The Ballerina programming language

Version: 2021-03-08

James Clark

Purpose of this presentation

- Describes the Ballerina programming language
- Applies to 2021 version of Ballerina ("Swan Lake")
- Covers all significant features
- Ballerina is a comprehensive platform, including extensive libraries: this presentation is only about the language
- Intended for experienced programmers, familiar with at least one C family language
    - C, C++, Java, JavaScript, C#, TypeScript
    - also assume familiarity with static typing: so not just JavaScript
- Ballerina Language Specification is more precise and detailed, but harder to understand

Presentation has three parts

1. How Ballerina provides basic functionality common to most programming languages
2. What makes Ballerina distinctive
3. Completing the picture

Part 1
How Ballerina does what all programming languages do

Familiar subset of Ballerina

- Many of the most widely used programming languages today are based on C
- Most importantly: C, C++, Java, C#, JavaScript, TypeScript
- They have a lot of commonality in how they provide basic functionality
- Ballerina is designed to take advantage of this
- Part 1 describes a subset of Ballerina having maximum commonality with these languages
- If the subset was all there was, it would be uninteresting
- Provides a foundation for using the distinctive features described in Part 2
- Ballerina is not a small language, but understanding a small subset is enough to get started

Programs and modules

```
import ballerina/io;

public function main() {
   io:println("Hello world!");
}
```

- Program consists of modules
- Modules are one or more `.bal` files
- Modules define named functions
- Module names look like *`org/x.y.z`*
- Standard library uses `ballerina` org
- `import` binds prefix to module name
- Prefix defaults to last part of module name
- Override with `import` *`org/x`* `as` *`m`*
- *m*:*f* means function *f* in module bound to prefix *m*
- `main` function is the program entry point
- `public` makes function visible outside module

Variables and types

```
import ballerina/io;

string greeting = "Hello";

public function greet() {
   string name = "James";
   io:println(greeting, " ", name);
}
```

- Modules and functions can declare variables
- A variables has a type, which constrains what values the variable can hold
- There is a built-in set of named types, including `int`, `float`, `boolean`, `string`
- Assignments are statements not expressions

Functions

```
function add(int x, int y) returns int {
   int sum = x + y;
   return sum;
}
```

- Parameters are declared as in C
- Not allowed to assign to parameters
- `return` statement returns value
- `returns` keyword specifies type of return value
- Function body contain statements

Syntax

```
// This is a comment
int count = 0;

// You can have Unicode identifiers
function พิมพ์ชื่อ(string ชื่อ) {
   io:println(ชื่\u{E2D});
}

string 'string = "xyz";
```

- Comments are // to end of line
- Module definitions/declarations and statements either use braces are terminated by semicolon
- Semicolons are not optional
- Identifier syntax like C
  - Keywords are reserved
  - Prefix reserved keyword with single quote
  - Prefix non-identifier character with \
  - Use \u{H} to specify character using Unicode code point in hex
  - Unicode characters also allowed
- Overall syntax is C-like

Integers

```
int m = 1;

int n = 0xFFFF;
```

- `int` type is 64-bit signed integers (same as `long` in Java)
- Integer literals can be hexadecimal (but not octal)
- Usual arithmetic operators: `+ - * / %`
- Operator precedence as in C
- Do *not* have increment or decrement operators
- Have compound assignment operations e.g. `+=`, `-=`
- Integer overflow is a runtime error
- Usual comparison operators: `== != < > <= >=`

Floating point numbers

```
float x = 1.0;

float y = x + <float>n;
```

- `float` type is IEEE 64-bit binary floating point (same as `double` in Java)
- Same operators as int
- No implicit conversion between integer and floating point values
- Use <*T*> for explicit conversions
- NaN is `==` to itself: `==` and `!=` on `float` test for same value not IEEE numerically equal

Booleans and conditionals

```
boolean flag = true;

// conditional expression
int n = flag ? 1 : 2;

function foo() {
   if flag {
      io:println(1);
   } else {
      io:println(2);
   }
}
```

- `boolean` type has two values: `true`, `false`
- Conditional expressions use C syntax
- Curly braces are required in if/else and other compound statements
- Parentheses are optional before curly braces
- `!` operator works on booleans only
- && and || operators short-circuit as in C

Nil

```
// value of v can be an int or ()
int? v = ();

// value of n cannot be ()
int n = v == () ? 0 : v;

// ?: operator
int n = v ?: 0;

function foo() {
}

function foo() returns () {
  return ();
}
```

- Ballerina's version of null is called nil and written `()`
- Types do not implicitly allow nil
- Type *T*`?` means *T* or nil
- Use `==` and `!=` to test whether a value is nil: no implicit conversion to boolean
- Elvis operator x ?: y returns x if it is not nil and y otherwise
- No `void` type
- Leaving off return type is equivalent to `returns ()`
- Falling off the end of a function or `return` by itself is equivalent to `return ()`

Strings

```
string grin = "\u{1F600}";

string greeting = "Hello" + grin;
```

- `string` type is immutable sequence of zero or more Unicode characters
- == if sequence has same characters
- String literals use double quotes
  - Usual C escapes e.g. `\n` `\t`
  - Numeric escapes specify Unicode code point using one or more hex digits `\u{H}`
- Concatenation uses + operator
- No separate character type: a character is represented by string of length 1
- `s[i]` accesses character at index `i` (zero-based)
- `< <= > >=` work by comparing code points
- Unpaired surrogates are not allowed

Langlib functions

```
string s = "abc".substring(1, 2);

// n will be 1
int n = s.length();

// Same as
int n = string:length(s);
```

- Langlib is small library defined by language providing fundamental operations on built-in datatypes
- Langlib functions can be called using convenient method-call syntax
- But these types are *not* objects!
- ballerina/lang.T module for each built-in type T
- Automatically imported using T prefix
- Standard library extends this with rich collection of modules: not part of this presentation

Arrays

```
int[] v = [1, 2, 3];
int n = v[0];
// result will be 3
int len = v.length();
```

- *T*[] is an array of *T*
- v[i] does indexed access
- Arrays are mutable: v[i] is an lvalue
- == and != on arrays is deep: two arrays are equal if they have the same members in the same order
- Ordering is lexicographical based on ordering of members
- Langlib arr.length() function gets the length; arr.setLength(n) sets the length

`foreach` statement

```
function sum(float[] v) returns float {
   float r = 0.0;
   foreach float x in v {
      r += x;
   }
   return r;
}
```

```
function sum(float[] v) returns float {
   float r = 0.0;
   foreach int i in 0 ..< v.length() {
      r += v[i];
   }
   return r;
}
```

- `foreach` iterates over an array, by binding a variable to each member of the array in order
- `m ..< n` creates a value that when iterated over will give the integers starting from `m` that are < n
- `foreach` also works for strings, and will iterate over each character of the string

`while` statement

```
type LinkedList record {
   string value;
   LinkedList? next;
};

function len(LinkedList? ll)
                  returns int {
   int n = 0;
   while ll != () {
      n += 1;
      ll = ll.next;
   }
   return n;
}
```

- More flexible iteration than `foreach`
- Usual `break` and `continue` statements

Binary data

```
byte[] data = base64`
   yPHaytRgJPg+QjjylUHakE
   wz1fWPx/wXCW41JSmqYW8=
`;

int x = 0xDEADBEEF;

// OK because byte & int
// will be byte
byte b = x & 0xFF;
```

- Binary data is represented by arrays of byte values
- Special syntax for byte arrays in base 64 and base 16
- Relationship between byte and int not the same as what you are used to
- A `byte` is an `int` in the range 0 to 0xFF
- `byte` is a subtype of `int`
- `int` type supports normal bitwise operators: `& | ^ ~ << >> >>>`
- Ballerina knows the obvious rules about when bitwise operations produce a byte

Maps

```
map<int> m = {
   "x": 1,
   "y": 2
};

int? v = m["x"];

m["z"] = 5;

// m["x"] wouldn't work because
// type would be int? not int
m["z"] = m.get("x");
```

- map<T> is a map from strings to T
- map syntax like JSON
- m[k] gets entry for k; nil if missing
- Use m.get(k) when you know that there's an entry for k
- Maps are mutable: m[k] is an lvalue
- `foreach` will iterate over values of the map
- Iterate over keys by using m.keys() to get the keys as an array of strings
- == and != on maps is deep: two maps are equal if they have the same set of keys and the values for each key are equal

Type definitions

```
type MapArray map<string>[];
MapArray arr = [
   {"x": "foo"},
   {"y": "bar"}
];
```

- Type definition gives a name for a type
- Name is just an alias for the type, like typedef in C

Records

```
record { int x; int y; } r = {
  x: 1,
  y: 2
};

type Coord record {
  int x;
  int y;
};

Coord c = { x: 1, y: 2 };

int x = c.x;
```

- A record type has specific named fields
- Access fields with r.x
- Records are mutable: r.x is an lvalue
- Construct using similar syntax to a map
- Typically combined with type definition
- As usual, name of type is not significant: record is just a collection of fields
- Record equality is like map equality

Structural typing

- Typing in Ballerina is structural: a type describes a set of values
- Semantic subtyping: subtype means subset
- Universe of values is partitioned into "basic" types
  - each value belongs to exactly one basic type
  - can think of each value as being tagged with its basic type
- There is complexity in making structural typing work with mutation

Immutable basic types (so far):

- nil
- boolean
- int
- float
- string

Mutable basic types (so far):

- array
- map and record

Unions

```
type StructuredName record {
   string firstName;
   string lastName;
};
type Name StructuredName|string;

function nameToString(Name nm)
                  returns string {
   if nm is string {
      return nm;
   }
   else {
      return nm.firstName
          + " " + nm.lastName;
   }
}
```

- $T_1|T_2$ is the union of the sets described by T_1 and T_2
- T? is completely equivalent to T|()
- Unions are untagged
- `is` operator tests whether value belongs to type
- `is` operator in condition causes declared type to be narrowed

Error reporting

```
function parse(string s)
              returns int|error {
   int n = 0;
   int[] cps = s.toCodePointInts();
   foreach int cp in cps {
      cp -= 0x30;
      if cp < 0 || cp > 9 {
         return error("not a digit");
      }
      n = n*10 + cp;
   }
   return n;
}
```

- Ballerina does not have exceptions
- Errors are reported by functions returning error values
- `error` is its own basic type
- An error value includes a string message
- Return type will be union with error
- Return type of `error?` used when the only values explicitly returned are errors
- Error value includes stack trace from point where error(msg) is called
- Error values are immutable

Error handling

```
// Convert bytes to a string
// and then to an int
function intFromBytes(byte[] bytes)
            returns int|error {
   string|error ret
     = string:fromBytes(bytes);
   if ret is error {
      return ret;
   }
   else {
      return int:fromString(ret);
   }
}
```

- Usually a function handles errors by passing them up to its caller
- `main` can return an error
- Can use `is` operator to distinguish errors from others value
- There's a shorthand for this pattern

check expression

```
// Convert bytes to a string
// and then to an int
function intFromBytes(byte[] bytes)
           returns int|error {
   string str =
      check string:fromBytes(bytes);
   return int:fromString(str);
}
```

- `check` *E* is used with expression *E* that might result in an error
- If *E* does result in an error, then `check` makes the function return that error immediately
- Type of `check` *E* does not include error
- Control flow remains explicit

Error subtyping

```
type XErr distinct error;
type YErr distinct error;
type Err XErr|YErr;

Err err = error XErr("Whoops!");

function desc(Err err)
                  returns string {
   return err is XErr ? "X" : "Y";
}
```

- `distinct` creates new subtype
- Use name of distinct type with error constructor to create error value
- Works like a nominal type: `is` operator to can distinguish distinct subtypes
- Each occurrence of `distinct` has a unique identifier, used to tag instances of the type

Panics

```
// n must not be 0
function divide(int m, int n)
                     returns int {
   if n == 0 {
      panic error("division by 0");
   }
   return m/n;
}
```

- Ballerina distinguished normal errors from abnormal errors
- Normal errors are handled by returning error values
- Abnormal errors are handled using `panic` statement
- Abnormal errors should typically result in immediate program termination
  - programming bug
  - out of memory
- A panic has an associated error value

any type

```
any x = 1;

// can cast any to specific type
int n = <int>x;

// can convert to string
string s = x.toString();

// can test its type with
// is operator
float f = x is int|float
          ? <float>x
          : 0.0;
```

- `any` means any value except an error
- Equivalent to a union of all non-error basic types
- Use `any|error` for absolutely any value
- Langlib `lang.value` module contains functions that apply to multiple basic types

Ignoring return values and errors

```
// allowed only if return value is ()
doX();

// allowed if return value does not
// include error
_ = getX();

// use checkpanic if you don't want
// to handle an error
checkpanic tryX();
```

- Ballerina does not allow silently ignoring return values
- To ignore a return value, assign it to _; this is like an implicitly declared variable that cannot be referenced
- When a return type includes an error, you have to do something with the error
- _ is of type any: you cannot use _ to ignore an error
- checkpanic is like check, but panics on error rather than returning

Covariance

```
int[] iv = [1, 2, 3];

any[] av = iv; // OK

function foo() {
   // runtime error; otherwise
   // iv[0] would have wrong type
   av[0] = "str";
}
```

- Arrays and maps are covariant
- Allowed to e.g. assign int[] to any[]
  - set of values allowed by int is subset of set of values allowed by any
  - set of values allowed by int[] is subset of set of values allowed by any[]
- Static type-checking guarantees that result of a read from a mutable structure will be consistent with static type
- Covariance means that a write to a mutable structure may result in a runtime error
- Arrays, maps and records have "inherent" type that constrains mutation

Object

```
function demoMyClass() {
   m:MyClass x = new m:MyClass(1234);
   x.foo();
   int n = x.n;
};
```

- Separate basic type
- An `object` value has named methods and fields
- Methods and fields are in the same symbol space
- A class both defines an object type and provides a way to construct an object
- Apply `new` operator to a class to get an object
- Call method using obj.foo(args)
- Access field using obj.x

Defining classes

```
public class Counter {
   private int n;

   public function init(int n = 0) {
      self.n = n;
   }

   public function get() returns int {
      return self.n;
   }

   public function inc() {
      self.n += 1;
   }
}
```

- Module can contain class definitions
- `init` method initializes the object
- Arguments to new are passed as arguments to init
- methods use `self` to access their object
- `private` means accessible only by code within the class definition

`init` return type

```
class File {
   string path;
   string contents;
   function init(string p)
                     returns error? {
      self.path = p;
      self.contents =
         check io:fileReadString(p);
   }
};

File f = check new File("test.txt");
```

- init function has a return type, which must be subtype of error?
- If init returns (), then new returns the newly constructed object
- If init returns an error, then new returns that error
- If init does not specify a return type, then return type defaults to () as usual, meaning that `new` will never return error

Identity

```
MyClass obj1 = new MyClass;
MyClass obj2 = new MyClass;

// true
boolean b1 = (obj1 === obj1);
// false
boolean b2 = (obj1 === obj2);
// true
boolean b3 = ([1,2,3] == [1,2,3]);
// false
boolean b4 = ([1,2,3] === [1,2,3]);
// true
boolean b5 = (-0.0 == +0.0);
// false
boolean b6 = (-0.0 === +0.0);
```

- `===` and `!==` operators test for identity
- Identical for mutable basic types means stored at the same address
- `==` and `!=` are not defined for objects
- -0.0 and +0.0 are equal but not identical

const and final

```
const MAX_VALUE = 1000;
const URL = "https://ballerina.io";

final string msg = loadMessage();
```

- const means immutable and known at compile-time
- Type is singleton: set containing single value
- Variable or class field can be declared as final, meaning cannot be assigned to after it has been initialized

Enumerations

```
enum Color {
   RED, GREEN, BLUE
}

// shorthand for
const RED = "RED";
const GREEN = "GREEN";
const BLUE = "BLUE":
type Color RED|GREEN|BLUE;

enum Color {
   RED = "red",
   GREEN = "green",
   BLUE = "blue"
};
```

- Enumerations are shorthand for unions of string constants
- A const can be used as a singleton type
- Not a distinct type
- Can specify string constants explicitly

`match` statement

```
const KEY = "xyzzy";

function mtest(any v) returns string {
   match v {
      17 => { return "number"; }
      true => { return "boolean"; }
      "str" => { return "string"; }
      KEY => { return "constant"; }
      0|1 => { return "or"; }
      _ => { return "any"; }
   }
}
```

- Like `switch` statement in C, JavaScript
- Matches value not type
- == is used to test whether left hand side matches value being matched
- Left hand side can be
  - simple literal (nil, boolean, int, float, string)
  - identifier referring to a constant
- Left hand side of _ matches if value is of type any
- Use | to match more than one value

Type inference

```
var x = "str";

function printLines(string[] sv) {
  foreach var s in sv {
     io:println(s);
  }
}

// Infer x as type MyClass
var x = new MyClass;

// Infer class for new as MyClass
MyClass x = new;
```

- Type inference is local: restricted to single expression
- Goal is: Do Not Repeat Yourself
- `var` says that type of variable from type of expression used to initialize it
  - Convenient with `foreach` statement
- Also infer type of value to be created from type of variable
- Overuse can make code harder to understand

Functional programming

```
var isOdd = function(int n) returns boolean {
   return n % 2 != 0;
}

type IntFilter function(int n) returns boolean;

function isEven(int n) returns boolean {
  return n % 2 == 0;
}

IntFilter f = isEven;

int[] nums = [1, 2, 3];

int[] evenNums = nums.filter(f);

int[] oddNums = nums.filter(n => n % 2 != 0);
```

- First-class functions: functions are values
- Function values are closures
- Separate basic type
- Anonymous function and type syntax look like function definition without the name
- Arrays provide the usual functional methods: filter, map, forEach, reduce
- Like `foreach`, also work on maps and strings
- Shorthand syntax for when type is inferred and body is an expression

Asynchronous function calls

```
// assume foo() returns int
future<int> fut = start foo();


int x = check wait fut;
```

- `start` calls a function asynchronously
- Runs on separate logical thread ("strand"): cooperatively multitasked by default
- Result will be of type future<T>
- `future` is a separate basic type
- wait for future<T> gives T|error (waiting for the same future more than once gives an error)
- Use f.cancel() to terminate a future

Annotations

```
// The @display annotation applies
// to the transform function
@display {
   iconPath: "transform.png"
}
public function transform(string s)
   returns string {
   //...
}

// annotation on start
future<int> f = @strand {
                    thread: "any"
                }
                start foo();
```

- Annotations start with `@tag`
- Annotations come before what they apply to
- Unprefixed tags refer to standard platform-defined annotations
- Prefixed tags refer to annotations declared in modules
- `@tag` can be followed by record constructor expression

Documentation

```
# Adds two integers.
# + x - an integer
# + y - another integer
# + return - the sum of `x` and `y`
public function add(int x, int y)
                     returns int {
  return x + y;
}
```

- Annotations would be inconvenient for specifying structured document
- Lines starting with # contain structured documentation in Markdown format
- Ballerina-flavored Markdown (BFM): additional conventions on top of Markdown, which make it more convenient for documenting Ballerina code

Part 2
What makes Ballerina distinctive

What makes Ballerina distinctive

- Part 1 describes features that are common to most languages
- Ballerina would be pointless if it provided only what was described in Part 1
- Part 2 describes the features that make Ballerina distinctive
- It's the combination that is distinctive: most of the features are not individually novel

Ballerina target

- Applications programming not systems programming
- Small to medium sized programs
- Integration: some similarities with scripting languages
- Pragmatic: success is satisfied users not published research papers
- Industry: reliability and maintainability matter
- Moderate cognitive load: more TypeScript than Haskell

Cloud has changed programming

	Pre-cloud	Cloud
Data access	Read/write files	Consume/provide network services
APIs	Function calls to libraries in e.g. native code, JVM, .NET	Network messages over e.g. HTTP/JSON
UI	OS libraries	JavaScript client
Concurrency	Most application programs do not need to deal with concurrency	Pervasive
Programming languages	C, C++, Java, C#	JavaScript, Ballerina, Go, Rust

Themes

- Network interaction
- Data
- Concurrency

Network interaction

- Consuming services
- Providing services

Consuming services: client objects

```
import ballerina/email;

function mailDemo() returns error? {
  email:SmtpClient sc
    = check new("smtp.example.com",
                "user123@example.com",
                "passwd123");
  check sc->sendEmailMessage({
     to: "jjc@jclark.com",
     subject: "Ballerina"
     body: "I love Ballerina!"
  });
}
```

- Client objects provide `remote` methods, which are used to interact with a remote service
- A client object is created by applying new to a client class
  - Defined by `client class {...}`
- Applications typically do not need to write client classes, which are either
  - provided by library modules
  - generated from some flavour of IDL
- Remote method calls use `->` syntax
  - support sequence diagram view
  - not allowed nested within expressions
  - separate symbol space for method names
  - remote methods implicitly public

Providing services

- Service object
  - remote methods defined by application; no need to define a class
  - attached to a Listener object
- Listener
  - receives network input
  - makes calls to remote methods of attached service objects
  - registered with module
- Module
  - initialized on program startup
  - starts up registered Listeners after initialization
  - shuts down registered Listener during program shutdown

Listener declaration

```
import ballerina/http;

listener http:Listener h = new(8080);
```

- Allowed at module level
- Like a variable declaration, but registers the newly created Listener object with the module
- If `new` returns an error, then module initialization fails

Module lifecycle

- All modules are initialized at program startup
- A module's listeners are registered during module initialization
- Module initialization is ordered so that if module A imports module B, then module A is initialized after module B
- Initialization phase ends by calling `main` function
- If there are registered listeners, then initialization phase is followed by listening phase
- Listening phase starts by calling `start` method on each registered listener
- Listening phase is terminated by signal (e.g. SIGINT, SIGTERM)
- Calls either `gracefulStop` or `immediateStop` on each registered listener

Module `init` function

```
import myService as _;

function init() {
   io:println("Hello world");
}
```

- A module can have an `init` function just like an object
- Initialization of a module ends by calling its `init` function if there is one
- Return type must be a subtype of `error?`
- Usually it's an error to import a module without using it
- If you want to import a module because of what its initialization does (e.g. registering services), then use `as _` in the import

Constructing objects without classes

```
var obj = object {
   function greet() returns string {
      return "Hello world";
   }
};

string greeting = obj.greet();
```

- An object can be constructed directly, without defining a class

Service declaration

```
import ballerina/udp;

listener udp:Listener ul = new(8080);

service on ul {
   remote function onDatagram(udp:Datagram dg)
   {
      io.println("bytes received: ",
                 dg.bytes.length());
   }
};
```

- Creates service object using object constructor
- Attaches service object to the listener
- Type of Listener determines required type of remote methods
- Annotations are used extensively e.g. for security

Service declaration desugaring

```
service on ul {
   remote function onDatagram(udp:Datagram dg) { ... }
}

// desugars to

var obj = service object {
   remote function onDatagram(udp:Datagram dg) { ... }
};

function init() {
   ul.attach(obj);
}
```

Combine listener declaration into service declaration

```
service on new udp:Listener(8080) {
   remote function onDatagram(udp:Datagram dg) {
      io.println("bytes received: ",
                 dg.bytes.length());
   }
};

// short for
listener ul = new udp:Listener(8080);

service on ul {
   remote function onDatagram(udp:Datagram dg) {
      io.println("bytes received: ",
                 dg.bytes.length());
   }
};
```

Representing responses

- Many protocols use request-response pattern
- When call to client remote method makes request, return value of call provides response
- When invocation of service remote method handles request, return value of method provides response
- But this has limitations:
    - application has no control when there is an error in sending a response
    - only supports exactly one response
- More flexible approach is for service remote method to have a parameter that is a client object representing the caller: service remote method provides responses by making remote calls on this client object

Resource concept

- Service objects use remote methods to expose services in procedural style: remote methods are named by verbs
- Service objects use resources to expose services in an RESTful style: resources are named by nouns
- Resources are motivated by HTTP, but are general enough also to work for GraphQL

Resources

- Resource method associated with combination of accessor and resource name
- Accessors determined by network protocol
- Network-oriented generalization of OO getter/setter concept
- Service declaration specifies base path for resource names
- In HTTP, function parameters come from query parameters

```
service / on new http:Listener(8080) {
   resource function get hello(string name) returns string {
      return "Hello, " + name;
   }
}
```

Hierarchical resources

- Resource name is relative path, which can have multiple path segments
- Base path is absolute path
- Single listener can have multiple services each with different base paths

```
service /demo on new http:Listener(8080) {
   resource function get greeting/hello(string name) returns string {
      return "Hello, " + name;
   }
}
```

Resource path parameters

- Path segments can be parameters

```
// GET /demo/greeting/James would return "Hello, James"

service /demo on new http:Listener(8080) {
   resource function get greeting/[string name]() returns string {
      return "Hello, " + name;
   }
}
```

Hierarchical services

- Resource methods can return service objects
- Semantics is that path of resource method becomes base path of service object: similar to filesystem mount
- Root service is special case of this
- Basis for GraphQL support: each GraphQL object is represented by a service object

Plain data

Plain data

- Ballerina has concept of "plain data": data that is independent of any specific code operating on the data
- Network interfaces between programs are based on plain data
- Opposite of objects, which combine data and code
- Plain data supports deep copy and deep equality
- Plain data supports serialization/deserialization without coupling
- Key goal of Ballerina is to facilitate programs that work on plain data

Ballerina basic types

Simple types

Always plain data

- nil
- boolean
- int
- float
- *decimal*

Sequence

Always plain data

- string
- *xml*

Structural

Plain data if members are

- array/*tuple*
- map/record
- *table*

Behavioural

Not plain data

- function
- object
- error
- *stream*
- *typedesc*
- *handle*

`decimal` type

```
function floatSurprise() {
  float f = 100.10 - 0.01;
  // will print 100.08999999999999
  io:println(f);
}

decimal nanos = 1d/1000000000d;
```

- Third numeric type
  - works like int and float
  - no implicit conversion
- Represents decimal fractions exactly
- Avoids surprises that you get with float
- Preserves precision: 2.1kg and 2.10kg don't mean the same to humans
- Separate basic type; counts as anydata
- Literal uses `d` suffix (`f` suffix is for float)
- Floating point, not infinite precision
  - 34 decimal digits
  - 22 digits enough for US national debt in ¢
  - 27 digits enough for age of universe in ns
- No infinity, NaN or negative zero

Plain data basic types to come

- table
  - works uniformly with array and map
  - table contains records
  - support access by key using concept similar to primary keys in relational databases
  - fields containing key are immutable
- xml
  - sequence of xml items (element, text, processing instruction, comment)
  - sequence concept similar to string and to XQuery
  - XML attributes represented as `map<string>`
  - xml literals support XML syntax

Immutability

- `anydata` values can be made immutable
- Simple and string values are inherently immutable
- A structural value can be constructed as mutable or immutable
  - Value includes an immutable flag
  - Immutable flag is fixed at the time of construction
  - Attempting to mutate an immutable structure causes a panic at runtime
- Immutability is deep: an immutable structure can only have immutable members
  - an immutable value is safe for concurrent access without locking

`anydata` type

```
anydata x1 = [1, "string", true];
anydata x2 = x1.clone();

// true
boolean eq = (x1 == x2);

const RED = {R: 0xFF, G: 0, B: 0};
```

- Type for plain data is `anydata`
- Subtype of `any`
- == and != operators test for deep equality
- `x.clone()` returns deep copy, with same mutability
- `x.cloneReadOnly()` returns deep copy that is immutable
  - Ballerina syntax uses `ReadOnly` to mean immutable
- Both `x.clone/cloneReadOnly()` do not copy immutable parts of x
- `const` structures are allowed
- Equality and cloning handle cycles

Configurable variables

```
# Port on which to run the service
configurable int port = 8080;

# Password must be supplied in
# configuration file
configurable string password = ?;
```

- A module-level variable can be declared as configurable
- The initializer of a configurable variable can be overridden at runtime (e.g. by a TOML file)
- A variable where configuration is required can use an initializer of `?`
- Type of variable must be subtype of `anydata`

User-defined types describe both data in memory and data on the wire

Optional fields

```
type Headers record {
   string from;
   string to;
   string subject?;
};

Header h = {
  from: "John",
  to: "Jill"
};

string? subject = h?.subject;
```

- Records can have optional fields
- Use `?.` operator to access optional field

Open records

```
type Person record {
   string name;
};
type Employee record {
   string name;
   int id;
};
Employee e = {
   name: "James", id: 10
};
Person p = e;
Person p2 = {
   name: "John", "country": "UK"
};
map<anydata> m = p2;
```

- Record types are by default open: they allow fields other than those specified
- Type of unspecified fields is `anydata`
- Records are maps
- Open records belongs to `map<anydata>`
- Use quoted keys for fields not mentioned in the record type

Controlling openness

```
type Coord record {|
   float x;
   float y;
|};
Coord x = { x: 1.0, y: 2.0 };
map<float> m = x;

type Headers record {|
   string from;
   string to;
   string...;
|};
Headers h = {
   from: "Jane", to: "John"
};
map<string> m = h; // OK
```

- Use `record {| ... |}` to describe a record type that allows exclusively what is specified in the body
- Use *`T...`* to allow other fields of type T
- `map<`*`T`*`>` same as `record {| `*`T`*`...; |}`

`json` type

```
import ballerina/lang.value;

json j = { "x": 1, "y": 2 };

string s = j.toJsonString();

json j2 =
  check value:fromJsonString(s);

// allow null for JSON compatibility
json j3 = null;
```

- `json` type is a union: `()|boolean|int|float|decimal|string|json[]|map<json>`
- A `json` value can be converted to and from JSON format straightforwardly
  - except for choice of Ballerina numeric type
- Ballerina syntax is compatible with JSON
  - allow `null` for `()` for JSON compatibility
- `json` is `anydata` without `table` and `xml`
- `toJson` recursively converts `anydata` to `json`
  - table values are converted to arrays
  - xml values are converted to strings
- `json` and `xml` types are not parallel

Working with JSON: two approaches

- Approach 1: Work with json values directly
- Approach 2: Work with application-specific, user-defined subtype of `anydata`
  - Convert from JSON to application-specific type
  - Process using application-specific subtype
  - Convert back to JSON from application-specific type
- Ballerina supports both approaches
- Ballerina's strength is making Approach 2 really easy

Working with json directly

```
json j = {
   x: {
      y: {
         z: "value"
      }
   }
};

json v = check j.x.y.z;
string s = check v;
// short for
string s =
   check value:ensureType(v, string);

// put it together
string s = check j.x.y.z;
```

- json values use "lax" typing
- Expressions that would usually be a compile-time error instead result in an error at runtime
- User experience similar to dynamic language
- Two cases
    - accessing a field with j.x or j?.x
    - implicit conversion from json value to unstructured type
- ensureType performs numeric conversions

match statement with maps

```
function foo(json j) returns error? {
   match j {
      { command: "add", amount: var x }
      => {
         decimal n = check x;
         add(n);
      }
      _ => {
         return error("invalid command");
      }
   }
}
```

- match statement can be used to match maps
- Patterns on LHS in a match statement can have variable parts that can be captured
- Useful for working directly with json
- Match semantics are open (may have fields other than those specified in the pattern)

Converting from user-defined type to JSON

```
// closed type
type Coord record {|
  float x;
  float y;
|};
Coord coord = { x: 1.0, y: 2.0 };
// nothing to do
json j = coord;
// If coord is is open:
type Coord record {
  float x;
  float y;
};
// usually happens automatically
json j = coord.toJson();
```

- Conversion from json value to JSON format is straightforward
- Problem here is converting from application-specific, user-defined subtype of anydata into json
- In many cases, this is a no-op: user-defined type will be subtype of json as well as of anydata
- With tables, xml or records open to anydata, use toJson to convert anydata to json
- APIs that generate JSON typically accept anydata and automatically apply toJson

Converting from JSON to user-defined type

```
// closed type
type Coord record {
  float x;
  float y;
};
json j = { x: 1.0, y: 2.0 };

// Runtime error!
Coord c = <Coord>j;

// This will work
Coord c = <Coord>j.cloneReadOnly();
```

- This way round is more interesting!
- With mutable values, would not be type-safe to allow a cast
- Mutable structures have inherent type that limits mutation
  - does not affect equality
  - clone copies the type
- Cast to T will work on mutable structure s only if inherent type of s is subtype of T
- Casting of immutable value will work, but does not do numeric conversions

Converting to user-defined type: `cloneWithType`

```
type Coord record {
  float x;
  float y;
};
json j = { x: 1.0, y: 2.0 };

Coord c
  = check j.cloneWithType(Coord);

// Argument defaulted from context
Coord c = check j.cloneWithType();
```

- Langlib function in lang.value
- Result recursively uses specified type as inherent type of new value
- Argument is a `typedesc` value
- Static return type depends on argument
- Argument defaulted from context
- Automatically performs numeric conversions as necessary
- Every part of value is cloned, including immutable structural values
- Graph structure is not preserved
- Variant `fromJsonWithType` also does reverse of conversions done by toJson

Resource method typing

```
import ballerina/http;

type Args record {|
  decimal x;
  decimal y;
|};

listener h = new http:Listener(9090);

service /calc on h {
  resource function post add(
          @http:Payload Args args)
       returns decimal {
     return args.x + args.y;
  }
}
```

- Resource method arguments can use user-defined types
- Listener will use introspection to map from protocol format (typically JSON) to user-defined type, using `cloneWithType`
- Return value that is subtype of anydata will be mapped from user-defined type to protocol format, typically JSON, using toJson
- Can generate API description (e.g. OpenAPI) from Ballerina service declaration
- Annotations can be used to refine the mapping between Ballerina-declared type and wire format

JSON numbers

- Problem: Ballerina has three numeric types; but JSON has one
- json type allows int|float|decimal
- toJsonString will convert int|float|decimal into JSON numeric syntax
- fromJsonString converts JSON numeric syntax into int, if possible, and otherwise decimal
- cloneWithType or ensureType will convert from int or decimal into user's chosen numeric type
- Net result is that you can use json to exchange full range of all three Ballerina numeric types
- -0 is an edge case: represented as float

Query expressions

SQL-like syntax for list comprehensions

```
int[] nums = [1, 2, 3, 4];

// Result is [10, 20, 30, 40]
int[] numsTimes10 =
   from var i in nums
   select i*10;

// Result is [2, 4]
int[] evenNums =
   from var i in nums
   where i % 2 == 0
   select i;
```

- Query-like expressions start with `from` and end with `select`
- List comprehension, based on mathematical "set builder" notation

$$\{\ 10 \times i \mid i \in nums\ \}$$

$$\{\ i \mid \mathrm{i} \bmod 2 = 0, i \in nums\ \}$$

Destructuring records

```
type Person record {
   string first;
   string last;
   int yearOfBirth;
};
Person[] persons = [];

// Projection with first and last fields
var names =
  from var {first: f, last: l} in persons
  select {first: f, last: l};

// more simply
var names =
  from var {first, last} in persons
  select {first, last};
```

- Particularly useful with query expressions, but works anywhere you can have `var`
- Thing following var is called a binding pattern
- Semantics of binding pattern is open
- `{x}` is short for `{x: x}` in both binding patterns and record constructors

Let clause

```
string[] names =
   from var {first, last} in persons
   let int len1 = first.length()
   where len1 > 0
   let int len2 = last.length()
   where len2 > 0
   let string name = first + " " + last
   select name;
```

- Query expressions can have `let` clauses
- Can be anywhere between `from` and `select`
- Multiple `where` clauses are allowed
- Semantics similar to XQuery FLWOR

Ordering

```
type Employee record {
   string firstName;
   string lastName;
   decimal salary;
};

Employee[] employees = [
  // ...
];

Employee[] sorted =
   from var e in employees
   order by e.lastName ascending,
            e.firstName ascending
   select e;
```

- Ordering works consistently with <, <=, >, >= operators
- Concept of some comparisons involving `()` and float NaN being unordered
- `order by` clause allows expressions not just field access
- A library module can enable Unicode-aware sorting by providing a unicode:sortKey(str, locale) function

Limit clause

```
Employee[] top100 =
   from var e in employees
   order by e.salary descending
   limit 100
   select e;
```

- `limit` clause limits number of results from earlier clauses

Tables

Table concept

- "It is better to have 100 functions operate on one data structure than to have 10 functions operate on 10 data structures" - Alan Perlis
- Ballerina encourages use of its built-in data structures: array, map and table
- A table is a collection of records; each record represents a row of the table
- A table is plain data if and only if its rows are plain data
- Table maintains invariant that each row is uniquely identified by a key
- Each row's key is stored in fields, which must be immutable
- Compared to maps:
  - key is part of the value, rather than separate
  - type of key is not restricted to string
  - order of members is preserved

Table syntax

```
type Employee record {
   readonly string name;
   int salary;
};
table<Employee> key(name) t = table [
  { name: "John", salary: 100 },
  { name: "Jane", salary: 200 }
];

Employee? e = t["Fred"];

function increaseSalary(int n) {
  foreach Employee e in t {
     e.salary += n;
  }
}
```

- A record field can be declared as `readonly`: cannot assign to the field after the record is created
- table type gives type of row and name of key field
- table constructor expression looks like an array constructor
- `foreach` statement will iterate over a table's rows in order
- Use t[k] to access a row using its key

Multiple key fields

```
type Employee record {
   readonly string firstName;
   readonly string lastName;
   int salary;
};

table<Employee> key(firstName, lastName) t = table [
  { firstName: "John", lastName: "Smith", salary: 100 },
  { firstName: "Fred", lastName: "Bloggs", salary: 200 }
];

Employee? e = t["Fred", "Bloggs"];
```

Structured keys

```
type Employee record {
   readonly record {
      string first;
      string last;
   } name;
   int salary;
};

table<Employee> key(name) t = table [
   { name: {first: "John", last: "Smith"}, salary: 100 },
   { name: {first: "Fred", last: "Bloggs"}, salary: 200 }
];

Employee? e = t[{first: "Fred", last: "Bloggs"}];
```

- Key fields can be structured: any subtype of plain data
- Value of key field must be immutable
- Initializer of readonly field will be constructed as immutable
- In other cases, can use `cloneReadOnly` to create an immutable value

Querying tables

```
type Employee record {|
   readonly int id;
   string firstName;
   string lastName;
   decimal salary;
|};

table<Employee> key(id) employees =
  table [...];

int[] salaries =
  from var { salary } in employees
  select salary;
```

- Tables combine nicely with query
- Maps not so much
- Basic type of output of query expression determined by
  - contextually expected type
  - input type

Creating tables with query

```
var highPaidEmployees =
   table key(id)
   from var e in employees
   where e.salary >= 1000
   select e;
```

- Query expressions can create tables
- Key of created table can be specified explicitly

Join clause

```
type User record {|
   readonly int id;
   string name;
|};
type Login record {|
   int userId;
   string time;
|};
table<User> key(id) users = [...];
Login[] logins = [...];

string[] loginLog =
  from var login in logins
  join var user in users
       on login.userId equals user.id
  select user.name + ":" + login.time;
```

- Query can take advantage of table keys by using a join clause
- Does inner equijoin
- Results similar to nested `from` clause and `where` clause
- Implemented as hash join: table keys allow you to avoid building a hash table
- Type to join on must be anydata

Streams

Stream type

- A stream represents a sequence of values that are generated as needed
- The end of a stream is indicated with a termination value, which is error or nil
- Type `stream<T,E>` is a stream where
  - members of the sequence are type T
  - termination value is type E
- `stream<T>` means `stream<T,()>`
- Separate basic type, but like an object

Querying with streams

```
type LS stream<string,io:Error?>;
// strip blank lines
function strip(LS lines) returns LS {
   stream from var line in lines
   where line.trim().length() > 0
   select line;
}

function count(LS lines)
              returns int|io:Error {
   int nLines = 0;
   check from var line in lines
   do {
      nLines += 1;
   }
   return nLines;
}
```

- If stream terminates with error, result of query expression is an error
- Cannot use `foreach` on stream type with termination type that allows error
- Instead use `from` with `do` clause; result is subtype of `error?`
- Use `stream` in front of `from` to create a stream
  - lazily evaluated
  - failure of check within the query will cause the stream to produce an error termination value

Templates

Backtick templates

```
string name = "James"
// Result is "Hello, James"
string s = string`Hello, ${name}`;
string s = string`Backtick:${"`"}`;
```

- Consists of tag followed by characters surrounded by backticks
  - Can contain expressions in `${...}` to be interpolated
  - No escapes recognized: use expression to escape
  - Can contain newlines
- Processed in two phases
  - Phase 1 does tag-independent parse: result is list of strings and expressions
  - Phase 2 is tag-dependent
- Phase 2 for ``string`...` `` converts expressions to strings and concatenates
- `base16` and `base64` tags do not allow expressions

Raw templates

- A raw template is a backtick template without a tag
- Exposes result of phase 1 without further processing
- Raw template is evaluated by evaluating each expression and creating an object containing
    - an array of the strings separated by insertions
    - an array of the results of expression evaluation and an array of strings separating
- Important use case: SQL parameters

```
function getOrders(int customerId) returns stream<Order,sql:Error?> {
   return db->query(`SELECT * FROM order
                     WHERE customer_id = ${customerId}`);
}
```

XML

XML overview

- Separate basic type xml
- Uses sequence concept similar to XQuery and XPath2
- Based on XML Infoset, rather than PSVI
- Allows XML syntax to be used to construct xml values
- xml type is designed to work well for HTML as well as XML
- Navigation syntax with XPath-like functionality
- Works with query expressions to provide XQuery FLWOR-like functionality
- No up pointers: elements do not have a reference to parents or siblings

Sequences

- Ballerina has two basic types that are sequences: string, xml
- A value is a sequence of basic type T if it is
  - an empty sequence of basic type T,
  - a singleton of basic type T, or
  - a concatenation of two sequences of basic type T
- Sequences differ from arrays:
  - sequences are flat: no nesting
  - there is no difference between a singleton *x* and a sequence consisting of just *x*
  - basic type of sequence determines basic type of members
- Membership of a sequence is immutable e.g. cannot mutate a sequence of one item into a sequence of two items
- A sequence has no identity: two sequences are === if their members are ===

XML data model

- xml value is a sequence representing the parsed content of an XML element
- xml value has four kinds of item
  - element, processing instruction and comment item correspond 1:1 to XML infoset items
  - text item corresponds to one or more Character Information Items
- XML document is an xml sequence with only one element and no text
- An element item is mutable and consists of:
  - name: type string
  - attributes: type map<string>
  - children: type xml
- A text item is immutable
  - it has no identity: == is the same as ===
  - consecutive text items never occur in an xml value: they are always merged

xml templates

```
string url = "https://ballerina.io";

xml content = xml`
<a href="${url}">Ballerina</a> is
an <em>exciting</em> new language!`;

xml p = xml`<p>${content}</p>`;
```

- xml values can be constructed using an XML template expression
- Phase 2 processing for `xml` template tag parses strings using the XML 1.0 Recommendation's grammar for content (what XML allows between a start-tag and and end-tag)
- Interpolated expressions can be
  - in content, xml or string values
  - in attribute values, string values

xml operations

```
xml x1 = xml`<para id="greeting">Hello</p>`
string id = check x1.id;
```

- + does concatenation
- == does deep equals
- foreach iterates over each item
- x[i] gives i-th item (empty sequence if none)
- x.id accesses required attribute named id: result is error if there is no such attribute or if x is not a singleton
- x?.id accesses optional attribute named id: result is () if there is no such attribute
- Langlib lang.xml provides other operations
- Mutate an element using e.setChildren(x)

xml subtyping

```
xml:Element p = xml`<p>Hello</p>`;

function stringToXml(string s)
                returns xml:Text {
   return xml:createText(s);
}

function rename(xml x, string oldName,
                string newName) {
   foreach xml:Element e in x.elements() {
      if e.getName() == oldName {
         e.setName(newName);
      }
      rename(e.getChildren());
   }
}
```

- An xml value belongs to `xml:Element` if it is consists of just an element item
- Similarly for `xml:Comment` and `xml:ProcessingInstruction`
- An xml value belongs to `xml:Text` if it consists of a text item or is empty
- An xml value belongs to the type `xml<T>` if each of its members belong to T
- Functions in lang.xml use this to provide safe and convenient typing e.g.
  - `x.elements()` returns element items in x as type `xml<xml:Element>`
  - `e.getName()` and `e.setName()` are defined when `e` has type `xml:Element`

XML navigation syntactic sugar

x.<para>	every element in x named para
x/*	for every element e in x, the children of e
x/<para>	for every element e in x, every element named para in the children of e
x/<th|td>	for every element e in x, every element named th or td in the children of e
x/<*>	for every element e in x, every element in the children of e
x/*.text()	for every element e in x, every text item in the children of e
x/**/<para>	for every element e in x, every element named para in the descendants of e
x/<para>[0]	for every element e in x, first element named para in the children of e

Querying with XML

- Can use query expressions to manipulate XML

```
function paraByLang(xml x, string lang) returns xml {
   return from var para in x.<para>
          where para?.lang == lang
          select para;
}
```

Combining XML templates and query

- XML templates combine nicely with query e.g. you can have a templates containing a query expression containing a template

```
type Person record {|
   string name;
   string country;
|};
function personsToXml(Person[] persons) returns xml {
   return xml`<data>${
      from var {name, country} in Persons
      select xml`<person country="${country}">${name}</person>`
   }</data>`;
}
```

XML namespaces

```
xml:Element e =
 xml`<p:e xmlns:p="http://example.com/"/>`;

// name will be "{http://example.com}e"
string name = e.getName();
```

- Goal is to support for namespaces, but no added complexity if you don’t use them
- Qualified name `ns:x` in XML is expanded into `{url}x` where `url` is namespace name bound to `ns`
- XML namespace declarations are kept as attributes using standard binding of `xmlns` to `http://www.w3.org/2000/xmlns/`

`xmlns` declarations

```
xmlns "http://example.com" as eg;

xml x = xml`<eg:doc>Hello</eg:doc>`;

xmlns "http://example.com" as ex;

// will be true
boolean b = (x === x.<ex:doc>);

// exdoc will be "{http://example.com}doc"
string exdoc = ex:doc;
```

- `xmlns` declarations are like import declarations, but bind the prefix to a namespace URL rather than a module
- `xmlns` declarations in the Ballerina module provide namespace context for parsing xml templates
- Qualified names in Ballerina modules are expanded into strings using the `xmlns` declarations in the module
- xmlns declarations also allowed at block level

Sequence-diagram based concurrency

Named workers

```
function main() {
   io:println("Initializing");
   worker A {
      io:println("In worker A");
   }
   worker B {
      io:println("In worker B");
   }
   io:println("In function worker");

}
```

- Normally all of a function's code belongs to the function's default worker, which has a single logical thread of control
- A function can also declare named workers, which run concurrently the function's default worker and other named workers
- Code before any named workers is executed before named workers starts
- Variables declared before all named workers and function parameters are accessible in named workers

Sequence diagrams

- A function can be viewed as a sequence diagram
- Lifeline (vertical line) for each worker (both named worker and function's default worker)
- Lifeline for each client object parameter or variable in initialization section, representing remote system to which the client object is sending messages
- Each remote method call on a client object is represented as a horizontal line between the lifeline of the worker making the call and the remote system

Waiting for workers

```
function main() {
   io:println("Initializing");
   worker A {
      io:println("In worker A");
   }
   io:println("In function worker");
   wait A;
   io:println("After wait A");
}
```

- Named workers can continue to execute after the function's default worker terminates and the function returns
- A worker (function or named) can use `wait` to wait for a named worker

Strands

- By default, named workers are multitasked cooperatively, not preemptively
- Each named worker has a "strand" (logical thread of control) and execution switches been strands only at specific "yield" points such as
  - doing a wait
  - when a library function invokes a system call that would block
- This avoids the need for users to lock variables that are accessed from multiple named workers
- An annotation can be used to make a strand run on a separate thread

Named worker return values

```
function demo(string s)
                   returns int|error {
   worker A returns int|error {
      int x = check int:fromString(s);
      return x + 1;
   }
   int y = check wait A;
   return y + 1;
}
```

- Named workers have a return type, which defaults to nil
- A return statement in a named worker terminates the worker not the function
- Using check in a named worker will thus
- Waiting on a named worker will give its return value

Alternate wait

```
function fetch(string url)
    returns string|error {...}

// Fetch from A or B
function altFetch(string urlA,
                  string urlB)
                returns string|error {
   worker A returns string|error {
     return fetch(urlA);
   }
   worker B returns string|error {
     return fetch(urlB);
   }
   return wait A|B;
}
```

- Can wait for one of several workers

Multiple wait

```
type Result record {
   string|error a; string|error b;
};

function multiFetch(string urlA,
                    string urlB)
                 returns Result {
  worker WA returns string|error {
    return fetch(urlA);
  }
  worker WB returns string|error {
    return fetch(urlB);
  }
  return wait { a: WA, b: WB };
}
```

- Can wait for multiple named workers
- `wait { X, Y }` means `wait { X: X, Y: Y }` so you can say

```
var r = wait { X, Y };
```

- Works with futures also

Named workers and futures

```
function demo() returns future<int> {
   worker A returns int {
      return 42;
   }
   return A;
}

type FuncInt function() returns int;

function startInt(FuncInt f)
                returns future<int> {
   worker F returns int {
      f();
   }
   return F;
}
```

- Futures and workers are the same thing
- A reference to a named worker can be implicitly converted into a future
- `start` is sugar for calling a function with a named worker and returning the named worker as a future
- Cancellation of futures only happens at yield points

Inter-worker message passing

```
function demo() returns int {
   worker A {
      1 -> B;
      2 -> C;
   }
   worker B {
      int x1 = <- A;
      x1 -> function;
   }
   worker C {
      int x2 = <- B
      x2 -> function;
   }
   int y1 = <- B;
   int y2 = <- C;
   return y1 + y2;
}
```

- Use `-> W` or `<- W` to send a message to or receive a message from worker W (use `function` to refer to the function's default worker)
- Messages are copied using clone(); implies immutable values are passed without copy
- Message sends and receives are paired up at compile-time
- Each pair turns into horizontal line in sequence diagram
- Easy to use and safe, but limited expressiveness

Inter-worker failure propagation

```
function demo() returns int|error {
   worker A returns error? {
      check foo();
      42 -> function;
   }
   int x = check <- A;
   return x;
}
```

- Workers may need to call functions that can return an error
- Pairing up of sends and receives guarantees that each send will be received, and vice-versa, *provided* neither sending nor receiving worker has failed
- Send to or receive from failed worker will propagate the failure

Transactions

Language support for transactions

- Language support for interacting with a transaction manager
- Not transactional memory
- Ballerina runtime includes transaction manager
- Syntax for delimiting transactions
- Current transaction part of execution context of a strand
- Composes with network interaction features to support distributed transactions

`transaction` statement

```
function demo() returns error? {
   transaction {
      doStage1();
      doStage2();
      check commit;
   }
}
```

- Compile-time guarantee that transactions are bracketed with begin and commit/rollback
- `transaction` statement begins a new transaction and executes a block
- Commit of a transaction must be done explicitly using `commit`
  - must be lexically within a transaction statement
  - `commit` may return an error; usual rules on not ignoring errors apply

`check` semantics

```
function demo() returns error? {
   do {
      check foo();
      check bar();
      if !isOK() {
         fail error("not OK");
      }
   }
   on fail var e {
      io:println(e.toString());
      return e;
   }
}
```

- `check` semantics is not simply to return on error
- When check gets an error, it *fails*
  - Enclosing block decide how to handle failure
  - Most blocks pass failure up to enclosing block
  - Function definition handles failure by returning the error
- `on fail` can catch the error
- `fail` statement is like `check` but always fails
- Differs from exceptions in that control flow is explicit

Rollback

```
function transfer(Update[] updates)
                        returns error? {
   transaction {
      foreach var u in updates {
         check doUpdate(u);
      }
      check commit;
   }
}

function doUpdate(Update u)
                        returns error? {
}
```

- If there is a fail or panic in the execution of the block, then the transaction is rolled back
- Transaction statement can also contain rollback statement
- Every possible exit from a transaction block must be one of
  - pass through explicit `commit`
  - pass through explicit `rollback`
  - fail exit (e.g. from check)
  - panic exit
- Rollback does not automatically restore Ballerina variables to values before the transaction

`retry transaction` statement

```
function demo() returns error? {
   // Short for
   // retry<DefaultRetryManager>(3)
   retry transaction {
      doStage1();
      doStage2();
      check commit;
   }
}
```

- Transactional errors are often transient: retrying will fix them
- This works by
  - creating a RetryManager object r, before executing the transaction
  - if the block fails with error e, it calls r.shouldRetry(e)
  - if that returns true, then it executes the block again
- `retry` has an optional type parameter giving class of RetryManager to create, and optional arguments to `new`
- `DefaultRetryManager` tries n times
- `retry` can be used without `transaction`

`transactional` qualifier

```
// called within transaction stmt
transactional function doUpdate(Update u)
                        returns error? {
   // call non-transactional function
   foo();
   // call transactional function
   bar();
}
function foo() {
   if transactional {
      // this is transactional context
      bar();
   }
}
transactional function bar() {
}
```

- At compile-time, regions of code are typed as being a transactional context - meaning guaranteed that whenever that region is executed, there will be a current transaction
- A function with a `transactional` qualifier can only be called from transactional context; function body will be a transactional context
- `transactional` is also a boolean expression that tests at runtime whether there is a current transaction: used in a condition results in transactional context

Distributed transactions

- Resource/remote method of service object can be declared `transactional`
- Remote method of client object can be declared as `transactional`
- Mostly a matter of implementation rather than additional language features
- Current transaction in Ballerina is actually a branch of a global transaction
- A client or Listener object can be transaction-aware
- Transaction-aware client object or Listener needs a network protocol
  - associate a network message with a global transaction
  - allow transaction manager of Ballerina program to communicate with other transaction managers
- Transaction-aware client object or Listener will makes calls to the Ballerina runtime's transaction manager
- When a transaction-aware Listener determines that the a request is part of a global transaction, it starts a new transaction branch for executing the service object remote/resource method

`transactional` named workers

```
// called within transaction stmt
transactional function exec(Update u)
                        returns error? {
   transactional worker A {
      bar();
   }
}

transactional function bar() {
}
```

- A named worker within a transactional function can be declared as transactional
- This will start a new transaction branch for the named worker, as with a distributed transaction

Commit/rollback handlers

```
transactional function update()
            returns error? {
  check updateDatabase();
  transaction:onCommit(sendEmail);
};
```

- Often code needs to get executed depending on whether a transaction committed
- Testing the result of the commit within the transaction statement works, but
  - inconvenient from a modularity perspective, particularly when you want to undo changes on rollback
  - much worse in a distributed transaction, when transaction statement is in another program
- Ballerina provides commit/rollback handlers - functions that get run when decision whether to commit is known

Concurrency safety

`lock` statement

```
int n = 0;

function inc() {
   lock {
      n += 1;
   }
}
```

- `lock` statement allows mutable state to be safely accessed from multiple strands that are running on separate threads
- Semantics are like an atomic section: execution of outermost lock blocks is not interleaved
- Naive implementation uses single, global, recursive lock
- Efficient implementation can do compile-time lock inference

Service concurrency

- Goal is "good enough" performance and "good enough" safety
- Good enough performance: Listener can service incoming requests concurrently
- Good enough safety: no undetected data races, but some errors detected at runtime rather than compile time
- Perfect safety would require type system that is more complex or restrictive
- Be able to look at the program and tell whether when it's safe for strands to be executed on separate threads
- `lock` by itself is not enough, because the user may not lock when they should

Isolated functions

```
type R record {
   int v;
};

final int N = getN();

isolated function set(R r) {
   r.v = N;
}

R r = {v: 0};

// This is not isolated
function setGlobal(int n) {
   r.v = n;
}
```

- Informal concept: a call to an isolated function is concurrency-safe if it is called with arguments that are safe at least until the call returns
- A function defined as isolated
  - has access to mutable state only through its parameters
  - has unrestricted access to immutable state
  - can only call functions that are isolated
- Constraints are enforced at compile-time
- `isolated` is part of the function type
- Weaker concept than pure function

`readonly` type

```
// Value of s is immutable array
readonly & string[] s = [
   "foo", "bar"
];

type Row record {
   // Both field and its value
   // are immutable
   readonly string[] k;
   int value;
};

table<Row> key(k) = table [
   // can safely use s as a key
   { key: s, value: 17 }
];
```

- A value belongs to type `readonly`, then the value is immutable
- For structural type T, `T & readonly` means immutable T
- `T & readonly` is subtype of `T` and subtype of `readonly`
- Guaranteed that if declared type of a value is a subtype of readonly, then at runtime value can never be mutated
    - enforced by runtime checks on mutating structures
- With `readonly` field, both the field and its value are immutable

`readonly` and `isolated`

```
type Entry map<json>;
type RoMap readonly & map<Entry>;

final RoMap m = loadMap();

function loadMap() returns RoMap {
   //...
}

isolated function lookup(string s)
      returns readonly & Entry? {
   return m[s];
}
```

- Isolated functions can access final variables with `readonly` type without locking
- Relies on the fact that immutability is deep
- `isolated` for functions complements `readonly` for data

Combining `isolated` functions and `lock`

- Goal is to allow isolated functions to use lock to access mutable module-level state
- Key concept is isolated root
- A value *r* is an *isolated root* if mutable state reachable from *r* cannot be reached from outside except through *r*
- An expression is an *isolated expression* if it follows rules that guarantee that its value will be an isolated root e.g.
  - an expression with a type that is a subtype of readonly is always isolated
  - an expression [E1, E2] is isolated if E1 and E2 are isolated
  - an expression f(E1, E2) is isolated if E1 and E1 are isolated, and the type of f is an isolated function

Isolated variables

```
isolated int[] stack = [];

isolated function push(int n) {
   lock {
      stack.push(n);
   }
}

isolated function pop() returns int {
   lock {
      return stack.pop();
   }
}
```

- When a variable is declared as isolated, compiler guarantees that it is an isolated root and accessed only within a lock statement
- Isolated variable declaration must be module-level, not public, initialized with isolated expression
- A `lock` statement that accesses an isolated variable must maintain isolated root invariant:
  - access only one isolated variable
  - call only isolated functions
  - transfers of values in and out must use isolated expressions
- Isolated functions are allowed to access isolated module-level variables, provided they follow the above rules

Isolated methods

- Object methods can be isolated
- An isolated method is the same as an isolated function with `self` treated as a parameter
- An isolated method call is concurrency-safe if both the object is safe and the arguments are safe
- This is not quite enough for service concurrency: when a Listener makes calls to a remote or resource method,
  - it can ensure the safety of arguments it passes
  - it has no way to ensure the safety of the object itself (since the object may have fields)

Isolated objects

```
isolated class Counter {
   private int n = 0;

   isolated function get()
                  returns int {
      lock {
         returns self.n;
      }
   }

   isolated function inc() {
      lock {
         self.n += 1;
      }
   }
}
```

- An object defined as isolated is similar to a module with isolated module-level variables
- Mutable fields of an isolated object
  - must be private and so can only be accessed using `self`
  - must be initialized with an isolated expression
  - must only be accessed within a `lock` statement
  - `lock` statement must follow the same rules for `self` as for an isolated variable
  - field is mutable unless it is final and has type that is subtype of `readonly`
- Isolated root concept treats isolated objects as opaque
- Isolated functions can access a final variable whose type is an isolated object

Inferring `isolated`

- `isolated` is a complex feature, which would be a lot for an application developer to understand
- A typical Ballerina application consists of a single module that imports multiple library modules
- Within a single module, we can infer `isolated` qualifiers
- Object w/o mutable fields is inherently isolated
- Application developer's responsibility is to use `lock` statement where needed
    - access self in a service object with mutable state
    - access mutable module-level variables
- Compiler can inform developer where missing locks are preventing a service object or method from being isolated

Part 3
Completing the picture

Default values for function parameters

```
function substring(
      string str,
      int start = 0,
      int end = str.length()
   ) returns int|() {
   //...
}
```

- Function parameters can have default values
- Defaults can use values of preceding parameters
- Type descriptor of a function value has closures to compute value for each defaultable parameter; each preceding parameter is parameter for the closure
- Caller of a function uses type descriptor to compute values for omitted defaultable parameters
- Does not affect type of function value

Providing function arguments by name

```
function foo(int x, int y, int z) {
}

// All these have the same  effect
foo(1, 2, 3);
foo(x = 1, y = 2, z = 3);
foo(z = 3, y = 2, x = 1);
foo(1, z = 3, y = 2);
```

- Arguments can be supplied by name as well as by position
- In a function call, named arguments are transformed into positional arguments using the function's type descriptor
- Argument list of a function described by a tuple type: names are not part of the type
- The names of arguments of remote methods and resource methods can be significant
- Changing argument names of public functions is an incompatible change to a module

Type inclusion for records

```
type Date record {
   int year;
   int month;
   int day;
};
type TimeOfDay record {
   int hour;
   int minute;
   decimal seconds;
};
type Time record {
   *Date;
   *TimeOfDay;
};
```

- Use *T to include a record type in a record type descriptor
- Effect is similar to copying fields of included record into including record

Included record parameters

```
type Options record {|
   boolean verbose = false;
   string? outputFile = ();
|};

function foo(string inputFile,
             *Options options) {
}

function main() {
   foo("file.text",
        verbose = true);
}
```

- With named arguments
  - function defines each parameter normally
  - caller supplies parameter by name
- With record-typed parameter
  - function uses record for all named parameters
  - caller supplies arguments using mapping constructor
- With included record parameter
  - function defines records for named parameter
  - caller supplies parameter by name
- Named arguments and included record parameters provide consistent experience for caller

Default values for record fields

```
type X record {
   string str = "";
};

X x = {};
```

- Record fields can have a default value
- A default value is specified with an expression, which must satisfy rules for body of isolated function
- Default value does not affect static typing: affects only use of type descriptor to construct record
- `cloneWithType(T)` will make use of defaults specified by T
- *T also copies default values: it copies the closure to compute value in the context of the original declaration

Object types

```
type Hashable object {
   function hash() returns int;
};

function h() returns Hashable {
  var obj = object {
     function hash() returns int {
        returns 42;
     }
  };
  // obj belongs to Hashable type
  return obj;
};
```

- Class definition combines object type with implementation - a function that can create objects belong to the object type
- Can define object type without implementation
- Object typing is structural: object type looks like pattern that object must match
- Analogous to interface in Java

Object type inclusion

```
type Cloneable object {
  function clone()
            returns Cloneable;
};

type Shape object {
   *Cloneable;
   function draw();
};

class Circle {
   *Shape;
   function clone() returns Circle {
       return new;
   }
   function draw() { }
}
```

- Ballerina does not have implementation inheritance
- *T can be used to include an object type T in another object type
- Constrained so including object type is a subtype of every included object type
- Provides interface inheritance
- Class declaration can include object type to check that class belongs to object type

Distinct object types

```
type Person distinct object {
   public string name;
};

distinct class Employee {
   *Person;
   function init(string name) {
      self.name = name;
   }
};
distinct class Manager {
   *Person;
   function init(string name) {
      self.name = name;
   }
};
```

- Objects are structurally typed like everything in Ballerina
- Distinct object types provide similar functionality to nominal typing within a structurally typed framework
- A distinct object value is tagged (branded) with a type-id that is unique to that occurrence of `distinct` in the source
- Both object type and class can be distinct
- Useful for interop with nominally typed object-oriented systems (e.g. Java, GraphQL)

Readonly objects and classes

```
type TimeZone readonly & object {
   function getOffset(decimal utc)
                  returns decimal;
};

readonly class FixedTimeZone {
   *TimeZone;
   final decimal offset;
   function init(decimal offset) {
      self.offset = offset;
   }
   function getOffset(decimal utc)
                      returns decimal {
      return self.offset;
   }
}
```

- Object is readonly if its fields are all final and have readonly type
- readonly & T is allowed when T is an object type
- If class declaration uses readonly, then object type defined by class is readonly & T, where T is type defined in class body

Error detail

```
error err =
  error("Whoops", httpCode = 27);

type HttpDetail record {
  int httpCode;
};

error<HttpDetail> err =
  error("Whoops", httpCode = 27);

HttpDetail d = err.detail();
```

- An error value contains map containing arbitrary extra details about the error
- Type error<T> describes error value with detail map that has type T
- Named arguments for error constructor specify fields of detail record
- An immutable copy is made of each field using cloneReadOnly function

Error cause

```
function foo(string s)
            returns error|int {
  var res = int:fromString(s);
  if res is error {
    return error("not an integer",
                 res);
  }
  else {
     return res;
  }
}
```

- error value has cause of type `error?`
- error(msg, cause) creates error with specified error and cause
- err.cause() gets the cause of an error

Type intersection for error types

```
type IoError distinct error;

type FileErrorDetail record {
   string filename;
};

type FileIoError
  IoError & error<FileErrorDetail>;
```

- Use intersection to define an error type based on both error detail and distinct type

Type intersection

```
type Foo object {
  function foo();
};

type Bar object {
  function bar();
};

type FooBar Foo & Bar;

// same as
type FooBar object {
   *Foo;
   *Bar;
};
```

- Type intersection works generally not just for `readonly`
- T_1 & T_2 means set intersection of types T_1 and T_2
- Convenient for `object` as well as `readonly` and `error`
- Cannot do everything that can be done with type inclusion

Expression-oriented style

```
function inc(int x) returns int => x + 1;

// same as

function inc(int x) returns int {
   return x + 1;
}

var obj = object {
   private int x = 1;
   function getX() returns int => self.x;
};

// let expressions
function hypot(float x) =>
   let float x2 = x * x in
      float:sqrt(x2 + x2);
```

- Ballerina supports statements for familiarity but also tries to enable an expression-oriented style of programs
- Query expressions, constructors, nil return type support expression-oriented style
- When function body is an expression can use => instead of block with returns
- Works for methods also
- Let expressions (like let clauses in query expressions) allow you to do more with an expression

Computed field key

```
const X = "x";
const Y = "y";

map<int> m = {
  [X]: 1,
  [Y]: 2
};
```

- In a mapping constructor, field name can be an expression in square brackets
- Particularly useful when you want to define constants for key values
- Could be done with statements using assignments

Tuples

```
// Fixed length array
type FloatPair float[2];
// Tuple
type FloatPair [float, float];

FloatPair p = [1.0, 2.0];

// Can mix types
type Id [string, int, int];

byte[*] a = base16`DEADBEEF`;
```

- Arrays and tuples are two ways of describing lists
- Tuples are constructed like arrays
- Tuple is to array as record is to map
- Arrays can be fixed length
- * for length infers fixed length from initializer

Destructuring tuples

```
type Time [int, decimal];

function toSeconds(Time ip)
           returns decimal {
   var [day, seconds] = ip;
   decimal s = 86400d*<decimal>day;
   s += seconds;
   return s;
}
```

- Like records, tuples can be destructured with a binding pattern

Binding patterns in assignment

```
int x = 0;
int y = 1;

type IntPair [int, int];

function assign(IntPair ip) {
   [x, y] = ip;
}

function swapXY() {
   [x, y] = [y, x];
}
```

- Binding patterns can be used in assignment statements

Rest type in tuples

```
// int followed by
// zero or more strings
type Id [int, string...];
```

- Already seen `T...` in record types
- `T...` also works in tuples
- `T[]` same as `[T...]`
- Tuples not open by default

Array/map symmetry

Basic type	Index type	JSON	Constructor	Type with uniform member type	Type with per-index member type	Open type
list	int	array	["foo", "bar"]	array T[]	tuple [T0,T1]	[T0,Tr...]

mapping	string	object	{
			x: "foo",
			y: "bar"
			}
				map
				
				map<T>
					record
					
					record {
					Tx x; Ty y;
					}
						record {|
						Tx;
						Tr...;
						|}

Rest parameters

```
function foo(int n, string... s) {
}

function bar() {
   // Param s will be ["x", "y", "z"]
   foo(1, "x", "y", "z");
}

service on hl {
  // With URL file/x/y/z
  // path will be ["x", "y", "z"]
  resource function
    get file/[string... path]()
       returns string|error {
}
```

- Functions can have rest parameters (varargs)
- Parameter `T... p` will make p have type `T[]`
- Also works with resource path parameters

Spread operator `...x`

```
type Date record {|
   int year; int month; int day;
|};
type TimeOfDay record {|
  int hour; int minute; int second;
|};
type DateTime record {|
  *Date; *TimeOfDay;
|};
function merge(Date d, TimeOfDay t)
   returns DateTime {
  return { ...d, ...t };
}
```

- `...x` where x is a list or mapping is equivalent to specifying each member of x separated by a comma
  - x is list - positional
  - x is mapping - named
- Works in
  - `f(...x)` - mapping or list
  - `[...x]` - list
  - `{...x}` - mapping
  - `error(msg, ...x)` - mapping
- Static type of x must ensure equivalent with each members is valid

Spread in binding patterns

```
type Id [int, string...];

function process(Id id) {
  var [n, ...path] = id;
  foreach string s in path {
    io:println(s);
  }
}
```

- `...x` works in binding patterns for
  - mappings
  - lists
  - errors
- Useful with open records

Binding patterns in `match` statement

```
type Pair record {
  int x;
  int y;
};

function foo(Pair pair) {
   match pair {
      var {x, y, ...rest} => {
        io:println(x, ", ", y, ", ",
                   rest);
      }
   }
}
```

- Variable part of match pattern is specified by binding pattern
- Match patterns: identifiers refer to constants
- Binding patterns: identifiers refer to variables
- Use `var` in a match pattern to include a binding pattern

`never` type

```
function whoops() returns never {
   panic error("whoops");
}

type Pair record {
  int x;
  int y;
};
Pair p = {
   x: 1, y: 2, "color": "blue"
};
var {x: _, y: _, rest } = p;
// Type of `rest` is
type PairRest
   record { never x?; never y?; };
```

- No value belongs to the `never` type
- Variable cannot have type `never`
- For a function, never means that it cannot return normally
- Other use cases:
  - `stream<int,never>` means infinite stream
  - `xml<never>` is type of empty xml sequence
  - Open record with optional field of type `never` is open to everything except that field

Interfacing to external code

```
public function open(string path)
               returns handle|error
   = external;
```

- Function body can be defined as `= external`
- `external` keyword can be annotated to say where the implementation comes from
- `handle` type represents opaque handle for use by external functions
  - in a JVM implementation might contain a reference to an object
- `handle` can be wrapped in an object for better type safety
- Alternative is to have an entire module that is implemented in something other than Ballerina

Built-in integer subtypes

```
function srand(int:Unsigned32 seed)
   = external;

function demo1(int n) {
  // OK: 72 is an int:Unsigned32
  srand(72);
  // use lo bits
  srand(n & 0xFFFFFFFF);
  // panic if out of range
  srand(<int:Unsigned32>n);
}
```

- Generalization of `byte` type
- Built-in subtype provided by int module
- int type has built-in subtypes
  - int:Signed32, int:Unsigned32
  - int:Signed16, int:Unsigned16
  - int:Signed8, int:Unsigned8 (same as byte)
- Runtime behaviour of operations on subtypes same as for `int` type
- Bitwise operations have special typing
- Useful for interfacing with external systems that use these types
- Allows implementation to optimize storage, particularly for arrays

Built-in string subtype

```
string:Char ch = "x";
int cp = ch.toCodePointInt();
```

- A string belongs to string:Char if it has length 1
- Analogous to built-in subtypes of xml
- A string:Char value can be converted to a code point represented as an int

`typedesc` type

```
type R record {
   int x;
   int y;
};

typedesc<record {}> t = R;

// Will return true
function demo() returns boolean {
   R r = { x: 1, y: 2 };
   any v = r;
   return typeof v === t;
}
```

- Built-in type representing a type descriptor
- Immutable: subtype of `readonly`
- A typedesc value belongs to type typedesc<T> if the type descriptor describes a type that is a subtype of T
- Using name of type definition in an expression
- `typeof` operator gets dynamic type of a value
- Dynamic type for mutable structure is inherent type

`ensureType` function

```
function demo(anydata v)
          returns float|error {
   return v.ensureType(float);
}
```

- `ensureType` langlib function is like a cast but gives an error rather than a panic if the cast cannot be done
- Does numeric conversions like a cast

Dependent types

```
// Declaration in lang.value
public isolated function ensureType(
       any|error v,
       typedesc<any> t = <>
  ) returns t|error
  = external;


function demo(json j) returns error? {
   float f = check j.ensureType();
}
```

- Type of result of function depends on value of parameter
- Not the same as generic functions
- Ballerina supports this for parameters of type `typedesc`
- Limited to external functions for now
- Parameter default value of `<>` means that default value of dependent type parameter from context of function call

Annotation declaration and access

```
// Module m
public type IntConstraints {
   int minInclusive?;
   int maxInclusive?;
};

public
annotation IntConstraints
      ConstrainedInt on type;

// In another module
@m:ConstrainedInt { minInclusive: 1 }
type PositiveInt int;

m:IntConstraints? c
   = PositiveInt.@m:ConstrainedInt;
```

- Modules can declare an annotation tag
- Declaration says
  - what syntactic constructs tag can be applied to
  - type of value associated with tag
- Annotations are accessed at runtime from a typedesc value using `.@` operator

Trapping panics

```
function safeAdd(int n1, int n2)
                returns int|error {
   // On overflow, get an error
   // rather than a panic
   return trap (n1 + n2);
}
```

- Panics can be trapped with a `trap` expression

Additional resources

Implementation: https://ballerina.io/downloads/#swanlake

James Clark's blogs on Ballerina: https://blog.jclark.com/search/label/Ballerina

Online Ballerina docs: https://ballerina.io/swan-lake/learn/

Language specification: https://ballerina.io/spec/lang/draft/latest/

Language issues: https://github.com/ballerina-platform/ballerina-spec/issues
(Use this to ask questions and provide feedback on this presentation)

Inferring type from context: numeric literals

```
int n = 1; // int
decimal n = 1; // decimal
float n = 1; // float
int|float|decimal n = 1; // int

float n = 1.0; // float
decimal n = 1.0; // decimal
float|decimal n = 1.0; // float
```

- Don't always need a `d` suffix for decimals
- A literal integer can be interpreted as int or float or decimal depending on context; defaults to int
- A literal floating point number can be interpreted as float or decimal depending on context; defaults to float

Not yet explained

- `fork` statement