```
interface BankAccount {
    balance: number;
    deposit(credit: number): number;
}
```

If we were to write out the function type declaration for the 'BankAccount' constructor variable, it would have the following form.

```
var BankAccount: new() => BankAccount;
```

The function signature is prefixed with the keyword 'new' indicating that the 'BankAccount' function must be called as a constructor. It is possible for a function's type to have both call and constructor signatures. For example, the type of the built-in JavaScript Date object includes both kinds of signatures.

If we want to start our bank account with an initial balance, we can add to the 'BankAccount' class a constructor declaration.

```
class BankAccount {
    balance: number;
    constructor(initially: number) {
        this.balance = initially;
    }
    deposit(credit: number) {
        this.balance += credit;
        return this.balance;
    }
}
```

This version of the 'BankAccount' class requires us to introduce a constructor parameter and then assign it to the 'balance' field. To simplify this common case, TypeScript accepts the following shorthand syntax.

```
class BankAccount {
    constructor(public balance: number) {
    }
    deposit(credit: number) {
        this.balance += credit;
        return this.balance;
    }
}
```

The 'public' keyword denotes that the constructor parameter is to be retained as a field. Public is the default visibility for class members, but a programmer can also specify private visibility for a class member. Private visibility is a design-time construct; it is enforced during static type checking but does not imply any runtime enforcement.

TypeScript classes also support inheritance, as in the following example.