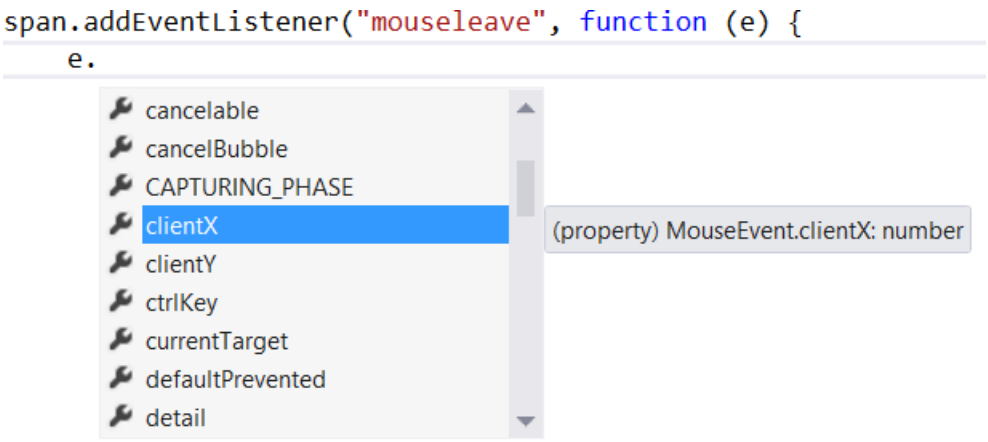
In the following screen shot, a programming tool combines information from overloading on string parameters with contextual typing to infer that the type of the variable 'e' is 'MouseEvent' and that therefore 'e' has a 'clientX' property.



Section 3.7.2.4 provides details on how to use string literals in function signatures.

1.9 Generic Types and Functions

Like overloading on string parameters, *generic types* make it easier for TypeScript to accurately capture the behavior of JavaScript libraries. Because they enable type information to flow from client code, through library code, and back into client code, generic types may do more than any other TypeScript feature to support detailed API descriptions.

To illustrate this, let's take a look at part of the TypeScript interface for the built-in JavaScript array type. You can find this interface in the 'lib.d.ts' file that accompanies a TypeScript distribution.

```
interface Array<T> {
    reverse(): T[];
    sort(compareFn?: (a: T, b: T) => number): T[];
    // ...
}
```

Interface definitions, like the one above, can have one or more *type parameters*. In this case the 'Array' interface has a single parameter, 'T', that defines the element type for the array. The 'reverse' method returns an array with the same element type. The sort method takes an optional parameter, 'compareFn', whose type is a function that takes two parameters of type 'T' and returns a number. Finally, sort returns an array with element type 'T'.

Functions can also have generic parameters. For example, the array interface contains a 'map' method, defined as follows:

```
map<U>(func: (value: T, index: number, array: T[]) => U, thisArg?: any): U[];
```