Algebraic Data Types and Generics in TypeScript

Here we'll inspect the Algebraic Data Types and Generics concepts applied to TypeScript.

Algebraic Data Types

An algebraic data type is a type composed by combining other types.

Algebraic Data Types have two important properties, the first is the fact that they can be *pattern matched*. Pattern matching allows to match a value (or an object) against some patterns to select a branch of the code. The second is that allow us to create recursive structures.

Union Types

A union type gives a list of types that a value can have. Must use the '|' symbol to make a union. For example:

```
type Alphanumeric = string | number;
function print(value: Alphanumberic) {
  console.log(value);
}
print("hello");
print(123);
```

This is a typical way to define recursive data structures in functional programming languages. We can define a tree structure like that:

```
type Empty = void;

class Leaf {
    constructor(public value: Alphanumeric | Empty){}

  toString(): string{
    return this.value;
  }
}
class Node {
  constructor(public leftChild: Node | Leaf, public rightChild: Node | Leaf){}

  toString(): string{
    return "Node";
  }
}
type Tree = Node | Leaf;
const leaf: Leaf = new Leaf("I'm a leaf");
const node: Node = new Node(leaf, leaf);
const tree: Tree = node;

console.log(tree.rightChild.toString()); // => We get I'm a leaf as output
```

Intersection Types

Whereas a union type can hold any of the types in its list, an intersection type can only hold a value that is simultaneously an instance of every type in its list.

Must use the '&' symbol to make a union. We can take its power with something like that:

```
interface Plugable {
  plug : (() => void);
}
interface Updatable {
  update : (() => void);
}
type UpdatablePlugin = Plugable & Updatable;
class PluginManager {
  installPlugin(plugin: UpdatablePlugin){
    plugin.plug();
  }

    updatePlugin(plugin: UpdatablePlugin){
    plugin.update();
  }
}
class MyPlugin implements Plugable, Updatable {
  constructor(public name: string){}

  plug() {
      console.log(`Installing plugin: ${this.name}`);
  }

  update() {
      console.log(`Updating plugin: ${this.name}`);
  }
}
const myPlugin: MyPlugin = new MyPlugin("Custom plugin");
const manager: PluginManager = new PluginManager();
manager.installPlugin(myPlugin);
manager.updatePlugin(myPlugin);

/*
The output:

"Instaling plugin: Custom plugin"
"Updating plugin: Custom plugin"
*/
```

Type Guards

A function can take a union type parameter, and execute different logic depending on which it got. A type guard is a function that makes sure a value has the properties you'd expect of that type.

```
function isPlugable(item: (Plugable | Updatable)) : item is Plugable {
  return (<Plugable>item).plug !== undefined;
}
function isUpdatable(item: (Plugable | Updatable)) : item is Updatable {
  return (<Updatable>item).update !== undefined;
}
console.log(isPlugable(myPlugin));  // we'll get true
```

isPlugable and isUpdatable return a type predicate. we must cast the argument to the type that we'd expect to get the property we're checking for.

Now we can do that:

```
class EnhancedPluginManager extends PluginManager {

  executePlugin(plugin : (Plugable | Updatable)) : void {
    if (isPlugable(plugin))
        plugin.plug();
    else
        pugin.update();
  }
}
class MyPlugin2 implements Plugable {
  constructor(public name: string){}

  plug() {
      console.log(`Installing plugin: ${this.name}`);
  }
}
const myPlugin2: MyPlugin2 = new MyPlugin2("Custom plugin");
const manager: EnhancedPluginManager = new EnhancedPluginManager();
manager.executePlugin(myPlugin2); //=> output: "Installing plugin: Custom plugin"
```

Notice that we can just use else , rather than else if in executePlugin function, and still works successfully. This is because TypeScript knows that, if our plugin isn't Plugable, it must be Updatable.
TypeScript is able to learn this from the type predicate.

Generics

Once we know what Union is, we can see that with Unions we can write more general code than specific types do. But Unions have two drawbacks:

- We have to know the types beforehand to define the union properly.
- Writing type guards and conditional logic we are getting a fragile and verbose code.

To solve these drawbacks we can use the any type, but this solution forces to us to do type checking entirely.

The best solution is using Generics. Generics allow you to define functions, classes, and interfaces that work in general, while preserving some of the type checks that any throws away.

Generic functions

The general formula is:

```
FUNCTION_NAME < TYPE_VARIABLE> (args) : RETURN_TYPE {
    FUNCTION_BODY
}
```

to use:

```
function getString<T>(input : T) : string {
  return JSON.stringify(input);
}
console.log(getString<Node>(node));

//or directly:
console.log(getString(node));
```

We can return a generic type as well:

```
function getFirst<T>(items: T[]) : T {
  return items[0];
}
```

Generic Interfaces

TypeScript also allows you to use generics when defining method signatures and data members in an interface.

```
interface Cache {
  registerItem<E>(items : E[]) : void;
}

//or:

interface Cache<E> {
  registerItem(items : E[]) : void;
}
```

Generic Classes

Following the same Generic interface's syntax we can simply follow the class name with brackets containing a type variable to use generics in classes. A simple example is:

```
class MySingleton<T> {
  constructor(private _instance: T) {}

  get instance() : T {
    return this._instance;
  }
}
const instance: Date = new Date();
const singleton: MySingleton = new MySingleton(instance);

console.log(singleton.instance);
```

We can implement a new PluginManager more generic: We can use a generic constraint that we wont allow any type:

```
class GenericPluginManager<T extends UpdatablePlugin> {
  constructor(private plugins: T[]) {}

  executePlugins(){
    plugins.forEach(plugin => {
      plugin.plug();
      plugin.update();
    });
  }
}
const plugins: UpdatablePlugin[] = [new MyPlugin("Plugin_1")];
const manager: GenericPluginManager = new
GenericPluginManager<UpdatablePlugin>(plugins);
manager.executePlugins();
```

You can find the code here