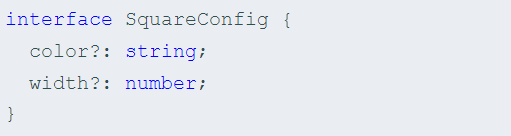
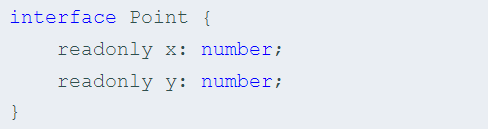
# TypeScript

1. **类型**
2. **接口**

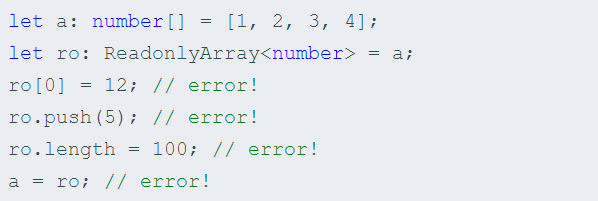
* **可选属性**

****

* **只读属性（readonly）**

****

TypeScript具有ReadonlyArray<T>类型，它与Array<T>相似，只是把所有可变方法去掉了，因此可以确保数组创建后再也不能被修改：



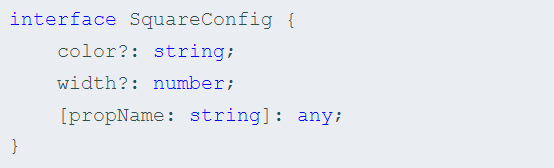
上面代码的最后一行，可以看到就算把整个ReadonlyArray赋值到一个普通数组也是不可以的。 但是你可以用类型断言重写：



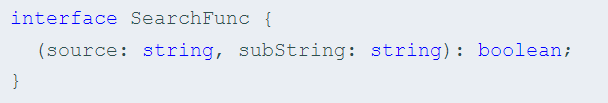
readonly vs const

最简单判断该用readonly还是const的方法是看要把它做为变量使用还是做为一个属性。 做为变量使用的话用 const，若做为属性则使用readonly。

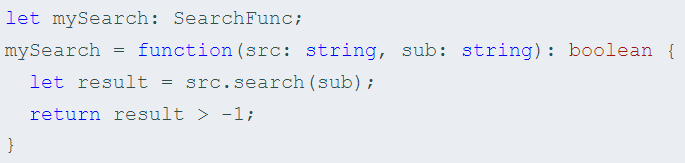
* **额外属性**

****

* **函数类型**

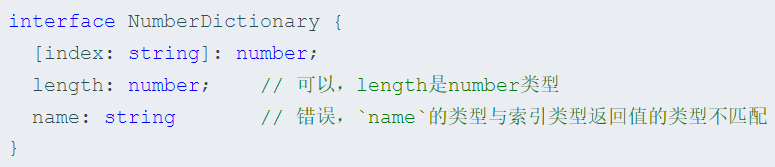
****

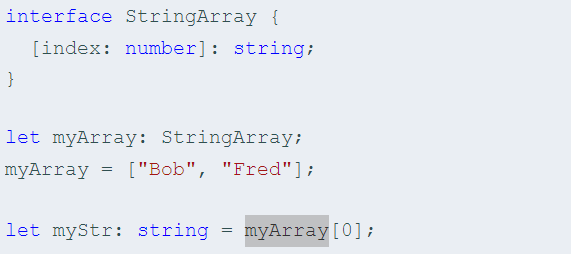
对于函数类型的类型检查来说，函数的参数名不需要与接口里定义的名字相匹配。 比如，我们使用下面的代码重写上面的例子：

****

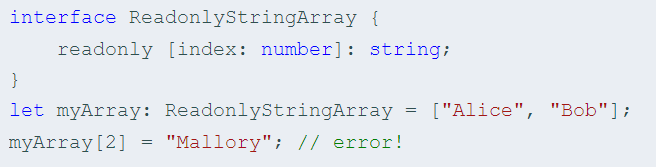
* **可索引的类型**

TypeScript支持两种索引签名：字符串和数字。 可以同时使用两种类型的索引，但是数字索引的返回值必须是字符串索引返回值类型的子类型。 这是因为当使用 number来索引时，JavaScript会将它转换成string然后再去索引对象。 也就是说用 100（一个number）去索引等同于使用"100"（一个string）去索引，因此两者需要保持一致。

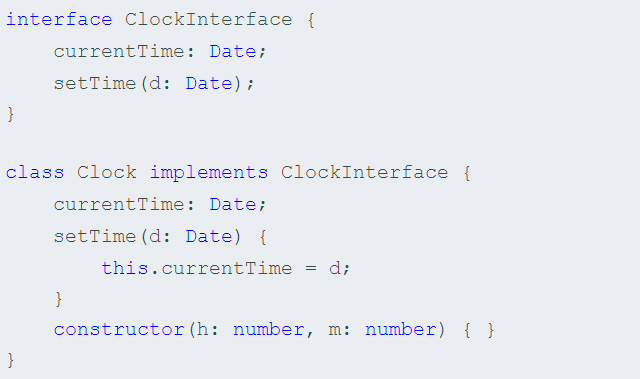


****

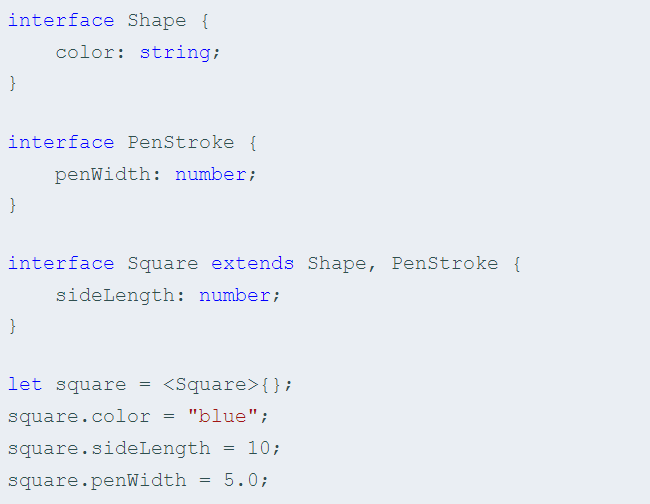
最后，你可以将索引签名设置为只读，这样就防止了给索引赋值：

****

* **类类型**
* **实现接口**

****

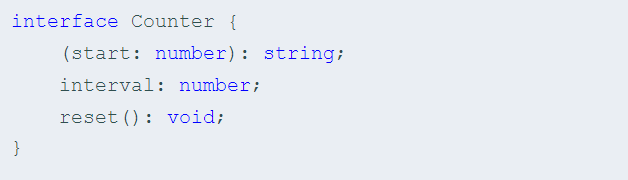
* **继承接口**

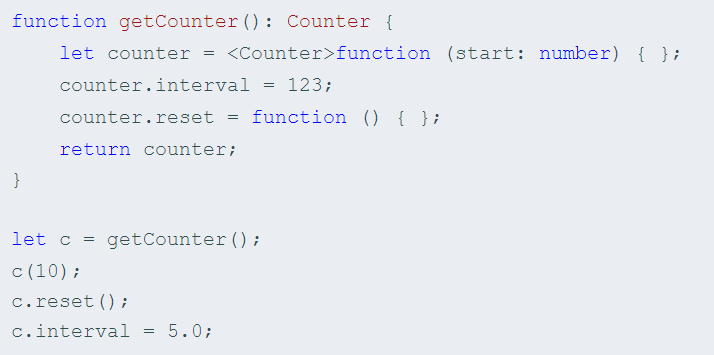
****

* **混合类型**

先前我们提过，接口能够描述JavaScript里丰富的类型。 因为JavaScript其动态灵活的特点，有时你会希望一个对象可以同时具有上面提到的多种类型。

一个例子就是，一个对象可以同时做为函数和对象使用，并带有额外的属性。



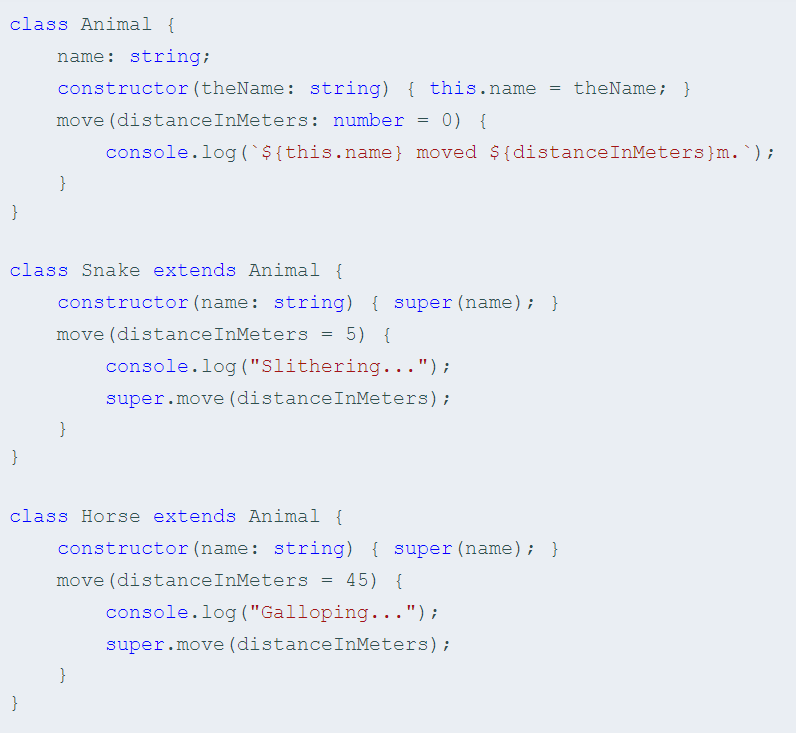


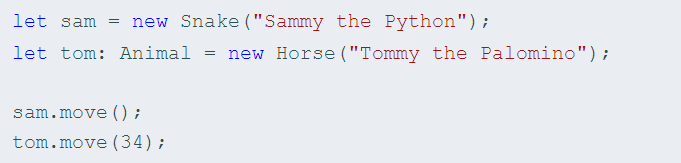
* **接口继承类**

****

1. **类**

* **继承**

****

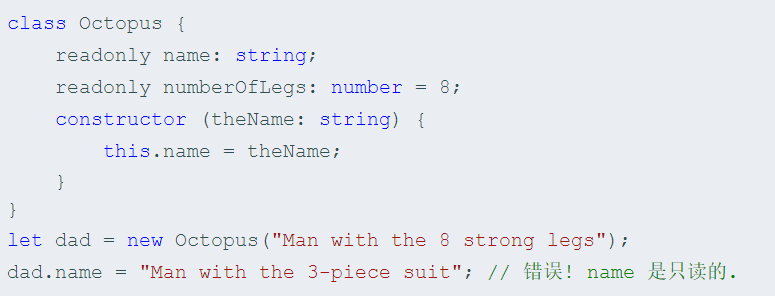
****

派生类包含了一个构造函数，它 必须调用 super()，它会执行基类的构造函数。 而且，在构造函数里访问 this的属性之前，我们 一定要调用 super()。 这个是TypeScript强制执行的一条重要规则。

这个例子演示了如何在子类里可以重写父类的方法。 Snake类和 Horse类都创建了 move方法，它们重写了从 Animal继承来的 move方法，使得 move方法根据不同的类而具有不同的功能。 注意，即使 tom被声明为 Animal类型，但因为它的值是 Horse，调用 tom.move(34)时，它会调用 Horse里重写的方法：

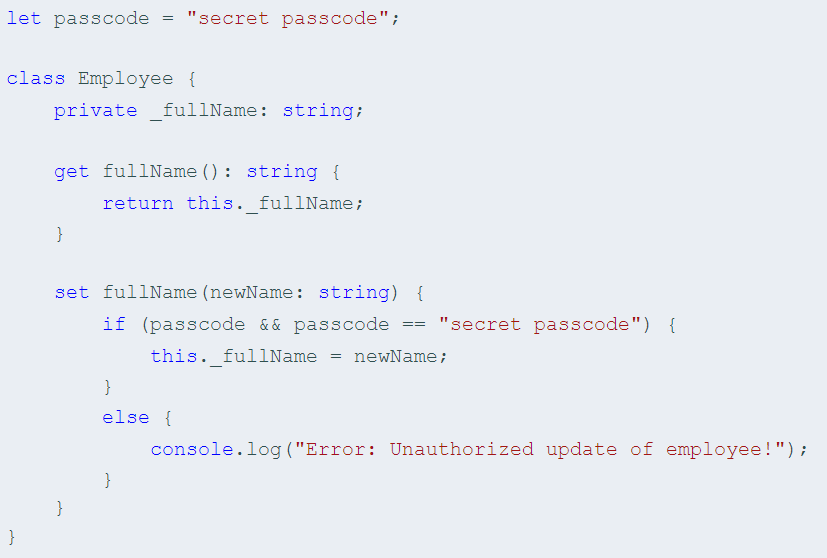
* **公共，私有与受保护的修饰符**
* **public：**公共的
* **private：**私有的
* **protected：**受保护的，只能父类和子类访问
* **readonly修饰符**

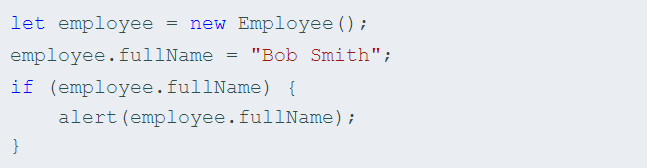
你可以使用 readonly关键字将属性设置为只读的。 只读属性必须在声明时或构造函数里被初始化。

****

* **存取器**

TypeScript支持通过getters/setters来截取对对象成员的访问。 它能帮助你有效的控制对对象成员的访问。





我们先检查用户密码是否正确，然后再允许其修改员工信息。 我们把对 fullName的直接访问改成了可以检查密码的 set方法。 我们也加了一个 get方法，让上面的例子仍然可以工作。

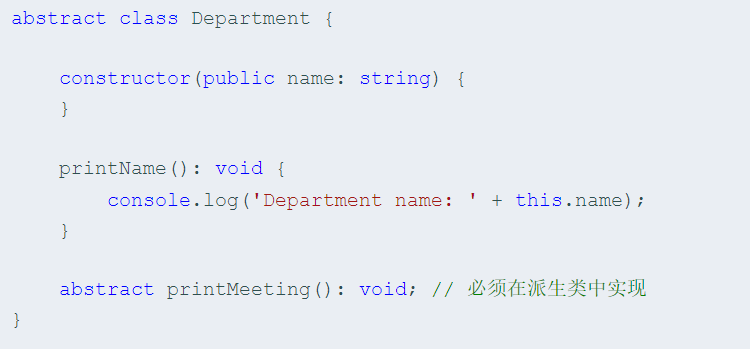
* **静态属性**

到目前为止，我们只讨论了类的实例成员，那些仅当类被实例化的时候才会被初始化的属性。 我们也可以创建类的静态成员，这些属性存在于类本身上面而不是类的实例上。

* **抽象类（abstract）**

抽象类做为其它派生类的基类使用。 它们一般不会直接被实例化。 不同于接口，抽象类可以包含成员的实现细节。 abstract关键字是用于定义抽象类和在抽象类内部定义抽象方法。

抽象类中的抽象方法不包含具体实现并且必须在派生类中实现。 抽象方法的语法与接口方法相似。 两者都是定义方法签名但不包含方法体。 然而，抽象方法必须包含 abstract关键字并且可以包含访问修饰符。

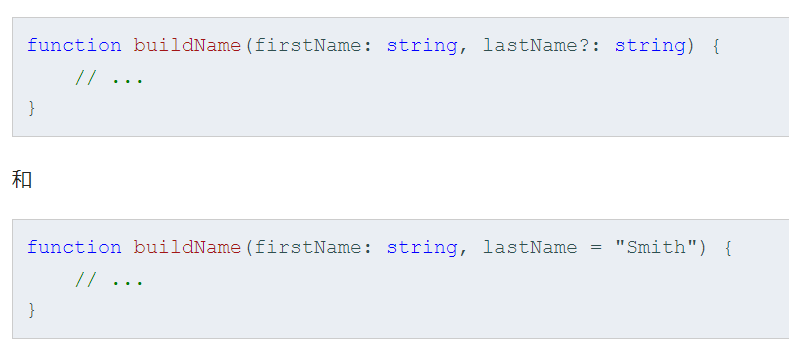
****

****

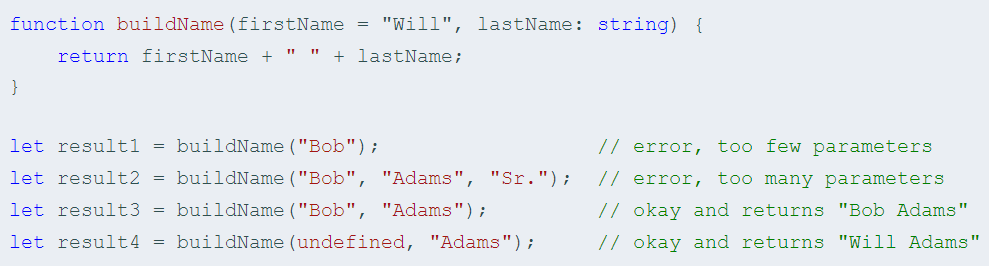
1. **函数**

* **默认参数**

在所有必须参数后面的带默认初始化的参数都是可选的，与可选参数一样，在调用函数的时候可以省略。 也就是说可选参数与末尾的默认参数共享参数类型。

****

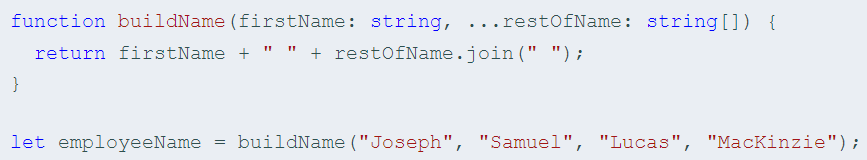
与普通可选参数不同的是，带默认值的参数不需要放在必须参数的后面。 如果带默认值的参数出现在必须参数前面，用户必须明确的传入 undefined值来获得默认值。 例如，我们重写最后一个例子，让 firstName是带默认值的参数：

****

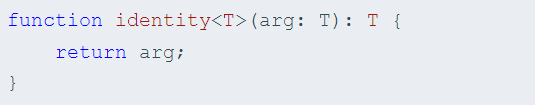
* **剩余参数**

必要参数，默认参数和可选参数有个共同点：它们表示某一个参数。 有时，你想同时操作多个参数，或者你并不知道会有多少参数传递进来。 在JavaScript里，你可以使用 arguments来访问所有传入的参数。

在TypeScript里，你可以把所有参数收集到一个变量里：

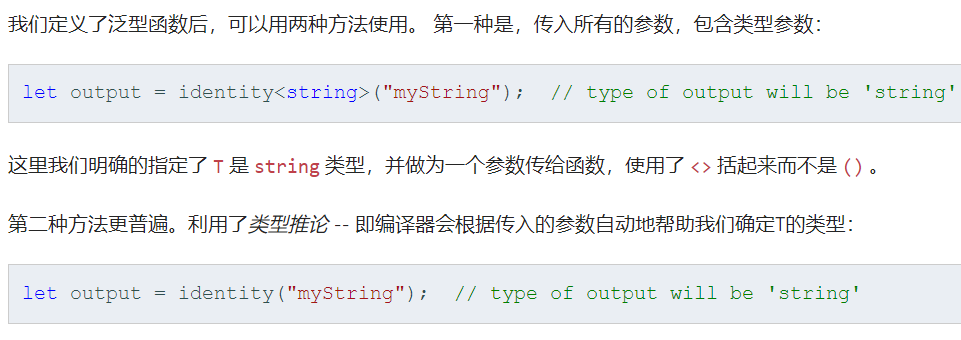
****

1. **泛型**

****

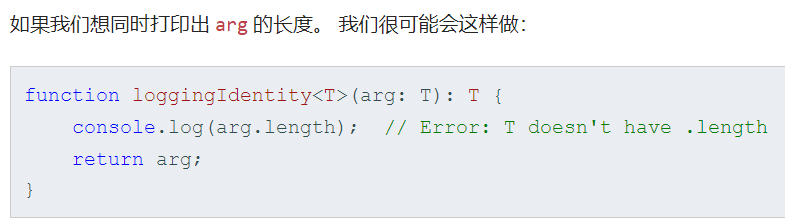
我们给identity添加了类型变量T。 T帮助我们捕获用户传入的类型（比如：number），之后我们就可以使用这个类型。 之后我们再次使用了 T当做返回值类型。

我们把这个版本的identity函数叫做泛型，因为它可以适用于多个类型。 不同于使用 any，它不会丢失信息

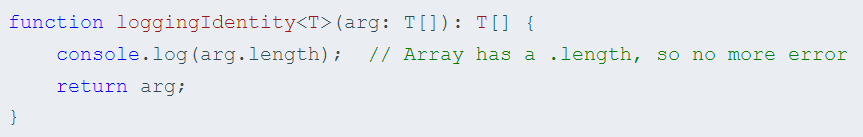
****

注意我们没必要使用尖括号（<>）来明确地传入类型；编译器可以查看myString的值，然后把T设置为它的类型。 类型推论帮助我们保持代码精简和高可读性。如果编译器不能够自动地推断出类型的话，只能像上面那样明确的传入T的类型

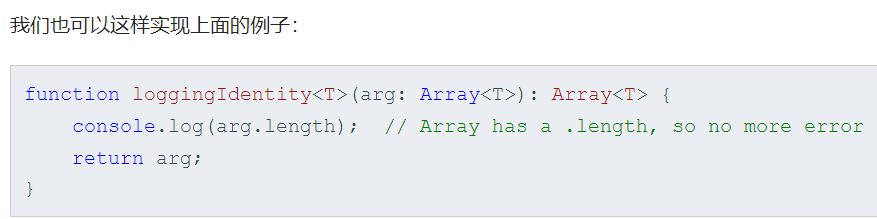
* **泛型变量**

****

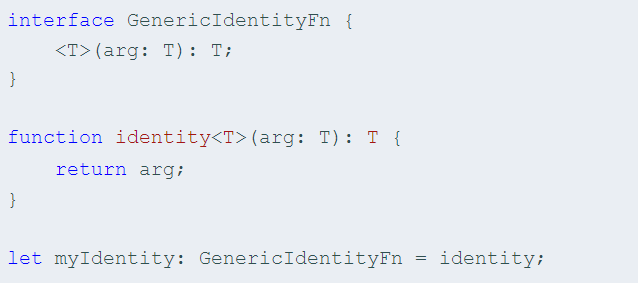
如果这么做，编译器会报错说我们使用了arg的.length属性，但是没有地方指明arg具有这个属性。 记住，这些类型变量代表的是任意类型，所以使用这个函数的人可能传入的是个数字，而数字是没有 .length属性的。

****

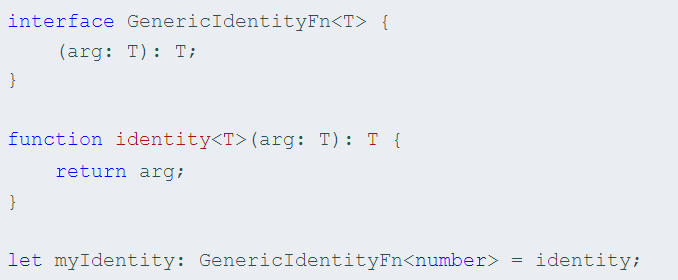
你可以这样理解loggingIdentity的类型：泛型函数loggingIdentity，接收类型参数T和参数arg，它是个元素类型是T的数组，并返回元素类型是T的数组。 如果我们传入数字数组，将返回一个数字数组，因为此时 T的的类型为number。

****

* **泛型接口**

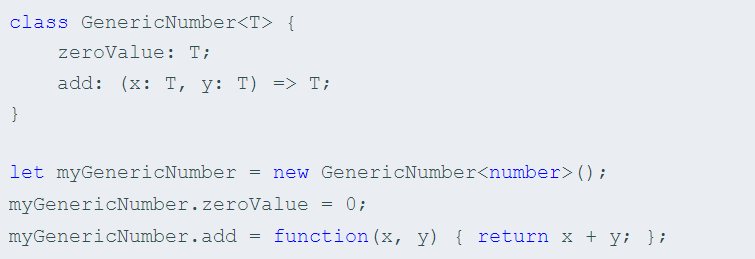
****

一个相似的例子，我们可能想把泛型参数当作整个接口的一个参数。 这样我们就能清楚的知道使用的具体是哪个泛型类型（比如： Dictionary<string>而不只是Dictionary）。 这样接口里的其它成员也能知道这个参数的类型了。

****

* **泛型类**

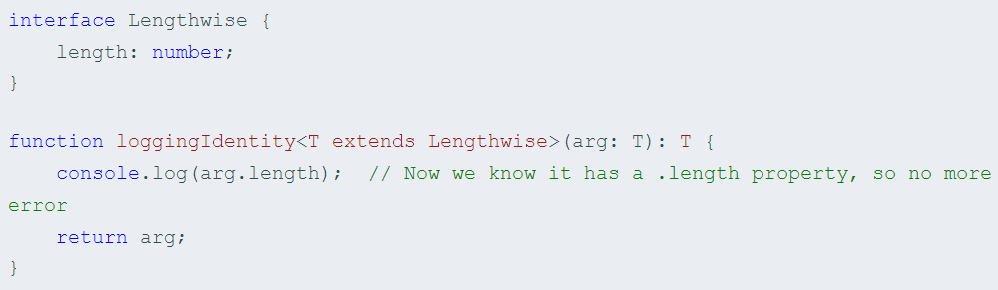
泛型类看上去与泛型接口差不多。 泛型类使用（ <>）括起泛型类型，跟在类名后面。

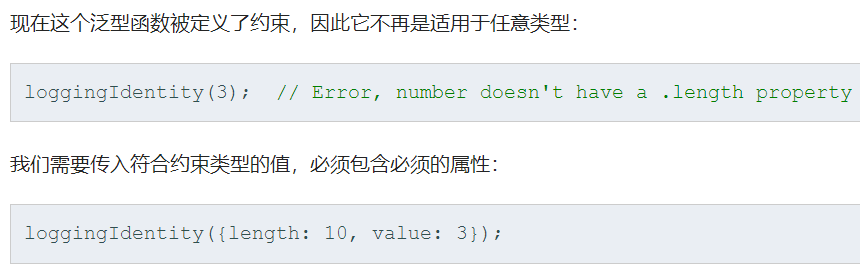
****

GenericNumber类的使用是十分直观的，并且你可能已经注意到了，没有什么去限制它只能使用number类型。 也可以使用字符串或其它更复杂的类型。

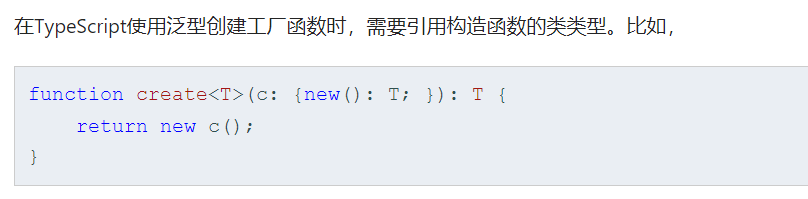
* **泛型约束**

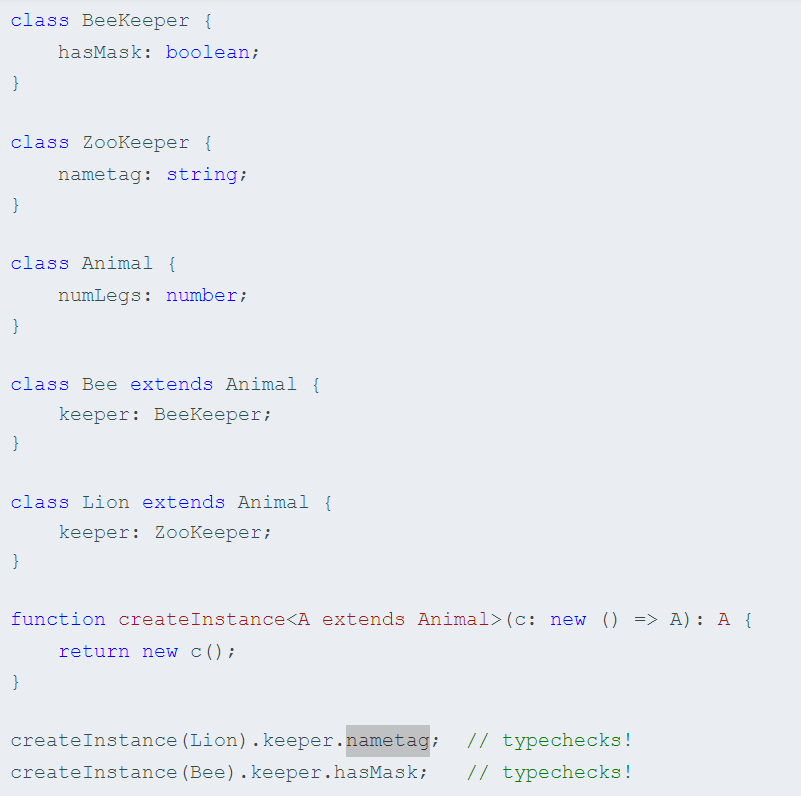
定义一个接口来描述约束条件。 创建一个包含 .length属性的接口，使用这个接口和extends关键字来实现约束：

****

****

* **在泛型里使用类类型**

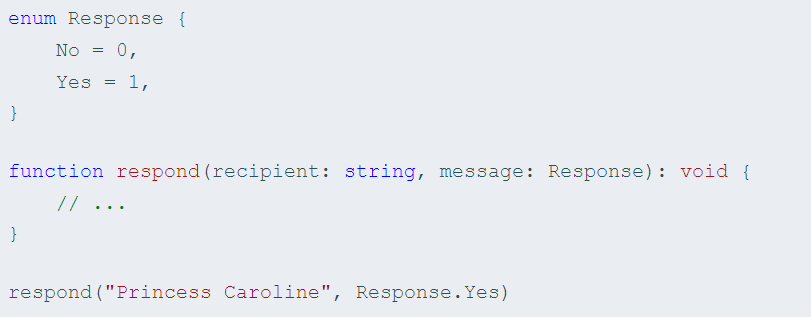
****

****

1. **枚举**

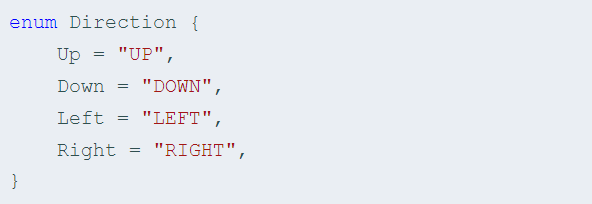
使用枚举很简单：通过枚举的属性来访问枚举成员，和枚举的名字来访问枚举类型：

* **数字枚举**

****

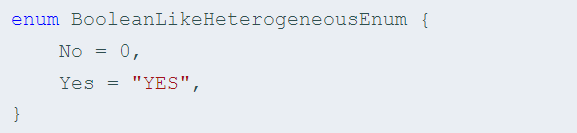
* **字符串枚举**

在一个字符串枚举里，每个成员都必须用字符串字面量，或另外一个字符串枚举成员进行初始化。

****

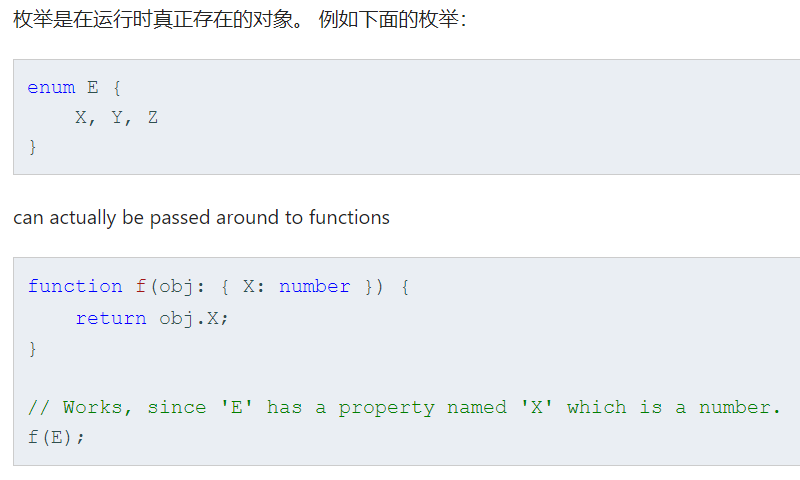
* **异构枚举**

从技术的角度来说，枚举可以混合字符串和数字成员，但是似乎你并不会这么做：

****

除非你真的想要利用JavaScript运行时的行为，否则我们不建议这样做。

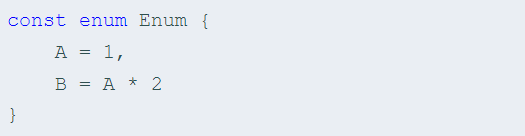
* **运行时枚举**

****

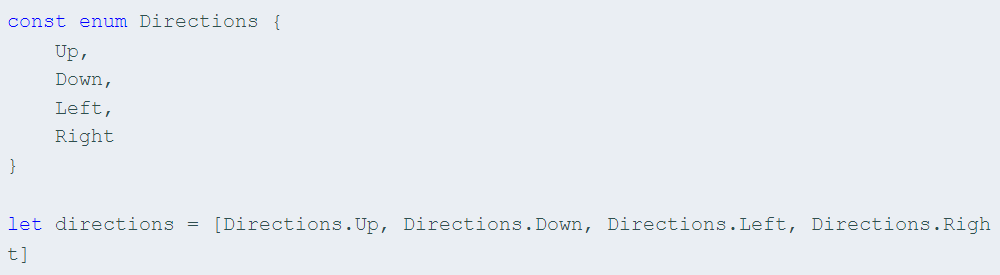
注意：返回值位0，默认位0、1、2

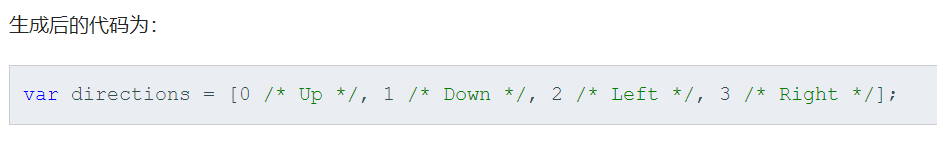
* **常量枚举**

为了避免在额外生成的代码上的开销和额外的非直接的对枚举成员的访问，我们可以使用 const枚举。 常量枚举通过在枚举上使用 const修饰符来定义。



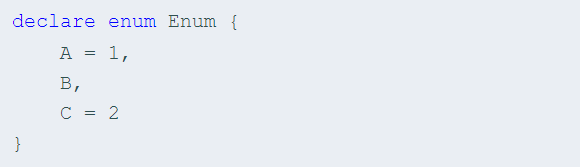
常量枚举只能使用常量枚举表达式，并且不同于常规的枚举，它们在编译阶段会被删除。 常量枚举成员在使用的地方会被内联进来。 之所以可以这么做是因为，常量枚举不允许包含计算成员。

****

****

* **外部枚举**

外部枚举用来描述已经存在的枚举类型的形状。

****

外部枚举和非外部枚举之间有一个重要的区别，在正常的枚举里，没有初始化方法的成员被当成常数成员。 对于非常数的外部枚举而言，没有初始化方法时被当做需要经过计算的。

1. **类型推论**