Generics

Our first generic

Classes, functions and interfaces support *generic type arguments*

```
function identity<T>(arg: T): T {
    return arg;
}
```

It looks like:

```
function anyIdentity(arg: any) {
    return arg;
}
```

Question: What's the difference?

`identity`'s return type resolves to the type of the first parameter.

Generic interfaces

The generic `identity` can also be modelled with an interface

```
interface IdentityFn {
    <T>(arg: T): T;
}
```

We can assign our `identity` function to it:

```
function identity<T> (arg: T){
    return arg;
}

const myIdentity: IdentityFn = identity;
```

Generic classes

We can use generics in classes to.

```
class Generator<T> {

    constructor(private seed: T, private step: (value: T) => T) { }

    next() {
        return this.seed = this.step(this.seed);
    }
}

const countGenerator = new Generator<number>(0, n => n + 1);
const dotGenerator = new Generator<string>('', n => n + '.');

countGenerator.next(); // => 1
countGenerator.next(); // => 2
dotGenerator.next();    // => .
dotGenerator.next();    // => ..
```

Generic inference

Often the generic type can be inferred and we don't have to worry about it.

```
const a = [1, 2, 5]; // => type: Array<number>
```

```
class Generator<T> {

    constructor(private seed: T, private step: (value: T) => T) { }

    next() {
        return this.seed = this.step(this.seed);
    }
}
const a = new Generator(0, n => n + 1); // Generator<number>
```

Generic constraints

Let's create a `LongCache`. It caches the longest value in `current`.

```
const longName = new LongCache('Obiwan');
longName.update('test'); // => longName.current = 'Obiwan'
longName.update('LongerName'); // => longName.current = 'LongerName'
longName.update('LongestName'); // => longName.current = 'LongestName'

const longArray = new LongCache([0,2]);
```

```
longArray.update([42]); // => longArray.current [0, 2]
longArray.update([1, 2, 3]); // => longArray.current [1, 2, 3]
```

`LongCache` implementation

```
class LongCache<T> {
    constructor(public current: T) { }

    update(value: T){
        if(this.current.length < value.length) {
            this.current = value;
        }
    }
}
```

Except this doesn't work...

```
// => Error  Property 'length' does not exist on type 'T'.
```

Type Constraints

In order to make this work, we should add a *constraint*

```
class Longest<T extends { length: number }> {
    // ...
}
```

(Or with a named interface)

```
interface Measurable {
    length: number;
}

class Longest<T extends Measurable> {
    // ...
}
```
