THE WIT CHEATSHEET

File Structure

Package
A single package can and often is represented by multiple files. A common pattern is types.wit, world.wit, and my-interface-name.wit. The package must include a namespace and an identifier, and can optionally include a version number which should follow semantic versioning.

↳ Interface
A collection of types and functions scoped to a package which can be used within a world. Interfaces are the only place that a type can be defined. Packages can contain multiple interfaces.

↳ World
Akin to a complete description of a component. A world is a collection of imports and exports that allow the component to interact with the host as well as other components.

Language Keywords
These keywords are used to describe some sort of action or property within a component.

`include`	used to include worlds into another
`use`	will include types from an interface in a world
`import`	used to import an entire interface into a world
`export`	used to export an interface/func from a world

Named Types
Types listed here all require an identifier. They provide more structure and flexibility than primitive types.

`record`	a collection of types accessible via named keys
`enum`	a type that can equal one of a set of values
`variant`	a type that be one of many predefined types
`flags`	a collection of flags that can be toggled independantly from each other
`func`	a function type
`resource`	a new abstract resource type. See below

Resource Keywords
Keywords here relate to resources. Resources are used to describe variables that should not be copied by value.

`constructor`	method that returns a handle of the containing resource type
`static`	mark a resource func as scoped to resource
`self`	the borrowed handle to the resource
`borrow`	mark a handle as a temporary loan from caller to callee
`owned`	a handle representing unique ownership of a resource

WASI Component Types

Command
A component defined as a Command has a main function (or in wasi:cli, a run function) and terminates when the function returns.

Reactor
This component is more like a library. It exposes an API once it is instantiated and remains live, allowing functions on the component to be called from the host or other components.

Primitive Types
These are the core types for a wasm module. Most languages will implement all of these types.

`bool`	boolean value; true or false
`s8, s16, s32, s64`	signed integers with 8, 16, 32 or 64 bit length
`u8, u16, u32, u64`	unsigned integers with 8, 16, 32 or 64 bit length
`float32, float64`	floating-point numbers with 32 or 64 bit length
`char`	single Unicode scalar character
`string`	Unicode string with a finite length

Container Types
These types are used to create collections of other types. They can contain any of the primitive or named types.

`tuple<...>`	a finite sequence of values of different types
`list<T>`	a sequence of values of the type T
`option<T>`	mark a type as optional, value will be type T or no value
`result<T,E>`	represents value or error, where the output will be one or the other but not both

Comments
Sometimes you just need to leave a note.

```
/// single line comment
/* comment block */
```

WASI Worlds to know
These are some useful worlds defined by the WASI proposals that you might find useful in writing your own components.

`wasi:cli/command`
`wasi:keyvalue/keyvalue`
`wasi:blob-store/blob-store`
`wasi:messaging/messaging`
`wasi:nn/ml`
`wasi:http/proxy`
`wasi:cloud-core/cloud-core`

WHAT DOES WIT LOOK LIKE?

a **package** is a collection of interfaces and worlds

an **interface** is a collection of functions and types scoped to the package

Identifiers are restricted to ASCII **kebab-case** but can be preceded by a single **%** if the identifier would otherwise be a wit keyword. For example, **interface** is a keyword, but **%interface** is an identifier

worlds describe the imports and exports for a component

worlds can **import** and **export** entire interfaces, **use** types from interfaces and can **include** other worlds

```
package acme:space-station@0.1.0;

interface types {
  type astronaut-id = u64;

  variant pods {
    none,
    list<u32>,
  }

  flags locations {
    bridge,
    nacelle,
    jefferies-tubes,
  }

  enum level {
    captain,
    commander,
    cadet,
  }

  record astronaut {
    id: astronaut-id,
    name: string,
    ship-access: locations,
    manager: option<astronaut>,
    level: level,
    start-date: u32,
    end-date: option<u32>,
    addresses: pods,
  }

  record inventory {
    name: string,
    cost: u32,
    description: string,
    stock: u32,
    tags: list<string>,
  }
}

interface directory {
  use types.{astronaut-id, astronaut};

  get-astronaut: func(id: astronaut-id) -> result<astronaut, e32>;
  update-astronaut: func(id: astronaut-id, changes: droid) -> result<astronaut, e32>;
}

world astronauts {
  import wasi:logging;

  export directory.{get-astronaut, update-astronaut};
}

world reporting {
    include astronauts;
    use types.{inventory};

    export get-inventory: func(item: option<string>) -> list<inventory>;
}
```

package must have a namespace and name—version is optional

type aliases are great for descriptive declarations

variants can be used to indicate that a value can be one of many types

flags are like a collection of bools that can be toggled individually

enums are a single type for a small set of specific values

id is referenced from the **type** alias above!

option<...> is useful for making values not required

record properties can be any valid type, including other **records**

lists can include any type. Indicate contents as many types using **variants**

result is great for returning error and/or success state

imports can be called from the component using them

exports are functions that can be called by the host runtime (or another component)

cosmonic

THE FUTURE OF DISTRIBUTED APPLICATIONS, TODAY

From napkin sketch to running apps anywhere, at scale, in minutes, Cosmonic is the lightweight, low-boilerplate platform that radically simplifies application development. Build your apps with composable Wasm components that run in any datacenter, cloud or edge.

Cosmonic

wasmCloud