용어정리

용어	의미
Environment	컴퓨터 시스템이나 프로그램이 동작하는 데 필요한 모든 하드웨어, 소프트웨어 및 시스템 구성 요소//
Closure	프로그래밍에서 함수와 그 함수가 선언된 언어 환경(Lexical Environment)의 조합
Scope	변수나 함수의 유효 범위
Dynamic	프로그래밍에서 실행 시간(Runtime)에 데이터 타입이나 변수의 값 등이 결정되는 것
Static type	프로그래밍에서 변수에 미리 정해진 데이터 타입을 지정하여 사용하는 방식
Symbol table	컴파일러나 인터프리터 등에서 변수나 함수와 같은 식별자(Identifier)에 대한 정보를 저장하는 자료 구조
Tree Structure	계층적인 구조를 나타내는 것으로, 상위 요소와 하위 요소의 관계를 트리(Tree) 구조로 표현
Binding Time	변수나 함수와 같은 이름이나 값이 실제 메모리상에서 할당되는 시점을 의미
Container	프로그램 실행환경에서 다른 프로세스나 모듈을 포함하는 구조를 의미
Thread	프로세스 내에서 실행되는 실행 단위로, 하나의 프로세스는 여러 개의 스레드를 가질 수 있습니다. 스레드는 하나의 프로세스 내에서 공유되는 자원, 즉 메모리 공간을 사용하므로, 스레드 간의 자원 공유 및 동기화 문제를 고려
Thread Safe	스레드 세이프(Thread Safe)란 멀티스레드 환경에서 여러 스레드가 동시에 공유하는 자원(변수, 함수, 객체 등)에 대한 접근을 제어하여, 스레드 간의 경쟁 상황으로 인한 오류를 방지하는 것
Context	프로그램이 실행되는 동안 해당 프로그램의 실행 상태를 나타내는 정보입니다. 이 정보에는 프로그램 카운터, 레지스터 값, 메모리 할당 정보 등이 포함
Dependency Injection	객체 지향 프로그래밍에서 객체간의 결합도(Coupling)를 낮추기 위해 사용하는 설계 패턴입니다. 디펜던시 인젝션은 객체가 필요로 하는 의존성(서비스, 객체 등)을 외부에서 제공받아 객체 내부에서 생성하지 않고 주입해주는 것
Beans	

Container는 밖으로 나가는걸 막는 용도

Beans

스프링에서 "bean"이란 스프링 컨테이너가 관리하는 객체(인스턴스)를 의미합니다. 스프링 컨테이너는 XML 파일, Java Configuration 클래스, 어노테이션 등을 통해 빈을 정의하고 생성, 관리합니다. 스프링의 핵심 기능 중 하나인 DI(Dependency Injection)를 통해 빈들 간의 의존성을 자동으로 관리하므로 개발자는 객체 생성 및 의존성 관리에 대한 로직을 작성할 필요가 없어집니다.

스프링에서 빈을 등록하고 관리하는 방법에는 다음과 같은 방식들이 있습니다.

XML 파일에 빈 정의하기
스프링에서 가장 오래된 방법 중 하나로, XML 파일을 통해 빈을 등록하고 관리하는 방식입니다. XML 파일에 <bean> 태그를 사용하여 빈을 정의하고, 스프링 컨테이너에서 해당 XML 파일을 로딩하여 빈을 생성하고 관리합니다.

Java Configuration 클래스를 사용하여 빈 정의하기
자바 코드를 통해 빈을 정의하고 관리하는 방식입니다. 스프링 3부터 지원되는 방식으로, XML 파일을 사용하지 않고 자바 코드로 빈을 생성하고 의존성을 주입할 수 있습니다.

어노테이션을 사용하여 빈 정의하기
자바 클래스나 메서드에 특정 어노테이션을 사용하여 빈을 정의하고 관리하는 방식입니다. 주로 @Component, @Service, @Repository, @Controller 등의 어노테이션을 사용하여 빈을 등록합니다.

스프링의 빈은 싱글톤(Singleton)으로 생성되어, 여러 번 요청해도 하나의 인스턴스만 생성되어 사용됩니다. 이는 스프링의 메모리 관리 및 성능 향상을 위해 적용되는 기본적인 방식 중 하나입니다.

Dependency Injection

의존성 주입(Dependency Injection)은 객체 지향 프로그래밍에서 객체간의 결합도(Coupling)를 낮추기 위해 사용하는 설계 패턴입니다. 디펜던시 인젝션은 객체가 필요로 하는 의존성(서비스, 객체 등)을 외부에서 제공받아 객체 내부에서 생성하지 않고 주입해주는 것입니다.

스프링 프레임워크에서는 IoC(Inversion of Control) 컨테이너를 사용하여 디펜던시 인젝션을 구현합니다. IoC 컨테이너는 객체 생성, 관리, 의존성 주입 등을 자동으로 처리하며, 개발자는 객체 생성 및 관리를 위한 코드를 작성하지 않고도 IoC 컨테이너를 통해 객체를 사용할 수 있습니다.

스프링에서는 대표적으로 3가지 방식으로 의존성을 주입할 수 있습니다.

생성자 주입(Constructor Injection)
생성자를 통해 의존성을 주입하는 방식입니다. 생성자를 통해 의존성을 주입받으면 해당 객체가 생성될 때 필수적으로 주입받아야 하는 의존성을 명확히 나타낼 수 있습니다.

세터 주입(Setter Injection)
Setter 메서드를 통해 의존성을 주입하는 방식입니다. Setter 메서드를 통해 주입하는 방식이므로, 의존성이 선택적인 경우에 사용됩니다.

필드 주입(Field Injection)
필드에 직접 의존성을 주입하는 방식입니다. 이 방식은 코드가 간결하며 사용하기 쉽다는 장점이 있지만, 외부에서 해당 객체의 의존성을 주입받는 것이 어렵다는 단점이 있습니다.

의존성 주입은 객체지향 설계의 핵심 원칙인 SOLID 원칙 중 DIP(Dependency Inversion Principle)를 준수하기 위해 사용되며, 객체간의 결합도를 낮추어 코드의 유지보수성, 확장성을 향상시키는데 도움을 줍니다.

Context

컨텍스트는 일반적으로 프로세스나 스레드와 관련이 있습니다. 예를 들어, 하나의 프로세스가 여러 개의 스레드를 가지고 있을 때, 각 스레드는 공유되는 컨텍스트와 각각의 독립적인 스택을 가지고 있습니다. 스레드 간에 컨텍스트를 전환하면, 해당 스레드의 실행 상태가 저장되고 다음 스레드의 실행 상태가 로드됩니다. 이를 컨텍스트 전환(Context Switching)이라고 합니다.

컨텍스트는 시스템 호출, 인터럽트 처리, 멀티태스킹 등에 중요한 역할을 합니다. 이러한 작업들은 컨텍스트 전환을 필요로 하며, 이를 효율적으로 수행하는 것이 시스템의 성능 향상에 큰 영향을 미칩니다. 따라서, 컨텍스트 전환은 운영체제의 핵심 기능 중 하나입니다.

Thread

전산학에서 스레드(Thread)는 프로세스 내에서 실행되는 실행 단위로, 하나의 프로세스는 여러 개의 스레드를 가질 수 있습니다. 스레드는 하나의 프로세스 내에서 공유되는 자원, 즉 메모리 공간을 사용하므로, 스레드 간의 자원 공유 및 동기화 문제를 고려해야 합니다.

스레드는 다중 처리 기능을 활용하여, 병렬 처리 및 동시성 작업을 수행하는 데에 유용하게 사용됩니다. 예를 들어, 웹 서버에서 클라이언트 요청을 처리하는 데에는 다수의 스레드를 활용하여 동시에 처리할 수 있습니다. 또한, 그래픽 애니메이션과 같은 대규모 계산 작업에서도 스레드를 활용하여 병렬 처리를 수행할 수 있습니다.

스레드는 각각의 실행 경로를 갖기 때문에, 다른 스레드에 비해 상대적으로 경량화되어 있어, 생성 및 소멸 비용이 낮고, 스위칭 비용이 적어 효율적인 프로그래밍이 가능합니다. 따라서 스레드를 적극적으로 활용하는 것이 성능 향상에 큰 도움이 됩니다.

스레드를 사용하는 언어로는 C++, Java, Python 등이 있으며, 이들 언어에서는 스레드 관련 라이브러리를 제공하여 스레드 프로그래밍을 보다 쉽게 할 수 있도록 도와줍니다.

Thread Safety

스레드 세이프한 코드는 여러 스레드가 동시에 접근해도 안정적으로 동작해야 합니다. 이를 위해서는 여러 스레드가 공유하는 자원에 대한 접근을 제한하는 방법이 필요합니다. 대표적인 방법으로는 뮤텍스(Mutex), 세마포어(Semaphore), 경쟁 상황을 방지하는 알고리즘 등이 있습니다.

뮤텍스는 하나의 자원에 대한 접근을 한 번에 하나의 스레드만 가능하도록 제어합니다. 다른 스레드가 접근하려고 할 때는 뮤텍스를 획득할 수 없으므로 대기 상태에 들어갑니다. 뮤텍스를 해제하면 다음 스레드가 뮤텍스를 획득할 수 있게 됩니다.

세마포어는 뮤텍스와 유사하나, 동시에 접근 가능한 스레드의 수를 설정할 수 있습니다. 이를 통해 자원의 사용량을 제한하거나, 우선순위를 설정하여 더 중요한 스레드가 자원을 우선적으로 사용할 수 있도록 할 수 있습니다.

스레드 세이프한 코드는 멀티스레드 환경에서 안정적인 동작을 보장하므로, 대규모 시스템에서는 필수적인 기술입니다. 따라서 스레드 세이프한 코드를 작성하는 것은 중요한 개발 스킬 중 하나입니다.

Container

전산학에서 컨테이너(Container)는 프로그램 실행환경에서 다른 프로세스나 모듈을 포함하는 구조를 의미합니다. 일반적으로 운영체제 레벨에서 프로세스를 격리하고, 각 프로세스마다 독립된 메모리 공간을 할당하여 안전하게 실행되도록 보장하기 위해 사용됩니다.

컨테이너 기술은 프로그램의 이식성과 확장성을 높일 수 있으며, 동일한 프로그램이 다양한 환경에서 실행될 수 있도록 지원합니다. 가상화 기술을 이용하여, 컨테이너는 호스트 운영체제에서 격리된 가상 환경을 제공하며, 컨테이너 안에서는 프로세스와 모듈을 자유롭게 실행할 수 있습니다.

대표적으로 Docker와 Kubernetes가 컨테이너 기술을 이용한 대표적인 플랫폼입니다. Docker는 컨테이너 이미지를 생성하고 배포하는 기능을 제공하며, Kubernetes는 컨테이너를 자동으로 배치하고 관리하는 기능을 제공합니다. 이러한 컨테이너 기술은 대규모 분산 시스템과 클라우드 환경에서 주로 사용되며, DevOps와 같은 개발 방법론과 연계하여 사용됩니다.

Tree Structure와 Scope

스코프(Scope)와 나무 구조(Tree Structure)는 비슷한 개념입니다. 스코프는 변수, 함수, 객체 등이 유효한 범위를 나타내는 것으로, 해당 범위 내에서만 변수나 함수, 객체 등에 접근할 수 있습니다. 나무 구조는 계층적인 구조를 나타내는 것으로, 상위 요소와 하위 요소의 관계를 트리(Tree) 구조로 표현합니다.

스코프와 나무 구조는 비슷한 개념이므로, 스코프도 트리 구조로 표현될 수 있습니다. 전역 스코프는 트리 구조에서 뿌리(Root)에 해당하며, 각각의 함수는 자신의 부모 스코프(Parent Scope)와 자식 스코프(Child Scope)를 가지고 있습니다. 자식 스코프는 부모 스코프의 하위 요소이므로, 부모 스코프의 변수나 함수에 접근할 수 있습니다. 하지만 부모 스코프는 자식 스코프의 변수나 함수에 접근할 수 없습니다.

스코프는 나무 구조와 같은 계층적인 구조를 가지고 있으며, 이를 이용하여 변수나 함수 등의 이름 충돌을 방지하고, 코드의 유지보수성을 높일 수 있습니다. 또한, 스코프 체인(Scope Chain)을 이용하여 클로저(Closure)와 같은 고급 기능을 구현할 수 있습니다. 따라서, 스코프와 나무 구조는 프로그래밍에서 중요한 개념이며, 이를 잘 이해하고 활용하는 것이 좋습니다.

바인딩 타임

바인딩 타임(Binding Time)은 변수나 함수와 같은 이름이나 값이 실제 메모리상에서 할당되는 시점을 의미합니다. 즉, 이름이나 값이 언제 어디서 결정되는지를 나타내는 개념입니다.

바인딩 타임은 크게 컴파일 타임(Compile Time)과 런타임(Runtime)으로 나뉩니다. 컴파일 타임에서는 변수나 함수의 이름이나 타입 등이 결정되며, 코드가 실행되기 전에 이미 결정이 끝납니다. 컴파일 타임에서 결정되는 정보는 실행 중에 변경할 수 없으며, 프로그램의 성능과 안정성을 높이는 데에 중요한 역할을 합니다.

반면, 런타임에서는 변수나 함수의 값이나 메모리상의 위치 등이 결정됩니다. 런타임에서 결정되는 정보는 실행 중에 변경될 수 있으며, 동적으로 메모리를 할당하거나 객체를 생성하는 등의 작업에서 바인딩 타임이 일어납니다.

바인딩 타임은 프로그램의 동작 방식을 결정하는 중요한 요소 중 하나입니다. 따라서, 프로그래밍 언어에서는 컴파일 타임과 런타임의 개념을 잘 이해하고, 이를 활용하여 효율적인 프로그램을 작성할 수 있도록 지원합니다.

1. 다음에서 클로저를 분석하시오

```
function outerFunction() {
  var outerVar = 10;

  function innerFunction() {
    var innerVar = 20;
    console.log(outerVar + innerVar); // 30
  }

  return innerFunction;
}

var innerFunc = outerFunction();
innerFunc();
```