Decorators:

TypeScript decorators are a powerful feature introduced in TypeScript 1.5 that allows you to add annotations and modify the behavior of classes, methods, properties, accessors, or parameters at design time. They provide a way to modify or extend classes and their members during compilation.

Decorators are defined by prefixing a function or expression with the `@` symbol. They can be applied to classes, methods, accessors, properties, or parameters. TypeScript supports four kinds of decorators:

1. **Class decorators:** Applied to classes, these decorators receive the constructor function of the class as their only parameter. They can be used to modify the class behavior or add metadata to it.

2. **Method decorators:** Applied to methods within a class, these decorators receive three parameters: the target object (the prototype of the class or constructor function if the method is static), the name of the method, and a property descriptor for the method.

3. **Property decorators:** Applied to properties of a class, these decorators receive two parameters: the target object (the prototype of the class) and the name of the property.

4. **Parameter decorators:** Applied to parameters of a method or constructor within a class, these decorators receive three parameters: the target object, the name of the method or constructor, and the index of the parameter.

Here's an example demonstrating how decorators can be used:

```typescript
// Class decorator
function Logger(target: Function) {
  console.log("Logging...");
}

// Method decorator
function Log(target: any, methodName: string, descriptor: PropertyDescriptor) {
  console.log("Method decorator logged:", methodName);
}

// Property decorator
function LogProperty(target: any, propertyName: string) {
  console.log("Property decorator logged:", propertyName);
}
```

```
// Parameter decorator
function LogParameter(target: any, methodName: string, parameterIndex: number) {
  console.log("Parameter decorator logged:", methodName, parameterIndex);
}

// Applying decorators
@Logger
class Example {
  @LogProperty
  prop: string;

  constructor(@LogParameter arg1: string, @LogParameter arg2: number) {}

  @Log
  method() {}
}
```

In this example, `Logger` is a class decorator, `Log` is a method decorator, `LogProperty` is a property decorator, and `LogParameter` is a parameter decorator. When the `Example` class is defined, the decorators are applied accordingly, and their associated logging statements are executed.

Decorators can be used for various purposes such as adding logging, validation, authentication, caching, or dependency injection to classes and their members. They provide a flexible and elegant way to extend and customize the behavior of TypeScript code at compile time.

Usages:

Decorators have a wide range of use cases in TypeScript, offering a powerful mechanism for modifying and extending the behavior of classes and their members. Here are some common usages of decorators:

1. **Logging and Debugging**: Decorators can be used to log method calls, parameter values, property accesses, or any other relevant information for debugging purposes.

2. **Validation**: Decorators can validate method arguments or object properties, ensuring that they meet certain criteria before execution.

3. **Authorization and Authentication**: Decorators can enforce authorization rules by restricting access to certain methods or properties based on the user's role or permissions.

4. **Caching**: Decorators can implement caching mechanisms to store the results of expensive method calls and retrieve them quickly on subsequent invocations.

5. **Dependency Injection**: Decorators can facilitate dependency injection by automatically injecting dependencies into class constructors or methods.

6. **Route Handling in Web Frameworks**: In web frameworks like NestJS or Angular, decorators are commonly used to define routes, middleware, and other HTTP-related functionality.

7. **Serialization**: Decorators can be used to control how objects are serialized to JSON or other formats, allowing customization of the serialization process.

8. **Memoization**: Decorators can memoize function calls, storing the results of previous invocations and returning them directly if the same inputs are provided again.

9. **Performance Monitoring**: Decorators can track the performance of methods or functions, recording execution times and identifying potential bottlenecks.

10. **Aspect-Oriented Programming (AOP)**: Decorators can implement cross-cutting concerns such as logging, caching, or error handling in a modular and reusable way, following the principles of AOP.

Examples:

Certainly! Here are code examples demonstrating the usages of decorators for the scenarios mentioned:

1. **Logging and Debugging**:

```typescript
function Log(target: any, methodName: string, descriptor: PropertyDescriptor) {
  const originalMethod = descriptor.value;

  descriptor.value = function(...args: any[]) {
    console.log(`Calling method ${methodName} with arguments:`, args);
    const result = originalMethod.apply(this, args);
    console.log(`Method ${methodName} returned:`, result);
    return result;
  };

  return descriptor;
}

class Example {
```

```
  @Log
  calculateSum(a: number, b: number): number {
    return a + b;
  }
}

const example = new Example();
example.calculateSum(3, 5);
```

2. **Validation**:

```typescript
function ValidateArgs(target: any, methodName: string, descriptor: PropertyDescriptor) {
  const originalMethod = descriptor.value;

  descriptor.value = function(...args: any[]) {
    if (args.some(arg => typeof arg !== 'number')) {
      throw new Error(`Invalid arguments provided to method ${methodName}`);
    }
    return originalMethod.apply(this, args);
  };

  return descriptor;
}

class Calculator {
  @ValidateArgs
  add(a: number, b: number): number {
    return a + b;
  }
}

const calculator = new Calculator();
console.log(calculator.add(3, 5)); // Output: 8
// console.log(calculator.add(3, '5')); // Throws error: Invalid arguments provided to method add
```

3. **Authorization and Authentication**: This example demonstrates a simple authorization decorator that restricts access to a method based on a user's role.

```typescript
function RequiresRole(role: string) {
  return function(target: any, methodName: string, descriptor: PropertyDescriptor) {
```

```
    const originalMethod = descriptor.value;

    descriptor.value = function(...args: any[]) {
      if (!currentUser || !currentUser.roles.includes(role)) {
        throw new Error(`User does not have permission to access method ${methodName}`);
      }
      return originalMethod.apply(this, args);
    };

    return descriptor;
  };
}

// Simulated current user
const currentUser = {
  roles: ['admin']
};

class AdminPanel {
  @RequiresRole('admin')
  deleteUser(userId: number): void {
    // Delete user logic
  }
}

const adminPanel = new AdminPanel();
// adminPanel.deleteUser(123); // Access allowed for admin user
// adminPanel.deleteUser(456); // Throws error: User does not have permission to access
method deleteUser
```

These are just a few examples to illustrate how decorators can be used for various purposes in TypeScript. Depending on your specific requirements and application architecture, you can create custom decorators or use existing libraries to address different use cases.