Kevlin Henney balances convention and reason when considering interfaces for similar functionality expressed in different languages.

Conventional & reasonable

THE PREVIOUS COLUMN[1] PROPOSED THAT THE essence of effective interface design is in being concise, consistent and conventional. Not exactly a rule of three that inspires the reader with its audacity, inspiration and wit, but certainly a solid foundation for communication rooted in the principle of least astonishment.

The following Java code expresses an interface with such aspirations:

```
interface RecentlyUsedList
{
    boolean isEmpty();
    int size();
    int capacity();
    void clear();
    void push(String newHead);
    String elementAt(int index);
    ...
}
```

A recently used list holds a potentially bounded sequence of unique strings, with the most recently pushed at the head (the zeroth position), and such that any overflow of the capacity (where the size would exceed the upper size bound) loses the least recently used.

What would this interface look like in C#? Java and C# are sufficiently similar that in theory one could define the interface almost identically in both languages. However, in practice this would not make the best use of either language's features and idioms, and would present any user with unconventional usage in one language or the other, or even both. The difference in features, most notably properties and indexers, would make the C# interface impossible to implement in Java and the Java interface an inappropriate pidgin in C#. The following is a C# interface that strikes a balance between convention and reason:

```
interface RecentlyUsedList
{
    int Count { get; }
    int Capacity { get; }
    void Clear();
    void Push(string newHead);
    string this[int index] { get; }
    ...
}
```

A different case convention is used for public members, and the names follow those typical of the .NET libraries rather than the Java style, e.g. Count rather than `size`. Both `Count` and `Capacity` are expressed as read-only properties (supporting get only, no set) and subscripted element access is expressed using an indexer rather than a named method. Indexers provide interface users with familiar syntax for working with collections by key or index, e.g. `list[0]`.

Note that one thing that is missing from the C# interface, which is found in the Java version, is a query for emptiness. This query is not offered directly by .NET collections, so the effect is achieved the long-handed way: `list.Count == 0`. Although technically this is the same, the intention of the idiom is weaker – compare "my glass is empty" with "my glass has zero content". Such an omission is also a questionable one on technical grounds. Although an emptiness query must have the same functional behaviour as comparing the size against zero, it is not always the case that it would have the same operational behaviour[2]. From a collection implementer's perspective, it is easy to determine whether any collection is empty in constant time, but it may prove impossible to determine the size in constant time without introducing an additional field to track it. The implementer is forced to adopt a representation strategy that is sometimes more general than common usage dictates.

Of course, there is nothing to prevent a programmer providing an emptiness query; it is just that it is not the conventional approach. However, its absence as part of convention does not mean that there is an absence in the surrounding idiom that would guide it: `IsEmpty`, rather than `Empty`, would be the name; a property, rather than a method, would be the means of expression.

The one minor but notable departure from the convention used in the .NET libraries is the name of the interface: the version here is named `RecentlyUsedList`, not `IRecentlyUsedList`. Although a popular convention, dating back to Microsoft's early forays into component architectures, the I-prefix one is a questionable one. As a practice it suffers

from inconsistency: classes are not prefixed with C, or delegate types with D, or enumeration types with E, or `struct types` with S, nor should they be – down that path lies the viral Hungarian Notation meme, where you would start adding prefixes signifying whether a type `is abstract`, `sealed`, `public`, `internal`, Pisces, vegetarian, available on Tuesdays, etc. Such practices are typically redundant and are often focused on addressing the wrong development issue: the type system (and therefore the development environment) already knows what the type is and, in order to use a type meaningfully, so must the programmer.

Having all the interfaces clustered in one part of the alphabet is also particularly unhelpful when it comes to alphabetically ordered listings: it adds little and can hide the presence of other types whose names legitimately begin with I but are not interfaces, e.g. `IndexOutOfRangeException`, `Icon` and `IISIntrinsicsAttribute` (now, just imagine if Microsoft had made that an interface...).

I have also noticed that the I-prefix convention often seems to encourage poor class naming. There is a temptation to name the first implementation of an interface as the interface name less the prefix. For example, under this naming scheme, `RecentlyUsedList` would be a concrete class that implemented an `IRecentlyUsedList` interface. The implication is that such an interface would have only one implementation: it would not be *an* implementation, it would be *the* implementation. This often glosses over the scope for variation that is possible, and makes it harder to distinguish between variants – if the first implementation of `IRecentlyUsedList` was called `RecentlyUsedList`, what would the second one be called? `RecentlyUsedList2`? Don't go there.

Concrete classes that implement interfaces should be named after their distinguishing behavioural or representational characteristic. For example, one implementation of a `RecentlyUsedList` interface might be bounded and implemented in terms of an array and another might be unbounded and implemented in terms of a linked list. Reasonable names for these variants would include `BoundedRecentlyUsedList` and `UnboundedRecentlyUsedList` or `RecentlyUsedArray` and `RecentlyUsedLinkedList`.

So, although the I-prefix convention is well meant, it is in the end misguided. However, it is common and nowhere near as damned as full Hungarian Notation, so it becomes a judgement call as to whether to adopt the convention or not. All other things being equal, it is worth considering not using it. Nevertheless, where there is a strong culture of using it, adopting it at least has the merit of consistency – whatever the idiom's demerits. This is where the balancing act between convention and reason plays its part. Either way, it is safe to say that it would be quite unidiomatic to adopt it in Java.

For objects, as in society, striving for meaningful equality can be tougher in practice than in principle. What does it mean for two objects to be considered equal?

For entity objects, the notion is quite simple: identity equality is the only meaningful criterion. This means that in both Java and C# the == operator is the right choice for such comparison, and there is no need to override the corresponding object equality method, `equals` and `Equals`, respectively. Similarly, in C++ entity objects are typically manipulated by pointer, and pointer equality corresponds to identity equality.

For value objects, equality is content- rather than identity-based[3]. In Java this means that, unless there is a one-to-one correspondence between identity and value, the `equals` method needs to be overridden to compare the content, which is often, but not always, a direct == or `equals` comparison between corresponding fields in the two objects. Where `equals` is overridden it is important for class users to use it rather than the == operator. For all of its abstractions and safety features, Java faithfully reproduces and amplifies one of the classic introductory C pitfalls, namely comparing strings with == rather than `strcmp`.

Values in C# can be implemented either as `class` objects, preferably immutable, or as `struct` objects. If a value type is implemented as a `struct`, the default version of `Equals` will use reflection to compare corresponding fields, which is often the right thing in terms of functionality but not necessarily in terms of performance, so overriding it manually is still a sensible option. If a value type is implemented as an immutable `class`, and there is no one-to-one correspondence between value and identity, the `Equals` method needs to be overridden. C# also supports the ability to overload the == operator. It is expected that if you override `Equals` for a value type, whether `struct` or `class`, you will also overload == for consistency – and if you overload == you are also required to overload `!=`. This is not usual for most reference types, and would be particularly odd for entity types, but where a value type is being modelled, such as `string`, this overloading is more in line with user expectation than the default, identity-based version.

In C++ only identity comparison, based on pointers, comes for free. However, in spite of the compiler's ability to default a memberwise assignment operator, there is no assumption of what constitutes correct equality between value objects. So for value types equality must be provided manually by overloading the == and `!=` operators.

OK, so for entities and values the advice seems clear cut: identity-based versus content-based equality. What about collections, such as recently used lists? It is here that black and white turns to grey. There are situations where what is needed is content-based equality, and situations where what is needed is identity-based equality. The difference being between "Do the collections referred to by these two variables contain the same values (possibly in the same arrangement)?" and "Do these two variables refer to the same collection?".

In C++, the idiom for container types is clear-cut because of the difference in syntax between using a value and a pointer. This distinction favours overloading the == operator for value-based equality; identity equality is already available via pointer comparison. The differences in Java and C# are not quite as clearly defined. However, the simplest way to cut through the choices without making things overly general and more complex for the user – Common Lisp, for example, supports four notions of equality – is to look at convention: identity- rather than content-based equality is supported for collection types. In other words, in these languages collections are not intrinsically considered to be value types, therefore equality is defaulted. This conclusion certainly simplifies any implementation of the `RecentlyUsedList` interface. ■

References

1. Kevlin Henney, "Form Follows Function", *Application Development Advisor*, March 2004.
2. Kevlin Henney, "Inside Requirements", *Application Development Advisor*, May 2003.
3. Kevlin Henney, "Objects of Value", *Application Development Advisor*, November 2003.