Objects and custom properties

Custom properties

When building your application you may need to attach to objects some custom properties. One very common need is to have an or a on objects.`id` `name`

If you are using Typescript or if you want your IDE to suggest you completition you will need to clearly state which those properties are.

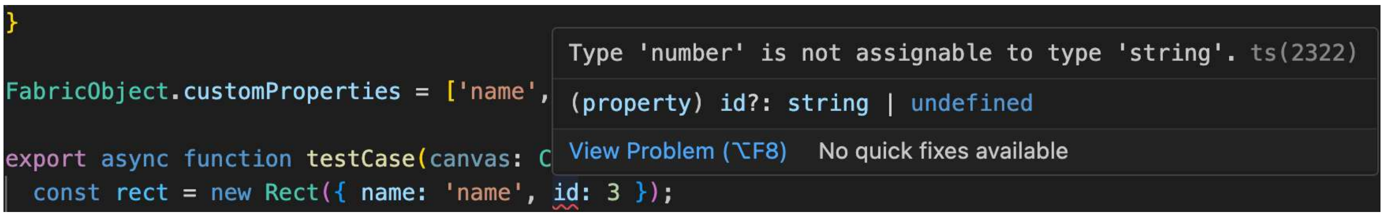
On top of that there is the serialization issue that will require you to pass down those properties in the argument of the function .`toObject`

```
// example without proper typing:
(myRect as any).name = 'rectangle';
myRect.toObject(['name', 'id']);
```

To make this nicer you have to use the Typescript feature for interfaces and a custom property hook in the object class

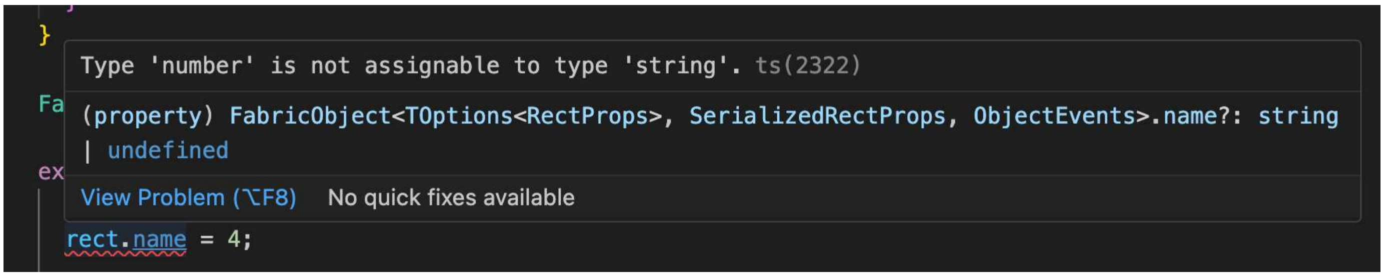
```
import { FabricObject } from 'fabric';

declare module "fabric" {
  // to have the properties recognized on the instance and in the constructor
  interface FabricObject {
    id?: string;
    name?: string;
  }
  // to have the properties typed in the exported object
  interface SerializedObjectProps {
    id?: string;
    name?: string;
  }
}

// to actually have the properties added to the serialized object
FabricObject.customProperties = ['name', 'id'];
```

This change will make the types work correcty:



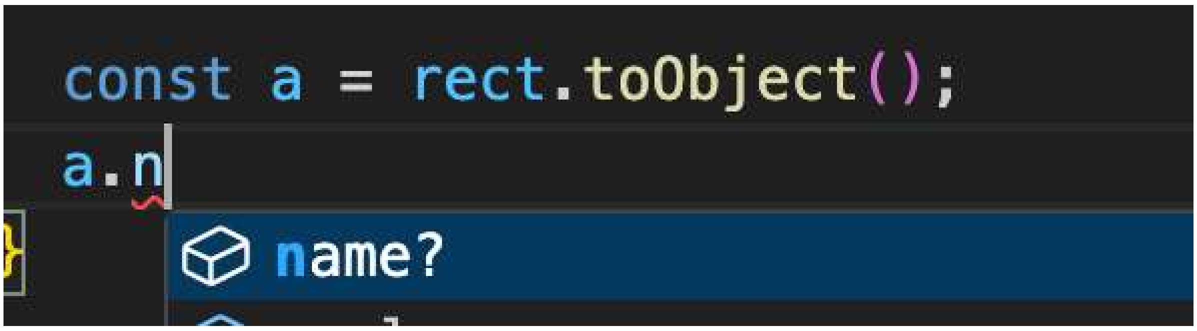
```
}

FabricObject.customProperties = ['name',

export async function testCase(canvas: C
  const rect = new Rect({ name: 'name', id: 3 });
```

```
Type 'number' is not assignable to type 'string'. ts(2322)
(property) id?: string | undefined
View Problem (⌥F8)    No quick fixes available
```



```
  }
Fa
ex
  rect.name = 4;
```

```
Type 'number' is not assignable to type 'string'. ts(2322)
(property) FabricObject<TOptions<RectProps>, SerializedRectProps, ObjectEvents>.name?: string
| undefined
View Problem (⌥F8)    No quick fixes available
```



```
const a = rect.toObject();
a.n
```

In general it makes things simplier if you can stick to external utility functions, but in the cases in which you want to attach a specific method to the prototype of different classes, you have to do modify the interface again:

```
// declare the methods for typescript
declare module "fabric" {
  // to have the properties recognized on the instance and in the constructor
  interface Rect {
    getText: () => string;
  }
  // to have the properties typed in the exported object
  interface Text {
    getText: () => string;
  }
}

// then add the methods to the classes:
Rect.prototype.getText = function() { return 'none'; }
Text.prototype.getText = function() { return this.text; }
```

Custom events

If you are looking to fire or understand custom events, look here: Events.

Subclassing

Subclassing is easier but is not always possible. If you want to subclass leaf nodes like , , , , that is possible and easy`RectTextboxITextPath`

```
import { classRegistry, SerializedPathProps } from 'fabric';

interface UniquePathPlusProps {
  id?: string;
  name?: string;
}

export interface SerializedPathPlusProps
  extends SerializedPathProps,
    UniquePathPlusProps {}

export interface PathPlusProps extends SerializedPathProps, UniqueRectProps {}

export class PathPlus<
  Props extends TOptions<PathPlusProps> = Partial<PathPlusProps>,
  SProps extends SerializedPathPlusProps = SerializedPathPlusProps,
  EventSpec extends ObjectEvents = ObjectEvents,
> extend Path<Props, SProps, EventSpec> {
  static type: 'path' // if you want it to override Path completely
  declare id?: string;
  declare name?: string;

  toObject<
    T extends Omit<Props & TClassProperties<this>, keyof SProps>,
    K extends keyof T = never,
  >(propertiesToInclude: K[] = []): Pick<T, K> & SProps {
    return super.toObject([...propertiesToInclude, 'id', 'name']);
  }
}

// to make possible restoring from serialization
classRegistry.setClass(PathPlus, 'path');
// to make PathPlus connected to svg Path element
classRegistry.setSVGClass(PathPlus, 'path');
```

But you can't subclass FabricObject and add it back in the prototype chain of the other objects.

You should add custom properties when they make your life simpler during rendering or event handling. In general Fabric.js classes/objects on the canvas shouldn’t contain data that is not relevant to their rendering needs or behaviour configuration, they shouldn’t become data stores of your application.