



UNIVERSITÀ
DEGLI STUDI
FIRENZE

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

Tesi di Laurea

ANALISI COMPARATIVA DI RUST E JAVA:
PARADIGMI, PERFORMANCE E
AFFIDABILITÀ

NICOLA PAPINI

Relatore/Relatrice: Lorenzo Bettini

Anno Accademico 2024-2025

CONTENTS

List of Figures	3
1 Introduction	7
2 Gestione della memoria	9
2.1 L'approccio di Java	9
2.2 L'approccio di Rust	10
2.3 Stack e Heap	10
2.4 Ownership	12
2.5 Borrowing	17
3 How to heat the water differently	23
3.1 How did I discover a novel type of heated water	23
3.2 How my heated water differs from the previous ones . . .	23
4 Numerical results	25
5 Conclusions and Future work	27
Bibliography	29

LIST OF FIGURES

Figure 1	Visualizzazione memoria Rust dopo il trasferimento di ownership.	13
Figure 2	Visualizzazione memoria Rust durante la chiamata di <code>print_string()</code>	19
Figure 3	Visualizzazione memoria Java durante la chiamata di <code>printString()</code>	19
Figure 4	Network Security - the sad truth	25

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

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.

GESTIONE DELLA MEMORIA

2.1 L'APPROCCIO DI JAVA

Java è un linguaggio di programmazione progettato per essere semplice da usare e portabile, con una gestione della memoria che mira a garantire sicurezza e facilità d'uso a scapito di un controllo diretto da parte del programmatore. Java raggiunge questi obiettivi principalmente attraverso l'uso di un *garbage collector* e un ambiente di esecuzione controllato (la Java Virtual Machine - JVM). Gli aspetti fondamentali dell'approccio di Java sono:

1. La gestione automatica della memoria tramite il garbage collector, che si occupa di identificare e liberare la memoria occupata da oggetti non più raggiungibili nel programma, prevenendo così problemi comuni, come i memory leak. Questo, di fatto, toglie al programmatore la responsabilità collegata al dover gestire manualmente l'allocazione e la deallocazione della memoria, riducendo la probabilità di errori nel codice.
2. La prevenzione di comportamenti indefiniti principalmente a runtime, attraverso un controllo rigoroso dell'accesso alla memoria e la gestione delle eccezioni.

Sebbene l'approccio di Java renda lo sviluppo più semplice e sicuro rispetto a linguaggi senza garbage collector, come C e C++, l'uso del garbage collector introduce una certa imprevedibilità nelle prestazioni, in quanto il processo di garbage collection deve essere eseguito in concorrenza con l'esecuzione del programma.

2.2 L'APPROCCIO DI RUST

Rust è un linguaggio di programmazione che si distingue per la sua gestione della memoria, evitando la necessità di un garbage collector e garantendo al contempo sicurezza e prestazioni elevate. 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. Prevenire 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. Inoltre, viene ridotto il numero di controlli a run-time, migliorando le prestazioni del programma.

Rust non può prevenire tutti i bug ma le metodologie messe in atto per la gestione della memoria rendono un programma scritto in Rust molto più sicuro rispetto a uno scritto in linguaggi con meno controlli.

Un esempio concreto dell'efficacia di Rust nella prevenzione degli errori di memoria è fornito da Google [2], 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, sono state scoperte zero vulnerabilità di sicurezza legate alla memoria in questo codice. Questo è un risultato significativo che dimostra come gli obiettivi prefissati da Rust siano stati raggiunti nella pratica.

Rust realizza questi obiettivi attraverso un sistema basato sui concetti di *ownership* e *borrowing*. Concetti fondamentali che verranno 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 con 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 non strutturata, 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 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, le 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`¹.
- 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, perché 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.

¹ `Box<T>` è uno smart pointer fornito dalla standard library di Rust che consente di allocare un valore di tipo `T` sull'heap.

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

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

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*, 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 quando la variabile esce dallo scope. In java questo non accade. Dato il seguente codice equivalente in Java:

```
{
    String s = "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);
```

In questo caso, l'ownership della stringa "Hello" viene trasferita da `s1` a `s2`. Dopo il trasferimento, `s1` non è più valida e qualsiasi tentativo di accedervi causerà un errore di compilazione.

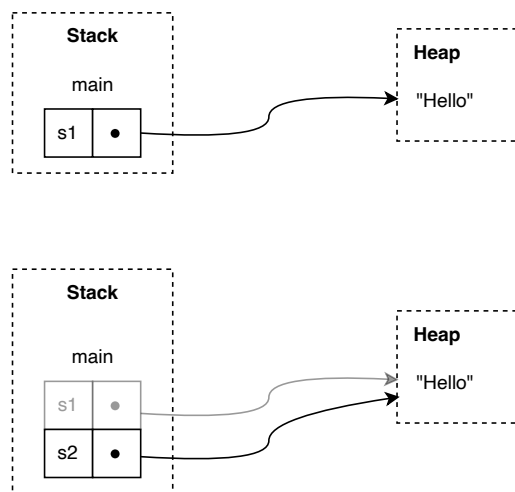


Figure 1: Visualizzazione memoria Rust dopo il trasferimento di ownership.

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*. 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, condividendolo 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`, 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. Consideriamo il seguente codice 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 p2
        System.out.println(p1.getName()); // Stampa "Bob"
    }
}
```

Codice 1: Modifica di un oggetto tramite un riferimento in Java.

In questo esempio, 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 difficili da individuare, specialmente in contesti complessi o concorrenti². 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.

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

² 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`.

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 `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 o modifiche inattese.
- Per i tipi con dimensione fissa e nota a compile time, che risiedono interamente nello stack e sono di solito immutabili per definizione (come interi o booleani)³, 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 restituito da una funzione comporta il trasferimento dell'*ownership* dalla funzione alla variabile. Ad esempio:

```
fn main() {
    let s1 = create_string();
```

³ In Rust le variabili sono immutabili a meno che non si vengano dichiarate con la keyword `mut`.

```

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 un valore a una 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. 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 calcola⁴.

```

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)
}

```

Codice 2: Trasferimento di ownership con ritorno di valore.

2.5 BORROWING

È evidente come l'ownership sia un concetto potente per la gestione della memoria, ma può risultare troppo restrittivo e macchinoso in situazioni come quella riportata nel codice (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. Un *borrowing* è realizzato attraverso il concetto di

⁴ Questo può essere fatto tramite il tipo `Tuple`: un array di dimensione fissa in cui è possibile memorizzare dati di tipo diverso

riferimento (reference). 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 punta a un 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"[1], 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, in generale, genera un riferimento alla variabile su cui viene applicato. Ad esempio:

```
fn main() {
    let s1 = String::from("Hello");
    print_string(&s);
}
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, si hanno due livelli di indirizzione: `s2` per poter accedere a "Hello" deve prima seguire il riferimento a `s1` e poi accedere al valore allocato nell'heap.

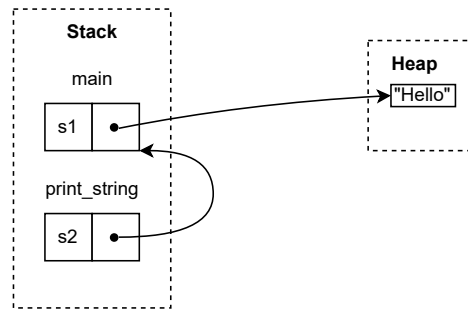


Figure 2: Visualizzazione memoria Rust durante la chiamata di `print_string()`.

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.

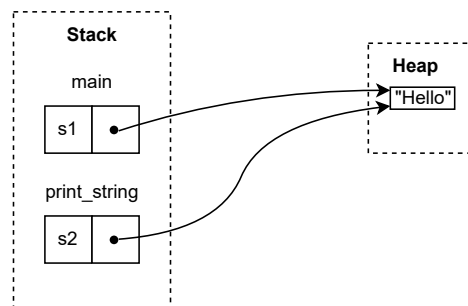


Figure 3: Visualizzazione memoria Java durante la chiamata di `printString()`.

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. Ad esempio:

- Deallocando il valore mentre un'altra parte del codice lo sta ancora utilizzando.
- Modificando il valore mentre un'altra parte del codice lo sta leggendo.
- Modificando il valore mentre un'altra parte del codice lo sta modificando.

In Java, i tre 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.

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  
    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. In particolare, lo scope di `r1` termina dopo la sua stampa, quindi se non ci fosse l'istruzione di stampa non si avrebbero errori di compilazione perché lo scope di `r1` terminerebbe dopo la sua dichiarazione. 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 non inizializzato o esplicitamente impostato a `null`. Il codice equivalente in Java sarebbe:

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

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 che punta non ha cambiato posizione in memoria.

HOW TO HEAT THE WATER DIFFERENTLY

This is the ‘main’ chapter of your thesis.

Here you have to show that *your* version of heated water differs slightly from any other known version of heated water, and this is important.

3.1 HOW DID I DISCOVER A NOVEL TYPE OF HEATED WATER

Explain in detail what are the steps to heat the water in a novel way.

3.2 HOW MY HEATED WATER DIFFERS FROM THE PREVIOUS ONES

Describe why and how your findings are different from the past versions.

Here you might want to add code (see for example Listing 1), or tables (see Table 1).

Note that figures, listings, tables, and so on, should never be placed ‘manually’. Let LaTeX decide where to put them - you’ll avoid headaches (and bad layouts). Furthermore, each of them must be referred to at least once in the body of the thesis.

Table 1: Example table

Country	Country code	ISO codes
Canada	1	CA / CAN
Italy	39	IT / ITA
Spain	34	ES / ESP
United States	1	US / USA

Listing 1: Python example

```

import numpy as np

def incmatrix(genl1,genl2):
    m = len(genl1)
    n = len(genl2)
    M = None #to become the incidence matrix
    VT = np.zeros((n*m,1), int) #dummy variable

    #compute the bitwise xor matrix
    M1 = bitxormatrix(genl1)
    M2 = np.triu(bitxormatrix(genl2),1)

    for i in range(m-1):
        for j in range(i+1, m):
            [r,c] = np.where(M2 == M1[i,j])
            for k in range(len(r)):
                VT[(i)*n + r[k]] = 1;
                VT[(i)*n + c[k]] = 1;
                VT[(j)*n + r[k]] = 1;
                VT[(j)*n + c[k]] = 1;

            if M is None:
                M = np.copy(VT)
            else:
                M = np.concatenate((M, VT), 1)

            VT = np.zeros((n*m,1), int)

    return M

```

NUMERICAL RESULTS

This is where you show that the novel ‘thing’ you described in Chapter 3 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 4.

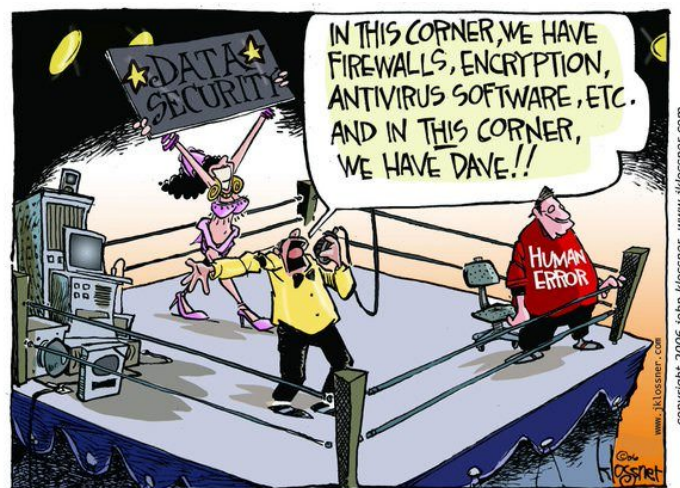


Figure 4: Network Security - the sad truth

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.

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., Grammarly¹ for English language).

¹ <https://www.grammarly.com/>

BIBLIOGRAPHY

- [1] Tony Hoare. Null references: The billion dollar mistake, August 2009. InfoQ Presentation.
- [2] Jeff Vander Stoep. Memory safe languages in android 13, December 2022. Google Security Blog.