Keep Sitting with YOUR PARTNER at your Assigned Tables!

Metaprogramming with Decorators and @Annotations

COMP423 / Fall 2024 / Lecture 12

Warm-up: What is the output?

```
interface AnyFunction {
  (...args: any[]): any;
}

const logged = (f: AnyFunction) => {
  return (...args: any[]) => {
    console.log("args:", args)
    let rv = f(...args);
    console.log("rv:", rv);
    return rv;
  };
};

let add = logged((x: number, y: number) => x + y);
let subtract = logged((x: number, y: number) => x - y);

console.log(subtract(5, 2));
```

Metaprogramming 101

Code that operates on *other code!*

- With meta programming, you can write code that treats *other code* in the program as data it can operate on.
  - Used in modern languages and frameworks to register functionality (e.g. @Components, @Injectable Services in Angular), to register test constructs (@pytest.fixture), to register routes in FastAPI (@app.get), add behavior niceties (@DataClass), and more!
- For common, "**cross-cutting**" concerns (e.g. Logging, Registering a Class w/ a Framework) it allows you to "decorate" existing classes/methods/functions and add capabilities *without reimplementing or modifying their implementation.*
  - This reduces a significant amount of "boilerplate" code that would otherwise be required to achieve the same result.
  - This concept is closely related to a pattern called Aspect-Oriented Programming (AOP)
- Not just in dynamic language runtimes. (*Although, they are handled more elegantly*.)
  - For example, in Java you can use functionality in the java.lang.reflect package for metaprogramming

- Open https://www.typescriptlang.org/
- Delete the example program already in the code
- Pop open the right hand side drawer (click the arrow) and go to Logs
- Open TS Config and go down to Language and Environment
  - Enable the Experimental Decorators Feature
  - Run the code below and investigate the output in the dev tools console

Exercise:

Look at the output you are seeing in console and then try to explain in English, step-by-step, how the code listing evaluates in the language runtime.

Write down the order of steps on your whiteboard.

```
function Logger(target: any, propertyKey: string, descriptor: PropertyDescriptor) {
  console.log(target, propertyKey, descriptor);
}

class Ops {
  @Logger
  add(a: number, b: number): number {
    return a + b;
  }
}

console.log("TODO...");
```

Let's Implement a method Logger Annotation!

```
function Logger(target: any, propertyKey: string, descriptor: PropertyDescriptor) {
  console.log(`Logger Decorating ${propertyKey}`);
  const originalMethod = descriptor.value;
  descriptor.value = function (...args: any[]) {
    console.log(`args: ${JSON.stringify(args)}`);
    const rv = originalMethod.call(this, ...args);
    console.log(`rv: ${rv}`);
    return rv;
  };
  return descriptor;
}
```

... after the class definition, try instantiating a new Ops object and calling its add method!

```ts
function Logger(target: any, propertyKey: string, descriptor: PropertyDescriptor) {
  console.log(`Logger Decorating ${propertyKey}`);
  const originalMethod = descriptor.value;
  descriptor.value = function (...args: any[]) {
    console.log(`args: ${JSON.stringify(args)}`)
    const rv = originalMethod.call(this, ...args);
    console.log(`rv: ${rv}`);
    return rv;
  };
  return descriptor;
}

class Ops {
  @Logger
  add(a: number, b: number): number {
    return a + b;
  }
}

let ops = new Ops();
console.log(ops.add(1, 2));
```

@Injectable