The Managed Data Structures Library: Java API

The Managed Data Structures Project, Hewlett Packard Labs

Monday, July 31, 2017

Contents

Introduction

The What and Why of Managed Data Structures

A Simple MDS Example

Before getting into the gory details of programming with MDS, we will first present a very simple worked example. To get the feel for what MDS brings, we'll first sketch out how you might program it if you weren't worrying about persistence or concurrency or multi-process sharing, then we'll discuss what you might do in a non-MDS world to take those into account, and finally, we'll walk through what it would look like with MDS.

The Scenario

Imagine that you are in charge of processing sales for a retailer. The company sells a large number of products, and for each product you have to keep track of how many are in stock, how many have been sold, what the current price is, and what the total revenue for the product is.

From time to time, you need to process one of the following actions that come in from outside:

- A new product has been introduced. You are told the product SKU, its initial price, and the initial stock.
- Stock has come in on a product. You are told the product SKU and the quantity received.
- The price has been changed on a product. You are told the product SKU and the new price.
- You are asked for a price check on a product. You are told the product SKU and return the current price.
- A sale has been made. You are given a list of line items, each containing a product SKU, a quantity, and a unit price. You must confirm that there is sufficient stock for each and that the price for each is correct. If it is, you must adjust the stock on hand and revenue for each.

These are all examples of transactional processing and should be done atomically. From time to time, you will also get requests to provide various reports that may require walking over all of the products and may take non-trivial time. It is important that the report be *consistent*. That is, everything the report says must be true as of some instant in time, as if a snapshot of all of the data had been taken. In our simple scenario, the report we will be asked to provide will contain the total revenue over all products, an array containing the top ten products by revenue, and the value of current stock. In addition to being asked for reports, you are expected to create a report once a day and stick it somewhere where it can be found, so that trends can be identified.

Doing it Without MDS

Doing this without MDS—and without worrying about persistence, concurrency, or sharing—is straightforward. You create a *Product* class that contains the SKU, number in stock, number sold, price, and revenue, and you also provide a way to look up a product by SKU and a way to walk through all of the products. Since we're not worrying about products going away, for this example, we can support walking through the products simply by having each product point to the next product in the inventory. So we'll create a *Product* class that looks like

```
class Product {
    private static final HashMap<String, Product> bySKU = new HashMap<>;
    static Product firstProduct = null;

    final String sku;
    int nInStock;
    int nSold;
    float price;
    float revenue;
    final Product nextProduct;

    Product(String sku, float price, int nInStock) {
        this.sku = sku;
        this.price = price;
        this.nInStock = nInStock;
        nextProduct = firstProduct;
        firstProduct = this;
        bySKU.put(sku, this);
    }
```

```
    void check(int q, float p) throws WrongPrice, InsufficientQuantity {
        if (price != p) { throw new WrongPrice(sku, p, price); }
        if (nInStock < q) { throw new InsufficientQuantity(sku, q, nInStock); }
    }

    void sell(int q, float p) {
        revenue += p*q;
        nSold += q;
        pnInStock –= q;
    }

    static boolean exists(String sku) {
        return bySKU.contains(sku);
    }

    static Product lookup(String sku) throws NoSuchSKU {
        Product p = bySKU.get(sku);
        if (p == null) {  throw new NoSuchSKU(sku); }
        return p;
    }
};
```

The actions then become straightforward:

```
Product newProduct(String sku, float price, int nInStock) throws SKUExists {
    if (Product.exists(sku)) { throw new SKUExists(sku); }
    return new Product(sku, price, nInStock);
}

void stockIn(String sku, int n) throws NoSuchSKU {
    Product.lookup(sku).nInStock += n;
}

void newPrice(String sku, float p) throws NoSuchSKU {
    Product.lookup(sku).price = p;
}

float priceCheck(String sku) throws NoSuchSKU {
    return Product.lookup(sku).price;
}

void processSale(List<LineItem> items)
        throws NoSuchSKU, WrongPrice, InsufficientQuantity
{
    for (LineItem item : items) {
        Product p = Product.lookup(item.sku);
        p.check(item.quantity, item.price);
    }
    for (LineItem item : items) {
        Product p = Product.lookup(item.sku);
        p.sell(item.quantity, item.price);
    }
}

Report getReport() {
    ArrayList<Product> all = new ArrayList<>();
    float totalRevenue = 0;
    float stockValue = 0;
```

```
        for (Product p = Product.firstProduct; p != null; p = p.nextProduct) {
            totalRevenue += p.revenue;
            stockValue += p.nInStock * p.price;
            all.add(p);
        }
        Product[ ] topTen
            = all.stream().sorted((p1, p2) -> Float.compare(p2.revenue, p1.revenue))
                          .limit(10)
                          .toArray(Product[ ]::new);
        return new Report(totalRevenue, stockValue, topTen);
    }
```

Dealing with Persistence

That's all very simple, but it does have some drawbacks. First of all, all of the data is sitting on the Java heap of our process. If the power goes out or if the process crashes, it all goes away. To avoid this, we need to make sure that the data is stored somewhere persistent, which typically means either a file system or a database.

If we opt for using files, we'll need a representation for our products that can be read from and written to disk. A straightforward approach is to use a comma-separated values (CSV) file, where each line in the file represents one product and contains (a string representation of) the product's associated data, separated by commas. Establishing the inventory becomes something like

```
Files.lines(csvFilePath)
        .forEach(line -> {
                    String[] fields = line.split(",");
                    String sku = fields[0];
                    int nInStock = Integer.parseInt(fields[1]);
                    int nSold = Integer.parseInt(fields[2]);
                    float price = Float.parseFloat(fields[3]);
                    float revenue = Float.parseFloat(fields[4]);
                    Product p = new Product(sku, price, nInStock);
                    p.nSold = nSold;
                    p.revenue = revenue;
                });
```

Except that we should probably add error checking just in case there's an I/O problem or one of the lines has the wrong number of fields or one of the fields doesn't parse as a number or the same SKU is in there twice or ... And we'd better be sure that the fields we refer to are in the same order that they were when they were written. There's a lot that can go wrong, and we have to be prepared for any of it.

In addition to reading the file, we have to write it, to make sure that it's there to be read next time. We could do this during each action (making sure that we are certain that the new file has been safely written before we acknowledge that the action has been performed. But this is very expensive. So what we'll probably do is write the file out relatively infrequently and for each action, we'll append to a "journal file", which just describes the actions performed. This is more efficient (but still costly, as we have to be sure that the journal entry has made it to disk before acknowledging each action), but it means that when we read the initial state, we have to not only read the CSV file, but also the journal, and we have to replay all of the journal entries that were written since the CSV file was written, taking care that we don't perform any externally-visible

side effects (like journaling!) as we do so. And with either solution, we have to be aware that writing any of these files can fail at any point, so we have to be prepared to recover from half-written files or journal entries.

The alternative is to use a database. This has advantages—persistence is now somebody else's problem—but it also has drawbacks. Using a database typically means writing code to talk to it in a completely different language (SQL) which is very different from the Java the programmer is used to. It also means that somebody knowledgeable needs to set up the schemas for the database tables that the program will use. And typically every interaction with the database involves communicating with another process, often on another machine, over a socket, with all of the expense that implies.

Dealing with Concurrency and Sharing

The second problem is that with the approach detailed so far, we can only safely handle one action at a time. To see why, consider what would happen if two threads each tried to process a sale of ten units of some SKU for which there are 15 in stock. It's perfectly possible for each to get through the first loop and determine that yes, in fact there are at least ten and then for both to subtract ten from the stock on hand, leading to a final stock of negative five, which is exactly what the initial check was supposed to prevent. So the program will need to make use of locks or critical sections (e.g., by use of the **synchronized** keyword) to ensure that only one thread is performing an action at a time. With careful planning, it may be possible to allow partial overlap of actions, but getting that right can be quite tricky. In any case, writing to the journal files will also need to be synchronized to ensure that the files themselves don't wind up with garbage lines containing partial lines from different threads.

And even if the appropriate synchronization methods are used to allow multiple threads to access the data at the same time, it won't be possible to actually allow multiple processes on the computer to access the data at the same time, because the actual data structures will be within the Java heap of a single process.

It's at this point that most programmers throw up their hands and either decide that everything will be done within a single process or that all of the actual work will be done by using a database, with little or no local data stored on the process's Java heap. This can work, as in addition to persistence, database provide a notion of *transactions* that allow a thread to perform a number of actions and be guaranteed that either it is possible to pretend that all of the actions happened at a single instant in time or nobody else can see any of the effects of the actions and the thread can try again. Programs written in this way tend to look very different from those just working in local data structures.

And databases tend to not be good at dealing with long-running computations that want to see a consistent version of the data. So the program will typically arrange for the database to make periodic snapshots, holding off all other transactions while it does so, with all long-running analytics operations working on that snapshot data. This means that it becomes impossible to ask questions about the current state of the database as opposed to the state of the database when the last snapshot was taken, and because

snapshotting is typically an expensive operation, it tends to be performed infrequently and at odd hours.

Doing it with MDS

With MDS, we can get much closer to the original in-memory solution. There are some differences, but they should be easy for a Java programmer to wrap their head around.

The MDS analogue of a Java (class) object is a *managed record*. Like a Java object, managed records have associated data and methods, and like Java classes, managed record types can be defined to form an inheritance hierarchy. Unlike Java objects, managed records are stored in a persistent *managed space*, which is shared between multiple processes. Because of this difference, there are a few minor differences between the way you use Java objects and the way you use managed records, notably:

- Rather than saying

  ```
  new T(args…)
  ```

  which will create a Java object on the process's Java heap, you say

  ```
  T.create.record(args…)
  ```

  which creates a managed record in the managed space.[1]

- You can't access fields directly. That is, you can't say

  ```
  foo.a += bar.b;
  ```

  Instead, you need to go through getter and setter methods, as is common in Java:

  ```
  foo.incA(bar.getB());
  ```

- You can't trust lack of identity. That is, just because `a == b` is **false** doesn't mean that `a` and `b` don't refer to the same managed record.[2] (The converse, however, is valid. If `a == b` is **true**, the two variables do refer to the same managed object.) If you want to be sure, use `a.isSameAs(b)`, which is also the default definition of `a.equals(b)`.

- You can't simply define a managed record type by defining a Java class (at least, not without a bit of work). The easy way to define a managed record type

---

[1] Yes, that looks a little weird. As with most bits of weirdness, especially around managed records, it was the least weird way we could come up with to get the Java compiler to do what we need it to do.

[2] This applies to all other managed object types as well.

is by defining a Java *interface* annotated with the *@RecordSchema* annotation, as will be described shortly.

So how do we declare our managed record type in MDS? As mentioned, everything will take place inside a *schema interface*:

```
@RecordSchema
interface ProductSchema {
    @Final ManagedString sku();
    int nInStock();
    int nSold();
    float price();
    float revenue();
    @Final Product nextProduct();

    static final Namespace dataNS = Namespace.fromPath("/MyCo/data");
    @Private static final Namespace bySKU = dataNS.resolve("SKUs");
    static final String firstProductKey = "firstProduct";

    static void Product(Product.Constructing self, String sku, float price, int nInStock)
    {
        self.setSku(sku);
        self.setPrice(price);
        self.setNInStock(nInStock);
        isolated( () -> {
           self.setNextProduct(dataNS.lookup(firstProductKey, Product.TYPE));
           self.bindName(dataNS, firstProductKey);
        });
        self.bindName(bySKU, sku);
    }

    static void check(Product.Private self, int q, float p)
            throws WrongPrice, InsufficientQuantity
    {
        float price = self.getPrice();
        if (price != p) { throw new WrongPrice(self.getSku(), p, price); }
        int nInStock = self.getNInStock();
        if (nInStock < q) { throw new InsufficientQuantity(self.getSku(), q, nInStock); }
    }

    static void sell(Product.Private self, int q, float p) {
        self.incRevenue(p*q);
        self.incNSold(q);
        self.decNInStock(q);
    }

    @Static
    static boolean exists(String sku) {
        return bySKU.isBound(sku);
    }

    @Static
    static Product lookup(String sku) throws NoSuchSKU {
        Product p = bySKU.lookup(sku, Product.TYPE);
        if (p == null) {  throw new NoSuchSKU(sku); }
        return p;
    }
};
```

I need to have a note on why it is (or at least soon will be) okay to have a set of a final field inside of an isolated block that may get redone several times.

Declaring the Record Schema

The record schema interface does the heavy lifting of defining the actual record interface. Let's go over it step by step.

```
@RecordSchema
interface ProductSchema {
    …
}
```

Every record schema is a Java interface (not a class) that is annotated by the *@RecordSchema* annotation and whose name is canonically the name of the managed record type with "Schema" appended. So in this case, the record type is to be named *Product*, so the schema interface is *ProductSchema*.[3]

You may be wondering why we chose to do it this way. It turns out that there is a fair amount of bookkeeping that's required to set up a managed record type, and we wanted to shield the programmer from it. (If you want to do it yourself, you can, and we explain below in "Managed Records: the Hard(er) Way".) It also turns out that in Java, you can add an *annotation processor* to process program annotations as the program is being compiled, and these annotation processors can generate code. Unfortunately, while they can *emit* code, there is no (portable, official) way to *modify* code, so while it's straightforward to emit a *Product* interface from an annotated *ProductSchema* interface, we couldn't simply add the extra stuff to an annotated *Product* interface.

As alluded to in the prior paragram, merely by defining the *ProductSchema* interface, a *Product* interface will magically appear, and this is what your code will use. No instances of classes implementing *ProductSchema* will ever be created.

Managed Record Inheritence

If the managed record type is a subtype of another managed record type, this is indicated by having the schema interfaces inherit. That is, if there were several different types of products, you might say

```
@RecordSchema
interface SoftwareProductSchema extends ProductSchema
```

[3] Note that in this short tutorial, we will mostly be describing the default behavior. There are many aspects that can be customized by means of annotation parameters and further annotations. See the "Managed Records: the Easy Way" reference section below for more detail on how to control what the *@RecordSchema* annotation processor emits.

With that, all *SoftwareProduct* managed records would be considered to be *Product* records, and would have the same fields (plus whatever fields were added for the subtype). MDS currently only supports single inheritance for record types.[4]

Other Implemented Interfaces

If a record schema extends a non-schema interface, the emitted record interfaces will do the same. So, for example, if *Product*s are ordered (say, by revenue), you might declare

```
@RecordSchema
interface ProductSchema extends Comparable<Product>
```

In this case, the schema interface must either declare a schema for the

```
int compareTo(Product)
```

method required by *Comparable* or be declared *@Abstract* (see below).

Declaring Fields

Next in the record schema, we define declare the fields that the managed record will have:

```
@Final ManagedString sku();
int nInStock();
int nSold();
float price();
float revenue();
@Final Product nextProduct();
```

Since Java does not allow interfaces to have data members, fields of managed records are declared as no-argument methods. Note that we can declare a field to be of type *Product* even though this is the type that will be emitted based on this schema.

Declaring Final Fields

Two of the fields, sku and nextProduct carry the *@Final* annotation. This is the equivalent of the **final** keyword for a field in a Java class. Fields annotated with *@Final* can only be set in the record's constructor and can only be set once. (The emitted code ensures this.) If a final field is not set by the end of the constructor, it is considered to be implicitly set to zero, **null**, or **false**, as appropriate.

Emitted Getters, Setters, and Modifiers

The emitted *Product* interface will also declare getter and setter methods for each of the fields, and for fields whose type is numeric, it will also declare modifier fields. So for the price field, *Product* will contain

```
float getPrice();
float setPrice(float val);
float incPrice(float delta);
float incPrice();
float decPrice(float delta);
float decPrice();
```

---

[4] If you extend multiple schema interfaces, the emitted code will compile, but you will get a runtime error when your program tries to register your type with the MDS runtime.

```
float mulPrice(float delta);
float divPrice(float delta);
```

Setters and modifiers return the value replaced.

Following the Java convention, by default, the getter for a **boolean** field uses *"is"* rather than *"get"*.

If you declare an instance method schema (see below) that would result in the same signature as one of the emitted accessor methods, the default method will not be emitted.

Field and Method Visibility

By default, all emitted accessors are considered to be public, so anybody who has a *Product* reference p, can call p.getPrice(). If the price field schema had been annotated with the *@Private* annotation:

```
@Private int price();
```

then the accessors would only be callable if p had been of type *Product.Private*. As you might expect, there are also *@Protected* and *@Public* annotations and a *Product.Protected* interface. A *Product.Private* is a *Product.Protected*, and a *Product.Protected* is a *Product*, so references with private access can also see protected and public methods. And it works as you'd expect with inheritance: if *ChildSchema* extends *ParentSchema*, a *Child.Protected* (and, therefore, a *Child.Private*) is a *Parent.Protected*, so child implementations can see protected, but not private, parts of their parents. Note that there is no *Product.Public* interface; just use *Product*.

If you need finer granularity over the visibility levels for the various emitted accessors, you can specify annotations that specify what they control. So you could say, for example

```
@Private
@Public(Emitted.GETTER)
@Protected(Emitted.SETTER)
@No(Emitted.MODIFIERS)
@Private(Emitted.INC)
int price();
```

This specifies that price is private, but getPrice() is public, setPrice() is protected, and no modifiers are emitted other than incPrice(), which is private. The arguments to the methods are members of the *Emitted* enumerated type, and include **GETTER**, **SETTER**, **INC**, **DEC**, **MUL**, **DIV**, **ACCESSORS** (all of the preceding), and **MODIFIERS** (all accessors except for getters and setters). The order of the annotations doesn't matter, but those with narrower scope have priority over those with wider scope.

Managed Strings

Aside from specifying fields as methods, the only other difference in the example field specifications from the non-MDS code is that the sku field is declared to be a *ManagedString* rather than a normal Java *String*. *ManagedString*s are stored in the managed space and can therefore be shared between processes. Like Java *String*s (and

unlike C++ *string*s), *ManagedString*s are immutable. The MDS library ensures that there is only a single copy of a given *ManagedString* in the managed space. This extends to a guarantee that there are never two *ManagedString*s with the same content. So s1 == s2 is sufficient to determine whether two strings have the same content.

The sku field could also have been declared to be a *String*, and the only difference would be the value returned by the getter. (The setter is overloaded to take any *CharSequence*, which both *String* and *ManagedString* implement.) *String*s and *ManagedString*s can be interconverted by

```
String s = ms.asString();
ManageString ms = ManagedString.valueOf(s);
```

There's some expense in performing the conversion. As far as the MDS library is concerned, every place string arguments are required, they are specified as *CharSequence*, so either managed or unmanaged strings can be passed, and the same holds for many Java APIs, e.g., *String*.join(). So it's typically more efficient to declare fields as *ManagedString* and avoid unnecessary conversions.

Note that the first time asString() is called on a given *ManagedString*, the resulting *String* is cached in the *ManagedString*, so as long as the *ManagedString* object sticks around, further calls to asString() are cheap. This comes at a cost in space, however, as there are now two copies of the string, one in the local cache and one in the shared persistent space. When the strings are huge, this can be a problem, so if you know that the cached string is not going to be needed again, you can call

```
String s = ms.asStringNoCache();
```

to get it or

```
String s = ms.asString();
…
ms.clearCache();
```

to ensure that the cache is cleared when you're done. (Note that toString() on a *ManagedString* calls asString().)

Defining Constructors

To define a constructor for the *Product* managed record type, we define a static method in the *ProductSchema* interface that has the same name as the emitted class:

```
static void Product(Product.Constructing self, String sku, float price, int nInStock)
{
    self.setSku(sku);
    self.setPrice(price);
    self.setNInStock(nInStock);
    isolated( () -> {
       self.setNextProduct(Product.TYPE.lookupName(dataNS, firstProductKey));
       self.bindName(dataNS, firstProductKey);
    });
    self.bindName(bySKU, sku);
}
```

This static method is called to initialize a managed record after it is created but before it is put anywhere where it could be visible to any other thread or process. Before the constructor is called, it can be assumed that all fields have no value, and getters will respond with **null**, zero, or **false**, as appropriate to the type.

A managed record schema interface may have multiple constructor schemas, which will turn into multiple overloaded constructors for the omitted type. If no constructor schema is defined, a default one will be emitted that takes no arguments and does does no initialization other than delegating to the constructor schema for the managed record types supertype (if any).

*The **self** Parameter*

The first argument to a constructor schema (or any instance method schema) should be named **self**. This will contain a reference to the managed record being initialized. For a managed record named *Product*, the type of self will be *Product.Constructing* in a constructor schema and *Product.Private* in other method schemas, meaning that the constructor schema can use it to access methods and field accessors declared *@Private* and *@Protected*. (It is also acceptable for self to be declared *Product.Protected* or *Product*, but this will limit the access the constructor schema has to the object.) If **self** is declared as *Product.Constructing* in a constructor schema, it will additionally be able to initialize final fields and delegate to other record or superclass constructors.

Calling Superclass Constructors

In Java, if a class *B* extends a class *A*, and class *A* has a constructor that takes arguments, class *B*'s constructor must begin with a call to **super**(…), passing in constructor parameters. In MDS, similarly, if *A* and *B* are record types, then in the constructor schemas for *B*, the constructor schema must include a call to self.superInit(…), passing in parameters to one of *ASchema*'s constructor schemas *not including the self parameter*. Unlike Java, MDS does not require this to be the first statement in the constructor schema, but the constructor schema should not assume that the parent type's fields have been initialized until it is called.

- Note that this must be done even if the parent's constructor schema takes no paremeters.[5]
- Note also that the annotation processor (and therefore the compiler) has no way of checking this.
- Note further that the superInit() method is an instance method on **self**.
- It is okay to omit the call to **self**.superInit() if this record schema does not extend another record schema or if you are sure that no ancestor record schema declares a user-defined constructor. It's always safe to include it.

As in Java, if a child record schema extends a parent record schema, and the parent doesn't have a no-argument constructor, the child must include a constructor schema, if only to call **self**.superInit(). If the child doesn't have a user-defined constructor, a

[5] The Java compiler can tell that **super**() wasn't called and silently insert a call. We can't.

default constructor will be emitted for it, and this constructor will call **self**.superInit(), which will fail to compile.

Delgating Between Constructors

As in Java, it is often useful for one constructor to delegate to another, either to change the parameters (or to supply default parameters) or to perform some basic initialization before going on to do additional work. In Java, this is accomplished by having the first line of code in the constructor be a call to **this**(…). In an MDS constructor schema, the same effect is accomplished by calling **self**.thisInit(…).

As with superInit(), calls to thisInit() should be the first line, but we can't check. If there is a call to thisInit() there should not be a call to superInit() and vice versa, since the end of the chain of thisInit() delegation should be to a constructor that either calls superInit() or is satisfied that no such call is required.

Initializing Final Fields

In the example, the final field nextProduct is initialized in the constructor by calling

```
self.setNextProduct(…)
```

This can only happen if **self** is of type *Product.Constructing* (i.e., in the constructor schema) and can only happen once. If an attempt is made to set it a second time, a *FinalFieldModifiedException* will be thrown.[6]

Namespaces

The other bit of strangeness in the constructor schema is

```
static final Namespace dataNS = Namespace.fromPath("/MyCo/data");
@Private static final Namespace bySKU = dataNS.resolve("SKUs");
static final String firstProductKey = "firstProduct";
…

static void Product(Product.Private self, String sku, float price, int nInStock) {
    …
    isolated( () -> {
       self.setNextProduct(dataNS.lookup(firstProductKey, Product.TYPE));
       self.bindName(dataNS, firstProductKey);
    });
    self.bindName(bySKU, sku);
}
```

We'll talk about the **isolated** bit in a second, but first we'll walk through the rest. In the original non-MDS code we had

```
private static final HashMap<String, Product> bySKU = new HashMap<>;
static Product firstProduct = null;
```

---

[6] An exception to the "final fields can only be set once" rule is if the set is in a subtask or nested isolation context (see below). In such a case, if the task is re-run or the isolated block needs to try multiple times, the final field is reset, so a subsequent set counts as the first time. Note that if nested isolation contexts are used for this purpose, they **must** be done using an isolated block or the equivalent. That is, the entirety of the isolation context, including any attempts at publishing, must take place within the constructor schema.

```
…

Product(String sku, float price, int nInStock) {
    …
    nextProduct = firstProduct;
    firstProduct = this;
    bySKU.put(sku, this);
}
```

Static data fields in the schema interface become static data fields on the emitted managed record interfaces, but these data fields are local to the Java process. It's important for our example that the first *Product* in the list and the mapping from SKU to *Product* be shared among processes, so we'll make use of an MDS *namespace*.

MDS namespaces are like directories (or folders) in a file system, but instead of mapping names to directories and files, they map names to namespaces and managed objects. As with directories in a filesystem, namespaces are hierarchical, and they can be specified by strings (or *ManagedString*s) that can represent paths, which are names separated by a delimiter (by default, a slash). So in

```
static final Namespace dataNS = Namespace.fromPath("/MyCo/data");
```

we establish a particular namespace that will hold our program's data. This namespace is the one named *"data"* within the namespace named *"MyCo"* within the root namespace, which is specified by an initial slash: *"/"*. Within this namespace, we will keep our shared "static" variables. Instead of creating a *HashMap* to hold the mapping from SKU to *Product*, we will use another namespace:

```
@Private static final Namespace bySKU = dataNS.resolve("SKUs");
```

As with the non-MDS map, we declare this to be *@Private*. And the namespace we use will be the one named *"SKUs"* within the dataNS namespace we established earlier. And instead of a static *Product* to hold the first product in the list, we will use the *Product* bound to the *"firstProduct"* name in the dataNS namespace.

Within the constructor schema, instead of saying nextProduct = firstProduct, we now use the more verbose

```
self.setNextProduct(dataNS.lookup(firstProductKey, Product.TYPE));
```

The use of the setter is straightforward, but the lookup() call deserves mention. The first parameter to the call is the name to look up in the namespace, in this case *"firstProduct"*. The second parameter is the type we expect to find bound to that name. All managed types have a **TYPE** static data member that holds an object that represents that type, and emitted managed record types are no exception. So we are expecting a *Product* to be bound to *"firstProduct"*. If the name is bound to something that's not a *Product* or some managed record that extends *Product*, an *IncompatibleType–Exception* will be thrown. So if we get back a reference to a *Product* (or a **null**), we can be confident that what was bound there was indeed a *Product*.

For primitive types, there are shortcuts, e.g., lookupInt(name), lookBoolean(name), so you don't have to specify the type. You can also make the call on the type object, by calling lookupName(), so that could have been specified as

```
Product.TYPE.lookupName(dataNS, firstProductKey).
```

The lookupName() method is a bit more flexible in that it can take a path string or a list of names instead of a namespace and name. It can also take an *HName* (hierarchical name) object, which represents a path. These will be described below in the reference section on namespaces.

Alongside the lookup() method, *Namespace* also has

```
boolean isBound(name);
boolean isBoundTo(name, type);
boolean isBoundTo(name, managed object)
```

These allow you to ask whether a name in the namespace is bound to anything, bound to an object of the given type, or is bound to the given object. They always return **true** or **false** and don't throw exceptions.[7]

Once we have set the *Product*'s nextProduct to be the old first product, we bind the *"firstProduct"* name to point to the current product:

```
self.bindName(dataNS, firstProductKey);
```

All managed object types provide bindName() methods to bind the objects to names in namespaces. As with lookupName(), there are variants that take path strings, lists of names, or *HName*s. A given object can be bound to any number of names, so we also use this method to assert that this is the *Product* with this particular SKU:

```
self.bindName(bySKU, sku);
```

This binds the *Product* to sku within the bySKU namespace.

Working in Isolation

Which brings us to that weird **isolated** call:

```
isolated( () -> {
    self.setNextProduct(dataNS.lookup(firstProductKey, Product.TYPE));
    self.bindName(dataNS, firstProductKey);
});
```

The namespace is shared between threads (and processes), and if we just naively ran the code inside, there is the possibility that in between the read of the current first *Product* and setting the first *Product* to be this one, some other thread also created a *Product*. Let's say that the first *Product* we read was P1, and we're creating P2. What

---

[7] *At the moment, only isBound(name) is implemented. (The others merely call it.) That should change before anybody reads this. This is still the case as of 5/26/17. I will try to fix this shortly.*

we'd like when we're done is for the first *Product* list to start with P2, which points to P1. But suppose after we read P1, somebody else created P3 and added it onto the list. So now, when we do the bindName(dataNS, firstProductKey) call, the *Product* list actually contains P3 pointing to P1, and we replace it with P2 pointing to P1. What happened to P3? It's still there, but nobody iterating over the list of *Product*s will ever see it.

So what we need is a guarantee that the reading and writing constitute a single *atomic* action. To do this, we create a new *isolation context* and do the work in it. Within an isolation context,

1. All writes to fields of managed records and slots in managed arrays and all bindings of names in namespaces are *isolated* from code working outside of the isolation context. That is, if another thread reads the field or slot or does a lookup on the name, it won't see the effect of your write.
2. When you read or write such a location, the MDS runtime guarantees that the next time you read, what you see won't be affected by subsequent writes to the location from outside of the isolation context.
3. When you *publish* the isolation context (at the end of the **isolated** block), the system checks to see whether anybody did anything outside that would *conflict* with what you did such that it isn't possible to pretend that it all happened instantaneously.
4. If there were no such conflicts, all of your changes become visible outside the isolation context as a single atomic action, as of the time you published.
5. If there were any conflicts, the isolation context you were working in, and all of the changes you made are thrown away, a new isolation context is created, and the code is run again, looping until the publication succeeds.

There's a *lot* more to isolation contexts, but that's the simplest basic mode of using them to do short atomic transactions like the one we want here. One thing to note is that isolation contexts are hierarchical, so don't think of it as "publishing to the 'real' state of the record or namespace". We're simply publishing it to whatever isolation context the constructor schema is running in, which may itself be isolated from (some or all) other threads in the system.

In the scenario described above, we read P1, set it as P2's nextProduct, and set the first *Product* to be P2, making the *Product* list P2, P1, … But we do this in an isolation context, so nobody else can see that the first *Product* is now P2. While we're doing this, another thread sets the first *Product* to P3, so the *Product* list now reads P3, P1, … When we try to publish, assuming that the other thread successfully published its isolation context, the system disallows the publication due to the conflict at the *"firstProduct"* name. We run the code again in a new isolation context, retrieve the first *Product* as P3, set that as P2's nextProduct, and set the first *Product* to be P2, making the *Product* list P2, P3, P1, … We then attempt to publish, and since

there are no conflicts, the publication succeeds, and this becomes the new visible *Product* list.

Now a word about **isolated** itself. While it may look like a keyword as its used in this example, **isolated** is actually just a function that's overloaded to take pretty much any functional form you care to throw at it. Canonically it will be a lambda expression, as shown here, but it could also be a method reference or a functional object (or a functional object and an argument or two to pass to it). If the argument functional returns a value, for the iteration that successfully publishes, will be the value of the **isolated** form.[8]

Importing MDS static functions

In fact, **isolated** is a static function of the *IsolationContext* interface. In order to be able to write the examples the way we have done here, you have to have one of

```
import static com.hpl.mds.IsolationContext.isolated;
import static com.hpl.mds.IsolationContext.*;
```

For convenience, the keyword-like static functions are also defined in the *MDS* interface, so it suffices to include

```
import static com.hpl.mds.MDS.*;
```

to get all of them, regardless of which class they are defined in.

Defining Instance Methods

Defining instance method schemas is just like defining constructor schemas, except that you get to pick the name, and they can return a value. As with constructor schemas, instance method schemas must have a first **self** parameter that can take an argument of type *Product.Private*.

```
static void check(Product.Private self, int q, float p)
        throws WrongPrice, InsufficientQuantity
{
    float price = self.getPrice();
    if (price != p) { throw new WrongPrice(self.getSku(), p, price); }
    int nInStock = self.getNInStock();
    if (nInStock < q) { throw new InsufficientQuantity(self.getSku(), q, nInStock); }
}

static void sell(Product.Private self, int q, float p) {
    self.incRevenue(p*q);
    self.incNSold(q);
    self.decNInStock(q);
}
```

[8] Note that if tasks (described below) are re-run to resolve conflicts, the value returned does **not** change. (If the conflict's top-level task—the one that would return the value—depends on tasks that need to be re-run, the entire function will be re-run, producing a different value.) An **isolated** block that returns a value and uses subtasks should return a Java object whose state is updated by those tasks.

Except for the use of getters, setters, and modifiers, this code is essentially identical to the non-MDS code. Note, however, that it is not sharing-safe. It's possible that while check() or sell() is working, another thread is modifying the *Product*. This code is written in the expectation that the caller, who will call both, will wrap the calls in a single **isolated** call, so these methods don't bother.

Defining Static Methods

Static methods are just like instance methods, except that (1) they don't have a self parameter and (2) they carry the *@Static* annotation:

```
@Static
static boolean exists(String sku) {
    return bySKU.isBound(sku);
}

@Static
static Product lookup(String sku) throws NoSuchSKU {
    Product p = bySKU.lookup(sku, Product.TYPE);
    if (p == null) {  throw new NoSuchSKU(sku); }
    return p;
}
```

Defining Constants

As in Java, constants are declared as **static final** fields:

```
static final Namespace dataNS = Namespace.fromPath("/MyCo/data");
```

All such constants are available from the emitted record class. That is, if the above code was in *RSchema*, the value can be accessed as *R*.dataNS. Constants can carry visibility annotations (e.g., *@Protected* or *@Private*), in which case they can be accessed using access parameter objects or by *R*.classAs(self), as described below.

Accessing Private and Protected Data and Methods

Within a Java method, you have complete access to all private and protected methods of the class and all private and protected data members of any object of the class you might get your hands on. Similarly, you have access to all protected methods of any ancestor class and protected data members of objects of ancestor classes. Within MDS instance method schemas, constructor schemas, and static method schemas, you have similar access, albeit in a somewhat more cumbersome manner. In the following, *R* is the record type whose schema the method is declared in and *A* is an ancestor type of *R*.

- Within a constructor schema or instance method schema, the self parameter is of type *R.Private* or (for constructor schemas) *R.Constructing*. This object has access to all private instance methods (including accessors).
- Within any of these schemas, if a parameter is declared *R.Private* or *R.Protected*, the parameter will have the appropriate access (i.e., private or protected). Note that from the caller's point of view, they need only provide an *R*. The emitted code will do the necessary downcast. If a parameter is declared *A.Protected*, it will similarly have protected access, and the caller will similarly only need to provide an *A*.
- For any x of managed record type *X*, the expression x.objAs(self) will allow the access that the method should have. That is, if *X* is *R*, it will be *R.Private*

and if *X* is some ancestor type *A*, it will be *A.Protected*, otherwise it will simply be *X*.

- A static method schema may optionally have a first parameter named access of type *R.StaticMethod*. This can be used instead of self as the argument to objAs(). It can also be used to access (as instance methods) any static methods (private, protected, and public) in *R* and any protected or public static methods of any ancestor type *A*. It also can be used to access public, protected, or private constants.
- Private or protected static methods of *R* (or protected static methods of *A*) may be called as *R*.classAs(self).m(…) or *R*.classAs(access).m(…).
- Similarly, private or protected constants of *R* (or protected constants of *A*) may be accessed as *R*.classAs(self).c or *R*.classAs(access).c.

Using the Methods

Now that we've defined the *Product* class, let's look at the action methods that use it. Adding a product becomes

```
Product newProduct(String sku, float price, int nInStock) throws SKUExists {
    return isolated(() -> {
        if (Product.lookup (sku) != null) { throw new SKUExists(sku); }
        return Product.create.record(sku, price, nInStock);
    });
}
```

There are two differences from the non-MDS code. First, rather than calling **new** *Product*(…), we call *Product*.create.record(…). Second, we can no longer count on nobody adding a new *Product* with this SKU in between the time we look and the time we create one, so we do the check and the creation in an **isolated** block. If another *Product* with this SKU is created after we check, it will result in a conflict when we attempt to publish, and we will throw an exception on the re-run of the block.[9] Note that by doing it this way, we create an isolation context to do the check and the creation, and the constructor schema creates a second isolation context within that one to add the *Product* to the *Product* list.

Getting stock in only has to be modified to use a modifier:

```
void stockIn(String sku, int n) throws NoSuchSKU {
    Product.lookup(sku).incNInStock(n);
}
```

Modifiers are defined to be atomic themselves, so we don't have to worry that somebody changed the value of nInStock in between the time we read the old value and the time we wrote the new value.

[9] Had we done the check and the binding in the isolation block of the constructor, it would have been caught there and we wouldn't need this isolation block.

Checking the price and establishing a new one simply get modified to use accessors:

```
void newPrice(String sku, float p) throws NoSuchSKU {
    Product.lookup(sku).setPrice(p);
}

float priceCheck(String sku) throws NoSuchSKU {
    return Product.lookup(sku).getPrice();
}
```

And processing a sale is nearly unchanged, but wraps its action in an **isolation** block:

```
void processSale(List<LineItem> items)
        throws NoSuchSKU, WrongPrice, InsufficientQuantity
{
    isolated(() -> {
        for (LineItem item : items) {
            Product p = Product.lookup(item.sku);
            p.check(item.quantity, item.price);
        }
        for (LineItem item : items) {
            Product p = Product.lookup(item.sku);
            p.sell(item.quantity, item.price);
        }
    }
}
```

Note that even though the check() and sell() schemas take a self parameter, the caller of the method doesn't provide them, it just calls the method on the *Product* object. Note also that a single **isolated** block wraps both **for** loops. This means that in order for the publish of the isolation context to succeed, there need to be no conflicts that would invalidate either any of the checks or any of the modifications made during the sale of an item. It also means that if there were no such conflicts, all of the products sold are updated at the same time.

Working with Snapshots

Now we turn to our analytics routine, getReport(). Here we look at every product and generate a report that gives the total revenue, the total value of the stock on hand, and the top ten products by revenue. But we want to make sure that we give a consistent report—the information should all be correct as of a single point in time, and that point should be when we ask for the report. We could do this with an **isolated** block, as we did above, walking all of the products and computing the report and then trying to publish the isolation context, hoping that nobody had changed anything while we were computing it. But since we work for the company, we don't really hope that nothing had changed as we walked the inventory, since that would mean that we hadn't sold anything. So we'll take a different approach. What we'll do instead is compute the report in a *snapshot*:

```
Report getReport() {
    ArrayList<Product> all = new ArrayList<>;
    float totalRevenue = 0;
    float stockValue = 0;

    Product p = inReadOnlySnapshot(() -> Product.getFirstProduct());
```

```
    for (; p != null; p = p.getNextProduct) {
        totalRevenue += p.getRevenue();
        stockValue += p.getNInStock() * p.getPrice();
        all.add(p);
    }
    Product[ ] topTen
        = all.stream().sorted((p1, p2) -> Float.compare(p2.getRevenue(),
                                                        p1.getRevenue()))
                      .limit(10)
                      .toArray(Product::new);
    return Report.create.record(totalRevenue, stockValue, topTen);
}
```

(To simplify the code, from now on, we'll assume that the *Product* schema has defined

```
@Static Product getFirstProduct() {
    return Product.dataNS.lookup(Product.firstProductKey, Product.TYPE);
}
```

and probably a similar method to set it.)

If you compare this with the non-MDS version, you'll see that only three things have changed. First, the data in the *Product* records is now accessed using getters. And second, the initial definition of p is obtained via a call to **inReadOnlySnapshot**:

```
Product p = inReadOnlySnapshot(() -> Product.getFirstProduct());
```

inReadOnlySnapshot is like **isolated** in that it executes its code in a new isolation context, but this isolation context is different from the one created by **isolated** in three ways. First, it's *read-only*. That means that code running in that isolation context can't modify anything in the managed space.[10] If you try to do anything other than read, the program will throw a *ReadOnlyContextException*. Second, since the isolation context is read-only, you can't publish it. (This is not a big deal, since there can't be any modifications to publish.) This means that code is executed only once in **inReadOnlySnapshot**. And, finally, the isolation context is a *snapshot*. This means that all values read from the managed space are read *as of the time the snapshot was created*. No matter how long you work in a snapshot, the values will never change.[11]

It might be surprising that we didn't wrap **inReadOnlySnapshot** around all of the code in getReport(). That would have worked, and, in this case it would have had the same effect, but doing it the way we did allows us to explore another aspect of snapshots (and isolation contexts in general). The only code that we ran in the snapshot was the code that looked up the first *Product*. That value was returned as the value of the lambda and, therefore, the value of the **inReadOnlySnapshot** call and was assigned

---

[10] It can, of course, modify variables or objects in the process's own space.

[11] You can also get non-read-only snapshots, by calling **inSnapshot**. Execution in a non-read-only snapshot can modify things, but anything it hasn't modified is seen as of the time the snapshot was created.

to p. But that value was the first *Product in the snapshot*. When we ask for its revenue, nInStock, and price, we get the values that the product had in the snapshot. And when we ask for its nextProduct, we get another record in the same snapshot. So even though we returned from the **inReadOnlySnapshot** call, the snapshot isolation context that was created sticks around as long as there are references to it. This makes values read in snapshots things that can be kept around for a long time, and it makes it possible to have lists of values from different snapshots that can be compared with one another, for example, to discover trends.

You may be wondering about the cost of taking a snapshot. It turns out that snapshots (especially read-only snapshots) are very cheap to create in the MDS system. Nothing actually gets copied. The only overhead is that as long as a snapshot exists, the MDS system has to preserve any historical values that somebody might read in it, so it has an effect on garbage collection.

Finally, the third difference is that the report object is created in the managed space rather than on the local Java heap. This allows us to store it away in a way that it can be used later. For instance, we can create one every day and store them in a namespace, keyed by date, or we could create a list of them and walk the list to find the first time a particular product cracked the top ten. Note that the creation of the report object takes place in the current isolation context, not the snapshot, even though the top ten products it will contain are in the snapshot.

Managed Arrays

To illustrate working with managed arrays, we'll go one step further in the example, and show what the *Report* might look like as a managed record:

```
@RecordSchema
interface Report {
    @Final float totalRevenue();
    @Final float stockValue();
    @Final ManagedArray<Product> topTenProducts();

    static void Record(Record.Constructing self, float r, float s, Product[ ] top10) {
        self.setTotalRevenue(r);
        self.setStockValue(s);
        self.setTopTenProducts(Product.TYPE.createArray(top10));
    }
}
```

The interesting part here is working with the arrays:

```
@RecordSchema
interface Report {
    ...
    ManagedArray<Product> topTenProducts();

    static void Record(Record.Constructing self, float r, float s, Product[ ] top10) {
        …
        self.setTopTenProducts(Product.TYPE.createArray(top10));
    }
}
```

Managed arrays are fixed-size mutable arrays that live in the managed space. For managed arrays of primitive types, there are interfaces named *ManagedFloatArray*, *ManagedBooleanArray*, *ManagedStringArray*, etc. For other types (e.g., records, arrays), the templated *ManagedArray<T>* interface is used.[12]

To actually create a managed array of records, the easiest way to do it is as shown, by calling the createArray() method on the record type object. This method takes an array size[13] or a Java array of managed objects. For primitive arrays, call createArray() on, e.g., *ManagedInt*.**TYPE**, For primitive arrays, there is also a createArray() method that takes a Java array of primitives (e.g., **int**[]).

Given the array defined above, you can say things like

```
ManagedArray<Product> a = r.getTopTenProducts();
for (Product p : a) { … }
p = a.get(5);
a.set(7, p);
if (a.contains(p)) { … }
a,forEach(p->System.out.format("Saw: %s%n", p);
Object[ ] javaArray = a.toArray();
Product[ ] javaArray = a.toArray(existingProductArray);
Product[ ] javaArray = a.toArray(Product[ ]::new);
int s = a.size();
long s = a.longSize();
ManagedType<Product> t = a.eltType()
int sold = a.stream().mapToInt(Product::getNSold).sum();
```

As can be seen from the examples, managed arrays implement the *Iterable<T>* interface and the toArray() methods of *Collection<T>* as well as a toArray() method that takes a generator function that creates a java array given an integer (typically *T*::**new**). They also support the Java 8 streaming API, including the ability to create parallel streams. For consistency with Java arrays and collections, the size() method returns an integer, even though the actual size of the array may be bigger than can fit in an integer. To get the full size, use longSize(). If the actual size is too big to fit in an integer, size() returns -1.

The *ManagedIntArray* interface extends *ManagedArray<ManagedInt>*, which means that the objects returned by the get(index) method are *ManagedInt*s (described below in the reference section), the toArray() method returns an array of *ManagedInt*s (as an array of Objects), the iterator returned by the iterator() method iterates over *ManagedInt*s, and the stream() method returns a stream of *ManagedInt*s. To return a normal Java **int**, use getInt(index). To convert to an **int***[ ]*, call toIntArray(). And to get an iterator of *Integer*, call intElements(). The toArray(array) method can take an

---

[12] Managed arrays of managed arrays are currently not implemented.

[13] Unlike Java, the size specified for the array is a **long** rather than an **int**, as we expect to handle arrays with more than two billion elements.

array of *E*, where *E* can be, e.g., *ManagedInt* (or a supertype), *Integer* (or a supertype) or **int**.[14] The forEach() method can take an *IntConsumer*. The intStream() method returns a Java *IntStream*.

For arrays of other primitive types, substitute *"String"*, *"Boolean"*, etc., for *"Int"*. Note that Java only supports *IntStream*, *LongStream*, and *DoubleStream* as primitive specializations, so only those managed arrays support the primitive stream methods.

For arrays of numbers, there are also modifiers. a.inc(9, 1) increments the value at element 9 by 1, while a.mult(5, 2) doubles the value of element 5. a.incAll(1) increments all elements of the array by one. Note that while modifying a single element of the array is atomic, modifying all elements of the array is not. If you need it to be, wrap it in an **isolated** block.

Resolving Conflicts

Now suppose that we're told that every product has to have a score associated with it. The score is a floating point number, and you need to write a function that will be passed a function for computing the score for a given product and will update the scores of all the products. The updates have to be atomic. That is, none of the products will have updated scores until they all do.

Adding the score itself is straightforward:

```
@RecordSchema
interface ProductSchema {
    …
    float score();
    …
}
```

And writing the function and making it atomic doesn't require anything we haven't seen before:

```
void computeScores(Function<? super Product, Float> scoreFn) {
    isolated( () -> {
        for (Product p = Product.getFirstProduct(); p != null; p = p.getNextProduct) {
            p.setScore(scoreFn.apply(p));
        }
    }
}
```

We create a new isolation context, look up the first product, and walk the products. For each one, we call the provided function to get the score and set the score on the product. And then at the end, we try to publishe the changes in the isolation context. If there are no conflicts—if nobody else changed any value that the score function read—all of the

[14] If *E* is a supertype of both *ManagedInt* and *Integer* (e.g., *Object*), what it gets filled with is *ManagedInt*s.

scores are updated atomically. If there were conflicts, we go back to the beginning and try again.

But therein lies the problem. We're touching *every* product, and the score computation could take a non-trivial amount of time. Unless we take care to only run this at a time when nobody is selling anything, there's a very good chance that by the time we get to the end, somebody will have sold something and changed one of the products, leading to a conflict. And if we go back and try again, we're just as unlikely to get through conflict-free. So what can we do?

Enter *tasks*. Tasks are bits of code that could be re-run if conflicts are found. When an attempt is made to publish an isolation context and conflicts are found, the system looks to see if the conflicts can be *resolved* by re-running some of the tasks that were run during the computation. The basic rule for deciding which tasks to re-run are:

1. If a task read some value (e.g., a field of a record object, a name in a namespace, or an element of an array) that had been written outside of the task's isolation context and that was later changed outside of the task's isolation context, the task is selected to be re-run.
    a. Note that if the task's isolation context is a snapshot isolation context, the notion of "later" includes "after the snapshot time but before the read occurred".
2. If a task read a value that was written by a task that was selected to be re-run, the reading task also is selected to be re-run.

There are a few other cases in which a task might be selected to be re-run, but that's the main idea. For other tasks, the presumption is that if they were re-run they would behave the way they did the first, so there's no point in actually re-running them. The goal is that if the program is partitioned into tasks, the system can chose a subset of the tasks to re-run, and this subset can be substantially smaller than the complete job.

To add tasks to our code, we simply call the work within a **asTask** function:

```
void computeScores(Function<? super Product, Float> scoreFn) {
    isolated( () -> {
        for (Product p = Product.getFirstProduct(); p != null; p = p.getNextProduct) {
            asTask( () -> {
                p.setScore(scoreFn.apply(p));
            }
        }
    }
}
```

As with **isolated**, **asTask** is a static function[15] that takes a function to run, although unlike **isolated**, **asTask** never returns a value.[16]

Now, when we complete iterating over our thousands of products and try to publish and the attempt fails because, say, twenty of them changed their values due to sales or price changes, only the twenty tasks that worked on those products are re-run. Re-running the tasks means re-running the functions, and that re-running takes place in the same, not-yet-published isolation context. But before any tasks are re-run, any writes made by taks to be re-run are *rolled forward*. This leaves the locations written with the value last written by a task not selected to be re-run or, if there were no such writes, to the current value in the parent isolation context. The result of this is that that when tasks are re-run, what they see is what the world would have looked like had all non-selected tasks been re-run at the time of the (failed) publication attempt. When the identified tasks have been re-run, the publication is attempted again. At this point, there may be new conflicts, but since the work took less time, there will probably be fewer of them and, therefore, fewer tasks that need to be re-run. (Note that the tasks that need to be re-run the second time might include some that didn't need to be re-run the first time.) The expectation is that after a small number of iterations, the amount of re-work needed should be small enough that there's a good chance of being able to get through without conflicts.

When tasks are re-run, the order in which this happens depends on the order in which the tasks ran initially. In particular, tasks which started first are re-run first.[17]

Conflicts Outside of (Explicit) Tasks

In the code we just wrote, not all of the work in the isolation context took place within a task we explicitly created. In particular, looking up the first product and finding the next product for each product takes place outside of explicit tasks. In our example system, the next product will never change (since it is declared *@Final*), but it's

---

[15] And, like **isolated**, it requires an **import static** declaration to use it without mentioning the class:

```
import static com.hpl.mds.Task.asTask;
```

As with **isolated**, the **asTask** function is also declared in *com.hpl.mds.MDS*.

[16] The logic for this restriction is that if the task needs to be re-run during conflict resolution, it might produce a different result, but the Java code that used the original value won't be re-run. To handle the situation in which a task should compute a value that will be used elsewhere, use *TaskComputed<T>* objects and the *Task*.computedValue(fn) function, which will be described below.

[17] In the current MDS library, task re-run happens in a single thread. This clearly needs to change, as it may lead to cases in which task A was selected to be re-run because it read a value written by task B, but the two tasks ran in parallel and task A happened to start before task B and will consequently be re-run in its entirety before task B.

definitely possible that somebody might have added a product, which will result in the first product changing. This will result in conflict on the initial read of the first product from the namespace. While the read is in a task (there is always a prevailing task), the task isn't associated with a function that the MDS library can re-run, so the only alternative the system has is to re-run the entire isolation context work. The MDS team will be looking at ways of minimizing such situations.

Working with Arrays and Tasks

One of the most common uses of tasks is to iterate over arrays and perform some computation on each element, each in its own task. To simplify this, *ManagedArray<T>* supports

```
void forEachInTasks(Consumer<? super T> fn);
void forEachInParallelTasks(Consumer<? super T> fn);
```

These walk over the array, creating a new task for each element and applying the given function to each. forEachInParallelTasks() uses the Java parallelStream() mechanism to attempt to do this in parallel. Both iteration methods guarantee that the created tasks will be children of the current prevailing task. Importantly, the actual read of the array element takes place inside of the created task, so that if the array element is changed, the task will get a conflict.[18]

Note that for arrays of primitive objects, such as *ManageIntArray*, the element type *T* is *ManagedInt*, and forEachInTasks() consequently takes a *ManagedInt* consumer. If you want to pass in a consumer of *Integer*, call one of

```
void forEachIntInTasks(Consumer<? super Integer> fn);
void forEachIntInParallelTasks(Consumer<? super Integer> fn);
```

instead. Alternatively, an *IntConsumer* can be passed in to any of the named iteration functions.[19]

In our example, if the *Products* had been stored in *ManagedArray<Product>* rather than a linked list, we could have simply said

```
void computeScores(Function<? super Product, Float> scoreFn) {
    isolated( () -> {
        Product.products().forEachInTasks( p -> p.setScore(scoreFn.apply(p)));
    }
}
```

[18] Assuming that the value read wasn't written by another task in the same isolation context, in which case the reading task will simply become dependent on the writing task.

[19] Java doesn't allow a method to overload on both *Consumer<? extends ManagedInt>* and *Consumer<? extends Integer>*. They think their reasons for this are good. They are wrong.

Computing Values in Tasks

At times, you may want to compute a value in a task such that whenever the task needs to be re-run, any task that read the computed value will also be selected to be re-run. To do this, you can use *Task*.computedValue(fn). This creates a new task as a child of the current task and computes the value based on the provided function, but what it returns is a *TaskComputed<T>* object (where *T* is the value returned by the function, which doesn't have to be a managed type). This object provides a get() method and guarantees that until the current isolation context is published, any task calling the get() method will depend on the task that computed the value.

The function provided to *Task*.computedValue() can either be a *Supplier<T>* or a *Function*<? **super** *T*, ? **extends** *T*>. In the latter case, the function is called with an argument of **null** the first time the task is run and the old value on each subsequent time. This allows the computed value to undo local state from the prior computation before recomputing.

Managing Tasks

In addition to what's been described above, there are a number of other things you can do with tasks:

```
Task t = Task.current();
t.dependsOn(task1, task2, …);
t.cannotRedo();
t.alwaysRedo();
t.onPrepareForRedo(fn);
```

current() gets the prevailing task. dependsOn(task) says that if any of the other tasks are selected to be re-run, this task should be as well. cannotRedo() says that this task cannot be re-run. If it is ever selected to be re-run, the conflict resolution will be determined to have failed. alwaysRedo(), conversely says that if *any* tasks in this isolation context are selected to be re-run, this one should be as well. It is mostly to be used when the task depends on something external to the MDS library, such as a sensor reading.

onPrepareForRedo() supplies a function to be run when the task has been selected to be re-run. A task can have any number of such functions, which are run in the order they are added, with the function for different tasks run in the order the tasks will be re-run. The functions are all called *before* any locations are rolled forward. Each function can optionally take the task as a parameter and can optionally return a **boolean** value. If it returns a **false** value, the entire conflict resolution is abandoned. This is intended to be used to clean up any local data structures that might have been modified during the computation.

As an alternative to calling these methods directly, they can be specified as options on the asTask() call. In such cases, they are specified by static methods in Options, as

```
asTask(Options.dependsOn(taskSet).onPrepareForRedo(undoStuff), ()-> {
    …
});
```

The options can be chained, as demonstrated above.

Snapshots as an Alternative to Conflict Resolution

For our current computeScore() example, there's another way to solve the problem in MDS. Instead of reading live values within the computation of the score, we can read them from a snapshot created for the purpose:

```
void computeScores(Function<? super Product, Float> scoreFn) {
    isolated( () -> {
        IsolationContext snapshot = IsolationContext.readOnlySnapshotFromCurrent();
        for (Product p = Product.getFirstProduct(); p != null; p = p.getNextProduct) {
            p.setScore(scoreFn.apply(snapshot.viewOf(p)));
        }
    }
}
```

By creating a snapshot and calling viewOf(p) to get the value of p in that snapshot, what gets passed in to the score function is the product as it was seen in the snapshot.[20] This means that (aside from the initial read) there is nothing here that can be conflicted, so publishing the isolation context should be straightforward.

Working with Accumulators

In the *Report* example above, we used a snapshot to compute (among other things) the total revenue and stock value:

```
Product p = inReadOnlySnapshot(() -> Product.getFirstProduct());

for (; p != null; p = p.getNextProduct) {
    totalRevenue += p.getRevenue();
    stockValue += p.getNInStock() * p.getPrice();
    …
}
```

This gives us a consistent value, but it's the value as of the time the snapshot was taken (i.e., the beginning of the computation). What if we had some *Store* object and wanted to update the revenue and stock totals atomically, so that the values would not only be consistent with one another but also be correct as of the time they are set.

We could try doing the computation within a snapshot **isolated** block:

```
isolated(Options.snapshot(), ()-> {
    store.setTotalRevenue(0);
    store.setStockValue(0);
    for (Product p = Product.getFirstProduct(); p != null; p = p.getNextProduct) {
        store.incTotalRevenue(p.getRevenue());
        store.incStockValue(p.getNInStock() * p.getPrice());
    }
});
```

This would work, but if any products changed revenue, stock totals, or price as we were computing, we would have to do the whole thing again.

---

[20] *viewOf() isn't actually implemented at the moment. It should be straightforward. Either it will be in there or I'll take out this section.*

We could use tasks to limit the amount of rework that needed to be done:

```
isolated(Options.snapshot(), ()-> {
    store.setTotalRevenue(0);
    store.setStockValue(0);
    for (Product p = Product.getFirstProduct(); p != null; p = p.getNextProduct) {
        Product finalP = p;
        asTask(() -> {
           store.incTotalRevenue(finalP.getRevenue());
           store.incStockValue(finalP.getNInStock() * finalP.getPrice());
        });
    }
});
```

At first glance, this looks as though it should do what we want. Only tasks that depend on values that have been changed will be re-run. But those increments are problematic. Each one is, logically, a read followed by a write. So if the 20th product's revenue changes, its task will need to be re-run. But its increment wrote a value that was read by the 21st task, whose increment was read by the 22nd task, and so on, so all tasks after the first with a change need to be re-run.

The way to deal with such situations is with accumulator. An accumulator is a process-local object that you can add to and read from, but which has the following properties:

- A task that reads from an accumulator is dependent on all tasks that added to it.
- Tasks that add to the accumulator are not dependent on one another.

If a task that adds needs to be re-run, its original contribution is removed from the accumulator before doing so.

With accumulators, the code looks like

```
isolated(Options.snapshot(), ()-> {
    Accumulator.ForDouble totalRevenue = new Accumulator.ForDouble();
    Accumulator.ForDouble stockValue = new Accumulator.ForDouble();
    for (Product p = Product.getFirstProduct(); p != null; p = p.getNextProduct) {
        Product finalP = p;
        asTask(() -> {
           totalRevenue.add(finalP.getRevenue());
           stockValue.add(finalP.getNInStock() * finalP.getPrice());
        });
    }
    asTask(() -> {
        store.setTotalRevenue(totalRevenue.get());
        store.setStockValue(stockValue.get());
    });
});
```

Now, if there are n products and k of them change during execution, only those k tasks, plus the one at the end that sets the values, need to be re-run.

SummaryStatistics

One useful accumulator is the *SummaryStatistics* class. This has two methods for adding values:

```
add(double val)
add(double val, long weight)
```

and a number of questions that can be asked:

```
long count()
double sum()
double mean()
double variance()
double stdDeviation()
double stdError()
double high95()
double low95()
```

The last two are the upper and lower limits of the 95% confidence interval around the mean.[21]

Preparing to Run

There are a few other bits of bookkeeping that you should also know about.

Dealing with Record Inheritance

As mentioned above, record types can form an inheritance hierarchy, with, e.g., a *SoftwareProduct* record extending the *Product* record. In addition to extra fields, *SoftwareProduct* can add instance methods or override instance methods of *Product*. If the program has a reference to a record that is actually a software product record, but the reference considers it to be a *Product*, the overridden method will be called.

But there is a wrinkle: *if the runtime doesn't know about the subtype yet, the supertype will be used*. That is, if nothing has caused the system to become aware of *SoftwareProduct* (and, specifically, *SoftwareProduct*.**TYPE**), what will be constructed will be an instance of *Product*, and it won't have the *SoftwareProduct* specializations. So what lets the system know about the subtype? Two things: creating an instance of the type (or a type derived from it) and mentioning the type object (or the type object of a type derived from it). The easiest way to ensure that the types you need are known to the system is to call *MDS*.ensureKnown(type,…), as in

```
MDS.ensureKnown(
    SoftwareProduct.TYPE,
    HardwareProduct.TYPE,
    SomeOtherRecord.TYPE
);
```

This can be done in a static block in the class that contains the main() function or in the main() function itself (or something it calls).

[21] I haven't yet figured out how to efficiently do max() and min() with accumulators, but I hope to.

Building and Running MDS Programs

Building and running MDS programs is straightforward. MDS programs can be built using a development environment such as Eclipse or from the command line.

Things you will need:

- A Java 8 JRE (or JDK)
- The MDS Java API jar, **mds-java-api.jar**.
- The MDS Annotations Processor jar, **mds-annotations-processor.jar**.
  - Only needed if you will be developing code that uses record schemas.
- The MDS JNI shared library: **libmds-jni.so**.
- The MPGC **createheap** binary.

Building MDS from the Command Line Using Ant

MDS programs can be built from the command line using **Apache ant** by supplying an ant build file like the **build.xml** file in the MDS distribution's **test** directory.

Setting Up External Dependencies

The supplied build file assumes that there are several libraries in (or symbolically linked to from) a directory named **libs** in the same directory as **build.xml**:

- **mds-java-api.jar**: the MDS Java API
- **mds-annotations-processor.jar**: the MDS Java API annotation processor
- **log4j-1.2.15.jar**: The Apache **Log4J** library.

Building the Code

The build.xml file further assumes that the source code for your project is in the **src** directory. Once this is ready, simply run

```
ant build
```

This will actually compile the project twice. The first pass ("build-records") ensures that the annotation processor has generated all of the record class source files from the record schemas. The second pass ("build-project") is necessary because recursive dependencies between record classes may mean that some generated classes will be referred to before their code is generated. A consequence of this is that the first pass is likely to generate errors, which will be ignored and which are not printed by default. To see errors during this phase, define the environment variable $PRINT_BUILD_-RECORD_ERRORS.

All generated code is emitted to the **generated-src** directory, which will be created as needed. Java .class files are generated to the **bin** directory, which is created if needed.

Occasionally, even this two-pass approach is insufficient to make sure that everything is correctly generated. When this is the case, the easiest fix is to run

```
ant clean
```

and then build again. This removes the generated-src and bin directories, which ensures that all emitted code is regenerated.

Running MDS Programs from the Command Line

The shell script **test/run/setup-run** demonstrates how to set up the CLASSPATH environment variable for running an MDS program. In addition to assuming that the various jars are in the project's lib directory and its source code is in the project's bin directory, it also assumes that **lib/native** contains or links to

- **libmds-jni.so**: the shell library containing MDS Java API JNI code

With the definitions in this file, MDS code can be run by executing

```
$JCMD my.package.MainClass args…
```

Creating the Managed Space

Before running the program, however, you need to create the shared "managed space" for the MDS data structures to be created in. This is done by running the MPGC **createheap** program, e.g.,

```
createheap 10GB
```

The size of the heap is specified as a number followed by an optional case-insensitive unit, one of KB, MB, GB, or TB (the B is optional). The number may include a decimal point, so, e.g., 1.5 TB is valid. If no unit is specified, GB is assumed. Units are binary multipliers (i.e., 1.5TB is 1.5×2^{40} bytes).

This program will create two files, by default in a directory called **heaps**. The files are

- **gc_heap**: the MPGC garbage-collected heap
- **managed_heap**: a "control heap" for MPGC.

The size of the control heap is computed automatically but can be specified explicitly by the **-s** (or **--ctrl-size**) argument, which takes as a parameter the size of the control heap. This will not typically need to be changed, but may need to be raised in some circumstances.

The locations of the heap files themselves can be specified by **-f** (**--heap-path**) for the managed heap and **-c** (**--ctrl-path**) for the control heap. Note that these are the full paths to the files, not merely the names within the heaps directory.

Specifying the Heap Files

The MDS runtime (actually, the underlying MPGC garbage collector) uses three optional environment variables to specify where to look to find the heap files:

- **MPGC_HEAPS_DIR** specifies the directory to use to look for the files. Defaults to heaps.
- **MPGC_GC_HEAP** specifies the MPGC heap. Defaults to gc_heap in $(MPGC_HEAPS_DR)

- **MPGC_CONTROL_HEAP** specifies the MPGC control heap. Defaults to managed_heap in $(MPGC_HEAPS_DIR).

If the two heap files cannot be found (i.e., if createheap was not run or the environment variables do not point to the files), an error message is printed and the program terminates.

Building MDS Programs Using Eclipse

This section assumes that you are building in the Eclipse environment, version 4.4 (Luna) or later.

Note: I haven't tested these eclipse instructions

Specifying Java Compliance and JDK

To tell Eclipse to use Java 8, if it is not the default in the Eclipse installation, open the project properties and choose **1.8** under Java **Compiler/Compiler compliance level**.

To tell Eclipse to use a particular Java 8 JRE, open the project peroperties and double click **JRE System Library** uner **Java Build Path/Libraries**. Specify the JRE or JDK.

Specifying the MDS Jar

To add the MDS jar to the project, open the project properties, go to the **Libraries** tab under **Java Build Path**, and choose **Add JARs…** Navigate to and select **mds-java-api.jar**. If the jar file is not in the Eclipse workspace, choose **Add External JARs…** instead.

Specifying the Annotations Processor

To enable the processing of record schemas, Eclipse must be told about the annotations processor jar. To specify this, open the project properties and navigate to **Java Compiler/Annotations Processing**. Make sure that both **Enable annotation processing** and **Enable processing in editor** are selected.[22] Under **Generated source directory** type the name of a folder to put the generated code in. It's probably best if this is not the same as your primary source directory, to remove the chance that a badly chosen name will clobber one of your source files. A good choice is something like "generated" or "generated-src". This directory will show up as a source folder inyour project.

Finally, navigate to **Factory Path** under **Annotations Processing** and use **Add JARs…** (or **Add External JARs…**) to add both **mds-java-api.jar** and **mds-annotations-processor.jar**.

Dealing with Failure to Update

Occasionally, the code generated by the annotations processor may appear to be out of date with respect to your schemas. This is a known bug and tends to happen when there are at least three record schemas that all refer to one another's generated record classes. When this happens, simply go to one of the schema files and make a trivial change (e.g.,

[22] You can actually do without the latter, but the consequences of edits won't show up immediately and you may not get the proper error messages and warnings when editing record schema interfaces.

adding and removing a space) and save the file. This should trigger a new compilation, which will generate the correct code.

Running Programs Using Eclipse

In addition to the jars, the system needs to be told about the shared library that contains the JNI code that plugs the MDS jar into the MDS runtime. If you are going to run within Eclipse, select the project properties and navigate to **Java Build Path** and select the **Libraries** tab. Click on the triangle next to **mds-java-api.jar**, click **Native library location**, and select **Edit…** Select or type the folder that contains **libmds-jni.so**. Note that this is the folder, not the file itself.

To run from the command line, your classpath (as specified by **–classpath**) must include **mds-java-api.jar** (including any path to find it from the current directory) and the library path (as specified by **–Djava.library.path=**). So if your jar is kept in a "jars" subdirectory (relative to the current directory) and the shared library is in a "libs" directory, you would say

```
java –classpath jars/mds-java-api.jar –Djava.library.path=libs MainClass arg1 arg2 …
```

If you have more jars or more library directories, separate them with colons. If any have spaces, they must be enclosed in quotation marks.