione `start_driving`: una per ogni tipo con cui è stata chiamata. Di conseguenza, nel codice compilato, ogni chiamata a `start_driving` viene tradotta in un indirizzo di memoria diverso, corrispondente alla specifica implementazione specializzata per quel tipo. Questo evidenzia come lo static dispatch risolva la chiamata alla funzione al momento della compilazione, senza alcun overhead a runtime.

Trait Objects e Dynamic Dispatch

Un trait object è definito come un valore opaco [5] di un altro tipo che implementa un insieme di traits. Un trait object è dichiarato utilizzando la keyword `dyn` seguita da un trait, ad esempio `dyn MyTrait`. Essendo opaco, la dimensione di un trait object non è conosciuta staticamente e quindi, in gergo Rust, sono *Dynamically Sized Types* (DST). Per questo motivo, i trait object devono essere sempre utilizzati dietro a un tipo puntatore, come `&dyn MyTrait` o `Box<dyn MyTrait>`. Internamente, un trait object è rappresentato come una coppia di puntatori:

- Un puntatore a un istanza di un tipo `T` che implementa `MyTrait`.
- Un puntatore alla *vtable* corrispondente alla combinazione tra il tipo concreto e il trait implementato. Una vtable è un array di puntatori a funzioni, dove ogni puntatore corrisponde a un metodo definito nel trait.

5 Per opaco si intende che il compilatore non sa esattamente quale sia il tipo concreto, ma che implementa certe caratteristiche (traits).



Per mostrare la necessità di utilizzare i trait objects, consideriamo l'implementazione del design pattern *Observer* [3]: il `Subject` deve poter mantenere la collezione degli `Observer`, i quali vanno trattati in maniera uniforme nonostante possano essere di tipi diversi. Per fare ciò, definiamo il trait `Observer` e le sue implementazioni:

```
trait Observer {
    fn update(&self, data: &str);
}

struct ConcreteObserverA;

impl Observer for ConcreteObserverA {
    fn update(&self, data: &str) {
        println!("ConcreteObserverA received: {}", data);
    }
}

struct ConcreteObserverB;

impl Observer for ConcreteObserverB {
    fn update(&self, data: &str) {
        println!("ConcreteObserverB received: {}", data);
    }
}
```

Ora, definiamo il `Subject` che mantiene una lista di `Observer` come trait objects:

```
struct Subject {
    observers: Vec<Box<dyn Observer>>,
}

impl Subject {
    fn new() -> Self {
```

```
            Subject {
                observers: Vec::new(),
            }
        }

        fn add_observer(&mut self, observer: Box<dyn Observer>) {
            self.observers.push(observer);
        }

        fn notify_observers(&self, data: &str) {
            for observer in &self.observers {
                observer.update(data);
            }
        }
    }
```

Quando si usa un trait object, Rust deve utilizzare il dynamic dispatch: il compilatore non conosce tutti i tipi che possono essere utilizzati nel codice che sta utilizzando il trait object, quindi non sa quale metodo implementato su quale tipo chiamare. La risoluzione della chiamata avviene a runtime utilizzando la vtable associata al trait object che avviene con complessità costante.

4.5.2 *Java*

In Java, il meccanismo di risoluzione dei metodi è strettamente legato al concetto di ereditarietà e polimorfismo. Al fine della trattazione, definiamo un metodo virtuale come un metodo di istanza che può essere sovrascritto. Quindi metodi dichiarati come `static`, `final` o `private` non sono considerati virtuali. Java segue la seguente semantica di invocazione di metodi:

- Se il metodo non è virtuale, la risoluzione avviene completamente a compile-time.
- Se il metodo è virtuale, la scelta dell'implementazione del metodo avviene a runtime. Questo avviene attraverso una vtable in maniera simile a Rust. Anche in questo caso si ottiene la selezione del metodo con complessità costante.

In Java, tutte le chiamate a metodi non virtuali sono risolte a compile-time in maniera simile a come Rust risolve le chiamate di funzione. Questo

significa che il compilatore sa esattamente quale metodo chiamare e non è necessario alcun meccanismo di dispatch a runtime. In particolare, per ottenere il comportamento di una funzione Rust si utilizza la keyword `static` che permette di definire metodi che appartengono alla classe stessa piuttosto che a una specifica istanza della classe. Ad esempio, considerando il listato 8, l'equivalente in Java sarebbe:

```
public class Main {
    public static int plusOne(int x) {
        return x + 1;
    }

    public static void main(String[] args) {
        int result = plusOne(5);
        System.out.println("The result is: " + result);
    }
}
```

Il caso più interessante è quello delle chiamate a metodi virtuali. Un metodo virtuale può essere sovrascritto in una sottoclasse per fornire una nuova implementazione, si parla di *overriding*. In particolare, affinché avvenga l'overriding, si devono realizzare due condizioni:

- Il metodo della sottoclasse deve avere lo stesso nome del metodo della superclasse.
- Il metodo della sottoclasse deve avere gli stessi parametri, cioè lo stesso numero, tipo e ordine degli argomenti; il tipo di ritorno deve essere lo stesso o un sottotipo (covarianza del tipo di ritorno).

In generale, il compilatore Java vede solo il riferimento statico di una variabile, ma non vede il tipo effettivo dell'oggetto a cui punta: questa informazione diventa disponibile solo durante l'esecuzione. Quando un metodo virtuale viene chiamato, l'implementazione del metodo da utilizzare è determinata a runtime in base al tipo effettivo su cui il metodo viene chiamato e sulla firma del metodo. Il *Dynamic Method Lookup* è il processo tramite cui la JVM determina quale implementazione del metodo chiamare per una determinata firma. Ad esempio:

```
class Vehicle {
    public void start() {
        System.out.println("Vehicle is starting");
    }
```

```
}
class Car extends Vehicle {
    @Override
    public void start() {
        System.out.println("Car is starting");
    }
}

class Motorcycle extends Vehicle {
    @Override
    public void start() {
        System.out.println("Motorcycle is starting");
    }
}

public static void main(String[] args) {
    Vehicle[] vehicles = {new Car(), new Motorcycle()};
    for (Vehicle v : vehicles) {
        v.start(); // Chiamata a metodo virtuale
    }
}
```

L'invocazione di `start()` è risolta a runtime tramite dynamic method lookup, in base al tipo effettivo dell'oggetto a cui il riferimento `v` punta. Quindi l'esecuzione del metodo `main` produce il seguente output:

```
Car is starting
Motorcycle is starting
```

In particolare, se un metodo virtuale non viene sovrascritto in una sottoclasse, la chiamata al metodo utilizza l'implementazione della superclasse. Questo viene ripetuto ricorsivamente lungo la gerarchia delle classi fino a trovare un'implementazione del metodo.

Java offre anche un meccanismo di polimorfismo statico tramite il concetto di *Method Overloading*, ossia la definizione di più metodi con lo stesso nome ma firme diverse (diverso numero o tipo di parametri) all'interno di una classe. La risoluzione della chiamata a un metodo in overload avviene a compile-time in base al tipo statico sia dell'oggetto su cui viene chiamato, che degli argomenti con cui il metodo viene chiamato. Ad esempio:

```
public class Calculator {
    public int add(int a, int b) {
```

```
            return a + b;
        }

        public double add(double a, double b, double c) {
            return a + b + c;
        }
    }

    public static void main(String[] args) {
        Calculator calc = new Calculator();

        // Chiama add(int, int)
        System.out.println(calc.add(2, 3));

        // Chiama add(double, double, double)
        System.out.println(calc.add(1.5, 2.5, 3.5));
    }
```

Rust non supporta l'overloading dei metodi nel senso di Java, però è possibile avere metodi con lo stesso nome, volendo anche la stessa firma, per uno stesso tipo tramite l'uso di blocchi `impl` nel seguente modo:

```
struct Calculator;

trait Adder {
    fn add(&self, a: i32, b: i32) -> i32;
}

impl Calculator {
    fn add(&self, a: i32, b: i32) -> i32 {
        a + b
    }
}

impl Adder for Calculator {
    fn add(&self, a: i32, b: i32) -> i32 {
        a + b
    }
}
```

Se si crea un'istanza di `Calculator` e si chiama il metodo `add`:

```
fn main() {
    let calc = Calculator;
```

```
    let result = calc.add(2, 3);
    println!("Result: {}", result); // Chiama Calculator::add
}
```

Il metodo chiamato è quello definito direttamente su `Calculator`, poiché il compilatore dà priorità ai metodi innati della struct, ossia quelli definiti nel blocco `impl` della struct stessa, rispetto ai metodi forniti da un trait implementato dalla struct. In generale, l'algoritmo di risoluzione è molto più complesso e possono verificarsi casi di ambiguità più sofisticati rispetto all'esempio appena visto, in cui il compilatore non riesce a determinare quale metodo chiamare. In questi casi, si deve sempre ricorrere alla fully qualified syntax [6] per risolvere l'ambiguità. Anche in Java si possono verificare ambiguità nel caso di overloading, ad esempio:

```
public interface A {}

public interface B {}

public class C implements A, B {}

public class D {

    public void method(A a) { /* ... */ }
    public void method(B b) { /* ... */ }

    public static void main(String[] args) {
        // Errore di compilazione per ambiguità
        new D().method(new C());
    }

}
```

In questo caso, la chiamata a `method(new C())` genera un errore di compilazione per ambiguità, poiché `C` implementa sia `A` che `B`, e quindi non è chiaro quale versione di `method` debba essere chiamata. Per risolvere l'ambiguità, si può utilizzare un cast esplicito oppure modificare il codice per evitare la situazione ambigua, ad esempio definendo un nuovo metodo `method(C c)`.

6 La sintassi per la fully qualified syntax è `<Type as Trait>::method(&received, args...)`

4.6 ESTENSIONE DEL COMPORTAMENTO IN JAVA E RUST

Considerando quanto detto nelle sezioni precedenti, in particolare le sezioni 2.2 e 4.4, si può notare una differenza sostanziale tra Java e Rust: in Java, una classe definisce tre diversi elementi:

- Lo stato degli oggetti attraverso i campi.
- Il comportamento attraverso i metodi.
- Il tipo degli oggetti attraverso la gerarchia di classi e interfacce.

questo è molto diverso da quello che accade in Rust in cui questi tre elementi sono rappresentati nel seguente modo:

- Lo stato e il tipo degli oggetti è definito dalle `struct`.
- Il comportamento è definito all'interno di un blocco `impl` che può essere associato a una `struct` o a un `trait`.

Quindi, in Java, stato e comportamento sono definiti insieme in una singola entità, la classe, mentre in Rust c'è una separazione di queste responsabilità. Questa differenza ha implicazioni significative su come si estende il comportamento di tipi esistenti in ciascun linguaggio. In particolare, poiché Rust non supporta l'ereditarietà tra tipi, l'estensione di comportamento deve essere ottenuta tramite composizione e implementazione di traits. Ad esempio, consideriamo il seguente codice Java:

```
class Employee {
    private double salary;

    public double getSalary() {
        return salary;
    }
}
```

Supponiamo di voler estendere il comportamento della classe `Employee` per aggiungere la capacità di calcolare un bonus. In Java, questo può essere fatto creando una sottoclasse:

```
class Manager extends Employee {
    private double bonusPercentage;

    @Override
    public double getSalary() {
```

```
            return getSalary() * bonusPercentage;
        }
    }
```

Si può notare che tramite ereditarietà e overriding, la sottoclasse `Manager` ha ereditato sia lo stato (campo `salary`) che il comportamento (metodo `getSalary()`) dalla superclasse `Employee`, e ha sovrascritto il metodo `getSalary()` per fornire un nuovo comportamento specifico per i manager. In Rust, per estendere il comportamento di `Employee`, si può utilizzare la composizione:

```
struct Employee {
    salary: f64,
}

impl Employee {
    fn get_salary(&self) -> f64 {
        self.salary
    }
}
```

Per aggiungere la capacità di calcolare un bonus, si può definire una nuova struct `Manager` che contiene un'istanza di `Employee`:

```
struct Manager {
    employee: Employee,
    bonus_percentage: f64,
}

impl Manager {
    fn get_salary(&self) -> f64 {
        self.employee.get_salary() * self.bonus_percentage
    }
}
```

Già da questo esempio si può notare come in Rust il comportamento viene esteso tramite composizione: la struct `Manager` contiene un'istanza di `Employee`.

Ora, consideriamo il caso in cui si voglia trattare `Manager` e `Employee` in maniera uniforme. In Java, questo è possibile grazie al fatto che `Manager` è un sottotipo di `Employee`, quindi si può usare il polimorfismo basato su sottotipi. Ad esempio, consideriamo la seguente classe Java:

```
class Payroll {
    public void paySalary(Employee emp) {
        double amount = emp.getSalary();
        System.out.println("Processing payment of: " + amount);
    }
}
```

In questo caso, tramite il polimorfismo dinamico di Java, il metodo `processPayment()` può accettare sia istanze di `Employee` che di `Manager`, e chiamare il metodo corretto `getSalary()` in base al tipo concreto dell'oggetto passato. In Rust, il codice precedente non permette questa flessibilità, poiché `Manager` ed `Employee` sono completamente scorrelati a livello di tipi. Per ottenere un comportamento simile in Rust, si può utilizzare un trait per definire un'interfaccia comune:

```
trait Payable {
    fn get_salary(&self) -> f64;
}

impl Payable for Employee {
    fn get_salary(&self) -> f64 {
        self.salary
    }
}

impl Payable for Manager {
    fn get_salary(&self) -> f64 {
        self.employee.get_salary() * self.bonus_percentage
    }
}
```

Adesso `Employee` e `Manager` implementano entrambi il trait `Payable`, quindi si può definire una funzione che accetta un parametro generico con vincolo di tipo `Payable`:

```
struct Payroll;

impl Payroll {
    fn pay_salary<T: Payable>(&self, emp: &T) {
        let amount = emp.get_salary();
        println!("Processing payment of: {}", amount);
    }
}
```

In questo modo, la funzione `pay_salary()` può accettare sia istanze di `Employee` che di `Manager`, e chiamare il metodo corretto `get_salary()` in base al tipo effettivo dell'oggetto passato. Tuttavia, a differenza di Java, questa flessibilità viene ottenuta tramite l'uso di generics e trait, piuttosto che tramite relazioni di sottotipo. Inoltre, a differenza di Java, il meccanismo utilizzato da Rust è completamente statico, come visto nella sezione 4.3.

Estendiamo ulteriormente l'esempio aggiungendo un nuovo metodo nella classe `Payroll`:

```
class Payroll {
    public void paySalary(Employee emp) { /* ... */ }

    public void payEmployees(List<Employee> employees) {
        for (Employee emp : employees) {
            paySalary(emp);
        }
    }
}
```

Ancora una volta, grazie al subtyping di Java, `payEmployees()` funziona correttamente. In Rust, limitarsi a usare generics non è sufficiente per ottenere lo stesso comportamento. Questo perché i generics vengono monomorfizzati, rendendo impossibile la creazione di una collezione di tipi eterogenei. Per ottenere questo comportamento in Rust, si deve utilizzare un trait object:

```
impl Payroll {
    fn pay_salary<T: Payable>(&self, emp: &T) { /* ... */ }

    fn pay_employees(&self, employees: &Vec<Box<dyn Payable>>) {
        for emp in employees {
            self.pay_salary(emp.as_ref());
        }
    }
}
```

In questo caso, tramite trait objects e dynamic dispatch, la funzione `pay_employees()` può accettare una collezione di oggetti eterogenei che implementano il trait `Payable`, e chiamare il metodo `get_salary()` corretto in base al tipo effettivo di ogni oggetto nella collezione.

Da questo esempio si notare differenze sostanziali tra come i due linguaggi supportano l'estensione del comportamento. Non esiste un

approccio migliore ma dipende dal contesto specifico e dai requisiti del progetto. In generale, si possono fare le seguenti osservazioni:

- Java offre un meccanismo di estensione del comportamento più diretto e intuitivo tramite l'ereditarietà e il polimorfismo basato su sottotipi. Questo rende più semplice la creazione di gerarchie di classi e l'estensione del comportamento esistente. Invece, Rust, richiede molto più lavoro per ottenere lo stesso risultato. Rust, sacrifica un po' di semplicità per ottenere una maggiore sicurezza e performance.

- Rust lascia allo sviluppatore più libertà di scelta riguardo l'utilizzo di meccanismi statici o dinamici. Questo anche perché Rust utilizza solamente static dispatch a meno che non sia dichiarato differentemente dallo sviluppatore tramite i trait objects. Java, invece, se utilizzato secondo i principi della programmazione orientata agli oggetti, utilizza principalmente il dynamic dispatch.

- La scelta della composizione è quasi obbligata in Rust, questo può essere visto come una cosa positiva perché l'estensione del comportamento tramite ereditarietà porta con sé diversi problemi quali:
  - Il *Fragile Base Class Problem*: quando una superclasse viene modificata, le sottoclassi che dipendono da essa possono rompersi in modi inaspettati.
  - Quando estendo una classe eredito tutti i suoi metodi (`public` e `protected`), anche quelli a cui non sono interessato, e non posso fare altrimenti. Questo può causare comportamenti indesiderati. Per risolvere questo problema si può nuovamente utilizzare il pattern *Adapter* per adattare il comportamento della superclasse.
  - La possibilità di estendere in maniera incontrollata le classi ha portato Java ad introdurre le classi (e interfacce) `Sealed` tramite le quali è possibile mettere dei vincoli su quali classi possono estendere una determinata superclasse.

  In generale, è buona pratica favorire la composizione rispetto all'ereditarietà.

BIBLIOGRAPHY

[1] Baeldung. Memory Leaks in Java, 2018.

[2] Dynatrace. Reducing Garbage-Collection Pause Time. Dynatrace eBook.

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[4] Ben Greenman, William J. Bowman, and Matthias Felleisen. Getting F-Bounded Polymorphism into Shape. In *PLDI '14: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 89–99. Association for Computing Machinery, 09-06-2014.

[5] Tony Hoare. Null References: The Billion Dollar Mistake, August 2009. InfoQ Presentation.

[6] Cay S. Horstmann. *Java for Impatient Programmers*. Addison-Wesley, 2nd edition, 2018.

[7] Jeff Vander Stoep. Memory Safe Languages in Android 13, December 2022. Google Security Blog.