Notes - 12/7

Wednesday, December 5, 2012 2:24 PM

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

Agenda:

- Private members vs. brands
  - Rule: The name of a private is a unique name. A private with the same user-provided name in a different class is a different name.
  - Classes don't have brands
  - But private members are effectively brands, so as soon as you add one, you are effectively branded
- Overload on string constants
  - Can specify a string literal in place of a type name (for a parameter).

```
interface Foo {
  createElement(name: string, options?: Stuff): Element;
  createElement(name: 'div', options?: DivStuff): HTMLDivElement;
  createElement(name: 'img'): HTMLImageElement;
}

var s = 'img';
createElement(s) // ambiguous unless the string overload added above

class Bar implements Foo {
 createElement(name: string): Element {
  if(name === 'div') return new HTMLDivElement();
  if(name === 'img') return new HTMLImageElement();
  else return new HTMLCustomElement();
 }
}
```

```
var f: (s: 'hello') => void  = function(s: string) {}; // invalid
var f: { (s: 'hello'): void } = function(s: string) {}; // invalid
```

  - Any overloaded signature that includes at least one string literal is a 'specialized signature'
  - Each specialized signature must be assignment compatible to at least one non-specialized signature
  - Specialized signatures are ignored for assignment compatibility to the type
  - Overload resolution rules need to handle string literals
  - Contextual typing always uses only the non-specialized overloads
- Events example for generics
  - Constraints
  - "self type" in classes (see forum thread @ Oct 8 at 3:22 PM)
  - Constants as type arguments
-
- Readonly
  - Yes - see notes below
- Fundules, clodules, namespacing
- Design backlog
  - Bringing back explicit "this" parameter type
  - … for arguments
  - String interpolation
  - Overloading on constants
  - Rubber stamp enum update
  - Type-only mixins
  - Decorations
  - Update on Fundules, clodules
  - Static initialization (#74)
  - Thinking on how to approach async via generators (#38)
  - SkyDrive feedback – implications for future thinking?
- Others?

This is what I was worried ab

While constraints at the inte
once you start implementing
use the constraints inside of
implement the example belo

```
    interface EventTarge
        removeEventListe
useCapture?: bool): void
        removeEventListe
EventListenerHandleEvent
        addEventListener
useCapture?: bool): void
        addEventListener
EventListenerHandleEvent
        dispatchEvent(ev
    }
    class EventTargetter
        var listeners: L
        addEventListener
useCapture?: bool): void
            //...
            listeners.ad
            //...
        }
        dispatchEvent(ev
            for (listene
{
                listener
handleEvent(event: TEven
            }
        }
    }
```

This typing (self typing) also
doesn't seem particularly co

Jonathan

From: Luke Hoban
Sent: Thursday, December 6
To: TypeScript Design Team
Subject: FW: defining an eve

Based on the below thread,
it. This does raise a few inte

1) Logical constraints are prom
per-se a problem at the leve
2) Can you combine generics w
literals as generic arguments
below.

```
module DOMEVENTS {
    // Would love to be
    // N must be a strin
constants???)  (see Opti
    // TEvent must be an
    // (Boy do those sou
    interface EventTarge
        removeEventListe
useCapture?: bool): void
        removeEventListe
EventListenerHandleEvent
        addEventListener
useCapture?: bool): void
        addEventListener
EventListenerHandleEvent
        dispatchEvent(ev
    }
    interface EventListe
        (evt: TEvent): v
```

bout.

erface (vs implementation) level might be a nice-to-have, g libraries in TypeScript you're going to need to be able to the generic functions. Like Luke implies, if we try to ow we run into trouble:

```
t<N, TEvent> {
ner(type: N, listener: EventListener<TEvent>,
;
ner(type: N, listener:
<TEvent>, useCapture?: bool): void;
(type: N, listener: EventListener<TEvent>,
(type: N, listener:
<TEvent>, useCapture?: bool): void;
t: Event<T>): bool;

 implements EventTarget<N, TEvent> {
ist<EventListener<TEvent>>;
(type: N, listener: EventListener<TEvent>,
 {

d(listener);


t: Event<T>): bool) {
r: EventListener<TEvent> in listeners)

.handleEvent(evt);  // ← FAILS.  evt: Event<T> but
t)
```

is something I've seen requests for a few times, and it ntroversial to also support.

, 2012 6:50 PM

ent in a Typescript interface

I tried taking the DOM events model and genericizing resting points:
inent in the type parameter to EventTarget. These aren't l of this example (there's no implementation code).
ith overload on constants? That is, can you pass string s? There's a compelling example of wanting this in #4

```
able to genericize over supported events
g (combination of generics and overload on
on #4 below)
 interface inheriting from Event
nd like constraints :-))
t<N, TEvent> {
ner(type: N, listener: EventListener<TEvent>,
;
ner(type: N, listener:
<TEvent>, useCapture?: bool): void;
(type: N, listener: EventListener<TEvent>,
;
(type: N, listener:
<TEvent>, useCapture?: bool): void;
t: Event<T>): bool;

ner<TEvent> {
oid;
```

Last time we discussed fundules/clodules we landed on this summary:

- *0th: (Mostly?) already supported, always*
  - *Module F.x {}*
  - *Modules with only types do not define variables*
  - *(We probably also need to remove the fact that a module defines a type)*
- *1st: Class and function declarations F can, in the same file and scope, be augmen*
  - *var F.x;*
  - *class F.x {}*
  - *function F.x() {}*
  - *NOT module F {}*
- *2nd: Interfaces can have dotted names which (this is actually orthogonal to 1st)*
  - *interface F.x {} // Allow namespacing interfaces/types*
- *3rd: interface + class is allowed (this is lightweight mixins, need to look at motiva*
  - *class F {}; interface F {}*

Common use cases for this were things like:

```
function foo() {}
var foo.data = 3;
function foo.bar() {}
```

The thing I continue to find frustrating about this approach is that it is so very clo same.

```
function foo() { }
foo.data = 3
```

```
var x = [1,2,3]
var x.Foo = class {
};
class x.Foo {
};
```

This (at least for me) begs the question – can we really not just interpret the latte

The thinking we've put into the dotted-name-bindings feature potentially helps t previous efforts to directly handle the code as JS users write it today above. Her

1) Function (and class if needed) declarations are extended in their scope by assign
2) As with the dotted-name-binding syntax, this applies only in the same scope, and binding.
3) It augments the type associated with the binding 'foo'

The argument against this approach that I can see is just that it is not as "declara binding. For me though, that weighs against asking JS developers to write their c and the opportunity to better type existing JS code.

Luke

ted using:

ting examples)

se to the code that users actually write, but is not the

er as meaning what we say the former should mean?

o answer some of the questions that have been raised in
e's a sketch of how it could work:
ments to properties on the function.
d only through reference to the name that declared the

tive" about the intent to introduce a new name
code using different syntax to accomplish the same thing,

```
    }
    interface EventListe
        handleEvent(even
    }
    // Note use of 'this
    interface Event {
        timeStamp: numbe
        defaultPrevented
        isTrusted: bool;
        currentTarget: E
        target: EventTar
        eventPhase: numb
        type: string;
        cancelable: bool
        bubbles: bool;
        initEvent(eventT
bool): void;
        stopPropagation(
        stopImmediatePro
        preventDefault()
        CAPTURING_PHASE:
        AT_TARGET: numbe
        BUBBLING_PHASE:
    }
    interface CustomEven
        detail: T;
        initCustomEvent(
bool, detailArg: Object)
    }

    // Tests
    // Option #1
    interface Glow exten
        isGlowing: bool;
        setGlow(glow: bo
        onglowstarted: E
    }

    // Option #2
    interface Glow exten
        isGlowing: bool;
        setGlow(glow: bo
        addEventListener
EventListener<CustomEven
    }

    // Option #3
    interface Glow exten
        isGlowing: bool;
        setGlow(glow: bo
        addEventListener
EventListener<CustomEven
        removeEventListe
EventListener<CustomEven
    }

    // Option #4
    interface Glow exten
        EventTarget,
        EventTarget<"glo
    {
        isGlowing: bool;
        setGlow(glow: bo
    }

}
```

From: Luke Hoban
Sent: Thursday, December 6
To: Max Reeder; TypeScript
Subject: RE: defining an even

This is a good question.

Even the DOM specs have a
interfaces are usually describ
the string names and associa
be challenging in general.

```
    `

nerHandleEvent<TEvent> {
t: TEvent); void;

' type

r;
: bool;

ventTarget<this>;
get<this>;
er;

;

ypeArg: string, canBubbleArg: bool, cancelableArg:

): void;
pagation(): void;
: void;
 number;
r;
number;

t<T> extends Event {

typeArg: string, canBubbleArg: bool, cancelableArg:
: void;




ds EventTarget {

ol): void;
ventHandler<CustomEvent<{glowiness: any}>>;



ds EventTarget {

ol): void;
(type: "glowStarted", listener:
t<{glowiness: any}>>, useCapture?: bool): void;



ds EventTarget {

ol): void;
(type: "glowStarted", listener:
t<{glowiness: any}>>, useCapture?: bool): void;
ner(type: "glowStarted", listener:
t<{glowiness: any}>>, useCapture?: bool): void;



ds

wStarted", CustomEvent<{glowiness: any}>>


ol): void;
```

, 2012 6:43 PM
Discussions
nt in a Typescript interface

hard time with this – the events supported by DOM
ped by a side table separate from the formal IDL. Capturing
ated types directly in the formal API surface area is going to

Assuming you also expose “o
from the DOM EventTarget i
addEventListener/removeEv

This example uses generics,
DOM EventHandler interface
is generic over its details pro
to COM EventHandler and C

```
interface Glow exten
    isGlowing: bool;
    setGlow(glow: bo
    onglowstarted: E
}
```

If you do not expose “onfoo”
aEL/rEL/dE. This is harder as
help here, being able to say
addEventListener is added w
string “glowStarted”:

```
interface Glow exten
    isGlowing: bool;
    setGlow(glow: bo
    addEventListener
EventListener<CustomEven
}
```

However, to be truly correct
pretty wordy:

```
interface Glow exten
    isGlowing: bool;
    setGlow(glow: bo
    addEventListener
EventListener<CustomEven
    removeEventListe
EventListener<CustomEven
}
```

To address the wordiness, In
write this. However, I’m pre
thing directly in TypeScript (

```
interface Glow exten
    EventTarget,
    EventTarget<"glo
{
    isGlowing: bool;
    setGlow(glow: bo
}
```

At the end of the day though
simpler and step outside the

Luke

From: Max Reeder
Sent: Thursday, December 6
To: TypeScript Discussions
Subject: defining an event in

Hello,

It’s API spec review time for
idea of using Typescript to d
…But nobody knows how to
On our team we maintain a f
feature “glow”, where our co

```
interface IGlow {
        //property:
    isGlowing: bool;

    //setter:
```

onfoo"-style events, you can strongly type these, and extend
interface to make clear that this also supports
entListener/dispatchEvent.

which we are working on now. It also assumes that the
e is generic over its Event object type, and that CustomEvent
perty type. (I'm not positive that WinJS events map exactly
ustomEvent, but it looked like they are trying to be close):

```
ds EventTarget {

ol): void;
ventHandler<CustomEvent<{glowiness: any}>>;
```

"-style events, you would likely want to try to strongly type
s noted above. We are looking at an extension that might
the following – where note that an additional overload of
which only applies when it's first parameter is the literal

```
ds EventTarget {

ol): void;
(type: "glowStarted", listener:
t<{glowiness: any}>>, useCapture?: bool): void;
```

, you'd need to include removeEventListner too. This gets

```
ds EventTarget {

ol): void;
(type: "glowStarted", listener:
t<{glowiness: any}>>, useCapture?: bool): void;
ner(type: "glowStarted", listener:
t<{glowiness: any}>>, useCapture?: bool): void;
```

principal, the following would be a more compact way to
etty sure we won't actually be able to support this sort of
passing a string literal as a generic argument):

```
ds

wStarted", CustomEvent<{glowiness: any}>>


ol): void;
```

, you might find that, like in DOM IDL, you want to keep this
formal type system in capturing these event interfaces.

, 2012 3:49 PM

a Typescript interface

Windows / DevX / UIP / WinJS, and somebody had the great
efine our API surface. Neat!
define a DOM event in an interface.
few javascript controls. For example, we want to add a new
ontrol will fire a customEvent "glowStarted":

```
        setGlow(glow: bool) =>

            //event:
            glowStarted (event
}
```

Any ideas on a best practice
pasting around ☺
Max

We've seen several TypeScript users struggle with how to model APIs that have morally read-only fields. To be 'safe' and ensure that errors are caught when TypeScript consumers of an API try to modify the fields, authors (like our dev team) often end up writing things like:

```
class DeleteExpressionSyntax extends UnaryExpressionSyntax {
    private _deleteKeyword: ISyntaxToken;
    private _expression: ExpressionSyntax;

    constructor(deleteKeyword: ISyntaxToken,
                expression: ExpressionSyntax) {
        super();

        this._deleteKeyword = deleteKeyword;
        this._expression = expression;
    }

    public kind(): SyntaxKind {
        return SyntaxKind.DeleteExpression;
    }

    public deleteKeyword(): ISyntaxToken {
        return this._deleteKeyword;
    }

    public expression(): ExpressionSyntax {
        return this._expression;
    }
}
```

This gets quite verbose. It also ends up exposing double the API surface in the underlying JavaScript (both _expression and expression). In practice – the author of this really just wants to say this:

```
class DeleteExpressionSyntax extends UnaryExpressionSyntax {
    readonly deleteKeyword: ISyntaxToken;
    readonly expression: ExpressionSyntax;

    constructor(deleteKeyword: ISyntaxToken,
                expression: ExpressionSyntax) {
        super();

        this._deleteKeyword = deleteKeyword;
        this._expression = expression;
    }
}
```

Or:

```
class DeleteExpressionSyntax extends UnaryExpressionSyntax {
    constructor(public readonly deleteKeyword: ISyntaxToken,
                public readonly expression: ExpressionSyntax) {
        super();
    }
}
```

const x = 3;

The assumption is that "readonly" above is similar semantics to C#. Statically, it is an error to write to the field outside the constructor. There is no runtime enforcement. In principle, a "checked" runtime could insert defineOwnProperty("deleteKeyword", {writable: false}) at the end of the constructor to enforce this at runtime as well, though similar to other runtime checks, that is likely rarely if ever worth the overhead.

There's a question of how this relates to 'const'. It's a bit different than the 'const' local feature in ES6 because the initialization can happen anywhere in the constructor. It may be that the semantic notions are still close enough that it's worth considering re-using the keyword here.

Notes:

1) This has to show up in types
2) Module exports are always readonly

> void;

tDetail: {glowiness: object});

here? What we have above is what all the PM's are copy

```
class Foo {
  bar() : number;
  readonly bar: () => number;
}
```

Luke