TypeScript

TypeScript is a language for application-scale JavaScript. TypeScript adds optional types, classes, and modules to JavaScript. TypeScript supports tools for large-scale JavaScript applications for any browser, for any host, on any OS. TypeScript compiles to readable, standards-based JavaScript. Try it out at the playground, and stay up to date via our blog and twitter account.

TypeScript lets you write JavaScript the way you want to. TypeScript is a typed superset of JavaScript that compiles to plain JavaScript. Any host. Any OS. Open Source.

Try it out at http://www.typescriptlang.org/playground

Basic Types

For programs to be useful, we need to be able to work with some of the simplest units of data: numbers, strings, structures, boolean values, and the like. In TypeScript, we support much the same types as you would expected in JavaScript, with a convenient enumeration type thrown in to help things along.

Boolean

The most basic datatype is the simple true/false value, which JavaScript and TypeScript (as well as other languages) call a 'boolean' value.

```
var isDone: boolean = false;
```

Number

As in JavaScript, all numbers in TypeScript are floating point values. These floating point numbers get the type 'number'.

```
var height: number = 6;
```

String

Another fundamental part of creating programs in JavaScript for webpages and servers alike is working with textual data. As in other languages, we use the type 'string' to refer to these textual datatypes. Just like JavaScript, TypeScript also uses the double quote (") or single quote (') to surround string data.

```
var name: string = "bob";
name = 'smith';
```

Array

TypeScript, like JavaScript, allows you to work with arrays of values. Array types can be written in one of two ways. In the first, you use the type of the elements followed by '[]' to denote an array of that element type:

```
var list:number[] = [1, 2, 3];
```

The second way uses a generic array type, Array<elemType>:

```
var list:Array<number> = [1, 2, 3];
```

Enum

A helpful addition to the standard set of datatypes from JavaScript is the 'enum'. Like languages like C#, an enum is a way of giving more friendly names to sets of numeric values.

```
enum Color {Red, Green, Blue};
var c: Color = Color.Green;
```

By default, enums begin numbering their members starting at 0. You can change this by manually setting the value of one its members. For example, we can start the previous example at 1 instead of 0:

```
enum Color {Red = 1, Green, Blue};
var c: Color = Color.Green;
```

Or, even manually set all the values in the enum:

```
enum Color {Red = 1, Green = 2, Blue = 4};
var c: Color = Color.Green;
```

A handy feature of enums is that you can also go from a numeric value to the name of that value in the enum. For example, if we had the value 2 but weren't sure which that mapped to in the Color enum above, we could look up the corresponding name:

```
enum Color {Red = 1, Green, Blue};
var colorName: string = Color[2];

alert(colorName);
```

Any

We may need to describe the type of variables that we may not know when we are writing the application. These values may come from dynamic content, eg from the user or 3rd party library. In these cases, we want to opt-out of type-checking and let the values pass through compile-time checks. To do so, we label these with the 'any' type:

```
var notSure: any = 4;
notSure = "maybe a string instead";
notSure = false; // okay, definitely a boolean
```

The 'any' type is a powerful way to work with existing JavaScript, allowing you to gradually opt-in and opt-out of type-checking during compilation.

The 'any' type is also handy if you know some part of the type, but perhaps not all of it. For example, you may have an array but the array has a mix of different types:

```
var list:any[] = [1, true, "free"];
```

```
list[1] = 100;
```

Void

Perhaps the opposite in some ways to 'any' is 'void', the absence of having any type at all. You may commonly see this as the return type of functions that do not return a value:

```
function warnUser(): void {
    alert("This is my warning message");
}
```

Interfaces

One of TypeScript's core principles is that type-checking focuses on the 'shape' that values have. This is sometimes called "duck typing" or "structural subtyping". In TypeScript, interfaces fill the role of naming these types, and are a powerful way of defining contracts within your code as well as contracts with code outside of your project.

Our First Interface

The easiest way to see how interfaces work is to start with a simple example:

```
function printLabel(labelledObj: {label: string}) {
  console.log(labelledObj.label);
}

var myObj = {size: 10, label: "Size 10 Object"};
printLabel(myObj);
```

The type-checker checks the call to 'printLabel'. The 'printLabel' function has a single parameter that requires that the object passed in has a property called 'label' of type string. Notice that our object actually has more properties than this, but the compiler only checks to that *at least* the ones required are present and match the types required.

We can write the same example again, this time using an interface to describe the requirement of having the 'label' property that is a string:

```
interface LabelledValue {
  label: string;
}

function printLabel(labelledObj: LabelledValue) {
  console.log(labelledObj.label);
}
```

```
var myObj = {size: 10, label: "Size 10 Object"};
printLabel(myObj);
```

The interface 'LabelledValue' is a name we can now use to describe the requirement in the previous example. It still represents having a single property called 'label' that is of type string. Notice we didn't have to explicitly say that the object we pass to 'printLabel' implements this interface like we might have to in other languages. Here, it's only the shape that matters. If the object we pass to the function meets the requirements listed, then it's allowed.

It's worth pointing out that the type-checker does not require that these properties come in any sort of order, only that the properties the interface requires are present and have the required type.

Optional Properties

Not all properties of an interface may be required. Some exist under certain conditions or may not be there at all. These optional properties are popular when creating patterns like "option bags" where the user passes an object to a function that only has a couple properties filled in.

Here's as example of this pattern:

```
interface SquareConfig {
  color?: string;
  width?: number;
}

function createSquare(config: SquareConfig): {color: string; area: number} {
  var newSquare = {color: "white", area: 100};
  if (config.color) {
    newSquare.color = config.color;
  }
  if (config.width) {
    newSquare.area = config.width * config.width;
  }
  return newSquare;
}

var mySquare = createSquare({color: "black"});
```

Interfaces with optional properties are written similar to other interfaces, which each optional property denoted with a '?' as part of the property declaration.

The advantage of optional properties is that you can describe these possibly available properties while still also catching properties that you know are not expected to be available. For example, had we mistyped the name of the property we passed to 'createSquare', we would get an error message letting us know:

```
interface SquareConfig {
  color?: string;
  width?: number;
}

function createSquare(config: SquareConfig): {color: string; area: number} {
  var newSquare = {color: "white", area: 100};
  if (config.color) {
    newSquare.color = config.collor;  // Type-checker can catch the mistyped
name here
  }
  if (config.width) {
    newSquare.area = config.width * config.width;
  }
  return newSquare;
}

var mySquare = createSquare({color: "black"});
```

Function Types

Interfaces are capable of describing the wide range of shapes that JavaScript objects can take. In addition to describing an object with properties, interfaces are also capable of describing function types.

To describe a function type with an interface, we give the interface a call signature. This is like a function declaration with only the parameter list and return type given.

```
interface SearchFunc {
  (source: string, subString: string): boolean;
}
```

Once defined, we can use this function type interface like we would other interfaces. Here, we show how you can create a variable of a function type and assign it a function value of the same type.

```
var mySearch: SearchFunc;
mySearch = function(source: string, subString: string) {
  var result = source.search(subString);
  if (result == -1) {
    return false;
  }
  else {
    return true;
  }
}
```

For function types to correctly type-check, the name of the parameters do not need to match. We could have, for example, written the above example like this:

```
var mySearch: SearchFunc;
```

```
mySearch = function(src: string, sub: string) {
  var result = src.search(sub);
  if (result == -1) {
    return false;
  }
  else {
    return true;
  }
}
```

Function parameters are checked one at a time, with the type in each corresponding parameter position checked against each other. Here, also, the return type of our function expression is implied by the values it returns (here *false* and *true*). Had the function expression returned numbers or strings, the type-checker would have warned us that return type doesn't match the return type described in the SearchFunc interface.

Array Types

Similarly to how we can use interfaces to describe function types, we can also describe array types. Array types have an 'index' type that describes the types allowed to index the object, along with the corresponding return type for accessing the index.

```
interface StringArray {
  [index: number]: string;
}

var myArray: StringArray;
myArray = ["Bob", "Fred"];
```

There are two types of supported index types: string and number. It is possible to support both types of index, with the restriction that the type returned from the numeric index must be a subtype of the type returned from the string index.

While index signatures are a powerful way to describe the array and 'dictionary' pattern, they also enforce that all properties match their return type. In this example, the property does not match the more general index, and the type-checker gives an error:

```
interface Dictionary {
  [index: string]: string;
  length: number;    // error, the type of 'length' is not a subtype of the
indexer
}
```

Class Types

Implementing an interface

One of the most common uses of interfaces in languages like C# and Java, that of explicitly enforcing that a class meets a particular contract, is also possible in TypeScript.

```
interface ClockInterface {
    currentTime: Date;
}

class Clock implements ClockInterface  {
    currentTime: Date;
    constructor(h: number, m: number) { }
}
```

You can also describe methods in an interface that are implemented in the class, as we do with 'setTime' in the below example:

```
interface ClockInterface {
    currentTime: Date;
    setTime(d: Date);
}

class Clock implements ClockInterface  {
    currentTime: Date;
    setTime(d: Date) {
        this.currentTime = d;
    }
    constructor(h: number, m: number) { }
}
```

Interfaces describe the public side of the class, rather than both the public and private side. This prohibits you from using them to check that a class also has particular types for the private side of the class instance.

Difference between static/instance side of class

When working with classes and interfaces, it helps to keep in mind that a class has *two* types: the type of the static side and the type of the instance side. You may notice that if you create an interface with a construct signature and try to create a class that implements this interface you get an error:

```
interface ClockInterface {
    new (hour: number, minute: number);
}

class Clock implements ClockInterface  {
    currentTime: Date;
    constructor(h: number, m: number) { }
}
```

This is because when a class implements an interface, only the instance side of the class is checked. Since the constructor sits in the static side, it is not included in this check.

Instead, you would need to work with the 'static' side of the class directly. In this example, we work with the class directly:

```
interface ClockStatic {
    new (hour: number, minute: number);
}

class Clock  {
    currentTime: Date;
    constructor(h: number, m: number) { }
}

var cs: ClockStatic = Clock;
var newClock = new cs(7, 30);
```

Extending Interfaces

Like classes, interfaces can extend each other. This handles the task of copying the members of one interface into another, allowing you more freedom in how you separate your interfaces into reusable components.

```
interface Shape {
    color: string;
}

interface Square extends Shape {
    sideLength: number;
}

var square = <Square>{};
square.color = "blue";
square.sideLength = 10;
```

An interface can extend multiple interfaces, creating a combination of all of the interfaces.

```
interface Shape {
    color: string;
}

interface PenStroke {
    penWidth: number;
}

interface Square extends Shape, PenStroke {
    sideLength: number;
}

var square = <Square>{};
square.color = "blue";
square.sideLength = 10;
square.penWidth = 5.0;
```

Hybrid Types

As we mentioned earlier, interfaces can describe the rich types present in real world JavaScript. Because of JavaScript's dynamic and flexible nature, you may occasionally encounter an object that works as a combination of some of the types described above.

One such example is an object that acts as both a function and an object, with additional properties:

```
interface Counter {
    (start: number): string;
    interval: number;
    reset(): void;
}

var c: Counter;
c(10);
c.reset();
c.interval = 5.0;
```

When interacting with 3rd-party JavaScript, you may need to use patterns like the above to fully-describe the shape of the type.

Classes

Traditional JavaScript focuses on functions and prototype-based inheritance as the basic means of building up reusable components, but this may feel a bit awkward to programmers more comfortable with an object-oriented approach, where classes inherit functionality and objects are built from these classes. Starting with ECMAScript 6, the next version of JavaScript, JavaScript programmers will be able to build their applications using this object-oriented class-based approach. In TypeScript, we allow developers to use these techniques now, and compile them down to JavaScript that works across all major browsers and platforms, without having to wait for the next version of JavaScript.

Classes

Let's take a look at a simple class-based example:

```
class Greeter {
    greeting: string;
    constructor(message: string) {
        this.greeting = message;
    }
    greet() {
        return "Hello, " + this.greeting;
    }
}

var greeter = new Greeter("world");
```

The syntax should look very familiar if you've used C# or Java before. We declare a new class 'Greeter'. This class has three members, a property called 'greeting', a constructor, and a method 'greet'.

You'll notice that in the class when we refer to one of the members of the class we prepend 'this.'. This denotes that it's a member access.

In the last line we construct an instance of the Greeter class using 'new'. This calls into the constructor we defined earlier, creating a new object with the Greeter shape, and running the constructor to initialize it.

Inheritance

In TypeScript, we can use common object-oriented patterns. Of course, one of the most fundamental patterns in class-based programming is being able to extend existing classes to create new ones using inheritance.

Let's take a look at an example:

```
class Animal {
    name:string;
    constructor(theName: string) { this.name = theName; }
    move(meters: number = 0) {
        alert(this.name + " moved " + meters + "m.");
    }
}

class Snake extends Animal {
    constructor(name: string) { super(name); }
    move(meters = 5) {
        alert("Slithering...");
        super.move(meters);
    }
}

class Horse extends Animal {
    constructor(name: string) { super(name); }
    move(meters = 45) {
        alert("Galloping...");
        super.move(meters);
    }
}

var sam = new Snake("Sammy the Python");
var tom: Animal = new Horse("Tommy the Palomino");

sam.move();
tom.move(34);
```

This example covers quite a bit of the inheritance features in TypeScript that are common to other languages. Here we see using the 'extends' keywords to create a subclass. You can see this where 'Horse' and 'Snake' subclass the base class 'Animal' and gain access to its features.

The example also shows off being able to override methods in the base class with methods that are specialized for the subclass. Here both 'Snake' and 'Horse' create a 'move' method that overrides the 'move' from 'Animal', giving it functionality specific to each class.

Private/Public modifiers

Public by default

You may have noticed in the above examples we haven't had to use the word 'public' to make any of the members of the class visible. Languages like C# require that each member be explicitly labelled 'public' to be visible. In TypeScript, each member is public by default.

You may still mark members a private, so you control what is publicly visible outside of your class. We could have written the 'Animal' class from the previous section like so:

```
class Animal {
    private name:string;
    constructor(theName: string) { this.name = theName; }
    move(meters: number) {
        alert(this.name + " moved " + meters + "m.");
    }
}
```

Understanding private

TypeScript is a structural type system. When we compare two different types, regardless of where they came from, if the types of each member are compatible, then we say the types themselves are compatible.

When comparing types that have 'private' members, we treat these differently. For two types to be considered compatible, if one of them has a private member, then the other must have a private member that originated in the same declaration.

Let's look at an example to better see how this plays out in practice:

```
class Animal {
    private name:string;
    constructor(theName: string) { this.name = theName; }
}

class Rhino extends Animal {
        constructor() { super("Rhino"); }
}

class Employee {
    private name:string;
    constructor(theName: string) { this.name = theName; }
}

var animal = new Animal("Goat");
var rhino = new Rhino();
var employee = new Employee("Bob");

animal = rhino;
animal = employee; //error: Animal and Employee are not compatible
```

In this example, we have an 'Animal' and a 'Rhino', with 'Rhino' being a subclass of 'Animal'. We also have a new class 'Employee' that looks identical to 'Animal' in terms of shape. We create some instances of these classes and then try to assign them to each other to see what will happen. Because 'Animal' and 'Rhino' share the private side of their shape from the same declaration of 'private name: string' in 'Animal', they are compatible. However, this is not the case for 'Employee'. When we try to assign from an 'Employee' to 'Animal' we get an error that these types are not compatible. Even though 'Employee' also has a private member called 'name', it is not the same one as the one created in 'Animal'.

Parameter properties

The keywords 'public' and 'private' also give you a shorthand for creating and initializing members of your class, by creating parameter properties. The properties let you can create and initialize a member in one step. Here's a further revision of the previous example. Notice how we drop 'theName' altogether and just use the shortened 'private name: string' parameter on the constructor to create and initialize the 'name' member.

```
class Animal {
    constructor(private name: string) { }
    move(meters: number) {
        alert(this.name + " moved " + meters + "m.");
    }
}
```

Using 'private' in this way creates and initializes a private member, and similarly for 'public'.

Accessors

TypeScript supports getters/setters as a way of intercepting accesses to a member of an object. This gives you a way of having finer-grained control over how a member is accessed on each object.

Let's convert a simple class to use 'get' and 'set'. First, let's start with an example without getters and setters.

```
class Employee {
    fullName: string;
}

var employee = new Employee();
employee.fullName = "Bob Smith";
if (employee.fullName) {
    alert(employee.fullName);
}
```

While allowing people to randomly set fullName directly is pretty handy, this might get us in trouble if we people can change names on a whim.

In this version, we check to make sure the user has a secret passcode available before we allow them to modify the employee. We do this by replacing the direct access to fullName with a 'set' that will check the passcode. We add a corresponding 'get' to allow the previous example to continue to work seamlessly.

```
var passcode = "secret passcode";

class Employee {
    private _fullName: string;

    get fullName(): string {
        return this._fullName;
    }

    set fullName(newName: string) {
        if (passcode && passcode == "secret passcode") {
            this._fullName = newName;
        }
        else {
            alert("Error: Unauthorized update of employee!");
        }
    }
}

var employee = new Employee();
employee.fullName = "Bob Smith";
if (employee.fullName) {
    alert(employee.fullName);
}
```

To prove to ourselves that our accessor is now checking the passcode, we can modify the passcode and see that when it doesn't match we instead get the alert box warning us we don't have access to update the employee.

Note: Accessors require you to set the compiler to output ECMAScript 5.

Static Properties

Up to this point, we've only talked about the *instance* members of the class, those that show up on the object when its instantiated. We can also create *static* members of a class, those that are visible on the class itself rather than on the instances. In this example, we use 'static' on the origin, as it's a general value for all grids. Each instance accesses this value through prepending the name of the class. Similarly to prepending 'this.' in front of instance accesses, here we prepend 'Grid.' in front of static accesses.

```
class Grid {
    static origin = {x: 0, y: 0};
    calculateDistanceFromOrigin(point: {x: number; y: number;}) {
        var xDist = (point.x - Grid.origin.x);
        var yDist = (point.y - Grid.origin.y);
        return Math.sqrt(xDist * xDist + yDist * yDist) / this.scale;
    }
    constructor (public scale: number) { }
}

var grid1 = new Grid(1.0);  // 1x scale
var grid2 = new Grid(5.0);  // 5x scale

alert(grid1.calculateDistanceFromOrigin({x: 10, y: 10}));
alert(grid2.calculateDistanceFromOrigin({x: 10, y: 10}));
```

Advanced Techniques

Constructor functions

When you declare a class in TypeScript, you are actually creating multiple declarations at the same time. The first is the type of the *instance* of the class.

```
class Greeter {
    greeting: string;
    constructor(message: string) {
        this.greeting = message;
    }
    greet() {
        return "Hello, " + this.greeting;
    }
}

var greeter: Greeter;
greeter = new Greeter("world");
```

```
alert(greeter.greet());
```

Here, when we say 'var greeter: Greeter', we're using Greeter as the type of instances of the class Greeter. This is almost second nature to programmers from other object-oriented languages.

We're also creating another value that we call the *constructor function*. This is the function that is called when we 'new' up instances of the class. To see what this looks like in practice, let's take a look at the JavaScript created by the above example:

```
var Greeter = (function () {
    function Greeter(message) {
        this.greeting = message;
    }
    Greeter.prototype.greet = function () {
        return "Hello, " + this.greeting;
    };
    return Greeter;
})();

var greeter;
greeter = new Greeter("world");
alert(greeter.greet());
```

Here, 'var Greeter' is going to be assigned the constructor function. When we call 'new' and run this function, we get an instance of the class. The constructor function also contains all of the static members of the class. Another way to think of each class is that there is an *instance* side and a *static* side.

Let's modify the example a bit to show this difference:

```
class Greeter {
    static standardGreeting = "Hello, there";
    greeting: string;
    greet() {
        if (this.greeting) {
            return "Hello, " + this.greeting;
        }
        else {
            return Greeter.standardGreeting;
        }
    }
}

var greeter1: Greeter;
greeter1 = new Greeter();
alert(greeter1.greet());

var greeterMaker: typeof Greeter = Greeter;
greeterMaker.standardGreeting = "Hey there!";
var greeter2:Greeter = new greeterMaker();
alert(greeter2.greet());
```

In this example, 'greeter1' works similarly to before. We instantiate the 'Greeter' class, and use this object. This we have seen before.

Next, we then use the class directly. Here we create a new variable called 'greeterMaker'. This variable will hold the class itself, or said another way its constructor function. Here we use 'typeof Greeter', that is "give me the type of the Greeter class itself" rather than the instance type. Or, more precisely, "give me the type of the symbol called Greeter", which is the type of the constructor function. This type will contain all of the static members of Greeter along with the constructor that creates instances of the Greeter class. We show this by using 'new' on 'greeterMaker', creating new instances of 'Greeter' and invoking them as before.

Using a class as an interface

As we said in the previous section, a class declaration creates two things: a type representing instances of the class and a constructor function. Because classes create types, you can use them in the same places you would be able to use interfaces.

```
class Point {
    x: number;
    y: number;
}

interface Point3d extends Point {
    z: number;
}

var point3d: Point3d = {x: 1, y: 2, z: 3};
```

Modules

This post outlines the various ways to organize your code using modules in TypeScript. We'll be covering internal and external modules and we'll discuss when each is appropriate and how to use them. We'll also go over some advanced topics of how to use external modules, and address some common pitfalls when using modules in TypeScript.

First steps

Let's start with the program we'll be using as our example throughout this page. We've written a small set of simplistic string validators, like you might use when checking a user's input on a form in a webpage or checking the format of an externally-provided data file.

Validators in a single file

```
interface StringValidator {
    isAcceptable(s: string): boolean;
}

var lettersRegexp = /^[A-Za-z]+$/;
var numberRegexp = /^[0-9]+$/;

class LettersOnlyValidator implements StringValidator {
    isAcceptable(s: string) {
        return lettersRegexp.test(s);
    }
}

class ZipCodeValidator implements StringValidator {
    isAcceptable(s: string) {
        return s.length === 5 && numberRegexp.test(s);
    }
}

// Some samples to try
var strings = ['Hello', '98052', '101'];
// Validators to use
var validators: { [s: string]: StringValidator; } = {};
validators['ZIP code'] = new ZipCodeValidator();
validators['Letters only'] = new LettersOnlyValidator();
// Show whether each string passed each validator
strings.forEach(s => {
    for (var name in validators) {
        console.log('"' + s + '" ' + (validators[name].isAcceptable(s) ? '
matches ' : ' does not match ') + name);
    }
});
```

Adding Modularity

As we add more validators, we're going to want to have some kind of organization scheme so that we can keep track of our types and not worry about name collisions with other objects. Instead of putting lots of different names into the global namespace, let's wrap up our objects into a module.

In this example, we've moved all the Validator-related types into a module called *Validation*. Because we want the interfaces and classes here to be visible outside the module, we preface them with *export*. Conversely, the variables *lettersRegexp* and *numberRegexp* are implementation details, so they are left unexported and will not be visible to code outside the module. In the test code at the bottom of the file, we now need to qualify the names of the types when used outside the module, e.g. *Validation.LettersOnlyValidator*.

Modularized Validators

```
module Validation {
    export interface StringValidator {
        isAcceptable(s: string): boolean;
    }

    var lettersRegexp = /^[A-Za-z]+$/;
    var numberRegexp = /^[0-9]+$/;

    export class LettersOnlyValidator implements StringValidator {
        isAcceptable(s: string) {
            return lettersRegexp.test(s);
        }
    }

    export class ZipCodeValidator implements StringValidator {
        isAcceptable(s: string) {
            return s.length === 5 && numberRegexp.test(s);
        }
    }
}

// Some samples to try
var strings = ['Hello', '98052', '101'];
// Validators to use
var validators: { [s: string]: Validation.StringValidator; } = {};
validators['ZIP code'] = new Validation.ZipCodeValidator();
validators['Letters only'] = new Validation.LettersOnlyValidator();
// Show whether each string passed each validator
strings.forEach(s => {
    for (var name in validators) {
        console.log('"' + s + '" ' + (validators[name].isAcceptable(s) ? '
matches ' : ' does not match ') + name);
    }
});
```

Splitting Across Files

As our application grows, we'll want to split the code across multiple files to make it easier to maintain.

Here, we've split our Validation module across many files. Even though the files are separate, they can each contribute to the same module and can be consumed as if they were all defined in one place. Because there are dependencies between files, we've added reference tags to tell the compiler about the relationships between the files. Our test code is otherwise unchanged.

Multi-file internal modules

Validation.ts

```
module Validation {
    export interface StringValidator {
        isAcceptable(s: string): boolean;
    }
}
```

LettersOnlyValidator.ts

```
/// <reference path="Validation.ts" />
module Validation {
    var lettersRegexp = /^[A-Za-z]+$/;
    export class LettersOnlyValidator implements StringValidator {
        isAcceptable(s: string) {
            return lettersRegexp.test(s);
        }
    }
}
```

ZipCodeValidator.ts

```
/// <reference path="Validation.ts" />
module Validation {
    var numberRegexp = /^[0-9]+$/;
    export class ZipCodeValidator implements StringValidator {
        isAcceptable(s: string) {
            return s.length === 5 && numberRegexp.test(s);
        }
    }
}
```

Test.ts

```
/// <reference path="Validation.ts" />
/// <reference path="LettersOnlyValidator.ts" />
/// <reference path="ZipCodeValidator.ts" />

// Some samples to try
var strings = ['Hello', '98052', '101'];
// Validators to use
var validators: { [s: string]: Validation.StringValidator; } = {};
validators['ZIP code'] = new Validation.ZipCodeValidator();
validators['Letters only'] = new Validation.LettersOnlyValidator();
// Show whether each string passed each validator
strings.forEach(s => {
    for (var name in validators) {
        console.log('"' + s + '" ' + (validators[name].isAcceptable(s) ? '
matches ' : ' does not match ') + name);
    }
});
```

Once there are multiple files involved, we'll need to make sure all of the compiled code gets loaded. There are two ways of doing this.

First, we can use concatenated output using the --*out* flag to compile all of the input files into a single JavaScript output file:

```
tsc --out sample.js Test.ts
```

The compiler will automatically order the output file based on the reference tags present in the files. You can also specify each file individually:

```
tsc --out sample.js Validation.ts LettersOnlyValidator.ts ZipCodeValidator.ts
Test.ts
```

Alternatively, we can use per-file compilation (the default) to emit one JavaScript file for each input file. If multiple JS files get produced, we'll need to use <*script*> tags on our webpage to load each emitted file in the appropriate order, for example:

MyTestPage.html (excerpt)

```
    <script src="Validation.js" type="text/javascript" />
    <script src="LettersOnlyValidator.js" type="text/javascript" />
    <script src="ZipCodeValidator.js" type="text/javascript" />
    <script src="Test.js" type="text/javascript" />
```

Going External

TypeScript also has the concept of an external module. External modules are used in two cases: node.js and require.js. Applications not using node.js or require.js do not need to use external modules and can best be organized using the internal module concept outlined above.

In external modules, relationships between files are specified in terms of imports and exports at the file level. In TypeScript, any file containing a top-level *import* or *export* is considered an external module.

Below, we have converted the previous example to use external modules. Notice that we no longer use the module keyword – the files themselves constitute a module and are identified by their filenames.

The reference tags have been replaced with *import* statements that specify the dependencies between modules. The *import* statement has two parts: the name that the module will be known by in this file, and the require keyword that specifies the path to the required module:

```
import someMod = require('someModule');
```

We specify which objects are visible outside the module by using the *export* keyword on a top-level declaration, similarly to how *export* defined the public surface area of an internal module.

To compile, we must specify a module target on the command line. For node.js, use --*module commonjs*; for require.js, use --*module amd*. For example:

```
tsc --module commonjs Test.ts
```

When compiled, each external module will become a separate .js file. Similar to reference tags, the compiler will follow *import* statements to compile dependent files.

Validation.ts

```
export interface StringValidator {
    isAcceptable(s: string): boolean;
}
```

LettersOnlyValidator.ts

```
import validation = require('./Validation');
var lettersRegexp = /^[A-Za-z]+$/;
export class LettersOnlyValidator implements validation.StringValidator {
    isAcceptable(s: string) {
        return lettersRegexp.test(s);
    }
}
```

ZipCodeValidator.ts

```
import validation = require('./Validation');
var numberRegexp = /^[0-9]+$/;
export class ZipCodeValidator implements validation.StringValidator {
    isAcceptable(s: string) {
        return s.length === 5 && numberRegexp.test(s);
    }
}
```

Test.ts

```
import validation = require('./Validation');
import zip = require('./ZipCodeValidator');
import letters = require('./LettersOnlyValidator');

// Some samples to try
var strings = ['Hello', '98052', '101'];
// Validators to use
var validators: { [s: string]: validation.StringValidator; } = {};
validators['ZIP code'] = new zip.ZipCodeValidator();
validators['Letters only'] = new letters.LettersOnlyValidator();
// Show whether each string passed each validator
strings.forEach(s => {
    for (var name in validators) {
        console.log('"' + s + '" ' + (validators[name].isAcceptable(s) ? '
matches ' : ' does not match ') + name);
    }
});
```

Code Generation for External Modules

Depending on the module target specified during compilation, the compiler will generate appropriate code for either node.js (commonjs) or require.js (AMD) module-loading systems. For more information on what the *define* and *require* calls in the generated code do, consult the documentation for each module loader.

This simple example shows how the names used during importing and exporting get translated into the module loading code.

SimpleModule.ts

```
import m = require('mod');
export var t = m.something + 1;
```

AMD / RequireJS SimpleModule.js:

```
define(["require", "exports", 'mod'], function(require, exports, m) {
    exports.t = m.something + 1;
});
```

CommonJS / Node SimpleModule.js:

```
var m = require('mod');
exports.t = m.something + 1;
```

Export =

In the previous example, when we consumed each validator, each module only exported one value. In cases like this, it's cumbersome to work with these symbols through their qualified name when a single identifier would do just as well.

The export = syntax specifies a single object that is exported from the module. This can be a class, interface, module, function, or enum. When imported, the exported symbol is consumed directly and is not qualified by any name.

Below, we've simplified the Validator implementations to only export a single object from each module using the export = syntax. This simplifies the consumption code – instead of referring to 'zip.ZipCodeValidator', we can simply refer to 'zipValidator'.

Validation.ts

```
export interface StringValidator {
    isAcceptable(s: string): boolean;
}
```

LettersOnlyValidator.ts

```
import validation = require('./Validation');
var lettersRegexp = /^[A-Za-z]+$/;
class LettersOnlyValidator implements validation.StringValidator {
    isAcceptable(s: string) {
        return lettersRegexp.test(s);
    }
}
export = LettersOnlyValidator;
```

ZipCodeValidator.ts

```
import validation = require('./Validation');
var numberRegexp = /^[0-9]+$/;
class ZipCodeValidator implements validation.StringValidator {
    isAcceptable(s: string) {
        return s.length === 5 && numberRegexp.test(s);
    }
}
export = ZipCodeValidator;
```

Test.ts

```
import validation = require('./Validation');
import zipValidator = require('./ZipCodeValidator');
import lettersValidator = require('./LettersOnlyValidator');

// Some samples to try
var strings = ['Hello', '98052', '101'];
// Validators to use
var validators: { [s: string]: validation.StringValidator; } = {};
validators['ZIP code'] = new zipValidator();
validators['Letters only'] = new lettersValidator();
// Show whether each string passed each validator
strings.forEach(s => {
    for (var name in validators) {
        console.log('"' + s + '" ' + (validators[name].isAcceptable(s) ? '
matches ' : ' does not match ') + name);
    }
});
```

Alias

Another way that you can simplify working with either kind of module is to use *import q = x.y.z* to create shorter names for commonly-used objects. Not to be confused with the *import x = require('name')* syntax used to load external modules, this syntax simply creates an alias for the specified symbol. You can use these sorts of imports (commonly referred to as aliases) for any kind of identifier, including objects created from external module imports.

Basic Aliasing

```
module Shapes {
    export module Polygons {
        export class Triangle { }
        export class Square { }
    }
}

import polygons = Shapes.Polygons;
var sq = new polygons.Square(); // Same as 'new Shapes.Polygons.Square()'
```

Notice that we don't use the *require* keyword; instead we assign directly from the qualified name of the symbol we're importing. This is similar to using *var*, but also works on the type and namespace meanings of the imported symbol. Importantly, for values, *import* is a distinct reference from the original symbol, so changes to an aliased *var* will not be reflected in the original variable.

Optional Module Loading and Other Advanced Loading Scenarios

In some cases, you may want to only load a module under some conditions. In TypeScript, we can use the pattern shown below to implement this and other advanced loading scenarios to directly invoke the module loaders without losing type safety.

The compiler detects whether each module is used in the emitted JavaScript. For modules that are only used as part of the type system, no require calls are emitted. This culling of unused references is a good performance optimization, and also allows for optional loading of those modules.

The core idea of the pattern is that the *import id = require('...')* statement gives us access to the types exposed by the external module. The module loader is invoked (through require) dynamically, as shown in the if blocks below. This leverages the reference-culling optimization so that the module is only loaded when needed. For this pattern to work, it's important that the symbol defined via import is only used in type positions (i.e. never in a position that would be emitted into the JavaScript).

To maintain type safety, we can use the *typeof* keyword. The *typeof* keyword, when used in a type position, produces the type of a value, in this case the type of the external module.

Dynamic Module Loading in node.js

```
declare var require;
import Zip = require('./ZipCodeValidator');
if (needZipValidation) {
    var x: typeof Zip = require('./ZipCodeValidator');
    if (x.isAcceptable('.....')) { /* ... */ }
}
```

Sample: Dynamic Module Loading in require.js

```
declare var require;
import Zip = require('./ZipCodeValidator');
if (needZipValidation) {
    require(['./ZipCodeValidator'], (x: typeof Zip) => {
        if (x.isAcceptable('...')) { /* ... */ }
    });
}
```

Working with Other JavaScript Libraries

To describe the shape of libraries not written in TypeScript, we need to declare the API that the library exposes. Because most JavaScript libraries expose only a few top-level objects, modules are a good way to represent them. We call declarations that don't define an implementation "ambient". Typically these are defined in .d.ts files. If you're familiar with C/C++, you can think of these as .h files or 'extern'. Let's look at a few examples with both internal and external examples.

Ambient Internal Modules

The popular library D3 defines its functionality in a global object called 'D3'. Because this library is loaded through a *script* tag (instead of a module loader), its declaration uses internal modules to define its shape. For the TypeScript compiler to see this shape, we use an ambient internal module declaration. For example:

D3.d.ts (simplified excerpt)

```
declare module D3 {
    export interface Selectors {
        select: {
            (selector: string): Selection;
            (element: EventTarget): Selection;
        };
    }

    export interface Event {
        x: number;
        y: number;
    }

    export interface Base extends Selectors {
        event: Event;
    }
}

declare var d3: D3.Base;
```

Ambient External Modules

In node.js, most tasks are accomplished by loading one or more modules. We could define each module in its own .d.ts file with top-level export declarations, but it's more convenient to write them as one larger .d.ts file. To do so, we use the quoted name of the module, which will be available to a later import. For example:

node.d.ts (simplified excerpt)

```
declare module "url" {
    export interface Url {
        protocol?: string;
        hostname?: string;
        pathname?: string;
    }

    export function parse(urlStr: string, parseQueryString?,
slashesDenoteHost?): Url;
}

declare module "path" {
    export function normalize(p: string): string;
    export function join(...paths: any[]): string;
    export var sep: string;
}
```

Now we can */// <reference>* node.d.ts and then load the modules using e.g. *import url = require('url');*.

```
///<reference path="node.d.ts"/>
import url = require("url");
var myUrl = url.parse("http://www.typescriptlang.org");
```

Pitfalls of Modules

In this section we'll describe various common pitfalls in using internal and external modules, and how to avoid them.

/// <reference> to an external module

A common mistake is to try to use the /// <reference> syntax to refer to an external module file, rather than using import. To understand the distinction, we first need to understand the three ways that the compiler can locate the type information for an external module.

The first is by finding a .ts file named by an *import x = require(...);* declaration. That file should be an implementation file with top-level import or export declarations.

The second is by finding a .d.ts file, similar to above, except that instead of being an implementation file, it's a declaration file (also with top-level import or export declarations).

The final way is by seeing an "ambient external module declaration", where we 'declare' a module with a matching quoted name.

myModules.d.ts

```
// In a .d.ts file or .ts file that is not an external module:
declare module "SomeModule" {
    export function fn(): string;
}
```

myOtherModule.ts

```
/// <reference path="myModules.d.ts" />
import m = require("SomeModule");
```

The reference tag here allows us to locate the declaration file that contains the declaration for the ambient external module. This is how the node.d.ts file that several of the TypeScript samples use is consumed, for example.

Needless Namespacing

If you're converting a program from internal modules to external modules, it can be easy to end up with a file that looks like this:

shapes.ts

```
export module Shapes {
    export class Triangle { /* ... */ }
    export class Square { /* ... */ }
}
```

The top-level module here *Shapes* wraps up *Triangle* and *Square* for no reason. This is confusing and annoying for consumers of your module:

shapeConsumer.ts

```
import shapes = require('./shapes');
var t = new shapes.Shapes.Triangle(); // shapes.Shapes?
```

A key feature of external modules in TypeScript is that two different external modules will never contribute names to the same scope. Because the consumer of an external module decides what name to assign it, there's no need to proactively wrap up the exported symbols in a namespace.

To reiterate why you shouldn't try to namespace your external module contents, the general idea of namespacing is to provide logical grouping of constructs and to prevent name collisions. Because the external module file itself is already a logical grouping, and its top-level name is defined by the code that imports it, it's unnecessary to use an additional module layer for exported objects.

Revised Example:

shapes.ts

```
export class Triangle { /* ... */ }
export class Square { /* ... */ }
```

shapeConsumer.ts

```
import shapes = require('./shapes');
var t = new shapes.Triangle();
```

Trade-offs for External Modules

Just as there is a one-to-one correspondence between JS files and modules, TypeScript has a one-to-one correspondence between external module source files and their emitted JS files. One effect of this is that it's not possible to use the --*out* compiler switch to concatenate multiple external module source files into a single JavaScript file.

Functions

Functions are the fundamental building block of any applications in JavaScript. They're how you build up layers of abstraction, mimicking classes, information hiding, and modules. In TypeScript, while there are classes and modules, function still play the key role in describing how to 'do' things. TypeScript also adds some new capabilities to the standard JavaScript functions to make them easier to work with.

Functions

To begin, just as in JavaScript, TypeScript functions can be created both as a named function or as an anonymous function. This allows you to choose the most appropriate approach for your application, whether you're building a list of functions in an API or a one-off function to hand off to another function.

To quickly recap what these two approaches look like in JavaScript:

```
//Named function
function add(x, y) {
    return x+y;
}

//Anonymous function
var myAdd = function(x, y) { return x+y; };
```

Just as in JavaScript, functions can return to variables outside of the function body. When they do so, they're said to 'capture' these variables. While understanding how this works, and the trade-offs when using this technique, are outside of the scope of this article, having a firm understanding how this mechanic is an important piece of working with JavaScript and TypeScript.

```
var z = 100;

function addToZ(x, y) {
    return x+y+z;
}
```

Function Types

Typing the function

Let's add types to our simple examples from earlier:

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

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

We can add types to each of the parameters and then to the function itself to add a return type. TypeScript can figure the return type out by looking at the return statements, so we can also optionally leave this off in many cases.

Writing the function type

Now that we've typed the function, let's write the full type of the function out by looking at the each piece of the function type.

```
var myAdd: (x:number, y:number)=>number =
    function(x: number, y: number): number { return x+y; };
```

A function's type has the same two parts: the type of the arguments and the return type. When writing out the whole function type, both parts are required. We write out the parameter types just like a parameter list, giving each parameter a name and a type. This name is just to help with readability. We could have instead written:

```
var myAdd: (baseValue:number, increment:number)=>number =
    function(x: number, y: number): number { return x+y; };
```

As long as the parameter types line up, it's considered a valid type for the function, regardless of the names you give the parameters in the function type.

The second part is the return type. We make it clear which is the return type by using a fat arrow (=>) between the parameters and the return type. As mentioned before, this is a required part of the function type, so if the function doesn't return a value, you would use 'void' instead of leaving it off.

Of note, only the parameters and the return type make up the function type. Captured variables are not reflected in the type. In effect, captured variables are part of the 'hidden state' of any function and do not make up its API.

Inferring the types

In playing with the example, you may notice that the TypeScript compiler can figure out the type if you have types on one side of the equation but not the other:

```
// myAdd has the full function type
var myAdd = function(x: number, y: number): number { return x+y; };

// The parameters 'x' and 'y' have the type number
var myAdd: (baseValue:number, increment:number)=>number =
    function(x, y) { return x+y; };
```

This is called 'contextual typing', a form of type inference. This helps cut down on the amount of effort to keep your program typed.

Optional and Default Parameters

Unlike JavaScript, in TypeScript every parameter to a function is assumed to be required by the function. This doesn't mean that it isn't a 'null' value, but rather, when the function is called the compiler will check that the user has provided a value for each parameter. The compiler also assumes that these parameters are the only parameters that will be passed to the function. In short, the number of parameters to the function has to match the number of parameters the function expects.

```
function buildName(firstName: string, lastName: string) {
    return firstName + " " + lastName;
}

var result1 = buildName("Bob");  //error, too few parameters
var result2 = buildName("Bob", "Adams", "Sr.");  //error, too many parameters
var result3 = buildName("Bob", "Adams");  //ah, just right
```

In JavaScript, every parameter is considered optional, and users may leave them off as they see fit. When they do, they're assumed to be undefined. We can get this functionality in TypeScript by using the '?' beside parameters we want optional. For example, let's say we want the last name to be optional:

```
function buildName(firstName: string, lastName?: string) {
    if (lastName)
        return firstName + " " + lastName;
    else
        return firstName;
}

var result1 = buildName("Bob");  //works correctly now
var result2 = buildName("Bob", "Adams", "Sr.");  //error, too many parameters
var result3 = buildName("Bob", "Adams");  //ah, just right
```

Optional parameters must follow required parameters. Had we wanted to make the first name optional rather than the last name, we would need to change the order of parameters in the function, putting the first name last in the list.

In TypeScript, we can also set up a value that an optional parameter will have if the user does not provide one. These are called default parameters. Let's take the previous example and default the last name to "Smith".

```
function buildName(firstName: string, lastName = "Smith") {
    return firstName + " " + lastName;
}

var result1 = buildName("Bob");  //works correctly now, also
var result2 = buildName("Bob", "Adams", "Sr.");  //error, too many parameters
var result3 = buildName("Bob", "Adams");  //ah, just right
```

Just as with optional parameters, default parameters must come after required parameters in the parameter list.

Optional parameters and default parameters also share what the type looks like. Both:

```
function buildName(firstName: string, lastName?: string) {
```

and

```
function buildName(firstName: string, lastName = "Smith") {
```

share the same type "(firstName: string, lastName?: string)=>string". The default value of the default parameter disappears, leaving only the knowledge that the parameter is optional.

Rest Parameters

Required, optional, and default parameters all have one thing in common: they're about talking about one parameter at a time. Sometimes, you want to work with multiple parameters as a group, or you may not know how many parameters a function will ultimately take. In JavaScript, you can work with the arguments direction using the arguments variable that is visible inside every function body.

In TypeScript, you can gather these arguments together into a variable:

```
function buildName(firstName: string, ...restOfName: string[]) {
        return firstName + " " + restOfName.join(" ");
}

var employeeName = buildName("Joseph", "Samuel", "Lucas", "MacKinzie");
```

Rest parameters are treated as a boundless number of optional parameters. You may leave them off, or have as many as you want. The compiler will build an array of the arguments you pass to the function under the name given after the ellipsis (...), allowing you to use it in your function.

The ellipsis is also used in the type of the function with rest parameters:

```
function buildName(firstName: string, ...restOfName: string[]) {
        return firstName + " " + restOfName.join(" ");
}

var buildNameFun: (fname: string, ...rest: string[])=>string = buildName;
```

Lambdas and using 'this'

How 'this' works in JavaScript functions is a common theme in programmers coming to JavaScript. Indeed, learning how to use it is something of a rite of passage as developers become more accustomed to working in JavaScript. Since TypeScript is a superset of JavaScript, TypeScript developers also need to learn how to use 'this' and how to spot when it's not being used correctly. A whole article could be written on how to use 'this' in JavaScript, and many have. Here, we'll focus on some of the basics.

In JavaScript, 'this' is a variable that's set when a function is called. This makes it a very powerful and flexible feature, but it comes at the cost of always having to know about the context that a function is executing in. This can be notoriously confusing, when, for example, when a function is used as a callback.

Let's look at an example:

```
var deck = {
    suits: ["hearts", "spades", "clubs", "diamonds"],
    cards: Array(52),
    createCardPicker: function() {
        return function() {
            var pickedCard = Math.floor(Math.random() * 52);
            var pickedSuit = Math.floor(pickedCard / 13);

            return {suit: this.suits[pickedSuit], card: pickedCard % 13};
        }
    }
}

var cardPicker = deck.createCardPicker();
var pickedCard = cardPicker();

alert("card: " + pickedCard.card + " of " + pickedCard.suit);
```

If we tried to run the example, we would get an error instead of the expected alert box. This is because the 'this' being used in the function created by 'createCardPicker' will be set to 'window' instead of our 'deck' object. This happens as a result of calling 'cardPicker()'. Here, there is no dynamic binding for 'this' other than Window. (note: under strict mode, this will be undefined rather than window).

We can fix this by making sure the function is bound to the correct 'this' before we return the function to be used later. This way, regardless of how its later used, it will still be able to see the original 'deck' object.

To fix this, we switching the function expression to use the lambda syntax (()=>{}) rather than the JavaScript function expression. This will automatically capture the 'this' available when the function is created rather than when it is invoked:

```
var deck = {
    suits: ["hearts", "spades", "clubs", "diamonds"],
    cards: Array(52),
    createCardPicker: function() {
        // Notice: the line below is now a lambda, allowing us to capture
'this' earlier
        return () => {
            var pickedCard = Math.floor(Math.random() * 52);
            var pickedSuit = Math.floor(pickedCard / 13);

            return {suit: this.suits[pickedSuit], card: pickedCard % 13};
        }
    }
}

var cardPicker = deck.createCardPicker();
var pickedCard = cardPicker();

alert("card: " + pickedCard.card + " of " + pickedCard.suit);
```

For more information on ways to think about 'this', you can read Yahuda Katz's Understanding JavaScript Function Invocation and “this”.

Overloads

JavaScript is inherently a very dynamic language. It's not uncommon for a single JavaScript function to return different types of objects based on the shape of the arguments passed in.

```
var suits = ["hearts", "spades", "clubs", "diamonds"];

function pickCard(x): any {
    // Check to see if we're working with an object/array
    // if so, they gave us the deck and we'll pick the card
    if (typeof x == "object") {
        var pickedCard = Math.floor(Math.random() * x.length);
        return pickedCard;
    }
    // Otherwise just let them pick the card
    else if (typeof x == "number") {
        var pickedSuit = Math.floor(x / 13);
        return { suit: suits[pickedSuit], card: x % 13 };
    }
}

var myDeck = [{ suit: "diamonds", card: 2 }, { suit: "spades", card: 10 }, {
suit: "hearts", card: 4 }];
var pickedCard1 = myDeck[pickCard(myDeck)];
alert("card: " + pickedCard1.card + " of " + pickedCard1.suit);

var pickedCard2 = pickCard(15);
alert("card: " + pickedCard2.card + " of " + pickedCard2.suit);
```

Here the 'pickCard' function will return two different things based on what the user has passed in. If the users passes in an object that represents the deck, the function will pick the card. If the user picks the card, we tell them which card they've picked. But how do we describe this to the type system.

The answer is to supply multiple function types for the same function as a list of overloads. This list is what the compiler will use to resolve function calls. Let's create a list of overloads that describe what our 'pickCard' accepts and what it returns.

```
var suits = ["hearts", "spades", "clubs", "diamonds"];

function pickCard(x: {suit: string; card: number; }[]): number;
function pickCard(x: number): {suit: string; card: number; };
function pickCard(x): any {
    // Check to see if we're working with an object/array
    // if so, they gave us the deck and we'll pick the card
    if (typeof x == "object") {
        var pickedCard = Math.floor(Math.random() * x.length);
        return pickedCard;
    }
    // Otherwise just let them pick the card
    else if (typeof x == "number") {
        var pickedSuit = Math.floor(x / 13);
        return { suit: suits[pickedSuit], card: x % 13 };
    }
}

var myDeck = [{ suit: "diamonds", card: 2 }, { suit: "spades", card: 10 }, {
suit: "hearts", card: 4 }];
var pickedCard1 = myDeck[pickCard(myDeck)];
alert("card: " + pickedCard1.card + " of " + pickedCard1.suit);

var pickedCard2 = pickCard(15);
alert("card: " + pickedCard2.card + " of " + pickedCard2.suit);
```

With this change, the overloads now give us type-checked calls to the 'pickCard' function.

In order for the compiler to pick the correct typecheck, it follows a similar process to the underlying JavaScript. It looks at the overload list, and proceeding with the first overload attempts to call the function with the provided parameters. If it finds a match, it picks this overload as the correct overload. For this reason, its customary to order overloads from most specific to least specific.

Note that the 'function pickCard(x): any' piece is not part of the overload list, so it only has two overloads: one that takes an object and one that takes a number. Calling 'pickCard' with any other parameter types would cause an error.

Generics

A major part of software engineering is building components that not only have well-defined and consistent APIs, but are also reusable. Components that are capable of working on the data of today as well as the data of tomorrow will give you the most flexible capabilities for building up large software systems.

In languages like C# and Java, one of the main tools in the toolbox for creating reusable components is 'generics', that is, being able to create a component that can work over a variety of types rather than a single one. This allows users to consume these components and use their own types.

Hello World of Generics

To start off, let's do the "hello world" of generics: the identity function. The identity function is a function that will return back whatever is passed in. You can think of this in a similar way to the 'echo' command.

Without generics, we would either have to give the identity function a specific type:

```
function identity(arg: number): number {
    return arg;
}
```

Or, we could describe the identity function using the 'any' type:

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

While using 'any' is certainly generic in that will accept any and all types for the type of 'arg', we actually are losing the information about what that type was when the function returns. If we passed in a number, the only information we have is that any type could be returned.

Instead, we need a way of capturing the type of the argument in such a way that we can also use it to denote what is being returned. Here, we will use a *type variable*, a special kind of variable that works on types rather than values.

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

We've now added a type variable 'T' to the identity function. This 'T' allows us to capture the type the user provides (eg, number), so that we can use that information later. Here, we use 'T' again as the return type. On inspection, we can now see the same type is used for the argument and the return type. This allows us to traffic that type information in one side of the function and out the other.

We say that this version of the 'identity' function is generic, as it works over a range of types. Unlike using 'any', it's also just as precise (ie, it doesn't lose any information) as the first 'identity' function that used numbers for the argument and return type.

Once we've written the generic identity function, we can call it in one of two ways. The first way is to pass all of the arguments, including the type argument, to the function:

```
var output = identity<string>("myString");  // type of output will be
'string'
```

Here we explicitly set 'T' to be string as one of the arguments to the function call, denoted using the <> around the arguments rather than ().

The second way is also perhaps the most common. Here we use /type argument inference/, that is, we want the compiler to set the value of T for us automatically based on the type of the argument we pass in:

```
var output = identity("myString");  // type of output will be 'string'
```

Notice that we didn't have explicitly pass the type in the angle brackets (<>), the compiler just looked at the value "myString", and set T to its type. While type argument inference can be a helpful tool to keep code shorter and more readable, you may need to explicitly pass in the type arguments as we did in the previous example when the compiler fails to infer the type, as may happen in more complex examples.

Working with Generic Type Variables

When you begin to use generics, you'll notice that when you create generic functions like 'identity', the compiler will enforce that you use any generically typed parameters in the body of the function correctly. That is, that you actually treat these parameters as if they could be any and all types.

Let's take our 'identity' function from earlier:

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

What if want to also log the length of the argument 'arg' to the console with each call. We might be tempted to write this:

```
function loggingIdentity<T>(arg: T): T {
    console.log(arg.length);  // Error: T doesn't have .length
    return arg;
}
```

When we do, the compiler will give us an error that we're using the ".length" member of 'arg', but nowhere have we said that 'arg' has this member. Remember, we said earlier that these type variables stand in for any and all types, so someone using this function could have passed in a 'number' instead, which does not have a ".length" member.

Let's say that we've actually intended this function to work on arrays of T rather that T directly. Since we're working with arrays, the .length member should be available. We can describe this just like we would create arrays of other types:

```
function loggingIdentity<T>(arg: T[]): T[] {
    console.log(arg.length);  // Array has a .length, so no more error
    return arg;
}
```

You can read the type of logging Identity as "the generic function loggingIdentity, takes a type parameter T, and an argument 'arg' which is an array of these T's, and returns an array of T's. If we passed in an array of numbers, we'd get an array of numbers back out, as T would bind to number. This allows us to use our generic type variable 'T' as part of the types we're working with, rather than the whole type, giving us greater flexibility.

We can alternatively write the sample example this way:

```
function loggingIdentity<T>(arg: Array<T>): Array<T> {
    console.log(arg.length);  // Array has a .length, so no more error
    return arg;
}
```

You may already be familiar with this style of type from other languages. In the next section, we'll cover how you can create your own generic types like Array<T>.

Generic Types

In previous sections, we created generic identity functions that worked over a range of types. In this section, we'll explore the type of the functions themselves and how to create generic interfaces.

The type of generic functions is just like those of non-generic functions, with the type parameters listed first, similarly to function declarations:

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

var myIdentity: <T>(arg: T)=>T = identity;
```

We could also have used a different name for the generic type parameter in the type, so long as the number of type variables and how the type variables are used line up.

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

var myIdentity: <U>(arg: U)=>U = identity;
```

We can also write the generic type as a call signature of an object literal type:

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

var myIdentity: {<T>(arg: T): T} = identity;
```

Which leads us to writing our first generic interface. Let's take the object literal from the previous example and move it to an interface:

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

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

var myIdentity: GenericIdentityFn = identity;
```

In a similar example, we may want to move the generic parameter to be a parameter of the whole interface. This lets us see what type(s) we're generic over (eg Dictionary<string> rather than just Dictionary). This makes the type parameter visible to all the other members of the interface.

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

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

var myIdentity: GenericIdentityFn<number> = identity;
```

Notice that our example has changed to be something slightly different. Instead of describing a generic function, we now have a non-generic function signature that is a part of a generic type. When we use GenericIdentityFn, we now will also need to specify the corresponding type argument (here: number), effectively locking in what the underlying call signature will use. Understanding when to put the type parameter directly on the call signature and when to put it on the interface itself will be helpful in describing what aspects of a type are generic.

In addition to generic interfaces, we can also create generic classes. Note that it is not possible to create generic enums and modules.

Generic Classes

A generic class has a similar shape to a generic interface. Generic classes have a generic type parameter list in angle brackets (<>) following the name of the class.

```
class GenericNumber<T> {
    zeroValue: T;
    add: (x: T, y: T) => T;
}

var myGenericNumber = new GenericNumber<number>();
myGenericNumber.zeroValue = 0;
myGenericNumber.add = function(x, y) { return x + y; };
```

This is a pretty literal use of the 'GenericNumber' class, but you may have noticed that nothing is restricting is to only use the 'number' type. We could have instead used 'string' or even more complex objects.

```
var stringNumeric = new GenericNumber<string>();
stringNumeric.zeroValue = "";
stringNumeric.add = function(x, y) { return x + y; };

alert(stringNumeric.add(stringNumeric.zeroValue, "test"));
```

Just as with interface, putting the type parameter on the class itself lets us make sure all of the properties of the class are working with the same type.

As we covered in Classes, a class has two side to its type: the static side and the instance side. Generic classes are only generic over their instance side rather than their static side, so when working with classes, static members can not use the class's type parameter.

Generic Constraints

If you remember from an earlier example, you may sometimes want to write a generic function that works on a set of types where you have some knowledge about what capabilities that set of types will have. In our 'loggingIdentity' example, we wanted to be able access the ".length" property of 'arg', but the compiler could not prove that every type had a ".length" property, so it warns us that we can't make this assumption.

```
function loggingIdentity<T>(arg: T): T {
    console.log(arg.length);  // Error: T doesn't have .length
    return arg;
}
```

Instead of working with any and all types, we'd like to constrain this function to work with any and all types that also have the ".length" property. As long as the type has this member, we'll allow it, but it's required to have at least this member. To do so, we must list our requirement as a constraint on what T can be.

To do so, we'll create an interface that describes our constraint. Here, we'll create an interface that has a single ".length" property and then we'll use this interface and the extends keyword to denote our constraint:

```
interface Lengthwise {
    length: number;
}

function loggingIdentity<T extends Lengthwise>(arg: T): T {
    console.log(arg.length);  // Now we know it has a .length property, so no
more error
    return arg;
}
```

Because the generic function is now constrained, it will no longer work over any and all types:

```
loggingIdentity(3);  // Error, number doesn't have a .length property
```

Instead, we need to pass in values whose type has all the required properties:

```
loggingIdentity({length: 10, value: 3});
```

Using Type Parameters in Generic Constraints

In some cases, it may be useful to declare a type parameter that is constrained by another type parameter. For example,

```
function find<T, U extends Findable<T>>(n: T, s: U) {    // errors because
type parameter used in constraint
  // ...
}
find (giraffe, myAnimals);
```

You can achieve the pattern above by replacing the type parameter with its constraint. Rewriting the example above,

```
function find<T>(n: T, s: Findable<T>) {
  // ...
}
find(giraffe, myAnimals);
```

Note: The above is not strictly identical, as the return type of the first function could have returned 'U', which the second function pattern does not provide a means to do.

Using Class Types in Generics

When creating factories in TypeScript using generics, it is necessary to refer to class types by their constructor functions. For example,

```
function create<T>(c: {new(): T; }): T {
    return new c();
}
```

A more advanced example uses the prototype property to infer and constrain relationships between the constructor function and the instance side of class types.

```
class BeeKeeper {
    hasMask: boolean;
}

class ZooKeeper {
    nametag: string;
}

class Animal {
    numLegs: number;
}

class Bee extends Animal {
    keeper: BeeKeeper;
}

class Lion extends Animal {
    keeper: ZooKeeper;
}

function findKeeper<A extends Animal, K> (a: {new(): A;
    prototype: {keeper: K}}): K {
```

```
    return a.prototype.keeper;
}

findKeeper(Lion).nametag;  // typechecks!
```

Common Errors

The list below captures some of the commonly confusing error messages that you may encounter when using the TypeScript language and Compiler

Commonly Confusing Errors

"tsc.exe" exited with error code 1.

Fixes:

- check file-encoding is UTF-8 - https://typescript.codeplex.com/workitem/1587

external module XYZ cannot be resolved

Fixes:

- check if module path is case-sensitive - https://typescript.codeplex.com/workitem/2134

Mixins

Along with traditional OO hierarchies, another popular way of building up classes from reusable components is to build them by combining simpler partial classes. You may be familiar with the idea of mixins or traits for languages like Scala, and the pattern has also reached some popularity in the JavaScript community.

Mixin sample

In the code below, we show how you can model mixins in TypeScript. After the code, we'll break down how it works.

```
// Disposable Mixin
class Disposable {
    isDisposed: boolean;
    dispose() {
        this.isDisposed = true;
    }

}

// Activatable Mixin
class Activatable {
    isActive: boolean;
    activate() {
        this.isActive = true;
    }
    deactivate() {
        this.isActive = false;
    }
}

class SmartObject implements Disposable, Activatable {
    constructor() {
        setInterval(() => console.log(this.isActive + " : " +
this.isDisposed), 500);
    }

    interact() {
        this.activate();
    }

    // Disposable
    isDisposed: boolean = false;
    dispose: () => void;
    // Activatable
    isActive: boolean = false;
    activate: () => void;
    deactivate: () => void;
}
applyMixins(SmartObject, [Disposable, Activatable])

var smartObj = new SmartObject();
setTimeout(() => smartObj.interact(), 1000);

////////////////////////////////////////
// In your runtime library somewhere
////////////////////////////////////////

function applyMixins(derivedCtor: any, baseCtors: any[]) {
    baseCtors.forEach(baseCtor => {
        Object.getOwnPropertyNames(baseCtor.prototype).forEach(name => {
            derivedCtor.prototype[name] = baseCtor.prototype[name];
        })
    });
}
```

Understanding the sample

The code sample starts with the two classes that will act is our mixins. You can see each one is focused on a particular activity or capability. We'll later mix these together to form a new class from both capabilities.

```
// Disposable Mixin
class Disposable {
    isDisposed: boolean;
    dispose() {
        this.isDisposed = true;
    }

}

// Activatable Mixin
class Activatable {
    isActive: boolean;
    activate() {
        this.isActive = true;
    }
    deactivate() {
        this.isActive = false;
    }
}
```

Next, we'll create the class that will handle the combination of the two mixins. Let's look at this in more detail to see how it does this:

```
class SmartObject implements Disposable, Activatable {
```

The first thing you may notice in the above is that instead of using 'extends', we use 'implements'. This treats the classes as interfaces, and only uses the types behind Disposable and Activatable rather than the implementation. This means that we'll have to provide the implementation in class. Except, that's exactly what we want to avoid by using mixins.

To satisfy this requirement, we create stand-in properties and their types for the members that will come from our mixins. This satisfies the compiler that these members will be available at runtime. This lets us still get the benefit of the mixins, albeit with a some bookkeeping overhead.

```
// Disposable
isDisposed: boolean = false;
dispose: () => void;
// Activatable
isActive: boolean = false;
activate: () => void;
deactivate: () => void;
```

Finally, we mix our mixins into the class, creating the full implementation.

```
applyMixins(SmartObject, [Disposable, Activatable])
```

Lastly, we create a helper function that will do the mixing for us. This will run through the properties of each of the mixins and copy them over to the target of the mixins, filling out the stand-in properties with their implementations.

```
function applyMixins(derivedCtor: any, baseCtors: any[]) {
    baseCtors.forEach(baseCtor => {
        Object.getOwnPropertyNames(baseCtor.prototype).forEach(name => {
            derivedCtor.prototype[name] = baseCtor.prototype[name];
        })
    });
}
```

Declaration Merging

Some of the unique concepts in TypeScript come from the need to describe what is happening to the shape of JavaScript objects at the type level. One example that is especially unique to TypeScript is the concept of 'declaration merging'. Understanding this concept will give you an advantage when working with existing JavaScript in your TypeScript. It also opens the door to more advanced abstraction concepts.

First, before we get into how declarations merge, let's first describe what we mean by 'declaration merging'.

For the purposes of this article, declaration merging specifically means that the compiler is doing the work of merging two separate declarations declared with the same name into a single definition. This merged definition has the features of both of the original declarations. Declaration merging is not limited to just two declarations, as any number of declarations can be merged.

Basic Concepts

In TypeScript, a declaration exists in one of three groups: namespace/module, type, or value. Declarations that create a namespace/module are accessed using a dotted notation when writing a type. Declarations that create a type do just that, create a type that is visible with the declared shape and bound to the given name. Lastly, declarations create a value are those that are visible in the output JavaScript (eg, functions and variables).

Declaration Type	Namespace	Type	Value
Module	X		X
Class		X	X
Interface		X	

Function	X
Variable	X

Understanding what is created with each declaration will help you understand what is merged when you perform a declaration merge.

Merging Interfaces

The simplest, and perhaps most common, type of declaration merging is interface merging. At the most basic level, the merge mechanically joins the members of both declarations into a single interface with the same name.

```
interface Box {
    height: number;
    width: number;
}

interface Box {
    scale: number;
}

var box: Box = {height: 5, width: 6, scale: 10};
```

Non-function members of the interfaces must be unique. The compiler will issue an error if the interfaces both declare a non-function member of the same name.

For function members, each function member of the same name is treated as describing an overload of the same function. Of note, too, is that in the case of interface A merging with later interface A (here called A'), the overload set of A' will have a higher precedence than that of interface A.

That is, in the example:

```
interface Document {
    createElement(tagName: any): Element;
}
interface Document {
    createElement(tagName: string): HTMLElement;
}
interface Document {
    createElement(tagName: "div"): HTMLDivElement;
    createElement(tagName: "span"): HTMLSpanElement;
    createElement(tagName: "canvas"): HTMLCanvasElement;
}
```

The two interfaces will merge to create a single declaration. Notice that the elements of each group maintains the same order, just the groups themselves are merged with later overload sets coming first:

```
interface Document {
    createElement(tagName: "div"): HTMLDivElement;
    createElement(tagName: "span"): HTMLSpanElement;
    createElement(tagName: "canvas"): HTMLCanvasElement;
    createElement(tagName: string): HTMLElement;
    createElement(tagName: any): Element;
}
```

Merging Modules

Similarly to interfaces, modules of the same name will also merge their members. Since modules create both a namespace and a value, we need to understand how both merge.

To merge the namespaces, type definitions from exported interfaces declared in each module are themselves merged, forming a single namespace with merged interface definitions inside.

To merge the value, at each declaration site, if a module already exists with the given name, it is further extended by taking the existing module and adding the exported members of the second module to the first.

The declaration merge of 'Animals' in this example:

```
module Animals {
    export class Zebra { }
}

module Animals {
    export interface Legged { numberOfLegs: number; }
    export class Dog { }
}
```

is equivalent to:

```
module Animals {
    export interface Legged { numberOfLegs: number; }

    export class Zebra { }
    export class Dog { }
}
```

This model of module merging is a helpful starting place, but to get a more complete picture we need to also understand what happens with non-exported members. Non-exported members are only visible in the original (un-merged) module. This means that after merging, merged members that came from other declarations can not see non-exported members.

We can see this more clearly in this example:

```
module Animal {
    var haveMuscles = true;

    export function animalsHaveMuscles() {
        return haveMuscles;
    }
}

module Animal {
    export function doAnimalsHaveMuscles() {
        return haveMuscles;  // <-- error, haveMuscles is not visible here
    }
}
```

Because haveMuscles is not exported, only the animalsHaveMuscles function that shares the same un-merged module can see the symbol. The doAnimalsHaveMuscles function, even though it's part of the merged Animal module can not see this un-exported member.

Merging Modules with Classes, Functions, and Enums

Modules are flexible enough to also merge with other types of declarations. To do so, the module declaration must follow the declaration it will merge with. The resulting declaration has properties of both declaration types. TypeScript uses this capability to model some of patterns in JavaScript as well as other programming languages.

The first module merge we'll cover is merging a module with a class. This gives the user a way of describing inner classes.

```
class Album {
    label: Album.AlbumLabel;
}
module Album {
    export class AlbumLabel { }
}
```

The visibility rules for merged members is the same as described in the 'Merging Modules' section, so we must export the AlbumLabel class for the merged class to see it. The end result is a class managed inside of another class. You can also use modules to add more static members to an existing class.

In addition to the pattern of inner classes, you may also be familiar with JavaScript practice of creating a function and then extending the function further by adding properties onto the function. TypeScript uses declaration merging to build up definitions like this in a type-safe way.

```
function buildLabel(name: string): string {
    return buildLabel.prefix + name + buildLabel.suffix;
}

module buildLabel {
    export var suffix = "";
    export var prefix = "Hello, ";
}

alert(buildLabel("Sam Smith"));
```

Similarly, modules can be used to extend enums with static members:

```
enum Color {
    red = 1,
    green = 2,
    blue = 4
}

module Color {
    export function mixColor(colorName: string) {
        if (colorName == "yellow") {
            return Color.red + Color.green;
        }
        else if (colorName == "white") {
            return Color.red + Color.green + Color.blue;
        }
        else if (colorName == "magenta") {
            return Color.red + Color.blue;
        }
        else if (colorName == "cyan") {
            return Color.green + Color.blue;
        }
    }
}
```

Disallowed Merges

Not all merges are allowed in TypeScript. Currently, classes can not merge with other classes, variables and classes can not merge, nor can interfaces and classes. For information on mimicking classes merging, see the Mixins in TypeScript section.

Type Inference

In this section, we will cover type inference in TypeScript. Namely, we'll discuss where and how types are inferred.

Basics

In TypeScript, there are several places where type inference is used to provide type information when there is no explicit type annotation. For example, in this code

```
var x = 3;
```

The type of the x variable is inferred to be number. This kind of inference takes place when initializing variables and members, setting parameter default values, and determining function return types.

In most cases, type inference is straightforward. In the following sections, we'll explore some of the nuance in how types are inferred.

Best common type

When a type inference is made from several expressions, the types of those expressions are used to calculate a "best common type". For example,

```
var x = [0, 1, null];
```

To infer the type of x in the example above, we must consider the type of each array element. Here we are given two choices for the type of the array: number and null. The best common type algorithm considers each candidate type, and picks the type that is compatible with all the other candidates.

Because the best common type has to be chosen from the provided candidate types, there are some cases where types share a common structure, but no one type is the super type of all candidate types. For example:

```
var zoo = [new Rhino(), new Elephant(), new Snake()];
```

Ideally, we may want zoo to be inferred as an Animal[], but because there is no object that is strictly of type Animal in the array, we make no inference about the array element type. To correct this, instead explicitly provide the type when no one type is a super type of all other candidates:

```
var zoo: Animal[] = [new Rhino(), new Elephant(), new Snake()];
```

When no best common type is found, the resulting inference is the empty object type, {}. Because this type has no members, attempting to use any properties of it will cause an error. This result allows you to still use the object in a type-agnostic manner, while providing type safety in cases where the type of the object can't be implicitly determined.

Contextual Type

Type inference also works in "the other direction" in some cases in TypeScript. This is known as "contextual typing". Contextual typing occurs when the type of an expression is implied by its location. For example:

```
window.onmousedown = function(mouseEvent) {
    console.log(mouseEvent.buton);  //<- Error
};
```

For the code above to give the type error, the TypeScript type checker used the type of the Window.onmousedown function to infer the type of the function expression on the right hand side of the assignment. When it did so, it was able to infer the type of the mouseEvent parameter. If this function expression were not in a contextually typed position, the mouseEvent parameter would have type any, and no error would have been issued.

If the contextually typed expression contains explicit type information, the contextual type is ignored. Had we written the above example:

```
window.onmousedown = function(mouseEvent: any) {
    console.log(mouseEvent.buton);  //<- Now, no error is given
};
```

The function expression with an explicit type annotation on the parameter will override the contextual type. Once it does so, no error is given as no contextual type applies.

Contextual typing applies in many cases. Common cases include arguments to function calls, right hand sides of assignments, type assertions, members of object and array literals, and return statements. The contextual type also acts as a candidate type in best common type. For example:

```
function createZoo(): Animal[] {
    return [new Rhino(), new Elephant(), new Snake()];
}
```

In this example, best common type has a set of four candidates: Animal, Rhino, Elephant, and Snake. Of these, Animal can be chosen by the best common type algorithm.

Type Compatibility

Type compatibility in TypeScript is based on structural subtyping. Structural typing is a way of relating types based solely on their members. This is in contrast with nominal typing. Consider the following code:

```
interface Named {
    name: string;
}
```

```
class Person {
    name: string;
}

var p: Named;
// OK, because of structural typing
p = new Person();
```

In nominally-typed languages like C# or Java, the equivalent code would be an error because the Person class does not explicitly describe itself as being an implementor of the Named interface.

TypeScript's structural type system was designed based on how JavaScript code is typically written. Because JavaScript widely uses anonymous objects like function expressions and object literals, it's much more natural to represent the kinds of relationships found in JavaScript libraries with a structural type system instead of a nominal one.

A Note on Soundness

TypeScript's type system allows certain operations that can't be known at compile-time to be safe. When a type system has this property, it is said to not be "sound". The places where TypeScript allows unsound behavior were carefully considered, and throughout this document we'll explain where these happen and the motivating scenarios behind them.

Starting out

The basic rule for TypeScript's structural type system is that x is compatible with y if y has at least the same members as x. For example:

```
interface Named {
    name: string;
}

var x: Named;
// y’s inferred type is { name: string; location: string; }
var y = { name: 'Alice', location: 'Seattle' };
x = y;
```

To check whether y can be assigned to x, the compiler checks each property of x to find a corresponding compatible property in y. In this case, y must have a member called 'name' that is a string. It does, so the assignment is allowed.

The same rule for assignment is used when checking function call arguments:

```
function greet(n: Named) {
    alert('Hello, ' + n.name);
}
greet(y); // OK
```

Note that ‘y’ has an extra ‘location’ property, but this does not create an error. Only members of the target type (‘Named’ in this case) are considered when checking for compatibility.

This comparison process proceeds recursively, exploring the type of each member and sub-member.

Comparing two functions

While comparing primitive types and object types is relatively straightforward, the question of what kinds of functions should be considered compatible. Let’s start with a basic example of two functions that differ only in their argument lists:

```
var x = (a: number) => 0;
var y = (b: number, s: string) => 0;

y = x; // OK
x = y; // Error
```

To check if x is assignable to y, we first look at the parameter list. Each parameter in y must have a corresponding parameter in x with a compatible type. Note that the names of the parameters are not considered, only their types. In this case, every parameter of x has a corresponding compatible parameter in y, so the assignment is allowed.

The second assignment is an error, because y has a required second parameter that ‘x’ does not have, so the assignment is disallowed.

You may be wondering why we allow ‘discarding’ parameters like in the example y = x. The reason is that assignment is allowed is that ignoring extra function parameters is actually quite common in JavaScript. For example, Array#forEach provides three arguments to the callback function: the array element, its index, and the containing array. Nevertheless, it’s very useful to provide a callback that only uses the first argument:

```
var items = [1, 2, 3];

// Don't force these extra arguments
items.forEach((item, index, array) => console.log(item));

// Should be OK!
items.forEach((item) => console.log(item));
```

Now let’s look at how return types are treated, using two functions that differ only by their return type:

```
var x = () => ({name: 'Alice'});
var y = () => ({name: 'Alice', location: 'Seattle'});
```

```
x = y; // OK
y = x; // Error because x() lacks a location property
```

The type system enforces that the source function’s return type be a subtype of the target type’s return type.

Function Argument Bivariance

When comparing the types of function parameters, assignment succeeds if either the source parameter is assignable to the target parameter, or vice versa. This is unsound because a caller might end up being given a function that takes a more specialized type, but invokes the function with a less specialized type. In practice, this sort of error is rare, and allowing this enables many common JavaScript patterns. A brief example:

```
enum EventType { Mouse, Keyboard }

interface Event { timestamp: number; }
interface MouseEvent extends Event { x: number; y: number }
interface KeyEvent extends Event { keyCode: number }

function listenEvent(eventType: EventType, handler: (n: Event) => void) {
    /* ... */
}

// Unsound, but useful and common
listenEvent(EventType.Mouse, (e: MouseEvent) => console.log(e.x + ',' +
e.y));

// Undesirable alternatives in presence of soundness
listenEvent(EventType.Mouse, (e: Event) => console.log((<MouseEvent>e).x +
',' + (<MouseEvent>e).y));
listenEvent(EventType.Mouse, <(e: Event) => void>((e: MouseEvent) =>
console.log(e.x + ',' + e.y)));

// Still disallowed (clear error). Type safety enforced for wholly
incompatible types
listenEvent(EventType.Mouse, (e: number) => console.log(e));
```

Optional Arguments and Rest Arguments

When comparing functions for compatibility, optional and required parameters are interchangeable. Extra optional parameters of the source type are not an error, and optional parameters of the target type without corresponding parameters in the target type are not an error.

When a function has a rest parameter, it is treated as if it were an infinite series of optional parameters.

This is unsound from a type system perspective, but from a runtime point of view the idea of an optional parameter is generally not well-enforced since passing ‘undefined’ in that position is equivalent for most functions.

The motivating example is the common pattern of a function that takes a callback and invokes it with some predictable (to the programmer) but unknown (to the type system) number of arguments:

```
function invokeLater(args: any[], callback: (...args: any[]) => void) {
    /* ... Invoke callback with 'args' ... */
}

// Unsound - invokeLater "might" provide any number of arguments
invokeLater([1, 2], (x, y) => console.log(x + ', ' + y));

// Confusing (x and y are actually required) and undiscoverable
invokeLater([1, 2], (x?, y?) => console.log(x + ', ' + y));
```

Functions with overloads

When a function has overloads, each overload in the source type must be matched by a compatible signature on the target type. This ensures that the target function can be called in all the same situations as the source function. Functions with specialized overload signatures (those that use string literals in their overloads) do not use their specialized signatures when checking for compatibility.

Enums

Enums are compatible with numbers, and numbers are compatible with enums. Enum values from different enum types are considered incompatible. For example,

```
enum Status { Ready, Waiting };
enum Color { Red, Blue, Green };

var status = Status.Ready;
status = Color.Green;  //error
```

Classes

Classes work similarly to object literal types and interfaces with one exception: they have both a static and an instance type. When comparing two objects of a class type, only members of the instance are compared. Static members and constructors do not affect compatibility.

```
class Animal {
    feet: number;
    constructor(name: string, numFeet: number) { }
}

class Size {
    feet: number;
    constructor(numFeet: number) { }
}

var a: Animal;
var s: Size;

a = s;  //OK
s = a;  //OK
```

Private members in classes

Private members in a class affect their compatibility. When an instance of a class is checked for compatibility, if it contains a private member, the target type must also contain a private member that originated from the same class. This allows, for example, a class to be assignment compatible with its super class but not with classes from a different inheritance hierarchy which otherwise have the same shape.

Generics

Because TypeScript is a structural type system, type parameters only affect the resulting type when consumed as part of the type of a member. For example,

```
interface Empty<T> {
}
var x: Empty<number>;
var y: Empty<string>;

x = y;  // okay, y matches structure of x
```

In the above, x and y are compatible because their structures do not use the type argument in a differentiating way. Changing this example by adding a member to Empty<T> shows how this works:

```
interface NotEmpty<T> {
    data: T;
}
var x: NotEmpty<number>;
var y: NotEmpty<string>;

x = y;  // error, x and y are not compatible
```

In this way, a generic type that has its type arguments specified acts just like a non-generic type.

For generic types that do not have their type arguments specified, compatibility is checked by specifying 'any' in place of all unspecified type arguments. The resulting types are then checked for compatibility, just as in the non-generic case.

For example,

```
var identity = function<T>(x: T): T {
    // ...
}

var reverse = function<U>(y: U): U {
    // ...
}

identity = reverse;  // Okay because (x: any)=>any matches (y: any)=>any
```

Advanced Topics

Subtype vs Assignment

So far, we've used 'compatible', which is not a term defined in the language spec. In TypeScript, there are two kinds of compatibility: subtype and assignment. These differ only in that assignment extends subtype compatibility with rules to allow assignment to and from 'any' and to and from enum with corresponding numeric values.

Different places in the language use one of the two compatibility mechanisms, depending on the situation. For practical purposes, type compatibility is dictated by assignment compatibility even in the cases of the implements and extends clauses. For more information, see the TypeScript spec.

Writing .d.ts files

When using an external JavaScript library, or new host API, you'll need to use a declaration file (.d.ts) to describe the shape of that library. This guide covers a few high-level concepts specific to writing definition files, then proceeds with a number of examples that show how to transcribe various concepts to their matching definition file descriptions.

Guidelines and Specifics

Workflow

The best way to write a .d.ts file is to start from the documentation of the library, not the code. Working from the documentation ensures the surface you present isn't muddied with implementation details, and is typically much easier to read than JS code. The examples below will be written as if you were reading documentation that presented example calling code.

Namespacing

When defining interfaces (for example, "options" objects), you have a choice about whether to put these types inside a module or not. This is largely a judgement call -- if the consumer is likely to often declare variables or parameters of that type, and the type can be named without risk of colliding with other types, prefer placing it in the global namespace. If the type is not likely to be referenced directly, or can't be named with a reasonably unique name, do use a module to prevent it from colliding with other types.

Callbacks

Many JavaScript libraries take a function as a parameter, then invoke that function later with a known set of arguments. When writing the function signatures for these types, **do not** mark those parameters as optional. The right way to think of this is *"What parameters will be provided?"*, not *"What parameters will be consumed?"*. While TypeScript 0.9.7 and above does not enforce that the optionality, bivariance on argument optionality might be enforced by an external linter.

Extensibility and Declaration Merging

When writing definition files, it's important to remember TypeScript's rules for extending existing objects. You might have a choice of declaring a variable using an anonymous type or an interface type:

Anonymously-typed var

```
declare var MyPoint: { x: number; y: number; };
```

Interfaced-typed var

```
interface SomePoint { x: number; y: number; }
declare var MyPoint: SomePoint;
```

From a consumption side these declarations are identical, but the type SomePoint can be extended through interface merging:

```
interface SomePoint { z: number; }
MyPoint.z = 4; // OK
```

Whether or not you want your declarations to be extensible in this way is a bit of a judgement call. As always, try to represent the intent of the library here.

Class Decomposition

Classes in TypeScript create two separate types: the instance type, which defines what members an instance of a class has, and the constructor function type, which defines what members the class constructor function has. The constructor function type is also known as the "static side" type because it includes static members of the class.

While you can reference the static side of a class using the typeof keyword, it is sometimes useful or necessary when writing definition files to use the *decomposed class* pattern which explicitly separates the instance and static types of class.

As an example, the following two declarations are nearly equivalent from a consumption perspective:

Standard

```
class A {
    static st: string;
    inst: number;
    constructor(m: any) {}
}
```

Decomposed

```
interface A_Static {
    new(m: any): A_Instance;
    st: string;
}
interface A_Instance {
    inst: number;
}
declare var A: A_Static;
```

The trade-offs here are as follows:

- Standard classes can be inherited from using extends; decomposed classes cannot. This might change in later version of TypeScript if arbitrary extends expressions are allowed.
- It is possible to add members later (through declaration merging) to the static side of both standard and decomposed classes
- It is possible to add instance members to decomposed classes, but not standard classes
- You'll need to come up with sensible names for more types when writing a decomposed class

Naming Conventions

In general, do not prefix interfaces with I (e.g. IColor). Because the concept of an interface in TypeScript is much more broad than in C# or Java, the IFoo naming convention is not broadly useful.

Examples

Let's jump in to the examples section. For each example, sample *usage* of the library is provided, followed by the definition code that accurately types the usage. When there are multiple good representations, more than one definition sample might be listed.

Options Objects

Usage

```
animalFactory.create("dog");
animalFactory.create("giraffe", { name: "ronald" });
animalFactory.create("panda", { name: "bob", height: 400 });
// Invalid: name must be provided if options is given
animalFactory.create("cat", { height: 32 });
```

Typing

```
module animalFactory {
    interface AnimalOptions {
        name: string;
        height?: number;
        weight?: number;
    }
    function create(name: string, animalOptions?: AnimalOptions): Animal;
}
```

Functions with Properties

Usage

```
zooKeeper.workSchedule = "morning";
zooKeeper(giraffeCage);
```

Typing

```
// Note: Function must precede module
function zooKeeper(cage: AnimalCage);
module zooKeeper {
    var workSchedule: string;
}
```

New + callable methods

Usage

```
var w = widget(32, 16);
var y = new widget("sprocket");
// w and y are both widgets
w.sprock();
y.sprock();
```

Typing

```
interface Widget {
    sprock(): void;
}
```

```
interface WidgetFactory {
    new(name: string): Widget;
    (width: number, height: number): Widget;
}

declare var widget: WidgetFactory;
```

Global / External-agnostic Libraries

Usage

```
// Either
import x = require('zoo');
x.open();
// or
zoo.open();
```

Typing

```
module zoo {
  function open(): void;
}

declare module "zoo" {
    export = zoo;
}
```

Single Complex Object in External Modules

Usage

```
// Super-chainable library for eagles
import eagle = require('./eagle');
// Call directly
eagle('bald').fly();
// Invoke with new
var eddie = new eagle(1000);
// Set properties
eagle.favorite = 'golden';
```

Typing

```
// Note: can use any name here, but has to be the same throughout this file
declare function eagle(name: string): eagle;
declare module eagle {
    var favorite: string;
    function fly(): void;
}
interface eagle {
    new(awesomeness: number): eagle;
}

export = eagle;
```

Callbacks

Usage

```
addLater(3, 4, (x) => console.log('x = ' + x));
```

Typing

```
// Note: 'void' return type is preferred here
function addLater(x: number, y: number, (sum: number) => void): void;
```

Please post a comment here if there's a pattern you'd like to see documented! We'll add to this as we can.