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

Tesi di Laurea

# ANALISI COMPARATIVA DI RUST E JAVA

NICOLA PAPINI

Relatore/Relatrice: Lorenzo Bettini

Anno Accademico 2024-2025

Nicola Papini *Analisi comparativa di Rust e Java,* Corso di Laurea in Informatica, © Anno Accademico 2024-2025

# CONTENTS

# LIST OF FIGURES

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

# 1

# INTRODUCTION

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

# 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.

## 2.1 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 verifichrebbe 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
    }
}
```

## 2.2 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 [5], 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.

## 2.3 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.

## 2.4 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);
```

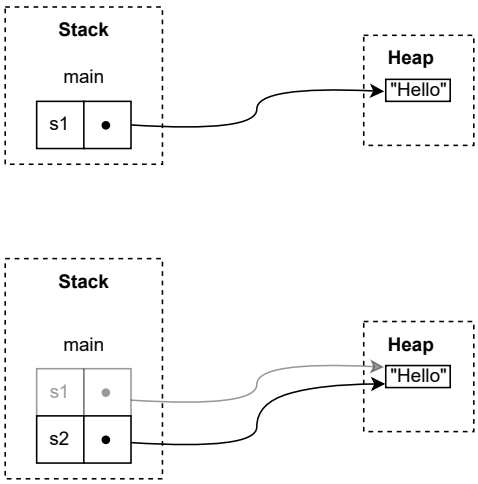


Figura 1: Visualizzazione memoria Rust dopo il trasferimento di ownership.

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 1, 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 1: 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 [4]

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)[5], Rust utilizza la semantica del *copy*, poiché la copia bit-a-bit è efficiente e non introduce rischi di inconsistenza o accessi errati.

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

5 In Rust le variabili sono immutabili a meno che non si vengano dichiarate con la keyword `mut`.

L'ownership può essere trasferita anche con le funzioni che ritornano un valore. In questo caso, un assegnamento a una variabile di un valore 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 2: 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[6] (vedi listato 2).

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

## 2.5 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 2. 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" [3], 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:

```rust
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.

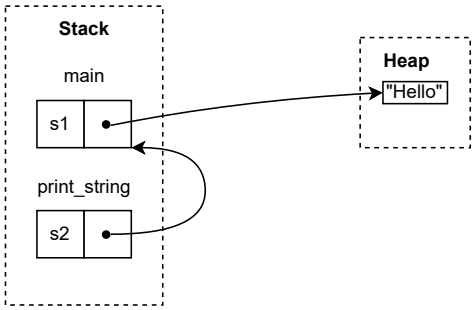


Figura 2: Visualizzazione memoria Rust durante la chiamata di `print_string()`.

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).

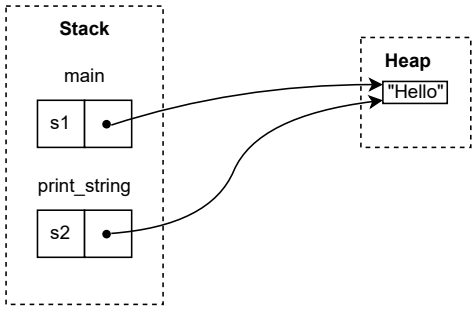


Figura 3: Visualizzazione memoria Java durante la chiamata di `printString()`.

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 1. 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[7]. 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);
    }
}
```

7 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).

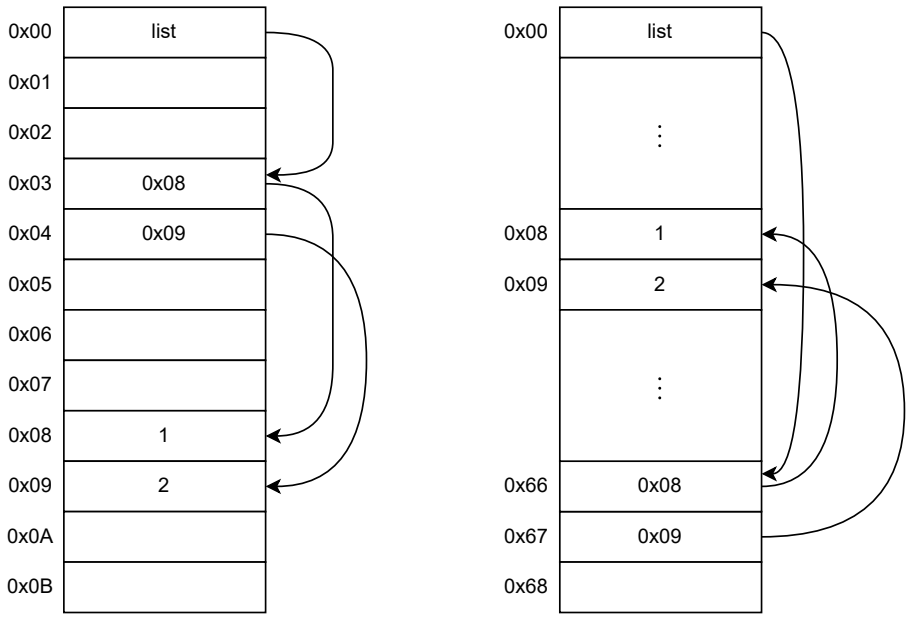


Figura 4: Memoria heap Java dopo il riallocamento dell'`ArrayList`.

## 2.6 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.

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)`[8] 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`[9] del listato 1, 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 3: 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.

In Rust, invece, l'unico modo di avere un comportamento equivalente a quello del listato 3 è quello di utilizzare il tipo `Option<Person>`:

```
struct Person {
    name: String,
}

struct PersonRepository {
    people: Vec<Person>,
}
```

8 In Rust una variante di un `enum` può contenere uno o più valori, ognuno con il suo tipo. In questo caso `Some(T)` contiene un valore di tipo `T`.

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

```rust
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[10] 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 3 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`.

10 `impl` è un costrutto Rust che consente di implementare metodi per una `struct`.

# 3

# 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 questo capitolo verrà fornita una panoramica delle modalità con cui Java e Rust implementano e sfruttano il polimorfismo, confrontando i due linguaggi.

## 3.1 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.

## 3.2 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*, come in Java, mentre attraverso i trait si ottiene il *bounded parametric polymorphism*, che consente di specificare vincoli sui tipi generici.

## 3.3 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 dover duplicare il codice per ogni tipo specifico. La sintassi per definire i generics in Rust e Java è simile, utilizzando parentesi angolare 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 il 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,
}
```

```rust
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`:

```rust
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 non ha 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.

## 3.4 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:

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

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

```
struct MyStruct;

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 l'interfaccia `MyInterface`. Questo consente di utilizzare i metodi definiti nell'interfaccia `MyInterface` all'interno della classe `MyClass`. Anche in Java è possibile definire più vincoli di tipo attraverso l'utilizzo dell'operatore &:

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

Si può facilmente notare come il concetto di trait in Rust sia molto simile alle interfacce in Java. Entrambi servono entrambi 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. Tuttavia, sono presenti alcune differenze significative:

- In Rust, un trait non è un tipo concreto, 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 può essere implementato per molti tipi diversi, e ciascuna implementazione è considerata sostanzialmente un trait distinto:

```
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, esiste la cosiddetta *Orphan Rule*: si può implementare un trait per un tipo solo se almeno uno dei due (trait o tipo) è definito nel proprio crate [1]. Quindi se il trait è definito nel proprio crate, si può implementare per tipi definiti altrove, ad esempio *String* o *i32*. In Java, invece, non è possibile aggiungere un'interfaccia a un tipo già definito senza modificarne la classe. Quindi non si può implementare un'interfaccia di un tipo su cui non si ha accesso al codice sorgente. Per ottenere un comportamento simile a Rust, si possono utilizzare tecniche come l'utilizzo di classi *Wrapper*:

```
public interface MyInterface {
    void doSomething();
}

public class MyWrapper implements MyInterface {
    private final String value;

    public MyWrapper(String value) {
        this.value = value;
    }
```

1 Un crate è un pacchetto di codice Rust.

```
        @Override
        public void doSomething() { /* ... */ }
    }
```

## 3.5 MECCANISMI DI DISPATCH

Quando il codice coinvolge il polimorfismo, sono necessari meccanismi per determinare quale versione specifica 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.

### 3.5.1 *Static Dispatch*

In Rust, il meccanismo di static dispatch viene realizzato attraverso *generics* e i *trait bounds*. Quando si utilizza un tipo generico con un trait bound, il compilatore genera una versione specializzata della funzione per ogni tipo concreto utilizzato. Questo processo è noto come *monomorphization* (descritto in dettaglio nella Sezione 3.3). Per mostrare esplicitamente come lo static dispatch venga implementato in Rust consideriamo il seguente codice:

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

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

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

Durante la compilazione, tramite monomorphization, verranno create tre copie della funzione `static_dispatch()`: una per ogni tipo con cui è stata chiamata. Questo si può vedere utilizzando i seguenti comandi[2] dal terminale:

```
rustc -C opt-level=0 -C debuginfo=2 main.rs
lldb ./main --batch -o "disassemble --name drive" > disassembly.txt
```

Il comando genera un file `disassembly.txt` contenente la disassemblazione delle funzioni `drive` con i vari tipi, mostrando chiaramente che ogni funzione ha un indirizzo di memoria distinto:

2 Il primo comando compila il file Rust senza ottimizzazioni e con informazioni di debug dettagliate. Questo è necessario per produrre un eseguibile "leggibile" e completo per il debug. Il secondo comando utilizza LLDB, un debugger per linguaggi come C e Rust, per disassemblare la funzione `drive` e mostrare i suoi indirizzi di memoria.

```
Car::drive        -> 0x100000a2c
Motorcycle::drive -> 0x100000a64
Boat::drive       -> 0x100000a9c
```

Questo evidenzia come lo static dispatch risolva la chiamata alla funzione al momento della compilazione, senza alcun overhead a runtime.

In Java, lo *static dispatch* è realizzato tramite il *method overloading*. In questo meccanismo, più metodi condividono lo stesso nome ma differiscono per la lista dei parametri (numero o tipo). La scelta del metodo corretto viene risolta dal compilatore in base ai tipi statici degli argomenti.

Consideriamo il seguente esempio:

```
public class StaticDispatch {

    public double sum(int op1, int op2) {
        return op1 + op2;
    }

    public double sum(double op1, double op2) {
        return op1 + op2;
    }

    public static void main(String[] args) {
        double x = new StaticDispatch().sum(1, 1);
        double y = new StaticDispatch().sum(1.0, 1.0);
    }
}
```

In questo caso, abbiamo due metodi omonimi `sum`, rispettivamente con parametri di tipo `int` e `double`. Durante la compilazione, la chiamata `sum(1, 1)` viene risolta come invocazione del metodo `sum(int, int)`, mentre la chiamata `sum(1.0, 1.0)` come invocazione del metodo `sum(double, double)`.

Per osservare cosa accade a livello di bytecode, compiliamo ed utilizziamo il comando:

```
javac .\StaticDispatch.java
javap -c .\StaticDispatch.class > bytecode.txt
```

L'output (semplificato) scritto sul file `bytecode.txt` è il seguente:

```
public static void main(int[]);
  Code:
```

```
 0: new           #7
 3: dup
 4: invokespecial #9    // Method "<init>":()V
 7: iconst_1
 8: iconst_1
 9: invokevirtual #10   // Method sum:(II)D
12: dstore_1
13: new           #7
16: dup
17: invokespecial #9    // Method "<init>":()V
20: dconst_1
21: dconst_1
22: invokevirtual #14   // Method sum:(DD)D
25: dstore_3
26: return
```

Possiamo notare che:

- Alla riga 9, `invokevirtual #10` corrisponde a `sum:(II)D`, ovvero il metodo con parametri `int`.
- Alla riga 22, `invokevirtual #14` corrisponde a `sum:(DD)D`, ovvero il metodo con parametri `double`.

Queste informazioni provengono dal *constant pool* [3] del bytecode, dove il compilatore ha già registrato a quale firma di metodo corrisponde ciascuna chiamata. In altre parole, nonostante entrambe le invocazioni utilizzino l'istruzione `invokevirtual`, a runtime la JVM non ha bisogno di fare alcuna ricerca aggiuntiva: la firma è già stata scelta a compile-time in maniera univoca.

Sebbene in entrambi i linguaggi si parli di *static dispatch*, i meccanismi adottati sono profondamente diversi.In Rust, lo static dispatch si traduce in specializzazione del codice (funzioni distinte generate dal compilatore). In Java, invece, lo static dispatch si traduce in selezione della firma corretta (una voce distinta nel constant pool per ogni metodo). In entrambi i casi, la risoluzione avviene a compile-time e non vi è alcun overhead di dispatch a runtime.

3 La constant pool contiene le costanti necessarie per l'esecuzione del codice di una specifica classe. In pratica, è una struttura dati a runtime simile a una tabella dei simboli.

### 3.5.2 *Dynamic Dispatch*

Il meccanismo di generics e trait bounds appena descritto è adatto solamente quando si lavora con tipi omogenei di cui si conosce il tipo esatto a compile-time. Tuttavia, quando si desidera lavorare con tipi eterogenei o quando il tipo esatto non è noto fino a runtime, è necessario utilizzare il *dynamic dispatch*.

Il dynamic dispatch in Rust viene realizzato attraverso l'uso di *trait objects*. Un trait object è un puntatore a un tipo che implementa un determinato trait, consentendo di chiamare metodi definiti nel trait senza conoscere il tipo concreto a compile-time. Un trait object si definisce con la sintassi `Box<dyn Trait>`[4], dove `Trait` è il trait che si desidera utilizzare. Ad esempio, consideriamo l'implementazione del design pattern Observer: il `Subject` deve poter mantenere la collezione dei `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:

4 Al posto di `Box<>` si può utilizzare qualsiasi altro tipo di puntatatore. Questo è richiesto perché il compilatore di Rust deve sapere a compile time la dimensione della variabile.

```rust
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);
        }
    }
}
```

# 4

# NUMERICAL RESULTS

This is where you show that the novel 'thing' you described in Chapter **??** is, indeed, much better than the existing versions of the same.

You will probably use figures (try to use a high-resolution version), graphs, tables, and so on. An example is shown in Figure 5.

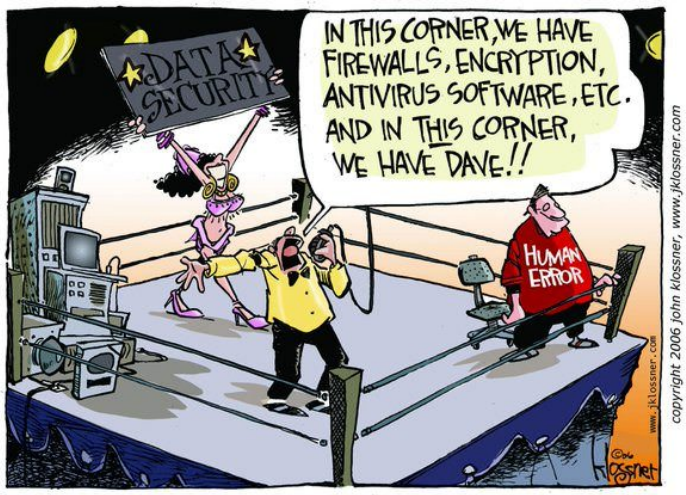


Figura 5: Network Security - the sad truth

Note that, likewise tables and listings, you shall not worry about where the figures are placed. Moreover, you should not add the file extension (LaTeX will pick the 'best' one for you) or the figure path.

# 5

# CONCLUSIONS AND FUTURE WORK

They say that the conclusions are the shortened version of the introduction, and while the Introduction uses future verbs (we will), the conclusions use the past verbs (we did). It is basically true.

In the conclusions, you might also mention the shortcomings of the present work and outline what are the likely, necessary, extension of it. E.g., we did analyse the performance of this network assuming that all the users are pedestrians, but it would be interesting to include in the study also the ones using bicycles or skateboards.

Finally, you are strongly encouraged to carefully spell check your text, also using automatic tools (like, e.g., Grammarly[1] for English language).

1 https://www.grammarly.com/

# BIBLIOGRAPHY

[1] Baeldung. Memory leaks in java, 2018.

[2] Dynatrace. Reducing garbage-collection pause time. Dynatrace eBook.

[3] Tony Hoare. Null references: The billion dollar mistake, August 2009. InfoQ Presentation.

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

[5] Jeff Vander Stoep. Memory safe languages in android 13, December 2022. Google Security Blog.