Notes - 12/14

Friday, December 14, 2012 9:43 AM

[In 41/2731 at 10:00 today.]

Agenda:

- Implementation update
- Topics from attached mail:
  - Generic constraints

```
Today: foo(x: Comparable): Comparable {}
Generics: foo<TComparable>(x: TComparable): TComparable {}


interface Bar {}  //or class Bar {}
class Foo<T extends Bar, U /* extends {} */, V extends Bar> {
  f() {
    var x: T;
    var y: Bar;
    y = x;
    x = y; // error

   var z: V;
    x = z; // error
  }
}


function foo<T extends Comparable>(x: T): T {
  x.compareTo();
}

var myComparable: MyComparable;
var z /* : MyComparable */ = foo(myComparable);


var foo: <T extends Comparable>(x: T) => T;

Interface Collection<T> {
  map: <U>(f: T=>U): Collection<U>;
}

var myColl: Collection<Foo> = {
  map: function(f) {

  }
}


interface Id<U> implements U {  // This is not allowed, can't know that all instantiations are legal types
  x: string;
}

function foo<T extends Comparable<U>, U>(x: T): T {}
function foo<T extends U, U>(x: T): T {} // probably should be alloweed
```

Comments to your comments inline below…

From: Jonathan Turner
Sent: Monday, December 10, 2012 1:49 PM
To: Anders Hejlsberg; TypeScript Design Team
Subject: RE: Thoughts on various topics

Some comments inline>>

From: Anders Hejlsberg
Sent: Saturday, December 8, 2012 5:37 PM
To: TypeScript Design Team
Subject: Thoughts on various topics

Luke and I had a conversation about some of our language design loose ends on Friday. Below is a summary plus some additional thoughts on my part.

Generic Constraints

I've become convinced we need them and I don't think they're all that hard. When you consider that our key value proposition is type checking, it would just be really odd to include generics with type parameters that aren't, and can't be, type checked. We'd force users to write type assertions whenever they want to access anything but the members of Object through a type parameter, and, adding insult to injury, since type assertions are never checked, there would be ***no way*** to ensure a type argument satisfies assumptions made by a generic type.

Definitely agree on this point.

I would propose we use the extends keyword to specify constraints. For example

Are we using 'extends' instead of 'implements' to imply that T is itself an interface type? If you instantiate SortedList with a class as T that implements the constraint it feels like the user might get a little confused, since the class 'implements' rather than 'extends' the interface used in the constraint.
[Anders Hejlsberg] A constraint can be any object type, including a class or an interface (and with brands gone there's really no difference). Since 'extends' is the keyword that's used in both class and interface declarations it seems to me to be the best choice. It's also what Java uses.

```
    interface Comparable<T> {
        compareTo(T other): number;
    }

    class SortedList<T extends Comparable<T>> {
        add(item: T) {
            var otherItem: T = ...;
            if (item.compareTo(otherItem) < 0) ...
            ...
        }
    }
```

This would require every type argument to SortedList to have an appropriate compareTo property and allow the compareTo property to be accessed through values of type T. Furthermore, values of type T would be subtypes of and assignable to Comparable<T>.

I like sticking with interfaces as constraints.

A few thoughts:

- Modules and namespacing
- “Export =”
  - Var: no
  - Function: no, supported via function expressoin
  - Interface: no
  - Module: no
  - Class: Yes
  - Enum:
- "export = class C {}"
- "export = expr"

class Foo { }
export = (Foo)

Export

- Modules implementing interfaces (both ‘implements’ and notation for declare files)

- Design backlog
  - Local infer from assign
  - Bringing back explicit “this” parameter type
  - ... for arguments
  - String interpolation
  - Type-only mixins
  - Decorators
  - Static initialization (#74)
  - Thinking on how to approach async via generators (#38)
  - SkyDrive feedback – implications for future thinking?
- Others?

- Are we preventing the constraints of inheritance between classes? Admittedly, the only thing I can see using this for is ensuring that the parent constructor is always called (eg class SortedList<T extends Parent> where Parent is a class with a specified constructor)
[Anders Hejlsberg] Not quite sure what you mean here. Constraints don’t ensure that parent constructs are called, they merely check that type arguments are subtypes of the constraint.
- Are we allowed to reference one type variable in the constraint of another?
  - class SortedContainer<T extends Comparable<T>, U extends List<T>> { … }
[Anders Hejlsberg] Yes. All type parameters are in scope when writing constraints, so you can even refer backwards.
- Seems like Java’s upper bounded wildcards just fall out, assuming inference helps you:
  - static sum<T extends Number>(x : Array<T>) { … }
- Lower bounded wildcards don’t seem to fall out. I’m okay with that.
[Anders Hejlsberg] That’s right, we would only support upper bounds.

Modules and namespacing

I think we should make the following changes to modules:

- A module declaration should not introduce a type as it does now. Modules should just have anonymous types.
- A module declaration containing only types and modules, transitively, should not introduce a variable or property.

This would allow code like the following:

```
module jQuery {
    // Only interfaces and modules
}

interface jQuery {
    ...
}

var jQuery: {
    ...
}
```

We would basically have three distinct declaration spaces: One for variables/properties, one for types, and one for type namespaces. In an expression context, an unqualified identifier would refer to a variable/property, and in a type context all but the last identifier of a dotted name would refer to namespaces and the last identifier would refer to a type.

Looks clean. Going back to fundules, would the change above be enough to support them?

function jQuery(x) { … } // query
module jQuery { … } // members
[Anders Hejlsberg] We still need to make a decision on that topic.

Functions and classes as modules

It is common for CommonJS and AMD modules to create an object and assign it to ‘module.exports’ instead of creating properties on the pre-allocated ‘exports’ object. We currently don’t support this pattern in TypeScript, but we could by allowing an external module to contain an “export assignment”:

```
export = Expression ;
```

Only one export assignment would be permitted in an external module, and it would be mutually exclusive with the ‘export’ modifier that we currently support. In other words, an external module either contains a single export assignment or multiple uses of the export modifier, but not both.

Since an exported expression can be a function expression, it is easy to export a function in a single statement:

```
export = function(x: number, y: number) {
    return x + y;
}
```

When the *Expression* in an export assignment consists of a single identifier that denotes a class or module, all meanings of the identifier are exported: A class will export both a value (the constructor function) and a type (the instance type), and a module will export both a value (the module object) and a namespace of types and namespaces. An import of such a module will create an alias will all those meanings.

Modules implementing interfaces

Similar to a class, I think an internal module declaration should allow an implements clause and check that the module implements the given interface(s). The topic of "how do I ensure a module has a certain shape" has come up on the forum and this would solve that problem.

Referencing a module's type

If module declarations and import statements no longer declare a type (as suggested above), it becomes hard to give an appropriate type to a dynamically loaded module, i.e. a CommonJS or AMD module loaded manually using a call to 'require'. However, if we support implements clauses on internal modules it would be possible to use interfaces. For example:

```
import Types = module("types");

export = Storage;

module MyStorage implements Types.Storage {
    // Implementation of Types.Storage interface
}
```

I'm okay with this. Seems like the drawback is that you would have to duplicate a modules type information into its interface, even if it's the only one that uses the interface.

[Anders Hejlsberg] Part of my reasoning is that you've likely created an abstraction for a dynamically loaded module, for example because you have multiple implementations. In that case you'd want an interface anyway.

This module could then be loaded dynamically and just treated as an instance of Types.Storage:

```
import Types = module("types");

function dynamicLoadStorage() {
    var storage = <Types.Storage> require("mystorage");
    ...
}
```

I think this means we don't really need any special mechanisms to reference a module's type. We can imply use interfaces.

Anders