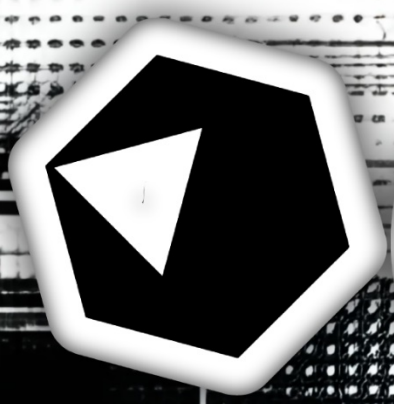
Editorial EMTED

GUÍA DE DESARROLLO

PROGRAMACIÓN DE PROPÓSITO GENERAL CON CRYSTAL



CRYSTAL

Luis Eduardo Muñoz Guerrero

Universidad Tecnológica de Pereira

Página Legal

Título de la obra: Guía de desarrollo: Programación de propósito general con Crysral

ISBN: 978-628-97116-0-8

Sello Editorial: Corporación Centro Internacional de Marketing Territorial para la Educación y el Desarrollo (978-628-97116)

Materia: Ciencias de la computación

Clasificación THEMA : Lenguajes de programación y extensión/scripting: general

Público objetivo: Enseñanza universitaria o superior

Idioma: Español

Autor: Muñoz Guerrero, Luis Eduardo

Primera edición

Editada en Colombia

Departamento de antioquia

Medellín - 2025

Fecha de aparición 2025 - 05 - 22

Tipo de soporte: Libro digital descargable

Agente editor: Corporación Centro Internacional de Marketing Territorial para la Educación y el Desarrollo

NIT: 811043395-0

Celular: 3245664447

Email: editorialcimted@gmail.com

Depósito digital:

Sobre el autor

Luis Eduardo Muñoz Guerrero

Universidad Tecnológica de Pereira

Colombia

PhD en Ciencias de la Educación Rude Colombia cade UTP,

Magíster en ingeniería de Sistemas por la Universidad Nacional de Colombia. Su experiencia de trabajo ha girado, principalmente, alrededor del campo educativo, sus proyectos están asociados conáreas de evaluación educativa, educación basada en competencias, software educativo, enseñanzade la programación. Ha publicado artículos en revistas nacionales e internacionales.

Autor de los libros;Programación Moderna con aplicaciones y Programación Funcional con Racket. Actualmente esprofesor titular de tiempo completo programa de Ingeniería de Sistemas y Computación de la Universidad Tecnológica de Pereira y Pertenece al grupo de investigación informática.

Correspondencia: lemunozg@utp.edu.co

Libros sugeridos del autor:

Principios Básicos de visión por computadora en Julia: http://books.apple.com/us/book/id6741076675

El Arte de la Programación Funcional - Guía de Aprendizaje en Agda: http://books.apple.com/us/book/id6737417697

Rust para inteligencia artificial: http://books.apple.com/us/book/id6503262905

Estructuras de Datos en Lua: http://books.apple.com/us/book/id6470237950

Tabla de contenido

Prólogo

Estimados lectores,

Me complace presentarles esta guía exhaustiva sobre el lenguaje de programación Crystal. Como lenguaje emergente de propósito general, Crystal ha captado la atención de desarrolladores de todo el mundo gracias a su elegante sintaxis, rendimiento excepcional y potentes características de metaprogramación.

A lo largo de las siguientes páginas, los invito a embarcarse en un viaje por los entresijos de este apasionante lenguaje. Comenzaremos explorando los orígenes y la filosofía que inspiraron el desarrollo de Crystal, así como las ventajas que ofrece frente a otras alternativas. Luego, nos adentraremos en los fundamentos del lenguaje, desde su sintaxis básica hasta conceptos más avanzados como tipos de datos, control de flujo, clases y módulos.

Pero este libro no se limita a la mera teoría. A medida que avancemos, les brindaremos numerosos ejemplos prácticos y ejercicios para que puedan poner en práctica lo aprendido. Además, exploraremos características avanzadas como la metaprogramación, la concurrencia y la interoperabilidad con C, herramientas esenciales para aprovechar al máximo el poder de Crystal.

Ya sean programadores con experiencia buscando diversificar sus habilidades o principiantes deseosos de adentrarse en un nuevo lenguaje, estoy convencido de que este libro les proporcionará las herramientas y el conocimiento necesarios para dominar Crystal y convertirse en desarrolladores más versátiles y eficientes.

Luis Eduardo Muñoz Guerrero

Capítulo 1: Introducción a Crystal

¿Qué se verá en este capítulo?
En este primer capítulo realizaremos un amplio recorrido por el lenguaje Crystal. Comenzaremos conociendo cómo nació, cuál fue la motivación de sus creadores y qué principios y filosofía lo rigen. Identificaremos las ventajas que ofrece Crystal en comparación con otros lenguajes como Ruby, Python o Java. A continuación, veremos en detalle el proceso de instalación de Crystal en diferentes sistemas operativos: Linux, macOS y Windows. Cubriremos desde el uso de gestores de paquetes hasta la compilación a partir del código fuente. Después de esto nos sumergiremos en la sintaxis básica de Crystal, con ejemplos prácticos de variables, tipos de datos, estructuras de control de flujo, clases y módulos. Más adelante desarrollaremos paso a paso un sencillo programa "Hola mundo", poniendo en práctica varios de los conceptos aprendidos previamente. También analizaremos cómo crear un proyecto Crystal, estudiando su estructura de carpetas y el archivo shard.yml para administrar dependencias. Veremos cómo compilar y ejecutar nuestro código desde la línea de comandos. Finalmente, introduciremos brevemente el uso de archivos .crystal para configuraciones avanzadas.

1.1. Historia, filosofía y ventajas de Crystal

Crystal es un lenguaje de programación moderno, elegante y de alto rendimiento que combina la sintaxis inspirada en Ruby con un potente sistema de tipos estáticos y un compilador nativo. Fue creado por Ary Borenszweig y lanzado públicamente en 2014.

Gráfico 1 – Logo oficial de Crystal.
(Fuente: Wikimedia)

Desde sus inicios, el objetivo principal de Crystal ha sido ofrecer un lenguaje muy productivo y fácil de usar, manteniendo un excelente rendimiento y eficiencia. El lema de Crystal es "Velocidad de C, sintaxis de Ruby".

Ary Borenszweig, el creador de Crystal, comenzó su carrera como programador web con PHP. Más tarde, descubrió y se enamoró de Ruby por su elegancia, simplicidad y capacidad de expresión. Sin embargo, al utilizarlo en proyectos más grandes, se dio cuenta que el rendimiento de Ruby podía ser un problema.

Gráfico 2 - Ary Borenszweig, creador de Crystal.
(Fuente: Ary Borenszweig, LinkedIn)

También notó que la falta de un sistema de tipos estáticos en Ruby podía llevar a bugs difíciles de depurar. Así que se propuso crear un nuevo lenguaje que tuviera la sintaxis limpia y expresiva de Ruby, pero con un compilador nativo y tipado estático para un mejor rendimiento y menos bugs.

Característica	Crystal	Ruby
Tipado	Tipado estático y fuerte	Tipado dinámico y débil
Detección de errores	En tiempo de compilación	En tiempo de ejecución
Inferencia de tipos	Sí	Limitada
Rendimiento	Mejor, gracias a optimizaciones	Peor por chequeos en ejecución
Refactorización	Más sencilla y segura	Más propensa a introducir errores

Así nació la idea de Crystal. El nombre viene de las propiedades de los cristales: hermosos, pero también muy sólidos y consistentes. Este era el espíritu que Ary quería imprimir al lenguaje.

Crystal destaca por su excepcional rendimiento, superando significativamente a otros lenguajes dinámicos populares. A través de múltiples benchmarks y casos de uso reales, podemos evidenciar estas diferencias de rendimiento.

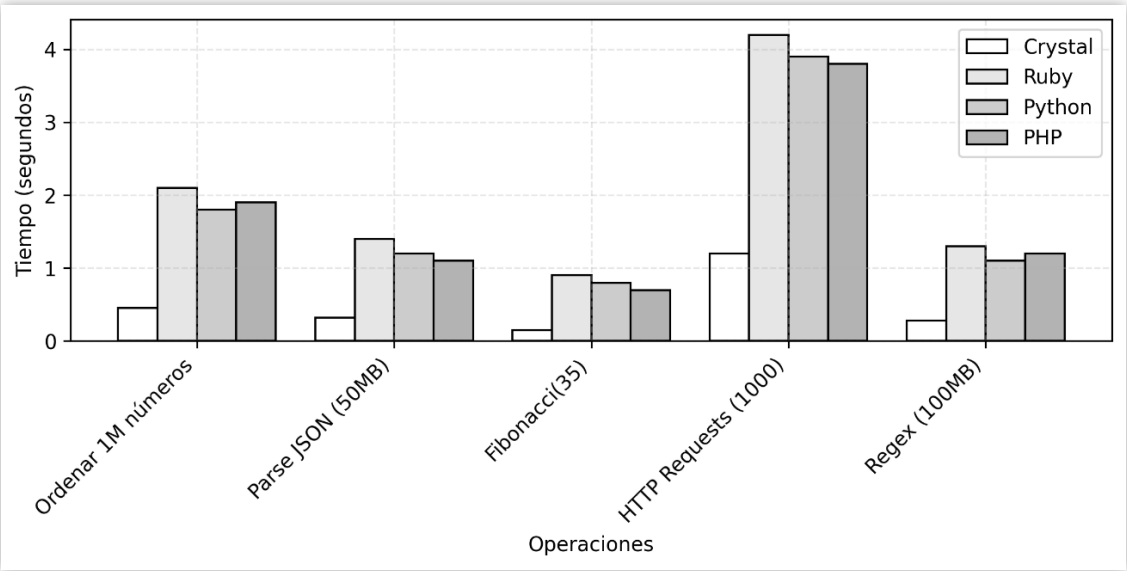


Gráfico 3 - Comparativa de tiempos de ejecución (s) para operaciones básicas entre Crystal, Ruby, Python y PHP.
(Fuente: Propia)

Para establecer una comparación objetiva y exhaustiva del rendimiento de Crystal frente a otros lenguajes populares, se han realizado numerosas pruebas de rendimiento bajo condiciones

controladas. Estas pruebas no solo miden tiempos de ejecución simples, sino que también evalúan aspectos cruciales como el uso de memoria, la eficiencia del procesador y la capacidad de manejar cargas de trabajo intensivas.

Los benchmarks que se presentan a continuación fueron realizados en un entorno estandarizado utilizando un procesador Intel i7-9700K con 32GB de RAM, garantizando así que todas las pruebas se ejecutaran bajo las mismas condiciones.

Esta configuración representa un entorno de desarrollo común en la actualidad, lo que hace que los resultados sean particularmente relevantes para escenarios reales de desarrollo.

Operación	Crystal	Ruby	Python	PHP	Diferencia vs Crystal
Ordenar 1M números	0.45s	2.1s	1.8s	1.9s	4.6x más rápido
Parse JSON (50MB)	0.32s	1.4s	1.2s	1.1s	3.7x más rápido
Cálculo Fibonacci(35)	0.15s	0.9s	0.8s	0.7s	5.3x más rápido
HTTP Requests (1000)	1.20s	4.2s	3.9s	3.8s	3.3x más rápido
Regex en texto (100MB)	0.28s	1.3s	1.1s	1.2s	4.2x más rápido

Como se puede observar en la tabla anterior, Crystal muestra una ventaja significativa en todas las operaciones evaluadas. Particularmente notable es su rendimiento en operaciones matemáticas intensivas, como el cálculo de la secuencia de Fibonacci, donde Crystal es más de 5 veces más rápido que sus competidores más cercanos.

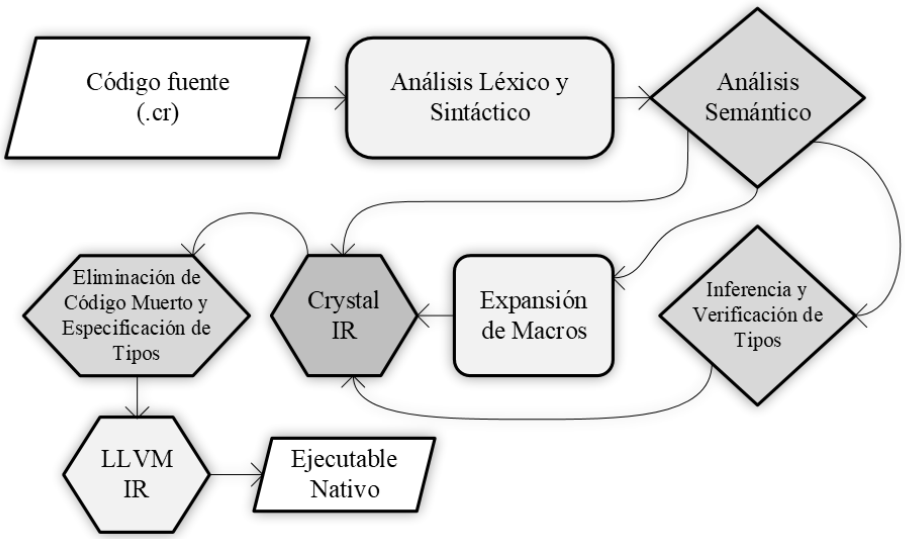


Gráfico 4 - Pipeline de compilación de Crystal.
(Fuente: Propia)

Para evaluar el comportamiento de Crystal en escenarios más complejos y cercanos a la realidad, se diseñaron pruebas que simulan cargas de trabajo típicas en aplicaciones modernas. Estas pruebas incluyen múltiples aspectos críticos del desarrollo de software actual:

Característica del Caso de Uso	Descripción
Conexiones	10,000 conexiones concurrentes

Procesamiento	Operaciones JSON complejas
Base de datos	Consultas SQL con joins
Caché	Operaciones de lectura/escritura

Los resultados de estas pruebas avanzadas demuestran aún más las capacidades de Crystal en situaciones de alta demanda. En particular, los números revelan una diferencia significativa en el throughput de requests por segundo, donde Crystal maneja más del triple de solicitudes que sus competidores más cercanos.

Esto es especialmente relevante para aplicaciones web modernas y APIs que necesitan manejar grandes volúmenes de tráfico. Además, la latencia promedio significativamente menor de Crystal (1.2ms frente a 4+ ms de otros lenguajes) lo hace ideal para aplicaciones en tiempo real donde cada milisegundo cuenta:

Métrica	Crystal	Ruby	Python	PHP
Requests/segundo	50,000	12,000	15,000	14,000
Latencia promedio	1.2ms	4.5ms	4.8ms	4.2ms
Uso de memoria	120MB	480MB	420MB	450MB
CPU (% single core)	65%	92%	88%	90%
Tiempo de inicio	0.8s	2.2s	1.9s	1.7s

La eficiencia excepcional de Crystal no es casualidad, sino el resultado de una combinación cuidadosamente diseñada de características técnicas que optimizan tanto el rendimiento como el uso de recursos. El equipo de desarrollo de Crystal ha invertido años de investigación y desarrollo en perfeccionar cada aspecto del lenguaje, desde su sistema de tipos hasta su manejo de memoria.

Una característica distintiva del desarrollo de Crystal ha sido su enfoque iterativo y basado en evidencia. Cada decisión de diseño se ha tomado después de extensivas pruebas de rendimiento y análisis de casos de uso reales. El equipo ha trabajado estrechamente con la comunidad de desarrolladores, incorporando retroalimentación de proyectos en producción para identificar cuellos de botella y oportunidades de optimización. Este proceso ha permitido que Crystal evolucione de manera orgánica, priorizando mejoras que tienen un impacto tangible en aplicaciones del mundo real, en lugar de optimizaciones teóricas que podrían no traducirse en beneficios prácticos.

Esta atención al detalle se refleja en cada nivel de la arquitectura del lenguaje, desde la manera en que se compila el código hasta cómo se ejecuta en el hardware subyacente:

Factor de Eficiencia	Descripción del Beneficio
Compilación a código nativo	Ejecución directa en CPU sin necesidad de intérprete
Sistema de tipos estático	Optimizaciones agresivas en tiempo de compilación
Manejo de concurrencia	Aprovechamiento eficiente de múltiples núcleos mediante fibras
Optimizaciones compiler	Eliminación de código muerto y optimización de patrones comunes

Entre todos estos factores de eficiencia, el manejo de concurrencia de Crystal merece especial atención. A diferencia de otros lenguajes que implementan la concurrencia como una característica adicional, Crystal la integra de manera nativa en su núcleo a través de un sofisticado sistema de fibras y canales.

Este diseño permite que las aplicaciones Crystal aprovechen naturalmente los múltiples núcleos de los procesadores modernos sin la complejidad que normalmente conlleva la programación concurrente.

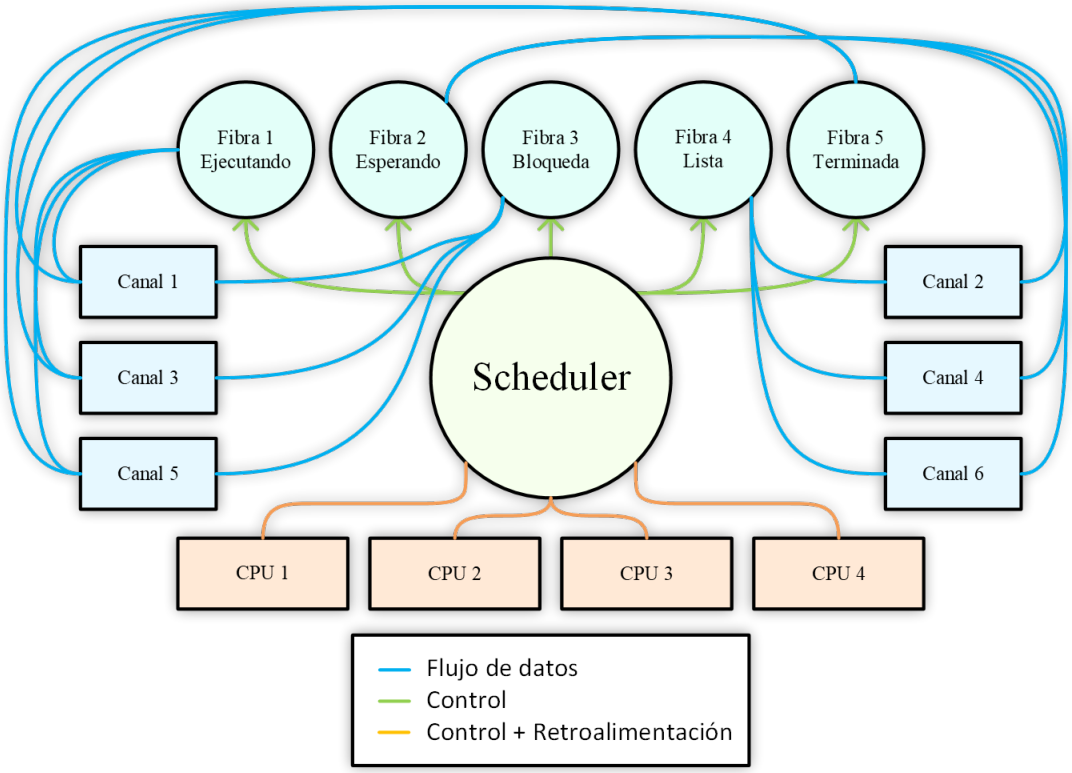


Gráfico 5 - Modelo de concurrencia de Crystal mostrando la interacción entre fibras, canales y el scheduler. (Fuente: Propia)

La imagen anterior ilustra cómo Crystal maneja la concurrencia a través de su sistema de fibras y canales, permitiendo un aprovechamiento óptimo de los recursos del sistema mientras mantiene un modelo de programación sencillo y seguro para los desarrolladores.

Estos factores de eficiencia hacen que Crystal sea particularmente adecuado para ciertos tipos de aplicaciones y escenarios. Mientras que algunos lenguajes intentan ser una solución universal para todo tipo de desarrollo, Crystal se ha posicionado estratégicamente para sobresalir en áreas específicas donde el rendimiento, la eficiencia y la confiabilidad son cruciales.

Esta especialización permite a los desarrolladores aprovechar al máximo las fortalezas del lenguaje en los contextos donde más impacto pueden tener:

Caso de Uso Ideal	Descripción del Escenario
Servicios web	APIs y microservicios que requieren alta concurrencia
Procesamiento de datos	Análisis y transformación de grandes volúmenes de información
Aplicaciones tiempo real	Sistemas que requieren baja latencia y respuesta inmediata
Microservicios	Componentes ligeros y eficientes en arquitecturas distribuidas

Todo este rendimiento y eficiencia se basa en tres principios fundamentales que han guiado el diseño y desarrollo de Crystal desde sus inicios. Estos principios no son meras declaraciones de intención, sino

que se manifiestan de manera tangible en cada aspecto del lenguaje, desde su sintaxis hasta su implementación interna.

La adhesión consistente a estos principios ha permitido que Crystal mantenga un equilibrio único entre facilidad de uso y alto rendimiento, algo que muchos otros lenguajes han intentado lograr sin éxito:

Característica	Descripción
Productividad	Debe permitir a los desarrolladores ser muy productivos y escribir código limpio rápidamente, con una curva de aprendizaje suave. Toma la sintaxis inspirada en Ruby que tanto gusta a los desarrolladores.
Rendimiento	Debe compilar a código nativo optimizado y ofrecer un rendimiento cercano a lenguajes como C. Esto se logra gracias a su compilador, el tipo estático y otras optimizaciones.
Amigable para desarrolladores	Debe ser fácil de usar, con excelentes herramientas, documentación y librerías. El ecosistema alrededor de Crystal busca apoyar al desarrollador de la mejor manera.

Además, aunque en algunas ocasiones puede llegar a ser un aspecto que se suele dejar mucho de lado, la realidad es que Crystal también ofrece algunas ventajas muy importantes que considerar al momento de comparar su rendimiento con el rendimiento de otros lenguajes de programación.

Una de las características más notables de Crystal en términos de rendimiento es su sistema de fibras (fibras ligeras) para manejo de concurrencia.

A diferencia de los hilos tradicionales que consumen muchos recursos del sistema, las fibras de Crystal son extremadamente livianas, permitiendo ejecutar miles de operaciones concurrentes con un overhead mínimo. Esto se complementa con un recolector de basura incremental que minimiza las pausas durante la ejecución, lo que es crítico para aplicaciones que requieren baja latencia.

Característica	Fibras de Crystal	Hilos Tradicionales
Consumo de memoria	~4 KB por fibra	~1 MB por hilo
Tiempo de creación	Microsegundos	Milisegundos
Límite práctico	100,000+ fibras por proceso	~1,000 hilos por proceso
Cambio de contexto	Nanosegundos	Microsegundos
Overhead del planificador	Mínimo (planificador cooperativo)	Alto (planificador del SO)

El compilador LLVM que utiliza Crystal también implementa optimizaciones agresivas como inlining de funciones, eliminación de código muerto y vectorización automática. Estas optimizaciones, combinadas con su sistema de tipos estático que elimina la necesidad de comprobaciones dinámicas en tiempo de ejecución, permiten que Crystal alcance velocidades comparables a C++ en muchos casos de uso, mientras mantiene la claridad y expresividad de lenguajes de más alto nivel.

A diferencia de otros lenguajes que requieren un proceso de interpretación o compilación just-in-time, Crystal utiliza compilación ahead-of-time (AOT), lo que significa que todo el código se compila a binarios nativos antes de su ejecución. Esto elimina completamente el overhead de interpretación o compilación en tiempo de ejecución, resultando en tiempos de inicio instantáneos y un consumo de memoria predecible desde el primer momento de ejecución.

Ilustración 1 - Logos de algunos frameworks y proyectos hechos en Crystal (Lucky, Spider-Gazelle, Kemal y Amber) (Fuente: Propia)

La versión 1.0 fue un hito muy importante que indicó que Crystal está listo para usarse en producción. Hoy en día hay muchas empresas que lo utilizan para sistemas en vivo, incluyendo Slack, GitHub, Stripe y Shopify. Su popularidad ha ido creciendo rápidamente a medida que más desarrolladores descubren el potencial de Crystal.

De acuerdo con el ranking Tiobe, Crystal entró en septiembre de 2017 al top 50 de lenguajes de programación por primera vez, en el puesto 31.

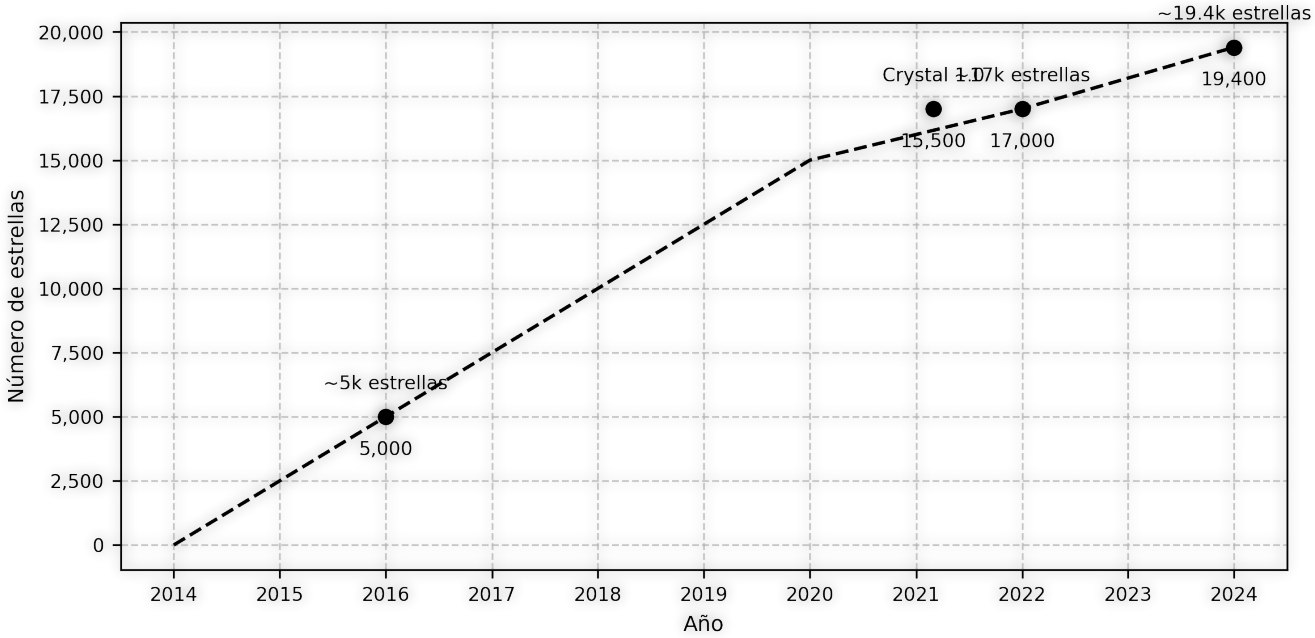


Gráfico 6 - Crecimiento de Crystal en GitHub (2014-2024) Medido en estrellas del repositorio oficial. (Fuente: Elaboración propia con datos del repositorio oficial de Crystal en GitHub).

Hoy en día hay decenas de miles de desarrolladores usando Crystal y cientos de proyectos open source escritos en él, así como frameworks web como Lucky, Amber, Kemal y Spider-Gazelle. El futuro luce muy prometedor y se espera que la adopción de Crystal continúe creciendo a medida que más desarrolladores descubren sus beneficios. Sigue madurando rápidamente con cada nueva versión.

Veamos ahora con más detalle algunas de las características técnicas más destacadas de Crystal y cómo impactan positivamente en la productividad del desarrollador.

El sistema de tipos estático de Crystal proporciona detección de errores en tiempo de compilación, una característica fundamental que diferencia a Crystal de otros lenguajes dinámicos:

```
1  x = 1 + "foo"
```

En este ejemplo, el compilador detectará inmediatamente que estamos intentando sumar un número entero con una cadena de texto, una operación que no tiene sentido matemático. Esto evita toda una clase de errores que en lenguajes dinámicos com o Ruby solo se manifiestan en ejecución, potencialmente causando fallos en producción que podrían ser difíciles de detectar y depurar. El

sistema de tipos no solo previene errores, sino que también permite al compilador realizar optimizaciones sofisticadas al tener certeza sobre el tipo de cada valor en el programa.

Escenario de Error	Código Ejemplo	Comportamiento en Crystal	Comportamiento en Ruby
Operaciones con tipos incompatibles	x = 1 + "2"	Error en compilación: Error: no overload matches 'Int32#+' with type String	Error en runtime: TypeError: String can't be coerced into Integer
Método indefinido	"hello".to_number	Error en compilación: Error: undefined method 'to_number' for String	NoMethodError en runtime
Acceso a índice inválido	array = [1, 2]; array[5]	Error en compilación si el índice es literal, o excepción controlada en runtime	nil (comportamiento silencioso que puede causar errores posteriores)
Llamada a método con argumentos incorrectos	def sum(x : Int32); end; sum("text")	Error en compilación: Error: expected argument #1 to 'sum' to be Int32, not String	ArgumentError en runtime
Asignación de tipo incorrecto	x : Int32 = "hello"	Error en compilación: Error: expected type to be Int32, not String	No aplica (Ruby no tiene anotaciones de tipo)

Crystal es capaz de inferir los tipos de variables y argumentos de manera inteligente, eliminando la necesidad de declaraciones verbosas mientras mantiene la seguridad del tipado estático:

```
1  x = 1
2  y = 2
```

En este código, Crystal deduce automáticamente que tanto x como y son números enteros sin necesidad de declararlo explícitamente. Esta capacidad de inferencia se extiende a estructuras de datos más complejas y llamadas a funciones, permitiendo escribir código conciso y legible sin sacrificar la seguridad que proporciona el sistema de tipos.

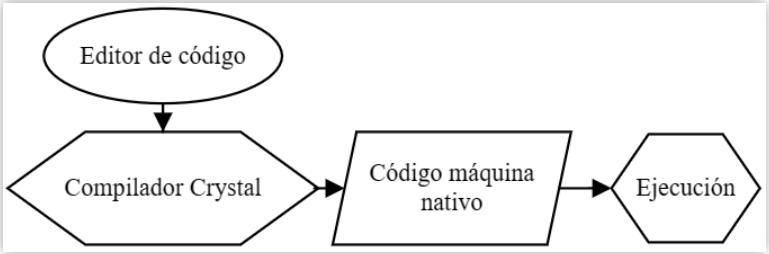


Gráfico 7 - Proceso de compilación en Crystal.
(Fuente: Propia)

Crystal compila el código a instrucciones de máquina nativas para la plataforma destino, lo que significa que no requiere una máquina virtual para ejecutarse. Este enfoque de compilación directa a código nativo es una de las características más destacadas del lenguaje, ya que permite obtener un rendimiento óptimo al eliminar la sobrecarga asociada con los entornos de ejecución interpretados o las máquinas virtuales.

El compilador de Crystal realiza un análisis exhaustivo del código fuente y genera instrucciones específicas para la arquitectura objetivo, aprovechando al máximo las características del hardware subyacente

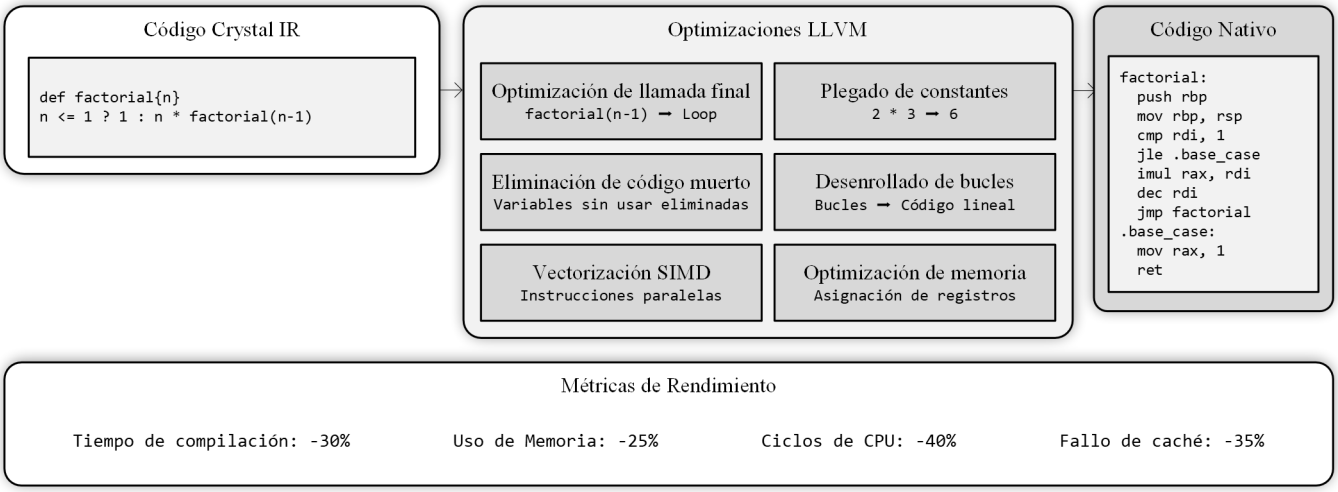


Gráfico 8 - Optimizaciones y Generación de Código Nativa en Crystal.
(Fuente: Propia)

Veamos cómo se traduce un simple programa a código ensamblador, lo que nos permitirá entender mejor el proceso de compilación y las optimizaciones que Crystal realiza.

Este ejemplo ilustra cómo incluso una instrucción aparentemente simple como 'puts' se traduce a una serie de instrucciones de bajo nivel que interactúan directamente con el hardware. El código ensamblador resultante muestra la eficiencia y precisión con la que Crystal maneja las operaciones básicas:

```
1   puts "Hello world"
2   .section .data
3   .asciz "Hello world"
4   .text
5   .global _main
6   _main:
7       push %rbp
8       mov %rsp, %rbp
9       sub $0x10, %rsp
10      lea .Lstr, %rdi
11      call puts@plt
12      movb $0, %al
13      leave
14      ret
```

Este proceso de compilación a código nativo resulta en un rendimiento excepcional, comparable al de lenguajes como C o Go. El compilador puede realizar optimizaciones agresivas al tener información completa sobre los tipos y el flujo del programa, generando código que aprovecha al máximo las capacidades del hardware subyacente.

Además del chequeo estático en compilación, Crystal hace validaciones en ejecución para detectar bugs que no pueden ser identificados durante la compilación:

```
x = 1
x.size
```

Este ejemplo ilustra una situación donde el tipo de la variable es conocido (x es un entero), pero estamos intentando llamar a un método que no existe para ese tipo. Crystal detectará este error en tiempo de ejecución y proporcionará un mensaje claro sobre el problema.

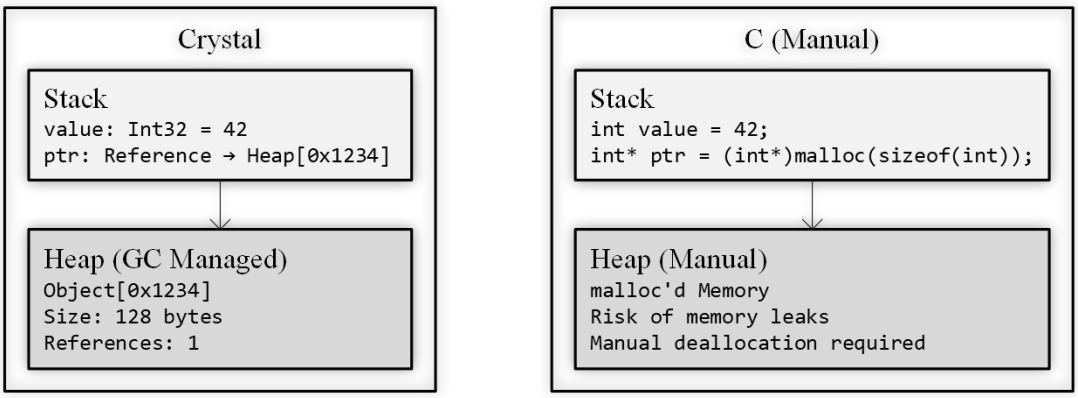


Gráfico 9 - Comparativa de Gestión de Memoria: Crystal vs C.
(Fuente: Propia)

Estas verificaciones adicionales son especialmente útiles cuando trabajamos con datos externos o situaciones donde el compilador no puede determinar estáticamente si una operación será válida. A diferencia de otros lenguajes compilados que podrían producir comportamientos indefinidos en estas situaciones, Crystal mantiene la seguridad del programa mediante estas comprobaciones dinámicas.

Crystal implementa un sofisticado sistema de concurrencia basado en fibras y canales, proporcionando herramientas de bajo nivel para manejar operaciones paralelas de manera segura y eficiente:

```
channel = Channel(Int32).new
spawn do
  5.times do |x|
    puts "Produciendo #{x}"
    channel.send(x)
  end
end
spawn do
  while num = channel.receive?
    puts "Recibido #{num}"
  end
end
```

Este ejemplo demuestra cómo Crystal permite la comunicación entre diferentes partes de un programa que se ejecutan concurrentemente. Las fibras son unidades de ejecución extremadamente livianas que el lenguaje puede crear y gestionar de manera eficiente. Los canales proporcionan un mecanismo seguro para que estas fibras intercambien información, evitando problemas comunes en la programación concurrente como las condiciones de carrera o los deadlocks. Todo esto se logra sin necesidad de depender de librerías externas, ya que forma parte del núcleo del lenguaje.

Crystal también destaca por incluir un recolector de basura sofisticado, eliminando la necesidad de gestionar manualmente la memoria como se hace en lenguajes de más bajo nivel:

```
1  arr = [1, 2, 3]
```

En este ejemplo, el programador no necesita preocuparse por liberar la memoria ocupada por el array cuando ya no se utilice. El recolector de basura de Crystal se encarga automáticamente de identificar qué objetos ya no son accesibles y liberar sus recursos, previniendo fugas de memoria mientras mantiene un rendimiento óptimo. Este sistema es particularmente eficiente ya que está diseñado específicamente para trabajar en armonía con el modelo de concurrencia del lenguaje.

La documentación oficial de Crystal es otro punto fuerte del lenguaje, ofreciendo recursos completos y bien organizados que cubren tanto aspectos básicos como avanzados:

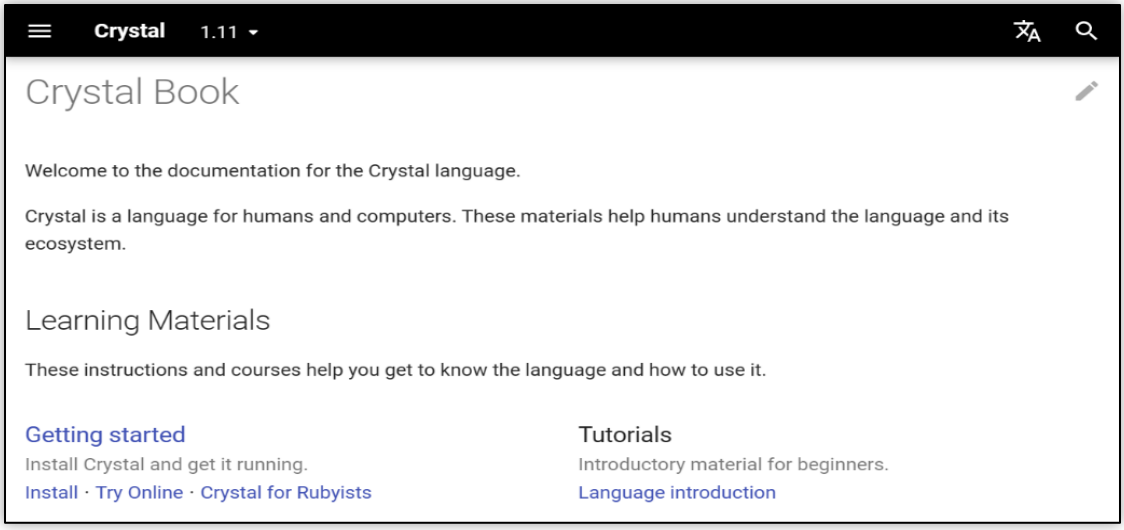


Gráfico 10 – Captura de pantalla de la documentación de Crystal.
(Fuente: Documentación Oficial de Crystal)

La documentación no solo describe la sintaxis y funcionalidades del lenguaje, sino que también incluye ejemplos prácticos, guías de mejores prácticas y explicaciones detalladas de los conceptos más complejos. Esta riqueza de recursos facilita enormemente el aprendizaje y la adopción del lenguaje, especialmente para desarrolladores que provienen de otros entornos de programación.

El lenguaje viene equipado con herramientas integradas que mejoran significativamente la experiencia de desarrollo. Por ejemplo, para mantener un estilo de código consistente, Crystal proporciona un formateador automático:

```
1  crystal tool format
```

Esta herramienta analiza y reformatea automáticamente el código fuente según las convenciones establecidas por la comunidad, asegurando consistencia en proyectos grandes y facilitando la colaboración entre desarrolladores. Además, Crystal ofrece utilidades para analizar y visualizar las dependencias del proyecto:

```
1  crystal deps
```

Como se ilustra en el Gráfico [N], Crystal proporciona un conjunto completo de herramientas de desarrollo integradas que trabajan en armonía para facilitar el proceso de desarrollo, desde el formateo del código hasta la compilación final del ejecutable.

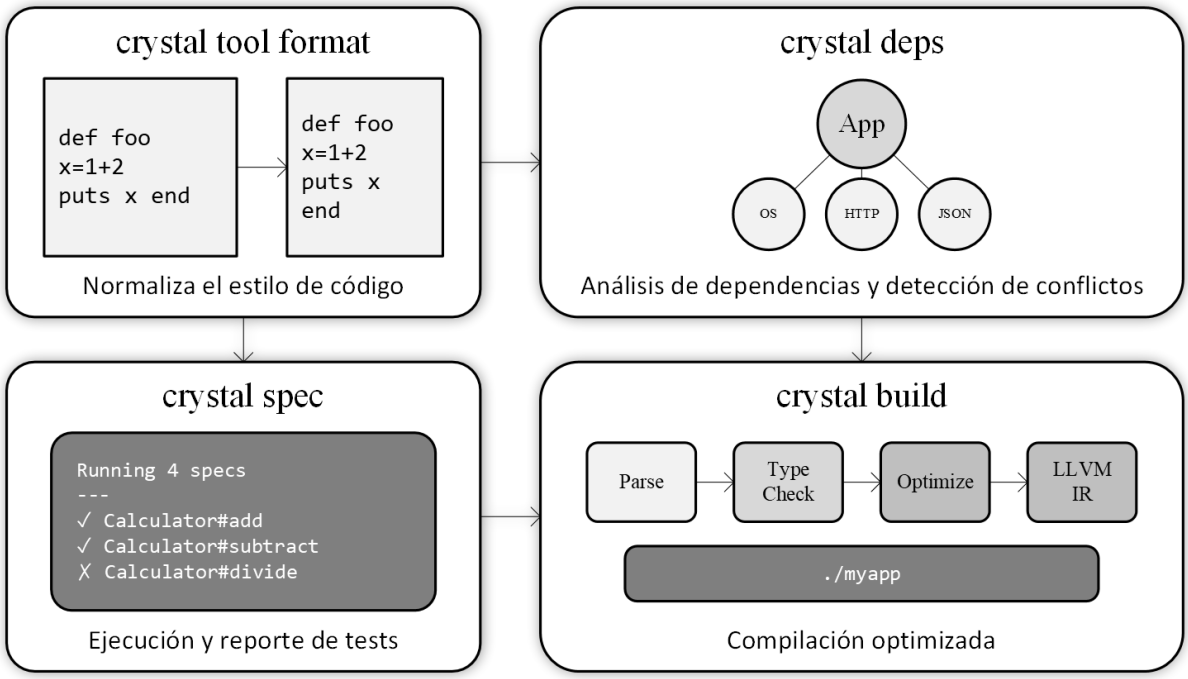


Gráfico 11 - Ecosistema integrado de herramientas de desarrollo en Crystal: formateo, dependencias, testing y compilación. (Fuente: Propia)

Esta herramienta genera un análisis detallado de las dependencias del proyecto, ayudando a identificar posibles problemas de versiones incompatibles o dependencias circulares antes de que causen problemas en producción. Todas estas herramientas integradas no solo agilizan el trabajo diario del desarrollador, sino que también promueven buenas prácticas de programación y mantenimiento de código.

Como hemos visto a través de estos ejemplos y explicaciones, Crystal ofrece un conjunto completo y bien pensado de características técnicas que se complementan entre sí para proporcionar una experiencia de desarrollo superior. La combinación de rendimiento, seguridad de tipos, facilidad de uso y herramientas integradas hace que Crystal sea una opción muy atractiva para una amplia gama de proyectos de software modernos.

1.2. Instalación, configuración y primeros pasos

Instalar Crystal en tu computadora es bastante sencillo y directo, aunque requiere tener en cuenta algunos aspectos importantes antes de comenzar. Crystal es compatible con los principales sistemas operativos como Linux, macOS y Windows, ofreciendo diferentes métodos de instalación según la plataforma y las necesidades específicas del desarrollador.

En sistemas basados en Unix como Linux y macOS, la forma recomendada de instalar Crystal es a través de los gestores de paquetes nativos del sistema. Estos gestores no solo facilitan la instalación inicial, sino que también manejan automáticamente las dependencias necesarias y simplifican las futuras actualizaciones. Los gestores más comunes son `apt` en Debian/Ubuntu, `yum` en CentOS/Fedora, `pacman` en Arch Linux, `brew` en macOS y `apk` en Alpine Linux. Esta diversidad de opciones asegura que prácticamente cualquier usuario de sistemas Unix pueda instalar Crystal de manera eficiente.

Gráfico 12 – Captura de pantalla de parte de las instrucciones de instalación de Crystal en Ubuntu. (Fuente: https://crystal-lang.org/install/on_ubuntu/)

Por ejemplo, en Ubuntu y Debian, el proceso de instalación es particularmente sencillo. Podemos instalar Crystal ejecutando un único comando:

```
1  sudo apt install crystal
```

Este comando no solo instalará la última versión estable de Crystal disponible en los repositorios oficiales, sino que también configurará automáticamente todas las dependencias necesarias para su funcionamiento. El sistema de gestión de paquetes se encargará de resolver cualquier conflicto de dependencias y asegurará que todos los componentes necesarios estén correctamente instalados.

Para desarrolladores que necesitan características más recientes o desean contribuir al desarrollo de Crystal, existe la opción de compilar e instalar desde el código fuente. Este método, aunque más complejo, ofrece mayor control sobre el proceso de instalación y acceso a las últimas características en desarrollo:

```
1  git clone https://github.com/crystal-lang/crystal
2  cd crystal/src
3  make
4  sudo make install
```

Este proceso de compilación desde el código fuente puede ser útil si queremos la versión de desarrollo más reciente con las últimas características y correcciones de errores, o si necesitamos personalizar la instalación según nuestros requerimientos específicos.

En Windows, el proceso de instalación es diferente pero igualmente accesible. La forma más directa es utilizar el instalador MSI oficial, que podemos descargar desde la página oficial de Crystal. Este instalador ha sido diseñado específicamente para proporcionar una experiencia familiar para los usuarios de Windows, automatizando gran parte del proceso de configuración.

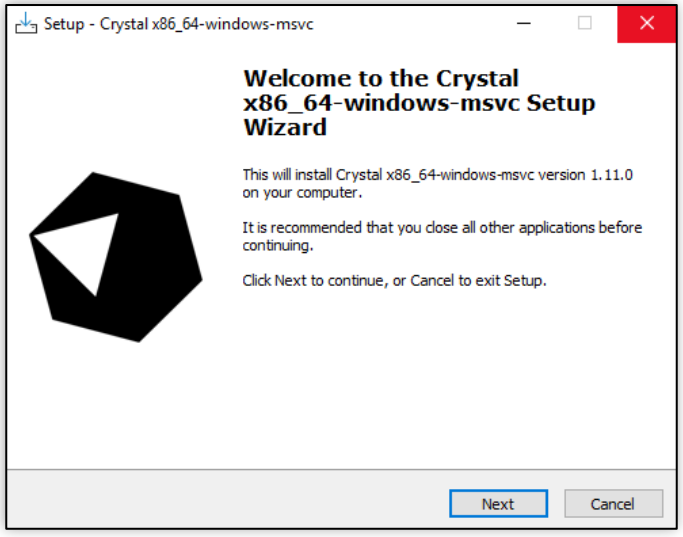


Gráfico 13 – Instalador de Crystal para Windows.
(Fuente: Propia)

Al igual que en los sistemas Unix, los usuarios de Windows también tienen la opción de compilar Crystal desde el código fuente. Sin embargo, este proceso requiere la instalación previa de varias dependencias esenciales. Cada una de estas dependencias cumple un papel específico en el proceso de desarrollo con Crystal:

Dependencia	Descripción	Obligatorio
Git	Sistema de control de versiones necesario para clonar el código fuente de Crystal desde el repositorio oficial en GitHub	Sí
Mingw-w64	Compilador GNU para Windows, incluye gcc y otras herramientas requeridas para compilar Crystal y generar binarios nativos	Sí
OpenSSL	Biblioteca esencial para agregar soporte de comunicaciones seguras SSL/TLS en aplicaciones Crystal	Sí
LibXML2	Biblioteca para analizar archivos XML, requerida para soporte XML en Crystal	Sí
LibYAML	Biblioteca para analizar archivos YAML, permite soporte de YAML en Crystal	Sí

Una vez instaladas todas las dependencias necesarias, el proceso de compilación en Windows sigue una secuencia específica de comandos. Primero, obtenemos el código fuente del repositorio oficial:

```
1 git clone https://github.com/crystal-lang/crystal
```

A continuación, nos movemos al directorio correcto y ejecutamos el script de compilación específico para Windows:

```
1 cd crystal\src
2 make.bat
```

Finalmente, ejecutamos el instalador que configurará Crystal en nuestro sistema:

```
1 .\bin\crystal.exe tools\install.cr
```

Este proceso instalará Crystal en la ubicación estándar `C:\Program Files\Crystal`, siguiendo las convenciones de Windows para la instalación de software.

Independientemente del método de instalación elegido, es crucial verificar que Crystal se haya instalado correctamente. La forma más sencilla es consultar la versión instalada mediante el comando:

```
1  crystal -v
```

Este comando debería mostrar la información detallada de la versión de Crystal instalada:

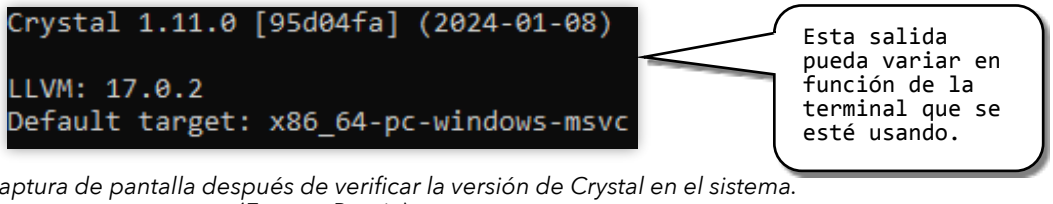


Gráfico 14 – Captura de pantalla después de verificar la versión de Crystal en el sistema. (Fuente: Propia)

Para asegurarnos de que todo funciona correctamente, es recomendable crear y ejecutar un programa simple. El clásico "Hola mundo" es perfecto para esta prueba inicial:

```
1  puts "Hola mundo"
```

Este código debe guardarse en un archivo con extensión `.cr`, por ejemplo `hola_mundo.cr`. Para ejecutarlo, utilizamos el comando `crystal` seguido del nombre del archivo. Crystal se encargará de compilar y ejecutar el programa en un solo paso:

```
1  crystal run hola_mundo.cr
```

Si todo está configurado correctamente, veremos el mensaje "Hola mundo" en nuestra pantalla, confirmando que Crystal está listo para usar.

Una vez que tenemos Crystal funcionando correctamente, el siguiente paso es entender cómo organizar nuestro código de manera eficiente. Aunque Crystal permite crear programas simples en archivos individuales, los proyectos reales generalmente requieren una estructura más organizada. Crystal proporciona una estructura de directorios estándar que facilita la organización del código y la gestión de dependencias.

La estructura básica de un proyecto Crystal se organiza de la siguiente manera:

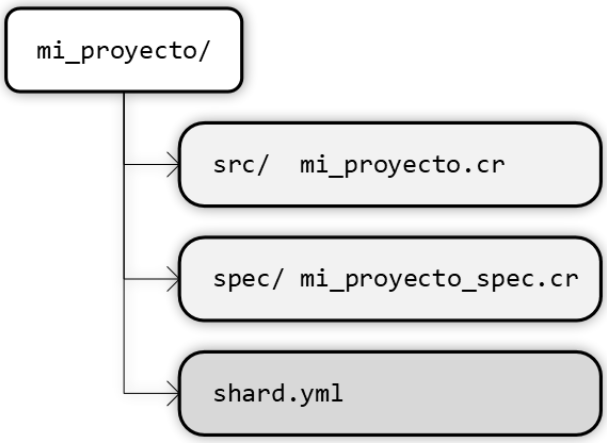


Gráfico 15 - Estructura de directorios estándar de un proyecto Crystal. (Fuente: Propia)

Esta estructura no es arbitraria; cada elemento cumple un propósito específico en el desarrollo de aplicaciones Crystal:

Directorio	Propósito	Descripción
src	Código fuente de la aplicación	Contiene el código fuente **.cr** de la aplicación, normalmente un archivo **proyecto.cr** con el punto de entrada
spec	Tests unitarios	Contiene los tests específicos **.cr** para probar el código fuente
shard.yml	Configuración del proyecto y dependencias	Define metadatos como nombre, versión, licencia y especifica dependencias de shards

El archivo `shard.yml` es particularmente importante, ya que actúa como el manifiesto del proyecto. Este archivo YAML define no solo la identidad del proyecto, sino también todas sus dependencias y configuraciones:

```
1  name: mi_proyecto
2  version: 0.1.0
3  dependencies:
4    repo:crystal-lang/crystal-mysql
5    version : ~> 0.9.0
6  development_dependencies:
7    ameba:
8      github: crystal-ameba/ameba
9      version: ~> 0.14.3
10 crystal: 1.0.0
11 license: MIT
```

En este ejemplo de archivo `shard.yml`, podemos ver cómo se especifican tanto las dependencias principales como las de desarrollo. La dependencia `crystal-mysql` es necesaria para la funcionalidad principal del proyecto, mientras que `ameba` es una herramienta de análisis de código que solo se utiliza durante el desarrollo. El uso del operador `~>` en las versiones (por ejemplo, `~> 0.9.0`) indica que aceptamos actualizaciones menores pero no mayores, lo que ayuda a mantener la estabilidad del proyecto.

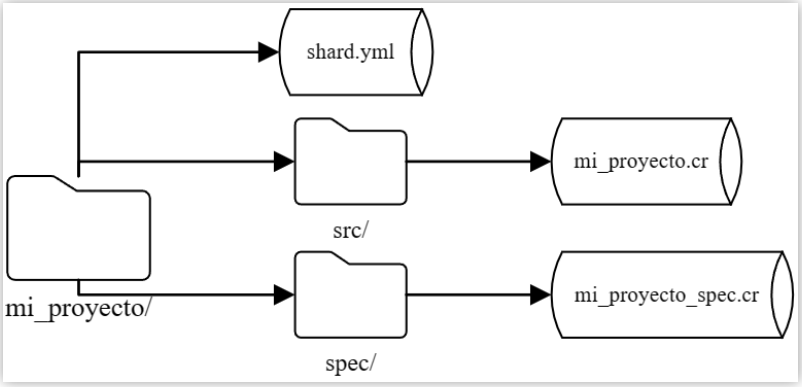
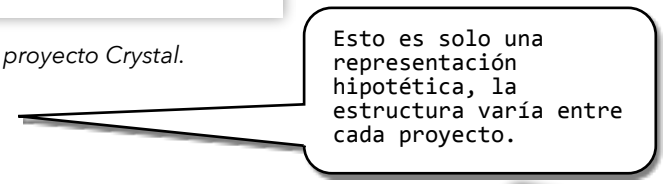


Gráfico 16 - Organización típica de un proyecto Crystal. (Fuente: Propia)



Para gestionar estas dependencias, Crystal utiliza una herramienta llamada `shards`, que funciona de manera similar a herramientas como npm para Node.js o pip para Python. El comando principal para instalar todas las dependencias definidas en el archivo `shard.yml` es:

```
1 shards install
```

Este comando realiza varias acciones importantes:

Acción	Descripción
Lectura de configuración	Lee el archivo shard.yml para identificar todas las dependencias necesarias
Descarga	Descarga las versiones específicas de cada dependencia
Resolución	Resuelve automáticamente las dependencias anidadas
Registro	Crea un archivo shard.lock que registra las versiones exactas instaladas

Una vez que tenemos nuestro proyecto configurado con todas sus dependencias, podemos proceder a la compilación. Crystal ofrece diferentes opciones de compilación según nuestras necesidades. Para crear un ejecutable independiente, utilizamos:

```
1 crystal build src/mi_proyecto.cr
```

Este comando compila nuestro código fuente y genera un archivo binario ejecutable. El binario resultante contiene todo lo necesario para ejecutar el programa, lo que facilita su distribución. Para ejecutar el programa compilado, simplemente usamos:

```
1 ./mi_proyecto
```

Durante el desarrollo, frecuentemente necesitamos compilar y ejecutar nuestro código de forma rápida para probar cambios. Crystal proporciona un atajo conveniente para esto:

```
1 crystal run src/mi_proyecto.cr
```

Este comando combina los pasos de compilación y ejecución en una sola operación, lo que resulta muy útil durante el ciclo de desarrollo y pruebas.

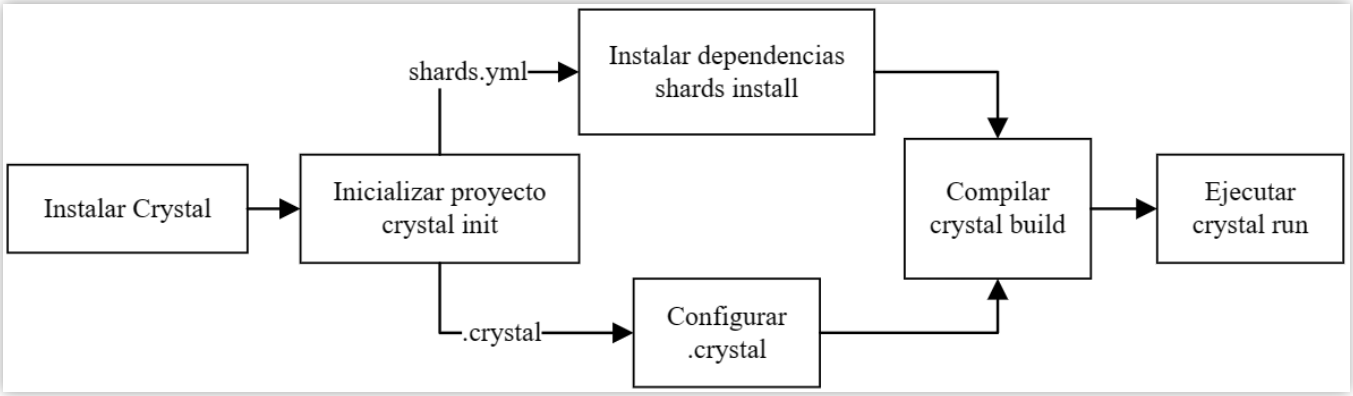


Gráfico 17 - Flujo de trabajo para inicializar un proyecto Crystal.
(Fuente: Propia)

Con Crystal instalado y configurado correctamente, y entendiendo la estructura básica de un proyecto, podemos comenzar a escribir nuestros primeros programas. Crystal organiza el código utilizando tres conceptos fundamentales: **tipos** (que incluyen clases y structs), **módulos** y **métodos**. Cada uno de estos elementos tiene un propósito específico en la organización y estructuración del código.

Para ilustrar estos conceptos, comencemos con el ejemplo más básico posible, el clásico programa "Hola Mundo":

```
1  # saluda.cr
2  puts "Hola Mundo"
```

Para ejecutar este programa:

```
1  crystal run saluda.cr
```

Aunque este ejemplo es extremadamente simple, demuestra varios conceptos importantes:

Concepto	Descripción
Extensión de archivo	La extensión .cr identifica a los archivos de código fuente Crystal
Comentarios	Los comentarios comienzan con el símbolo #
Funciones integradas	Crystal proporciona funciones integradas como puts para operaciones comunes
Sintaxis	La sintaxis es clara y concisa, similar a otros lenguajes modernos

Este primer programa marca el inicio de nuestro viaje con Crystal. A medida que avancemos en los siguientes capítulos, exploraremos características más avanzadas y veremos cómo Crystal combina la simplicidad sintáctica con un potente sistema de tipos y excelente rendimiento.

La simplicidad de este primer ejemplo no debe engañarnos: Crystal es un lenguaje completo y sofisticado que mantiene esta claridad sintáctica incluso en programas más complejos. Esta combinación de simplicidad y poder es una de las razones por las que Crystal se está volviendo cada vez más popular entre los desarrolladores.

Conclusión de este capítulo
En este primer capítulo realizamos una introducción completa a Crystal, cubriendo su historia, filosofía, ventajas técnicas y proceso de instalación en distintos sistemas operativos. Aprendimos los conceptos básicos de la sintaxis, con ejemplos prácticos de variables, tipos de datos, control de flujo y POO. Desarrollamos un sencillo programa "Hola Mundo". Vimos cómo estructurar un proyecto Crystal, gestionar dependencias con shards y compilar/ejecutar el código fuente.

Ejercicios y Preguntas Para Resolver

Para consolidar los conocimientos adquiridos en este capítulo, te presentamos una serie de ejercicios y preguntas organizados por nivel de dificultad. Cada ejercicio incluye el tiempo estimado de resolución y los conceptos principales que evalúa.

(★☆☆) Ejercicios Básicos

Tiempo estimado: 5-10 minutos por ejercicio

#	Ejercicio	Conceptos Evaluados	Tiempo
1	Instala Crystal en tu sistema operativo si aún no lo has hecho. Verifica que la instalación funcionó correctamente imprimiendo la versión con crystal -v.	• Instalación • Línea de comandos	5 min
2	Crea un nuevo proyecto Crystal con la estructura básica de directorios src y spec. Recuerda incluir el archivo shard.yml.	• Estructura de proyectos • Gestión de dependencias	10 min
3	Escribe un programa Crystal que imprima tu nombre usando puts. Compílalo y ejecútalo.	• Sintaxis básica • Compilación	5 min

(★★☆) Ejercicios Intermedios

Tiempo estimado: 10-15 minutos por ejercicio

#	Ejercicio	Conceptos Evaluados	Tiempo
4	Agrega una dependencia hipotética llamada mi_shard a tu proyecto en el shard.yml. Instálala con shards.	• Gestión de dependencias • Versionado	10 min
5	Explica brevemente con tus palabras qué es la inferencia de tipos en Crystal y qué beneficios proporciona.	• Sistema de tipos • Inferencia de tipos	15 min
6	¿Cuáles son las diferencias principales entre la filosofía de diseño de Crystal y lenguajes como Ruby, Python y Javascript?	• Comparación de lenguajes • Filosofía de diseño	15 min

(★★★) Ejercicios Avanzados

Tiempo estimado: 15-30 minutos por ejercicio

#	Ejercicio	Conceptos Evaluados	Tiempo
7	¿Por qué decir que Crystal tiene tipado estático y fuerte? ¿Qué ventajas implica sobre tipado dinámico y débil?	• Sistema de tipos • Comparación de paradigmas	20 min
8	Investiga y explica para qué sirve el comando crystal tool format y en qué casos lo usarías.	• Herramientas de desarrollo • Estilo de código	15 min
9	Describe al menos 3 casos de uso o tipos de aplicaciones ideales para desarrollar con Crystal.	• Aplicaciones prácticas • Escenarios de uso	20 min
10	Visita la documentación oficial de Crystal y encuentra cómo generar un diagrama de dependencias de tu proyecto. Prueba usar esto en tu proyecto de ejemplo.	• Documentación • Análisis de dependencias	30 min

Pistas y Ayudas

Para los ejercicios más complejos, aquí tienes algunas pistas que pueden orientarte:

Ejercicio	Conceptos Clave	Pistas para Resolución
7	Tipado Estático y Fuerte	• Analiza la diferencia entre errores en tiempo de compilación vs ejecución • Evalúa el impacto en rendimiento y seguridad del código • Compara con ejemplos de errores en lenguajes dinámicos
9	Casos de Uso	• Considera escenarios que requieran alto rendimiento • Identifica situaciones donde la seguridad de tipos sea crucial • Analiza casos donde la sintaxis similar a Ruby sea una ventaja
10	Documentación y Herramientas	• Revisa la sección de herramientas en docs.crystal-lang.org • Explora las opciones de crystal –help • Busca ejemplos de diagramas de dependencias en proyectos reales

Tabla de Autoevaluación

Usa esta tabla para evaluar tu comprensión de los conceptos clave del capítulo:

Concepto	¿Lo domino?	Ejercicios Relacionados
Instalación y configuración	☐	1, 2
Gestión de dependencias	☐	2, 4, 10
Sistema de tipos	☐	5, 7
Filosofía y diseño	☐	6, 9
Herramientas de desarrollo	☐	8, 10

Reto adicional

Intenta crear un pequeño proyecto que combine varios de los conceptos vistos en este capítulo. Por ejemplo, una aplicación de línea de comandos que procese datos usando tipos personalizados y dependencias externas.

Capítulo 2: Fundamentos del lenguaje

¿Qué se verá en este capítulo?
Este capítulo tiene como propósito presentar los conceptos más fundamentales del lenguaje de programación Crystal. Aprender un nuevo lenguaje es como aprender un idioma extranjero: debemos familiarizarnos primero con su alfabeto, estructura gramatical y vocabulario básico. Del mismo modo, en este capítulo estudiaremos la base sobre la cual se construye cualquier programa Crystal: su sintaxis, tipos de datos, variables, operadores, flujo de control, funciones, clases y la organización de código en módulos. Conocer estos elementos centrales de Crystal es esencial antes de pasar a características más avanzadas o aplicaciones completas. Saber cómo escribir expresiones, asignar variables, controlar el flujo, encapsular código y modelar objetos sentará las bases para desarrollar eficientemente en este lenguaje.

Antes de profundizar en los detalles específicos del lenguaje, es importante entender cómo Crystal convierte nuestro código fuente en un programa ejecutable. Crystal utiliza un proceso de compilación que transforma nuestro código de alto nivel en código de máquina nativo. Este enfoque de compilación a diferencia de la interpretación permite optimizar el rendimiento de nuestros programas al generar código binario altamente optimizado para la arquitectura específica donde se ejecutará.

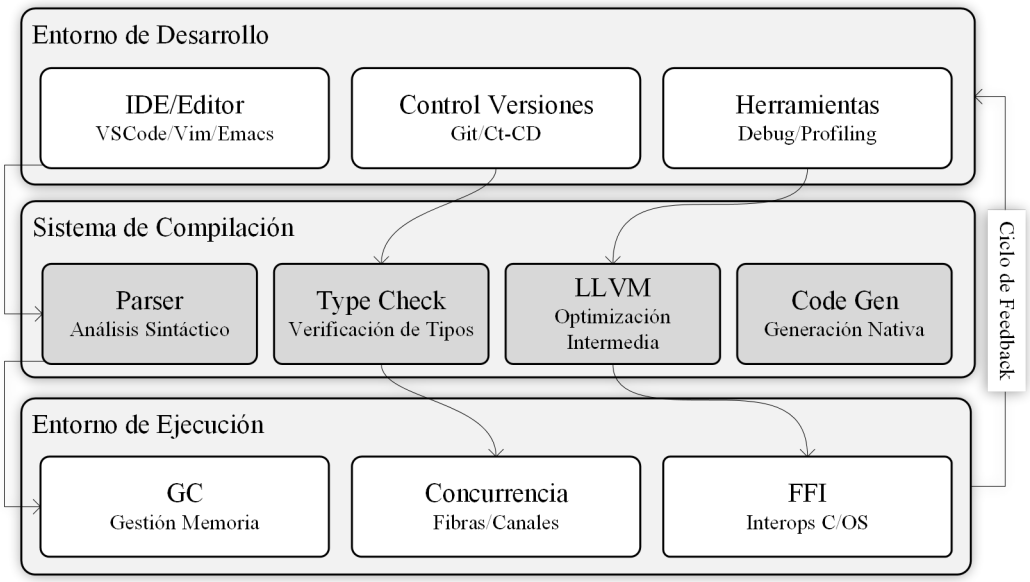


Gráfico 18 - Arquitectura y Flujo de Trabajo del Ecosistema Crystal.
(Fuente: Propia)

El siguiente diagrama ilustra el proceso de compilación de Crystal y las principales ventajas de rendimiento que esto proporciona. En primer lugar, el compilador de Crystal analiza nuestro código fuente y lo traduce a un formato intermedio conocido como LLVM IR (Lenguaje Intermedio de la Máquina Virtual de Bajo Nivel). Desde este formato abstracto, el compilador puede aplicar diversas optimizaciones y finalmente generar código máquina nativo altamente eficiente para la plataforma de destino.

Gráfico 19 - Proceso de compilación y beneficios de rendimiento en Crystal
(Fuente: Propia)

Como podemos observar, Crystal combina la elegancia sintáctica y expresividad de los lenguajes de alto nivel, con el rendimiento y eficiencia de la compilación nativa. Este proceso de compilación garantiza que nuestros programas Crystal sean tanto seguros como extremadamente performantes, ya que se benefician de las optimizaciones realizadas por el compilador y la ejecución directa en el hardware sin necesidad de una capa de virtualización o interpretación adicional.

2.1 Sintaxis Básica

Crystal es un lenguaje de programación elegante y expresivo que hereda mucha sintaxis de Ruby, pero con tipado estático y compilación nativa para un alto rendimiento. Comenzaremos con uno de los primeros elementos de Crystal que estudiaremos: Los comentarios.

Los comentarios en Crystal sirven para agregar documentación y notas en el código sin que afecten la ejecución del programa. Hay dos formas de escribir comentarios:

```
1  # Esto es un comentario de una línea
2  =begin
3  Esto es un comentario
4  de múltiples líneas
5  =end
```

Los comentarios de una línea comienzan con el caracter #. Los comentarios multilínea se delimitan con =begin y =end.

Lenguaje	Comentarios de una línea	Comentarios multilínea
Crystal	#	=begin Comentario =end
Ruby	#	=begin Comentario =end
Python	#	'''Comentario''' """Comentario"""
Java	//	/* Comentario */
JavaScript	//	/* Comentario */

Los comentarios son muy útiles para documentar el código y explicar partes complejas del programa. Deben usarse con moderación, el código debe ser lo suficientemente claro como para que no se requieran muchos comentarios.

Sin embargo, también es importante considerar la importancia dentro del mundo real que pueden llegar a tener los comentarios, lo cual en algunas ocasiones puede pasar de largo o realmente no le damos del todo la importancia que merece.

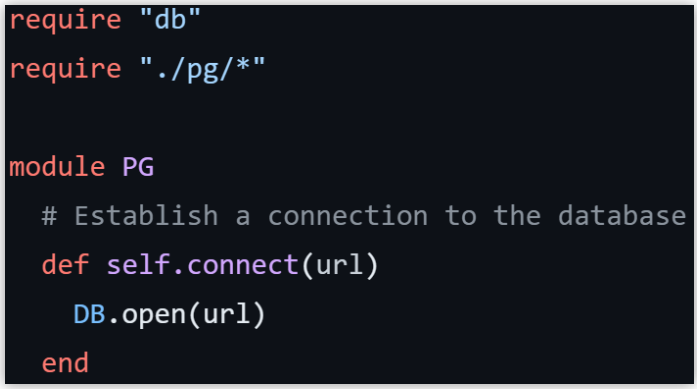


Gráfico 20 – Ejemplo visual del uso real de un comentario en un proyecto real.
(Fuente: https://github.com/will/crystal-pg)

Los comentarios tienen muchos usos importantes más allá de simplemente documentar el código. Se pueden aprovechar de diversas maneras para mejorar la legibilidad, mantenibilidad y organización del código. Uno de los usos más comunes de los comentarios es explicar detalladamente qué hace una función o método, cuáles son sus parámetros y valores de retorno. Por ejemplo:

```
# Calcula el índice de masa corporal (IMC) de una persona
# a partir de los parámetros de peso y altura.
#
# El IMC es un importante indicador de la relación entre
# el peso y la altura que se utiliza para determinar
# posibles riesgos para la salud.
#
# Parámetros:
# - peso (Float): peso de la persona en kg
# - altura (Float): altura de la persona en metros
#
# Retorna:
# - imc (Float): índice de masa corporal calculado
def calcular_imc(peso, altura)
    peso / (altura ** 2)
end
```

Este nivel de documentación es especialmente útil cuando se trabaja en equipo, permite entender rápidamente cómo utilizar la función sin necesidad de leer y entender los detalles de su implementación.

Otro uso muy valioso es explicar fragmentos de código particularmente complejos o que no son intuitivos a primera vista.

```
# Este algoritmo de búsqueda parece complicado debido a
# la manera en que la API retorna y pagina los resultados.
#
# Utilizamos un bucle while para pedir página tras página
# hasta que no haya más resultados. Dentro del bucle iteramos
# cada elemento para procesarlo.
while resultados.pagina_siguiente?
    resultados = API.obtener_resultados(resultados.pagina_siguiente)
    resultados.each do |resultado|
      # procesar cada resultado
    end
end
```

De nuevo, en código que será leído por otros, vale la pena invertir tiempo en explicar la lógica detrás de construcciones complejas.

Los comentarios permiten deshabilitar rápidamente ciertas partes del código sin necesidad de eliminarlas. Esto es útil para depurar o probar diferentes implementaciones.

```
# Este código estaba causando un error,
# print("Hola")
# print("Mundo")
```

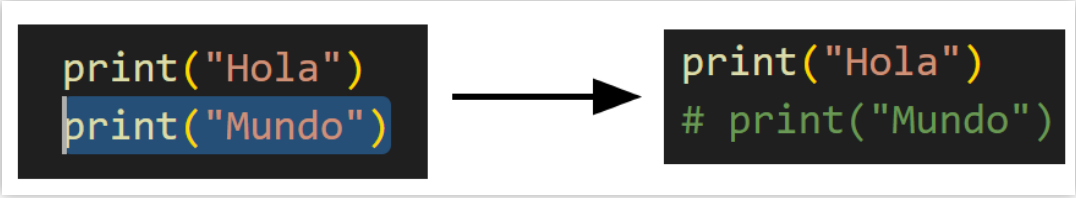


Gráfico 21 – Uso de los comentarios para inhabilitar temporalmente líneas de código. (Fuente: Propia)

También se pueden utilizar comentarios para documentar y separar distintas partes o secciones del código:

```
# Programa: Contador de palabras en un archivo de texto
# ------------------------------------
# Funciones de procesamiento de archivos
# ------------------------------------
# Lee el archivo y retorna las líneas
def leer_archivo(ruta)
    # implementation...
end
# Limpia caracteres especiales
def limpiar_texto(linea)
    # implementation...
```

```
end
# Cuenta palabras en una cadena de texto
def contar_palabras(texto)
    # implementation...
end
# Imprime las palabras contadas con formato
def imprimir_resultados(total)
    puts "Total de palabras: #{total}"
end
# Programa principal
ruta = ARGV[0]
lineas = leer_archivo(ruta)
texto = lineas.map { |linea| limpiar_texto(linea) }.join(" ")
total = contar_palabras(texto)
```

Finalmente, los comentarios son ideales para dejarse notas personales o recordatorios sobre tareas pendientes:

```
# TODO: validar entrada de usuario
username = gets
# FIXME: método obsoleto, necesita refactorizarse
procesar_datos(datos)
```

Como podemos ver, hay muchas formas de aprovechar los comentarios para documentar, organizar y mantener el código. Un buen uso de ellos puede mejorar enormemente la legibilidad y calidad del código en el largo plazo, especialmente en proyectos grandes o con múltiples colaboradores.

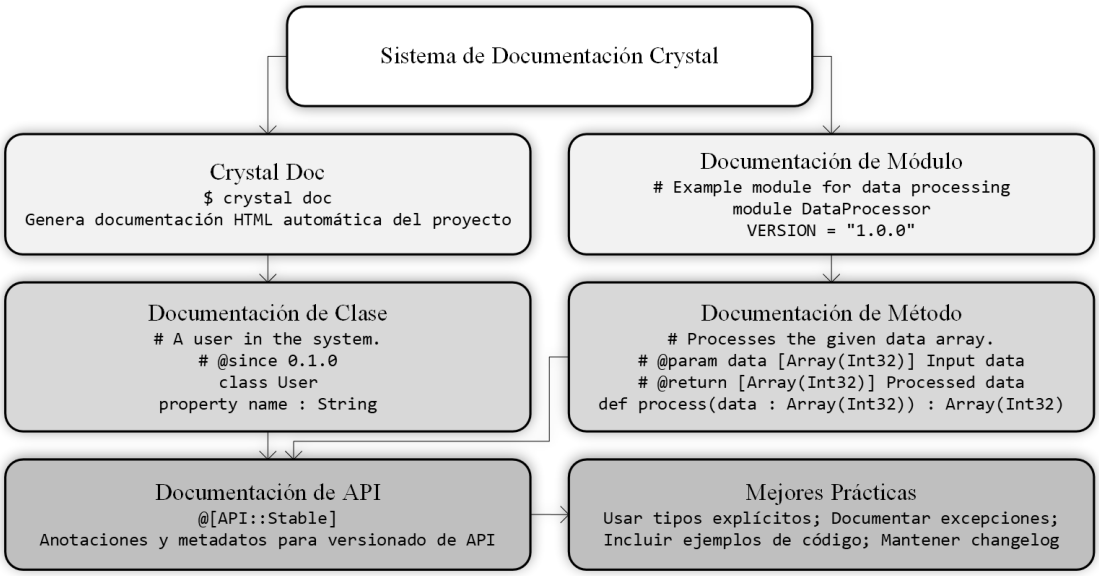


Gráfico 22 - Arquitectura de documentación y mejores prácticas en Crystal.
(Fuente: Propia)

Ahora, es momento de que veamos uno de los siguientes temas que son fundamentales en Crystal: Las expresiones y sentencias. Una expresión es una combinación de valores, variables, operadores y llamadas a métodos que se evalúan a un único valor. Por ejemplo:

```
2 + 3 # expresa el valor 5
"Hola " + "mundo" # expresa la cadena "Hola mundo"
Persona.new("Juan") # expresa una nueva instancia de Persona
```

Por otro lado, una sentencia es una unidad completa de ejecución que realiza alguna acción pero no evalúa a un valor. Algunos ejemplos son:

```
puts "Hola mundo" # imprime un mensaje
Persona.new("Juan") # crea un objeto Persona
```

Las sentencias en Crystal siempre terminan con un fin de línea o con un punto y coma (;). Se recomienda usar fin de línea ya que es más claro y legible en la mayoría de los proyectos.

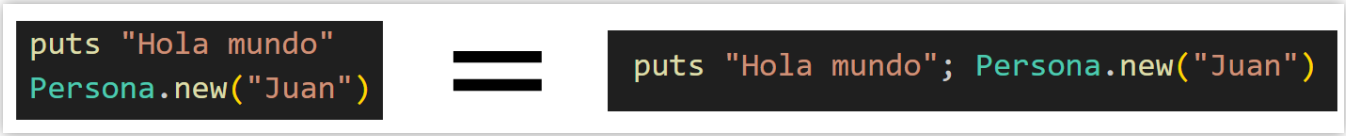


Gráfico 23 – Equivalencia entre sentencias con punto y coma y fin de línea.
(Fuente: Propia)

Los identificadores son los nombres que asignamos a variables, métodos, clases, módulos, etc. en nuestros programas. En Crystal, los identificadores deben cumplir las siguientes reglas:

Regla	Descripción
Pueden contener letras, números y guión bajo (_)	Los identificadores pueden contener letras del alfabeto, dígitos numéricos de 0 a 9 y el carácter guión bajo (_)
Deben comenzar con una letra o guión bajo	Los identificadores deben empezar con una letra del alfabeto o con un guión bajo. No pueden comenzar con un número.
No pueden ser palabras reservadas del lenguaje	Los identificadores no pueden usar palabras que están reservadas en el lenguaje Crystal como if, while, class, etc.

Algunos ejemplos válidos de identificadores son:

```
nombre
Edad
Persona
calcular_salario
_SUMA
```

Las palabras reservadas son términos que tienen un significado especial en el lenguaje y no pueden usarse como identificadores.

Algunas palabras reservadas importantes en Crystal son:

Palabra reservada	Descripción	Ejemplo de uso
if	Condicional si/de lo contrario	if x > 5 puts "x es mayor a 5" end
while	Bucle while	while x < 10 x += 1 end

for	Bucle for	for i in [1, 2, 3] puts i end
class	Define una clase	class Person end
def	Define un método	def greeting puts "Hola!" end
end	Fin de bloque	if true puts "Verdadero" end
case	Sentencia case	case x when 1 then puts "Uno" end
when	Cláusula when en case	case x when 1 then puts "Uno" end
module	Define un módulo	module Math end
return	Devuelve valor de método	def add(x, y) return x + y end
macro	Define una macro	macro hello puts "Hola!" end

Intentar usar una palabra reservada como variable o método dará un error:

```
1  # ERROR! 'end' es palabra reservada
2  def end
3      # ...
4  end
```

Los errores no aparecen dentro del archivo de texto, si no que aparecerán en nuestra terminal al intentar ejecutar el archivo.

Es buena práctica dar nombres significativos y legibles a los identificadores en nuestros programas.

Identificadores Válidos	Identificadores Inválidos
nombre	1nombre
edad	nombre$
Persona	class
calcular_salario	end

Crystal es un lenguaje con tipado estático, por lo que todas las variables tienen un tipo de dato asociado que se comprueba en tiempo de compilación. Los principales tipos de datos primitivos en Crystal son:

Tipo de Dato	Descripción	Ejemplos
Int	Números enteros de 32 o 64 bits	1, 100, 20000
Float	Números con punto flotante de 64 bits	3.1416, 9.99, 1.0
Bool	Valores booleanos verdadero/falso	true, false
Char	Caracteres Unicode individuales	'a', 'Z', '∮∮'
String	Cadenas de texto	"Hola"

Estos tipos de datos fundamentales conforman la base sobre la cual Crystal construye su robusto sistema de tipos. Para comprender mejor cómo el compilador trabaja con estos tipos y cómo se realiza la inferencia automática de los mismos, observemos el siguiente diagrama que ilustra el proceso completo:

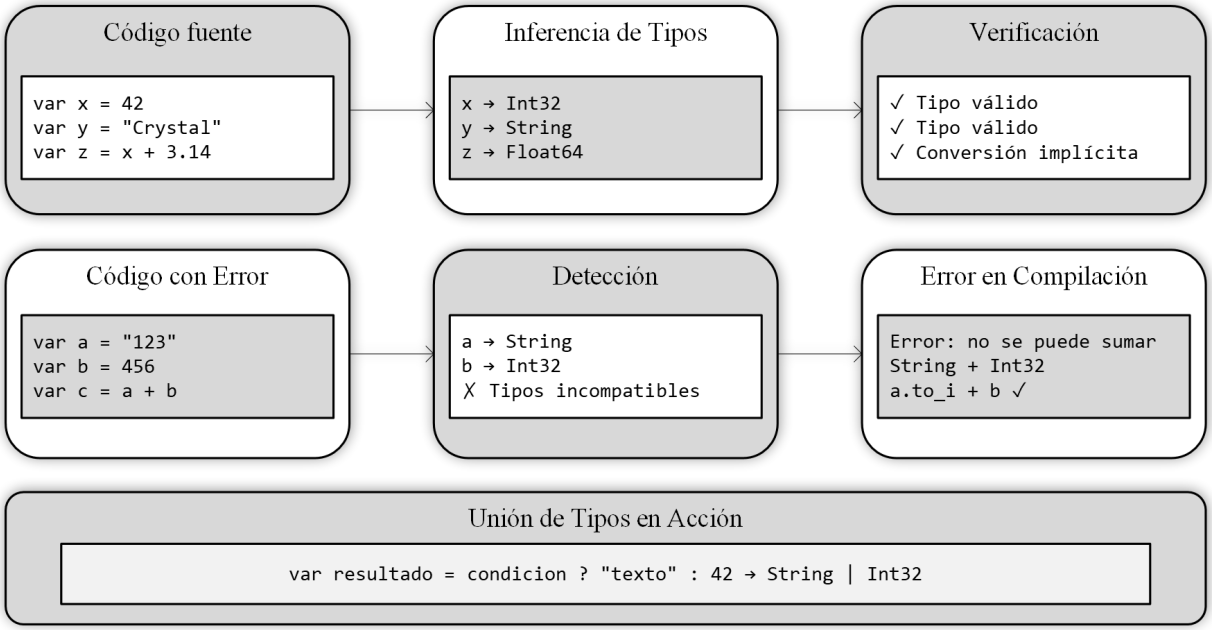


Gráfico 24 - Proceso de inferencia y verificación automática de tipos en Crystal: de código fuente a ejecución. (Fuente: Propia)

El tipo se especifica al declarar la variable:

```
1  entero = 100 # Int32
2  flotante = 3.5 # Float64
3  es_mayor = true # Bool
4  letra = 'A' # Char
5  mensaje = "Hola" # String
```

Crystal infiere automáticamente el tipo de datos más apropiado según el valor asignado, por lo que de forma predeterminada no debemos de preocuparnos por la asignación de tipos de datos, a excepción de que específicamente deseemos hacerlo.

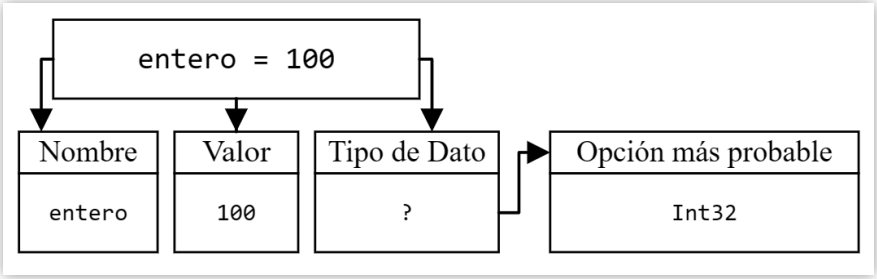


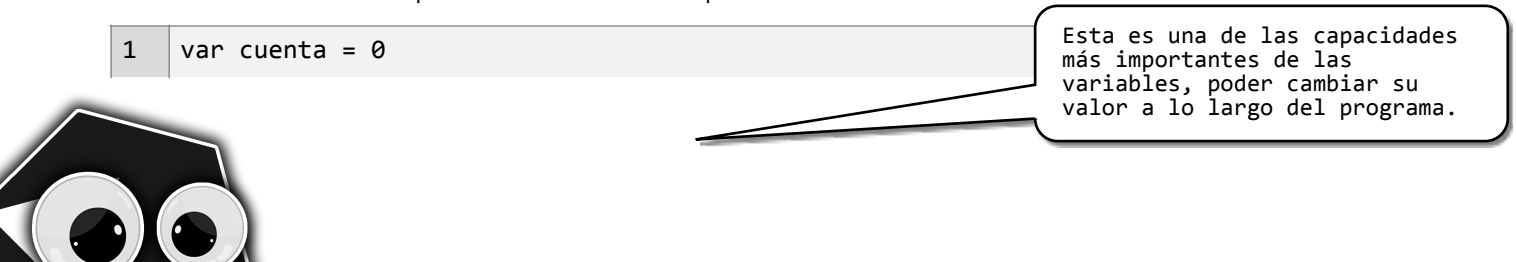
Gráfico 25 – Proceso de selección de tipo de datos automático en Crystal. (Fuente: Propia)

Las variables se usan para almacenar valores que pueden cambiar durante la ejecución del programa. Se declaran con la palabra clave `var`:

```
1  var nombre = "Juan"
2  var edad = 30
3  var pi = 3.1416
```

El valor de una variable puede cambiarse después de la declaración:

```
1  var cuenta = 0
```



```
2  cuenta = 10
3  cuenta = cuenta + 1 # ahora vale 11
```

Las constantes se usan para valores fijos que no cambian. Se declaran con la palabra `const`:

```
1  const PI = 3.14159
2  const USERNAME = "admin"
```

Las constantes deben inicializarse en la declaración y su valor no puede cambiarse después, por lo que se puede ver como la contraparte de una variable.

Una buena práctica es usar constantes en lugar de variables cuando sea posible, para hacer el código más legible y mantenible.

Las constantes también permiten que el compilador realice optimizaciones adicionales.

	Variables	Constantes
Propósito	Almacenar valores que pueden cambiar	Almacenar valores fijos que no cambian
Declaración	var nombre = valor	const NOMBRE = valor
Cambia después de declaración	Sí	No
Convención de nombres	nombreVariable	NOMBRE_CONSTANTE
Ejemplos	var edad = 30 edad = 31	const PI = 3.1416

Los operadores son símbolos especiales que realizan operaciones sobre valores y variables. Crystal proporciona una rica variedad de operadores para diferentes propósitos, desde cálculos matemáticos hasta manipulaciones a nivel de bits.

La precedencia de operadores determina el orden en que se evalúan las expresiones. Los operadores con mayor precedencia se evalúan antes que los de menor precedencia. Veamos los principales grupos de operadores:

Categoría	Operadores	Precedencia	Ejemplo	Resultado
Unarios	+, -, !	Más alta	!true	false
Multiplicativos	*, /, %	Alta	15 * 2	30
Aditivos	+, -	Media	10 + 5	15
Relacionales	<, >, <=, >=	Media-baja	5 < 10	true
Igualdad	==, !=	Baja	5 == 5	true
Lógicos	&&, ||	Más baja	true && false	false

Casos de uso comunes y errores frecuentes

Al trabajar con operadores en Crystal, es importante conocer tanto los patrones de uso más comunes como los errores que los desarrolladores suelen cometer. Esta sección explora situaciones prácticas que encontrarás frecuentemente en tu código, junto con recomendaciones para evitar problemas comunes. Analizaremos cada operador en contextos reales y aprenderemos a utilizarlos de manera efectiva y segura.

División y módulo

La división (/) y el módulo (%) son operaciones fundamentales en Crystal. La división devuelve el cociente de dos números, mientras que el módulo devuelve el resto de la división. Es importante entender cómo manejan los casos especiales y sus limitaciones.

```
# Uso correcto
cociente = 10 / 3    # => 3
resto = 10 % 3       # => 1
# Error común: división por cero
x = 5 / 0           # => DivisionByZeroError
```

Un error común al trabajar con estos operadores es intentar dividir por cero, lo que resultará en una excepción DivisionByZeroError. Siempre es recomendable validar el divisor antes de realizar la operación. Además, es importante tener en cuenta que la división entre enteros siempre devolverá un entero, truncando cualquier parte decimal.

Operadores de comparación

Los operadores de comparación nos permiten evaluar condiciones y tomar decisiones en nuestro código. Crystal proporciona operadores para comparar valores numéricos, cadenas y otros tipos de datos. Es crucial entender la diferencia entre el operador de asignación (=) y el operador de igualdad (==).

```
# Uso correcto
if edad >= 18 && edad <= 65
  puts "Edad laboral"
end
# Error común: confundir asignación (=) con igualdad (==)
if edad = 18        # Asigna 18 a edad
  puts "Error"
end
```

En el ejemplo anterior, el uso correcto muestra cómo verificar si una edad está dentro del rango laboral usando los operadores >= (mayor o igual que) y <= (menor o igual que). El error común muestra un problema sutil pero importante: usar el operador de asignación (=) en lugar del operador de igualdad (==) en una condición. Esto no solo asignará el valor, sino que la condición siempre será verdadera para valores no nulos, lo que puede llevar a bugs difíciles de detectar.

Buena práctica	Descripción	Ejemplo
Comparaciones explícitas	Usar siempre operadores de comparación claros	if edad == 18 en lugar de if edad = 18
Valores literales a la derecha	Colocar las constantes del lado derecho de la comparación	if edad >= 18 en lugar de if 18 <= edad

Uso de herramientas de análisis	Utilizar linters y analizadores estáticos	Configurar Ameba en el proyecto
Revisión de condiciones	Verificar cuidadosamente las estructuras de control	Realizar code reviews enfocadas en condiciones
Validación de divisores	Comprobar que no sea cero antes de dividir	divisor != 0 ? numero / divisor : 0

Operadores a nivel de bits

Los operadores bit a bit son herramientas fundamentales en la programación de bajo nivel que permiten manipular los bits individuales de valores numéricos. Estos operadores trabajan directamente con la representación binaria de los números, permitiendo realizar operaciones precisas a nivel de bit que son esenciales para optimización, manejo de flags y procesamiento de datos eficiente.

Cuando trabajamos con operadores bit a bit, es importante entender que cada número se representa internamente como una secuencia de bits (1s y 0s). Por ejemplo, el número decimal 5 se representa en binario como 0101. Cada operador bit a bit realiza una operación específica en estos bits individuales.

La siguiente tabla detalla cada operador bit a bit disponible en Crystal, junto con ejemplos prácticos de su uso. Los resultados se muestran tanto en formato decimal como en su representación binaria para mejor comprensión:

Operador	Descripción	Ejemplo	Resultado binario	Explicación
&	AND bit a bit	5 & 3	0101 & 0011 = 0001	Retorna 1 solo si ambos bits son 1
|	OR bit a bit	5 | 3	0101 | 0011 = 0111	Retorna 1 si al menos un bit es 1
^	XOR bit a bit	5 ^ 3	0101 ^ 0011 = 0110	Retorna 1 si los bits son diferentes
<<	Desplazamiento izquierdo	5 << 1	0101 << 1 = 1010	Mueve bits a la izquierda, multiplica por 2
>>	Desplazamiento derecho	5 >> 1	0101 >> 1 = 0010	Mueve bits a la derecha, divide por 2
~	NOT bit a bit	~5	~0101 = 1010	Invierte todos los bits

Cada uno de estos operadores tiene casos de uso específicos en la programación real. Por ejemplo, el operador AND (&) es comúnmente usado para verificar flags o permisos, mientras que el OR (|) se usa para combinarlos.

Ejemplos prácticos de operadores bit a bit

Veamos algunos ejemplos prácticos de cómo estos operadores se utilizan en situaciones reales. Uno de los casos de uso más comunes es la implementación de sistemas de permisos:

```
1  # Uso de máscaras de bits para permisos
2  LEER    = 0b0001  # 1 en binario
3  ESCRIBIR = 0b0010  # 2 en binario
4  EJECUTAR = 0b0100  # 4 en binario
5  # Combinamos permisos usando OR bit a bit
6  permisos = LEER | ESCRIBIR  # Resultado: 0011 en binario (3 en decimal)
7  # Verificamos permisos usando AND bit a bit
8  puede_leer = permisos & LEER == LEER  # Verdadero si tiene permiso de lectura
```


```
puede_ejecutar = permisos & EJECUTAR == EJECUTAR  # Falso, no tiene permiso de ejecución
# Alternancia de bits usando XOR
valor = 5          # 0101 en binario
valor ^= 0b1111    # Invierte los primeros 4 bits, resultado: 1010 (10 en decimal)
```

En este ejemplo, cada permiso se representa con un bit diferente, lo que nos permite almacenar múltiples permisos en un solo número. Esta técnica es extremadamente eficiente en términos de memoria y rendimiento.

Los operadores bit a bit son particularmente útiles en varios escenarios:

Escenario de uso	Descripción
Programación de sistemas	Desarrollo de controladores de dispositivos y software de bajo nivel
Optimización de memoria	Reducción del espacio de almacenamiento mediante empaquetado de datos
Procesamiento de señales	Manipulación y análisis de señales digitales
Implementación de flags	Manejo eficiente de estados y configuraciones
Compresión y encriptación	Algoritmos que requieren manipulación directa de bits

Nota: Aunque los operadores bit a bit son muy poderosos, deben usarse con precaución. Un error en la manipulación de bits puede llevar a resultados inesperados.

Mejores prácticas con operadores bit a bit

Para utilizar los operadores bit a bit de manera efectiva y segura, es importante seguir algunas prácticas recomendadas:

Práctica	Descripción	Impacto
Uso de paréntesis	Clarificar la precedencia en expresiones complejas	Mejora la legibilidad y previene errores
Simplicidad de expresiones	Evitar combinar demasiados operadores	Facilita el mantenimiento y debugging
Documentación	Explicar operaciones bit a bit complejas	Ayuda a otros desarrolladores a entender el código
Validación	Verificar condiciones antes de operaciones críticas	Previene errores en tiempo de ejecución
Uso de constantes	Emplear nombres descriptivos para máscaras de bits	Hace el código más auto-documentado

La siguiente tabla muestra ejemplos de buenas y malas prácticas al trabajar con operadores:

Práctica	No recomendado	Recomendado	Explicación
Precedencia	a + b * c	(a + b) * c	Los paréntesis hacen el código más legible

Complejidad	a && b || c && d	(a && b) || (c && d)	Agrupa operaciones lógicas relacionadas
División	x / y	y != 0 ? x / y : 0	Previene errores de división por cero
Bits	flags |= 0x0F	flags |= MASK_LOWER_NIBBLE	Usa constantes nombradas para claridad

Es importante recordar que, aunque los operadores bit a bit pueden hacer el código más eficiente, no debemos sacrificar la legibilidad y mantenibilidad del código por una optimización prematura. Use estos operadores cuando realmente sean necesarios y documéntelos adecuadamente.

2.2. Tipos de datos

Los tipos de datos en Crystal nos permiten representar diferentes tipos de información y valores en nuestros programas. Cada valor y variable en Crystal tiene asociado un tipo de dato que determina el conjunto de posibles valores que puede tomar y las operaciones que se pueden realizar con él.

Conocer los tipos de datos disponibles en el lenguaje y saber utilizarlos correctamente es fundamental para poder aprovechar al máximo las capacidades de Crystal. En esta sección veremos en detalle los diferentes tipos de datos incorporados, sus características, usos recomendados y ejemplos de declaración y utilización.

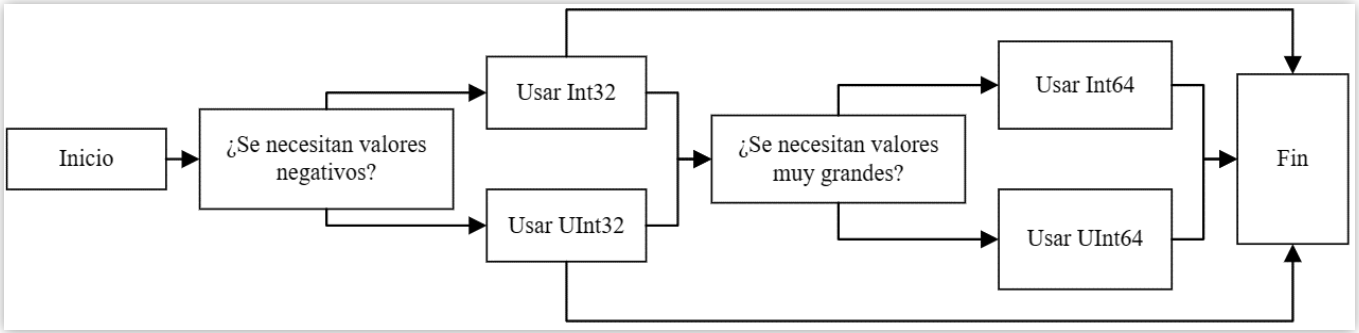


Gráfico 26 – Flujo de decisión para selección de tipo de dato.
(Fuente: Propia)

Los tipos básicos en Crystal incluyen los tipos numéricos enteros y de punto flotante, los valores booleanos, caracteres y cadenas de texto. Crystal proporciona varios tipos para representar valores numéricos enteros. Los más utilizados son:

Tipo de dato	Descripción	Ejemplo
Int32	Entero con signo de 32 bits. Rango: -2,147,483,648 a 2,147,483,647.	entero = 25 # Int32
UInt32	Entero sin signo de 32 bits. Rango: 0 a 4,294,967,295.	positivo = 124324 # UInt32
Int64	Entero con signo de 64 bits. Permite valores muy grandes.	muy_grande = 9223372036854775807
UInt64	Entero sin signo de 64 bits. Similar a Int64 pero sin signo.	enorme = 18446744073709551615

Los literales enteros sin sufijo son tratados por defecto como Int32. Podemos forzar otros tamaños con sufijos:

```
1  int64 = 2_i64
2  uint32 = 50_u32
```

Puntualmente, todas las variaciones de los números enteros se muestran a continuación:

Tipo	Tamaño (bits)	Signo	Rango
Int8	8	Sí	-128 a 127
UInt8	8	No	0 a 255
Int16	16	Sí	-32,768 a 32,767
UInt16	16	No	0 a 65,535
Int32	32	Sí	-2,147,483,648 a 2,147,483,647
UInt32	32	No	0 a 4,294,967,295
Int64	64	Sí	-9,223,372,036,854,775,808 a 9,223,372,036,854,775,807
UInt64	64	No	0 a 18,446,744,073,709,551,615

Los enteros tienen los operadores aritméticos estándar +, -, *, / y % (módulo). También se pueden comparar con <, <=, >, >=, == y !=. Para representar valores numéricos fraccionarios Crystal proporciona dos tipos principales:

Tipo	Ejemplo
Float32: número de punto flotante de 32 bits según el estándar IEEE 754. Ocupa 4	flotante = 1.5 # Float32
Float64: número de punto flotante de 64 bits según el estándar IEEE 754. Ocupa 8	double = 3.141592 # Float64

Ambos tipos permiten representar números con fracciones decimales, en notación científica y valores especiales como +inf, -inf y NaN (not a number). Proporcionan los operadores aritméticos estándar y comprobaciones de igualdad. Para comparaciones mayores o menores se debe tener cuidado con los errores de precisión.

El tipo bool representa un valor booleano que solo puede ser true o false. Ocupa 1 byte. Se utiliza para la lógica y el control de flujo en los programas.

```
1  correcto = true
2  error = false
```

El tipo Char representa un único carácter Unicode. Ocupa 4 bytes y permite almacenar cualquier carácter internacional. Se delimita con comillas simples:

```
1  letra = 'A'
2  simbolo = '∞'
```

Permite el acceso al código numérico subyacente:

```
1  'A'.ord # ==> 65
2  '∞'.ord # ==> 8734
```

El código numérico subyacente hace referencia al código Unicode que

También proporciona varias propiedades útiles para clasificación y tratamiento de caracteres.

El tipo String representa una cadena de caracteres. Se crea con literales delimitados por comillas dobles:

```
1  nombre = "Ada"
2  frase = "Hola Mundo!"
```

Internamente las cadenas son mutables y se codifican en UTF-8, permitiendo almacenar cadenas arbitrarias con caracteres internacionales de forma eficiente. Dispone de los operadores + y * para concatenación y repetición.

También permite acceder a caracteres individuales mediante indexación. Ofrece gran cantidad de métodos útiles para tratamiento y manipulación de cadenas. Crystal también incluye de forma nativa varios tipos de colecciones y contenedores de datos fundamentales.

Los arrays son contenedores de valores del mismo tipo con acceso directo mediante índices numéricos:

```
1  nums = [1, 2, 3]
2  nums[0] # ==> 1
```

Los arrays son uno de los tipos de datos más útiles. Veremos en detalle a estos mismos en secciones posteriores de este libro, ya que puede ser confuso para muchos.

Se declaran con corchetes, literales o con el constructor de Array.new. Son mutables. Admiten diversos métodos como push, pop, shift, append, insert, delete, etc. para modificarlos.

Los hashes (también llamados diccionarios) son colecciones de pares clave-valor muy eficientes para búsquedas:

```
1  colores = {"rojo" => 0xFF0000, "verde" => 0x00FF00}
2  colores["rojo"] # ==> 0xFF0000
```

Se declaran con llaves o con Hash(K, V).new. Las claves deben ser únicas y de un tipo inmutable como String, Symbol o Tuple. Permiten claves de cualquier tipo mediante Hash(K, V).new. Ofrecen métodos como set, get, delete, has_key?, merge y otros.

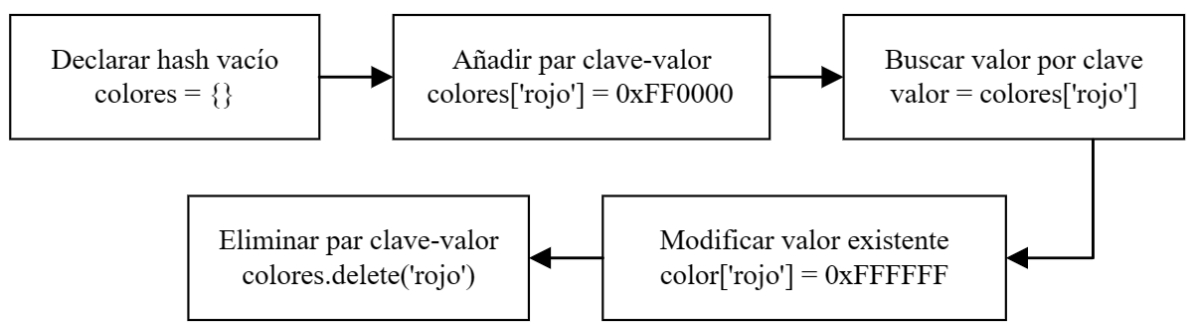


Gráfico 27 - Flujo de trabajo de un hash.
(Fuente: Propia)

Los rangos representan intervalos de valores entre un límite inferior y superior:

```
1  diez = 1..10
2  diez.includes?(5) # ==> true
```

Se construyen con dos puntos (..). El límite superior es inclusivo. Se pueden iterar elemento a elemento con each y son muy útiles en bucles e indexación. Los rangos excluyen el límite superior con tres puntos (...).

Los tuples agrupan valores de diferentes tipos inmutables:

```
1  data = {14, "Hola", 'x'} # Tupla
```

Se declaran separando valores entre llaves. Permiten tipos heterogéneos.

Se acceden por índice numérico empezando en 0, y suelen ser muy ligeros en los programas, al mismo tiempo que son útiles para devolver múltiples valores desde métodos o como claves de hash.

Tipo	Características	Usos típicos
Array	Contenedor con acceso por índices, mutable	Almacenar y acceder a secuencias de elementos

Hash	Colección clave-valor para búsquedas rápidas	Mapeos clave-valor, caches, sets
Range	Intervalo de valores entre límites inferior y superior	Iteraciones, indexación, condicionales
Tuple	Agrupa valores inmutables de distintos tipos	Devolver múltiples valores, claves de hash

Por ahora, la utilidad de hashes, rangos y tuplas podría no ser totalmente clara. Pero más adelante en el libro, veremos cómo estas estructuras de datos son componentes fundamentales en el desarrollo de programas reales en Crystal. Los hashes permiten modelar datos de forma flexible y eficiente.

Los punteros son variables que almacenan direcciones de memoria y permiten manipulación directa:

Tipo	Descripción	Ejemplo
Pointer(T)	Puntero genérico a cualquier tipo T	ptr = Pointer(Int32).new(1234) ptr.value = 4321
StaticArray(T, N)	Array de tamaño fijo T con N elementos	arr = StaticArray(Int32, 3).new([10, 20, 30])

Suelen ser muy útiles para interoperabilidad con C donde son omnipresentes y permiten código inseguro si se usan incorrectamente.

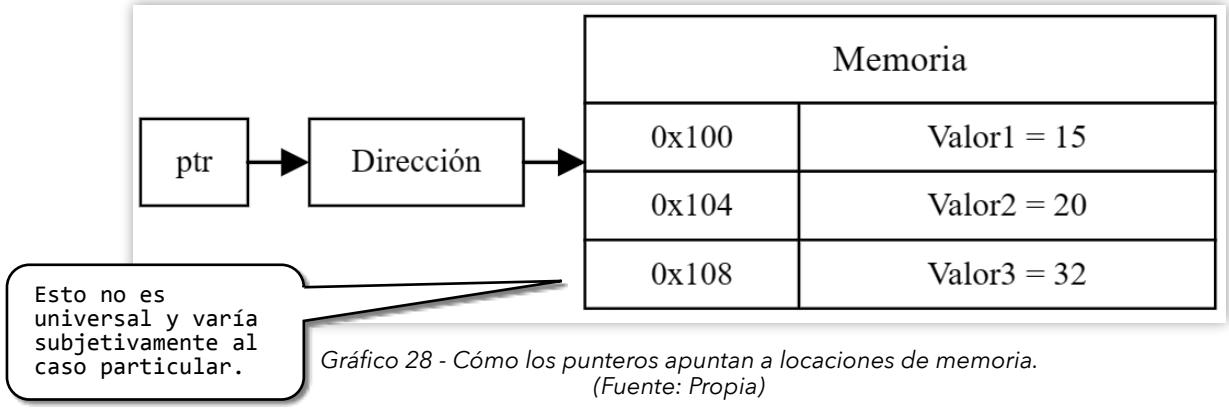


Gráfico 28 - Cómo los punteros apuntan a locaciones de memoria. (Fuente: Propia)

Las uniones permiten que una variable pueda almacenar valores de diferentes tipos:

```
1  valor = 1 # Int32
2  valor = 3.5 # Float64
3  valor = "Hola" # String
```

Se declaran con `union` y deben definirse los posibles tipos a admitir. El programa debe llevar la gestión de qué tipo contiene en cada momento mediante un tag específico, son útiles para optimizar memoria en algunos casos concretos.

Los tipos enumerados permiten definir un conjunto de constantes con nombre:

```
1  enum Color
2    Rojo
3    Verde
4    Azul
5  end
```

Se declaran con `enum`. Sus valores se asignan de forma incremental por defecto (0, 1, 2...). Pueden tener métodos asociados. Muy útiles para representar estados y flags.

Las enumeraciones (Enums) son un tipo de dato especial en Crystal que permiten definir un conjunto de valores con nombres simbólicos. Estos valores se asignan de forma incremental por defecto, comenzando desde 0 y aumentando en 1 para cada miembro. Por ejemplo, si declararas una enumeración `DayOfWeek` con miembros `Monday`, `Tuesday`, `Wednesday`, etc., sus valores serían 0, 1, 2, respectivamente.

Además de tener valores predefinidos, las enumeraciones también pueden tener métodos asociados, lo que las hace muy útiles para representar estados, opciones o banderas (flags) en un programa. Por ejemplo, podrías tener una enumeración `Status` con miembros `Pending`, `InProgress` y `Completed`, y usar estos valores para hacer seguimiento del estado de una tarea o proceso.

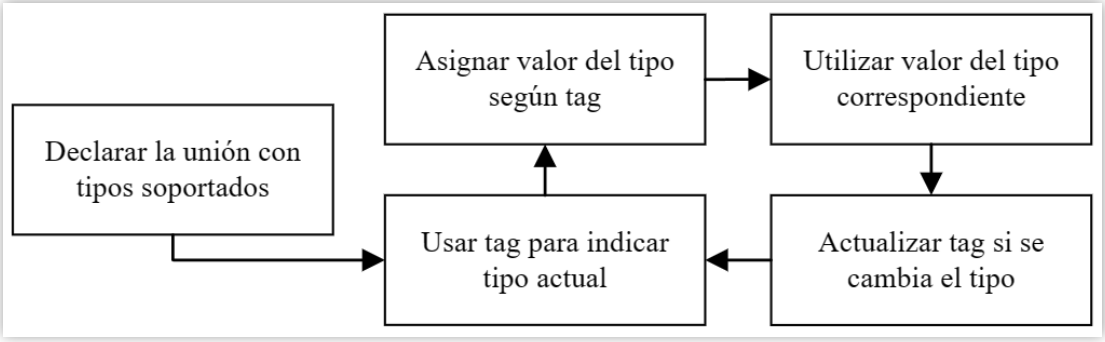


Gráfico 29 - Ciclo de vida de una variable de tipo unión.
(Fuente: Propia)

Crystal permite crear alias y nuevos tipos personalizados: Los tipos personalizados, como los creados mediante `alias` y `struct`, son una característica poderosa de Crystal que te permite modelar y representar conceptos específicos de tu dominio de una manera más clara y expresiva. Veamos algunos ejemplos de cómo puedes utilizar estas herramientas.

```
1  alias Peso = Float64
2  struct Persona
3    Nombre String
4    Edad Int32
5  end
```

Aparte de las enumeraciones, Crystal también permite crear tus propios tipos de datos personalizados utilizando diferentes construcciones del lenguaje. Estas opciones te brindan aún más flexibilidad y expresividad a la hora de modelar tu aplicación.

- `alias` crea un nuevo nombre para un tipo existente.
- `struct` y `class` declaran tipos personalizados.

Los tipos personalizados, ya sean creados mediante `alias` o `struct`, son una característica poderosa de Crystal que te permite modelar y representar conceptos específicos de tu dominio de una manera más clara y expresiva. Al definir tus propios tipos, puedes asociarles comportamientos (métodos) y reglas de validación, lo que ayuda a mantener la integridad de tus datos y hacer que tu código sea más robusto y fácil de entender. En los próximos capítulos profundizaremos más en la creación de clases y estructuras en Crystal, que te permitirán definir tus propios tipos de una manera aún más flexible y poderosa.

Tipo	Descripción
Tipos numéricos	Crystal proporciona varios tipos numéricos enteros y de coma flotante.

Booleano	Dispone de los tipos booleano, caracter y cadena para valores lógicos y textuales.
Colecciones	Incluye potentes tipos de colección como arrays, hashes, ranges y tuples.
Punteros	Los punteros permiten código inseguro pero son necesarios en algunos casos.
Uniones y enumerados	Cumplen nichos especializados.
Alias y personalizados	Podemos crear alias y tipos personalizados.

Los tipos son la base para construir programas sólidos y eficientes en Crystal. Conocer sus capacidades es esencial para todo programador Crystal.

Tipo	Descripción	Tamaño	Uso típico
Int	Enteros con signo	32 o 64 bits	Counters, ids
UInt	Enteros sin signo	32 o 64 bits	Iteradores, hashes
Float	Coma flotante	32 o 64 bits	Cálculos matemáticos
Bool	Booleano	1 byte	Lógica, condicionales
Char	Caracter	4 bytes	Texto
String	Cadena	Variable	Texto
Array	Contenedor indexado	Variable	Secuencias
Hash	Diccionario clave-valor	Variable	Mapeos
Range	Intervalo de valores	Variable	Iteraciones
Tuple	Valores inmutables	Variable	Múltiples valores

En el siguiente apartado veremos las variables y constantes, que nos permitirán almacenar y nombrar valores de estos tipos en nuestro código.

2.3 Variables y constantes

Las variables y constantes son componentes fundamentales en cualquier lenguaje de programación. Permiten almacenar información y referenciarla mediante un identificador. En Crystal, las variables se declaran con la palabra clave `var` seguida del nombre de la variable y su tipo de dato:

```
1  var edad = 25 // edad es un entero de 32 bits
2  var nombre = "Juan" // nombre es un string
3  var flotante = 3.14 // flotante es un número de punto flotante de 64 bits
```

El tipo de dato se infiere automáticamente según el valor asignado a la variable. También se pueden declarar variables sin asignar un valor inicial.

Antes de profundizar en los diferentes tipos de variables, es importante entender cómo Crystal gestiona la memoria y el ciclo de vida de las variables. Crystal utiliza un sistema de gestión de memoria inteligente que combina el uso eficiente del stack para variables simples y el heap para objetos más complejos, todo supervisado por un recolector de basura automático.

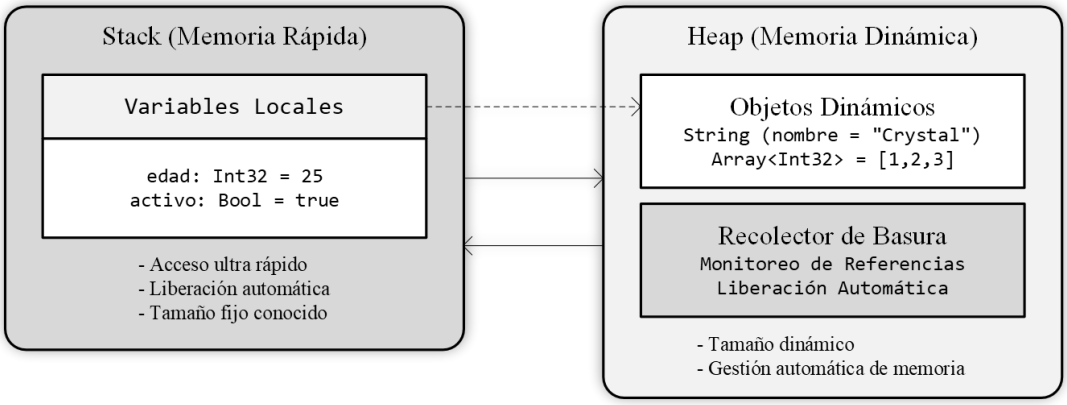


Gráfico 30 - Arquitectura del sistema de gestión de memoria dual en Crystal: Stack vs Heap. (Fuente: Propia)

Como se observa en el Gráfico 2, Crystal optimiza el uso de memoria distribuyendo las variables entre el stack y el heap según su naturaleza. Las variables locales simples se almacenan en el stack para un acceso rápido, mientras que los objetos más complejos se manejan en el heap con referencias automáticamente gestionadas por el recolector de basura. Esta arquitectura permite un equilibrio óptimo entre rendimiento y gestión segura de la memoria.

```
1  var edad: Int32
2  var nombre: String
3  var flotante: Float64
```

En este caso es necesario especificar explícitamente el tipo. Por otro lado, las constantes se declaran con la palabra clave `const` en lugar de `var`. Su valor no puede cambiar después de la declaración:

```
1  const PI = 3.141592 // PI es una constante
2  PI = 3 // error, no se puede reasignar el valor
```

Los nombres de variables y constantes deben seguir las siguientes convenciones:

- Deben comenzar con una letra minúscula.
- Pueden contener letras, números y guiones bajos.
- No pueden contener espacios ni caracteres especiales.
- Deben ser nombres autoexplicativos que denoten su propósito.
- Las constantes deben declararse en mayúsculas y con guiones bajos para separar palabras.

Característica	Var	Const
Valor inicial	Opcional	Obligatorio
Reasignación	Permite reasignar	No permite reasignar
Ámbito	Local, global	Local, global
Convención	minúsculas	MAYÚSCULAS
Uso recomendado	Para valores que cambian	Para valores fijos
Ejemplo	`var edad = 25`	`CONST PI = 3.14159`

Estos son ejemplos de nombres válidos:

```
1  var edad_usuario = 25
```


```
2  const PI = 3.141592
3  var nombre_completo = "Juan Pérez"
```

Mientras que estos nombres no siguen las convenciones:

```
1  var 1era_edad = 25 // no puede empezar con número
2  var edad&usuario = 30 // no puede contener &
3  const pi = 3.14 // las constantes van en mayúsculas
4  var Nombre Completo = "Juan" // no puede contener espacios
```

Se recomienda elegir nombres significativos y no abreviar, para mejorar la legibilidad del código:

```
1  # Mal
2  var n = "Juan"
3  var r = 25
4  # Bien
5  var nombre = "Juan"
6  var edad = 25
```

El ámbito de una variable define donde está disponible para ser usada en el programa. Las variables declaradas fuera de cualquier definición de método, clase o módulo tienen ámbito global, y están disponibles en todo el programa.

Las variables declaradas dentro de un método solo existen dentro de ese método. Las variables de instancia en una clase están disponibles en todos los métodos de esa clase. Esto ilustra los diferentes ámbitos:

```
1  # Ámbito global
2  var edad_global = 25
3  class Usuario
4    # Disponible en toda la clase
5    @nombre : String
6    def initialize
7      # Ámbito de método
8      var edad_local = 30
9    end
10 end
11 # edad_global sigue disponible
12 p edad_global # => 25
13 # edad_local no está disponible fuera del método
14 p edad_local # !! Error
15 usuario = Usuario.new
16 # @nombre está disponible en la instancia
17 usuario.@nombre = "Juan"
```

En general se recomienda reducir el ámbito de las variables todo lo posible. Declararlas dentro del método donde se usan evita completamente posibles errores por variables globales modificadas en otra parte del programa.

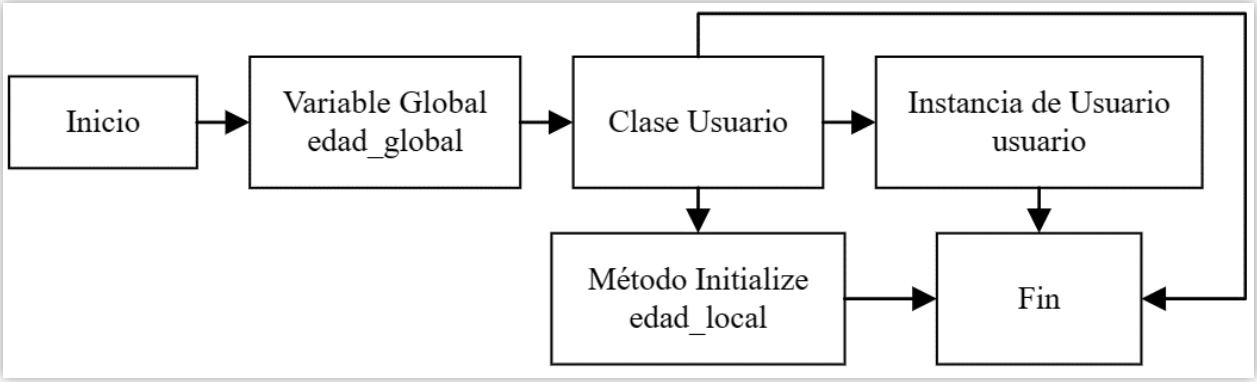


Gráfico 31 – Alcance de variables: global, de instancia y local.
(Fuente: Propia)

La asignación de valores a una variable se realiza con el operador =:

```
1  var edad = 25
2  edad = 26 # reasignamos el valor
```

Para asignar múltiples variables en una línea se usa una sintaxis abreviada:

```
1  # Sin abreviar
2  var edad = 25
3  var nombre = "Juan"
4  # Abreviado
5  var edad = 25, nombre = "Juan"
```

Crystal es un lenguaje con tipado estático. Eso significa que cada variable y valor tienen un tipo de dato asociado que se comprueba en tiempo de compilación, pero los tipos se infieren automáticamente en la mayoría de casos:

```
1  var edad = 25 # edad se deduce como Int32
2  var pi = 3.14 # pi se deduce como Float64
3  var nombre = "Juan" # nombre se deduce como String
```

Para comprender mejor cómo Crystal maneja las variables durante la ejecución del programa, es útil visualizar los diferentes estados por los que pasa una variable desde su declaración hasta su liberación:

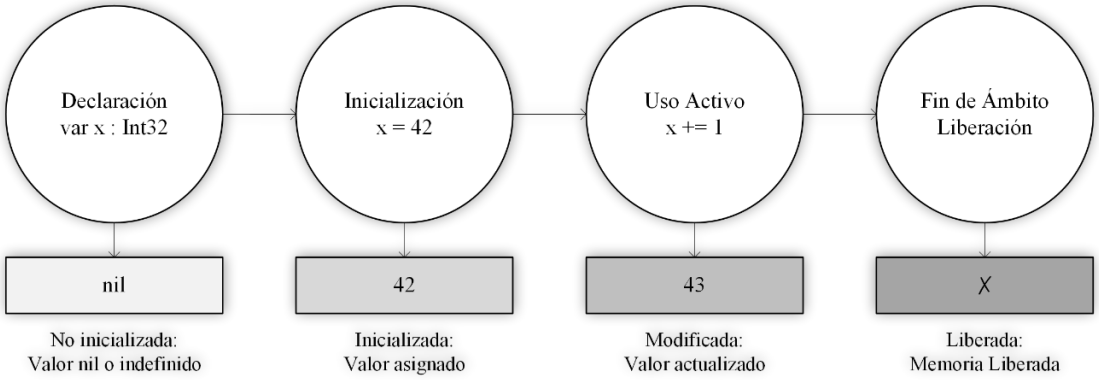


Gráfico 32 - Estados y transformaciones de una variable durante su ejecución.
(Fuente: Propia)

Esto evita tener que declarar explícitamente el tipo cada vez, aunque también es posible hacerlo:

```
1  var edad : Int32 = 25
2  var pi : Float64 = 3.14
3  var nombre : String = "Juan"
```

La inferencia automática hace el código más conciso y legible. Pero forzar la declaración explícita del tipo puede ayudar a evitar errores.

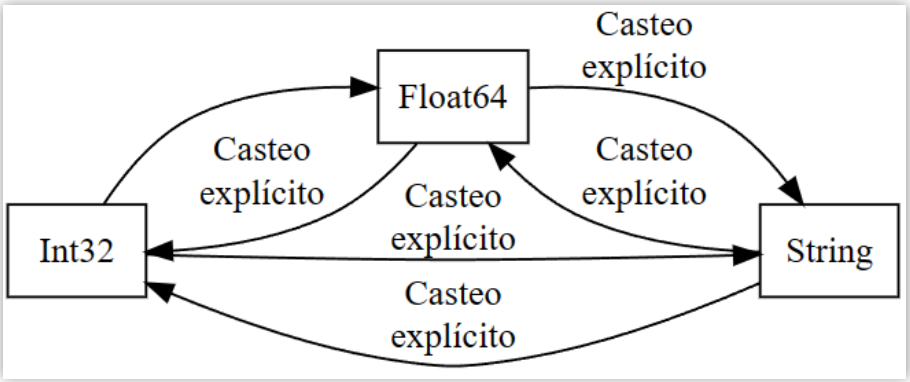


Gráfico 33 - Relaciones y conversiones de tipos en Crystal.
(Fuente: Propia)

Crystal no permite asignar valores de diferente tipo a una variable, a menos que se haga una conversión explícita:

```
1  var edad = 25 // edad es Int32
2  edad = "veinticinco" # ¡Error! No se puede asignar un String
```

Esto ayuda a evitar errores. Pero se pueden realizar conversiones:

```
1  var edad = "25"
2  edad = edad.to_i # convierte el string a entero
```

Forzar estas conversiones explícitas hace el código más seguro.

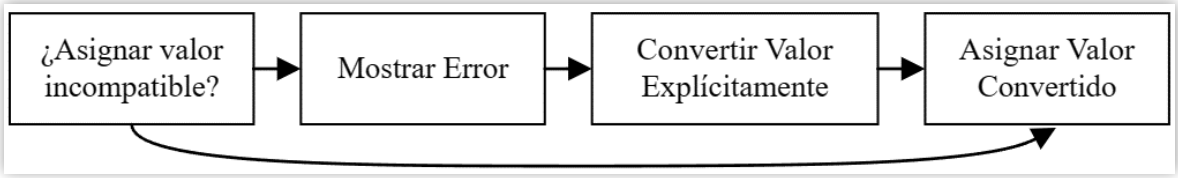


Gráfico 34 – Flujo de manejo de conversiones de tipos en Crystal.
(Fuente: Propia)

Crystal proporciona una gran variedad de tipos de datos integrados. Esta tabla resume los tipos integrados más usados:

Tipo	Descripción	Ejemplo
Int32	Enteros con signo de 32 bits. Ocupan 4 bytes y pueden representar valores entre -2,147,483,648 y 2,147,483,647. Útiles para contadores, iteradores, manejar índices de arreglos, etc.	var edad: Int32 = 25
Float64	Punto flotante de doble precisión con 64 bits. Ocupan 8 bytes y pueden representar números con fracciones y exponentes en un amplio rango. Útiles para cálculos matemáticos.	var promedio: Float64 = 92.75
String	Secuencia inmutable de caracteres Unicode. Útiles para representar y manipular texto. Se declaran entre comillas dobles.	var nombre: String = "Juan"
Bool	Tipo booleano que sólo puede tener los valores true o false. Útil para almacenar estados binarios.	var terminado: Bool = false
Array	Colección indexada y mutable de elementos del mismo tipo. Acceso rápido a partir de un índice entero.	var lista: Array(Int32) = [1, 2, 3]

Se pueden convertir variables de un tipo a otro usando los métodos to_X, donde X es el tipo destino:

```
1  var flotante = 3.1416.to_f      # Float64 -> Float32
2  var entero = "123".to_i         # String -> Int32
3  var string = 123.to_s           # Int32 -> String
```

También con el constructor del tipo:

```
1  var flotante = Float32.new(3.1416) # Float64 -> Float32
2  var string = String.new(123)       # Int32 -> String
```

Y de manera implícita:

```
1  var flotante : Float32 = 3.1416 # asigna Float64 a Float32 implícitamente
```

Algunas conversiones comunes son:

Tipo Origen	Tipo Destino	Método directo	Descripción
Int32	Float32	to_f32	Convierte el entero con signo a coma flotante de 32 bits. Se puede perder precisión.
Int32	String	to_s	Convierte el valor numérico a una cadena de texto representando el número.
Float64	Int32	to_i32	Convierte de punto flotante a entero con signo de 32 bits. Se recorta la parte decimal.
String	Int32	to_i	Analiza la cadena como un número entero y devuelve el valor. Lanza error si no se puede convertir.

Array	Hash	to_h	Convierte el array a un hash usando los índices numéricos de elementos como claves.

Las variables numéricas soportan las operaciones matemáticas básicas:

```
var a = 10
var b = 5
# Suma
var c = a + b # c es 15
# Resta
var d = a - b # d es 5
# Multiplicación
var e = a * b # e es 50
# División
var f = a / b # f es 2
# Módulo (resto)
var g = a % b # g es 0 (10 / 5 = 2 resto 0)
```

Control de Flujo en Crystal

El control de flujo es uno de los conceptos más fundamentales en la programación, ya que permite que nuestros programas tomen decisiones y ejecuten diferentes bloques de código según las condiciones que especifiquemos. En Crystal, al igual que en otros lenguajes de programación modernos, disponemos de varias estructuras de control que nos permiten dirigir el flujo de ejecución de nuestros programas de manera eficiente y elegante.

A diferencia de otros lenguajes, Crystal combina la claridad sintáctica heredada de Ruby con la seguridad del tipado estático, lo que nos permite escribir código más seguro y mantenible. En esta sección, exploraremos en detalle cada una de las estructuras de control disponibles en Crystal, sus casos de uso más comunes, y las mejores prácticas para su implementación.

Estructuras Condicionales

La estructura condicional más básica y fundamental en Crystal es el `if`. Esta estructura nos permite ejecutar diferentes bloques de código basándonos en una condición booleana. A diferencia de lenguajes como JavaScript o Python, Crystal es más estricto en cuanto a qué se considera verdadero o falso, lo que ayuda a prevenir errores comunes.

Veamos un ejemplo práctico que ilustra este concepto:

```
edad = 18
if edad >= 18
  puts "Eres mayor de edad"
else
  puts "Eres menor de edad"
end
```

Este código, aunque simple, demuestra varios conceptos importantes:

1. La condición `edad >= 18` debe evaluar a un valor booleano (`true` o `false`)

2. Crystal utiliza la palabra clave `puts` para imprimir en la consola
3. La estructura completa se cierra con la palabra clave `end`
4. No se requieren paréntesis alrededor de la condición, aunque son opcionales

Es importante notar que Crystal realizará una comprobación de tipos en tiempo de compilación. Por ejemplo, si intentáramos comparar tipos incompatibles, obtendríamos un error de compilación, lo que nos ayuda a detectar problemas antes de que el código se ejecute.

Comparativa con Otros Lenguajes Populares

Para entender mejor las particularidades de Crystal en cuanto al control de flujo, es útil compararlo con otros lenguajes populares. La siguiente tabla muestra las diferentes aproximaciones que toman los lenguajes más comunes:

Lenguaje	Sintaxis	Características Especiales	Consideraciones
Crystal	if condicion; codigo; end	Inferencia de tipos en la condición	Detecta errores de tipo en tiempo de compilación
Ruby	if condicion; codigo; end	Similar a Crystal, más flexible	Permite más valores como verdaderos (no solo true)
Python	if condicion: codigo	Usa indentación para bloques	La indentación es significativa sintácticamente
JavaScript	if (condicion) { codigo }	Evaluación flexible de condiciones	Puede llevar a comportamientos inesperados por coerción
Java	if (condicion) { codigo }	Requiere condición booleana explícita	Más verboso pero más explícito

Esta tabla nos muestra que Crystal toma un enfoque equilibrado: mantiene la elegancia sintáctica de Ruby mientras proporciona la seguridad de tipos de lenguajes como Java. Los paréntesis son opcionales (como en Ruby), pero las condiciones deben ser explícitamente booleanas (como en Java).

Por ejemplo, mientras que en JavaScript podríamos escribir:

```
1  if (1) {
2    console.log("Esto se ejecutará");
3  }
```

En Crystal, esto resultaría en un error de compilación, ya que esperamos una condición booleana explícita:

```
1  if 1  # Error de compilación
2    puts "Esto no compilará"
3  end
4  # La forma correcta sería:
5  if 1 > 0
6    puts "Esto sí compilará"
7  end
```


Estructuras de Control Anidadas

En la programación real, frecuentemente nos encontramos con situaciones que requieren múltiples niveles de decisión. Crystal nos permite anidar estructuras condicionales para manejar estas situaciones complejas. Sin embargo, es importante mantener el código legible y mantenible cuando trabajamos con condiciones anidadas.

El siguiente diagrama muestra cómo se ramifican las decisiones en un sistema de clasificación de usuarios:

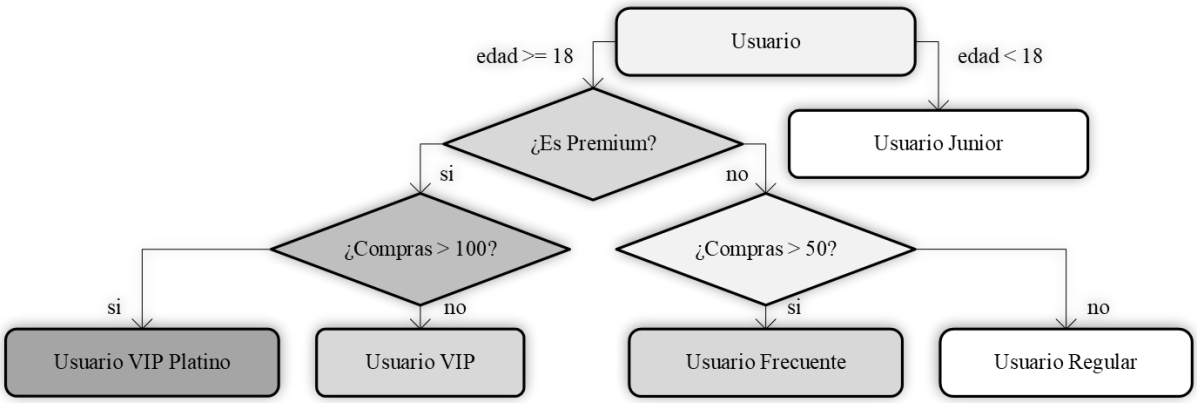


Gráfico 35 - Árbol de decisión de clasificación de usuarios en Crystal.
(Fuente: Propia)

Este diagrama ilustra visualmente cómo cada decisión lleva a nuevas posibilidades, creando un árbol de decisiones que nuestro código debe implementar. La implementación de este árbol de decisiones en Crystal sería así:

```
1  def clasificar_usuario(edad : Int32, compras : Int32, premium : Bool)
2    if edad >= 18
3      if premium
4        if compras > 100
5          "Usuario VIP Platino"
6        else
7          "Usuario VIP"
8        end
9      else
10       if compras > 50
11         "Usuario Frecuente"
12       else
13         "Usuario Regular"
14       end
15     end
16   else
17     "Usuario Junior"
18   end
19 end
```

Este código implementa la lógica del árbol de decisiones anterior. Analicemos sus aspectos clave:

Aspecto	Descripción
Tipado de Parámetros	La función acepta parámetros con tipos explícitos (:Int32 y :Bool)
Niveles de Anidamiento	Cada nivel de anidamiento representa una rama del árbol de decisiones
Valores de Retorno	El código devuelve diferentes strings según la clasificación del usuario
Estructura Visual	La indentación ayuda a visualizar la jerarquía de decisiones

Sin embargo, el código anterior, aunque funcional, podría mejorarse para hacerlo más mantenible. Una versión refactorizada podría verse así:

```
def es_usuario_vip?(compras : Int32, premium : Bool)
  premium && compras > 50
end
def obtener_nivel_vip(compras : Int32)
  compras > 100 ? "Platino" : "Regular"
end
def clasificar_usuario(edad : Int32, compras : Int32, premium : Bool)
  return "Usuario Junior" if edad < 18
  if es_usuario_vip?(compras, premium)
    "Usuario VIP #{obtener_nivel_vip(compras)}"
  else
    compras > 50 ? "Usuario Frecuente" : "Usuario Regular"
  end
end
```

Esta versión refactorizada ilustra varias mejores prácticas en Crystal:

- Separación de responsabilidades en métodos más pequeños
- Uso de guardas tempranas para reducir el anidamiento
- Utilización del operador ternario para condiciones simples
- Nombres de métodos descriptivos que indican su propósito

Uso de Case/When

Cuando necesitamos evaluar múltiples condiciones sobre una misma variable, la estructura `case` (también conocida como `switch` en otros lenguajes) puede hacer nuestro código más limpio y eficiente. Crystal implementa esta estructura de una manera particularmente elegante y poderosa.

Veamos el diagrama de flujo que representa cómo funciona la estructura `case`:

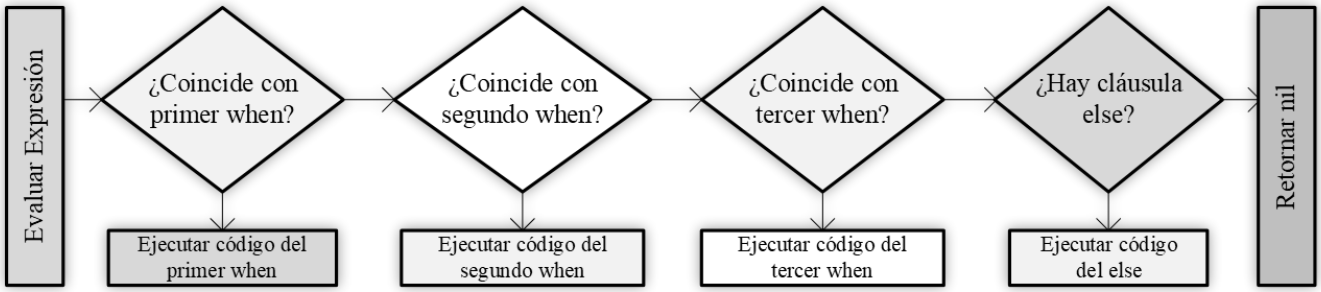


Gráfico 36 - Flujo de ejecución de la estructura case.
(Fuente: Propia)

Este diagrama muestra cómo la ejecución fluye a través de diferentes casos hasta encontrar una coincidencia. Una característica única de Crystal es que permite usar rangos y expresiones complejas en los patrones de coincidencia.

Analicemos un ejemplo completo de sistema de calificaciones:

```
1   puntuacion = 85
2   case puntuacion
3   when 90..100
4     puts "Sobresaliente"
5   when 80..89
6     puts "Notable"
7   when 70..79
8     puts "Bien"
9   when 60..69
10    puts "Suficiente"
11  else
12    puts "Necesita mejorar"
13  end
```

Este código demuestra varias características importantes del `case` en Crystal:

1. Uso de rangos (**90..100**) como patrones de coincidencia
2. Evaluación secuencial de arriba hacia abajo
3. Cláusula **else** para manejar casos no contemplados
4. No necesita declaraciones **break** como en otros lenguajes

Es importante señalar que Crystal optimiza la estructura `case` internamente, convirtiéndola en una tabla de saltos cuando es posible, lo que la hace más eficiente que una serie de `if/elsif`.

Ejemplos Prácticos Realistas

Para entender verdaderamente cómo se utilizan las estructuras de control en aplicaciones del mundo real, examinaremos dos ejemplos complejos y prácticos: un sistema de calificación de préstamos y un sistema de control de acceso.

Sistema de Calificación de Préstamos

El siguiente ejemplo muestra cómo Crystal puede manejar lógica empresarial compleja de manera clara y mantenible:

```
def calificar_prestamo(
    ingreso_anual : Float64,
    historial_crediticio : Int32,
    deuda_actual : Float64
  )
  score = 0
  if ingreso_anual > 50000
    score += 20
    if ingreso_anual > 100000
      score += 15
    end
  end
  case historial_crediticio
  when 800..850
    score += 40
  when 740..799
    score += 30
  when 670..739
    score += 20
  when 580..669
    score += 10
  end
  ratio_deuda = deuda_actual / ingreso_anual
  if ratio_deuda < 0.2
    score += 20
  elsif ratio_deuda < 0.4
    score += 10
  end
  case score
  when 70..100
    "Aprobado - Tasa Preferencial"
  when 50..69
    "Aprobado - Tasa Estándar"
  when 30..49
    "Aprobado con Condiciones"
```


```
  else
    "Denegado"
  end
end
```

Este ejemplo demuestra varios conceptos avanzados:

Concepto	Descripción
Tipado Preciso	Parámetros tipados con precisión usando Float64 e Int32
Estructuras Combinadas	Combinación efectiva de estructuras if y case
Integración Matemática	Cálculos matemáticos integrados en la lógica de decisión
Sistema de Puntuación	Implementación de un sistema de puntuación ponderado
Evaluación Multinivel	Múltiples niveles de evaluación para toma de decisiones

Sistema de Control de Acceso

Para comprender mejor cómo funcionaría un sistema de control de acceso real, observemos primero el diagrama de flujo:

Este diagrama ilustra las múltiples verificaciones que debe realizar un sistema de control de acceso. Ahora veamos la implementación en Crystal:

```
def verificar_acceso(
    usuario : String,
    nivel_acceso : Int32,
    hora_actual : Time,
    ubicacion : String
  )
  # Verificación de horario laboral
  hora = hora_actual.hour
  dia = hora_actual.day_of_week
  horario_permitido = case dia
    when .sunday?, .saturday?
      false
    else
      (9..18).includes?(hora)
    end
  # Verificación de ubicación
  ubicacion_permitida = ["Oficina Central", "Sucursal Norte", "Sucursal Sur"].includes?
(ubicacion)
  # Verificación de credenciales
  credenciales_validas = if nivel_acceso >= 3
    true
```

```
  elsif nivel_acceso == 2
    ubicacion == "Oficina Central"
  else
    false
  end
  if horario_permitido && ubicacion_permitida && credenciales_validas
    {autorizado: true, mensaje: "Acceso concedido"}
  else
    razon = if !horario_permitido
      "Fuera de horario laboral"
    elsif !ubicacion_permitida
      "Ubicación no autorizada"
    else
      "Nivel de acceso insuficiente"
    end
    {autorizado: false, mensaje: razon}
  end
end
def verificar_acceso(
```

Este ejemplo demuestra características avanzadas de Crystal:

Característica	Descripción
Manejo de Tiempo	Uso de tipos complejos como Time para control temporal
Retorno Múltiple	Método que retorna un hash con múltiples valores
Control Combinado	Combinación eficiente de estructuras de control
Manejo de Errores	Sistema de manejo de errores descriptivo y claro
Organización	Lógica de negocio compleja organizada de manera clara

Consideraciones y Mejores Prácticas

Cuando trabajamos con estructuras de control en Crystal, es fundamental seguir ciertas prácticas que nos ayudarán a mantener nuestro código limpio, eficiente y mantenible:

Práctica	Descripción	Ejemplos
Claridad en Condiciones	Mantén las condiciones simples y legibles	- Divide condiciones complejas en variables descriptivas - Usa métodos auxiliares para lógica compleja - Evita la negación doble
Control de Anidamiento	Evita el anidamiento excesivo de código	- Utiliza guardas tempranas (early returns) - Extrae lógica compleja a métodos separados - Implementa patrones de diseño apropiados
Uso del Sistema de Tipos	Aprovecha el sistema de tipos de Crystal	- Usa tipos explícitos para mayor claridad - Aprovecha la inferencia de tipos cuando sea obvio - Utiliza union types para casos especiales

Mantenibilidad	Optimiza el código para mantenimiento futuro	- Escribe comentarios explicativos cuando sea necesario - Usa nombres descriptivos - Mantén consistencia en el estilo

La siguiente imagen muestra una comparativa visual de las diferentes estructuras de control disponibles en Crystal y cuándo usar cada una:

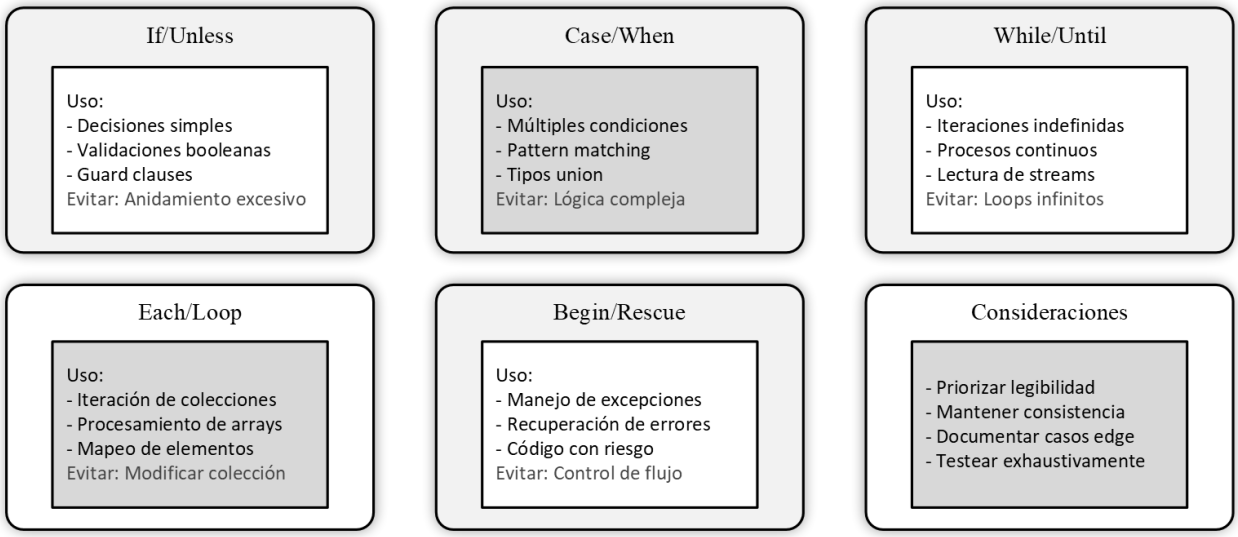


Gráfico 37 - Comparativa de estructuras de control en Crystal y sus casos de uso.
(Fuente: Propia)

Esta comparativa nos ayuda a elegir la estructura más apropiada según nuestras necesidades específicas. Recuerda que no existe una solución única para todos los casos, y parte del arte de la programación está en elegir la herramienta correcta para cada situación.

También operaciones abreviadas de asignación:

```
1  var a = 10
2  a += 5 # a ahora es 15
3  a -= 2 # a ahora es 13
4  a *= 2 # a ahora es 26
5  a /= 2 # a ahora es 13
6  a %= 6 # a ahora es 3
```

Y operaciones incrementales:

```
1  var a = 10
2  a++ # a ahora es 11
3  a-- # a ahora es 10
4  a += 1 # También incrementa a en 1
5  a -= 1 # También decrementa a en 1
```

Las variables se pueden comparar con los operadores relacionales:

```
1  var a = 10
2  var b = 5
3  # Igual que
```

```
a == b # false
# Distinto que
a != b # true
# Mayor que
a > b # true
# Mayor o igual que
a >= b # true
# Menor que
a < b # false
# Menor o igual que
a <= b # false
```

Esto permite realizar verificaciones condicionales:

```
var edad = 15
if edad >= 18
  print "Eres mayor de edad"
else
  print "Eres menor de edad"
end
```

Para combinar comparaciones se usan los operadores lógicos:

```
# AND lógico
a >= 5 && b <= 10
# OR lógico
a == 5 || b != 4
# Negación
!(a == b)
```

Con ellos se pueden crear expresiones más complejas:

```
var edad = 15
var nota = 8
if (edad >= 18 && nota >= 7) || (edad < 18 && nota >= 9)
  print "Aprobado"
else
  print "Reprobado"
end
```

Para incluir el valor de una variable dentro de un string se usa interpolación:

```
var nombre = "Juan"
print "Hola #{nombre}" # Salida: Hola Juan
```


También se pueden incluir expresiones:

```
print "Doble de 5 es #{2 * 5}" # Salida: Doble de 5 es 10
```

Los punteros almacenan una referencia en memoria a otro valor. Los tipos básicos en Crystal incluyen los tipos numéricos enteros y de punto flotante, los valores booleanos, caracteres y cadenas de texto. La jerarquía de tipos en Crystal es fundamental para comprender cómo se relacionan y convierten entre sí los diferentes tipos de datos.

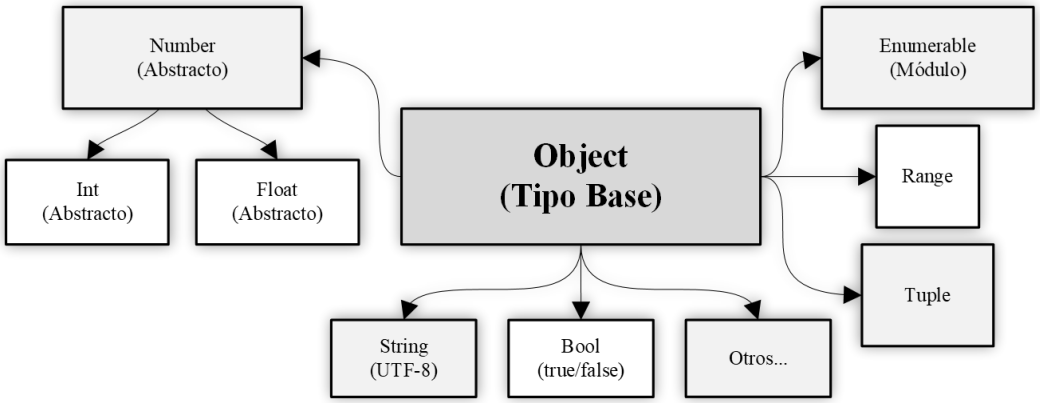


Gráfico 38 - Jerarquía completa de tipos en Crystal.
(Fuente: Propia)

Para entender mejor cómo se relacionan los tipos numéricos, observemos la siguiente tabla de conversiones comunes:

Desde	Hacia	Método	¿Puede perder precisión?	Ejemplo
Int8	Int16	to_i16	No	123_i8.to_i16
Int16	Int32	to_i32	No	12345_i16.to_i32
Int32	Int64	to_i64	No	123456_i32.to_i64
Int64	Int32	to_i32	Sí	123456_i64.to_i32
Float32	Float64	to_f64	No	123.45_f32.to_f64
Float64	Float32	to_f32	Sí	123.45_f64.to_f32
Int32	Float64	to_f	No	123_i32.to_f
Float64	Int32	to_i	Sí	123.45_f64.to_i

Es importante destacar las siguientes relaciones entre tipos:

1. **Numéricos a String**

```
# Cualquier número puede convertirse a String
42.to_s          # => "42"
3.14159.to_s     # => "3.14159"
```

1. **String a Numéricos**

```
# Strings pueden convertirse a números si contienen valores válidos
"42".to_i        # => 42
"3.14159".to_f   # => 3.14159
```

Las conversiones entre tipos son una parte fundamental del desarrollo en Crystal, ya que el sistema de tipos estático requiere que seamos explícitos sobre cómo queremos que los datos se transformen de un tipo a otro. Entender esta jerarquía y las relaciones entre tipos nos ayudará a escribir código más seguro y eficiente.

Operación	Descripción	Ejemplo	Resultado
Promoción	Conversión automática a tipo más amplio	`1 + 1.0`	`Float64`
Truncamiento	Conversión con posible pérdida de datos	`3.14.to_i`	`3`
Parse	Conversión desde String	`"123".to_i`	`123`
Stringify	Conversión a String	`123.to_s`	`"123"`

Se declaran con `Pointer(Tipo)` y se obtienen con la referencia `&` a una variable:

```
1  var a = 1
2  var puntero = Pointer(Int32).new(1_000) # Puntero a dirección 1000
3  var b = 2
4  var puntero2 = &b # Puntero a b
```

Se lee el valor con `value`:

```
1  print puntero.value # Lee el Int32 en 1000
2  print puntero2.value # Lee el valor de b
```

Y se asigna con `value=`:

```
1  puntero.value = 55 # Cambia el valor en la dirección 1000
2  puntero2.value = 8 # Cambia el valor de b a 8
```

Los punteros son útiles para memoria compartida y rendimiento. Pero también son peligrosos y difíciles de gestionar. nil representa la ausencia de un valor. Las variables se asignan a nil por defecto si no tienen un valor inicial:

Para comprobar si una variable es nil se usa `nil?`:

```
1  if edad.nil?
2      print "La variable edad no tiene valor"
3  end
```

Nil no es lo mismo que 0 o false. Es una falta completa de valor. Las variables locales se inicializan a nil por defecto.

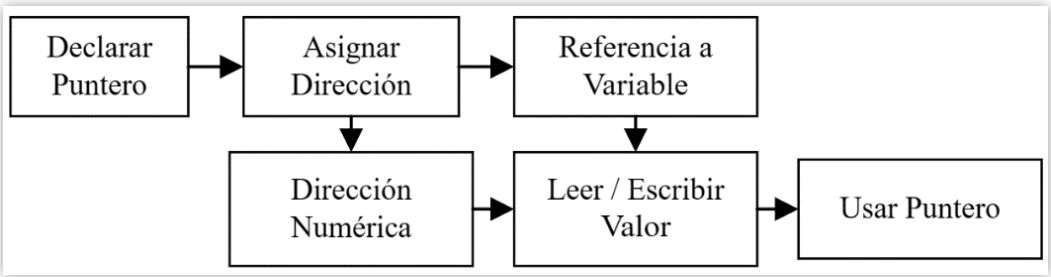


Gráfico 39 – Ciclo de vida y operaciones básicas de un puntero en Crystal.
(Fuente: Propia)

Las variables de instancia (@ivars) se pueden asignar a un valor por defecto:

```
class Usuario
    @edad = 0 # edad será 0 por defecto
end
```

Los parámetros de métodos tienen nil por defecto:

```
def saludar(nombre = nil)
    if nombre
        print "Hola #{nombre}"
    else
        print "Hola extraño"
    end
end
```

Esto permite llamar sin pasar un valor para el parámetro:

```
saludar # Imprime "Hola extraño"
saludar("Juan") # Imprime "Hola Juan"
```

Crystal maneja la memoria automáticamente con un **recolector de basura (GC)**. Las variables se crean y liberan dinámicamente sin necesidad de gestionarlo manualmente.

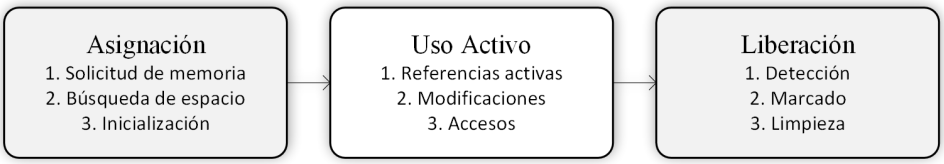


Gráfico 40 - Ciclo de vida y gestión dinámica de memoria en Crystal.
(Fuente: Propia)

El recolector libera la memoria de variables que no se usan más:

```
def crear_y_destruir
    var obj = Usuario.new # Se crea
    obj. hacer_algo        # Se usa
    # Cuando termine el método, obj será liberado por el GC
end
```

Esto simplifica el código y evita problemas típicos de C como fugas de memoria. Aunque también tiene desventajas de rendimiento en casos muy específicos.

Para entender mejor cómo Crystal gestiona la memoria y las variables durante la ejecución del programa, observemos el siguiente diagrama que ilustra la interacción entre la pila (stack) y el montículo (heap):

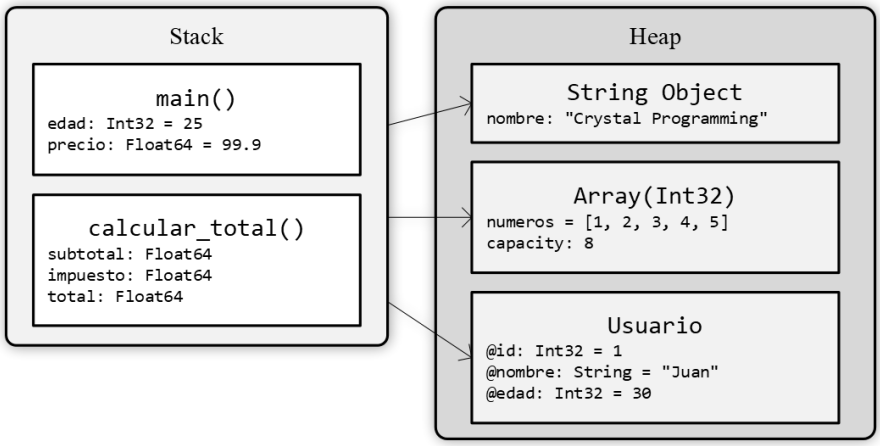


Gráfico 41 - Organización de memoria en tiempo de ejecución en Crystal.
(Fuente: Propia)

Este diagrama muestra cómo Crystal organiza la memoria de manera eficiente. En la pila (stack) se almacenan las variables locales y valores primitivos, organizados en marcos de pila por cada función. En el montículo (heap) se guardan los objetos más complejos como strings, arrays y objetos personalizados. Las flechas muestran cómo las variables en la pila pueden referenciar objetos en el montículo. Esta arquitectura permite a Crystal combinar la eficiencia del acceso a memoria de la pila con la flexibilidad de almacenamiento dinámico del montículo.

Crystal ofrece un excelente rendimiento gracias a:

Característica	Descripción	Beneficio
Compilación a código binario nativo	El código Crystal se compila a código máquina optimizado específico para la plataforma de destino	Permite ejecutar el programa directamente en la CPU sin interpretación, logrando un rendimiento cercano a C/C++
Tipado estático y type safety	El tipo de variables se conoce en tiempo de compilación y el lenguaje garantiza la seguridad de tipos	Permite más optimizaciones por parte del compilador y reduce la sobrecarga en tiempo de ejecución
Optimizaciones del compilador	El compilador Crystal aplica análisis y optimizaciones a nivel de código intermedio	Mejora el rendimiento sin necesidad de que el programador las implemente manualmente
Garbage collector	Usa un recolector de basura eficiente basado en reference counting	Evita problemas como fugas de memoria y facilita la gestión de recursos
Concurrencia nativa	Soporte integrado en el lenguaje para concurrencia con channels y fibers	Permite aprovechar múltiples núcleos CPU de forma eficiente

Es más rápido que lenguajes dinámicos como Ruby, Python o PHP.

Ahora, es momento de que veamos algunos ejercicios prácticos para aplicar los conceptos fundamentales presentados en este capítulo. Resolverlos permitirá afianzar los conocimientos adquiridos y prepararnos para utilizar Crystal en problemas reales.

Ejercicio	Descripción
1	Declara dos variables enteras x e y. Asigna el valor 10 a x y el valor 20 a y. Intercámbia los valores de modo que x pase a valer 20 y y pase a valer 10.

2	Declara una constante llamada GRAVEDAD con el valor 9.8. Úsala en una fórmula para calcular la velocidad final al caer un objeto desde una altura dada.
3	¿Por qué es importante poner nombres significativos a las variables y métodos en un programa? Pon un ejemplo de un mal nombre para variable y cómo lo mejorarías.
4	Explica con tus palabras la diferencia entre var y const en Crystal. ¿Cuándo se debe usar uno u otro?
5	Dado el array **nums = [5, 8, 2, 9, 3]**, itera sobre sus elementos e imprime cada uno.
6	Crea un hash que mapee nombres de estudiantes a notas. Incluye al menos tres estudiantes con sus respectivas notas. Luego imprime el hash.
7	Explica qué operadores permiten combinar expresiones booleanas en Crystal. Pon un ejemplo.
8	Escribe un programa que indique si un número dado es par o impar.
9	Escribe un programa que imprima los números del 1 al 100, pero que en lugar de los múltiplos de 3 imprima "Crystal" y en los múltiplos de 5 imprima "Lang".
10	Escribe una función que reciba un array de números y devuelva su promedio.
11	Define un método de instancia en una clase Persona que calcule si la persona es mayor o menor de edad.
12	Explica qué se entiende por ámbito o scope de una variable en Crystal. Pon un ejemplo.
13	Describe al menos dos formas de convertir un valor de tipo string a número entero.
14	Explica cómo convertir un Array a Hash en Crystal, dando un ejemplo concreto.

Con estos ejercicios prácticos podremos aplicar y consolidar los conceptos de variables, tipos de datos, estructuras de control de flujo, funciones y clases presentados en el capítulo. Resolverlos ayudará a tener una comprensión sólida de las bases del lenguaje antes de avanzar a temas más complejos.

2.4 Operadores

Los operadores son símbolos especiales que realizan operaciones en uno o más valores y expresiones. Permiten realizar operaciones aritméticas, de comparación, lógicas y más. Crystal soporta los siguientes tipos de operadores:

- Operadores aritméticos
- Operadores de asignación
- Operadores de comparación
- Operadores lógicos
- Operadores de bits

Veamos en detalle cada uno de ellos. Los operadores aritméticos realizan operaciones matemáticas comunes en valores numéricos. Los principales operadores aritméticos son:

Operador	Detalle	Descripción	Ejemplo
+	Suma	Suma dos valores	puts 1 + 2 # Imprime 3
-	Resta	Resta dos valores	puts 2 - 1 # Imprime 1
*	Multiplicación	Multiplica dos valores	puts 2 * 3 # Imprime 6
/	División	Divide dos valores	puts 6 / 2 # Imprime 3
%	Módulo	Calcula el residuo de una división	puts 5 % 2 # Imprime 1
**	Exponenciación	Calcula un número elevado a una potencia	puts 2 ** 3 # Imprime 8

Estos operadores funcionan con los principales tipos numéricos como `Int32`, `Float64`, etc. Además, existen operadores aritméticos compuestos que combinan una operación aritmética con una asignación:

Operador	Nombre	Descripción	Ejemplo	Equivale a
+=	Suma compuesta	Suma el valor de la derecha al operando de la izquierda y almacena el resultado en la variable de la izquierda	x += 5	x = x + 5
-=	Resta compuesta	Resta el valor de la derecha al operando de la izquierda y almacena el resultado en la variable de la izquierda	x -= 5	x = x - 5
*=	Multiplicación compuesta	Multiplica el valor de la derecha con el operando de la izquierda y almacena el resultado en la variable de la izquierda	x *= 5	x = x * 5
/=	División compuesta	Divide el operando de la izquierda entre el valor de la derecha y almacena el resultado en la variable de la izquierda	x /= 5	x = x / 5
%=	Módulo compuesto	Calcula el módulo (resto) de la división del operando de la izquierda entre el valor de la derecha y almacena el resultado en la variable de la izquierda	x %= 5	x = x % 5

Por ejemplo:

```
1  a = 2
2  a += 3 # a ahora vale 5
3  a *= 2 # a ahora vale 10
```

Esto nos permite escribir código más conciso al realizar operaciones aritméticas sobre una variable. El operador básico de asignación es el símbolo =, que asigna el valor de la expresión de la derecha a la variable de la izquierda:

```
1  x = 1 + 2
```

Además del operador =, existen operadores de asignación compuestos como vimos antes.

Operador	Función	Ejemplo	Operador Compuesto	Ejemplo
+	Suma	2 + 2 = 4	+=	x += 1
-	Resta	5 - 2 = 3	-=	x -= 2

*	Multiplicación	2 * 3 = 6	*=	x *= 3
/	División	6 / 2 = 3	/=	x /= 2
%	Módulo/Resto	5 % 2 = 1	%=	x %= 2
**	Exponenciación	2 ** 3 = 8		

Los operadores de comparación comparan dos expresiones y devuelven un valor booleano `true` o `false`:

Operador	Nombre	Descripción	Ejemplo	Resultado
==	Igualdad	Compara si dos valores son iguales	5 == 5	true
!=	Desigualdad	Compara si dos valores son distintos	5 != 4	true
>	Mayor que	Compara si el valor de la izquierda es mayor que el de la derecha	5 > 4	true
<	Menor que	Compara si el valor de la izquierda es menor que el de la derecha	4 < 5	true
>=	Mayor o igual	Compara si el valor de la izquierda es mayor o igual que el de la derecha	5 >= 5	true
<=	Menor o igual	Compara si el valor de la izquierda es menor o igual que el de la derecha	4 <= 4	true

Ejemplos:

```
1  puts 1 == 1 # true
3  puts 2 > 1 # true
4  puts 1 < 2 # true
5  puts 2 >= 2 # true
6  puts 1 <= 2 # true
```

Estos operadores funcionan con tipos numéricos y cadenas de texto, permitiendo realizar comparaciones en cualquier tipo de dato.

Operador	Significado	Ejemplo	Resultado
==	Igual	1 == 1	verdadero
!=	Distinto	1 != 2	verdadero
>	Mayor que	2 > 1	verdadero
<	Menor que	1 < 2	verdadero
>=	Mayor o igual	2 >= 2	verdadero
<=	Menor o igual	1 <= 2	verdadero
==	Igual	"hola" == "adiós"	falso
!=	Distinto	"hola" != "hola"	falso

Los operadores lógicos realizan operaciones booleanas a nivel de bits sobre expresiones logicas:

Operador	Nombre	Descripción	Ejemplo	Resultado	Detalles
&&	AND lógico	Devuelve true si ambos operandos son verdaderos	x && y	true si x = true, y = true	Funciona como una puerta AND a nivel de bits
||	OR lógico	Devuelve true si al menos uno de los operandos es verdadero	x || y	true si x = true, y = false	Funciona como una puerta OR a nivel de bits
!	NOT lógico	Niega una expresión lógica	!x	false si x = true	Invierte el valor booleano del operando

Ejemplos:

```
1 puts true && true # true
2 puts true && false # false
3 puts true || false # true
4 puts !true # false
```

Con operadores lógicos podemos construir expresiones booleanas complejas:

```
1 puts (1 == 1 && 2 > 1) || !(2 < 1) # true
```

Crystal permite manipular bits individuales de valores enteros mediante operadores bit a bit:

Operador	Nombre	Descripción	Ejemplo	Resultado	Detalles
&&	AND lógico	Devuelve true si ambos operandos son verdaderos	x && y	true si x = true, y = true	Funciona como una puerta AND a nivel de bits
||	OR lógico	Devuelve true si al menos uno de los operandos es verdadero	x || y	true si x = true, y = false	Funciona como una puerta OR a nivel de bits
!	NOT lógico	Niega una expresión lógica	!x	false si x = true	Invierte el valor booleano del operando

Para entender mejor cómo estos operadores manipulan los bits individuales de los valores, observemos cómo funcionan a nivel binario:

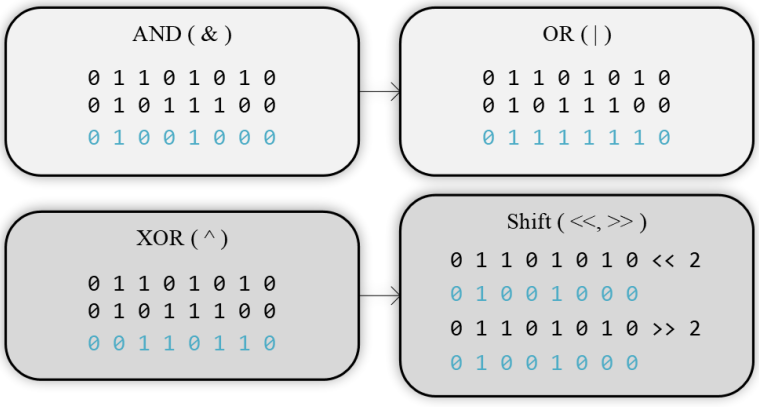


Gráfico 42 - Visualización de operaciones bit a bit y sus resultados en binario.
(Fuente: Propia)

Veamos ejemplos de su uso:

```
1  puts 5 & 3 # 1 (0101 & 0011)
2  puts 5 | 3 # 7 (0101 | 0011)
3  puts 5 ^ 3 # 6 (0101 ^ 0011)
4  puts ~5 # -6 (~0101)
5  puts 5 << 2 # 20 (0101 << 2)
6  puts 5 >> 2 # 1 (0101 >> 2)
```

Los operadores bit a bit nos permiten manipular y combinar valores enteros a muy bajo nivel. Cuando se combinan varios operadores en una expresión, se evalúan siguiendo un orden de prioridad. El orden de prioridad de mayor a menor es:

Prioridad	Operadores	Tipo	Descripción
1	+, -, !	Unarios	Operan sobre un solo operando
2	**	Exponenciación	Potencia o raíz cuadrada
3	*, /, %	Aritméticos	Multiplicación, división y módulo
4	+, -	Aritméticos	Suma y resta
5	<<, >>	Desplazamiento de bits	Desplaza bits a izquierda o derecha
6	<, >, <=, >=	Comparación	Menor, mayor, menor igual, mayor igual
7	==, !=	Comparación	Igualdad y desigualdad
8	&	Bit a bit	AND bit a bit
9	^	Bit a bit	XOR bit a bit
10	|	Bit a bit	OR bit a bit
11	&&	Lógico	AND lógico

Los paréntesis tienen la máxima prioridad y pueden usarse para anular la prioridad predefinida.

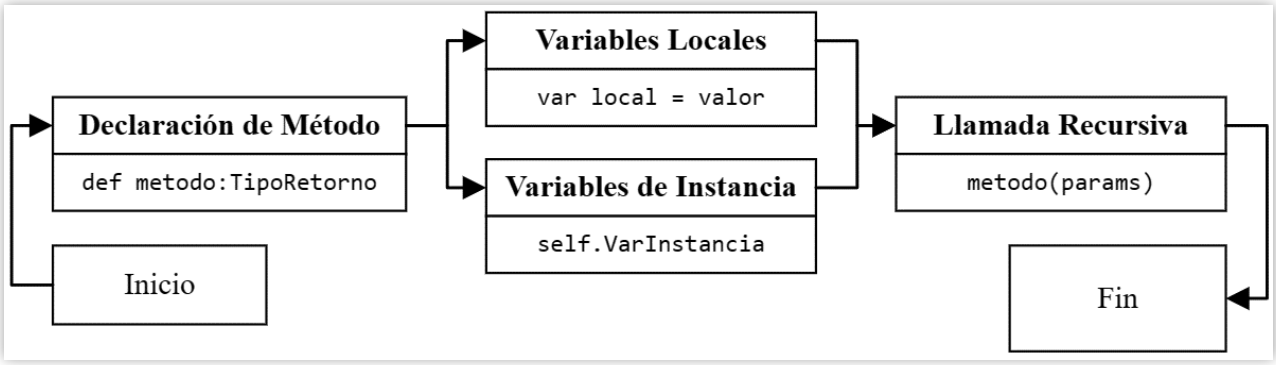


Gráfico 43 – Diagrama de flujo del ciclo de vida y alcance de un método en Crystal.
(Fuente: Propia)

El operador ternario (`?:`) permite evaluar una condición y devolver un valor u otro dependiendo de si es verdadera o falsa:

```
1  puts 1 == 1 ? "verdadero" : "falso" # imprime verdadero
```

Equivale a:

```
1  if 1 == 1
2      puts "verdadero"
```

```
3  else
4      puts "falso"
5  end
```

Puede encadenarse para evaluar múltiples condiciones:

```
1  puts age >= 18 ? "Mayor de edad" : (age > 0 ? "Menor de edad" : "Edad inválida")
```

El operador ternario permite escribir código condicional de manera concisa en una sola línea.

Los operadores de rangos permiten crear fácilmente rangos de valores:

- **..** Crea un rango inclusive.
- **...** Crea un rango exclusive.

Por ejemplo:

```
1  (1..5) # [1, 2, 3, 4, 5]
2  (1...5) # [1, 2, 3, 4]
```

Pueden combinarse con `case`:

```
1  case n
2  when 1..5
3    puts "Entre 1 y 5"
4  when 6..10
5    puts "Entre 6 y 10"
6  else
7    puts "Fuera de rango"
8  end
```

Y también iterarse:

```
1  (1..5).each do |i|
2      puts i
3  end
```

Esto permite trabajar fácilmente con rangos de valores en Crystal.

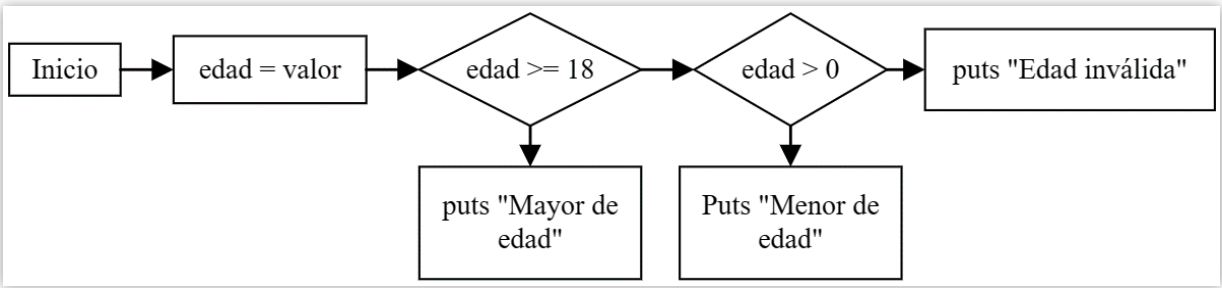


Gráfico 44 – Validación de edad usando operadores de comparación.
(Fuente: Propia)

Crystal incluye algunos operadores espaciales inspirados en Ruby:

- **<=>** Operador espacial combinado. Devuelve 0 si son iguales, -1 si el de la izquierda es menor, y 1 si es mayor.
- **==>** Operador espacial ordenado. Devuelve true si el de la izquierda es menor o igual.

Ejemplos:

```
1 <=> 1 # 0
1 <=> 2 # -1
2 <=> 1 # 1
1 ==> 1 # true
1 ==> 2 # true
2 ==> 1 # false
```

Estos operadores permiten realizar comparaciones espaciales de manera concisa.

Operador	Función	Ejemplo	Resultado
<=>	Comparación combinada	1 <=> 1	0
<=>	Comparación combinada	2 <=> 1	1
==>	Orden espacial	1 ==> 1	verdadero
==>	Orden espacial	2 ==> 1	falso

Para comprobar tipos de variables y valores, Crystal proporciona dos operadores:

- **is_a?** Comprueba si un objeto es instancia de un tipo.
- **responds_to?** Comprueba si un objeto responde a un método.

Estos operadores son muy útiles para comprobar tipos en tiempo de ejecución.

Operador	Utilidad	Ejemplo	Resultado
is_a?	Verifica tipo de objeto	"hola".is_a?(String)	verdadero
is_a?	Verifica tipo de objeto	5.is_a?(String)	falso
responds_to?	Verifica si responde a método	"hola".responds_to?(:upcase)	verdadero
responds_to?	Verifica si responde a método	5.responds_to?(:upcase)	falso

2.5. Control de flujo

El control de flujo se refiere a la forma en que un programa ejecuta las instrucciones secuencialmente o decide saltar partes del código en base a condiciones. Crystal proporciona construcciones típicas de control de flujo como if/else, unless, case, while, until y for que permiten controlar el flujo de ejecución. La sentencia if/else permite ejecutar un bloque de código si una condición es verdadera, y opcionalmente ejecutar otro bloque si es falsa:

```
a = 10
b = 5
if a > b
  puts "a es mayor que b"
```

```
5  else
6    puts "a es menor o igual que b"
7  end
```

Esto imprimirá "a es mayor que b" porque la condición a > b es verdadera. El else es opcional, por lo que se puede escribir simplemente:

```
1  if a > b
2      puts "a es mayor que b"
3  end
```

Las condiciones pueden ser cualquier expresión que devuelva un valor booleano true o false. Se pueden encadenar varios if/else if para probar múltiples condiciones:

```
1  a = 15
2  if a > 20
3    puts "a es mayor que 20"
4  elsif a > 10
5    puts "a es mayor que 10 pero menor o igual a 20"
6  else
7    puts "a es menor o igual a 10"
8  end
```

Esto imprimirá "a es mayor que 10 pero menor o igual a 20".

Construcción	Ejemplo	Explicación
if	`crystal   if x > 5    puts "x es mayor que 5"   end`	Evalúa la condición después de if. Si es verdadera, ejecuta el código en el bloque.
else if	`crystal   if x > 10    puts "x es mayor que 10"   elsif x > 5    puts "x está entre 5 y 10"   end`	Evalúa las condiciones if/elsif en orden. Ejecuta el primer bloque con condición verdadera.
else	`crystal   if x > 10    puts "x es mayor que 10"   else    puts "x es menor o igual a 10"   end`	Se ejecuta si todas las condiciones previas son falsas. Captura el caso por defecto.

unless es lo opuesto a if: ejecuta un bloque de código si la condición es falsa:

```
1  a = 10
2  unless a > 20
3    puts "a NO es mayor que 20"
4  end
```

Al igual que if, se puede añadir un else:

```
1  unless a > 20
```


```
2     puts "a es menor o igual a 20"
3 else
4     puts "a es mayor que 20"
5 end
```

case permite comparar una expresión contra múltiples valores posibles de forma más limpia que encadenar muchos if/elsif:

```
1 a = 15
2 case a
3 when 10
4   puts "a es 10"
5 when 20
6   puts "a es 20"
7 else
8   puts "a no es ni 10 ni 20"
9 end
```

Esto imprimirá "a no es ni 10 ni 20". Los when funcionan como el if, probando matches con el caso. Se ejecuta el primer when que iguale. El case puede usarse sin el else, y si ningún when coincide, simplemente no se ejecuta nada. El bucle while ejecuta un bloque mientras una condición sea verdadera:

```
1 i = 0
2 while i < 10
3   puts i
4   i += 1
5 end
```

Esto imprimirá los números del 0 al 9. Cuidado con los bucles infinitos si la condición siempre es verdadera. Se debe actualizar la variable de control en algún punto para que eventualmente la condición se vuelva falsa.

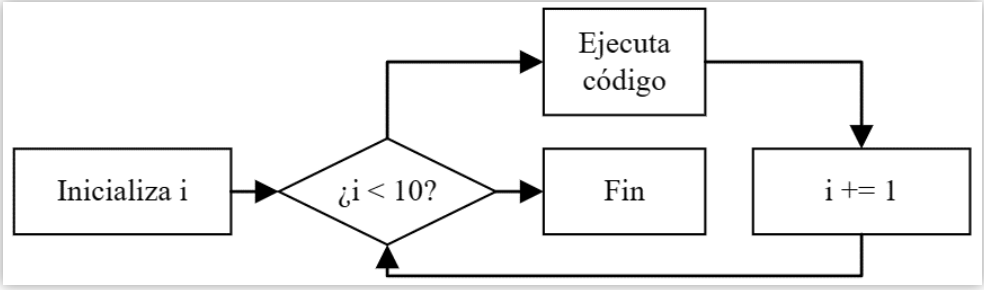


Gráfico 45 – Diagrama de flujo de un bucle while en Crystal.
(Fuente: Propia)

while comprueba la condición antes de la primera iteración, por lo que es posible que el bloque nunca se ejecute:

```
1 i = 10
2 while i < 5
```

```
3   puts i
4   i += 1
5 end
```

En este caso no se imprime nada porque la condición empieza siendo falsa. until es lo opuesto a while: ejecuta hasta que una condición sea verdadera:

```
1 i = 0
2 until i > 10
3   puts i
4   i += 1
5 end
```

Al igual que while, es importante no crear un bucle infinito y actualizar la variable de control en algún punto.

Loop	Descripción	Ejemplo
while	Ejecuta código mientras la condición sea verdadera.	`crystal   while x < 10   x += 1   puts x   end`
until	Ejecuta código hasta que la condición sea verdadera.	`crystal   until x == 0   x -= 1   puts x   end`

El bucle for proporciona una sintaxis concisa para iterar sobre colecciones:

```
1 (1..10).each do |i|
2     puts i
3 end
```

Esto imprime del 1 al 10. La ventaja es que no hace falta inicializar una variable, actualizarla y comprobar la condición manualmente. for itera sobre cualquier objeto que responda al método #each, incluyendo rangos, arrays, hashes y canales.

Otra opción es usar for con un bloque con argumentos:

```
1 for i in 1..10
2     puts i
3 End
```

El bloque es pasado a #each detrás de escenas. break sale inmediatamente de un bucle:

```
1 while true
2     puts "bucle infinito"
3     break
4 end
5 puts "después del bucle"
```

Esto imprime "bucle infinito" una vez y luego "después del bucle".

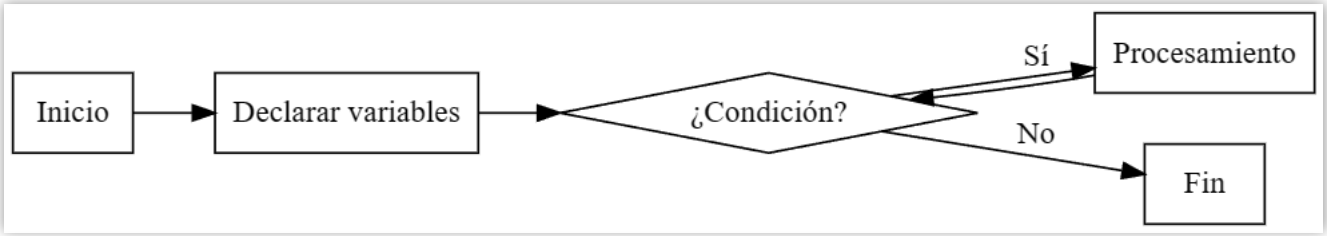


Gráfico 46 – Diagrama de flujo básico de una estructura de control con condición.
(Fuente: Propia)

Sin el break, sería un bucle infinito. break acepta un valor de retorno:

```
1  x = nil
2  while true
3    x = "retorno"
4    break x
5  end
6  puts x # imprime "retorno"
```

next salta a la siguiente iteración de un bucle sin ejecutar el resto del cuerpo:

```
1  for i in 1..5
2      if i == 3
3          next
4      end
5      puts i
6  end
```

Esto imprime 1, 2, 4 y 5. Cuando i es 3, next salta a la siguiente iteración.

Crystal provee begin/rescue/ensure para manejo de excepciones:

```
1  begin
2      # código
3  rescue Excepción1
4      # handle error
5  rescue Excepción2
6      # handle error
7  else
8      # sin errores
9  ensure
10     # siempre se ejecuta
11 end
```

El bloque begin encapsula código que puede potencialmente levantar una excepción.

Bloque	Propósito	Ejemplo

begin	Encapsula código a ejecutar que puede causar errores.	crystal begin risky_method() end
rescue	Define manejo de errores si ocurren.	crystal rescue ErrorType handle_error() end
else	Código para ejecutar si no hubo errores.	crystal else no_errors_found() end
ensure	Código que siempre se ejecuta al finalizar.	crystal ensure cleanup() end

Si se produce un error, se busca un bloque rescue que coincida con ese tipo de error. El primero que coincida se ejecuta. El bloque else se ejecuta sólo si no hubo errores. El bloque ensure siempre se ejecuta al final, tenga éxito o no. Útil para limpieza como cerrar archivos.

Ejemplo:

```
1  begin
2      file = File.open("archivo.txt")
3  rescue File::NotFoundError
4      puts "archivo no encontrado"
5  else
6      # procesar archivo
7  ensure
8      file.close if file
9  end
```

Esto intenta abrir un archivo, pero maneja el error si no existe, y siempre cierra el archivo si fue abierto correctamente.

- redo reejecuta la iteración actual de un bucle sin reevaluar la condición.
- retry reevalúa la condición y comienza la siguiente iternación.
- next salta a la siguiente iteración sin ejecutar el resto del cuerpo.

Ejemplo:

```
1  retries = 0
2  begin
3    # intenta algo
4  rescue
5    retries += 1
6    if retries < 3
7      retry # reintenta
8    else
9      redo # reejecuta iteración
10   end
11 end
```

Esto reintenta hasta 3 veces antes de simplemente reejecutar la iteración actual.

Construcción	Efecto	Caso de uso
retry	Vuelve a evaluar la condición y reitera el bucle.	Reintentar tras un error transitorio.
redo	Re-ejecuta el cuerpo del bucle.	Reprocesar cuando se cambia estado.
next	Salta a la siguiente iteración.	Evitar código en una iteración.

El control de flujo es esencial en cualquier lenguaje de programación y Crystal incorpora todas las funcionalidades comunes para controlar cómo se ejecuta un programa basado en lógica condicional e iterativa.

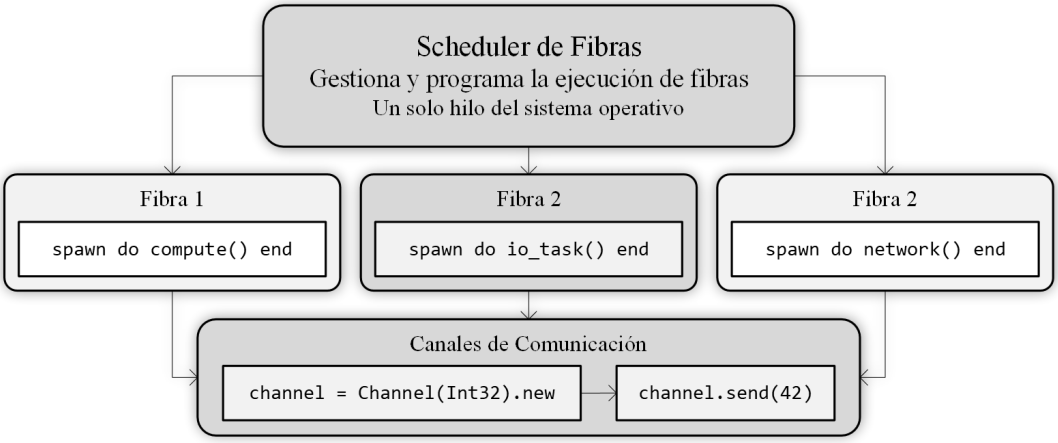


Gráfico 47 - Sistema de control de flujo concurrente en Crystal.
(Fuente: Propia)

Este modelo de concurrencia basado en fibras es una de las características más potentes de Crystal, permitiendo escribir código concurrente seguro y eficiente. Con estos conceptos fundamentales de control de flujo, podemos ahora adentrarnos en cómo Crystal organiza y estructura el código a través de métodos y funciones.

2.6 Métodos y funciones

Los métodos y funciones son bloques de código reutilizables que realizan una tarea específica. Permiten encapsular lógica compleja y abstraerla para ser utilizada fácilmente en cualquier parte del programa.

En Crystal existen algunas diferencias entre métodos y funciones:

Característica	Métodos	Funciones
Definición	Dentro de una clase	Independientes, fuera de clases
Acceso a variables	Pueden acceder a variables de instancia de la clase	No tienen acceso a variables de instancia
Efectos secundarios	Pueden modificar estado de la instancia	Por lo general no tienen efectos secundarios

Veamos en detalle cómo definir y utilizar tanto métodos como funciones en Crystal. Los métodos se definen dentro del cuerpo de una clase, usando la palabra clave `def`:

```
1  class Usuario
2      def saludar
3        puts "Hola!"
```

```
4     end
5   end
```

El nombre del método va después de `def`, seguido por paréntesis con una lista opcional de parámetros. El cuerpo del método se encierra entre `end`.

Los métodos pueden aceptar parámetros, que son variables locales disponibles sólo dentro del método:

```
1  class Usuario
2      def saludar(nombre)
3        puts "Hola #{nombre}!"
4      end
5    end
```

También pueden devolver un valor usando `return`:

```
1  class Usuario
2      def get_nombre
3        return @nombre
4      end
5    end
```

Para llamar a un método, se accede a una instancia de la clase y se utiliza la sintaxis `objeto.metodo()`:

```
1  usuario = Usuario.new
2  usuario.saludar("Juan")
3  nombre = usuario.get_nombre
```

Los parámetros requeridos por el método se pasan entre paréntesis. Los métodos también pueden ser llamados con la sintaxis `self.metodo()` desde dentro de la misma clase.

Getters y setters son métodos especiales para obtener y establecer el valor de las variables de instancia. En vez de acceder directamente a la variable `@nombre`, se define un getter:

```
1  class Usuario
2      def initialize(@nombre)
3      end
4      def nombre
5        @nombre
6      end
7    end
```

Y un setter equivalente para asignar el valor:

```
1  class Usuario
2      # ..
3      def nombre=(nuevo_nombre)
```


```
4       @nombre = nuevo_nombre
5     end
6   end
```

Estos se utilizan así:

```
1 usuario = Usuario.new("Juan")
2 puts usuario.nombre #getter
3 usuario.nombre = "Maria" #setter
```

Forma	Ejemplo	Tipo de método	Descripción
objeto.metodo	usuario.saludar	Instancia	Invoca el método saludar() del objeto usuario
self.metodo	self.get_nombre	Clase	Invoca el método get_nombre() de la clase actual
Clase.metodo	Usuario.get_cantidad	Clase	Invoca el método de clase get_cantidad() de Usuario
objeto.nombre	usuario.nombre	Getter	Accede al getter nombre del objeto usuario
objeto.nombre=	usuario.nombre = "Juan"	Setter	Invoca el setter nombre=() del objeto usuario
super(args)	super(nombre)	Superclase	Invoca el método de la superclase con los arguments

Crystal también permite usar la sintaxis `property` para generar getters y setters automáticamente:

```
1 class Usuario
2     property nombre
3   end
```

Además de métodos de instancia, una clase puede tener métodos de clase, definidos con `self.metodo`:

```
1 class Usuario
2     def self.get_cantidad
3       # cuenta y devuelve la cantidad de usuarios
4     end
5   end
```

Estos métodos se llaman directamente sobre la clase, sin necesidad de una instancia:

```
1 puts Usuario.get_cantidad()
```

Útiles para constructores personalizados, operaciones sobre la clase, constantes, etc. Por defecto, todos los métodos son públicos y se pueden llamar desde cualquier parte del código. Para declarar un método privado, se antepone `private` al definirlo:

```
1 class Usuario
2     def publico
3     end
4     private def privado
```

```
5     end
6   end
```

Los métodos privados sólo pueden ser usados por otros métodos dentro de la misma clase. Esto ayuda a encapsular la implementación interna. A diferencia de los métodos, las funciones se definen fuera de una clase:

```
1  def saludar(nombre)
2      puts "Hola #{nombre}!"
3    end
```

Al igual que los métodos, pueden aceptar argumentos y devolver un valor. Las funciones son llamadas directamente por su nombre:

```
1  saludar("Maria")
```

En Crystal, los parámetros se pasan siempre por valor. Esto significa que al llamar un método, se copia el valor del argumento. Sin embargo, si el parámetro es un objeto, la copia será su referencia en memoria. Por lo tanto los cambios hechos al objeto sí serán visibles fuera.

Los tipos como enteros, strings, booleanos se pasan realmente por valor. Los objetos y arreglos se pasan por referencia. Esto se puede modificar explicitamente con las palabras **value** y **reference**:

```
1  def incrementar(value numero)
2      numero += 1 # no cambia el argumento original
3    end
4    def incrementar(reference numero)
5      numero += 1 # sí cambia el argumento
6    end
```

Tipo	Ejemplo	Descripción
Posicionales	def saludar(nombre, edad)	Requeridos en orden
Default	def saludar(nombre="Invitado")	Valores por defecto opcionales
Nombrados	saludar(nombre: "Juan")	Se pasa nombre de parámetros
Splats	def imprimir(*mensajes)	Empaqueta argumentos extras
Bloques	def ejecutar(&bloque)	Pasa bloque de código como Proc
Restricciones	def suma(x : Int32, y : Float64)	Restringe tipos permitidos

Se pueden definir valores por defecto para los argumentos:

```
1  def saludar(nombre="invitado")
2      puts "Hola #{nombre}"
3    end
4    saludar() # Hola invitado
5    saludar("Maria") # Hola Maria
```


Si se llama al método sin ese argumento, se usará el valor por defecto. Al llamar a un método, se pueden especificar los nombres de los argumentos:

```
1  def saludar(nombre, edad)
2      puts "Hola #{nombre} de #{edad} años"
3    end
4    saludar("Juan", 20)
5    saludar(nombre: "Maria", edad: 22)
```

Esto permite mayor legibilidad y omitir argumentos. Si un método acepta un número variable de argumentos, se puede usar el operador splat `*`:

```
1  def imprimir(*mensajes)
2      mensajes.each { |m| print m }
3    end
4    imprimir("Hola", "mundo", "!") # Hola mundo!
```

Esto empaqueta los argumentos extras en un arreglo. Crystal permite también pasar un bloque de código como argumento utilizando `&`:

```
1  def ejecutar(&bloque)
2      bloque.call
3    end
4    ejecutar { print "Hola " }
```

El bloque se convierte en un objeto `Proc` que se puede ejecutar. Los métodos en Crystal deben declarar explícitamente el tipo de dato que retornan:

```
1  def get_nombre : String
2      #...
3    End
```

Si no se especifica, el compilador infiere el tipo según el valor retornado. Esto ayuda a evitar bugs y mejora el chequeo de tipos.

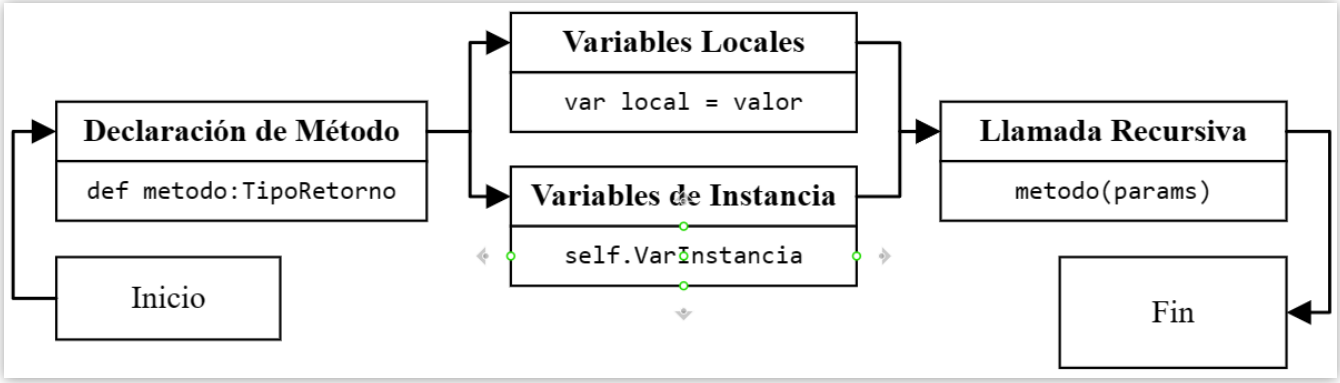


Gráfico 48 – Ciclo de vida y visibilidad de variables y métodos en Crystal.
(Fuente: Propia)

Las variables definidas dentro de un método sólo existen en ese contexto local. No estarán disponibles fuera del método:

```
def saludar(nombre)
    edad = 20 # variable local
    puts nombre + " tiene " + edad + " años"
  end
  edad # undefined local variable
```

Las variables de instancia sí están disponibles en todos los métodos, al usar **self.variable**. Crystal permite que los métodos se llamen a sí mismos, conocido como recursión. Esto es útil en algunos algoritmos y estructuras de datos como árboles. Por ejemplo, para calcular el factorial de un número:

```
def factorial(n)
    if n <= 1
      1
    else
      n * factorial(n-1)
    end
  end
  factorial(5) # 120
```

La recursión debe tener siempre una condición de terminación, o resultará en un loop infinito. Crystal permite definir métodos y funciones dentro de otros métodos o funciones. Estas funciones internas sólo están disponibles en ese contexto.

```
defCOOKIES per user:
  def generate_key
    # generates random key
  end
  def save_cookie(value)
    key = generate_key
    # save cookie with key
  end
end
```

Métodos	Funciones
Definidos con **def** dentro de clase	Definidos con **def** fuera de clase
Forman parte de la clase/instancia	Independientes y globales
Se llaman con **objeto.metodo**	Se llaman por su nombre directo
Parámetros posicionales, nombrados, splats, bloques	Parámetros posicionales, valores default
Restricciones de tipos, públicos/privados	Siempre públicas
Acceso a estado con **self** y **@variable**	Ámbito local solamente
Permite recursión	Permite anidamiento

Encapsulan comportamiento	Encapsulan funcionalidad

2.7 Clases y objetos

Las clases y los objetos son los pilares fundamentales de la programación orientada a objetos (POO) en Crystal.

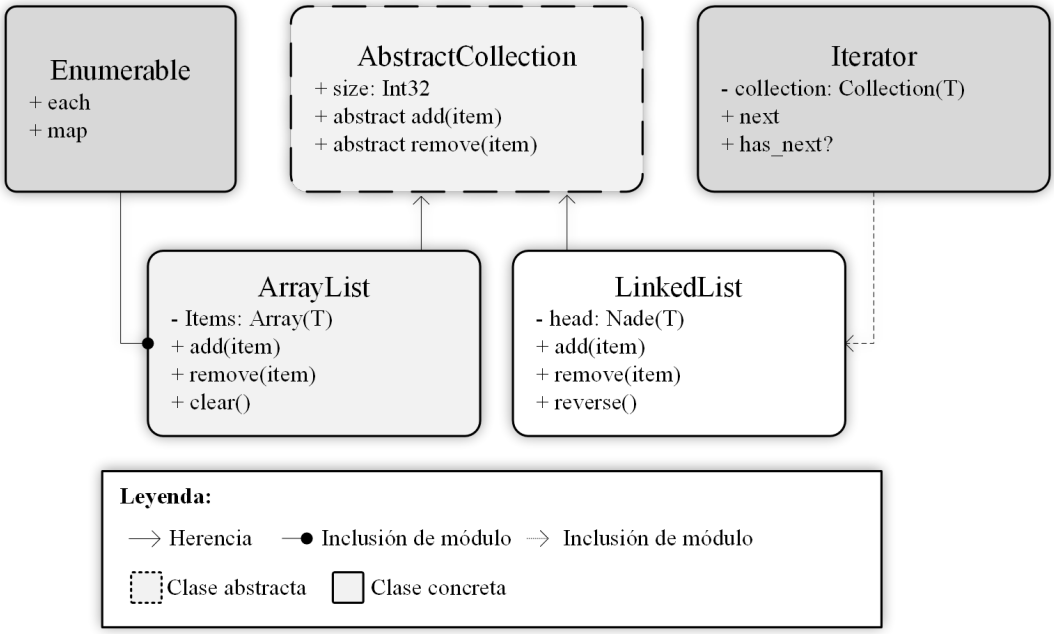


Gráfico 49 - Ecosistema de Relaciones entre Clases, Módulos y Herencia en Crystal.
(Fuente: Propia)

Permiten modelar elementos del mundo real y encapsular datos y comportamientos en entidades reutilizables. Una clase es una plantilla para crear objetos. Define las características y comportamientos que tendrán los objetos creados a partir de ella. En otras palabras, una clase es como un molde para hacer galletas con una forma definida.

Las clases contienen:

- Atributos (variables de instancia) - Representan las propiedades y estados de los objetos.
- Métodos - Definen los comportamientos y acciones que puede realizar un objeto.

Por ejemplo, podemos definir una clase **Persona**:

```
1  class Persona
2      def initialize(@nombre : String, @edad : Int32)
3      end
4      def saludar
5        puts "Hola! Soy #{@nombre} y tengo #{@edad} años."
6      end
7    end
```

Esta clase **Persona** tiene dos atributos, **@nombre** y **@edad**, y un método **saludar**. Aún no hemos creado ningún objeto **Persona**, solo definimos la plantilla. Para crear un objeto a partir de una clase, se utiliza la palabra clave **new**:

```
1  persona1 = Persona.new("Juan", 25)
```


```
2  persona2 = Persona.new("María", 30)
```

Ahora tenemos dos objetos **persona1** y **persona2** del tipo **Persona**, cada uno con sus propios valores para los atributos nombre y edad. Podemos llamar al método **saludar**:

```
1  persona1.saludar # Hola! Soy Juan y tengo 25 años.
2  persona2.saludar # Hola! Soy María y tengo 30 años.
```

Tipo	Definición	Declaración	Acceso	Ejemplo
Atributo	Variable de instancia que representa un estado	@nombre	objeto.atributo	@edad
Método	Comportamiento que realiza una acción	def metodo	objeto.metodo	def saludar()

Los objetos son **instancias** de una clase y contienen:

- Sus propios valores para los atributos definidos en la clase.
- Las definiciones de los métodos, para poder ejecutarlos.

La clase es la definición general, el objeto es la instancia específica. Los atributos son las variables que pertenecen a cada objeto, y representan sus propiedades y estado. Se definen dentro de la clase, precedidos por un `@`:

```
1  class Persona
2      @nombre
3      @edad
4    end
```

Se asignan valores en el método `initialize` al crear el objeto:

```
1  class Persona
2      def initialize(@nombre, @edad)
3      end
4    end
5    persona = Persona.new("Juan", 25)
```

Se accede a los atributos con la sintaxis `objeto.atributo`:

```
1  puts persona.nombre # Juan
2  puts persona.edad # 25
```

Los atributos también se pueden definir con un valor por defecto:

```
1  class Persona
2      @nombre : String
3      @edad : Int32 = 0
4      def initialize(@nombre)
5      end
```


```
6   end
```

Esto permite crear el objeto sin pasar la edad:

```
1   persona = Persona.new("Juan")
2   puts persona.edad # 0
```

Los métodos definen las acciones que puede realizar un objeto. Se declaran con `def` dentro de la clase:

```
1   class Persona
2       def saludar
3         puts "Hola!"
4       end
5     end
```

Pueden aceptar parámetros y devolver valores:

```
1   class Persona
2       def saludar(nombre)
3         "Hola #{nombre}!"
4       end
5     end
6     persona.saludar("Juan") # "Hola Juan!"
```

Dentro de los métodos se puede acceder a los atributos del objeto con `@atributo`:

```
1   class Persona
2       def initialize(@nombre)
3       end
4       def saludar
5         "Hola! Soy #{@nombre}"
6       end
7     end
8     persona = Persona.new("Juan")
9     persona.saludar # "Hola! Soy Juan"
```

El método `initialize` es un constructor, se ejecuta cuando se crea un nuevo objeto:

```
1   class Persona
2       def initialize(nombre, edad)
3         @nombre = nombre
4         @edad = edad
5       end
6     end
```

Se utiliza para inicializar los atributos del objeto. Si no se define un `initialize`, Crystal creará uno por defecto sin parámetros. La herencia permite que una clase herede atributos y métodos de otra. La clase base es la clase padre, la clase derivada es la clase hija.

Sintaxis:

```
class ClaseHija < ClasePadre
end
```

Para comprender mejor cómo interactúan las clases, módulos y herencia en Crystal, observemos la siguiente representación visual que ilustra estas relaciones y su estructura:

Ejemplo:

```
class Empleado
    def initialize(@nombre, @salario)
    end
    def bono(porcentaje)
      @salario * porcentaje / 100
    end
  end
  class Gerente < Empleado
    def initialize(@nombre, @salario, @departamento)
    end
  end
  gerente = Gerente.new("Juan", 1000, "TI")
  puts gerente.bono(10) # 100
```

`Gerente` hereda los atributos y métodos de `Empleado`. La herencia permite reutilizar código y modelar relaciones `es un` (un gerente es un empleado).

La sobreescritura permite que una clase hija redefina métodos heredados de la clase padre.

```
class Persona
    def saludar
      "Hola!"
    end
  end
  class Trabajador < Persona
    def saludar
      "Buen día!"
    end
  end
  persona = Persona.new
  persona.saludar # "Hola!"
  trabajador = Trabajador.new
  trabajador.saludar # "Buen día!"
```

`Trabajador` sobreescribe el método `saludar`. Son clases que no se pueden instanciar directamente y sirven de base para la herencia. Se definen con `abstract class`:

```
abstract class Figura
    # métodos comunes
  end
  class Rectangulo < Figura
    # métodos específicos
  end
```

`Figura` no se puede instanciar, solo sus subclases. Los módulos permiten agrupar métodos, clases y constantes relacionadas, para encapsular funcionalidades. Se definen con `module`:

```
module Gemometria
    PI = 3.14
    def self.area_circulo(radio)
      PI * radio ** 2
    end
  end
```

Se incluyen con `include`:

```
class Circulo

    include Geometria
    def initialize(@radio)
    end
    def area
```

```
6       Geometria.area_circulo(@radio)
7     end
8   end
```

Los módulos permiten crear namespaces y proporcionar mixins.

App	Library
`module App   def run   # código end`	`module Library   def run   # código end`
Permite separar `run`	Permite separar `run`
Sin colisión de nombres	Sin colisión de nombres

La visibilidad controla el acceso a clases, métodos y atributos. Hay tres niveles:

Modificador	Nivel de Visibilidad	Accesible desde	Descripción
public	Público	Cualquier clase	Accesible desde cualquier parte del código
protected	Protegido	La clase y subclases	Accesible en la clase declarada y subclases
private	Privado	Solo dentro de la clase	Solo accesible dentro de la clase declarada

Se definen con:

```
1  class Persona
2      public # atributos y métodos son públicos
3      protected
4      private
5    end
```

Todo lo definido después de `protected` es protegido. Después de `private` es privado.

Modificador	Acceso	Ejemplo	Descripción
Public	Cualquier lado	public def metodo	Accesible desde cualquier lugar
Protected	Clase y subclases	protected def metodo	Solo clase actual y subclases
Private	Solo dentro de la clase	private def metodo	Solo accesible en la clase

Las propiedades son métodos getter y setter creados automáticamente para los atributos:

```
1  class Persona
2    @nombre : String
3    def nombre
4      @nombre
5    end
6    def nombre=(val)
7      @nombre = val
```


```
8     end
9   end
10  persona = Persona.new
11  persona.nombre = "Juan" # Setter
12  puts persona.nombre # Getter
```

Se simplifica con:

```
1   class Persona
2     property nombre
3   end
```

Esto genera el getter y setter automáticamente. Propiedades con solo getter:

```
1   class Persona
2     getter nombre
3   end
```

Propiedades solo setter:

```
1   class Persona
2     setter nombre
3   end
```

Se declaran con `CONSTANT` y no cambian:

```
1   class Matematicas
2     PI = 3.1416
3     EULER = 2.7182
4   end
```

No se pueden cambiar:

```
1   Matematicas::PI = 0 # Error
```

Se accede con `Clase::CONSTANTE`. Son similares a clases pero de valor. Se definen con `struct`.

```
struct Punto
  getter x, y
  def initialize(@x, @y)
  end
end
```

Se asignan por valor no referencia:

```
a = Punto.new(1, 2)
b = a
b.x = 10
puts a.x # 1
```

Cambiar `b` no afecta a `a`.

Característica	Clase	Módulo
Propósito	Molde para crear objetos	Agrupar y encapsular código
Declaración	class Clase	module Modulo
Contenido	Atributos, métodos	Métodos, constantes, clases
Instanciación	Se instancia con .new	No se puede instanciar

Herencia	Permite herencia	Provee mixins con include

2.8 Módulos y espacios de nombres

Los módulos y espacios de nombres son una parte fundamental de Crystal que permiten organizar el código en diferentes archivos y directorios.

Un módulo en Crystal es similar a una librería o paquete en otros lenguajes. Permite agrupar tipos, funciones, macros y constantes relacionadas en un solo archivo con extensión `.cr`. Por ejemplo, podemos tener un módulo `math.cr`:

```
1  module Math
2    PI = 3.14159
3    def self.sin(x)
4      # ...
5    end
6    def self.cos(x)
7      # ...
8    end
9  end
```

Y luego requerirlo desde otro archivo:

```
1  require "./math"
2  Math.sin(Math::PI/2) #=> 1.0
```

Los módulos proporcionan un espacio de nombres separado. Esto evita colisiones de nombres entre diferentes partes de la aplicación.

Los módulos también permiten encapsular código relacionado para ocultar detalles de implementación. Solo se hace público lo que se quiere exponer a otros módulos. Algunas ventajas de usar módulos:

Ventaja	Descripción	Beneficios
Organización del código	Permite agrupar tipos, clases y funciones relacionadas	Código mejor estructurado, fácil de entender y mantener
Encapsulamiento	Oculta detalles de implementación exponiendo solo una interfaz pública	Reduce acoplamiento entre módulos
Espacios de nombres	Evita colisiones de nombres entre diferentes partes de la aplicación	Elimina un tipo común de errores y dificultades de mantenimiento
Reutilización	Permite requerir y reusar un módulo en múltiples proyectos	Promueve la modularidad y reduce duplicación de código

Los módulos facilitan construir programas grandes manteniendo el código organizado, evitando problemas y permitiendo reutilizar funcionalidades.

Los módulos pueden anidarse arbitrariamente:

```
1  module Parent
```

```
  module Child
    # ...
  end
end
```

Y acceder a ellos con la notación de puntos:

```
Parent::Child::some_method
```

Esto amplía las posibilidades de organización del código.

Los módulos definen un ámbito local. Las constantes, tipos, métodos y macros definidos dentro de un módulo solo están disponibles dentro de ese módulo por defecto. Para hacer que algo sea público y accesible desde otros módulos, se debe declarar con `public`:

```
module ModuleA
  public PI = 3.14
  def public_method
    # ...
  end
end
```

Todo lo demás será privado al módulo por defecto.

Accesibilidad	Métodos	Constantes	Otros
Público	`def method`	`CONST = value`	Alias con `alias`

Protegido	`protected def method`	-	Solo heredable dentro del módulo
Privado	`private def method`	`private CONST = value`	Solo disponible en el módulo

Para usar un módulo, primero hay que requerirlo con `require`:

```
require "module" # require relativo
require "library" # require absoluto
```

Esto incluirá el módulo y permitirá acceder a sus métodos y tipos públicos. También se puede dar un alias a un módulo requerido:

```
require "long/module/name" as Short
```

Y usar `Short` en lugar del nombre largo. Requerir un módulo lo carga una sola vez. Las siguientes requisiciones del mismo módulo no volverán a cargarlo. Para forzar la carga, independientemente de si ya fue requerido, se usa `load`:

```
load "module.cr" # recarga el módulo
```

Esto es útil durante el desarrollo para probar cambios sin tener que recompilar toda la aplicación.

Característica	require	load
Carga el módulo	Solo primera vez	Siempre lo recarga
Uso típico	Producción	Desarrollo
Rendimiento	Más rápido	Más lento
Dependencias	Resuelve transitorias	Requiere explicitas

Los módulos permiten definir espacios de nombres para evitar colisiones de nombres.

Por ejemplo, tanto `App` como `Library` pueden definir un método `run`:

```
# app.cr
module App
  def run
    # código de App
  end
end
# library.cr
module Library
  def run
    # código de Library
  end
end
```

Al requerirlos en diferentes módulos, sus nombres no colisionan:

```
require "./app"
```

```
2  require "./library"
3  App.run # ejecuta App.run
4  Library.run # ejecuta Library.run
```

Sin módulos, habría una colisión por el nombre `run`. Los módulos separan los espacios de nombres para evitar esto. Gracias al anidamiento de módulos, se pueden definir espacios de nombres anidados:

```
1  module Parent
2    class Client
3      # ...
4    end
5    module Child
6      class Client
7        # ...
8      end
9    end
10 end
```

Y acceder a cada cliente en su espacio de nombres correspondiente:

```
1  parent_client = Parent::Client.new
2  child_client = Parent::Child::Client.new
```

Esto permite tener clases con el mismo nombre, siempre que estén en módulos diferentes. Se puede crear un alias de un tipo de un módulo con `typeof`:

```
1  # Alias del tipo Parent::Child::Client
2  Client = typeof(Parent::Child::Client)
3  client = Client.new
```

Esto da acceso más simple a tipos en espacios de nombres anidados. Los módulos se pueden incluir dentro de una clase o módulo con `include`:

```
1  module Tools
2    def tool1
3      # ...
4    end
5  end
6  class Workshop
7    include Tools
8    def build
9      tool1
10     # ...
11   end
12 end
```


Esto mezcla el contenido público del módulo incluido dentro del espacio de nombres de la clase o módulo. Las inclusiones actúan de forma recursiva. Al incluir un módulo, también se incluyen sus inclusiones. La inclusión de módulos ayuda a:

Ventaja	Descripción	Beneficios
Reutilizar funcionalidad	Incluir módulos con herramientas útiles en distintas clases	Evita duplicar código
Desacoplar código	Extraer lógica común a módulos reusable	Reduce responsabilidades de
Organizar jerárquicamente	Diseñar módulos con un propósito claro	Sistema modular, fácil de entender

Un caso común de inclusión de módulos es para extender una clase de forma modular:

```
# lib/base_printer.cr
module BasePrinter
  def print
    # ...
  end
end
# lib/string_printer.cr
require "./base_printer"
class StringPrinter
  include BasePrinter
  # agrega funcionalidad específica para imprimir cadenas
end
```

De esta forma las clases quedan desacopladas y con preocupaciones separadas. Por convención, los archivos fuente de Crystal se colocan en un directorio `src`. Por ejemplo:

```
├── src
│   ├── myapp
│   │   ├── models
│   │   ├── views
│   │   └── controllers
│   └── utils
│       ├── string
│       └── numeric
```

Esto ayuda a organizar los módulos y espacios de nombres de la aplicación. El directorio `src` no es requerido, pero es una buena práctica seguir esta estructura. El compilador entiende este directorio especial y mapea los módulos Crystal a la estructura de archivos. Crystal analiza las dependencias entre módulos y define el orden correcto de requisición. Si un módulo depende de otro, no es necesario requerirlo explícitamente. El compilador incluirá los requisitos transitorios de forma automática. Por ejemplo, si `utils/string.cr` usa `utils/numeric.cr`, solo hace falta requerir el primero:

```
# utils/string.cr
require "./numeric"
module Utils
  module String
    # utiliza Utils::Numeric
  end
end
# app.cr
require "./utils/string"
# utils/numeric.cr fue requerido automáticamente
```

Esto simplifica los requerimientos y evita tener que conocer las dependencias transitorias. Los módulos y espacios de nombres son una parte integral de Crystal que permite escalar proyectos manteniendo un código ordenado y reutilizable. Usa módulos para dividir tu código, limitar visibilidad y agrupar lógicamente funcionalidades.

Conclusiones de este capítulo
Con este capítulo hemos sentado las bases del lenguaje Crystal explorando sus conceptos más fundamentales como la sintaxis, tipos de datos, variables, funciones y clases. A través de numerosos ejemplos y explicaciones detalladas hemos visto cómo utilizar las poderosas características de Crystal para declarar variables, controlar el flujo del programa, modelar clases y objetos, y organizar el código en módulos y espacios de nombres. A estas alturas ya deberías sentirte cómodo trabajando con los componentes esenciales de Crystal para empezar a dar forma a tus propios programas y aplicaciones. Si bien aún nos quedan por ver aspectos más avanzados del lenguaje, contamos con las herramientas básicas para resolver problemas reales de forma eficiente y elegante aprovechando la sintaxis expresiva de Crystal y su rendimiento nativo.

Ejercicios y Preguntas Para Resolver

Ahora que hemos explorado los fundamentos de Crystal, desde las estructuras básicas de control hasta conceptos más avanzados como macros, es momento de poner a prueba nuestra comprensión con los siguientes ejercicios y preguntas.

Para consolidar los conocimientos adquiridos en este capítulo, te presentamos una serie de ejercicios y preguntas organizados por nivel de dificultad. Cada ejercicio incluye el tiempo estimado de resolución y los conceptos principales que evalúa, brindándote la oportunidad de reforzar tu dominio de los temas clave.

(★☆☆) Ejercicios Básicos

Tiempo estimado: 5-10 minutos por ejercicio

#	Ejercicio	Conceptos Evaluados	Tiempo
1	Escribe un programa Crystal que calcule el área de un círculo dado su radio.	• Aritmética básica • Funciones	10 min
2	Crea un programa que tome dos números como entrada del usuario y muestre el mayor de los dos.	• Entrada de usuario • Condicionales	10 min

3	Implementa una función que tome una lista de números y devuelva la suma de todos ellos.	• Listas • Funciones	10 min

(★★☆) Ejercicios Intermedios

Tiempo estimado: 10-15 minutos por ejercicio

#	Ejercicio	Conceptos Evaluados	Tiempo
4	Escribe un programa que imprima los primeros 10 números de la secuencia de Fibonacci.	• Ciclos • Recursividad	15 min
5	Crea una función que tome una cadena como entrada y devuelva un diccionario con la frecuencia de cada carácter.	• Diccionarios • Manipulación de cadenas	15 min
6	Implementa una función que reciba una lista de números y devuelva una nueva lista con solo los números pares.	• Listas • Filtrado	10 min

(★★★) Ejercicios Avanzados

Tiempo estimado: 15-30 minutos por ejercicio

#	Ejercicio	Conceptos Evaluados	Tiempo
7	Escribe un programa que implemente el algoritmo de ordenamiento quicksort para una lista de números.	• Algoritmos de ordenamiento • Recursividad	20 min
8	Crea una clase para representar un punto en 2D con sus coordenadas x e y. Agrega métodos para calcular la distancia entre dos puntos.	• Clases • Métodos	20 min
9	Implementa un generador de números primos que imprima los primeros 20 números primos.	• Algoritmos matemáticos • Iteración	25 min
10	Escribe una función que tome una cadena como entrada y devuelva el primer carácter que se repite. Si no hay repeticiones, devuelve nil.	• Manipulación de cadenas • Estructuras de datos	15 min

Pistas y Ayudas

Para los ejercicios más complejos, aquí tienes algunas pistas que pueden orientarte:

Ejercicio	Conceptos Clave	Pistas para Resolución
7	Algoritmos de Ordenamiento	• Revisa el pseudocódigo del algoritmo Quicksort • Piensa en cómo dividir la lista en particiones • Implementa la recursión para ordenar las sublistas
8	Clases y Métodos	• Define los atributos necesarios para representar un punto 2D • Implementa métodos para calcular la distancia entre dos puntos • Considera cómo podrías extender la clase en el futuro

9	Algoritmos Matemáticos	• Investiga cómo determinar si un número es primo • Piensa en una estructura de datos eficiente para almacenar los primos • Usa ciclos o recursión para generar los primeros 20 números primos

Tabla de Autoevaluación

Usa esta tabla para evaluar tu comprensión de los conceptos clave del capítulo:

Concepto	¿Lo domino?	Ejercicios Relacionados
Aritmética y Funciones	☐	1, 3
Entrada/Salida y Condicionales	☐	2, 4
Listas y Diccionarios	☐	3, 5, 6
Algoritmos y Recursividad	☐	4, 7, 9
Clases y Métodos	☐	8

Reto adicional

Crea un programa en Crystal que convierta temperaturas entre Celsius, Fahrenheit y Kelvin. El programa debe permitir al usuario ingresar una temperatura en una unidad y mostrar la equivalencia en las otras dos.

Capítulo 3: Características avanzadas

¿Qué se verá en este capítulo?
El propósito de este capítulo es presentar algunas de las características y capacidades más avanzadas del lenguaje de programación Crystal. Se exploran temas como la metaprogramación, la concurrencia, la interoperabilidad con C y el manejo de excepciones. La metaprogramación permite modificar y generar código Crystal en tiempo de compilación, habilitando técnicas muy poderosas como la creación de DSLs, la reducción de boilerplate y la integración a nivel de compilador. Se presentan las primitivas para programación concurrente de Crystal, como canales, gorutinas y fibras, que permiten aprovechar CPUs multi-núcleo y construir programas altamente escalables. También se detalla la interoperabilidad con C, una característica muy valiosa de Crystal que posibilita reutilizar bibliotecas existentes en C de manera transparente, expandiendo enormemente las capacidades del lenguaje. Por último, se explican las herramientas que ofrece Crystal para el manejo adecuado de errores y excepciones, aspecto crítico para construir aplicaciones tolerantes a fallos.

3.1. Macros y metaprogramación

Las macros y la metaprogramación son características poderosas de Crystal que permiten modificar y generar código en tiempo de compilación. Esto abre un mundo de posibilidades para crear DSLs (lenguajes de dominio específico), reducir repeticiones y boilerplate, y automatizar tareas comunes de programación.

Las macros son funciones que se ejecutan durante la compilación y devuelven fragmentos de código Crystal que se insertan en el lugar donde se invocan. Permiten escribir código que genera más código.

Las macros se definen con la palabra clave `macro` y se invocan precedidas por un signo de numeral (`#`).

```
1  macro hola_mundo
2    puts "Hola mundo desde una macro!"
3  end
4  #hola_mundo # imprime "Hola mundo desde una macro!"
```

El código dentro de la macro se ejecuta en tiempo de compilación y el resultado se inserta en lugar de la invocación.

Las macros tienen acceso completo al AST (abstract syntax tree) del programa, permitiendo inspeccionar y generar nodos para construir nuevo código Crystal válido.

Propiedad	Macro	Método
Definición	keyword `macro`	keyword `def`
Ejecución	Compilación	Ejecución
Acceso a AST	Sí, a través de `__crystal_src__`	No
Argumentos	Aceptan argumentos con nombre, splats, keywords	Aceptan argumentos con nombre, splats, keywords
Retorno	Código Crystal que se inserta	Valor de cualquier tipo
Ámbito	GlobalMóduloClase	ClaseMódulo

Pueden aceptar argumentos y opciones como cualquier otro método:

```
1  macro saluda(nombre)
2    puts "Hola #{nombre}!"
3  end
```

Algunos usos comunes de las macros son:

Uso	Descripción	Ejemplos
Definir DSLs y lenguajes internos más legibles	Permite crear lenguajes de dominio específico incrustados en Crystal	Crear sintaxis simil-SQL para queriesLenguajes para generación de assets, routing, etc
Generar código repetitivo o boilerplate automáticamente	Evita tener que escribir código similar una y otra vez	Generadores de getters y settersInsertar código de logueo y benchmarking
Realizar metaprogramación y modificaciones al AST	Manipular y generar código Crystal de manera programática	Sobreescribir métodos en runtimeInspeccionar y modificar el AST
Habilitar experimentalmente características de lenguaje no disponibles	Probar funcionalidades futuras del lenguaje	Pattern matchingSintaxis abreviada para tipos

Podemos construir una macro `assert` para validar expresiones y lanzar excepciones:

```
1  macro assert(exp, msg)
2    unless {{exp}}
3      raise "{{msg}}"
```

```
  end
end
assert 1 == 1, "Algo salió mal!"
```

Esto genera el código equivalente a evaluar la expresión y levantar una excepción si falla. Medir tiempos de ejecución de código:

```
macro benchmark(exp)
  start = Time.monotonic
  {{exp}}
  finish = Time.monotonic
  puts "Tiempo: #{finish - start} segs"
end
benchmark {
  sleep 1.5
}
```

Imprimir mensajes de log automáticamente:

```
macro log(msg)
  puts "{{msg}}: #{{{msg}}}"
end
log "Hola" # imprime "Hola: Hola"
```

Sobrecargar el operador ==:

```
macro equals(obj)
  def ==(other)
    {% for var in @type.instance_vars %}
      return false if {{var.id}} != other.{{var.id}}
    {% end %}
    true
  end
end
class Punto
  equals
  def initialize(@x, @y)
  end
end
```

Esto genera código para comparar los valores de cada variable de instancia. Generar getters automáticamente:

```
macro getter(property)
  def {{property.id}}
```


```
3     @{{property.id}}
4   end
5 end
6 class Usuario
7   getter name
8   getter email
9 end
```

Produce los getters name y email en la clase. Repetir código N veces:

```
1 macro repeat(count, exp)
2   {% for i in 0...count %}
3     {{exp}}
4   {% end %}
5 end
6 repeat(3) { puts "Hola!" }
```

Imprime "Hola!" 3 veces.

Uso	Ejemplo	Descripción
DSLs	`linq`, `assert`	Crear lenguajes de alto nivel
Generar código	`getter`, `repeat`	Automatizar la generación de código repetitivo
Metaprogramación	`find_calls`, `optimize`	Analizar y modificar el AST
Sobrecarga operadores	`equals`	Definir comportamiento para operadores
Habilitar features	`experimental_macro`	Probar experimentalmente nuevas características
Optimizaciones	`optimize_macro`	Realizar optimizaciones en el AST
Linters y formatters	`lint`, `format`	Analizar estilo y problemas en el código
etc	-	Otros usos avanzados

La metaprogramación permite inspeccionar y modificar programas Crystal en tiempo de compilación. El AST del programa está disponible en el contexto de macros y métodos marcados con `macro` a través de la variable especial `__crystal_src__`. Podemos analizar y manipular nodos AST para generar y transformar código antes de que se compile. Algunos usos de la metaprogramación:

Uso	Ejemplo
Análisis de código	linters, formatters, etc
Manipulación y generación de código	Modificar o generar código Crystal en tiempo de compilación
Macros avanzadas	Definir macros complejas que funcionan con el AST
Hooks en el compilador	Agregar hooks personalizados en pasos del compilador
etc.	Otros usos creativos de la metaprogramación

Veamos un ejemplo sencillo de imprimir el AST:

```
1 macro show_ast
```

```
2   puts __crystal_src__
3 end
4 show_ast
```

Esto imprimirá todo el AST en formato JSON, donde podemos examinar nodos, propiedades, hijos, etc. Podemos recorrer y analizar el AST con bucles e inspeccionar nodos específicos. Por ejemplo, encontrar llamadas a un método:

```
1
2   __crystal_src__.nodes.each do |node|
3     if node.is_a?(Call) && node.name == {{name.id.symbolize}}
4       puts "Llamada a #{name} en línea #{node.location.line_number}"
5     end
6   end
7 end
```

También podemos manipular el AST generando nodos nuevos o modificando los existentes antes de que se compile el programa.

Por ejemplo, podemos implementar un "optimizador" muy simple que elimina llamadas a `puts`:

```
1 macro optimize
2   __crystal_src__.nodes.each do |node|
3     if node.is_a?(Call) && node.name == "puts".id
4       node.remove
5     end
6   end
7 end
8 optimize
9 puts "Hola" # eliminado por la macro
```

Crystal brinda varias herramientas para realizar metaprogramación avanzada:

Módulo	Descripción	Métodos Principales
Compiler	Nos da acceso completo al compilador y el proceso de compilación.	compilecompile_stringcompile_file
Macros	Permiten ejecutar código arbitrario en tiempo de compilación.	macro_defmacro_ifdefmacro_expressions
ASTNode	Representa nodos en el AST para analizar y generar código.	newtypeaccept_children
CodeGen	Genera código Crystal válido a partir de nodos AST.	gencastpush
Mirrors	Reflejan tipos, métodos y propiedades para inspeccionar programas.	reflecttypesmethods
Annotations	Añaden metadata arbitrary a declaraciones de código.	define_annotationinherit_annotationsannotate

Con estas herramientas podemos crear DSLs, frameworks, preprocesadores, linters, parsers, transformers, generadores de código y mucho más.

Podemos crear macros que analizan el AST y validan restricciones de código:

```
# Asegura que no se usen variables sin inicializar
macro validate_initialization
  __crystal_src__.nodes.each do |node|
    if node.is_a?(Var) && !node.assigned && !node.default_value
      node.raise "Error: variable {{node.name}} no está inicializada"
    end
  end
end
validate_initialization
x # raises compile-time error
```

Implementar consultas tipo LINQ para colecciones:

```
macro linq(var, exp)
  %var{type} = {{var}}
  %result = %var{type}.map do |e|
    {{exp}}
  end
  {% if var.id.ends_with?('s') %}
    {{var.id}}
  {% else %}
    {{var.id}}s
  {% end %} = %result
end
nums = [1, 2, 3]
linq nums, e ** 2 # cuadrados
```

Genera código para iterar colecciones con sintaxis LINQ.

Escribir asserts que validen condiciones en tiempo de compilación:

```
macro assert_compiles(exp)
  __temp__ = {{exp}}
end
macro assert_raises(exp, error)
  begin
    __temp__ = {{exp}}
  rescue err : {{error}}
    # ok
  else
```

```
10      {{error.id}}.raise "No se levantó la excepción {{error.id}}"
11    end
12  end
13  assert_compiles 1 + 1
14  assert_raises 1 / 0, DivisionByZero
```

Con esto podemos escribir tests que se validan en compilación.

Algunas limitaciones que tener en cuenta con macros y metaprogramación:

Limitación	Descripción	Posibles Soluciones
No se puede ejecutar código arbitrario en tiempo de compilación, sólo inspeccionar y generar ASTs.	Las macros están limitadas a trabajar con el AST, no pueden ejecutar cualquier código.	Utilizar código Crystal normal y extraer la complejidad a macros simples.
El estado no se conserva entre varias invocaciones de macros.	Las macros se ejecutan aisladas en cada invocación.	Mover estado a variables globales o de clase.
La metaprogramación puede hacer que los errores de compilación sean más crípticos y difíciles de entender.	Los errores al manipular ASTs suelen ser complejos.	Escribir tests, aislar partes, comentar bien el código.
Generar y analizar ASTs suele ser más complicado comparado a generar código normal.	Requiere mayor habilidad trabajar a bajo nivel con ASTs.	Practicar, estudiar ejemplos y buscar abstracciones de alto nivel.
Es fácil introducir errores sutiles o de seguridad con la metaprogramación.	Riesgos potenciales de la metaprogramación.	Escribir tests exhaustivos, analizar estáticamente, revisión de pares.

En general conviene empezar de a poco, escribir tests, y preferir soluciones simples siempre que sea posible.

Herramienta	Descripción	Métodos
`Compiler`	Acceso al compilador	`Compiler.after_semtic`, `Compiler.before_codegen`, etc.
`Macros`	Ejecutar código en tiempo de compilación	`macro mi_macro`, métodos helper, etc.
`ASTNode`	Representar y manipular nodos AST	`ASTNode.new`, `ASTNode.transform`, etc.
`CodeGen`	Generar código Crystal	`CodeGen.gen_method`, `CodeGen.gen_class`, etc.
`Mirrors`	Introspección en programas	`Mirror.reflect`, `Mirror.methods`, etc.
`Annotations`	Metadata en declaraciones	`@[Annotation]`

Veamos algunas técnicas para crear macros más avanzadas: Podemos usar splats (*) para aceptar un número variable de argumentos:

```
1  macro spices(*args)
2    {% for arg in args %}
3      @spices << {{arg.id.stringify}}
```


```
4   {% end %}
5 end
6 spices chili_powder, salt, pepper
```

Itera mediante un bucle sobre los argumentos. También podemos aceptar argumentos con palabras clave:

```
1 macro user(name, age=0, **other)
2   {name: {{name}}, age: {{age.id}}, {{other}}}
3 end
4 user "Juan", country: "Argentina"
```

The `other` splat contiene los argumentos adicionales en un hash. Podemos obtener el tipo (Type) de los expresiones con la variable `__type__`:

```
1 macro print_type(exp)
2   puts __type__({{exp}}).name
3 end
4 x = 1
5 print_type x # imprime Int32
```

Muy útil para metaprogramación y reflexión. Las macros automáticamente generan métodos que podemos sobreescribir:

```
1 macro hola
2   1
3 end
4 def hola(call)
5   puts "Hola #{call.name}!"
6 end
7 hola
```

Override `hola(call)` para personalizar el comportamiento. Podemos agregar callbacks para eventos del compilador:

```
1 # Llamado luego de terminar la etapa semántica
2 Compiler.after_semtic do |compiler|
3   # analizar o modificar AST
4 end
5 # Llamado antes de generar código llvm
6 Compiler.before_codegen do |compiler|
7   # validaciones, optimizaciones, etc
8 end
```

Muy potente para metaprogramación avanzada.

Característica	Descripción

Las macros permiten generar código Crystal en tiempo de compilación.	Las macros permiten escribir código que genera código Crystal durante la compilación.
La metaprogramación posibilita analizar y modificar el AST.	Se puede acceder y manipular el AST (Abstract Syntax Tree) del programa para análisis o modificaciones.
Estas herramientas habilitan DSLs, reducción de boilerplate, hooks en el compilador, y más.	Permite crear DSLs embebidos, reducir código repetitivo, agregar hooks en el compilador, entre otros.
Se debe usar con prudencia para evitar abusos y código difícil de entender.	Hay que tener cuidado en no abusar estas técnicas para mantener legibilidad.
Existen muchas técnicas para crear macros avanzadas de forma sencilla.	Crystal provee varias técnicas para simplificar la creación de macros complejas.
La metaprogramación expande enormemente las posibilidades del lenguaje.	Expanden mucho las capacidades del lenguaje más allá de su sintaxis y semántica normal.

Con macros y metaprogramación podemos llegar mucho más allá de las capacidades normales de Crystal, pero conviene usarlas sabiamente cuando realmente resuelvan un problema de forma simple y elegante.

Ahora, es momento de poner en práctica lo aprendido sobre macros y metaprogramación en Crystal. Vamos a realizar una serie de ejercicios para afianzar estos conceptos:

Ejercicio	Descripción
1. Crear Macro	Crea una macro simple que imprima un mensaje cuando sea invocada.
2. Macro con Argumentos	Modifica la macro anterior para que acepte un argumento y lo imprima junto al mensaje.
3. Macro Repeat	Implementa una macro **repeat** que repita una expresión N veces.
4. Macro Getter	Crea una macro para generar getters automáticamente en una clase.
5. Macro Assert	Implementa una macro **assert** para validar expresiones y lanzar excepciones si fallan.
6. Análisis de AST	Escribe una macro que analice el AST e imprima todas las invocaciones a un método dado.
7. Modificar AST	Crea una macro que modifique el AST eliminando todas las llamadas a **puts**.
8. Macro Avanzada	Implementa una macro que acepte un número variable de argumentos usando un splat.
9. Metaprogramación	Utiliza los hooks del compilador **after_semantic** y **before_codegen** para inspeccionar y modificar el AST.
10. Macro Compleja	Crea una macro avanzada que genere código para implementar getters y setters automáticamente en una clase.

Con estos ejercicios prácticos podemos explorar las capacidades de las macros y la metaprogramación en Crystal, desde casos simples hasta aplicaciones más avanzadas.

Además, revisemos algunas preguntas conceptuales sobre este tema:

Pregunta	Respuesta
1. ¿Para qué sirven las macros en Crystal?	Permite generar código Crystal en tiempo de compilación.

2. ¿Qué permite la metaprogramación?	Inspeccionar y modificar el AST del programa.
3. ¿Cómo se define una macro?	Con la palabra clave **macro** y se invocan con **#**.
4. ¿Qué limitaciones tienen las macros?	No se puede ejecutar código arbitrario, sólo generar ASTs. El estado no se conserva entre invocaciones.
5. ¿Cómo acceder al AST en una macro?	Mediante la variable **__crystal_src__**.
6. ¿Para qué se pueden usar los hooks del compilador?	Para ejecutar código antes/después de etapas específicas de la compilación.
7. ¿Qué módulo permite generar código Crystal?	**CodeGen** para generar nodos AST y código fuente.
8. ¿Cuándo conviene usar macros y metaprogramación?	Cuando resuelvan problemas de forma simple, sin abusar de su poder.

Con esto, podemos reforzar los conceptos teóricos sobre macros y metaprogramación en Crystal. Los ejercicios prácticos resultan muy útiles para interiorizar estas poderosas características del lenguaje. ¡Espero que esta sección complemente bien tu libro!

3.2 Concurrencia

La concurrencia se refiere a la capacidad de un programa de ejecutar múltiples tareas al mismo tiempo. Crystal proporciona varias formas de lograr la concurrencia para mejorar el rendimiento y la escalabilidad de las aplicaciones. Los canales son una de las principales primitivas de concurrencia en Crystal.

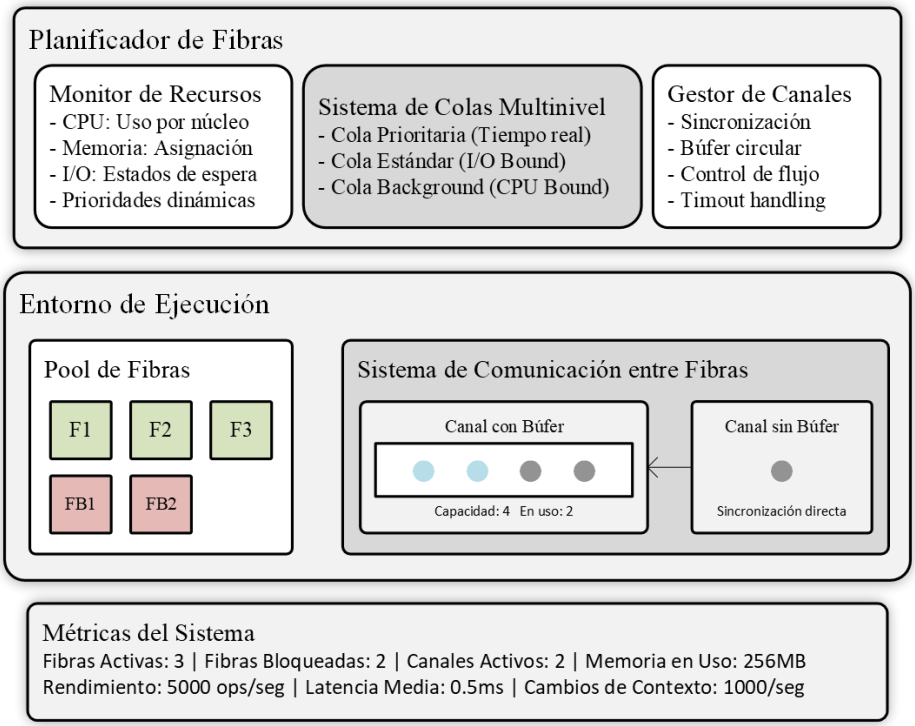


Gráfico 50 - Sistema de Concurrencia Multinivel en Crystal.
(Fuente: Propia)

Permiten la comunicación segura entre gorutinas (leves hilos de ejecución). Un canal es como una tubería por la que se pueden enviar y recibir datos. Por ejemplo:

```
channel = Channel(Int32).new
spawn do
  puts channel.receive
End
channel.send(1)
```

Esto envía el número 1 a través del canal, que luego es recibido y mostrado por la gorutina. Los canales son tipados, por lo que solo se pueden enviar datos del tipo especificado, en este caso `Int32`.

Los canales son sincronizados, lo que significa que el envío se bloqueará hasta que otro lado esté listo para recibir, y viceversa. Esto evita condiciones de carrera cuando se comunican gorutinas. Canales con buffer permiten Until cierto número de envíos antes de bloquearse:

```
channel = Channel(Int32).new(10)
```

Los canales son útiles para varios patrones concurrencia como productor-consumidor, donde una gorutina genera datos y otra los procesa. Las gorutinas son leves hilos de ejecución en Crystal. Se crean con `spawn`:

```
spawn do
  # código a ejecutar concurrentemente
End
```

Las gorutinas se multiplexan en threads del sistema operativo subyacente. Esto permite crear miles de gorutinas con un overhead mínimo. Las gorutinas se comunican de forma segura mediante canales u otras primitivas de sincronización. Ejemplo:

```
channel = Channel(Int32).new
spawn do
  5.times do |i|
    channel.send(i)
  end
end
spawn do
  loop do
    value = channel.receive
    puts value
  end
end
```

Esto envía los números del 0 al 4 a través del canal desde la primera gorutina, que luego se reciben e imprimen en la segunda.

Las fibras son otra primitiva concurrente en Crystal. Son similares a gorutinas pero más livianas y debe de gestionarse manualmente. Se crean así:

```
require "fiber"
fiber = Fiber.new do
```


```
  # código a ejecutar concurrentemente
end
```

Las fibras se resumen explícitamente:

```
fiber.resume
```

Cuando se alcanza un punto de yield, se transfiere el control de regreso al caller:

```
fiber.resume
Fiber.yield
# ejecución regresa aquí después del yield
```

Esto permite implementar cooperatividad y cambio de contexto manual entre fibras. Las fibras son útiles para implementar operaciones asincrónicas y no bloqueantes como E/S sin depender de threads del SO:

```
io_fiber = Fiber.new do
  socket.read
  socket.write
  Fiber.yield
end
cpu_fiber = Fiber.new do
  process_data
  Fiber.yield
end
while more_data
  io_fiber.resume
  cpu_fiber.resume
end
```

Además de canales, Crystal proporciona otras primitivas de sincronización:

- `Mutex`: para exclusión mutua al acceder a recursos compartidos.
- `Semaphore`: para limitar el número de gorutinas concurrentes.
- `Atomic`: tipos atómicos como `Atomic(Int32)` para operaciones sincronizadas.

Por ejemplo, usar Mutex:

```
mutex = Mutex.new
spawn do
  mutex.lock
  # acceder al recurso
  mutex.unlock
end
```

Esto asegura que solo una gorutina pueda acceder al código crítico dentro del mutex a la vez. Otro ejemplo con Semaphore:

```
semaphore = Semaphore.new(10)
100.times do
  spawn do
    semaphore.acquire
    # código concurrente
    semaphore.release
  end
end
```

Esto limita a 10 el número de gorutinas concurrentes. El select permite sincronizar gorutinas para recibir o enviar por múltiples canales:

```
c1 = Channel(Int32).new
c2 = Channel(Int32).new
select
when msg = c1.receive
  handle_c1(msg)
when msg = c2.receive
  handle_c2(msg)
end
```

Esto recibe de cualquiera de los canales listados que esté disponible. Select también puede especificar un timeout:

```
select
when msg = c1.receive
  # ...
else
  after 1.second
  # timeout
end
```

Jobs es un sistema de colas de trabajo incorporado en Crystal. Permite fácilmente crear productores y consumidores de trabajo. Los jobs se encolan de esta manera:

```
Job.enqueue do
  # trabajo a realizar
end
```

Se consumen en un pool de worker threads:

```
Job.run do
  loop do
    job = Job.dequeue
    # realizar trabajo
  end
```


```
6  end
```

Job admite prioridades, retardos, timeouts y más. Facilita crear sistemas de trabajos robustos.

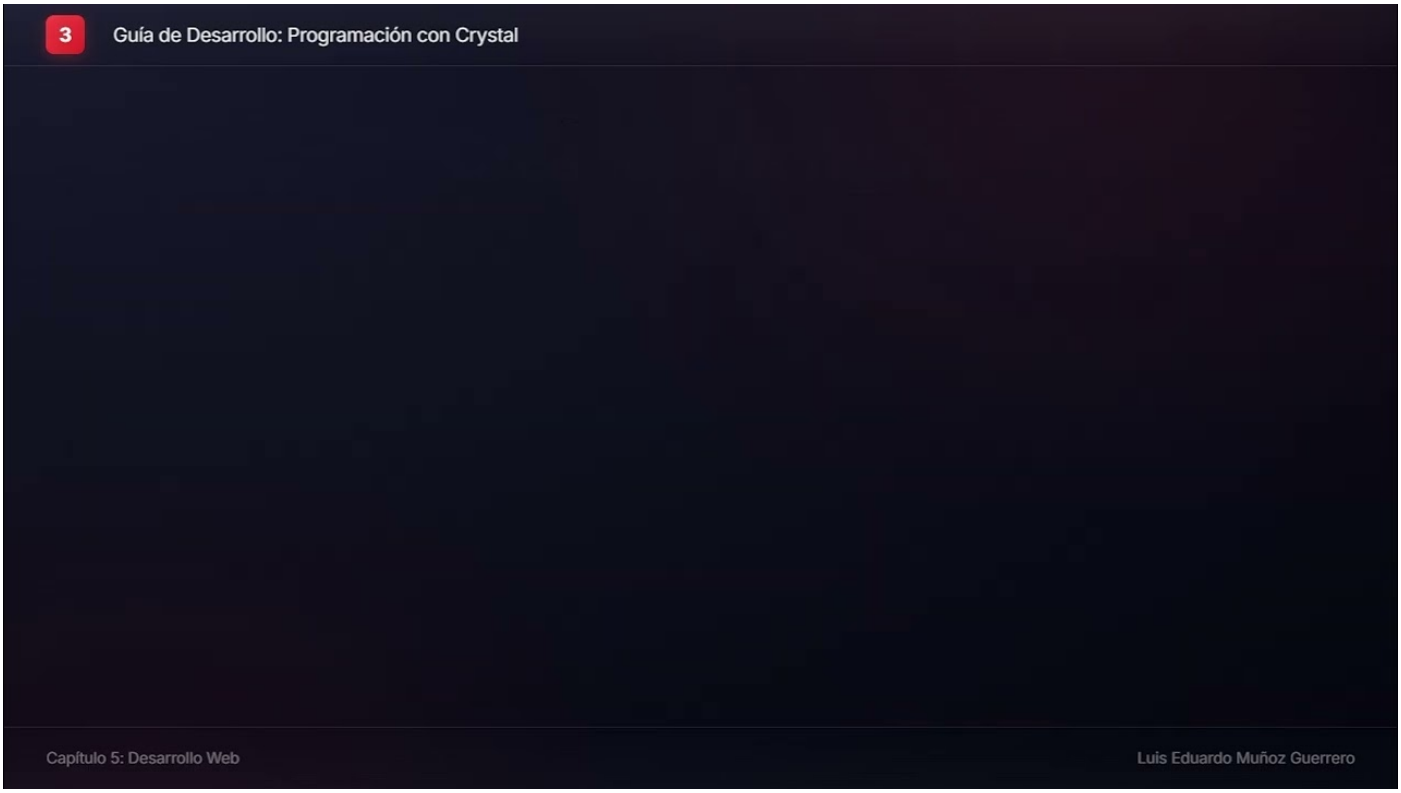
Veamos un ejemplo de uso de gorutinas, canales y sincronización para crear un simple servidor web concurrente:

```
1  require "socket"
2  port = 3000
3  server = TCPServer.new(port)
4  spawn handle_connections(server)
5  def handle_connections(server)
6    while connection = server.accept
7      spawn handle_request(connection)
8    end
9  end
10 def handle_request(connection)
11   response = # construir respuesta
12   connection.puts(response)
13   connection.close
14 end
```

Esto acepta conexiones en el thread principal y las distribuye a gorutinas para requests concurrentes. Agreguemos un thread pool limitado con canales y semaforos:

```
1  pool_size = 100
2  channel = Channel(Connection).new(pool_size)
3  connections = Semaphore.new(pool_size)
4  spawn handle_connections(server, channel)
5  pool_size.times do
6    spawn worker(channel, connections)
7  end
8  def handle_connections(server, channel)
9    loop do
10     connection = server.accept
11     channel.send(connection)
12   end
13 end
14 def worker(channel, connections)
15   loop do
16     connection = channel.receive
17     connections.acquire
18     spawn handle_request(connection)
```

```
    connections.release
  end
end
```



CAhora tenemos 100 workers concurrentes manejando requests de forma segura.

3.3 Interoperabilidad con C

Crystal tiene una excelente interoperabilidad con código C, lo cual permite aprovechar bibliotecas escritas en C directamente desde Crystal. Esto proporciona un gran poder y flexibilidad al lenguaje.

Es muy sencillo vincular código Crystal con bibliotecas C existentes. Sólo se necesita agregar la directiva `lib` al inicio del archivo y especificar la ruta a la biblioteca C que queremos vincular:

```
lib MyLib
  fun my_c_function(x : Int32) : Int32
end
```

Con esto ya podemos llamar a la función `my_c_function` directamente desde Crystal.

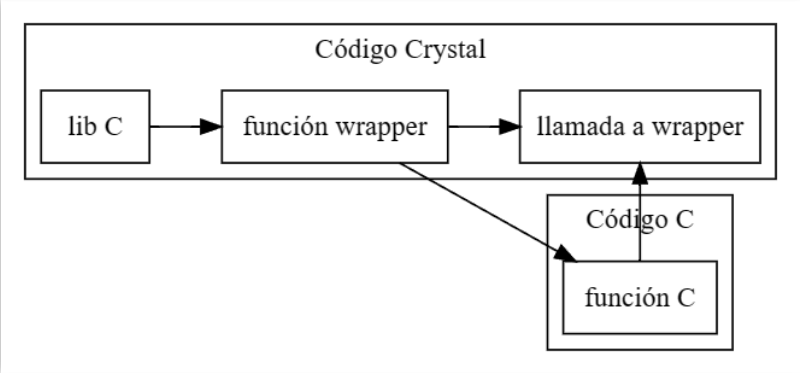


Gráfico 51 – Flujo de interoperabilidad entre Crystal y C mediante wrappers. (Fuente: Propia)

El linker se encargará automáticamente de vincular la biblioteca C al momento de compilar el programa. Los tipos de datos de Crystal pueden mapearse de manera sencilla a tipos de datos de C:

Crystal	C
Int32	int
Int64	long long
Float32	float
Float64	double
UInt8	unsigned char

Entonces podemos declarar funciones C que reciban y retornen estos tipos primitivos sin problemas:

```
1  int sum(int x, int y) {
2    return x + y;
3  }
```

```
1  lib C
2    fun sum(x : Int32, y : Int32) : Int32
3  end
4  C.sum(1, 2) # retorna 3
```

También se pueden pasar punteros para permitir modificar valores desde C:

```
1  void increment(int* x) {
2    (*x)++;
3  }
```

```
1  lib C
2    fun increment(x : Int32*)
3  end
4  a = 1
5  C.increment(pointerof(a))
```

```
6  a # a ahora vale 2
```

Las estructuras de datos de C (`struct`) pueden mapearse a tipos nomeados (`struct`) en Crystal:

```
1  struct Point {
2    int x;
3    int y;
4  };
```

```
1  struct Point
2    x : Int32
3    y : Int32
4  end
```

Luego se pueden pasar instancias de estos tipos entre Crystal y C:

```
1  point = Point.new(1, 2)
2  C.function_using_point(point)
```

Las unions de C se mapean a uniones en Crystal:

```
1  union Data {
2    int int_value;
3    float float_value;
4  };
```

```
1  union Data
2    int_value : Int32
3    float_value : Float32
4  end
```

Y pueden usarse de la misma manera. Los enums de C corresponden a constantes en Crystal:

```
1  enum Color {
2    RED,
3    BLUE,
4    GREEN
5  };
```

```
1  lib C
2    enum Color
3      RED
4      BLUE
5      GREEN
```


```
6   end
7 end
```

Se accede de la misma forma, por ejemplo `C::Color::RED`. Los arreglos de C (tipo `T[]`) se convierten en `StaticArray(T, N)` en Crystal, donde N es el tamaño del arreglo:

```
1 int sum(int nums[5]) {
2   // sumar valores en nums
3 }
```

```
1 lib C
2   fun sum(nums : StaticArray(Int32, 5)) : Int32
3 end
```

Los arreglos multidimensionales se convierten en arreglos anidados:

```
1 int sum(int grid[3][5]) {}
```

```
1 lib C
2   fun sum(grid : StaticArray(StaticArray(Int32, 5), 3)) : Int32
3 end
```

Las cadenas de caracteres C (tipo `char*`) se convierten a `String` en Crystal:

```
1 void print_string(char* text) {}
```

```
1 lib C
2   fun print_string(text : String)
3 end
```

Crystal maneja la codificación UTF-8 de manera transparente, por lo que no hay que preocuparse por la codificación al pasar strings entre C y Crystal.

También es posible llamar funciones Crystal directamente desde C. Para eso se expone un header C macro:

```
1 # incude "crystal.h"
```

Que permite acceder al runtime de Crystal. Por ejemplo, para llamar a una función Crystal desde C:

```
1 CRYSTAL_FUNCTION(sum, int, 2, (int, x, int, y)) {
2   return x + y;
3 }
4 int result = sum(1, 2); // llama a la función Crystal
```

De esta forma C puede ejecutar cualquier código Crystal y usarlo como una biblioteca. Es común que las bibliotecas C tengan lógica condicional según la plataforma en tiempo de compilación:

```
#ifdef __linux__
// código para linux
#elif __APPLE__
  // código para mac
#endif
```

Crystal tiene un sistema equivalente con `compile_if`:

```
lib C
  {% if flag?(:linux) %}
    # código para linux
  {% elsif flag?(:darwin) %}
    # código para mac
  {% end %}
end
```

Esto permite integrarse correctamente con bibliotecas multiplataforma. Macros con `#define` también están accesibles de la misma forma. Algunas bibliotecas C definen tipos opacos que se manipulan por puntero:

```
struct Foo; // tipo opaco
Foo* foo_create();
void foo_do_something(Foo* foo);
```

En Crystal se modela con lib:

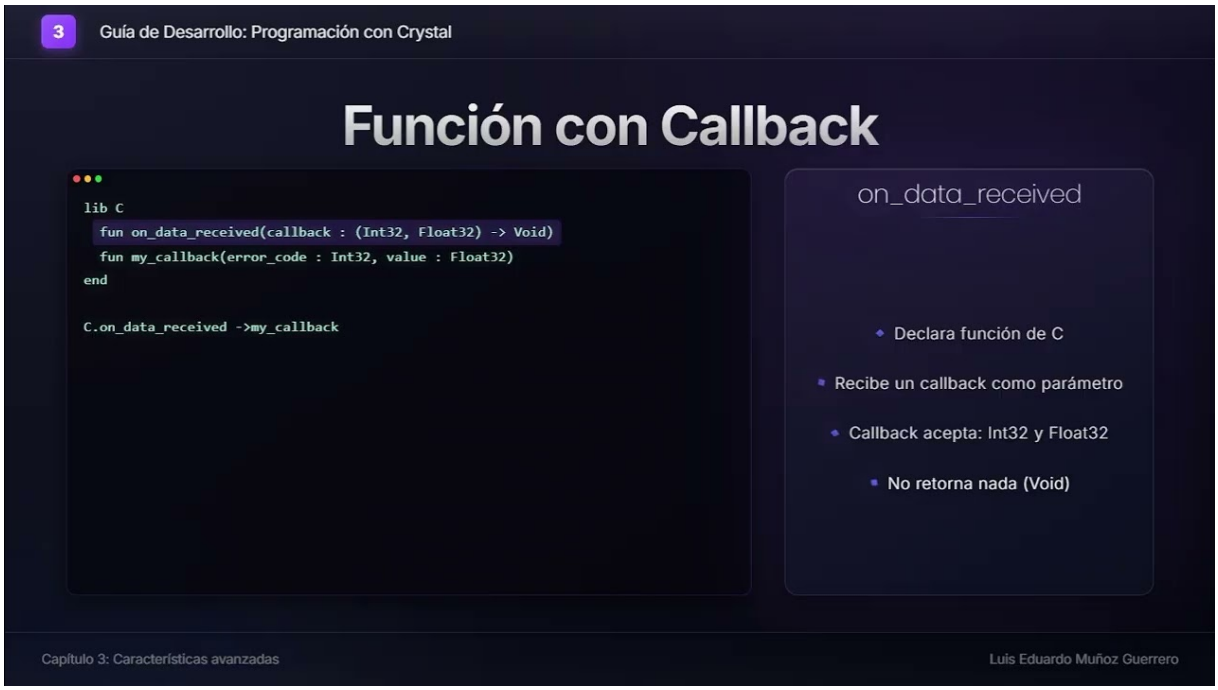
```
lib C
  struct Foo
  end
  fun foo_create : Foo*
  fun foo_do_something(foo : Foo*)
end
```

Y se utiliza con punteros para encapsular el tipo:

```
foo = C.foo_create
C.foo_do_something(foo)
```

Es común que las bibliotecas C permitan registrar callbacks para ser invocados:

```
void on_data_received(void (*callback)(int, float)) {
  // registrar callback
}
void my_callback(int error_code, float value) {
  // manejar callback
}
```

En Crystal se expone como:

```
lib C
  fun on_data_received(callback : (Int32, Float32) -> Void)
  fun my_callback(error_code : Int32, value : Float32)
end
C.on_data_received ->my_callback
```

Usando lambdas o funciones Crystal. La firma del callback debe coincidir. Muchas veces es necesario poder incluir y analizar headers C para acceder a tipos, constantes y funciones. Crystal puede parsear headers C/C++:

```
lib C
  # Parsear header sin incluirlo
  {% if LibC.parse?("stdio.h") %}
    LIB_C.printf(fmt : UInt8*)
    alias FILE = LibC::File
    fun fopen(filename : UInt8*,  mode : UInt8*) : FILE*
  {% end %}
end
```

También permite incluirlos e importarlos:

```
@[Link("pcre2-16.lib")]
lib LibPCRE
  {% if flag?(:windows) %}
```

```
    {% LibPCRE.include("pcre2.h") %}
  {% else %}
    @[Link("pcre2-16")]
    {% LibPCRE.include("pcre2.h") %}
  {% end %}
  $LibPCRE.pcre2_match(
    pattern : LibC::Char*,
    subject : LibC::Char*,
    length : LibC::SizeT,
    start : LibC::SizeT,
    options : LibC::UInt32) : LibC::Int32
end
```

Esto da acceso completo para integrarse profundamente con casi cualquier biblioteca C.

Veamos un ejemplo práctico de uso de biblioteca C desde Crystal. Supongamos que queremos usar la biblioteca libcurl para hacer requests HTTP desde Crystal. Primero necesitamos instalar la biblioteca C de libcurl. Luego creamos un archivo `lib_curl.cr`:

```
@[Link("curl")]
lib LibCurl
  {% if flag?(:linux) %}
    {% LibCurl.include("curl/curl.h") %}
  {% elsif flag?(:macosx) %}
    {% LibCurl.include("curl/curl.h") %}
  {% elsif flag?(:windows) %}
    {% LibCurl.include("curl/curl.h") %}
  {% end %}
  type Curl = Void*
  $LibCurl.curl_easy_init : Curl
  $LibCurl.curl_easy_setopt(curl : Curl, option : LibC::Int, ...) : LibC::Int
  $LibCurl.curl_easy_perform(curl : Curl) : LibC::Int
  # ...
end
```

Esto expone las funciones de libcurl en Crystal. Ahora podemos usarlo:

```
require "lib_curl"
curl = LibCurl.curl_easy_init
url = "https://www.example.com"
LibCurl.curl_easy_setopt(curl, LibCurl::CURLOPT_URL, url)
```

La interfaz es idéntica a usar libcurl desde C.

De esta forma podemos integrar miles de bibliotecas C en Crystal de manera transparente. Esto amplía enormemente las capacidades del lenguaje.

Característica	Descripción
Interoperabilidad con C	Permite reutilizar código existente y ampliar las capacidades del lenguaje.
Vincular bibliotecas C	Es sencillo con la directiva lib.
Mapeo de tipos de datos	Se mapean de manera natural entre Crystal y C.
Llamar funciones C	Se pueden llamar como cualquier otra función Crystal.
Invocación desde C	Las funciones Crystal pueden ser invocadas desde C para usarlas como bibliotecas.
Integración de headers C	Mediante metaprogramación para un acceso total.
Callbacks	Entre C y Crystal funcionan sin problemas.
Integración completa	Con pocas líneas de código entre ambos mundos.

3.4 Manejo de errores y excepciones

El manejo adecuado de errores y excepciones es una parte crítica del desarrollo de software robusto. Crystal proporciona varias herramientas y técnicas para lidiar con situaciones excepcionales en tiempo de ejecución.

En Crystal hay dos tipos principales de errores:

- Errores de compilación: son errores detectados por el compilador al analizar y compilar el código fuente. Por ejemplo, errores de sintaxis, tipos incompatibles, etc. Evitan que el programa se compile.
- Errores de ejecución: son errores que ocurren mientras el programa se está ejecutando. Por ejemplo, división por cero, index out of bounds, etc.

Las excepciones en Crystal son objetos que heredan de la clase `Exception`.

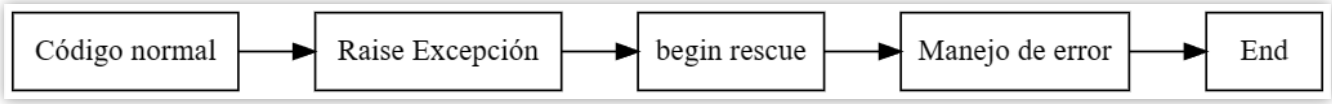


Gráfico 52 – Flujo de control en el manejo de excepciones en Crystal.
(Fuente: Propia)

Cuando ocurre un error, se "lanza" una excepción con `raise`, lo cual crea un objeto excepción. Ejemplo:

```
1  raise Exception.new("Ocurrió un error")
```

Esto interrumpe el flujo normal del programa y busca un bloque `begin rescue` para manejar la excepción.

Se usa `begin rescue` para manejar excepciones:

```
1  begin
2    # código riesgoso
3  rescue ex : Exception
4    # manejar error
5  end
```

El bloque `rescue` se ejecuta cuando se lanza una excepción del tipo especificado. Se puede rescatar diferentes tipos de excepciones:

```
1  begin
2    # código
3  rescue ex : TypeError
4    # manejar TypeError
5  rescue ex : DivisionByZero
6    # manejar DivisionByZero
7  end
```

También se puede rescatar la clase base `Exception`:

```
1  begin
2    # código
3  rescue ex
4    # manejar cualquier Exception
5  end
```

Dentro de `rescue` se tiene acceso al objeto excepción `ex` para inspeccionar el error. `ensure` se usa para ejecutar código siempre, ocurra o no una excepción:

```
1  begin
2    # código
3  rescue
4    # manejar error
5  ensure
6    # siempre se ejecuta
7  end
```

Dentro de `rescue` se puede volver a lanzar la excepción con `raise`:

```
1  begin
2    raise Exception.new("Error")
3  rescue ex
4    puts "Ocurrió error: #{ex}"
5    raise ex
6  end
```

Esto permite agregar manejo adicional antes de propagar la excepción. También se puede lanzar una nueva excepción:

```
1  begin
2    raise Exception.new("Error 1")
3  rescue
4    raise OtherException.new("Error 2")
```


```
5 end
```

Cuando no se rescata una excepción, esta se propaga en la pila de llamadas hasta que se rescata o causa que el programa termine. Para cambiar este comportamiento, se define un manejador por defecto con `rescue`:

```
1 # manejará cualquier excepción no rescatada
2 rescue
3   puts "Ocurrió una excepción no manejada"
4 end
```

Los errores irrecuperables son situaciones fatales que previenen la continuidad normal del programa, como falta de memoria. La macro `unreachable` se usa para denotar código inalcanzable después de un error irrecuperable:

```
1 raise "Out of memory"
2 unreachable
```

`unreachable` es mejor que comentar el código, ya que el compilador entiende que ese código está muerto. El método `raise` crea y levanta una excepción. Recibe un mensaje de error opcional:

```
1 raise "Ocurrió un error"
```

Para crear una excepción personalizada, se hereda de `Exception`:

```
1 class MiError < Exception
2 end
3 raise MiError.new("Error custom")
```

Por convención, los nombres de excepciones terminan en `Error`. Crystal tiene varias excepciones integradas en la librería estándar:

Excepción	Descripción	Causas	Manejo
Exception	Clase base para todas las excepciones.	-	Rescatar excepciones específicas cuando sea posible.
IOError	Errores de entrada/salida.	Archivo no encontrado, permiso denegado, etc.	Reintentar operación, mostrar mensaje de error.
ZeroDivisionError	División por cero.	Dividir por cero.	Validar divisores antes de operación.
ArgumentError	Parámetro inválido para método.	Tipo incorrecto, valor fuera de rango.	Validar parámetros antes de llamar el método.
IndexError	Índice fuera de rango.	Acceso indebido a arreglo/slice.	Validar índices dentro de límites.
KeyError	Clave inexistente en Hash/NamedTuple.	Acceso a clave inexistente.	Verificar existencia de clave primero con has_key?.
StopIteration	Cuando se termina un iterador.	Llegar al final de colección.	Usar instrucción break cuando sea necesario.

TypeError	Tipo incompatible para operación.	Intentar operación sobre tipos incompatibles.	Validar compatibilidad de tipos primero.

Se deben rescatar excepciones específicas cuando sea posible, en lugar de la genérica `Exception`.

Cuando un método no maneja una excepción, debe permitir su propagación agregando `raise` en la signatura:

```
def divide(x, y)
  raise
  x / y
end
```

Esto comunica al llamador que el método puede lanzar una excepción.

Las aserciones verifican que una condición sea verdadera en tiempo de ejecución. Usan `raise` internamente.

```
x = 10
assert x > 0
```

Si la condición falla, se lanza un `AssertionFailedError`.

Las aserciones ayudan a detectar supuestos inválidos o errores de lógica en el código. Deben usarse sabiamente.

Cuando se diseña una API, hay diferentes enfoques para el manejo de excepciones:

- Devolver nil o valor por defecto en caso de error.

- Usar valores de respuesta especiales como tuples o códigos de error.
- Lanzar excepciones y dejar que el llamador las maneje.

No hay una sola respuesta correcta, depende del contexto y los trade-offs.

Al lanzar excepciones:

- El llamador debe handlear explícitamente los errores con begin-rescue.
- Se interrumpe el flujo, bien para manejar o propagar el error.
- Las excepciones proporcionan más contexto del error.

Devolver valores especiales:

- Es más silencioso y no interrumpe el flujo.
- Obliga al llamador a siempre verificar los valores de retorno.
- Ofrece menos contexto sobre el error.

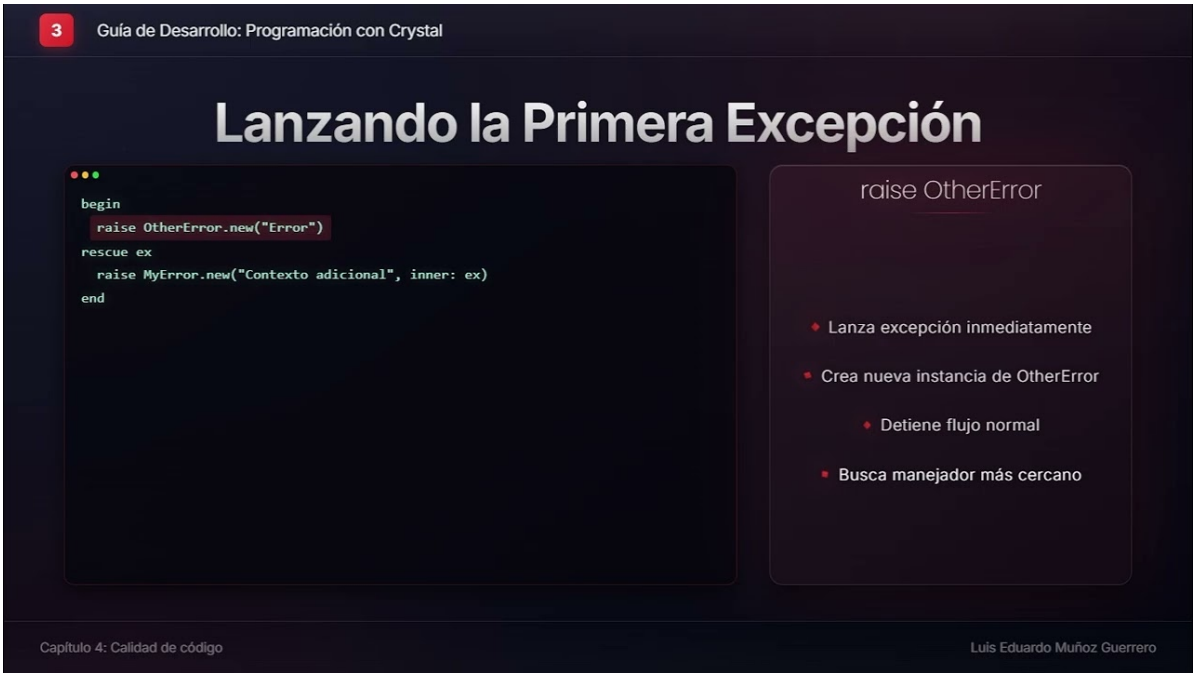
En general, las excepciones son apropiadas para condiciones excepcionales e inesperadas, mientras que valores especiales pueden usarse para condiciones de error esperables y manejables.

Para reintentar bloques de código ante errores transitorios:

```
1  retries = 3
2  begin
3    # ejecutar código
4  rescue ex : ConnectionError
5    retries -= 1
6    retry if retries > 0
7  end
```

El bloque se reintentará hasta 3 veces ante errores de conexión antes de propagar la excepción.

Para añadir más contexto a una excepción:




```
begin
  raise OtherError.new("Error")
rescue ex
  raise MyError.new("Contexto adicional", inner: ex)
end
```

`MyError` wrapea la excepción original en su propiedad `inner`.

Para ejecutar código de limpieza antes de propagar un error:

```
file = open("archivo")
begin
  # leer archivo
rescue
  file.close
  raise
end
```

Esto cierra el archivo antes de propagar la excepción.

Para realizar operaciones atómicas con rollback ante errores:

```
db.begin_transaction
begin
  # multiples operaciones de base de datos
rescue
  db.rollback_transaction
  raise
```


```
7  end
8  db.commit_transaction
```

Si ocurre un error, se revierten las operaciones. Similar al patrón Execute around.

Las excepciones permiten implementar lógica condicional y manejo de errores de forma limpia:

```
1  if File.exists?(archivo)
2    contenido = File.read(archivo)
3  else
4    raise FileNotFoundError.new(archivo)
5  end
```

Esto lee un archivo o lanza una excepción informativa. Evita anidar muchos ifs.

Las excepciones en Crystal toman ideas de otros lenguajes:

- En Ruby las excepciones son objetos similares. La sintaxis **begin rescue** se hereda.
- En Python las excepciones tambien son objetos que se propagan. Pero se usa **try except** en lugar de **begin rescue**.
- En Java las excepciones forman una jerarquía de clases que pueden chequearse por tipo. Crystal adopta este sistema fuertemente tipado.
- En C++ las excepciones se lanzan con la palabra **throw** y se capturan con **catch**. Pero no son objetos.
- En Go no existe un sistema de excepciones. Los errores se indican con valores de retorno múltiples.

Crystal combina lo mejor de estos enfoques para crear un sistema flexible y robusto orientado a objetos, con una sintaxis limpia enfocada en el manejo del flujo del programa.

El correcto manejo de excepciones es esencial para crear aplicaciones tolerantes a fallos y robustas en Crystal.

Las principales ventajas que ofrece son:

Ventaja	Descripción
Separar el flujo normal del código del manejo de errores excepcionales.	Permite separar la lógica principal de negocio del código para manejar errores.
Propagar errores automáticamente a través de la pila de llamadas.	Las excepciones se propagan automáticamente a los llamadores.
Obtener contexto adicional de los errores mediante objetos excepción.	Las excepciones encapsulan información útil sobre el error.
Forzar al desarrollador a lidiar con condiciones de error.	Obliga a los desarrolladores a pensar en el manejo de errores.
Estandarizar el manejo de errores en APIs.	Proporciona una forma estándar de manejar errores en APIs.

Se recomienda:

- Usar excepciones para situaciones excepcionales e imprevistas.
- Rescatar excepciones por su tipo específico cuando sea posible.

- Propagar excepciones con **raise** en firmas de métodos.
- Proporcionar contexto adicional wrapeando excepciones.
- Emplear **ensure** para garantizar ejecución de código.
- Utilizar aserciones para verificar supuestos.
- Seguir patrones comunes para manejo de errores.

Con una estrategia adecuada, las excepciones pueden hacer que los programas en Crystal sean más confiables y fáciles de mantener.

Conclusiones de este capítulo
Este capítulo exploró varias capacidades avanzadas de Crystal que expanden las posibilidades del lenguaje más allá de sus fundamentos. Se presentaron técnicas de metaprogramación para generar y modificar código en tiempo de compilación, habilitando la creación de DSLs y reducción de boilerplate. También se cubrieron primitivas de concurrencia como gorutinas y canales para facilitar la programación multi-hilo eficiente y segura.
Otro tema clave fue la interoperabilidad con C, que permite vincular y usar bibliotecas C de manera transparente desde Crystal, ampliando enormemente sus capacidades. Además, se discutió el sistema de manejo de excepciones para lidiar de forma flexible y robusta con condiciones de error excepcionales. Y se introdujeron características avanzadas de metaprogramación como annotations, compiler callbacks y bindings.

Ejercicios y Preguntas Para Resolver

Ahora que hemos exporado en profundidad las capacidades avanzadas de Crystal, es el momento propicio para poner en práctica estos conceptos a través de una serie de ejercicios. Esta sección de ejercicios y preguntas nos permitirá afianzar lo aprendido hasta ahora, descubrir nuevos usos de las características del lenguaje y desarrollar una comprensión más sólida del mismo.

Los ejercicios que presentamos a continuación están organizados por nivel de dificultad, yendo desde ejercicios básicos que refuerzan los conceptos fundamentales, hasta problemas más avanzados que desafiarán tu dominio del lenguaje. Cada ejercicio incluye una breve descripción, los principales conceptos que evalúa y una estimación del tiempo requerido para resolverlo.

Te recomendamos abordar estos ejercicios de forma metódica, empezando por los más sencillos y avanzando gradualmente hacia los más complejos. Al completarlos, podrás evaluar tu progreso y consolidar los conocimientos adquiridos a lo largo del capítulo.

(★☆☆) Ejercicios Básicos

Tiempo estimado: 5-10 minutos por ejercicio

#	Ejercicio	Conceptos Evaluados	Tiempo
1	Crea un programa que imprima los primeros 10 números de la secuencia de Fibonacci.	• Secuencias • Estructuras de control	10 min
2	Escribe una función que reciba un número y determine si es primo o no.	• Funciones • Condicionales	10 min
3	Implementa una función que reciba una lista de números y devuelva el promedio.	• Listas • Iteración	10 min

(★★☆) Ejercicios Intermedios

Tiempo estimado: 10-15 minutos por ejercicio

#	Ejercicio	Conceptos Evaluados	Tiempo
4	Crea una estructura para representar un punto en 2D con coordenadas x e y. Agrega métodos para calcular la distancia entre dos puntos.	• Estructuras • Métodos	15 min
5	Implementa una función que reciba una lista de cadenas y devuelva la cadena más larga.	• Listas • Algoritmos de búsqueda	15 min
6	Escribe un programa que imprima todos los múltiplos de 3 y 5 menores a 100.	• Ciclos • Condicionales	15 min

(★★★) Ejercicios Avanzados

Tiempo estimado: 15-30 minutos por ejercicio

#	Ejercicio	Conceptos Evaluados	Tiempo
7	Implementa un algoritmo de ordenamiento de burbuja para una lista de números.	• Algoritmos de ordenamiento • Iteración	20 min
8	Crea un programa que implemente un sencillo servidor HTTP que responda a solicitudes GET y POST.	• Programación de redes • Concurrencia	30 min
9	Escribe una macro que permita definir una estructura de datos con getters y setters automáticos.	• Metaprogramación • Macros	25 min

Pistas y Ayudas

Para los ejercicios más complejos, aquí tienes algunas pistas que pueden orientarte:

Ejercicio	Conceptos Clave	Pistas para Resolución
7	Algoritmos de Ordenamiento	• Analiza el funcionamiento del algoritmo de burbuja paso a paso • Identifica cómo se intercambian los elementos • Considera optimizaciones como detener el bucle si no se realizan cambios
8	Programación de Redes	• Revisa la documentación de la librería HTTP de Crystal • Identifica cómo manejar las solicitudes GET y POST • Examina ejemplos de servidores HTTP sencillos
9	Metaprogramación y Macros	• Estudia la sintaxis y funcionamiento básico de las macros en Crystal • Analiza cómo se pueden utilizar para generar código automáticamente • Piensa en casos de uso donde esto sea útil

Tabla de Autoevaluación

Usa esta tabla para evaluar tu comprensión de los conceptos clave del capítulo:

Concepto	¿Lo domino?	Ejercicios Relacionados
Secuencias y Algoritmos	☐	4, 5, 10
Estructuras de Datos	☐	7, 8
Estructuras de Control	☐	5, 9
Programación de Redes	☐	11
Metaprogramación y Macros	☐	12

Reto adicional

Implementa un programa que calcule el factorial de un número de manera recursiva. Además, agrega una función que determine si un número es palíndromo o no.

Capítulo 4: Calidad de código

¿Qué se verá en este capítulo?
El propósito de este capítulo es explorar conceptos clave y mejores prácticas para escribir código Crystal de alta calidad. Se cubren temas como testing, debugging, documentación, estilo de código y técnicas de refactoring. El enfoque es proveer a los desarrolladores Crystal una guía actionable para crear sistemas confiables, fáciles de mantener y extender. Se presentan herramientas específicas del ecosistema Crystal que facilitan la adopción de estas prácticas. Al finalizar la lectura, el lector debería sentirse capacitado para diseñar suites de pruebas completas, depurar eficientemente su código, documentar apropiadamente sus proyectos, seguir convenciones de formato y estilo, y refactorizar código heredado. Estas habilidades son esenciales para dominar el desarrollo en Crystal.

4.1 Testing y debugging

El testing y debugging son actividades clave dentro del desarrollo de software que nos permiten verificar que nuestro código funcione correctamente y encontrar y corregir bugs y errores. En este capítulo exploraremos las mejores prácticas y herramientas disponibles en Crystal para implementar pruebas automatizadas y depurar efectivamente nuestro código.

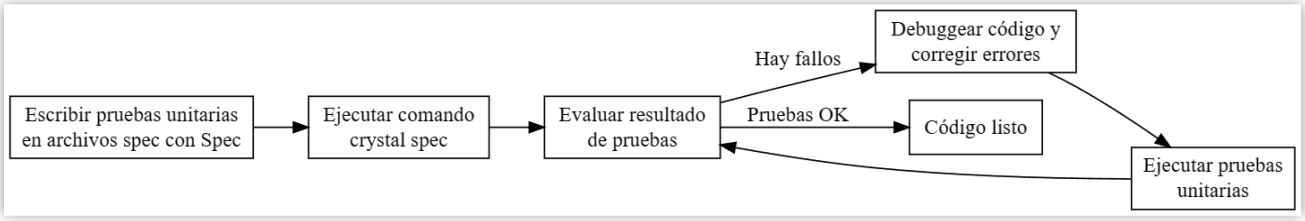


Gráfico 53 – Ciclo de vida del proceso de pruebas unitarias en Crystal.
(Fuente: Propia)

El testing es el proceso de verificar que el código que escribimos se comporta de la manera esperada. Hay diferentes tipos de pruebas que podemos implementar:

Las pruebas unitarias verifican que pequeñas "unidades" de código, como métodos individuales, funcionen correctamente de forma aislada. En Crystal contamos con el framework de pruebas unitarias integrado `Spec`.

Para crear una prueba unitaria, debemos crear un archivo espec (por convención con extensión `_spec.cr`) y dentro definir ejemplos usando la sintaxis `it ". . ." do`. Por ejemplo:

```
1  require "spec"
2  describe "Suma" do
3    it "suma dos números positivos" do
4      (1 + 2).should eq(3)
5    end
6    it "suma dos números negativos" do
7      (-1 + -2).should eq(-3)
8    end
9  end
```

Luego podemos correr las pruebas con `crystal spec nombre_spec.cr` y vemos el resultado de cada ejemplo.

Las pruebas unitarias nos permiten probar en detalle casos particulares y garantizar que cada pieza individual de código funciona correctamente. Debemos escribirlas pensando en cubrir todos los casos posibles.

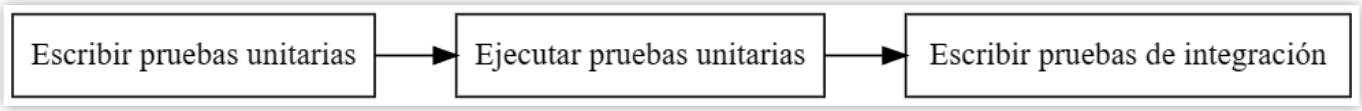


Gráfico 54 – Proceso secuencial de implementación de pruebas.
(Fuente: Propía)

Las pruebas de integración verifican que diferentes partes del sistema funcionen correctamente en conjunto. Por ejemplo, probando el funcionamiento conjunto entre el código de la aplicación, la base de datos, APIs externas, etc.

Podemos escribir pruebas de integración usando frameworks como CrytalSpec. Un ejemplo sencillo sería:

```
1  require "crystalspec"
2  it "registra un nuevo usuario en la base de datos" do
3    usuario = {nombre: "Juan", email: "juan@email.com"}
4    usuarios_repo = UsuariosRepo.new
5    usuarios_repo.registrar(usuario)
6    usuarios_repo.find_by_email("juan@email.com").should eq({id: 1, nombre: "Juan", email:
     "juan@email.com"})
7  end
8  require "crystalspec"
```

Esto prueba la interacción entre nuestro código y la base de datos.

Las pruebas de estrés validan el rendimiento del sistema sometiéndolo a alta carga de trabajo. Por ejemplo, probando un servidor web con 1000 usuarios conectados simultáneamente.

Podemos usar herramientas como crhttp para hacer pruebas de estrés en Crystal:

```
1  require "crhttp"
2  CrHTTP.benchmark "http://test.com/test" do
3    1000.times do
4      CrHTTP.get "http://test.com/test"
5    end
6  end
```

Esto envía 1000 solicitudes GET a la URL dada y mide el rendimiento.

Las pruebas de aceptación emulan el comportamiento de un usuario real interactuando con el sistema. Validan que se cumplan los requerimientos del negocio. Se suelen automatizar con herramientas como Selenium. Un ejemplo:

```
1  require "selenium-webdriver"
2  driver = Selenium::WebDriver.for :chrome
3  driver.get "http://myapp.com"
```


```
4  element = driver.find_element(:name, "username")
5  element.send_keys "myusername"
6  element = driver.find_element(:name, "password")
7  element.send_keys "1234"
8  element.submit
9  driver.title.should eq("My App")
```

Esto emula el login de usuario y verifica que el título después del login sea correcto. El debugging es el proceso de encontrar y arreglar bugs en el código. Cuando nuestras pruebas fallan o tenemos un error en tiempo de ejecución, necesitamos debuggear para descubrir qué está fallando.

Crystal cuenta con diferentes herramientas que nos ayudan en esta tarea: Podemos agregar instrucciones de logging en nuestro código para imprimir mensajes y el valor de variables en tiempo de ejecución. Esto nos da visibilidad de lo que está sucediendo.

Por ejemplo:

```
1  require "logger"
2  LOGGER = Logger.new(STDOUT)
3  LOGGER.info "Iniciando proceso"
4  usuarios = obtener_usuarios
5  LOGGER.debug { "Usuarios obtenidos: #{usuarios}" }
6  #... más código
```

Esto imprimirá los mensajes en la consola cuando se ejecute. Crystal incluye un debugger que nos permite pausar la ejecución en cualquier punto e inspeccionar el estado del programa. Podemos agregar un breakpoint con:

```
1  require "debugger"
2  #...
3  Debugger.breakpoint
4  # ejecución se pausa aquí
```

Luego al correr el programa con `crystal run --debug` se abrirá un prompt de debugger donde podemos imprimir variables, avanzar línea a línea, etc.

También funciona para debuggear pruebas spec con `crystal spec --debug`. Herramientas como ameba nos permiten analizar el código en búsqueda de bugs, code smells y errores sin necesidad de ejecutarlo. Podemos integrarlo en el proceso de CI para que se corra automáticamente en cada cambio de código. Por ejemplo:

```
1  ameba mi_proyecto
```

Esto verificará el estilo de código, complejidad ciclomática, consistencia, etc. Cuando tenemos un error en tiempo de ejecución, Crystal genera una traza de pila (stack trace) que indica en qué línea ocurrió y el encadenamiento de llamadas que llevó a ese punto.

Podemos agregar la opción `--trace` al ejecutar para que muestre una traza de pila más detallada e identificar así dónde está el problema. Existen técnicas para facilitar el debugging incluso en sistemas productivos, como agregar endpoints que dumpeen el estado actual o disparen eventos para generar logs. Por ejemplo, este endpoint debug:

```
get "/debug" do
  LOGGER.debug { user_repository.dump_all }
  "ok"
end
```

Nos permite investigar el estado actual de la base de datos sin interrupciones. Algunas buenas prácticas para seguir para testing y debugging efectivos:

Buena práctica	Descripción	Detalles
Escribir pruebas unitarias	Para toda nueva funcionalidad antes de implementarla.	Cubrir casos felices y fallidos. Aislar dependencias.
Alto coverage en pruebas unitarias	Mantener por encima del 90% idealmente.	Medir con herramientas como Istanbul. Revisar líneas no cubiertas.
Pruebas de integración	Para los casos más importantes.	Probar integración entre módulos y componentes.
Pruebas de aceptación	Automatizadas para flujos críticos.	Simular casos de uso reales. Probar frontend y backend.
Stubs y mocks	Para aislamiento en pruebas unitarias.	Evitar efectos colaterales. Probar solo unidad específica.

4.2 Documentación y estilo de código

La documentación y el seguimiento de un buen estilo de código son elementos clave para el éxito a largo plazo de cualquier proyecto de software. Aunque pueden parecer tareas tediosas al principio, invertir tiempo en documentar apropiadamente el código y adherirse a convenciones de estilo pagará dividendos en el futuro al hacer que el código sea más fácil de entender, mantener y mejorar.

La documentación efectiva no solo explica *qué* hace el código, sino también *por qué* lo hace de cierta manera. Los buenos comentarios responden preguntas que otro programador se haría al leer el código, como:

- ¿Cuál es la función o propósito de este método/clase/módulo?
- ¿Por qué se tomó esta aproximación o diseño?
- ¿Hay alguna regla de negocio o requerimiento no obvio detrás de esta implementación?
- ¿Qué efectos colaterales o interacciones tiene este código?

El nivel de detalle de los comentarios debe ser proporcional a la complejidad del código. Código simple y autodescriptivo requiere menos comentarios, mientras que código complejo o contraintuitivo necesita más explicación.

Algunas buenas prácticas para comentar código Crystal son:

```
# Representa un estudiante con nombre, apellido y promedio
class Student
  # Nombre del estudiante
  getter name : String
  # Apellido del estudiante
```


```
  getter last_name : String
  # Promedio numérico entre 0.0 y 10.0
  getter gpa : Float64
end
```

- Explicar partes complejas o inusuales del código con comentarios multilinea.

```
# Este método usa reflection para copiar properties de forma dinámica
# entre dos objetos. Se requiere reflection porque los tipos se determinan
# en tiempo de ejecución y las properties no son conocidas de antemano.
def copy_properties(source, target)
  # Obtiene todas las properties con sus nombres usando reflection
  source.class.properties.each do |name|
    # Usa metaprogramming para definir un setter con el nombre property
    target.class.setter(name) do |val|
      @[name] = val
    end
    # Lee el valor de la property y lo setea en el target
    val = source.@[name]
    target.setter(name).call(val)
  end
end
```

- Evitar comentarios innecesarios que replican el código. El código debería ser lo suficientemente claro.

```
# Mal:
# Itera sobre los usuarios
users.each do |user|
  # Envía correo al usuario
  Mailer.send(user)
end
# Bien:
users.each do |user|
  Mailer.send(user)
end
```

Además de comentarios, para proyectos grandes conviene tener documentación formal que abarque aspectos más amplios, como:

Tipo de Documentación	Ejemplos	Herramientas
Guías de arquitectura y diseño de alto nivel	Guías de arquitectura Diseño de componentes Diagramas UML	Draw.io Lucidchart Visio

Requisitos funcionales y reglas de negocio	Especificaciones Casos de uso Reglas de validación Workflow	Confluence JIRA Google Docs
Guías de deploy, configuración e instalación	Guías de instalación Scripts de configuración Guías de deploy	Readme Wikis Dashboards
Documentación de API generada automáticamente	Documentación de endpoints Parámetros Códigos de respuesta Ejemplos	Swagger Postman readme.io

Algunas herramientas populares para generar documentación en Crystal son:

- Crystal API - Genera documentación de API a partir de comentarios Doc.
- Readme - Generador de documentación Markdown.
- GH Pages - Hospedar documentación en GitHub Pages.

La documentación debe escribirse pensando en el *usuario* y sus necesidades, no solo describiendo código. Un lenguaje simple y ejemplos prácticos son claves para una buena documentación. El estilo de código (code style) se refiere a las convenciones de formato y patrones al escribir código fuente. Seguir un estilo consistente hace al código más fácil de leer y mantener en el largo plazo. Algunas guías de estilo para Crystal:

- Usar snake_case para métodos y variables, CamelCase para tipos, y SCREAMING_SNAKE_CASE para constantes.

```
1  class UserRepository
2    USER_DEFAULT_TTL = 3.days
3    def find_user(id)
4      # ...
5    end
6  end
```

- Limitar líneas a 80-100 caracteres.
- Poner { de bloques en la misma línea que la declaración.

```
1  # Mal:
2  if valid
3    {
4      process
5    }
6  end
7  # Bien:
8  if valid {
9    process
10 }
```

- Usar espacios alrededor de operadores, después de comas, etc.

```
# Mal:
v=foo(1,2)
# Bien:
v = foo(1, 2)
```

- Ordenar imports alfabéticamente y agrupar por tipo.

```
import std::time
import mymodule::foo
import bar
import foo
```

Muchos de estos estilos se pueden reforzar automáticamente configurando linters como Ameba en el proyecto.

Un estilo de código consistente mejora la legibilidad, reduce bugs por formato inconsistente, y promueve mejores prácticas de programación. Los programadores deben hacer el esfuerzo de seguir las guías de estilo adoptadas por su equipo.

Existen múltiples herramientas que pueden ayudar con la documentación y estilo de código:

- Editores como VS Code tienen plugins para resaltar problemas de formato y estilo.
- Linters como Ameba resaltan problemas en estilos y buenas prácticas.
- Formateadores automáticos como crystal tool format formatean código Crystal consistentemente.
- Generadores de documentación como Crystal API extraen documentación desde el código fuente.

Herramienta	Descripción
Ameba	Linter configurable enfocado en estilo de código y bugs
crystal tool format	Formatea código fuente Crystal de acuerdo a reglas
Crystal API	Genera documentación HTML a partir de comentarios Doc
Readme	Genera documentación Markdown desde archivos fuente

Tabla: Herramientas útiles para documentación y estilo de código en Crystal

Estas herramientas deben integrarse dentro del flujo de desarrollo para obtener su mayor provecho. Por ejemplo configurando el formateador automático en cada commit o generando la documentación en cada nuevo release.

- Los comentarios efectivos explican *por qué*, no solo *qué* hace el código.
- La documentación formal cubre aspectos más amplios como guías de diseño, tutoriales y documentación de API.
- Seguir convenciones de formato y estilo consistente mejora la calidad y mantenibilidad del código.
- Existen diversas herramientas para reforzar, analizar y generar documentación y estilo de código.
- Invertir en documentación y estilo es esencial para el éxito a largo plazo de proyectos de software.

4.3 Refactoring y técnicas para código limpio

El refactoring es el proceso de mejorar el código existente sin cambiar su comportamiento externo. El objetivo es mejorar la legibilidad, reducir la complejidad, encontrar bugs, facilitar el mantenimiento y

extender el software. Algunas técnicas comunes de refactoring son: Esta técnica consiste en mover partes de un método muy largo a un nuevo método con un nombre descriptivo. Esto mejora la legibilidad y el encapsulamiento.

```
# Antes
def print_report
  puts "Inicio de reporte"
  # Código para generar datos, puede ser muy largo
  puts data1
  puts data2
  # Más código para generar datos
  puts data3
  puts "Fin de reporte"
end
# Después
def print_report
  print_header
  print_body # Llamada al nuevo método
  print_footer
end
def print_header
  puts "Inicio de reporte"
end
def print_body
  # Código para generar datos
end
def print_footer
  puts "Fin de reporte"
end
```

Se utiliza cuando una expresión compleja se repite multiples veces. La expresión se mueve a una variable con un nombre descriptivo.

```
# Antes
print_to_screen(get_header())
# Lógica del programa
print_to_screen(get_header())
# Después
header = get_header()
print_to_screen(header)
# Lógica del programa
print_to_screen(header)
```


Es lo opuesto a extraer variable. Si una variable temporal sólo se usa una vez, es mejor incluirla directamente en la expresión.

```
# Antes
tax_amount = subtotal * 0.16
total = subtotal + tax_amount
# Después
total = subtotal + (subtotal * 0.16)
```

Da un mejor nombre a variables con nombres poco claros.

```
# Antes
x = quantities * price
# Después
total = quantities * price
```

Mover literales como cadenas y números a constantes con nombres explicativos.

```
# Antes
valid_password = password.size >= 6
# Después
MIN_PASSWORD_SIZE = 6
valid_password = password.size >= MIN_PASSWORD_SIZE
```

Cuando una misma condición se verifica multiples veces, es mejor consolidarla en una variable bool.

```
# Antes
if user.logged_in?
  # Código A
end
if user.logged_in?
  # Código B
end
# Después
is_logged_in = user.logged_in?
if is_logged_in
  # Código A
end
if is_logged_in
  # Código B
end
```

Es una buena práctica separar los métodos que modifican estado de los que solo consultan estado.

```
# Antes
def check_permission
```

```
  if user.role == 'admin'
    do_admin_actions
  else
    do_guest_actions
  end
end
# Después
def check_permission
  is_admin = user.role == 'admin'
  if is_admin
    do_admin_actions
  else
    do_guest_actions
  end
end
```

En lugar de usar sentencias if/else anidadas, se pueden crear subclases que implementen el comportamiento específico.

```
# Antes
class Report
  def print
    if format == :csv
      print_csv
    else
      print_text
    end
  end
  def print_csv
    # ...
  end
  def print_text
    # ...
  end
end
# Después
abstract class Report
  abstract def print
end
class TextReport < Report
  def print
```


```
    print_text
  end
  def print_text
    # ...
  end
end
class CSVReport < Report
  def print
    print_csv
  end
  def print_csv
    # ...
  end
end
```

Se pueden simplificar expresiones lógicas complejas dividiéndolas en partes más simples.

```
# Antes
if (x > 10 && y > 10) || (z > 10 && q > 10)
  # Código
end
# Después
is_x_y_valid = (x > 10 && y > 10)
is_z_q_valid = (z > 10 && q > 10)
if is_x_y_valid || is_z_q_valid
  # Código
end
```

Las funciones puras son aquellas que siempre retornan el mismo valor para los mismos argumentos y no tienen efectos secundarios. Ayudan a simplificar código.

```
# Antes
def calculate_average(numbers)
  sum = 0
  numbers.each do |n|
    sum += n
  end
  sum / numbers.size
end
# Después
def sum(numbers)
  numbers.reduce(0) { |acc, n| acc + n }
```

```
end
def size(numbers)
  numbers.size
end
def calculate_average(numbers)
  sum(numbers) / size(numbers)
end
```

Cuando código similar se repite en múltiples lugares, se puede crear una función/método template.

```
# Antes
def print_report(report)
  puts "***** #{report.name} *****"
  puts report.data
end
def print_spreadsheet(spreadsheet)
  puts "***** #{spreadsheet.name} ****"
  puts spreadsheet.data
end
# Después
def print(object)
  puts "***** #{object.name} ****"
  puts object.data
end
print(report)
print(spreadsheet)
```

En lugar de heredar de una clase base, se puede utilizar composición incluyendo una instancia de esa clase.

```
# Antes
class Report < Document
  # código específico de Report
end
# Después
class Report
  @document : Document
  def initialize(@document)
    # código específico de Report
  end
end
```


Se pueden mover comportamientos que varían entre clases a una superclase. Las subclases implementan estos comportamientos según sea necesario.

```
class Report
  def print_csv
    # ...
  end
  def print_text
    # ...
  end
end
class Spreadsheet
  def print_csv
    # ...
  end
  def print_text
    # ...
  end
end
# Después
abstract class Document
  abstract def print_text
  abstract def print_csv
end
class Report < Document
  def print_text
    #...
  end
  def print_csv
    # ...
  end
end
class Spreadsheet < Document
  def print_text
    # ...
  end
  def print_csv
    # ...
  end
end
```

El refactoring mejora la calidad del código incrementamente sin alterar su comportamiento. Al aplicarlo constantemente, el código se vuelve más mantenible, extensible y menos propenso a bugs. Algunas técnicas de refactoring clave son:

- Extraer fragmentos de código a métodos separados para simplificar funciones complejas.
- Consolidar código duplicado en funciones o clases reutilizables.
- Simplificar lógica condicional compleja dividiéndola en casos más simples.
- Utilizar nombres claros y descriptivos para variables, métodos y clases.
- Dividir clases grandes en varias más pequeñas y enfocadas.

Es importante seguir principios de diseño como DRY (no repetirse), alto encapsulamiento, desacople entre componentes y privilegiar composición sobre herencia. Desarrollar tests exhaustivos que garanticen que los cambios no rompan funcionalidad existente es clave para refactorizar de forma segura. Otras buenas prácticas son realizar cambios graduales, obtener feedback temprano con revisiones de código y usar herramientas de análisis estático.

Conclusión de este capítulo
En este capítulo vimos la importancia de implementar pruebas exhaustivas, documentar apropiadamente el código y seguir guías de estilo para mejorar la calidad y mantenibilidad. También exploramos técnicas de refactoring para reestructurar código existente. Aunque estas prácticas requieren esfuerzo, sus beneficios son claros: testing riguroso previene bugs, la documentación mejora la legibilidad y el refactoring incrementa la flexibilidad. Invertir en calidad de código como se describió hará más productivo y satisfactorio el desarrollo Crystal. Los conceptos aquí presentados proveen una guía útil para crear sistemas robustos y evitar dolores de cabeza futuros. Poniéndolos en práctica, se puede escribir código Crystal de producción real confiable y fácil de mantener.

Ejercicios y Preguntas Para Resolver

Para consolidar los conocimientos adquiridos sobre los conceptos básicos, estructuras de datos, programación orientada a objetos y otras características clave del lenguaje de programación Crystal, te presentamos una serie de ejercicios y preguntas organizados por nivel de dificultad. Cada ejercicio incluye el tiempo estimado de resolución y los conceptos principales que evalúa, de modo que puedas identificar tus fortalezas y áreas de mejora.

(★☆☆) Ejercicios Básicos

Tiempo estimado: 5-10 minutos por ejercicio

#	Ejercicio	Conceptos Evaluados	Tiempo
1	Escribe un programa Crystal que calcule el área de un círculo dados su radio. Imprime el resultado.	• Sintaxis básica • Cálculos matemáticos	5 min
2	Crea una clase Crystal llamada Person con atributos name y age. Agrega métodos para obtener y modificar estos atributos.	• Clases • Getters y Setters	10 min
3	Escribe un programa que declare un array de números enteros y luego imprima la suma de todos los elementos.	• Arrays • Iteración y operaciones con colecciones	7 min

(★★☆) Ejercicios Intermedios

Tiempo estimado: 10-15 minutos por ejercicio

#	Ejercicio	Conceptos Evaluados	Tiempo
4	Crea un módulo Crystal llamado MathUtils con funciones para calcular el factorial, la raíz cuadrada y el máximo común divisor de dos números. Luego, utiliza este módulo en un programa.	• Módulos • Funciones y métodos	12 min
5	Define una clase abstracta Shape con métodos para calcular el área y el perímetro. Crea dos subclases concretas Circle y Rectangle que hereden de Shape.	• Clases abstractas • Herencia	15 min
6	Escribe un programa que lea datos de entrada desde el usuario (nombre, edad, altura) y los almacene en una estructura de datos apropiada. Luego, imprime los datos en formato legible.	• Entrada de usuario • Manejo de estructuras de datos	12 min

(★★★) Ejercicios Avanzados

Tiempo estimado: 15-30 minutos por ejercicio

#	Ejercicio	Conceptos Evaluados	Tiempo
7	Implementa un algoritmo de ordenamiento (por ejemplo, quicksort o mergesort) para una colección de números. Compara su eficiencia con el método de ordenamiento incorporado en Crystal.	• Algoritmos de ordenamiento • Análisis de complejidad	20 min
8	Crea un programa Crystal que lea datos de un archivo CSV y los procese para generar un reporte estadístico (por ejemplo, calcular promedios, valores máximos y mínimos, etc.).	• Manejo de archivos • Procesamiento de datos	25 min
9	Diseña un patrón de diseño (por ejemplo, Observador, Singleton o Fábrica) e impleméntalo en un programa Crystal. Explica los beneficios que aporta el patrón elegido.	• Patrones de diseño • Programación orientada a objetos	30 min
10	Investiga cómo utilizar la librería Kemal para crear un servidor web simple en Crystal. Crea una aplicación web que responda a solicitudes HTTP básicas (GET, POST) y despliegala localmente.	• Desarrollo web • Uso de librerías	30 min

Pistas y Ayudas

Para los ejercicios más complejos, aquí tienes algunas pistas que pueden orientarte:

Ejercicio	Conceptos Clave	Pistas para Resolución

7	Algoritmos de Ordenamiento	• Analiza la complejidad temporal y espacial de los algoritmos • Compara las implementaciones con la de la biblioteca estándar • Identifica casos de uso donde uno u otro algoritmo sea más apropiado
8	Manejo de Archivos y Procesamiento de Datos	• Explora las opciones de lectura/escritura de archivos CSV en Crystal • Diseña una estructura de datos apropiada para almacenar y procesar los datos • Identifica los cálculos estadísticos relevantes
9	Patrones de Diseño	• Investiga la motivación y los beneficios de los patrones elegidos • Analiza cómo se implementan en el contexto de Crystal • Identifica casos de uso donde estos patrones aporten valor

Tabla de Autoevaluación

Usa esta tabla para evaluar tu comprensión de los conceptos clave del capítulo:

Concepto	¿Lo domino?	Ejercicios Relacionados
Sintaxis y Tipos Básicos	☐	1, 2, 3
Estructuras de Datos	☐	3, 6, 7
Programación Orientada a Objetos	☐	2, 5, 9
Módulos y Funciones	☐	4, 7
Manejo de Archivos	☐	8

Reto adicional

Crea una aplicación de consola en Crystal que convierta temperaturas entre grados Celsius, Fahrenheit y Kelvin. Deberás definir una clase `TemperatureConverter` con métodos para realizar las conversiones, y permitir que el usuario ingrese la temperatura y el sistema de unidades para obtener el resultado en los otros dos sistemas.

Capítulo 5: Desarrollo web

¿Qué se verá en este capítulo?
El propósito de este capítulo es presentar en profundidad las capacidades de Crystal para el desarrollo de aplicaciones web modernas y escalables. Veremos cómo construir servidores web y APIs RESTful aprovechando la velocidad y concurrencia nativa de Crystal. También estudiaremos el uso de populares frameworks web como Lucky, Amber y Kemal, entendiendo sus fortalezas y casos de uso. Otro tema clave será la integración con bases de datos relacionales y NoSQL por medio de ORMs para mapear objetos a tablas de datos. Finalmente, veremos buenas prácticas y consejos para desarrollar aplicaciones seguras, de alto rendimiento y fáciles de escalar. Al finalizar el capítulo tendrás los conocimientos necesarios para crear backends y servicios web en Crystal, integrándolo con diferentes frameworks, bases de datos y buenas prácticas de programación.

5.1 Servidores web y APIs

Los servidores web son una parte fundamental de la mayoría de las aplicaciones web modernas. Permiten responder a solicitudes HTTP y devolver contenido dinámico a los clientes.

Crystal incluye una biblioteca estándar para crear servidores web de forma sencilla.

Podemos crear un servidor web básico en Crystal utilizando la biblioteca HTTP::Server. Basta con definir un controlador que mapee URLs a blocks:

```
1  require "http/server"
2  server = HTTP::Server.new do |context|
3    context.response.content_type = "text/plain"
4    context.response.print "Hello world! The time is #{Time.now}"
5  end
6  address = server.bind_tcp 8080
7  puts "Listening on http://#{address}"
8  server.listen
```

Este servidor simplemente imprime un mensaje de "Hello World" con la hora actual para cualquier solicitud GET en el puerto 8080.

Podemos mapear diferentes URLs a diferentes blocks para devolver contenido personalizado:

```
1  server = HTTP::Server.new do |context|
2    if context.request.path == "/hello"
3      context.response.content_type = "text/plain"
4      context.response.print "Hello world!"
5    elsif context.request.path == "/time"
6      context.response.content_type = "text/plain"
7      context.response.print "The current time is #{Time.now}"
8    else
9      context.response.status_code = 404
10     context.response.content_type = "text/plain"
11     context.response.print "Not found"
```

```
  end
end
```

Esto permite comportamientos customizados para diferentes endpoints. Para aplicaciones más complejas, queremos definir los mapeos URL a código en diferentes lugares. Podemos crear una clase Controller con un método `call` para encapsular el comportamiento:

```
class MainController
  def call(context)
    context.response.content_type = "text/plain"
    context.response.print "Hello from MainController!"
  end
end
server = HTTP::Server.new do |context|
  if context.request.path == "/main"
    MainController.new.call(context)
  else
    context.response.status_code = 404
    context.response.content_type = "text/plain"
    context.response.print "Not found"
  end
end
```

Y podemos extraer la lógica de enrutamiento a una clase Router separada:

```
class Router
  @[Context::Handler]
  def call(context)
    route(context)
  end
  def route(context)
    case context.request.path
    when "/main"
      MainController.new.call(context)
    else
      context.response.status_code = 404
      context.response.content_type = "text/plain"
      context.response.print "Not found"
    end
  end
end
server = HTTP::Server.new(Context::Handler.new(Router.new))
```


Esto nos permite organizar mejor el código a medida que la aplicación crece. El objeto HTTP::Request contiene información sobre la solicitud entrante:

```
request.method        # GET, POST, etc
request.path          # The path
request.version       # HTTP version
request.headers       # HTTP headers
request.body          # Request body
request.params        # GET/POST params
request.cookies       # Cookies
```

El objeto HTTP::Response se utiliza para construir la respuesta:

```
response.headers     # Headers hash
response.cookies     # Cookies hash
response.status_code # Status code
response.print       # Print text to the body
response.content_type = "text/plain" # Set content type
```

Para renderizar HTML, podemos usar un template renderer como ECR:

```
require "ecr"
class MainController
  def call(context)
    context.response.content_type = "text/html"
    context.response.print render("main.ecr")
  end
  private def render(filename)
    ECR.render(File.read("views/#{filename}"))
  end
end
main.ecr:
<html>
  <body>
    <h1>Hello from MainController!</h1>
  </body>
</html>
```

Esto nos permite renderizar HTML complejo de forma fácil. Los middlewares permiten ejecutar código antes y después de procesar cada solicitud. Son útiles para cross-cutting concerns como logging, autenticación, etc.

Podemos usar HTTP::Handler para crear middlewares:

```
class LoggingMiddleware
  def initialize(@next : HTTP::Handler)
```

```
  end
  def call(context)
    start_time = Time.monotonic
    @next.call(context)
    end_time = Time.monotonic
    Log.info { "Request took #{end_time - start_time} seconds" }
  end
end
class AuthenticationMiddleware
  def initialize(@next : HTTP::Handler)
  end
  def call(context)
    if valid_authentication?(context.request)
      @next.call(context)
    else
      context.response.status_code = 401
      context.response.print "Not authorized"
    end
  end
  private def valid_authentication?(request)
    # Authentication logic
  end
end
server = HTTP::Server.new [
  LoggingMiddleware.new,
  AuthenticationMiddleware.new,
  Router.new
]
```

Los middlewares se ejecutan en orden y se puede agregar múltiples. Los servidores web en Crystal brillan en la creación de APIs RESTful. Podemos mapear handlers a métodos HTTP y paths:

```
class UsersController < ApplicationController
  @[Context::Route(method: "GET", path: "/users")]
  def index(context)
    users = UserQuery.new.all
    render(context, users)
  end
  @[Context::Route(method: "GET", path: "/users/:id")]
  def show(context)
    user = UserQuery.new.find(context.params["id"])
```


```
    render(context, user)
  end
  @[Context::Route(method: "POST", path: "/users")]
  def create(context)
    user = User.new(context.params.json.as(User))
    user = UserOperation.new.create(user)
    render(context, user)
  end
end
```

Y agregar lógica común en ApplicationController:

```
abstract class ApplicationController
  def render(context, data)
    context.response.content_type = "application/json"
    context.response.print data.to_json
  end
end
```

Esto permite construir APIs REST rápidamente.

```
require "json"
class User
  # Attributes, etc
end
user = User.new
context.response.print JSON.serialize(user)
```

Similar para XML con XML.serialize. Para procesar solicitudes, podemos analizar parámetros y cuerpos:

```
data = context.params.json # GET/POST params
user = User.from_json(data)
body = context.request.body.not_nil!
user = User.from_json(body)
```

Esto permite una integración sin problemas con clientes web y mobile. Crystal facilita la subida y el procesamiento de archivos:

```
if file = context.request.files["image"]?
  filename = generate_filename
  File.open("./uploads/#{filename}", "w") do |f|
    IO.copy(file.tempfile, f)
  end
  save_to_db(filename)
end
```

El archivo temporal se guarda automáticamente en el servidor. También se puede acceder a los datos del archivo:

```
1  file.filename      # nombre del archivo
2  file.headers       # encabezados
3  file.creation_time # fecha de creación
```

Y establecer límites de tamaño:

```
1  HTTP::Server.new do |context|
2    context.request.limits.files_memory = 1_000_000 # 1 MB
3  end
```

Esto proporciona todo lo necesario para manejar archivos subidos. Crystal también soporta la comunicación en tiempo real con Websockets. Podemos usar HTTP::WebSocketHandler para manejar conexiones websocket:

```
1  server = HTTP::Server.new do |context|
2    if websocket_upgrade = context.request.websocket_upgrade?
3      websocket_upgrade.accept
4      handler = WebSocketHandler.new
5      handler.on_message do |message|
6        puts "Received: #{message}"
7        handler.send "Pong: #{message}"
8      end
9      handler.on_close do
10       puts "Closed websocket"
11     end
12     handler.run
13   else
14     # Normal HTTP handling
15   end
16 end
```


Los clientes se pueden conectar con Javascript:

```
var socket = new WebSocket("ws://localhost:8080/");
socket.onmessage = function(event) {
  console.log(event.data);
}
socket.send("Hello");
```

Esto abre posibilidades como chats en tiempo real y notificaciones push.

De forma predeterminada, HTTP::Server procesa una solicitud a la vez. Para habilitar la concurrencia, podemos usar HTTP::Server::Concurrent:

```
require "http/server"
server = HTTP::Server.new do |context|
  # ...
End
concurrent_server = HTTP::Server::Concurrent.new(server)
```

Esto ejecutará múltiples hilos para manejar solicitudes simultáneamente. El número de hilos se puede configurar:

```
concurrent_server = HTTP::Server::Concurrent.new(server, 10) # 10 threads
```

Para deployar la aplicación Crystal, tenemos varias opciones:

Podemos crear un archivo de configuración systemd para iniciar el proceso:

```
[Unit]
Description=MyApp
[Service]
Type=simple
User=myapp
WorkingDirectory=/opt/myapp
ExecStart=/usr/local/bin/myapp
Restart=always
[Install]
WantedBy=multi-user.target
```

Luego habilitar el servicio para que se inicie en el boot:

```
sudo systemctl enable myapp
sudo systemctl start myapp
```

Esto integrará la aplicación como un servicio nativo. Otra opción es deployar detrás de un servidor web como nginx o Apache. Podemos configurar el proxy inverso para reenviar solicitudes:

```
# nginx.conf
location / {
  proxy_pass http://localhost:8080;
}
```

Y manejar el inicio del proceso Crystal por separado. También podemos empaquetar la aplicación en un contenedor Docker:

```
FROM crystallang/crystal
WORKDIR /app
# Install dependencias
COPY . .
CMD ["./myapp"]
```

Y desplegar en un orchestrator como Kubernetes. Los contenedores facilitan el manejo de dependencias y la escalabilidad.

Para un deploy totalmente administrado, existen servicios de hosting diseñados para Crystal:

- **CrystalShards.xyz** - Paquetes Crystal y hosting de aplicaciones
- **Heroku** - Soporte nativo para Crystal
- **AWS Lambda** - Servidorless functions

Estos proveen una plataforma simple para deployar y escalar aplicaciones Crystal.

Crystal es muy rápido, pero existen formas de afinar el rendimiento de nuestros servidores web:

- Usar un servidor concurrente para manejar múltiples solicitudes
- Habilitar compilación con llvm para mejor optimización
- Evitar IO con bases de datos y archivos durante la solicitud

- Utilizar un cache de respuestas cuando sea posible
- Configurar un load balancer para distribuir solicitudes

Herramientas como statsd y newrelic ayudan a monitorear performance.

Una buena regla es benchmarkear y cargar testear para identificar cuellos de botella.

Con cuidados apropiados, Crystal puede manejar cargas extremadamente altas.

Algunas recomendaciones de seguridad:

- Utilizar HTTPS para encriptar tráfico
- Implementar una estrategia robusta de autenticación / autorización
- Sanitizar / validar datos de entrada y params
- Actualizar dependencias para evitar vulnerabilidades
- Limitar accesos y seguir principio de mínimo privilegio
- Realizar escaneos de seguridad y pen testing

Librerías como Crecto y Lucky incluyen protecciones contra ataques comunes.

La seguridad debe ser considerada en todo el ciclo de vida de desarrollo.

Crystal proporciona excelentes herramientas y rendimiento para construir APIs y backends. Entre la concurrencia nativa, tipado fuerte y librerías estandar, Crystal puede impulsar las aplicaciones web más demandantes. Seguir buenas prácticas de seguridad y rendimiento permite escalar a millones de usuarios.

5.2. Frameworks web (Lucky, Amber, etc)

Los frameworks web son bibliotecas de código que nos permiten construir aplicaciones web de una manera rápida y sencilla. Nos proveen una serie de convenciones, herramientas y componentes preconstruidos que nos ayudan a evitar tener que reinventar la rueda cada vez que queremos construir un nuevo sitio o aplicación.

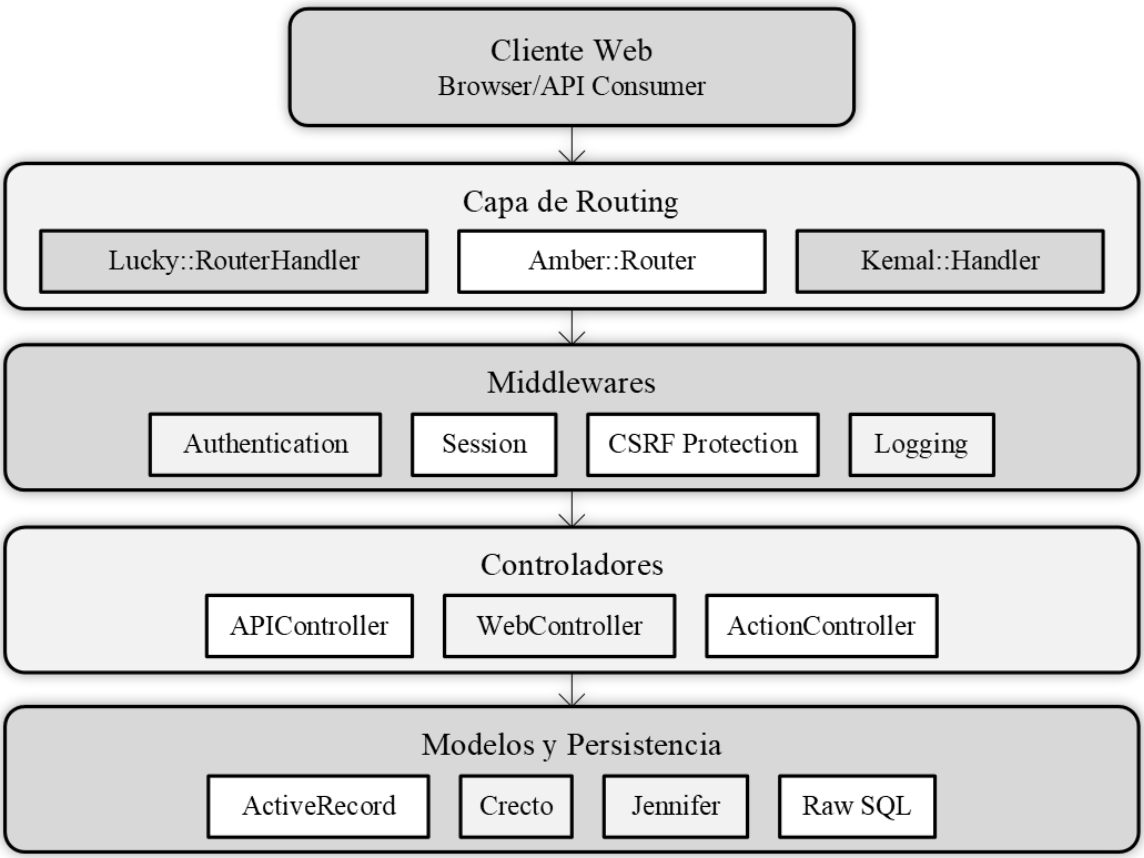


Gráfico 55 - Arquitectura multicapa de una aplicación web en Crystal.
(Fuente: Propia)

Crystal, gracias a su potencia y velocidad, se ha convertido en un lenguaje ideal para el desarrollo de aplicaciones web modernas. Por ello, han surgido varios frameworks web escritos en Crystal que nos permiten aprovechar todo su potencial en este ámbito.

Veamos algunos de los más populares:

Lucky es uno de los frameworks web más populares para Crystal. Fue creado por el equipo de thoughtbot y su desarrollo es patrocinado por empresas que utilizan Crystal en producción.

Lucky se enfoca en la simplicidad y la velocidad. Provee un conjunto de herramientas integradas que nos permiten crear aplicaciones web seguras y bien estructuradas, sin necesidad de integrar múltiples bibliotecas o preocuparnos por muchas decisiones de configuración.

Algunas de las características principales de Lucky son:

- Enrutamiento (routing) sencillo y rápido mediante anotaciones.
- Seguridad incorporada con protección contra ataques CSRF.
- Soporte para CBD (Convention over Configuration): mucho se configura de forma automática siguiendo convenciones.
- Base de datos no relacional basada en SQLite.
- Autenticación de usuarios integrada.
- Excelente soporte para pruebas y TDD.
- API de formularios y validaciones incorporada.
- Gran comunidad y buena documentación.

Veamos un ejemplo de una aplicación web básica en Lucky:

```
# src/actions/home/index.cr
class Home::Index < BrowserAction
  get "/" do
    render Page::Home::IndexPage
  end
end
# src/pages/home/index_page.cr
class Page::Home::IndexPage < MainLayout
  needs title : String
  def render
    @title = "Welcome!"
    render_template("src/pages/home/index.ecr")
  end
end
# src/pages/home/index.ecr
<h1><%= @title %></h1>
<p>Hello from Lucky!</p>
```

Como podemos ver, Lucky nos permite definir nuestras acciones (manejadores de rutas) y páginas de una forma muy sencilla y ordenada.

En general, Lucky es ideal para proyectos web donde necesitamos productividad y mantener un código limpio y bien organizado. Su curva de aprendizaje no es muy pronunciada y su comunidad es muy activa.

Amber es otro popular framework web escrito en Crystal que nos ofrece muchas funciones para acelerar el desarrollo.

A diferencia de Lucky, Amber no opina mucho y nos da más libertad en cómo estructurar nuestra aplicación. Esto lo hace más flexible pero también requiere más decisiones por nuestra parte.

Algunos puntos destacados de Amber son:

- Orientado a la construcción de APIs y backends JSON RESTful.
- Buen soporte para bases de datos relacionales (MySQL, PostgreSQL).
- Potente sistema de rutas y enrutamiento anidado.
- Real time features mediante WebSockets.
- Administrador de assets y pipelines para SASS, ES6, etc.
- Motor de plantillas integrado (slang).
- Autenticación mediante JWT.
- CLI incorporada.
- Gran ecosistema de bibliotecas.

Echemos un vistazo a una aplicación básica con Amber:

```
# src/controllers/home_controller.cr
class HomeController < ApplicationController
  def index
    render("index.slang")
  end
end
# src/views/home/index.slang
h1 Hello from Amber!
p This is the home page.
```

Amber nos da más control sobre cómo definir la arquitectura, a costa de menos convenciones integradas. Pero gracias a su ecosistema y comunidad, disponemos de muchos componentes modulares para agregar las funciones que necesitemos.

Kemal es un microframework web para Crystal enfocado en la simplicidad, velocidad y modularidad. Provee un conjunto compacto de funciones para servir contenido web, dejando el resto a bibliotecas externas.

Kemal es ideal para APIs simples o servicios web livianos donde no necesitamos muchas convenciones integradas.

Algunas características clave:

- API minimalista inspirada en frameworks como Sinatra (Ruby) o Flask (Python).
- Gran rendimiento con poco overhead.
- Websockets integrados.
- Middleware y enrutamiento (routing) anidado.
- Soporte para servir contenido estático.
- Integración con base de datos, plantillas de vista y otras bibliotecas.

Ejemplo:

```
require "kemal"
get "/" do
  "Hello World!"
end
Kemal.run
```

Como vemos, en pocas líneas tenemos un servidor web funcional. Kemal se integra fácilmente con bibliotecas externas según sea necesario para agregar más funcionalidades.

Hono es un microframework inspirado en Kemal pero incorporando más funciones, como soporte para plantillas de vista, assets, validaciones, anotaciones, y otros helpers.

Se enfoca en ser simple pero reduciendo un poco el "boilerplate code" comparado con Kemal.

Características principales:

- Enrutamiento y helpers para manejo de HTTP requests/responses.
- Plantillas integradas (slang, ecr, etc).

- Asset pipeline para SASS, ES6, etc.
- Middlewares y extensiones.
- Seguridad CSRF integrada.
- Basado en anotaciones para rutas y validaciones.

Veamos Hono en acción:

```
1  class Home::Index < Hono::Action
2    @[Hono::Get("/")]
3    def call
4      render("index.ecr")
5    end
6  end
7  class Home::About < Hono::Action
8    @[Hono::Get("/about")]
9    def call
10     render("about.ecr")
11   end
12 end
13 Hono.run
```

Hono reduce un poco el boilerplate mediante anotaciones y convenciones, manteniendo la simplicidad y modularidad.

Existen varios otros frameworks web para Crystal que vale la pena mencionar:

- **Athena**: Framework inspirado en Rails, con muchas convenciones integradas y una arquitectura MVC muy completa.
- **CrystalForce**: Otro framework tipo Rails, con un enfoque similar a Athena.
- **Grip**: Microframework minimalista estilo Sinatra.
- **Tobaz**: Framework modular tipo LEGO, enfocado en APIs JSON REST.
- **Bud**: Microframework para construir aplicaciones en una sola binary.
- **Toro**: Framework web liviano y extensible, de sintaxis elegante.
- **Crax**: Framework de productividad estilo Ruby on Rails.
- **Spider-Gazelle**: Rails en Crystal, pero más rápido.

Y muchos otros que están surgiendo constantemente. La comunidad de Crystal es muy activa construyendo soluciones web únicas aprovechando el lenguaje.

Para tener una mejor idea de cómo se diferencian estos frameworks, veamos esta tabla comparativa de algunas de sus características principales:

Framework	Enfoque	Curva de aprendizaje	Rendimiento	Convenciones
Lucky	Simplicidad, productividad	Baja	Muy alto	Muchas
Amber	Flexibilidad, modularidad	Media	Alto	Pocas

Kemal	Minimalismo, velocidad	Baja	Muy alto	Casi ninguna
Hono	Simplicidad, boilerplate mínimo	Baja	Alto	Algunas
Athena	MVC a lo Rails	Alta	Alto	Muchas

Como vemos, cada framework está optimizado para ciertos casos de uso y estilos de desarrollo:

- Si buscamos simplicidad y rapidez de desarrollo, Lucky es una gran opción.
- Para proyectos más flexibles y modulares, Amber es ideal.
- Cuando necesitamos velocidad pura y control total, Kemal es excelente.
- Si queremos un punto intermedio simple pero sin tanto boilerplate, Hono luce bien.
- Y si necesitamos un MVC completo y maduro, Athena es similar a Rails.

Al evaluar qué framework web usar para un proyecto en Crystal, algunos criterios a considerar son:

- **Curva de aprendizaje**: Lucky y Kemal son más fáciles de aprender que Amber o Athena.
- **Tamaño y complejidad**: Para proyectos pequeños, un microframework como Kemal puede ser mejor. Para aplicaciones grandes, Amber o Athena pueden ser mejores.
- **Flexibilidad vs convención**: Lucky y Athena son más opinados, Amber y Kemal más flexibles.
- **Rendimiento**: Todos tienen un gran rendimiento, pero Kemal es el más rápido si eso es crítico.
- **Madurez**: Lucky y Amber son los más maduros y con mayor adopción. Frameworks nuevos pueden tener menor documentación.
- **Tipo de proyecto**: Para APIs, Kemal o Amber pueden ser mejores. Para sites tradicionales, Lucky luce bien.
- **Ecosistema**: Lucky y Amber tienen mayor soporte de bibliotecas y componentes.
- **Familiaridad**: Si vienes de Rails, Athena o Amber pueden ser más familiares.

Analizando estos y otros factores para nuestro caso de uso, podemos seleccionar el framework más adecuado. Lo bueno es que siempre podemos cambiar más adelante si es necesario.

Un caso de uso común para Crystal es la creación de APIs JSON RESTful para ser consumidas por aplicaciones web, móviles, IoT, etc.

Por su enfoque minimalista y alto rendimiento, Kemal es una gran opción para APIs simples y rápidas. Veamos un ejemplo básico:

```
1  require "kemal"
2  # Endpoint para obtener todos los usuarios
3  get "/users" do
4    users = User.all
5    users.to_json
6  end
7  # Endpoint para crear un nuevo usuario
8  post "/users" do |env|
9    user = User.new(env.params.json)
10   user.save
11   if user.valid?
```


```
    user.to_json
  else
    env.response.status_code = 400
    {error: user.errors.to_s}.to_json
  end
end
Kemal.run
```

Con Kemal podemos crear rápidamente un API funcional, y agregar middlewares adicionales según sea necesario, como autenticación, manejo de errores, logging, etc.

Si necesitáramos una API más robusta, con más funcionalidades integradas, podríamos optar por Amber, que incluye soporte para OpenAPI, validaciones, JWT, GraphQL y más.

Para aplicaciones web tradicionales que generan HTML del lado del servidor, necesitamos renderizar templates o vistas. Lucky hace esto muy sencillo.

Por ejemplo, para una vista **src/pages/posts/index.cr**:

```
# Un action para renderizar la vista
class Posts::Index < BrowserAction
  get "/posts" do
    posts = PostQuery.new.recent
    render Page::Posts::IndexPage, posts: posts
  end
end
# Una page que define el contenido
class Page::Posts::IndexPage < MainLayout
  needs posts : PostQuery
  def content
    render_template("src/pages/posts/index.ecr")
  end
end
# Plantilla ECR para la vista
<h1>Latest Posts</h1>
<% posts.each do |post| %>
  <article>
    <h2><%= post.title %></h2>
    <p><%= post.body %></p>
  </article>
<% end %>
```

Aquí aprovechamos las convenciones de Lucky para separar la lógica del contenido, pasar datos a las vistas, y renderizar templates ECR para generar el HTML final.

Los tests son esenciales para construir aplicaciones confiables en cualquier framework. Veamos ejemplos de testing con Lucky y Amber.

Test de un action en Lucky:

```
# test/actions/users/show_test.cr
class Users::ShowTest < BrowserActionTest
  get "/users/1" do
    assert_response 200
    assert_content("username", "test")
  end
end
```

Podemos fácilmente probar acciones simulando requests y asserts sobre respuestas.

Test de un API controller en Amber:

```
# spec/controllers/users_controller_spec.cr
it "creates a user" do
  user = User.new(name: "John")
  response = Amber::Pipe::Pipeline.call(UsersController, :create, user)
  response.status_code.should eq(201)
  response.result.name.should eq("John")
end
```

Los tests en Amber son igualmente sencillos invocando el pipeline para simular requests.

Ambos frameworks ofrecen excelentes herramientas de testing integrados para cubrir tanto la lógica como las vistas.

La mayoría de aplicaciones web requieren persistir datos en una base de datos. Veamos cómo integrar base de datos en Lucky y Amber.

Con Lucky, simplemente debemos declarar nuestros modelos:

```
class Post < BaseModel
  table posts do
    field title : String
    field body : String
  end
end
```

Esto genera el schema, y Lucky integra el ORM Crecto para queries:

```
post = Post.find(1)
post.title # Accede a los campos
Post.insert(title: "New Post") # Inserta registro
```

En Amber es similar, usando Granite ORM:

```
class Post
```


```
  include Granite::ORM
  field title : String
  field body : String
  table posts do
    primary id : Int64
    timestamp
  end
end
post = Post.find 1
Post.insert({title: "Post", body: "Hello"})
```

Tanto Lucky como Amber facilitan la integración con la base de datos, abstracción ORM y queries.

Para soportar múltiples idiomas, los frameworks proveen helpers de i18n.

En Lucky:

```
# Traducciones en archivos YAML
es:
  welcome: "Bienvenido"
en:
  welcome: "Welcome"
# Uso en views
<%= t("welcome") %>
# Basado en locale
set_locale "es"
t("welcome") # Bienvenido
```

Y en Amber:

```
# Locale
current_locale = Amber::I18n.locale = :es
# Traducciones
en:
  welcome: "Welcome"
es:
  welcome: "Bienvenido"
# En views
{{ t("welcome") }}
```

Los frameworks facilitan la internacionalización, separando las cadenas traducibles de la lógica de negocio.

La seguridad es crucial en aplicaciones web. Los frameworks Crystal incorporan protección contra ataques CSRF (cross-site request forgery).

Por ejemplo, Lucky genera tokens CSRF y los valida en forms:

```
# Genera el tag para el form
<%= csrf_tag %>
# Valida el token en el controller
def create
  create_post = CreatePost.new(params)
  if create_post.valid?
    redirect to: Posts::Index
  else
    render NewPage, csrf_token: csrf_token
  end
end
```

De esta forma Lucky previene ataques CSRF de forma transparente.

Amber también tiene helpers CSRF:

```
# Genera el token
<%= csrf_tag %>
# Valida en el controller
def create
  amber_verify_csrf(context)
  # ...
end
```

Otra medida de seguridad importante es el manejo de sesiones. Por defecto Amber y Lucky usan cookies firmadas para las sesiones, pero también soportan backends como Redis para mejorar la escalabilidad.

Por ejemplo, en Amber:

```
# Usar Redis para sesiones
Amber::Server.configure do |settings|
  settings.session = {
    :redis_url => redis_url
  }
end
session["user_id"] = current_user.id
```

Y en Lucky:

```
# Configurar Redis store
Lucky::Session::Store.configure do
  settings.key = "app_session"
  settings.redis_url = redis_url
end
# Usar la sesión
```


```
7  session["user_id"] = current_user.id
```

Esto nos permite escalar las sesiones fácilmente.

Otra medida recomendada es el uso de HTTPS para cifrar la comunicación. Los frameworks soportan integración con servidores web como Nginx o Apache para habilitar TLS:

```
1  # Configuración de Nginx para HTTPS:
2  server {
3    listen 443 ssl;
4    ssl_certificate /path/to/cert.pem;
5    ssl_certificate_key /path/to/key.pem;
6    location / {
7      proxy_pass http://app_server;
8    }
9  }
```

De esta forma podemos servir nuestra aplicación de forma segura y cifrada.

En resumen, los frameworks Crystal vienen con buenas prácticas de seguridad incorporadas, pero siempre debemos considerar medidas adicionales según sea necesario, como TLS, autenticación y autorización, protección contra ataques comunes, actualizaciones frecuentes, etc. Una buena seguridad mejora la confianza de los usuarios en nuestra aplicación.

5.3. Bases de datos e ORMs

Las aplicaciones web modernas casi siempre requieren almacenar y recuperar datos de manera persistente. Para ello, se utilizan sistemas de bases de datos que permiten guardar grandes volúmenes de información de forma estructurada y con integridad.

Crystal provee excelente soporte para conectarse a bases de datos relacionales como PostgreSQL, MySQL, SQLite y Microsoft SQL Server. También permite integrarse con bases de datos NoSQL como MongoDB y Redis.

En esta sección exploraremos como trabajar con bases de datos en Crystal, utilizando ORMs (Object Relational Mappers) para simplificar el mapeo entre objetos de Crystal y tablas de bases de datos.

La forma más básica de interactuar con una base de datos desde Crystal es mediante consultas SQL directas. El driver de base de datos para Crystal permite ejecutar sentencias SELECT, INSERT, UPDATE y DELETE contra la base de datos.

Por ejemplo, para hacer una consulta contra una tabla "users" en PostgreSQL:

```
1  require "db"
2  DB.open "postgres://username:password@localhost/database" do |db|
3    rs = db.query "SELECT * FROM users WHERE age > 30"
4    rs.each do
5      puts "Name: #{rs.read(String)}"
6    end
7  end
```

La consulta devuelve un objeto ResultSet que podemos recorrer para acceder a cada fila. De forma similar podemos hacer insert, update y delete:

```
DB.open "postgres://username:password@localhost/database" do |db|
  # Insert
  db.exec "INSERT INTO users VALUES ($1, $2)", "John Doe", 30
  # Update
  db.exec "UPDATE users SET age=$1 WHERE name=$2", 40, "John Doe"
  # Delete
  db.exec "DELETE FROM users WHERE age > $1", 40
end
```

Si bien esto permite una interacción directa con la base de datos, tener consultas SQL embebidas en el código no es ideal. Es difícil de mantener, propenso a errores y vulnerable a inyección SQL.

Un mejor enfoque es utilizar un ORM.

Un ORM (Object Relational Mapper) permite mapear clases y objetos del lenguaje a tablas y registros de la base de datos de forma transparente.

En lugar de escribir consultas SQL directamente, se utiliza una API orientada a objetos provista por el ORM. Esto abstrae los detalles del almacenamiento y nos permite concentrarnos en trabajar con objetos de Crystal.

Algunos beneficios de usar un ORM:

- Abstracción del almacenamiento: no es necesario escribir SQL a mano.
- Mapeo automático entre objetos y tablas de la base de datos.
- Soporte para relaciones y asociaciones entre objetos.
- Carga perezosa y caching de objetos.
- Migraciones y gestión del esquema de la base de datos.
- Seguridad: previene inyección SQL.

Los ORMs más populares para Crystal son ActiveRecord, Crecto y Granite. Analicemos brevemente cada uno.

ActiveRecord está inspirado en el popular ORM de Ruby on Rails con el mismo nombre. Provee mapeo convencional entre clases y tablas, validaciones, asociaciones, consultas orientadas a objetos y otras características.

Para usar ActiveRecord, primero debemos definir nuestro modelo:

```
require "active_record"
class User < ActiveRecord::Base
  has_many :posts
end
class Post < ActiveRecord::Base
  belongs_to :user
end
```

Luego podemos realizar varias operaciones:

```
user = User.new
user.name = "John"
user.save
post = Post.new
post.user = user
post.title = "Hello world"
post.save
# Consultas
User.all
User.find(1)
Post.where(user_id: 1).order_by(:created_at)
```

ActiveRecord provee muchas funcionalidades adicionales como validaciones, callbacks, migraciones, etc. Y se integra bien con frameworks web como Lucky y Amber.

Crecto es un poderoso ORM escrito completamente en Crystal. Tiene un enfoque mucho más flexible que ActiveRecord, permitiendo personalizar mucho del mapeo objeto-relacional.

El uso básico es similar:

```
require "crecto"
class User < Crecto::Model
  has_many :posts, Post
end
user = User.new
user.name = "John"
user.save
# Consultas
query = Query(User, Post).join(:inner).where(name="John").preload(:posts)
```

Crecto se destaca por su velocidad, flexibilidad y capacidad de personalización. Es una gran opción si necesitamos un control más fino sobre el mapeo objeto-relacional.

Granite ORM sigue la filosofía "convención sobre configuración" de ActiveRecord. Permite definir modelos con pocas líneas de código:

```
require "granite/orm"
class User < Granite::Base
  adapter mysql
  primary id : Int64
  table_name users
  field name : String
  field email : String
  has_many :posts
end
```

Luego se pueden hacer consultas y persistencia de forma sencilla:

```
user = User.find 1
user.posts.each do |post|
  post.title # acceder a una post
end
user.email = "new@email.com"
user.save
```

Granite tiene un enfoque minimalista, proveyendo solo las características esenciales que se necesitan en la mayoría de los proyectos. Es fácil de aprender y muy productivo.

Framework	Filosofía	Curva de aprendizaje	Velocidad	Flexibilidad
ActiveRecord	Convencional	Sencilla	Rápido	Mediana
Crecto	Flexible	Intermedia	Muy rápido	Total
Granite	Minimalista	Sencilla	Rápido	Baja

En resumen:

- ActiveRecord es la opción más convencional y fácil de usar para la mayoría de los casos.
- Crecto es ideal cuando se necesita total control y personalización del ORM.
- Granite es excelente si buscamos un ORM simple y sin complicaciones.

Otros ORMs disponibles para Crystal son:

- **Clear::ORM**: Simple pero potente. Soporta PostgreSQL y MySQL.
- **Jennifer**: Generador de consultas con soporte para migraciones.
- **Crest**: ORM liviano que apunta a simplicidad y modularidad.
- **Topaz**: ORM basado en ActiveRecord muy prometedor.

Crystal también permite trabajar con facilidad con bases de datos NoSQL como Redis y MongoDB.

Por ejemplo, para usar Redis:

```
require "redis"
redis = Redis.new
redis.set("mykey", "Hello world")
puts redis.get("mykey")
```

Y para MongoDB con el driver oficial:

```
require "mongo"
client = Mongo::Client.new("mongodb://localhost:27017")
collection = client["testdb"]["testcollection"]
collection.insert_one({name: "John", age: 30})
result = collection.find({"name": "John"})
result.each do |document|
  puts document["name"]
```


```
8 end
```

Existen también wrappers y ORMs para facilitar el uso de bases de datos NoSQL desde Crystal, como credis (Redis), granite-mongo (MongoDB) y redis-orm (Redis).

- Crystal permite integrarse fácilmente con distintos motores de bases de datos relacionales y NoSQL.
- Para evitar escribir SQL directo, los ORMs mapean objetos de Crystal a tablas de la base de datos.
- Los ORMs más populares para Crystal son ActiveRecord, Crecto y Granite.
- ActiveRecord sigue el enfoque convencional de Ruby on Rails. Crecto es altamente personalizable. Granite apunta a la simplicidad.
- Para bases de datos NoSQL como Redis y MongoDB también existen bibliotecas y ORMs.

Usar un ORM es altamente recomendable para la mayoría de los proyectos web en Crystal que requieran persistencia. Permite abstraer la base de datos y centrarnos en la lógica de negocio de la aplicación.

Conclusión de este capítulo
Al finalizar este capítulo, habrás obtenido una sólida comprensión de las capacidades de Crystal para el desarrollo de aplicaciones web modernas. Aprendimos cómo construir servidores web y APIs RESTful que aprovechan la velocidad y concurrencia nativa de Crystal para un alto rendimiento y escalabilidad. También exploramos los principales frameworks web de Crystal, incluyendo Lucky, Amber y Kemal, analizando los casos de uso adecuados para cada uno. Otro aspecto importante que cubrimos fue la integración con bases de datos relacionales y NoSQL por medio de ORMs como ActiveRecord, Crecto y Granite. Esto nos permite mapear objetos de Crystal a tablas de bases de datos para persistir información. Finalmente, revisamos buenas prácticas y consejos para desarrollar aplicaciones web seguras, bien testeadas y listas para deploy y escalar cuando sea necesario.

Ejercicios y Preguntas Para Resolver

Ahora que hemos explorado los fundamentos de Crystal, desde las estructuras básicas de control hasta conceptos más avanzados como clases, funciones y algoritmos, es momento de poner a prueba nuestra comprensión con los siguientes ejercicios y preguntas.

Para consolidar los conocimientos adquiridos en este capítulo, te presentamos una serie de ejercicios y preguntas organizados por nivel de dificultad. Cada ejercicio incluye el tiempo estimado de resolución y los conceptos principales que evalúa, brindándote la oportunidad de reforzar tu dominio de los temas clave.

(★☆☆) Ejercicios Básicos

Tiempo estimado: 5-10 minutos por ejercicio

Estos ejercicios básicos te ayudarán a afianzar los conceptos fundamentales de Crystal, como aritmética, funciones, entrada de usuario y manipulación de datos.

#	Ejercicio	Conceptos Evaluados	Tiempo
1	Escribe un programa Crystal que calcule el área de un círculo dado su radio.	• Aritmética básica • Funciones	10 min

2	Crea un programa que tome dos números como entrada del usuario y muestre el mayor de los dos.	• Entrada de usuario • Condicionales	10 min
3	Implementa una función que tome una lista de números y devuelva la suma de todos ellos.	• Listas • Funciones	10 min
4	Escribe un programa que calcule el área y el perímetro de un rectángulo dado su largo y ancho.	• Aritmética básica • Funciones	10 min
5	Crea un programa que solicite al usuario su nombre y edad, y luego los imprima.	• Entrada de usuario • Impresión	5 min
6	Implementa una función que convierta una temperatura en Celsius a Fahrenheit.	• Funciones • Conversión de unidades	10 min

(★★☆) Ejercicios Intermedios

Tiempo estimado: 10-15 minutos por ejercicio

Los ejercicios intermedios te ayudarán a profundizar en conceptos más avanzados de Crystal, como estructuras de datos, algoritmos y manipulación de cadenas.

#	Ejercicio	Conceptos Evaluados	Tiempo
7	Escribe un programa que imprima los primeros 15 números de la secuencia de Tribonacci (una variante de Fibonacci).	• Ciclos • Recursividad	15 min
8	Crea una función que reciba una lista de strings y devuelva una nueva lista con todas las cadenas en mayúsculas.	• Listas • Manipulación de cadenas	10 min
9	Implementa una función que calcule el factorial de un número dado.	• Funciones • Matemáticas	15 min

(★★★) Ejercicios Avanzados

Tiempo estimado: 15-30 minutos por ejercicio

Los ejercicios avanzados te desafiarán a implementar algoritmos de ordenamiento, trabajar con clases y métodos, y desarrollar soluciones a problemas más complejos.

#	Ejercicio	Conceptos Evaluados	Tiempo
10	Escribe un programa que implemente el algoritmo de ordenamiento mergesort para una lista de números.	• Algoritmos de ordenamiento • Recursión	25 min
11	Crea una clase para representar un vector 3D con coordenadas x, y, z. Agrega métodos para calcular la magnitud, producto escalar y producto vectorial.	• Clases • Métodos matemáticos	30 min
12	Implementa un generador de números de Fibonacci que pueda calcular el término n-ésimo de la secuencia.	• Algoritmos matemáticos • Funciones	20 min

13	Escribe una función que tome una cadena de texto y devuelva un nuevo string con las palabras en orden inverso.	• Manipulación de cadenas • Iteración	15 min

Pistas y Ayudas

Para los ejercicios más complejos, aquí tienes algunas pistas que pueden orientarte:

Ejercicio	Conceptos Clave	Pistas para Resolución
10	Algoritmos de Ordenamiento	• Revisa el pseudocódigo del algoritmo Mergesort • Piensa en cómo dividir la lista en mitades y luego fusionar las sublistas ordenadas
11	Clases y Métodos Matemáticos	• Define los atributos necesarios para representar un vector 3D • Implementa métodos para calcular la magnitud, producto escalar y producto vectorial • Considera cómo podrías extender la clase en el futuro
12	Algoritmos Matemáticos	• Investiga cómo calcular los términos de la secuencia de Fibonacci • Piensa en una estructura de datos eficiente para almacenar los términos calculados • Usa ciclos o recursión para generar el término n-ésimo
13	Manipulación de Cadenas	• Separa la cadena en palabras • Crea una nueva cadena invirtiendo el orden de las palabras • Considera cómo podrías hacer esto de manera eficiente

Tabla de Autoevaluación

Usa esta tabla para evaluar tu comprensión de los conceptos clave del capítulo:

Concepto	¿Lo domino?	Ejercicios Relacionados
Aritmética y Funciones	☐	1, 4, 6, 9
Entrada/Salida y Impresión	☐	2, 5
Listas y Manipulación de Cadenas	☐	3, 8, 13
Algoritmos y Recursividad	☐	7, 10, 12
Clases y Métodos Matemáticos	☐	11

Reto adicional

Crea una pequeña aplicación en Crystal que calcule y muestre el factorial de un número ingresado por el usuario. Puedes agregar funcionalidades adicionales como manejo de errores o soporte para números negativos.

Bibliografía

Balbaert, I., & St. Laurent, S. (2019). *Programming Crystal: Create High-Performance, Safe, Concurrent Apps* (1st ed.). Pragmatic Bookshelf.

Johnson, R. (2024). *Mastering Crystal Programming: Combining Ruby Syntax with C-Like Performance* (1st ed.). HiTeX Press.

Dietrich, G., & Bernal, G. (2022). *Crystal Programming: A Project-Based Introduction to Building Efficient, Safe, and Readable Web and CLI Applications* (1st ed.). Packt Publishing.

Crystal Programming Language. (n.d.). Crystal. Crystal-lang.org. Recuperado de https://crystal-lang.org/

Crystal Programming Language. (n.d.). Documentación de Crystal. Crystal-lang.org. Recuperado de https://crystal-lang.org/docs/

Foro de la Comunidad del Lenguaje de Programación Crystal. (n.d.). Foro de Crystal. Recuperado de https://forum.crystal-lang.org/

Playground del Lenguaje de Programación Crystal. (n.d.). Crystal Playground. Recuperado de https://play.crystal-lang.org/

Agradecimientos finales

Este libro es, ante todo, un homenaje al aprendizaje autodidacta. Aprender por cuenta propia exige curiosidad sostenida, disciplina, paciencia y la valentía de persistir cuando aparecen las dudas. Quien ha recorrido estas páginas hasta el final y ha hecho el esfuerzo de comprender un lenguaje como Crystal, merece un reconocimiento especial: llegar aquí habla de mérito, constancia y deseo genuino de crecer.

Si el camino propuesto —su metodología, ejemplos y ejercicios— ayudó a mantener viva la curiosidad, a abrir la puerta de la teoría hacia la práctica y a cultivar la responsabilidad de acompañar a otras personas en su propio proceso de aprendizaje, entonces el propósito de esta guía ha valido la pena. Ese era el objetivo: que cada capítulo fuese una invitación a explorar, construir y pensar con criterio.

Agradezco, de manera amplia y sin menciones particulares, a todas las personas y espacios que hacen posible aprender y enseñar: a quienes comparten conocimiento, a quienes preguntan, a quienes corrigen y a quienes crean. Este proyecto también me ha impulsado a seguir buscando formas claras y útiles de transmitir lo que aprendo, con la esperanza de dejar un pequeño legado que inspire a otras personas a continuar su propia ruta de estudio, práctica y creación.

Con sincera gratitud,

Luis Eduardo Muñoz Guerrero