1 Rust – Einführung

In diesem Kapitel werfen wir einen ersten Blick auf Rust-Programme, betrachten die Installation von Rust und der Sprachunterstützung in verschiedenen Entwicklungsumgebungen, sodass wir möglichst schnell praktische Schritte mit der Sprache unternehmen, ein Beispielprogramm schreiben und mit dem Rust-eigenen Build-System übersetzen und starten können.

1.1 Warum Rust?

Rust ist eine moderne Sprache, die sehr stark auf Geschwindigkeit und Parallelverarbeitung ausgelegt ist. Vielfach wird Rust als Systemprogrammiersprache und Ersatz für C dargestellt, der Anwendungsbereich ist aber sehr viel breiter. Betrachten wir ein paar der interessanten Eigenschaften von Rust.

1.1.1 Rust und der Speicher

Das absolute Alleinstellungsmerkmal ist die Art, wie Rust mit Speicher umgeht. Rust kann garantieren, dass durch die Verwaltung des Speichers zur Übersetzungszeit keine Fehler zur Laufzeit auftreten können. Damit braucht Rust auch keinen Garbage Collector. Das verhindert unbeabsichtigte Unterbrechungen im Programmablauf, um den Speicher aufzuräumen. Wir haben also nicht nur korrektere Programme, die schneller laufen, sie verhalten sich auch deterministischer.

Um dies zu erreichen, wird für jeden Wert ein Eigentümer festgelegt. Dies kann ein primitiver Wert sein oder eine beliebig komplexe Struktur. Ein Wert lebt, solange der Eigentümer lebt.

Der Eigentümer kann wechseln, und für den Zugriff auf ein Objekt können Referenzen ausgeliehen werden (*Borrowing*). Ausgeliehene Referenzen sind im Normalfall Lesereferenzen, es kann aber alternativ auch maximal eine Schreib-/Lese-Referenz auf einen Wert

definiert werden. Dies impliziert, dass wir keine aktive Lesereferenz haben. Die Beschränkung auf eine einzige schreibende Instanz sorgt bei Neulingen meist für Überraschungen, hat aber den großen Vorteil, dass es keine undefinierten Zustände durch gleichzeitiges Schreiben oder nicht synchronisiertes Lesen geben kann.

Dieses *Ownership* genannte Konzept ist extrem mächtig, braucht aber zum vollständigen Verinnerlichen etwas Zeit und Übung. Wir werden dies in Abschnitt 7.2 kennenlernen und in Kapitel 15 im Detail beleuchten.

1.1.2 Rust und Objektorientierung

Rust ist eine Programmiersprache, die mit der Kapselung von Daten und Funktionen und Methoden auf diesen Daten objektorientierte Konzepte unterstützt.

Rust erreicht dies durch die Einführung von Modulen, die private und öffentliche Daten und Funktionen enthalten. Polymorphismus wird durch das Konzept der *Traits* erreicht, die inzwischen in vielen anderen Programmiersprachen wie Kotlin oder Scala auch verwendet werden. Eine vergleichbare Funktionalität gibt es in Java seit der Version 8 mit den Default-Methoden in Interface-Spezifikationen.

Rust bietet allerdings anders als die gewohnten objektorientierten Sprachen keine Vererbung. Dies mag im ersten Moment überraschen und ist eine Abkehr vom normalen objektorientierten Denken, hat aber gute Gründe.

Aus konzeptioneller Sicht ist es problematisch, dass wir bei der Vererbung nicht kontrollieren können, welche Teile unserer Elternklasse wir erben möchten. Dies kann dazu führen, dass wir in abgeleiteten Klassen Funktionalität haben, die dort nicht gewollt ist.

Das praktischere Argument ist aber, dass durch Verzicht auf Vererbung ein hoher Aufwand zur Identifikation der richtigen auszuführenden Methode/Funktion wegfällt. Dies macht Rust-Programme deutlich laufzeiteffizienter.

Wir werden uns mit objektorientierten Konzepten in Kapitel 9 auseinandersetzen.

1.1.3 Rust und funktionale Programmierung

Zur Unterstützung funktionaler Programmierung bietet Rust *Closures*, anonyme Funktionen, die auf ihre Umgebung zur Zeit der Definition zugreifen können. Dieses vielseitige Konstrukt findet sich in mehr und mehr Sprachen und erlaubt eine sehr elegante Kapselung von Funktionalität und Daten.

Zusammen mit Iteratoren, die die Verarbeitung von Sammlungen von Daten kapseln, erlauben Closures sehr mächtige funktionale Abstraktionen. Iteratoren und Closures werden wir in Kapitel 11 kennenlernen.

1.1.4 Rust und Parallelverarbeitung

Rust bietet eine direkte Abstraktion der Thread-Funktionalität des unterliegenden Betriebssystems. Dies sorgt für den geringstmöglichen Mehraufwand zur Laufzeit, beschränkt aber natürlich die Flexibilität in der Verwendung von Threads auf die Unterstützung durch das unterliegende System. Bei Bedarf können allerdings auch Thread-Module verwendet werden, die eine unabhängige und damit flexiblere Implementierung anbieten. Dies erlaubt uns, von Fall zu Fall zu entscheiden, ob wir die größere Flexibilität oder den geringeren Speicherbedarf bevorzugen. Während die Entscheidung in vielen Fällen in Richtung der Flexibilität getroffen werden wird, gibt es eingeschränkte Umgebungen (wie zum Beispiel Mikro-Controller), in denen die Möglichkeit der expliziten Wahl sehr vorteilhaft ist.

Viele der Probleme, die bei der normalen Programmierung von paralleler Verarbeitung zu sehr hoher Komplexität und damit zu schwer auffindbaren Fehlern führen, finden wir in Rust nicht. Dies entsteht durch das *Ownership*-Modell, das dafür sorgt, dass der Compiler problematische Stellen im Quelltext sehr früh identifizieren und damit entfernen kann. Das heißt nicht, dass Rust alle Probleme im Zusammenhang mit Parallelprogrammierung löst. Es erlaubt uns aber, uns auf die wirklich schwierigen Probleme zu konzentrieren.

Threads in Rust können kommunizieren, indem sie Nachrichten in verschiedene Kanälen senden oder aus diesen empfangen. Zusätzlich können sie Teile ihres Zustands geschützt durch eine Mutex-Abstraktion mit anderen Threads teilen.

Parallelprogrammierung in Rust ist sehr mächtig, und wir werden uns in Kapitel 16 eingehend damit beschäftigen.

1.2 Ein Beispielprogramm

Als ein erstes Beispiel, um Ihren Appetit für Rust zu wecken, betrachten wir ein kleines Programm, das bereits viele Eigenschaften von Rust zeigt. Da dieses deutlich über das klassische »Hallo Welt«-Programm hinausgeht, sollten Sie sich keine Sorgen machen, wenn einzelne Funktionalitäten noch nicht vollständig klar sind. Die Erklärungen sind an dieser Stelle notwendigerweise etwas kurz, alle angesprochenen Eigenschaften werden wir später deutlich detaillierter betrachten.

Listing 1–1
Zeilenweises Lesen und Ausgabe einer Datei

```
use std::fs::File as Datei;
use std::io::{BufReader, BufRead};

fn main() {
    let file = Datei::open("hallo.txt")
        .expect("Konnte Datei nicht öffnen");
    let reader = BufReader::new(file);

    for line in reader.lines() {
        let line = line
            .expect("Konnte Zeile nicht lesen");
        println!("{}", line);
    }
}
```

Wir beginnen mit dem Import benötigter Funktionalität (wie auch aus Java bekannt). Wir benennen den aus dem Namensraum beziehungsweise Modul `std::fs` (durch den Pfadtrenner `::` getrennte Namen sind hierarchische Pfadangaben) importierten Typ `File` um in `Datei` und importieren im nächsten Schritt die beiden Typen `BufReader` und `BufRead`. Der Typ `BufReader` unterstützt gepuffertes Lesen aus einer Quelle und ist damit deutlich effizienter als ein direktes Lesen.

Dann folgt die Definition unserer ersten Funktion, gekennzeichnet durch das Schlüsselwort `fn`. In unserem Fall ist dies die Funktion `main()`, die keine Parameter und keinen Rückgabewert hat. Der Körper der Funktion findet sich im durch geschweifte Klammern definierten Block von Anweisungen, die durch Semikolon getrennt sind. Wie in vielen anderen Sprachen hat diese Funktion eine Sonderrolle: Sie ist der Einstiegspunkt in unser Programm und wird als Erstes aufgerufen, um den Programmabfluss zu starten.

Wir versuchen mit der Methode `open()` (eine Methode des Typs `Datei`, unserem umbenannten Typ `File`) eine Datei mit dem Namen *hallo.txt* zu öffnen. Dieser Aufruf liefert ein Objekt vom Typ `Result` zurück, das entweder das Ergebnis des erfolgreichen Aufrufs oder den durch den Aufruf ausgelösten Fehler enthält. Die Funktion `expect()` nimmt dieses Objekt und liefert im Erfolgsfall das Ergebnis zurück, im Fehlerfall wird die als Parameter übergebene Nachricht ausgegeben und das Programm mit einem Fehler beendet.

Hintergrund

Tatsächlich erzeugt die Funktion `expect()` einen nichtbehandelbaren Fehler vom Typ `std::Panic`. Dies ist ein völlig normaler Weg für Rust, Probleme zu behandeln, solange es der eigene Quelltext ist oder wir uns in der Prototypphase befinden. Im Falle von Produktionssoftware oder aber Bibliotheken sind andere Wege zur Fehlerbehandlung zu bevorzugen.

Wir weisen das Ergebnis, ein Objekt vom Typ `Datei`, der Variable `file` zu und erzeugen im nächsten Schritt ein Objekt vom Typ `BufReader` darauf, das wir in der Variable `reader` halten.

Nun iterieren wir mit einer `For`-Schleife über die einzelnen Zeilen der Datei in einem Iterator, den wir über den Aufruf von `reader.lines()` erhalten. Hierbei ist wichtig, dass die Funktion `lines()` die Zeilen ohne abschließenden Zeilenvorschub liefert. Der nachfolgende Block (durch geschweifte Klammern definiert) wird für jede Zeile ausgeführt. Das Ergebnis des Leseversuches landet als `Result`-Objekt in der Variablen `line`.

Auch hier extrahieren wir die eigentliche Zeichenkette wieder mit einem Aufruf der Funktion `expect()` mit einer Fehlermeldung, falls das Lesen nicht erfolgreich war. Das Ergebnis weisen wir der Variablen `line` zu. Der Effekt ist, dass wir keinen Zugriff mehr auf das `Result`-Objekt haben, das uns der Iterator zurückgegeben hat, sondern jetzt das eigentliche Resultat, die Zeichenkette mit der aktuellen Zeile der Datei, verwenden.

Hintergrund

Diese *Shadowing* genannte Funktionalität von Rust ist in vielen Fällen sehr vorteilhaft und elegant, kann aber bei Missbrauch zu schlechter Lesbarkeit führen. Der Umgang mit `Result`-Objekten ist einer der Fälle, in denen dies die Absicht des Programmierers klar kommuniziert.

Die letzte Anweisung unseres Programmes ist der Aufruf des Makros `println!`, das die Argumente mit einem abschließenden Zeilenvorschub ausgibt, so wie es die Formatzeichenkette (das erste Argument) vorgibt. Die Variante ohne Zeilenvorschub heißt `print!`.

Makros werden gekennzeichnet durch das Ausrufezeichen am Ende des Namens. Makros haben insbesondere aufgrund der Herausforderungen im Zusammenhang mit dem C-Präprozessor einen schlechten Ruf. In C entsteht dieser aus der direkten Ersetzung von Makroaufrufen durch ihren Inhalt, ohne dass der Präprozessor in irgendeiner Weise prüft, ob die Änderung syntaktisch korrekten Quelltext hinterlässt. Rust-Makros sind hier deutlich besser, da die Umsetzung durch den Compiler erfolgt und grundsätzlich gültige Ausdrücke erzeugt werden.

Das Makro `println!` ist ein exzellentes Beispiel für die Eleganz von Makros. In Rust müssen wir Funktionen mit der vollständigen Anzahl ihrer Parameter definieren, ein wichtiger Teil der Funktionalität von `println!` ist aber gerade, mit einer beliebigen Zahl von Parametern

Hintergrund

Es gibt zwei Arten von Makros in Rust, die deklarativen und die prozeduralen. In beiden Fällen übernimmt der Rust-Compiler die Aufgabe der Makroübersetzung, was Fehler sehr viel schneller und besser erkennen lässt.

Die deklarativen Makros sind relativ einfach zu schreiben, aber in ihrer Mächtigkeit etwas beschränkt. Wir werden später ein eigenes definieren.

Prozedurale Makros in Rust sind eine elegante Art der Metaprogrammierung, anders als der C-Präprozessor, der nur einfache Textersetzungen durchführt. Sie operieren direkt auf dem *Abstract Syntax Tree,* den der Rust-Compiler aus dem Quelltext erzeugt. Dies erlaubt eine extrem hohe Mächtigkeit dieser Art von Makros, dafür sind sie schwerer zu schreiben.

umgehen zu können. Das Rust-Makro `println!` erzeugt nun aus dem jeweiligen Quelltext den korrekten Aufruf der zugehörigen Bibliotheksfunktionen. Dies führt dazu, dass wir `println!` mit einer der Formatierungszeichenkette entsprechenden Anzahl von Argumenten aufrufen können.

Das erste Argument von `println!` ist diese Formatierungszeichenkette (ein Literal), die Formatierungsanweisungen und Platzhalter für die Ausgabe enthält. Die Zeichenkette muss ein Literal sein, da das Makro aus dieser den eigentlichen Code generiert (präziser: die Manipulationen des *Abstract Syntax Tree* durchführt).

In unserem Fall enthält diese Zeichenkette einfach einen Platzhalter `{}`, der durch das zweite Argument, unsere aktuelle Zeile, belegt wird. Dies führt dazu, dass alle Zeilen der Datei *hallo.txt* ausgegeben werden.

1.3 Installation von Rust

Die Rust-Entwicklung schreitet sehr schnell voran. Um dies zu reflektieren, werden in vergleichsweise kurzen Abständen (zum Zeitpunkt der Veröffentlichung alle 6 Wochen) neue Versionen von Rust veröffentlicht. Um die Installation jederzeit aktuell halten zu können, einfach zwischen verschiedenen Kanälen (*stable, beta, nightly*) wechseln zu können oder zum Beispiel Übersetzungen für andere Zielarchitekturen zu ermöglichen, stellt das Rust-Projekt das Werkzeug `rustup` zur Verfügung, das die Installation und Aktualisierung sehr stark vereinfacht. Hierbei werden normalerweise alle Werkzeuge im Verzeichnis `.cargo` im Benutzerverzeichnis installiert. Konfigurationsoptionen erlauben aber auch eine systemweite Installation.

Natürlich stehen auch jeweils aktuelle Installationspakete für die manuelle Installation bereit, aber für die Entwicklung mit Rust ist die Verwendung von `rustup` die beste Wahl.

1.3.1 Installation von rustup

Detaillierte Anweisungen inklusive aller Varianten zur Installation von `rustup` finden sich unter:

https://rust-lang.github.io/rustup/installation/index.html

Deshalb werden wir hier nur den einfachen Installationspfad betrachten.

1.3.1.1 Windows

Gehen Sie zur Website

https://www.rust-lang.org/tools/install

und laden Sie `Rustup-Init.exe` herunter. Nach der Ausführung des Installationsprogrammes können Sie den Erfolg der Installation testen, indem Sie ein CMD-Fenster öffnen und `rustc –version` eingeben. Falls dies nicht klappt, prüfen Sie, ob der Pfad korrekt erweitert wurde, und versuchen Sie den Aufruf mit `%userprofile%/.cargo/bin/rustc –version`.

1.3.1.2 Andere Systeme

Die allgemeine Methode, um `rustup` zu installieren, funktioniert für OSX, Linux, aber auch für das Linux-Subsystem unter Windows. Hierbei wird ein Skript ausgeführt, das von einem Server heruntergeladen wird. Man kann argumentieren, dass dies gefährlich ist. Es besteht aber natürlich die Möglichkeit, das Skript vor der Ausführung zu betrachten und zu prüfen. Es prüft, auf welcher Plattform es läuft, wählt dementsprechend das Installationsprogramm aus, lädt es herunter und führt es aus. Führen Sie das Skript mit dem folgenden Befehl aus:

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

Wenn Sie dem Skript nicht trauen, dann können Sie das Installationsprogramm auch von Hand identifizieren und herunterladen.

1.3.1.3 Installation über Paketmanager

Neben der direkten Installation gibt es auch die Möglichkeit der Installation über Paketmanager. Unter OSX gibt es `Homebrew` und `MacPorts` als bekannteste Paketmanager, unter Windows gibt es `Chocolatey` oder `Scoop`. Auch unter gängigen Linux-Distributionen gibt es die Möglich-

keit der Installation über Paketmanager, für die Entwicklung wird aber generell die Verwendung von `rustup` empfohlen.

Auch hier besteht natürlich das Problem des Vertrauens, es kann aber durchaus sinnvoll sein, dem Ersteller eines Pakets für den verwendeten Paketmanager mehr zu vertrauen als einem Skript von einem Webserver. Diese Entscheidung liegt alleine bei Ihnen.

1.4 IDE-Integration

Neben der direkten Verwendung der durch `rustup` zur Verfügung gestellten Werkzeuge auf der Kommandozeile gibt es auch sehr gute Unterstützung für die Programmierung in Rust durch verschiedene Entwicklungsumgebungen (IDE – Integrated Development Environment). Insbesondere das von den IDEs angebotene Auto-Vervollständigen, die Auflistung der Parameter von aufgerufenen Funktionen und die Unterstützung für Refactoring machen die Entwicklung deutlich effizienter. Die starke Integration eines Debuggers und einer Versionsverwaltung tut ihr Übriges, um schnell Software zu entwickeln.

1.4.1 Rust Language Server und Rust-Analyzer

Rust bietet zur Unterstützung von Entwicklungsumgebungen seit langer Zeit bereits den Rust Language Server RLS an, der im Hintergrund läuft und die IDE durch Informationen zu verwendeten Symbolen unterstützt. Diese Unterstützung beinhaltet Dokumentation, Umformatierung, Autovervollständigung, Refactoring, das Auffinden der Definition eines Symbols (dies ermöglicht in der Entwicklungsumgebung das Springen zur Funktion, die man aufruft) oder auch die Übersetzung im Hintergrund. Dies funktioniert in den meisten Fällen auch akzeptabel, allerdings wird RLS seit längerer Zeit nicht mehr weiterentwickelt und ist im Wartungsmodus.

Hintergrund

Die Idee eines Language Servers und des damit verbundenen Protokolls LSP (Language Server Protocol) wurde ursprünglich von Microsoft für Visual Studio Code entwickelt. Das dahinterliegende Konzept ist, dass der Aufwand für die Entwicklung von sprachspezifischen Funktionen wie Syntaxhervorhebung, Autovervollständigung, Refactoring bis hin zur Übersetzung aus der Entwicklungsumgebung extrahiert und in einen eigenen Prozess ausgelagert wird. Dies erlaubt die Entkopplung und Verwendung des Language Servers in verschiedenen Entwicklungsumgebungen. Das zugehörige Protokoll wurde standardisiert und wird mehr und mehr auch von anderen Entwicklungsumgebungen verwendet.

Die Alternative ist der Rust-Analyzer, eine Neuimplementierung des RLS. Dieser ist zwar noch in einer frühen Phase, trotzdem aber schon weiter entwickelt als RLS und bietet eine deutlich vollständigere Unterstützung. Nachdem inzwischen auch das Rust-Projekt selbst an einer Transition von RLS zu Rust-Analyzer arbeitet und sogar der Originalentwickler des RLS ganz explizit sagt, man solle Rust-Analyzer verwenden, empfehlen auch wir die Verwendung des Rust-Analyzers anstelle des RLS. Dieser wird zwar (zum Zeitpunkt der Veröffentlichung) immer noch als Preview und Alphaversion bezeichnet, wir haben aber mit dieser Implementierung nur positive Erfahrungen gemacht.

1.4.2 Visual Studio Code

Visual Studio Code ist eine kostenlose IDE von Microsoft, die auf dem Electron-Framework basiert. Damit ist sie plattformübergreifend in allen gängigen Systemumgebungen verfügbar. Visual Studio Code bietet zwei Erweiterungen, die die Entwicklung mit Rust unterstützen. Sie können diese IDE hier herunterladen:

https://code.visualstudio.com/

1.4.2.1 Erweiterungen zur Entwicklung mit Rust

Es gibt zwei Erweiterungen, die die Entwicklung von Rust in Visual Studio Code unterstützen: »Rust for Visual Studio Code« und »Rust-Analyzer«. Die erste Erweiterung nutzt den Rust Language Server, die zweite den Rust-Analyzer.

Da der Rust-Analyzer der von Rust empfohlene Language Server ist, ist die zugehörige Erweiterung die logische Wahl. Auch diese Erweiterung wird wie der Rust-Analyzer selbst (zum Zeitpunkt der

Veröffentlichung) immer noch als Preview und Alphaversion bezeichnet, unsere Erfahrungen sind aber ausnahmslos positiv.

Zur Installation wechseln Sie in die Extensions-Sicht und geben in dem Suchfeld »Rust« ein. Die beiden beschriebenen Plugins sollten direkt angezeigt werden. Wählen Sie das »Rust-Analyzer«-Plugin aus und klicken Sie auf »Install«. Das Plugin installiert den Rust-Analyzer mit, sodass sie hier keinen zusätzlichen Aufwand haben.

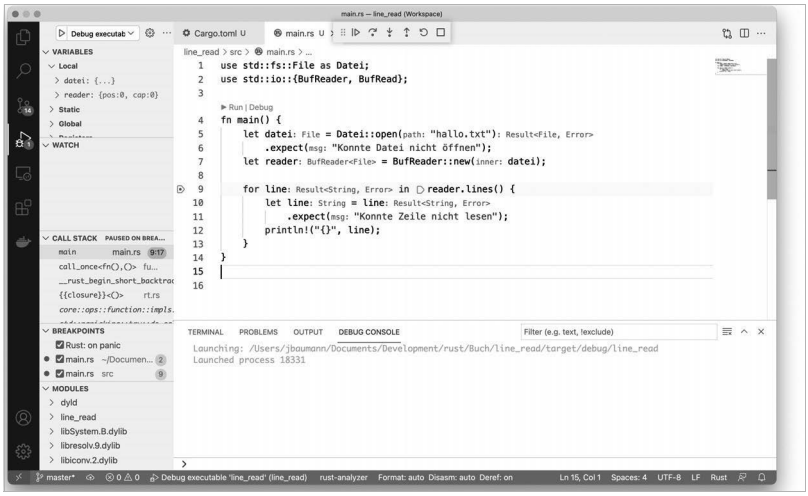

Abb. 1–1
Debugging-Session mit Visual Studio Code

1.4.3 IntelliJ IDEA

IntelliJ IDEA ist eine sehr leistungsfähige Entwicklungsumgebung der Firma JetBrains, die es sowohl in einer kostenlosen Community-Version als auch in einer kommerziellen Ultimate-Version gibt. Sie finden die verschiedenen Versionen der IDE hier:

https://www.jetbrains.com/de-de/idea/

Das für diese IDE verfügbare Rust-Plugin ist unabhängig sowohl von RLS als auch von Rust-Analyzer implementiert, und es bietet eine sehr weitreichende Unterstützung für Rust an. Insbesondere die Refactoring-Unterstützung ist vorbildlich.

Der Nachteil ist allerdings, dass Debugging mit dem Rust-Plugin nur in der Ultimate-Edition freigeschaltet wird, es also keine kostenlose Unterstützung für Debugging gibt.

Aber auch ohne Debugging bietet das Plugin eine große Menge an Funktionalität und ist empfehlenswert, insbesondere wenn IntelliJ bereits für andere Projekte in Verwendung ist.

Zur Installation wechseln Sie in die Einstellungen (*Preferences*), wählen dort Plugins, suchen nach »Rust« und installieren das Rust-Plugin.

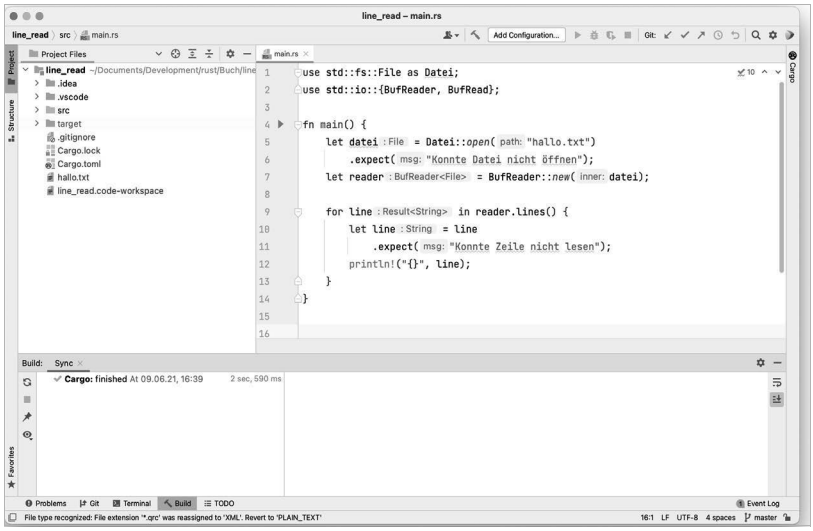

Abb. 1–2
Jetbrain Intellij IDEA in Aktion

1.4.4 Eclipse

Eclipse bietet mit der »Corrosion«-Erweiterung eine Unterstützung für Rust-Programmierung, die auf dem Rust-Analyzer basiert.

Diese Erweiterung für Eclipse ist gefühlt noch nicht ganz so weit wie zum Beispiel die Unterstützung in Visual Studio Code, aber durchaus verwendbar. Aufgrund der Tatsache, dass der Rust-Analyzer verwendet wird, stehen sämtliche der hierdurch bereitgestellten Funktionalitäten ähnlich zur Verfügung wie in Visual Studio Code. Sie finden die verschiedenen Versionen der Eclipse-IDE hier:

https://www.eclipse.org/downloads/

Corrosion erwartet eine Installation des Rust-Analyzers, die Sie getrennt vornehmen müssen. Das jeweils aktuelle Release finden Sie auf Github zum Herunterladen.

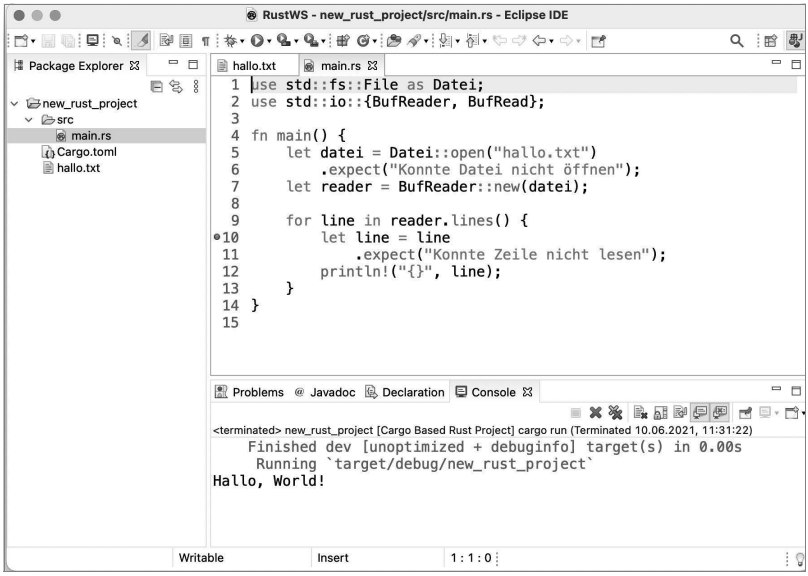

Abb. 1–3
Programmausführung mit Eclipse Corrosion

1.4.5 Welche Entwicklungsumgebung ist die beste?

Aufgrund der sehr guten Unterstützung der Sprachspezifika durch den Rust-Analyzer, der sowohl von Visual Studio Code als auch von Eclipse Corrosion integriert wird, lässt sich die Entscheidung frei nach dem eigenen Geschmack treffen, wenn Sie eine kostenlose Entwicklungsumgebung nutzen wollen. Egal ob Sie sich mit Eclipse oder mit Visual Studio Code wohler fühlen, Sie bekommen in beiden Fällen eine gute Unterstützung.

Wenn Sie bereit sind, Geld auszugeben, oder falls Sie die Ultimate Edition von IntelliJ IDEA für andere Zwecke bereits erworben haben, dann haben Sie hier eine fantastische Unterstützung durch das zugehörige Rust-Plugin.

Zu guter Letzt können Sie auch mit einem Editor wie dem VIM oder Emacs mit Syntaxhervorhebungen gut arbeiten. Diese bieten ebenso Unterstützung für das Language-Server-Protokoll an und damit ähnliche Funktionalität wie die bereits genannten Entwicklungsumgebungen.

Werkzeuge

Wenn wir uns über die Kommandozeile Gedanken machen, kommen wir irgendwann auch zum Thema Debugging. Rust bietet nicht nur die Unterstützung für den seit 30 Jahren konstant weiterentwickelten GDB an, sondern auch den neueren LLDB, der auf der LLVM-Infrastruktur basiert (auch Visual Studio Code bietet die Möglichkeit, nicht nur GDB zu verwenden, sondern über eine Erweiterung auch den LLDB). Die zugehörigen Kommandos lauten `rust-gdb` und `rust-lldb`.

Aktuell schlagen wir die Verwendung von GDB vor, da es für diesen eine schier endlose Menge an Frontends gibt, die die Verwendung vereinfachen.

1.5 Unsere erste praktische Erfahrung

Nachdem wir jetzt einen ersten Blick auf die Rust-Syntax geworfen, Rust auf unserem System installiert und uns die zur Verfügung stehenden Entwicklungsumgebungen kurz angeschaut haben, wollen wir die ersten praktischen Schritte machen.

Für diese Erfahrung wählen wir das klassische HelloWorld-Programm, mit dem wir die ersten Tests der Rust-Werkzeuge durchführen. Im folgenden Listing finden wir den Quelltext für dieses simple Programm. Natürlich können Sie genauso gut das Programm aus unserem ersten Listing verwenden.

```
fn main() {
    println!("Hallo Welt!");
}
```

Listing 1–2
Das klassische Hello-World-Programm

Wir definieren die Funktion `main()`, in der wir das Makro `println!` aufrufen mit der Zeichenkette »Hallo Welt!«. Wir speichern dieses Programm unter dem Namen `hallo_welt.rs` (`.rs` ist die Endung, die typischerweise für Quelltext in Rust verwendet wird).

Aufruf des Rust-Compilers

Um dieses Programm zu übersetzen, rufen wir den Rust-Compiler auf der Kommandozeile auf:

```
> rustc hallo_welt.rs
```

Der Compiler übersetzt jetzt den Quelltext und produziert ein ausführbares Programm mit dem gleichen Namen wie der Quelltext `hallo_welt`. Wir starten das Programm auf der Kommandozeile:

```
> ./hallo_welt
Hallo Welt!
>
```