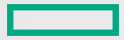
I/O Serializers

Ben Harshbarger

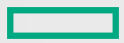
October 8, 2025

Overview

- Using Serializers
- Customizing Serialization of Types
- Implementing Serializers



Using Serializers



I/O Basics

- Chapel's **IO** module provides a 'file' type, which can be used to create a 'fileReader' or 'fileWriter'

```
use IO;

var f: file = open("foo.txt", ioMode.cwr);

var w = f.writer();
w.writeln("Hello!");
w.flush();

var r = f.reader();
const str = r.read(string);
writeln("read: ", str); // prints 'read: Hello!'
```

I/O Basics

- Chapel's **IO** module provides a 'file' type, which can be used to create a 'fileReader' or 'fileWriter'

```
use IO;

var f: file = open("foo.txt", ioMode.cwr);

var w = f.writer();
w.writeln("Hello!");
w.flush();

var r = f.reader();
const str = r.read(string);
writeln("read: ", str); // prints 'read: Hello!'
```

What dictates how "Hello!" is printed? **Serializers**!

What are Serializers?

- Serializers are records that implement an API that defines how the I/O module will read or write values
  - Primitive types (e.g. int, string), records, classes, arrays, etc.
  - Generally implemented using basic I/O methods but can invoke other serializers
  - Serialization: the process of turning objects into bytes or text, to store on disk or transmit over a network

- Serializers also invoke methods on classes and records that allow users to customize serialization
  - For writing, invokes a method called 'serialize'
  - For reading, invokes either an initializer or a method called 'deserialize'
  - The Chapel compiler generates default implementations of these for all classes and records

- For more information than contained in this tutorial, refer to the I/O Serializers technote

The Default Serializer

- `fileReader` and `fileWriter` types have a default serializer implementation
  - ... via a type called `defaultDeserializer` or `defaultSerializer` defined in the `IO` module

- Methods for creating a `fileReader` or `fileWriter` have a 'deserializer' or 'serializer' default argument

```
var w = f.writer();
var r = f.reader();

// equivalent to...

var w = f.writer(serializer=new IO.defaultSerializer());
var r = f.reader(deserializer=new IO.defaultDeserializer());
```

Other Serializers

- Chapel's standard library also includes serializers for JSON and a rudimentary binary format
  - `binarySerializer` and `binaryDeserializer` can be found in the `IO` module
  - JSON (de)serializers can be found in the `JSON` module
  - Others, like the `YAML` module, are currently unstable

```
use IO, JSON;

record R { var x: int; }

var f = new file(1); // opens stdout, assuming POSIX system...
var wj = f.writer(serializer=new jsonSerializer());

var rec = new R(42);
wj.writeln(rec); // prints: {"x":42}
```

The 'withSerializer' method

- The 'withSerializer' method allows for using a different serializer with an existing `fileWriter`
  - For `fileReader`, use the `withDeserializer` method instead

- Let's use this to simplify our previous example

```
use IO, JSON;

record R { var x: int; }

// use the builtin 'stdout' variable from the 'IO' module
var wj = stdout.withSerializer(new jsonSerializer());

var rec = new R(42);
wj.writeln(rec); // prints: {"x":42}
```

Expanded Example

```
use IO, JSON;

record X { var data: int; }

var f = openMemFile();

var w = f.writer(serializer=new jsonSerializer());
var rec = new X(42);
w.writeln(rec); // prints to file: {"data":42}
w.flush();

var r = f.reader(deserializer=new jsonDeserializer());
var x = r.read(X);
writeln(x.data); // prints '42'
```

Exercise 1: Basic Serializer Usage

- Write a list as JSON, then read it back in
- Starter code:

```
use IO, List, JSON;

var li : list(int);
for i in 1..10 do li.pushBack(i);

var f = openMemFile();

// a) write out to file

// b) read back in
```

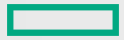
Exercise 1: Basic Serializer Usage

- Write a list as JSON, then read it back in
- Solution:

```
// a) write out to file
var w = f.writer(serializer=new jsonSerializer());
w.write(li);
w.flush();

// b) read back in
var r = f.reader(deserializer=new jsonDeserializer());
var li2 = r.read(list(int));
writeln(li2);
```

Customizing Serialization of Types



Customizing Serialization

- The (De)Serializers API can be broken into roughly three pieces
  1. Methods called by the 'IO' module to hand off control to a (De)Serializer (relevant for (De)Serializer authors)
  2. Methods a (De)Serializer can invoke on user types to allow for customized I/O
  3. Methods a user-defined type can invoke on a (De)Serializer to perform format-agnostic I/O

- (De)Serializers support format-agnostic I/O for several kinds of abstract types
  - For example, many formats support their own notion of a "List" or "Map"
  - A portion of the API is devoted to each kind of abstract type

See the IO Serializers technote for full details of the API

User-Defined Serialization on Types

- The 'serialize' method is invoked on classes and records
  - Defined by the 'writeSerializable' interface:

```
proc T.serialize(writer: fileWriter(?), ref serializer: ?st) throws
```

- Can be implemented for any serializer, or specialized by type:

```
record MyType : writeSerializable {
  var name: string;
  var id: int;
}

proc MyType.serialize(writer: fileWriter(?), ref serializer: ?st) throws {
  writer.write("<MyType: ", this.name, ">");
}
```

Exercise 2: Custom Serialize Method

- Make the 'Counter' record print as an integer, using a custom 'serialize' method

```
use IO;

record Counter {
  var count: int;

  proc ref inc() { count += 1; }
}

var c : Counter;

for i in 1..100 {
  if i % 7 == 0 && i % 3 == 0{
    c.inc();
  }
}

writeln(c);
```

Before:
```
(count = 4)
```

After:
```
4
```

Exercise 2: Custom Serialize Method

- Solution:

```
proc Counter.serialize(writer: fileWriter(?), ref serializer: ?) throws {
  writer.write(count);
}
```

User-Defined Serialization on Types

- Example continued: a 'serialize' method only for JSON:

```
proc MyType.serialize(writer: fileWriter(serializerType=jsonSerializer),
                      ref serializer: jsonSerializer) throws {
  writer.writeLiteral('{"name": ');
  writer.write(this.name);
  writer.writeLiteral("}");
}
```

- Here we use 'writeLiteral' to bypass serializers entirely, and write the string as-is
  - Otherwise, with 'writer.write' it would print as a JSON string
  - Somewhat necessitates re-inventing the wheel by having to comply with the format's specifications explicitly

- But there's a better way to do this without re-inventing the wheel: the format-agnostic API

Format-Agnostic API

- Serializers provide six ‘start’ methods to begin serializing a kind of type
  - Type-kinds: Class, Record, Tuple, Array, List, Map
- Each ‘start’ method takes a ‘fileWriter’ and returns a record with methods for the specific type-kind
  - Each ‘start’ method also accepts a ‘size’ argument to represent, e.g., the number of fields or elements

	Class	Record	Tuple	Array	List	Map
Methods on serializer	startClass	startRecord	startTuple	startArray	startList	startMap
Methods on returned record	writeField	writeField	writeElement	writeElement	writeElement	writeKey
	startClass*			startDim		writeValue
				endDim		
	endClass	endRecord	endTuple	endArray	endList	endMap

* note: second ‘startClass’ exists to support inheritance

Format-Agnostic API

- Let's use the format-agnostic API to write this as a record with just one field

Record
startRecord
writeField
endRecord

```
use IO, JSON;
record MyType : writeSerializable {
  var name: string;
  var id: int;
}
proc MyType.serialize(writer: fileWriter(?), ref serializer: ?st) throws {
  var ser = serializer.startRecord(writer, /*name=*/"MyType", /*size=*/1);
  ser.writeField("name", this.name);
  ser.endRecord();
}

var mt = new MyType("Sam", 1);
stdout.writeln(mt);
stdout.withSerializer(new jsonSerializer()).writeln(mt);
```

```
(name = Sam)
{"name":"Sam"}
```

Format-Agnostic API: Example

- Example usage: Write a type as an abstract 'List' (demo7.chpl):


```
// first, explicitly indicate interface
record MyList : writeSerializable { /*...*/ }
proc MyList.numElements : int { /*...*/ }
iter MyList.these() : int { /*...*/ }

// Write once, use with any Serializer
proc MyList.serialize(writer: fileWriter(?), ref serializer: ?st) throws {
  var ser = serializer.startList(writer, this.numElements); // in JSON, write "["
  for elem in this do
    ser.writeElement(elem); // in JSON, write "," if necessary, then 'elem'
  ser.endList(); // in JSON, write "]"
}
```

- If printing a list of squared numbers from 1 to 10 in JSON:

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Format-Agnostic API

- Deserializers also provide six ‘start’ methods to begin deserializing kinds of type

- Each ‘start’ method takes a ‘fileReader’, and returns an object with methods for the specific type-kind
  - The various ‘read’ methods accept either a value by ‘ref’, or a ‘type’, to match ‘fileReader.read’

Class	Record	Tuple	Array	List	Map
startClass	startRecord	startTuple	startArray	startList	startMap
readField	readField	readElement	readElement	readElement	readKey
startClass*			startDim		readValue
			endDim	hasMore	hasMore
endClass	endRecord	endTuple	endArray	endList	endMap

* note: second ‘startClass’ exists to support inheritance

Format-Agnostic API: Example Deserialization

- Users may override default in-place deserialization behavior with a ‘deserialize’ method
  - The ‘deserialize’ method is defined by the ‘readDeserializable’ interface:

```
proc ref T.deserialize(reader: fileReader(?), ref deserializer: ?dt) throws
```

- Intended to provide behavior for ‘fileReader.read’ that accepts values by-ref

Format-Agnostic API: Example Deserialization

Example usage: Read a type as an abstract ‘List’ (demo8.chpl):

```
record MyList : writeSerializable, readDeserializable { ... }
// assume we have methods like 'clear' and 'append'

// Write once, use with any Deserializer
proc ref MyList.deserialize(reader: fileReader(?), ref deserializer: ?dt) throws {
  this.clear(); // reading in-place, so clear the data
  var des = deserializer.startList(reader); // in JSON, reads '['
  while des.hasMore() do // in JSON, checks for more elements
    this.append(des.readElement(int));
  des.endList(); // in JSON, reads ']'
}
```

List
startList
readElement
hasMore
endList

The Deserializing Initializer

- Users may override default 'read(type)' deserialization behavior with an initializer
  - Useful for types that cannot be default-initialized
  - The initializer signature is defined by the ‘initDeserializable’ interface:

```
proc T.init(reader: fileReader(?), ref deserializer: ?dt) throws
```

- Initializer may throw, but only after all fields are initialized
  - Future editions of Chapel may relax this requirement

- Otherwise, works the same as a ‘deserialize’ method

- See IO Serializers technote for information on initializing generic types while deserializing

- For example, see 'demo9.chpl' in tutorials repo

Exercise 3: Format-Agnostic API

- Add implementation to serialize coordinates as a tuple
- Starter code:

```
use IO, List;

record point { var x, y: int; }

var li : list(point);
for i in 1..10 by 2 do li.pushBack(new point(i, i+1));

var f = openMemFile();
var w = f.writer();
w.writeln(li);
w.flush();

var r = f.reader();
var li2 = r.read(list(point));
writeln("read: ", li2);
```

Before:

```
read: [(x = 1, y = 2), (x = 3, y = 4), (x = 5,
y = 6), (x = 7, y = 8), (x = 9, y = 10)]
```

After:

```
read: [(1, 2), (3, 4), (5, 6),(7, 8), (9, 10)]
```

Exercise 3: Format-Agnostic API

- Method signatures to get you started:

```
proc point.serialize(writer: fileWriter(?), ref serializer: ?) throws {
}

proc ref point.deserialize(reader: fileReader(?), ref deserializer: ?) throws {
}

proc point.init(reader: fileReader(?), ref deserializer: ?) throws {
}

// adding a user-defined initializer for reading prevents generation of the
// default initializer that takes two ints, so recreate it here
proc point.init(x: int, y: int) {
  this.x = x;
  this.y = y;
}
```

Exercise 3: Format-Agnostic API

- Relevant Format-Agnostic API Methods

```
proc Serializer.startTuple(writer: fileWriter(?),
                           size: int) : TupleSerializer throws { ... }

proc TupleSerializer.writeElement(const elt: ?T) throws { ... }
proc TupleSerializer.endTuple() throws { ... }


proc Deserializer.startTuple(reader: fileReader(?)) : TupleDeserializer throws;

proc TupleDeserializer.readElement(type eltType) : eltType throws
proc TupleDeserializer.endTuple() throws
```

- Links to the serializers technote:
  - Serializer tuple methods
  - Deserializer tuple methods

Exercise 3: Format-Agnostic API

- 'serialize' solution:

```
proc point.serialize(writer: fileWriter(?), ref serializer: ?) throws {
  var ser = serializer.startTuple(writer, 2);
  ser.writeElement(x);
  ser.writeElement(y);
  ser.endTuple();
}
```

- For full solution, refer to 'solution3.chpl' in the github tutorial repo for ChapelCon25

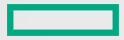
Other API Notes

- User types implementing all three methods can use the combined ‘serializable’ interface
  - Reminder: 'serialize', 'deserialize', and 'init'

- ‘serialize’ and ‘deserialize’ methods on classes must use ‘override’
  - Required because all classes inherit from the RootClass, which can itself be serialized or deserialized

- Implementing ‘serialize’, ‘deserialize’, or an initializer prevents compiler-generation of all three
  - Rationale: User has possibly diverged from default behavior, so do not generate incompatible implementations

Implementing Serializers



Implementing Serializers

- To implement a Serializer, users must implement a ‘serializeValue’ method on a record

```
proc YourSerializer.serializeValue(writer: fileWriter(?), const val: ?) throws
```

- ‘serializeValue’ accepts either primitive types, or types with the ‘writeSerializable’ interface
- Once invoked, ‘serializeValue’ has complete control over serialization
- Users must also implement the format-agnostic API of the previous section

Implementing Deserializers

- For a Deserializer, users must implement ‘deserializeValue’ and ‘deserializeType’ methods

```
proc YourDeserializer.deserializeType(reader: fileReader,
                                      type readType) : readType throws

proc YourDeserializer.deserializeValue(reader: fileReader,
                                       ref val: ?readType) : void throws
```

- These methods accept types with either the ‘readDeserializable’ or ‘initDeserializable’ interface
  - Or primitive types
- Once invoked, these methods have complete control over deserialization
- Users must also implement the format-agnostic API of the previous section

References for Implementing (De)Serializers

- Several modules implement the (De)Serializer API:
  - Standard modules
    - JSON
    - PrecisionSerializer
  - Unstable package modules:
    - YAML (uses a third-party library extensively)
    - ObjectSerialization (binary format)
    - ChplFormat (attempts to print values as Chapel code/literals)

- The test/io/serializers/ directory in the Chapel repository may also be useful
  - Contains various stress tests of common cases, easy to plug in a different serializer

Thank you