Structured Concurrency, Virtual Threads and Goroutines

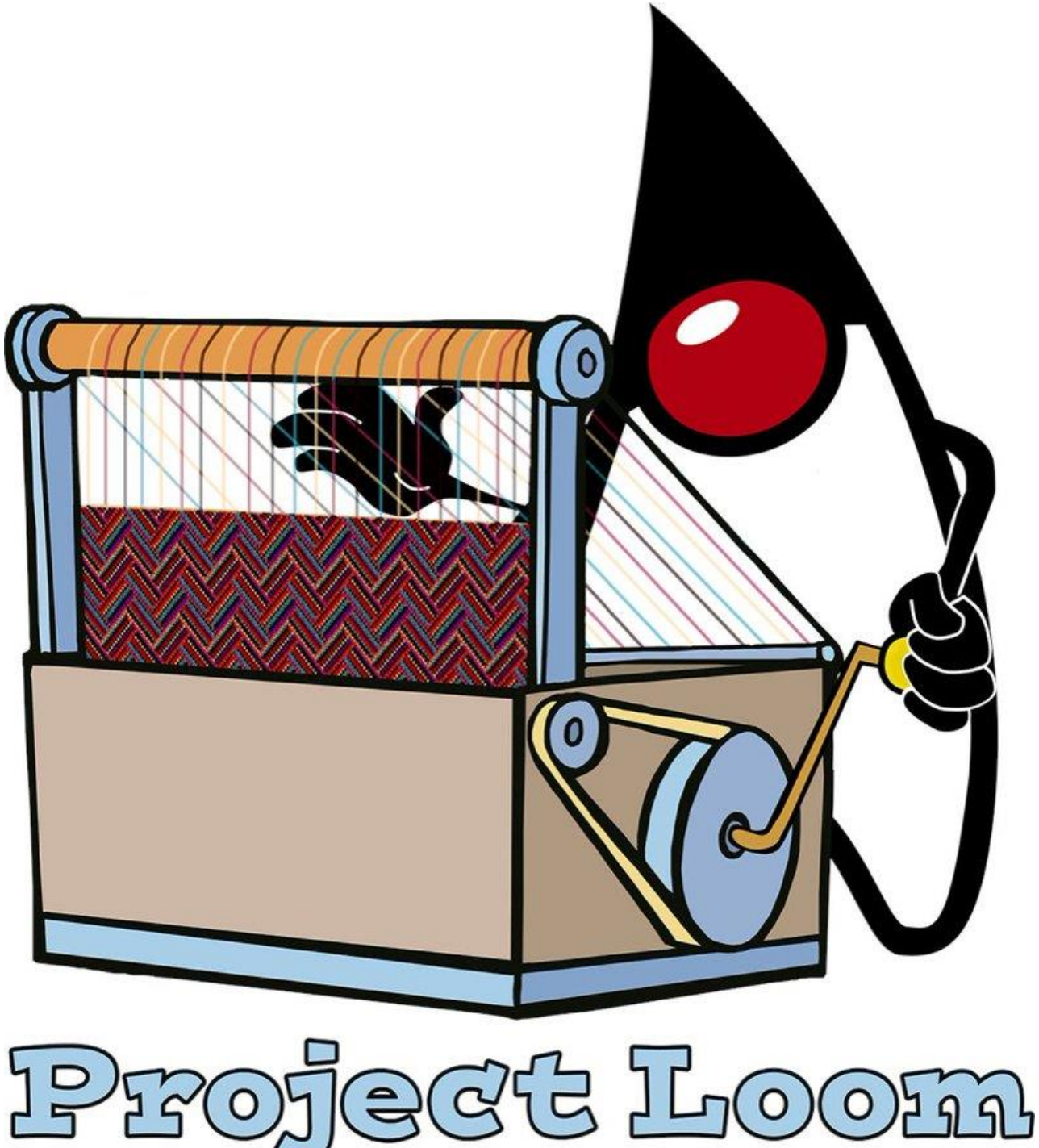
Java vs. Go

Where to Find The Code and Materials?

- https://github.com/iproduct/concurrency-java-vs-go

Project Loom

- Project Loom aims to drastically reduce the effort of writing, maintaining, and observing high-throughput concurrent applications that make the best use of available hardware.
- Led by Ron Pressler, work on Project Loom started in late 2017.
- Like all OpenJDK projects, it will be delivered in stages, with different components arriving in GA at different times.

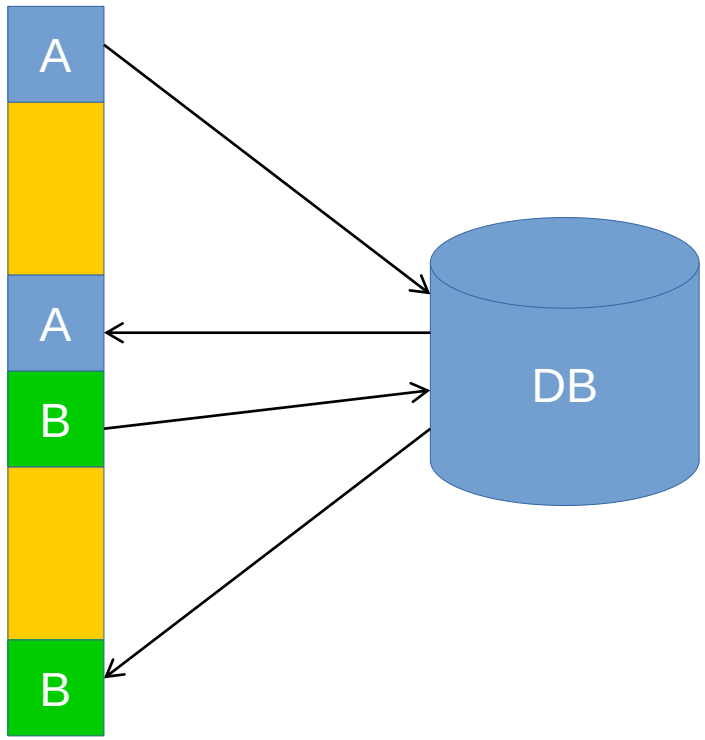


Virtual Threads

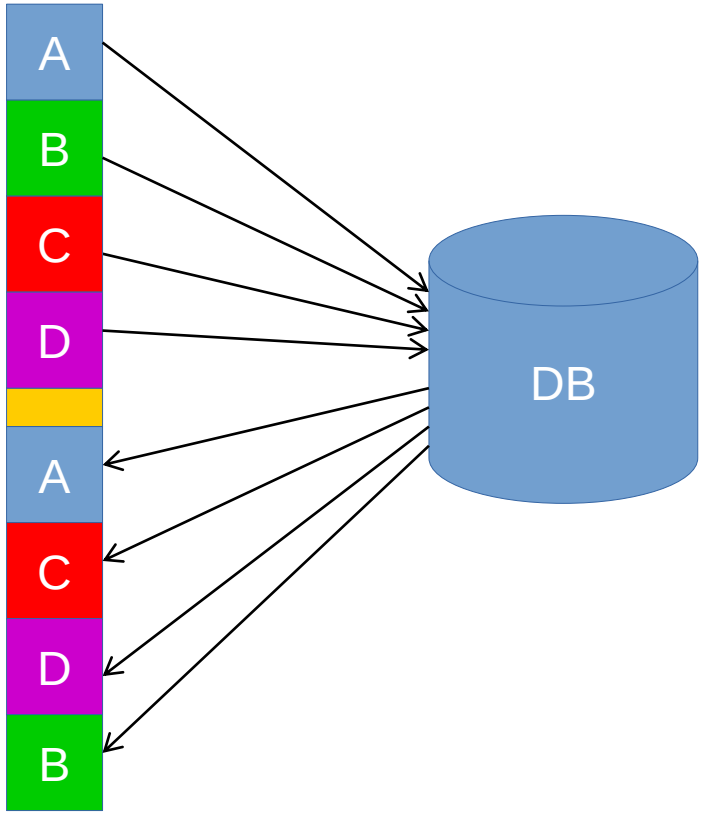
- A virtual thread is a Thread — in code, at runtime, in the debugger and in the profiler.
- A virtual thread is not a wrapper around an OS thread, but a Java entity.
- Creating a virtual thread is cheap — have millions, and don't pool them!
- Blocking a virtual thread is cheap — be synchronous!
- No language changes are needed.
- Pluggable schedulers offer the flexibility of asynchronous programming.

Synchronous vs. Asynchronous IO

Synchronous

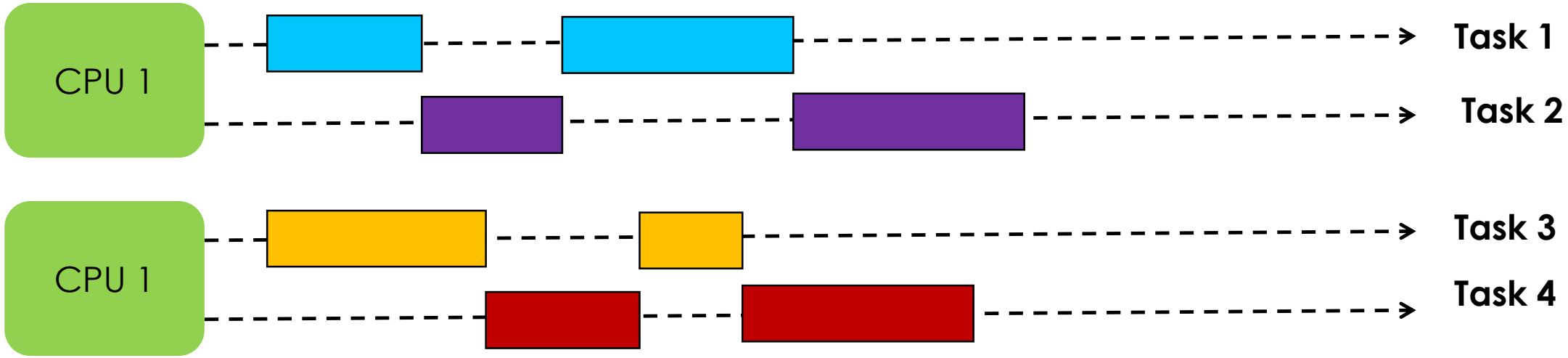


Asynchronous



Concurrency vs. Parallelism

- Concurrency refers to how a single CPU can make progress on multiple tasks seemingly at the same time (AKA concurrently).
- Parallelism allows an application to parallelize the execution of a single task - typically by splitting the task up into subtasks which can be completed in parallel.



Scalability Problem

- Scalability is the ability of a program to handle growing workloads.
- One way in which Java programs scale is parallelism: if we want to process a large chunk of data, we describe its transformation as a pipeline of lambdas on a stream, and by setting it to parallel we ask multiple processing cores to process their parts of the task simultaneously.
- The problem is that the thread, the software unit of concurrency, cannot match the scale of the application domain's natural units of concurrency — a session, an HTTP request, or a database transactional operation.
- A server can handle upward of a million concurrent open sockets, yet the operating system cannot efficiently handle more than a few thousand active threads. So it becomes a mapping problem - M:N

Solutions To Thread Scalability Problem

- Because threads are costly to create, we pool them -> but we must pay the price: leaking thread-locals and a complex cancellation protocol.
- Thread pooling is coarse grained – not enough threads for all tasks.
- So instead of blocking the thread, the task should return the thread to the pool while it is waiting for some external event, such as a response from a database or a service, or any other activity that would block it.
- The task is no longer bound to a single thread for its entire execution.
- Proliferation of asynchronous APIs, from NIO in JDK, through asynchronous servlets, to the many "reactive" libraries (Reactor, RxJava, etc.) => intrusive, all-encompassing and constraining frameworks, even basic control flow, like loops and try/catch, need to be reconstructed in "reactive" DSLs, supporting classes with hundreds of methods.

Problems with Async/Reactive Libraries

- Because, much of the Java platform assumes that execution context is embodied in a thread, all that context is lost once we dissociate tasks from threads:
  - Exception stack traces no longer provide a useful context;
  - When stepping in the debugger we find ourselves in scheduler code, jumping from one task to another;
  - Profiling I/O intensive application show us idle thread pools because tasks waiting for I/O do not hold their threads, and instead, return them to the pool;
- It is virally-intrusive and makes clean integration with synchronous code virtually impossible. [**What Color is Your Function?**]
- Synchronous APIs, from synchronization to I/O, that are duplicated – e.g. Kotlin: Thread.sleep() vs. delay()

What Color is Your Function?

[http://journal.stuffwithstuff.com/2015/02/01/what-color-is-your-function/]

- Synchronous functions return values, async ones do not and instead invoke callbacks.
- Synchronous functions give their result as a return value, async functions give it by invoking a callback you pass to it.
- You can't call an async function from a synchronous one because you won't be able to determine the result until the async one completes later.
- Async functions don't compose in expressions because of the callbacks, have different error-handling, and can't be used with try/catch or inside a lot of other control flow statements.
- Node's whole idea is that the core libs are all asynchronous. (Though they did dial that back and start adding ___Sync() versions of a lot functions.)

Async/Await

- Cooperative scheduling points are marked explicitly with await => scalable synchronous code – but we mark it as async – a bit of confusing!
- Solves the context issue by introducing a new kind of context that is like thread but is incompatible with threads – one blocks and the other returns some sort of Future or Flow or Flux - you can not easily mix sync and async code.

Right-Sized Threads

- If we could make threads lighter, we could have more of them, and can use them as intended:
  1. to directly represent domain units of concurrency;
  2. by virtualizing scarce computational resources;
  3. hiding the complexity of managing those resources.
- Not a new idea: Erlang, Go.
- **Virtual threads** are just threads, but creating and blocking them is cheap
- Managed by the Java runtime and implemented in userspace in the JDK, unlike the existing platform threads which are one-to-one wrappers of OS threads.

Why are OS Threads Heavy?

- Universal – represent all languages and types of workloads
- Can be suspended and resumed - this requires preserving its state, which includes the instruction pointer, as well as all of the local computation data, stored on the stack.
- The stack should be quite large, because we can not assume constraints in advance.
- Because the OS kernel must schedule all types of threads that behave very differently it terms of processing and blocking — some serving HTTP requests, others playing videos => its scheduler must be all-encompassing, and not optimized.

How Are Virtual Threads Better?

- The Java runtime knows how Java code makes use of the stack, so it can represent execution state more compactly.
- Direct control over execution also lets us pick schedulers - ordinary Java schedulers, that are better-tailored to our workload; we can use pluggable custom schedulers.
- Millions of virtual threads => every unit of concurrency in the application domain can be represented by its own thread
- Forget about thread-pools, just spawn a new thread, one per task.
- Example: HTTP request - a new virtual thread is already spawned to handle it, but now, in the course of handling the request, you want to simultaneously query a database, and issue outgoing requests to three other services? No problem: spawn more threads.

How Are Virtual Threads Better?

- You need to wait for something to happen without wasting precious resources – forget about callbacks or reactive stream chaining – just block!
- Write straightforward, boring code.
- VTs preserve all the benefits threads give us are preserved by : control flow, exception context, debugging flow, profiling organization; only the runtime cost in footprint and performance is gone.
- There is no loss in flexibility compared to asynchronous programming because, we have not ceded fine-grained control over scheduling.

Fibers or Virtual Threads?

- The Thread class carries a lot of luggage (more than 20 years):
  - Deprecated methods: suspend, resume, stop and countStackFrames
  - context-classloader
  - ThreadGroup, Thread.enumerate
- Thread.currentThread() and ThreadLocal – extensively used

How to Create VTs?

```
Thread t = Thread.startVirtualThread(() ->
System.out.println("Hello, Loom!"));

Thread t2 = Thread.ofVirtual().unstarted(() ->
System.out.println("Hello, Loom!"));
t2.start();
```

```
ThreadFactory factory = Thread.ofVirtual().name("worker", 0).factory();
Thread t3 =factory.newThread(() -> {
    System.out.println("Hello, Loom!");
});
t3.start();
```

- A new method, Thread.isVirtual(), can be used to distinguish between the two implementations, but otherwise they are interchangable

Using Executor Service - I

```
ThreadFactory tf = Thread.ofVirtual().factory();
var deadline = Instant.now().plusSeconds(2);
ExecutorService e = Executors.newThreadExecutor(tf, deadline);


// spawns a new virtual thread
Future<String> f = e.submit(() -> "Hello, Loom!");


String result = f.get(); // joins the virtual thread
System.out.printf("Result: %s%n", result);
```

Using Executor Service - II

```
var deadline = Instant.now().plusSeconds(2);
ExecutorService e = Executors.newVirtualThreadExecutor(deadline);


// spawns a new virtual thread
Future<String> f = e.submit(() -> "Hello, Loom!");

String result = f.get(); // joins the virtual thread
System.out.printf("Result: %s%n", result);
```

What New Concepts You Should Learn?

- Using virtual threads well does not require learning new concepts so much as it demands we unlearn old habits developed over the years to cope with the high cost of threads and that we've come to automatically associate with threads merely because we've only had the one implementation.
- Every task, within reason, can have its own thread entirely to itself; there is never a need to pool them – we just let the entire task run start-to-finish, in its own thread, and use a semaphore in the service-call code to limit concurrency.

Scheduling of VTs

- By default, virtual threads are scheduled by a global scheduler with as many workers as there are CPU cores (or as explicitly set with -Djdk.defaultScheduler.parallelism=N)
- Initially, the default global scheduler is the work-stealing ForkJoinPool – good default for many applications where tasks come on short bursts, such as transactions and message processing.
- Virtual threads are preemptive, not cooperative — they do not have an explicit await operation at scheduling (task-switching) points. Rather, they are preempted when they block on I/O or synchronization.

Preemptiive Scheduling

- Platform threads are sometimes forcefully preempted by the kernel if they occupy the CPU for a duration that exceeds some allotted time-slice. Time-sharing works well as a scheduling policy when active threads don't outnumber cores by much and only very few threads are processing-heavy.
- Whith VTs when we have millions of threads, this policy is less effective: if many of them are so CPU-hungry that they require time-sharing, then we're under-provisioned and no scheduling policy could save us.
- In all other circumstances, either a work-stealing scheduler would automatically smooth over sporadic CPU-hogging or we could run problematic threads as platform threads and rely on the kernel scheduler. For this reason, none of the schedulers in the JDK currently employs time-slice-based preemption of virtual threads => Forced Preemption.

Custom Scheduling Example

```
ExecutorService pool = Executors.newFixedThreadPool(4);
Executor scheduler = (task) -> {
    Thread vthread = ((Thread.VirtualThreadTask) task).thread();
    System.out.println(vthread);
    pool.execute(task);
};


Thread thread = Thread.ofVirtual()
        .name("VirtualThread_01")
        .scheduler(scheduler)
        .start(() -> System.out.printf("Hello from thread %s%n",
                                       Thread.currentThread().getName()));
thread.join();
```

Structured Concurrency - I

- **Structured concurrency** <=> when a task splits into concurrent tasks, they must join up again. If a main task splits into several concurrent sub-tasks to be executed by spawning threads then those threads must terminate before the main task can complete.
- Useful abstraction: the caller of a method that is invoked to do a task should not care if the method decomposes the work into sub-tasks that are executed by a million threads - when the method completes then all threads scheduled by the method should have terminated.
- An early of prototype of Project Loom had API named FiberScope to support the scheduling of fibers (a precursor to virtual threads) with initial support in this area.

Structured Concurrency Support

- There is no explicit support in the current prototype but it is possible to use existing constructs without needing too many new APIs.
- ExecutorService has been retrofitted to extend AutoCloseable
- Executors has been updated to define a number of static factory methods that support usage in a structured manner.
- If a thread blocked in ExecutorService::close is interrupted then it will attempt to stop all tasks/threads as if by invoking the exector's shutdownNow method. If all tasks are well behaves and terminate quickly when interrupted then it allow the executor to terminate quickly.

Example1: Structured Concurrency

```
void top() {
    try (ExecutorService executor = Executors.newVirtualThreadExecutor()) {
        executor.submit(List.of(() -> foo().join(), () -> bar().join()))
                .filter(Future::isCompletedNormally)
                .map(Future::join)
                .forEach(System.out::println);
    }
}
Future<String> foo() {
    try (ExecutorService executor = Executors.newVirtualThreadExecutor()) {
        return executor.submit(() -> "foo");
    }
}
Future<String> bar() {
    try (ExecutorService executor = Executors.newVirtualThreadExecutor()) {
        return executor.submit(() -> "bar");
    }
}
```

Example2: Structured Concurrency

```
void top() {
    var deadline = Instant.now().plusSeconds(2);
    try (ExecutorService executor = Executors.newVirtualThreadExecutor(deadline)) {
        executor.submit(List.of(() -> foo().join(), () -> bar().join()))
                .filter(Future::isCompletedNormally)
                .map(Future::join)
                .forEach(System.out::println);
    }
}
Future<String> foo() {
    try (ExecutorService executor = Executors.newVirtualThreadExecutor()) {
        return executor.submit(() -> "foo");
    }
}
Future<String> bar() {
    try (ExecutorService executor = Executors.newVirtualThreadExecutor()) {
        return executor.submit(() -> "bar");
    }
}
```

Example3: Structured Concurrency - Canceling

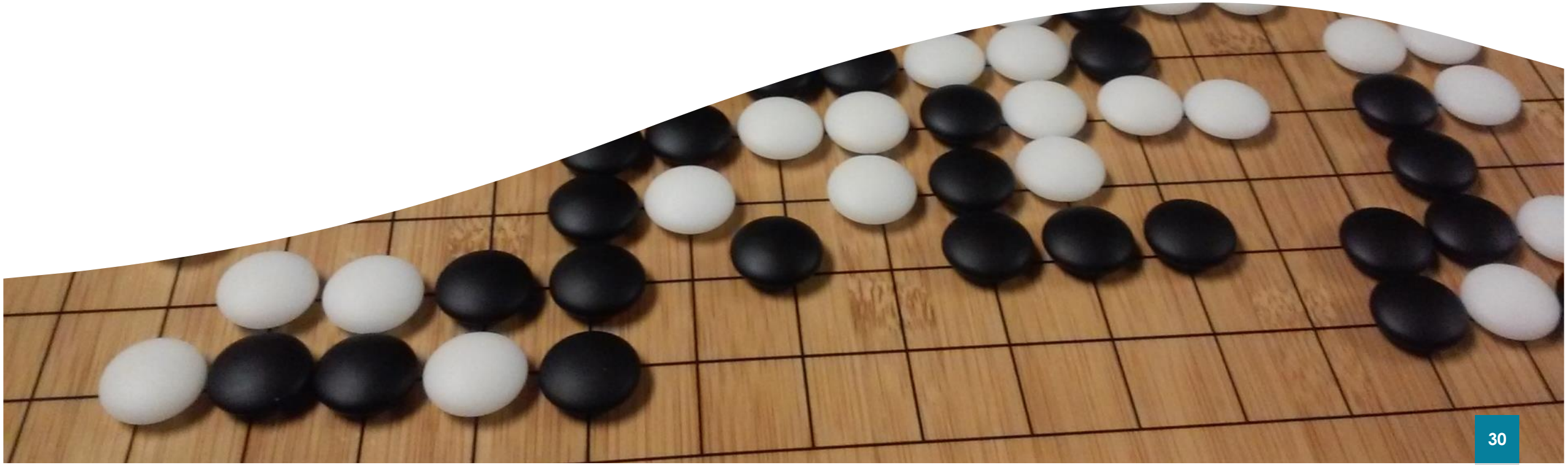
```
try (var e = Executors.newVirtualThreadExecutor()) {
    Stream<Future<String>> tasks = e.submit(List.of(
          () -> {Thread.sleep(10000); return "a";},
          () -> {Thread.sleep(500); return "b"; },
            () -> {
                throw new IOException("too lazy for work");
            }
    ));

    String first = tasks
            .filter(Future::isCompletedNormally)
            .map(Future::join)
            .findFirst()
            .orElse(null);
    System.out.println("one result: " + first); // prints: one result: b
    e.shutdownNow();
}
```

Example 4: Structured Concurrency – Scoped Vars

```
static final ScopeLocal<String> sv = ScopeLocal.forType(String.class);
void foo() {
    ScopeLocal.where(sv, "A").run(() -> {
        bar();
        baz();
        bar();
    });
}
void bar() {
    System.out.println(sv.get()); // prints: A, B, A
}
void baz() {
    ScopeLocal.where(sv, "B").run(() -> {
        bar();
    });
}
```

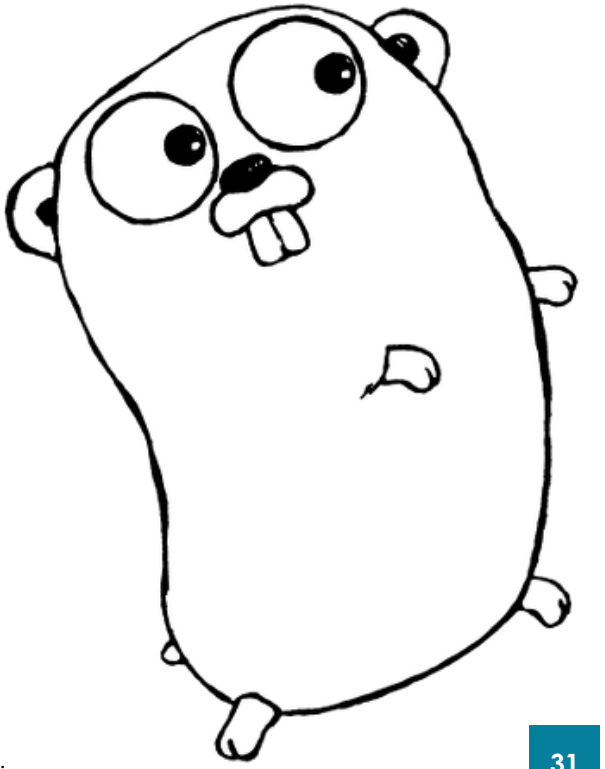
Golang

History, main features, advantages



Origins of GO

- Go was conceived in September 2007 by Robert Griesemer, Rob Pike, and Ken Thompson, at Google.
- It was publically announced in November 2009, and version 1.0 was released in March 2012.
- Go is widely used in production at Google and in many other organizations and open-source projects.



Aims of Go

- The aim of Go language, was to fill the same niche today that C fit into in the '80s.
- According to Moore's law, the number of transistors on a CPU can be expected to double roughly every 18 months => now more cores
- It is a low-level language for multiprocessor development.
- Experience with C taught that a successful systems programming language ends up being used for application development.
- Go incorporates a number of high-level features, allowing developers to use it for things like web services or desktop applications, as well as very low-level systems.

Who Uses Go?

- Docker, a set of tools for deploying Linux containers
- Ethereum, blockchain for the *Ether* cryptocurrency
- InfluxDB, an open source database specifically to handle time series data with high availability and high performance requirements.
- Juju, a service orchestration tool by Canonical, packagers of Ubuntu
- Kubernetes container management system
- OpenShift, a cloud computing platform as a service by Red Hat
- Terraform, an open-source, multiple cloud infrastructure provisioning

Who Uses Go?

- Cloud Foundry, a platform as a service
- Container Linux (formerly CoreOS), a Linux-based operating system that uses Docker containers and rkt containers.
- Couchbase, Query and Indexing services within the Couchbase Server
- Dropbox, migrated some of their critical components from Python to Go
- Heroku, for Doozer, a lock service
- MongoDB, tools for administering MongoDB instances
- Netflix, for two portions of their server architecture
- Uber, for handling high volumes of geofence-based queries

Why Go?

- *Minimalism* - Go language specification is only 50 pages, with examples, easy to read. Core language consists of a few **simple, orthogonal** features that can be combined in a relatively small number of ways.
- *Code transparency* - your need to understand your code:
  - you **always** need to know **exactly what** your coding is doing;
  - you **sometimes** need to **estimate the resources** (time and memory) it uses;
  - one standard code format, automatically generated by the **fmt** tool.
- *Compatibility* - Go 1 has succinct and strict compatibility guarantees for the core language and standard packages. BSD-style license.
- *Performance* - compiled language, single standalone binary, low latency garbage collection, optimized standard libraries, fast build, scales well.

Go Main Features

- *Static typing* and *run-time efficiency* (like C++)
- *Syntax and environment patterns* more common in dynamic languages
- *Readability*, *usability* and *simplicity*
- *Fast compilation* times
- *High-performance networking* and *multiprocessing*
- Optional *concise variable declaration* and initialization through *type inference* (x := 0 not int x = 0; or var x = 0;).
- Remote *package management (go get)* and online *package documentation*.

Distinctive Approaches to Particular Problems

- Go is *strongly* and *statically* typed with no implicit conversions, but the syntactic overhead is small by using simple type inference in assignments together with untyped numeric constants.
- An *interface* system in place of *virtual inheritance*, and *type embedding* instead of *non-virtual inheritance*.
- Structurally typed *interfaces* provide *runtime polymorphism* through *dynamic dispatch*.
- Programs are constructed from *packages* that offer clear code separation and allow efficient management of dependencies.
- Built-in concurrency primitives: light-weight processes *(goroutines)*, *channels*, and the *select* statement

Distinctive Approaches to Particular Problems

- A toolchain that, by default, produces statically linked *native binaries without external dependencies*.
- Built-in frameworks for *testing* and *profiling* are small and easy to learn, but still fully functional.
- It's possible to *debug* and *profile* an optimized binary running in production through an HTTP server.
- Go has *automatically generated documentation* with *testable examples*.

Built-in Types

- Strings are provided by the language; a string behaves like a slice of bytes, but is immutable.
- Hash tables are provided by the language. They are called maps.

Pointers and References

- Go offers pointers to values of all types, not just objects and arrays. For any type T, there is a corresponding pointer type *T, denoting pointers to values of type T.
- Arrays in Go are values. When an array is used as a function parameter, the function receives a copy of the array, not a pointer to it. However, in practice functions often use slices for parameters; slices are references to underlying arrays.
- Certain types (maps, slices, and channels) are passed by reference, not by value. That is, passing a map to a function does not copy the map; if the function changes the map, the change will be seen by the caller. In Java terms, one can think of this as being a reference to the map.

Error Handling

- Instead of exceptions, Go uses errors to signify events such as end-of-file;
- And run-time panics for run-time errors such as attempting to index an array out of bounds.

Object-Oriented Programming

- Go does not have classes with constructors. Instead of instance methods, a class inheritance hierarchy, and dynamic method lookup, Go provides structs and interfaces.
- Go allows methods on any type; no boxing is required. The method receiver, which corresponds to this in Java, can be a direct value or a pointer.
- Go provides two access levels, analogous to Java's public and package-private. Top-level declarations are public if their names start with an upper-case letter, otherwise they are package-private.

Functional Programming. Concurrency

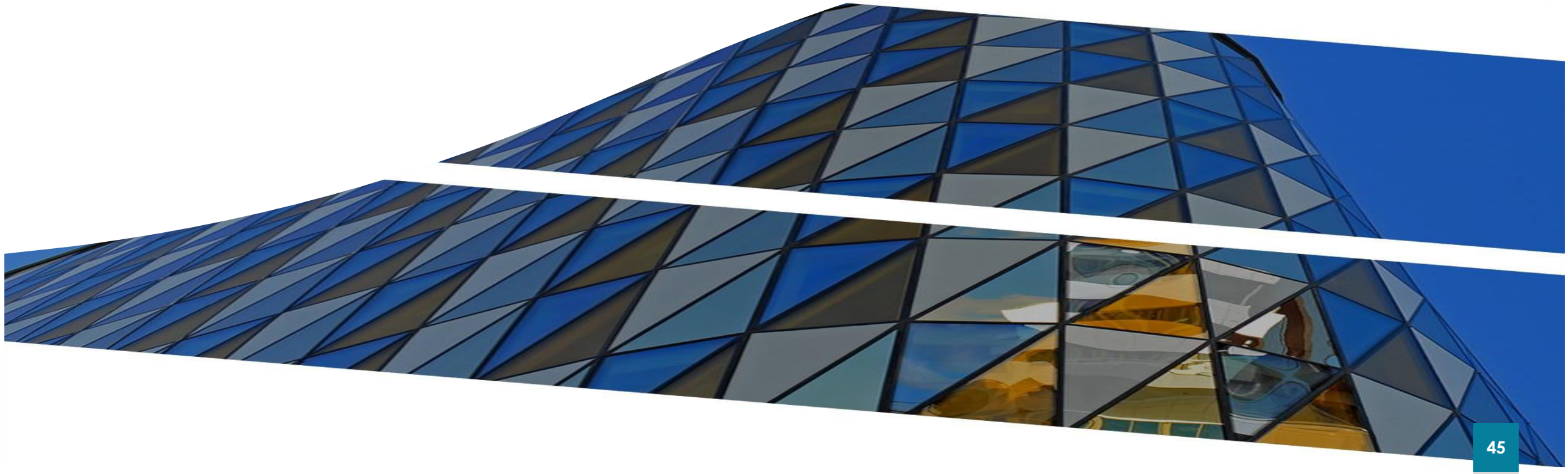
- Functions in Go are first class citizens. Function values can be used and passed around just like other values and function literals may refer to variables defined in a enclosing function (closure).
- Concurrency: Separate threads of execution, goroutines, and communication channels between them, channels, are provided by the language.

Omitted Features

- Go does not support implicit type conversion. Operations that mix different types require an explicit conversion. Instead Go offers Untyped numeric constants with no limits.
- Go does not support function overloading. Functions and methods in the same scope must have unique names. As alternatives, you can use optional parameters.
- Go has some built-in generic data types, such as slices and maps, and generic functions, such as append and copy.

Go Basic Syntax

Download, installation, environment setup



The Structure of a Go Source File

Go code is arranged in **packages**, which fill the roles of both libraries and header files in C

```
package main

import "fmt"

func main() {

      fmt.Println("Hello, world!")

}
```

Every program must contain a **main** package, which contains a **main()** function, which is the program entry point

fmt package has been imported, any of its exported **types, variables, constants**, and **functions** can be used, prefixed by the package name; packages are imported when the code is linked, rather than when it is run; **access control** in Go is available only at package level.

Println() exported (public) function prints the text on the console

Creating Simple Library Package

```go
// Package stringutil contains utility functions for working with strings.
package stringutil

// Reverse returns its argument string reversed rune-wise left to right.
func Reverse(s string) string {
    r := []rune(s)
    for i, j := 0, len(r)-1; i < len(r)/2; i, j = i+1, j-1 {
        r[i], r[j] = r[j], r[i]
    }
    return string(r)
}
```

Using It

```
package main

import "fmt"
import "github.com/iproduct/coursego/simple/stringutil"

func main() {
    s := "Hello Go World!"
    fmt.Println(s)
    fmt.Println(stringutil.Reverse(s))
}
```

More Examples:

https://github.com/iproduct/coursego

- Variables
- Loops
- Functions
- Enums
- Structures and Methods
- Interfaces
- Polymorphism
- Casting
- Errors
- Http Client and Server

Golang Concurrency Example: Goroutines, Channels, Context

https://github.com/iproduct/concurrency-java-vs-go/blob/main/goroutines-channels/downloader-using-context/downloader-using-context.go

Recommended Literature

- State of Loom, Ron Pressler, May 2020: http://cr.openjdk.java.net/~rpressler/loom/loom/sol1_part1.html
- OpenJDK Wiki - Structured Concurrency: https://wiki.openjdk.java.net/display/loom/Structured+Concurrency
- The Go Documentation - https://golang.org/doc/
- The Go Bible: Effective Go - https://golang.org/doc/effective_go.html
- David Chisnall, *The Go Programming Language Phrasebook*, Addison Wesley, 2012
- Alan A. A. Donovan, Brian W. Kernighan, *The Go Programming Language*, Addison Wesley, 2016

Thank's for Your Attention!

Trayan Iliev

IPT – Intellectual Products & Technologies

http://iproduct.org/

http://robolearn.org/

https://github.com/iproduct

https://twitter.com/trayaniliev

https://www.facebook.com/IPT.EACAD