# Notes - 11/2

Friday, November 2, 2012 12:38 PM

[In 41/1719 at 1:00 today.]

Agenda:

- Generics
- Design backlog
  - Update on Fundules, clodules and lightweight mixins
  - … for arguments
  - "boolean"? (#135)
    - We make the breaking change, but keep bool for awhile
  - Better typing for Function.prototype.bind (#133)
    - f.bind(thisVal: any): this;
  - Static initialization (#74)
  - Generics, map out requirements (#185)
  - Thinking on how to approach async via generators (#38)
  - Overloading on constants
  - String interpolation
  - Type the 'this' parameter
    - Just another parameter?
  - SkyDrive feedback – implications for future thinking?
- Others?

```
Interface Foo {
  bar(callback: (this: Foo) => string);
}

Foo.bar(function(this) {
  this
}
```

```
Class Foo {
  constructor() {
    this.bar = this.bar.bind(this);
  }

  bar() {
  }
}
```

Generics recap:

- Type parameters on:
  - Interfaces
  - Classes
  - Functions (some debate, but let's assume all)
- No constraints
- Type parameter type: like a fresh {}
  - (<T>x). Shows Object members
  - null, undefined and any assignable to T
- Assignment and subtyping
- Function signatures can have type params
- Type argument inference

Examples:

```
var f: {
  <T>(t1: T, t2: T): void;
}

var g: {
  <T,U>(t1: T, t2: U): void;
}

Var h: {
  (t1: any, t2: any): void
}

f = g;  // ok
g = f;  // error
f = h; // ok
h = f; // ok
g = h; // ok
h = g; // ok
```

Notes;

1. Error to have a type parameter on a function signature which is
   - Inference

```
foo<T>(x: T): T {… }
foo(5) // should infer foo<number>(5)
```

Goal is to rarely have to say the type arguments

```
class List<T> {
  first: T;
  getAt(i: number): T
  setAt(i: number, v: T): void;
}


map<T,U>(items: List<T>, mapper: (item: T) => U): List<U>
var foos = new List<Customer>();
var names = map(foos, x => x.name);
```

```
var f: (this: Foo) => string = foo.bar

f()

setTimeout(foo.bar);
```

```
Class Point {
  foo(x: string): string {}
}
```

```
Declare class Point {
  foo(x:string): string {}
}
```

```
// Dig out mail on this types
interface Function {
  bind(thisVal: any): this;
  call(thisVal: any, /* all of the call signatures inlined here */): /*return type of corresponding signature */
  apply(thisVal: any, args: any[]); any; /*return type of corresponding signature */
}
```

```
interface Promise {
  then(success: (x: any) => any): Promise;
}

interface Promise<T> {
  then<U>(success: (x: T) => U): Promise<U>;
  then<U>(success: (x: T) => Promise<U>): Promise<U>;
}
```

Part 1: match foos with List<T> and make a bunch of inferences for T. Then run best common type on these and use result or error.

Part 2: Idea:

1. Each argument expression has a list of type parameters it depends on
2. In left to right order, when you get to an argument expression that depends on type parameters, lock down those parameters based on all inferense so far, and keep going.

Inference of generic parameters is fully separate from overload resolution.

```
f<T>(t1: T, t2: T)
f<t,U>(t: T, t2: U)
```

Follow same sort of rule as:

```
var f : {
        (x: number): number;
        (x: number, options?: any): string;
}
```

```
var z = f(5)
```

Q: Are there really any uses for method-level type parameters only in return position?

Later topic: Contextual typing: var x: number = bar();

Type arguments and constructors:

```
class List<T> {
  constructor(defaultItem: T) {…}
}

var List:{
  new <T>(defaultItem: T): List<T>
}
```

Things that you can't do here:

- Flatten<T> below

```
flatten<T>(this: List<List<T>>): List<T>;
```

```
Var p: Promise;

function baz(x: any): number {}

var q = p.then(function(x) {
   return baz(x);
})

Var r : Promise<void> = q.then(function(y) {
 console.log(y);
})


r.then(function(z) {
   // z: void
})



// Could - but won't:
interface Promise<T> {
 then<U>(success: (x: T) => void): Promise;
 then<U>(success: (x: T) => U): Promise<U>;
 then<U>(success: (x: T) => Promise<U>): Promise<U>;
}

interface Promise {
 then<U>(success: () => U): Promise<U>;
 then<U>(success: () => Promise<U>): Promise<U>;
}
```