Polymorphism and Interfaces in Mathematica

Brian Beckman
21 December 2013

Introduction

Imagine the classic C# Enumerable API, leaving out *Reset*, and implement it for *Mathematica* Lists and *Mathematica* hash tables, which are lists of replacement rules subject to *Dispatch*. This serves as an illustration of a way to implement polymorphic interfaces, that is, interfaces implemented differently by different concrete providers.

Then step via the now-classic duality argument [1] to interfaces for Observable and Observer.

References

1. Enumerable Dual to Observable: http://stanford.io/1kw535m
2. IEnumerator Interface: http://bit.ly/1jxWqHe
3. Pattern-Matching Bug: http://bit.ly/1bosOYv
4. Standard Query Operators: http://bit.ly/IS29t3
5. Possible Notebook Bug: http://bit.ly/1gUoJQD
6. Clojure's hash-map API: http://clojure.org/data_structures

Object-Oriented Programming (OOP)

Represent oop's objects as explicit lists of rules that transform patterns into expressions. When an object goes out of scope, its list of rules is garbage-collected. This representation avoids stocking *Mathematica*'s global symbol table with rules, avoiding in-the-offing explicit memory management for those rules.

Every object-as-instance-of-a-class must have a rule for every class member. Likewise, an object-as-implementor-of-interfaces must have a rule for every member of every interface that the object implements.

We say that an object ***provides*** implementations for its class's and interface's members, or that the object is a ***provider*** of its class's and interface's members. In both cases, the object provides implementations by providing rewrite rules.

Type-Checking

We insist that an object must be a list of rewrite rules explicitly constructed in a `Dispatch` table, i.e., an expression with Head `Dispatch`.

Since *Mathematica* rewrites dispatches back to lists at its own discretion, we cannot be sure that every object is a `Dispatch` even if we insist that the user explicitly construct objects by invoking `Dispatch`. Instead, we can only check that an object is either a `List` or a `Dispatch`, and that's what the helper function `ObjectQ` does. This helper is invoked as part of the pattern-matching at call sites of functions that require objects. This invocation at the call site implements run-time type checking.

In[1]:=
```
ClearAll[RuleQ, PatternQ, ObjectQ];

RuleQ[this_] := With[{k = Head@this}, k === Rule || k === RuleDelayed];

PatternQ[pattern_] := True;
(* Is this even well founded question,
since any expression may appear in a pattern position *)

ObjectQ[this_] :=
 Switch[Head@this,
  Dispatch, True,
  List, (* Ensure every element is a rule *)this === Select[this, RuleQ],
  _, False]
```

To type-check that an object provides a certain class or interface, just check that its list-of-rules or its `Dispatch` contains rules for each required member.

Represent types -- i.e., either classes or interfaces -- as lists of the patterns -- i.e., left-hand-sides -- required for each rule -- i.e., member -- of some concrete representation -- i.e., object. For example, every provider of `IEnumerator` must provide a rule with a pattern `MoveNext[]` -- a void-to-Boolean method -- and a pattern for `Current` -- a property producing a value.

To check that an object provides such rules, extract the patterns on the left-hand sides of the rules of the object, then check that they are a superset of the required patterns. A subtlety is that names must be stripped from parameters of patterns-with-parameters, leaving only the types of parameters. For example, we must convert `MoveNext[i_Integer]` to `MoveNext[_Integer]`. Accomplish this by rewriting `Pattern` expressions to `Blank` expressions (though see this known bug in the patterm-matcher [3]).

TODO: recurse `ProvidesTypeQ` on parameter types.

In[5]:=
```
ClearAll[ProvidesTypeQ, SubsetQ,
  GetRules, GetPatterns, GetReplacements, StripName];

SubsetQ[A_List, B_List] := Complement[A, B] === {};
SubsetQ[else___] := Throw["IllegalArgumentsException: " <> ToString@{else}]

StripName[
   Verbatim[Pattern][nym_, typeSpec : Verbatim[Blank][type___]]] := typeSpec;
StripName[hd_[args___]] := hd @@ StripName /@ {args};
StripName[else_] := else

GetRules[this_?ObjectQ] :=
  Sort@With[{h = Head@this},
    If[h === Dispatch,
     this[[1]],
     If[h === List,
      this,
      Throw["InvalidOperationException"]]]];

GetPatterns[this_?ObjectQ] :=
  StripName /@ First /@ GetRules@this;

GetReplacements[this_?ObjectQ] :=
 #[[2]] & /@ GetRules@this

ProvidesTypeQ[this_?ObjectQ, type_List] :=
  SubsetQ[StripName /@ type, GetPatterns[this]];
ProvidesTypeQ[else___] :=
 Throw["IllegalArgumentsException: " <> ToString@{else}]
```

This does not handle generics, type-wildcards, subtyping, and co- and contra-variance. We leave those developments for another time and place.

Unit Tests

In[16]:=
```
ProvidesTypeQ[{f[x_Integer, y_Real] :> x + y}, {f[_Integer, _Real]}]
```

Out[16]=
```
True
```

In[17]:=
```
ProvidesTypeQ[{f[x_Integer, y_Real] :> x + y}, {f[p_Integer, q_Real]}]
```

Out[17]=
```
True
```

In[18]:=
```
ProvidesTypeQ[{f[x_Integer, y_Real] :> x + y} // Dispatch, {f[_Integer, _Real]}]
```

Out[18]=
```
True
```

In[19]:=
```
! ProvidesTypeQ[{f[x_Integer, y_Real] :> x + y}, {f[_Integer, _Integer]}]
```
Out[19]=
```
True
```

In[20]:=
```
ProvidesTypeQ[
 {f[x_Integer, y_Real] :> x + y, g[s_String] :> s}, {f[_Integer, _Real]}]
```
Out[20]=
```
True
```

In[21]:=
```
! ProvidesTypeQ[
  {f[x_Integer, y_Real] :> x + y, g[s_String] :> s},
  {f[_Integer, _Real], h[_String]}]
```
Out[21]=
```
True
```

In[22]:=
```
$obj = {
      h[s2_String] :> s2,
      f[x_Integer, y_Real] :> x + y,
      g[s_String] :> s} // Sort // Dispatch;
```

In[23]:=
```
ProvidesTypeQ[$obj,
 {f[_Integer, _Real], h[_String]}]
```
Out[23]=
```
True
```

In[24]:=
```
ProvidesTypeQ[$obj,
 {f[q_Integer, p_Real], h[r_String]}]
```
Out[24]=
```
True
```

In[25]:=
```
ProvidesTypeQ[$obj, {f[_Integer, _Real], h[accidentalName_String]}]
```
Out[25]=
```
True
```

Utility Functions on Types

We need a way to destructively add a rule to a given object, and it's useful to have a method to check for existence of a single rule pattern; `ProvidesTypeQ` above checks for the existence of multiple rule patterns.

In[26]:=
```
ClearAll[HasPattern, IsPattern, WriteRule, RemoveRule];

HasPattern[this_?ObjectQ, pattern_?RuleQ] :=
  SubsetQ[{StripName@First@pattern}, GetPatterns@this];
HasPattern[this_?ObjectQ, pattern_?PatternQ] :=
  SubsetQ[{StripName@pattern}, GetPatterns@this];

IsPattern[this_, that_?RuleQ] :=
  StripName@First@this === StripName@First@that;
IsPattern[this_, that_?PatternQ] := StripName@First@this === StripName@that;

RemoveRule[this_?ObjectQ, ruleOrPattern_] :=
  Sort@Select[GetRules@this, ! IsPattern[#, ruleOrPattern] &];

WriteRule[this_?ObjectQ, rule_?RuleQ] :=
  Append[RemoveRule[this, rule], rule] // Sort // Dispatch;
WriteRule[else___] := Throw["IllegalArgumentsException: " <> ToString@{else}]
```

Digression: Consider Clojure hash-maps

Clojure has an exceptionally well conceived hash-map data structure and collection of functions. We may, in a future version of this document, emulate that structure and its API. Quoting from [6]

Maps (IPersistentMap)
A Map is a collection that maps keys to values.Two different map types are provided - hashed and sorted.Hash maps require keys that correctly support hashCode and equals.Sorted maps require keys that implement Comparable, or an instance of Comparator.Hash maps provide faster access (log32N hops) vs (logN hops), but sorted maps are, well, sorted.count is O (1).conj expects another (possibly single entry) map as the item, and returns a new map which is the old map plus the entries from the new, which may overwrite entries of the old.conj also accepts a MapEntry or a vector of two items (key and value).seq returns a sequence of map entries, which are key/value pairs.Sorted map also supports rseq, which returns the entries in reverse order.Maps implement IFn, for invoke () of one argument (a key) with an optional second argument (a default value), i.e.maps are functions of their keys.nil keys and values are ok.Related functions
Create a new map : hash - map sorted - map sorted - map - by
' change' a map : assoc dissoc select - keys merge merge - with zipmap
Examine a map : get contains?find keys vals map?Examine a map entry : key val

UnitTests

In[34]:=
```
{id → Unique[]}
```

Out[34]=
```
{id → $3}
```

In[35]:=
```
{id → Unique[]}.id
```

Out[35]=
```
{id → $4}.id
```

In[36]:=
```
HasPattern[{id → Unique[]}, jd]
```
Out[36]=
```
False
```

In[37]:=
```
HasPattern[{id → Unique[]}, id]
```
Out[37]=
```
True
```

In[38]:=
```
HasPattern[$obj,
 h[accidentalName_String]]
```
Out[38]=
```
True
```

In[39]:=
```
IsPattern[#, f[x_Integer, y_Real]] & /@ GetRules@$obj === {True, False, False}
```
Out[39]=
```
True
```

In[40]:=
```
RemoveRule[$obj,
  f[x_Integer, y_Real]
 ] === Sort@{
   h[s2_String] :→ s2,
   g[s_String] :→ s}
```
Out[40]=
```
True
```

In[41]:=
```
RemoveRule[$obj,
  f[x_Integer, y_Real] :→ x + y
 ] === Sort@{
   h[s2_String] :→ s2,
   g[s_String] :→ s}
```
Out[41]=
```
True
```

In[42]:=
```
RemoveRule[$obj,
  f[z_Integer, q_Real] :→ x + y
 ] === Sort@{
   h[s2_String] :→ s2,
   g[s_String] :→ s}
```
Out[42]=
```
True
```

In[43]:=
```
RemoveRule[$obj,
  f[z_Integer, q_Real] :> z + q
 ] === Sort@{
   h[s2_String] :> s2,
   g[s_String] :> s}
```
Out[43]=
```
True
```

In[44]:=
```
RemoveRule[$obj,
  j[l_List] :> Length@l] === $obj
```
Out[44]=
```
True
```

In[45]:=
```
WriteRule[$obj, j[l_List] :> Length@l] === Append[$obj,
  j[l_List] :> Length@l]
```
Out[45]=
```
True
```

In[46]:=
```
WriteRule[$obj, f[q_Integer, z_Real] :> q^2 + z^2]
```
Out[46]=
```
{f[q_Integer, z_Real] :> q^2 + z^2, g[s_String] :> s, h[s2_String] :> s2}
```

In[47]:=
```
WriteRule[$obj, f[q_Integer, z_Real] :> q^2 + z^2] ===
 Sort@{h[s2_String] :> s2, g[s_String] :> s, f[q_Integer, z_Real] :> q^2 + z^2}
```
Out[47]=
```
True
```

Inheritance and Its Style of Polymorphism

Particular rules could simulate inheritance and its style of polymorphism either by chasing prototype chains (upward inheritance) or dispatching at run time (downward inheritance). We don't need to take a cosmic position on this; each class and interface can do it in its own way.

Using Objects

To produce application-level results, apply objects' rules to other, arbitrarily rich work-expressions. This is "rewrite at run time." Contrast with the more familiar "rewrite at compile time," which is how standard oop systems transform work-expressions into actionable, application-level work-code. In a future version of this document, we might exhibit a compile-time option.

Overload Resolution; Member Lookup

In the current design, member lookup (overload-resolution) is done at run time. Later, we might include lookup at compile time.

IEnumerator and IEnumerable

In[48]:=
```
ClearAll[
  IEnumerable,
  IEnumerator,
  IEnumerableType,
  IEnumeratorType,
  GetEnumerator,
  MoveNext,
  Current];
```

Abstract Types

Here are the abstract specifications of the types `IEnumerable` and `IEnumerator`. This is not a full type system, but it is a start in the right direction. You can check these types by calling `ProvidesTypeQ[obj,type]`.

In[49]:=
```
ClearAll[IEnumerableType];
IEnumerableType = {GetEnumerator[]};
```

In[51]:=
```
ClearAll[IEnumeratorType];
IEnumeratorType = {MoveNext[], Current};
```

Contracts

Unlike C#, where interfaces are abstract, our `IEnumerator` provide an implementation that *enforces* its contract by requiring every provider object to implement the private protocol of type `MoveNext[_Integer]` and `Current[_Integer]`. Enforcing is better than simply hoping. The private contract for the private protocol is documented in the comments of the code below.

In[53]:=
```
ClearAll[privateProtocolType];
privateProtocolType = {MoveNext[_Integer], Current[_Integer]};
```

To get the `IEnumerator` interface for an object named `this`, "call" the following "function" on an object (i.e., dispatch list of rewrite rules) to get a new object that implements (provides rules for) the enumerator's state machine. The new object will call the private protocol of the old object by imposing the old object's rewrite rules on expressions of the form `MoveNext[i_Integer]` and `Current[i_Integer]` that must be provided by the old object.

- **SIDEBAR**: The quotes on "call" and "function" are to remind you that what you're really doing is rewriting the expression `IEnumerator[this_]`, after substituting an actiual object for the variable `this_`, into the right-hand-side of the `:=` assignment symbol. The rewrite rule for `IEnumerator` is permanently installed into the global symbol table because `IEnumerator` is a permanent part of our programming environment. *Mathematica* implements function-calling with expression-rewriting, as can be observed by evaluating the following two expressions. `Trace[x+x/.{x->42}]`, which rewrites first and substitutes second, i.e., rewrites `x+x` into `2x` first, then substitutes `42` for `x`; and `Trace[Function[x,x+x][42]]`, which substitues first and rewrites second, i.e., substitutes `42` for `x` and `x+x` for `Function[x,x+x][42]` in one step, and then rewrites `42+42` into `84` after the substitutions.

In[55]:=
```
ClearAll[IEnumerator];
(* Checks that the argument provides the privateProtocol. *)
IEnumerator[this_?(ProvidesTypeQ[#, privateProtocolType] &)] :=
 Module[{(* Variables for the state-machine. *)
     i = 0,
     iPlus = Undefined}, {
     (* MoveNext is, syntactically, a method. *)
     MoveNext[] :> (
       (* Access the private implementation of MoveNext[i] *)
       iPlus = MoveNext[i] /. this;
       (* The private protocol returns integer
        indices only when MoveNext stays 'inside' the sequence. *)
       With[{result = (Integer === Head[iPlus])},
        If[result, i = iPlus];
        (* The public protocol
         for IEnumerator's MoveNext produces a Boolean. *)
        result]),
     (* Current is, syntactically, a property. *)
     Current :>
      (* The following effects the public, documented protocol in
       terms of a private protocol member, namely Current[i]. *)
      If[Not[Integer === Head[iPlus]],
       Throw["InvalidOperationException"],
       If[i === 0,
        Undefined,
        (* Access this's private implementation of Current[i] *)
        Current[i] /. this]]
    }] // Sort // Dispatch
```

To get the **IEnumerable** interface for an object, call the following function, which returns a new object implementing the interface, i.e., providing rewrite rules for the interface's members.

In[57]:=
```
ClearAll[IEnumerable];
IEnumerable[this_?ObjectQ] := {
   GetEnumerator[] :> IEnumerator[this]
  } // Dispatch
```

List: a Provider of IEnumerable

list is a provider of **IEnumerable**. Its type is **list**; its constructor is **list`list**, using the class **list** as a namespace. It must implement the private protocol **MoveNext[_Integer]** and **Current[_Integer]**.

In[59]:=
```
ClearAll[list`list];
list`list[data_List] :=
 Module[{len = Length[data]},
  IEnumerable[{(* what follows is the required private protocol. *)
      MoveNext[i_Integer] :>
       With[{iPlus = i + 1},
        (* Required to produce an integer iff the new index is in-range. *)
        If[iPlus > 0 && iPlus ≤ len, iPlus, False]],
      Current[i_Integer] :>
       If[i > 0 && i ≤ len,
        data[[i]],
        Throw["IndexOutOfRangeException"]]
     } // Sort // Dispatch
  ]]
```

ForEach

Here is a straightforward implementation of **forEach**, which applies a function to every element of an **IEnumerable** for side-effect. It corresponds to *Mathematica*'s **Scan**. It's a prototype for the entire suite of LINQ-ish *Standard Query Operators* [4].

In[61]:=
```
ClearAll[forEach];
forEach[enumerable_?(ProvidesTypeQ[#, IEnumerableType] &), someFunction_] :=
 With[{enumerator = GetEnumerator[] /. enumerable},
  While[
   MoveNext[] /. enumerator,
   someFunction[Current /. enumerator]
  ]]
```

In[63]:=
```
forEach[list`list[{"John Smith", "Jim Johnson", "Sue Rabon"}], Print]
```

```
John Smith

Jim Johnson

Sue Rabon
```

A Syntactic Improvement

Leo Bushkin and I figured out how to overload *Dot* -- normally vector inner product -- so we can use more natural OOP notation.

In[64]:=
```
ClearAll[Flip];
Flip[fn_] := Function[{x, y}, fn[y, x]];
Unprotect[Dot];
SetAttributes[Dot, HoldRest];
Dot[rules_, member_] :=
  Fold[Flip@ReplaceAll, List @@ rules, {Unevaluated[member]}];
Dot[rules_, member_, members__] :=
  ((*Print@rules;Print@member;Print@members;*)
   Fold[Flip@ReplaceAll, List @@ rules, Unevaluated@{member, members}]);
Protect[Dot];
```

Unit Test

In[71]:=
```
{m → {n → {p → 42}}}.m.n.p
```

Out[71]=
```
42
```

Here is the more natural notation in action:

In[72]:=
```
ClearAll[forEach];
forEach[enumerable_?(ProvidesTypeQ[#, IEnumerableType] &), someFunction_] :=
 With[{enumerator = enumerable.GetEnumerator[]},
  While[
   enumerator.MoveNext[],
   someFunction[enumerator.Current]
  ]]
```

In[74]:=
```
forEach[list`list[{"John Smith", "Jim Johnson", "Sue Rabon"}], Print]
```

```
John Smith

Jim Johnson

Sue Rabon
```

Should behave well for empty lists.

In[75]:=
```
forEach[list`list[{}], Print]
```

Should throw for things that don't provide the `IEnumerable` type.

In[76]:=
```
Catch[forEach[foobar, Print]]
```

Out[76]=
```
IllegalArgumentsException: {foobar, {GetEnumerator[]}}
```

HashMap: a Provider of IEnumerable

In[77]:=
```
aHashMap =
 RandomSample@MapThread[Rule, {CharacterRange["a", "z"], Range[26]}] // Dispatch
```

Out[77]=
```
Dispatch[{y → 25, x → 24, t → 20, p → 16, m → 13, f → 6, q → 17, b → 2, s → 19,
  o → 15, e → 5, v → 22, d → 4, h → 8, w → 23, j → 10, n → 14, r → 18, g → 7,
  a → 1, z → 26, l → 12, i → 9, u → 21, c → 3, k → 11}, -DispatchTables-]
```

My impl of hashMap cheats by just using the impl of list`list.

In[78]:=
```
hashMap`hashMap[kvs_Dispatch] := list`list[List @@ # & /@ kvs[[1]] // Sort]
```

In[79]:=
```
forEach[hashMap`hashMap[aHashMap], Print]
```

```
{a, 1}
{b, 2}
{c, 3}
{d, 4}
{e, 5}
{f, 6}
{g, 7}
{h, 8}
{i, 9}
{j, 10}
{k, 11}
{l, 12}
{m, 13}
{n, 14}
{o, 15}
{p, 16}
{q, 17}
{r, 18}
{s, 19}
{t, 20}
{u, 21}
{v, 22}
{w, 23}
{x, 24}
{y, 25}
{z, 26}
```

IObserver and IObservable

In[80]:=
```
ClearAll[
  IObservable,
  ISubject,
  IObserver,
  IObservableType,
  IObserverType,
  Subscribe,
  OnNext,
  OnError,
  OnCompleted,
  Current];
```

Delay and Force

Due to a possible bug in the Notebook interface [5], we must abstract definitions of *Delay* and *Force*. It's natural to express *Delay* by wrapping a delayed expression in a thunk -- function of no arguments -- and then *Force* evaluation by calling the thunk. This is so natural that it hardly merits calling out, and I would not have done but for the fact that it sometimes does not work in the *Mathematica* Notebook interface, though it does work in the command-line version of *Mathematica*. The solution is to abstract the operations into explicit calls of *Delay* and *Force* and use *Hold* and *ReleaseHold*, which work in the Notebook and in the command-line version of *Mathematica*.

In[81]:=
```
ClearAll[Delay, Force];
Delay = Hold;
Force = ReleaseHold;
```

Unit Tests

In[84]:=
```
Remove[foo$, bar$];
foo$ = Delay[bar$ = 42];
Print[foo$ === Hold[bar$ = 42]];
Print[! ValueQ@bar$];
Force[foo$];
Print[ValueQ@bar$];
```

```
True

True

True
```

Hash Tables

In[90]:=
```
ClearAll[TagItem, InsertItem, DeleteItem];
```

Abstract Types

In[91]:=
```
ClearAll[IObservableType];
IObservableType = {Subscribe[_]};
```

TODO: Should be `Subscribe[_IObserverType]`. Also should have a type for exceptions.

In[93]:=
```
ClearAll[IObserverType];
IObserverType = {
   OnNext[observation_],
   OnError[exception_],
   OnCompleted[]};
```

Contracts

In[95]:=
```
ClearAll[ISubject];
ISubject[this_?ObjectQ] :=
 Module[{subscriptions = {}},
    {DebugReport[] :> subscriptions,
     OnNext[obn_] :>
      Scan[#.OnNext[obn] &, GetReplacements@subscriptions],
     OnError[exc_] :> (Scan[#.OnError[exc] &,
        GetReplacements@subscriptions]; subscriptions = Null),
     OnCompleted[] :> (Scan[#.OnCompleted[] &, GetReplacements@subscriptions];
       subscriptions = Null),
     Subscribe[that_?(ProvidesTypeQ[#, IObserverType] &)] :>
      Module[{id, subscription},
       id = If[HasPattern[that, SubscriptionId],
         that.SubscriptionId,
         Unique[]];
       subscription = (id -> that);
       AppendTo[subscriptions, subscription];
       With[
        {unsubscribe = Delay[subscriptions = RemoveRule[subscriptions, id]]},
        unsubscribe]]
    }] // Sort // Dispatch
```

In[97]:=
```
$myObl = ISubject[{}];
```

In[98]:=
```
$myObl.DebugReport[]
```

Out[98]=
```
{}
```

User-Supplied Subscription Ids

In[99]:=
```
ClearAll[$obr];
$obr[subscriptionId_] := {
     SubscriptionId :> subscriptionId,
     OnNext[msg_] :>
      Print[ToString@subscriptionId <> ": Observer OnNext: " <> ToString@msg],
     OnError[exc_] :> Print[ToString@subscriptionId <>
        ": Observer OnError: " <> ToString@exc],
     OnCompleted[] :> Print[ToString@subscriptionId <>
        ": Observer OnCompleted!"]} // Sort // Dispatch
```

In[101]:=
```
unsubscribeFirstObserver = $myObl.Subscribe[$obr[Unique[]]]
```

Out[101]=
```
Hold[subscriptions$294 = RemoveRule[subscriptions$294, id$295]]
```

In[102]:=
```
$myObl.DebugReport[]
```

Out[102]=
```
{$7 → {SubscriptionId :> $7, OnCompleted[] :>
     Print[ToString[$7] <> : Observer OnCompleted!], OnError[exc$_] :>
     Print[ToString[$7] <> : Observer OnError:  <> ToString[exc$]],
    OnNext[msg$_] :> Print[ToString[$7] <> : Observer OnNext:  <> ToString[msg$]]}}
```

In[103]:=
```
$myObl.OnNext[42]
```

```
$7: Observer OnNext: 42
```

In[104]:=
```
unsubscribeSecondObserver = $myObl.Subscribe[$obr[Unique[]]]
```

Out[104]=
```
Hold[subscriptions$294 = RemoveRule[subscriptions$294, id$297]]
```

In[105]:=
```
$myObl.DebugReport[]
```

Out[105]=
```
{$7 → {SubscriptionId :> $7, OnCompleted[] :>
     Print[ToString[$7] <> : Observer OnCompleted!], OnError[exc$_] :>
     Print[ToString[$7] <> : Observer OnError:  <> ToString[exc$]],
    OnNext[msg$_] :> Print[ToString[$7] <> : Observer OnNext:  <> ToString[msg$]]},
 $8 → {SubscriptionId :> $8, OnCompleted[] :>
     Print[ToString[$8] <> : Observer OnCompleted!], OnError[exc$_] :>
     Print[ToString[$8] <> : Observer OnError:  <> ToString[exc$]],
    OnNext[msg$_] :> Print[ToString[$8] <> : Observer OnNext:  <> ToString[msg$]]}}
```

In[106]:=
```
$myObl.OnNext[42]
```

```
$7: Observer OnNext: 42

$8: Observer OnNext: 42
```

In[107]:=
```
Force[unsubscribeFirstObserver];
```

In[108]:=
```
$myObl.OnNext[42]
```

```
$8: Observer OnNext: 42
```

System-Supplied Subscription Ids

In[109]:=
```
$obr[] := {
    OnNext[msg_] :> Print["Observer OnNext: " <> ToString@msg],
    OnError[exc_] :> Print["Observer OnError: " <> ToString@exc],
    OnCompleted[] :> Print["Observer OnCompleted!"]} // Sort // Dispatch
```

In[110]:=
```
unsubscribeThirdObserver = $myObl.Subscribe[$obr[]]
```

Out[110]=
```
Hold[subscriptions$294 = RemoveRule[subscriptions$294, id$298]]
```

In[111]:=
```
$myObl.DebugReport[]
```

Out[111]=
```
{$8 → {SubscriptionId :> $8, OnCompleted[] :>
    Print[ToString[$8] <> : Observer OnCompleted!], OnError[exc$_] :>
    Print[ToString[$8] <> : Observer OnError: <> ToString[exc$]],
   OnNext[msg$_] :> Print[ToString[$8] <> : Observer OnNext: <> ToString[msg$]]},
 $9 → {OnCompleted[] :> Print[Observer OnCompleted!],
   OnError[exc_] :> Print[Observer OnError: <> ToString[exc]],
   OnNext[msg_] :> Print[Observer OnNext: <> ToString[msg]]}}
```

In[112]:=
```
unsubscribeFourthObserver = $myObl.Subscribe[$obr[]]
```

Out[112]=
```
Hold[subscriptions$294 = RemoveRule[subscriptions$294, id$299]]
```

In[113]:=
```
$myObl.OnNext[42]
```

```
Observer OnNext: 42

$8: Observer OnNext: 42

Observer OnNext: 42
```

In[114]:=
```
Force[unsubscribeThirdObserver]
```

Out[114]=
```
{$10 → {OnCompleted[] :> Print[Observer OnCompleted!],
   OnError[exc_] :> Print[Observer OnError: <> ToString[exc]],
   OnNext[msg_] :> Print[Observer OnNext: <> ToString[msg]]},
 $8 → {SubscriptionId :> $8, OnCompleted[] :>
    Print[ToString[$8] <> : Observer OnCompleted!], OnError[exc$_] :>
    Print[ToString[$8] <> : Observer OnError: <> ToString[exc$]],
   OnNext[msg$_] :> Print[ToString[$8] <> : Observer OnNext: <> ToString[msg$]]}}
```

In[115]:=
```
$myObl.OnNext[42]
```

```
Observer OnNext: 42
```

```
$8: Observer OnNext: 42
```

In[116]:=
```
$myObl.OnCompleted[]
```

```
Observer OnCompleted!
```

```
$8: Observer OnCompleted!
```

In[117]:=
```
$myObl.OnNext[42]
```

ReplaceAll::reps : {Null} is neither a list of replacement rules nor a valid dispatch table, and so cannot be used for replacing. ≫