# 자바 프로그래밍

강경미(carami@nate.com)

# JAVA언어의 특징과 JDK 설치

# 자바 프로그래밍을 공부하는 이유?

- 기업에서 서버 프로그래밍시 가장 많이 사용하는 언어

# 자바 언어의 특징

- 객체지향 언어
- 자바 언어는 느리지만, 버전업 되면서 다른 언어들의 장점들을 흡수하고 있다. (모던 자바)
  - 람다(Lambda) : 함수형 프로그래밍
  - Stream API : 람다 표현식과 메서드 참조 등의 기능과 결합해서 매우 복잡하고 어려운 데이터 처리 작업을 쉽게 조회하고 필터링하고 변환하고 처리할 수 있도록 한다.
  - 병렬 프로그래밍 : 여러개의 CPU코어에서 작업을 배분해서 동시에 작업을 수행한다.

# 자바 프로그램 작성과 실행

- JDK(Java Development Kit)이라는 프로그램을 다운로드하고 설치해야한다.
- 여러 종류의 JDK가 존재한다. openjdk, Oracle JDK, Azul Julu JDK, Amazon Corretto OpenJDK, Adoptium Temurin 등
- 이클립스 재단(The Eclipse Foundation)의 어댑티움(Adoptium) 프로젝트가 '이클립스 테무린(Eclipse Temurin) 자바 SE 바이너리'의 첫 번째 릴리즈를 출시했다. 이는 인텔 64비트 프로세서 기반 윈도우, 리눅스, 맥OS용 자바 SE 8, 자바 SE 11, 자바 SE 16의 최신 버전을 다루는 오픈JDK(OpenJDK)의 '프로덕션 레디(production-ready)' 빌드다.

# Visual Studio Code 설치

# iTerm 설치

- Mac 사용자일 경우

# Git 설치

- Git 설치 후 다음의 내용을 설정한다.

```
git config --global user.name "이름"
git config --global user.email 이메일
git config --global core.autocrlf true
```

# 터미널에서 Visual Studio Code를 실행할 수 있도록 설정

1. 명령 팔레트를 실행한다.

2. path로 검색한 후, `셸 명령 : PATH에 'code' 명령 설치`를 선택한다.

# JDK 11 설치

- 초보자가 공부할 때는 JDK 8도 충분. 서비스 업체들의 경우 11이상을 사용하는 경우가 많다.
- JDK 17을 고려하는 경우
  - Java 17이 2021년 9월 출시.
  - 10부터는 6개월마다 출시. LTS(Long Term Support)버전은 3년마다 출시.
  - JDK8 LTS가 JDK11 LTS보다 유지보수 기간이 더 길다. 그래서, 직접 17로 업데이트하는 것을 고려
- M1 Mac용 JDK는 17버전에서 지원.
- https://adoptium.net/releases.html?variant=openjdk11&jvmVariant=hotspot
- M1 Mac용 11JDK - Azul Zulu JDK
- https://www.azul.com/downloads/?version=java-11-lts&os=macos&architecture=arm-64-bit&package=jdk

# JDK 11 설치

- 윈도우 사용자는 zip파일을, 맥 사용자는 tar.gz을 다운로드한다.
- 압축을 해제한다. jdk-XXX 폴더가 생성된다. XXX는 버전에 따라 다른 부분을 표현하였다.
- 디렉토리 구조 ( `tree -d -L 3 jdk-XXX` 명령으로 확인)

# JDK 11 설치

- `sudo mv jdk-XXX /Library/Java/JavaVirtualMachines`
- XXX는 버전에 따라 달라지는 부분을 표현하였다.
- sudo 명령은 root관리자 권한으로 실행하라는 명령이다. 본인의 암호를 입력한다.
- 윈도우 사용자는 `c:\\Program Files\` 폴더에 복사한다.

# JDK 11 설치 - Mac사용자

- 터미널에서 `echo $0` 이라고 명령한다.
- zsh라고 나올 경우
  `code ~/.zshrc` 명령을 수행한다.
- bash라고 나올 경우
  `code ~/.bashrc` 명령을 수행한다.
- 파일을 열었으면 마지막 줄에 다음을 추가한다.

```
export JAVA_HOME=$(/usr/libexec/java_home -v 11)
export PATH=$PATH:$JAVA_HOME/bin
```

# JDK 11 설치 - Mac사용자

- 터미널을 종료하고 재시작한다. 다음과 같이 명령한다.

```
java --version
javac -version
```

# JDK 11 설치 - 윈도우 사용자

구글에서 JAVA_HOME PATH 환경변수라고 검색해보자.

https://vmpo.tistory.com/6

# 자바 프로그램 작성과 실행

- 터미널에서 특정 디렉토리로 이동한다. (소스가 저장될 디렉토리)
- `code Hello.java` 을 실행한 후 아래와 같이 저장한다.

```
public class Hello{
    public static void main(String[] args){
        System.out.println("Hello");
    }

}
```

# 기본형 타입과 레퍼런스 타입

# 기본형 타입 vs 레퍼런스 타입

- 기본형 타입(primitive type)
  - 정수, 부동 소수점, 논리 값 등을 나타내는 데이터 타입.
  - 기본형 타입은 메모리 상에 값을 직접 저장한다.
  - 예를 들어, int는 32 비트 정수 값을 저장하는 데 사용되며, double은 64 비트 부동 소수점 값을 저장하는 데 사용된다.
- 레퍼런스 타입(reference type)
  - 객체를 참조하는 데이터 타입.
  - 이러한 객체는 메모리의 힙(heap) 영역에 저장되며, 이러한 객체의 참조(reference)가 변수에 저장된다.
  - 예를 들어, String은 문자열을 나타내는 레퍼런스 타입이다. String 변수는 문자열 객체의 참조를 저장하며, 실제 문자열 데이터는 힙 메모리에 저장된다.

# 기본형 타입과 Wrapper클래스

- 기본형 타입(primitive type)은 int, double, boolean 등이 있다. (모두 소문자)
- 기본형 타입은 객체로서의 기능을 제한합니다. 이를 보완하기 위해 Wrapper 클래스(wrapper class)가 제공된다.
- Wrapper 클래스는 해당하는 기본형 타입의 값을 포함하며, 이 값을 객체로 래핑(wrapping)하여 객체 지향 프로그래밍에서 더 유용하게 사용할 수 있도록 한다. Wrapper 클래스에는 Integer, Double, Boolean 등이 있다.
- Wrapper 클래스를 사용하면 기본형 타입과 마찬가지로 변수에 값을 저장할 수 있으며, 또한 이러한 변수를 메서드에 전달할 수도 있다. Wrapper 클래스는 자동으로 기본형 타입으로 변환되어 처리된다.

```java
public class WrapperExample {
    public static void main(String[] args) {
        // int 값을 String으로 변환
        int num = 42;
        // Wrapper 클래스인 Integer를 사용하여 int 값을 String으로 변환
        String strNum = Integer.toString(num);
        System.out.println(strNum); // 출력 결과: "42"

        // String 값을 int로 변환
        String strNum2 = "42";
        // Wrapper 클래스인 Integer를 사용하여 String 값을 int로 변환
        int num2 = Integer.parseInt(strNum2);
        System.out.println(num2); // 출력 결과: 42
    }
}
```

```java
public class AutoboxingExample {
    public static void main(String[] args) {
        // autoboxing 예제
        int num1 = 42;
        Integer wrapperNum1 = num1; // int 값을 Integer 객체에 대입 (autoboxing)
        System.out.println(wrapperNum1); // 출력 결과: 42

        // unboxing 예제
        Integer wrapperNum2 = 123;
        int num2 = wrapperNum2; // Integer 객체에 저장된 값을 int 변수에 대입 (unboxing)
        System.out.println(num2); // 출력 결과: 123
    }
}
```

# 자바 프로그램 작성과 실행

- java 컴파일러 javac명령으로 hello.java를 컴파일 한다.

  ```
  javac Hello.java
  ```

- 컴파일이 성공하면 오류메시지가 없이 `Hello.class` 파일이 생성된다.
- `ls -la` 명령으로 파일이 생성되었는지 확인한다.
- JVM(자바 가상 머신)으로 `Hello.class` 를 실행한다. java명령이 JVM을 의미한다. 이 때 확장자는 입력하지 않는다.

  ```
  java Hello
  ```

# 1. 자바 객체지향 문법

자바는 객체 지향 프로그래밍 언어로, 객체 지향적인 문법을 가지고 있습니다. 객체 지향 프로그래밍은 코드를 객체로 구성하고, 이러한 객체들이 서로 상호작용하면서 프로그램을 구성하는 것을 의미합니다.

Java에서는 다음과 같은 객체 지향적인 문법을 사용합니다.

1. 클래스와 객체: 클래스는 객체를 만들기 위한 템플릿으로, 객체는 클래스를 토대로 생성된 인스턴스입니다. 객체는 클래스의 속성을 가지며, 클래스에서 정의된 메소드를 호출하여 작업을 수행할 수 있습니다.
2. 캡슐화: 캡슐화는 객체의 데이터와 메소드를 하나로 묶는 것을 말합니다. 이를 통해 객체의 내부 상태를 외부로부터 보호하고, 재사용성을 높일 수 있습니다.
3. 상속: 상속은 부모 클래스의 속성과 메소드를 자식 클래스에서 재사용하는 것을 말합니다. 이를 통해 코드의 중복을 줄이고, 유지 보수성을 높일 수 있습니다.

4. 다형성: 다형성은 같은 메소드 이름을 가진 다른 기능의 메소드를 만드는 것을 말합니다. 이를 통해 코드의 재사용성과 유연성을 높일 수 있습니다.
5. 추상화: 추상화는 객체의 공통된 특성을 추출하여 클래스로 표현하는 것을 말합니다. 이를 통해 코드의 가독성을 높이고, 유지 보수성을 향상시킬 수 있습니다.
6. 인터페이스: 인터페이스는 클래스의 기능을 정의하는 일종의 추상 클래스입니다. 인터페이스는 다중 상속이 가능하며, 다형성을 구현하는데 사용됩니다.

Java는 이러한 객체 지향적인 문법을 사용하여 프로그램을 작성할 수 있습니다. 이를 통해 코드의 재사용성과 유지 보수성을 높일 수 있습니다.

## 1-1. 클래스

자바에서 클래스는 객체를 생성하기 위한 일종의 설계도 또는 템플릿입니다. 클래스는 필드(Field)와 메소드(Method)의 집합으로 구성되며, 객체의 속성과 행동을 정의합니다.

Java에서 클래스는 다음과 같은 구문으로 정의됩니다.

```
[접근 제어자] class [클래스 이름] {
// 클래스 멤버 (필드(Field), 메소드(Method) 등)
}
```

위의 구문에서 대괄호는 생략 가능한 부분을 나타냅니다.

- 접근 제어자: 클래스의 접근 가능 범위를 지정합니다. public, private, protected, 또는 default 중 하나를 사용할 수 있습니다.
- 클래스 이름: 클래스의 이름을 지정합니다.

예를 들어, 다음은 Person 클래스의 예시입니다.

```
public class Person {
        String name;
        int age;

        public void sayHello() {
                System.out.println("안녕하세요!");
        }
}
```

위의 예시에서 Person 클래스는 public으로 선언되었으며, name과 age라는 두 개의 변수와 sayHello()라는 하나의 메소드를 가지고 있습니다. name과 age는 문자열과 정수 값을 저장하기 위한 변수이며, sayHello()는 "안녕하세요!"라는 메시지를 출력하는 메소드입니다.

- 클래스는 객체를 만들기 위한 설계도이며, 객체의 속성과 행동을 정의하는 변수와 메소드의 집합입니다.

## 1-2. 클래스의 필드(Field)

자바 클래스의 필드는 클래스의 속성을 나타내는 변수입니다. 필드는 클래스 내부에서 선언되며, 클래스의 모든 메소드에서 사용될 수 있습니다.

Java에서 필드는 다음과 같은 구문으로 선언됩니다.

```
[접근 제어자] [static] [데이터 타입] [필드 이름];
```

위의 구문에서 대괄호는 생략 가능한 부분을 나타냅니다.

- 접근 제어자: 필드의 접근 가능 범위를 지정합니다. public, private, protected, 또는 default 중 하나를 사용할 수 있습니다.
- static: 필드가 정적 필드인지 여부를 나타냅니다.
- 데이터 타입: 필드가 저장할 값의 데이터 타입을 지정합니다.
- 필드 이름: 필드의 이름을 지정합니다.

```
public class Person {
private String name;
private int age;

        public String getName() {
            return name;
        }

        public void setName(String name) {
            this.name = name;
        }

        public int getAge() {
            return age;
        }

        public void setAge(int age) {
            this.age = age;
        }

}
```

## 1-3. 메소드(Method)/정적 메소드

```
[접근 제어자] [static] [리턴 타입] [메소드 이름]([매개 변수]) {
// 메소드 실행 내용
}
```

위의 구문에서 대괄호는 생략 가능한 부분을 나타냅니다.

- 접근 제어자: 메소드가 접근 가능한 범위를 지정합니다. public, private, protected, 또는 default 중 하나를 사용할 수 있습니다.
- static: 메소드가 정적 메소드인지 여부를 나타냅니다.
- 리턴 타입: 메소드가 반환하는 값의 타입을 나타냅니다. 반환 값이 없는 경우에는 void를 사용합니다.
- 메소드 이름: 메소드의 이름을 지정합니다.
- 매개 변수: 메소드에 전달되는 인수의 타입과 이름을 지정합니다.

```java
public int addNumbers(int x, int y) {
        int sum = x + y;
        return sum;
}
```

- Java에서 메소드는 코드의 구성 요소 중 하나입니다.
- 메소드는 클래스 내에서 정의되며, 일련의 문장을 실행하는 데 사용됩니다.
- 메소드는 클래스의 특정 기능을 수행하는 코드 블록이며, 클래스의 변수와 상호 작용합니다.

- Java에서 정적 메소드는 클래스에 속하지만 인스턴스에 속하지 않는 메소드입니다.
- 이러한 메소드는 클래스를 초기화하지 않고 직접 호출할 수 있습니다.
- 정적 메소드는 주로 유틸리티 메소드 또는 유틸리티 클래스의 일부로 사용됩니다.

예를 들어, Math 클래스에는 정적 메소드인 abs()가 있습니다. 이 메소드는 주어진 값을 절대값으로 반환합니다. Math.abs(-5)는 5를 반환합니다. 이 메소드는 Math 클래스의 인스턴스를 생성할 필요없이 직접 호출할 수 있습니다.

- 정적 메소드는 클래스에 속하지만 인스턴스와 관련이 없는 메소드입니다.
- 이러한 메소드는 클래스를 초기화하지 않고 직접 호출할 수 있으며, 유틸리티 메소드 또는 유틸리티 클래스에서 주로 사용됩니다.

1. 메소드 파라미터

자바 메소드 파라미터는 메소드에 전달되는 인수를 나타내는 변수입니다. 메소드는 하나 이상의 파라미터를 가질 수 있으며, 각 파라미터는 해당 메소드가 호출될 때 전달된 값에 대한 참조로 사용됩니다.

Java에서 메소드 파라미터는 다음과 같은 구문으로 정의됩니다.

```
[데이터 타입] [파라미터 이름]
```

위의 구문에서 대괄호는 생략 가능한 부분을 나타냅니다.

- 데이터 타입: 파라미터가 저장할 값의 데이터 타입을 지정합니다.
- 파라미터 이름: 파라미터의 이름을 지정합니다.

다음은 addNumbers()라는 메소드가 두 개의 int형 파라미터 x와 y를 가지고 있는 예시입니다.

```
public int addNumbers(int x, int y) {
        int sum = x + y;
        return sum;
}
```

- addNumbers() 메소드는 int형 파라미터 x와 y를 가지고 있습니다.
- 메소드의 실행 내용에서는 x와 y를 더하고 그 결과를 sum 변수에 저장합니다. 마지막으로 sum 값을 반환합니다. 이렇게 파라미터를 사용하여 메소드에 값을 전달할 수 있습니다.

자바 메소드 파라미터는 메소드에 전달되는 인수를 나타내는 변수이며, 각 파라미터는 해당 메소드가 호출될 때 전달된 값에 대한 참조로 사용됩니다.

1. 메소드 파라미터가 0개이고 리턴값이 없는 예제:

```
public class Example {
    public static void main(String[] args) {
        hello();
    }

    public static void hello() {
        System.out.println("Hello, world!");
    }
}
```

2. 메소드 파라미터가 1개이고 리턴값이 없는 예제:

```
public class Example {
    public static void main(String[] args) {
        String name = "John";
        sayHello(name);
    }

    public static void sayHello(String name) {
        System.out.println("Hello, " + name + "!");
    }
}
```

3. 메소드 파라미터가 2개이고 리턴값이 있는 예제:

```
public class Example {
    public static void main(String[] args) {
        int a = 10;
        int b = 20;
        int sum = add(a, b);
        System.out.println("The sum of " + a + " and " + b + " is " + sum);
    }

    public static int add(int x, int y) {
        return x + y;
    }
}
```

## 2. 메소드 동적 파라미터

자바에서 메소드 동적 파라미터는 정해지지 않은 개수의 인수를 전달할 수 있는 기능을 제공합니다. 동적 파라미터는 가변 인수(varargs)를 사용하여 구현됩니다.

Java에서 가변 인수를 사용하려면 메소드 정의에서 데이터 타입 뒤에 "... "을 추가하면 됩니다. 이후에는 메소드 내에서 배열로 처리됩니다. 가변 인수는 단 한 번만 사용할 수 있으며, 가장 마지막에 선언되어야 합니다.

다음은 addNumbers()라는 메소드가 정해지지 않은 개수의 int형 파라미터를 가지는 예시입니다.

```
public int addNumbers(int... numbers) {
        int sum = 0;
        for (int num : numbers) {
                sum += num;
        }
        return sum;
}
```

자바에서 메소드 동적 파라미터는 정해지지 않은 개수의 인수를 전달할 수 있는 기능을 제공합니다. 이를 구현하기 위해 가변 인수(varargs)를 사용합니다.

## 동적 파라미터를 이용한 실행 가능한 예제

```
import java.util.Arrays;
import java.util.List;

public class DynamicParameterExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
        int sum = sum(numbers.toArray(new Integer[0]));
        System.out.println(sum);
    }

    private static int sum(Integer... numbers) {
        int sum = 0;
        for (int number : numbers) {
            sum += number;
        }
        return sum;
    }
}
```

## 1-4. 메소드 오버로딩(Overloading)

자바에서 메소드 오버로딩은 같은 이름의 메소드를 여러 개 정의하는 것을 의미합니다. 이를 통해 메소드 이름을 일관성 있게 유지하면서도, 서로 다른 매개 변수 타입 또는 개수에 따라 다른 동작을 수행할 수 있습니다.

Java에서 메소드 오버로딩은 다음과 같이 선언됩니다.

```
[접근 제어자] [static] [리턴 타입] [메소드 이름]([매개 변수]) {
// 메소드 실행 내용
}
```

다음은 addNumbers()라는 메소드가 int형 두 개의 파라미터와 double형 두 개의 파라미터를 받아들이도록 오버로딩된 예시입니다.

```
public int addNumbers(int x, int y) {
        return x + y;
}

public double addNumbers(double x, double y) {
        return x + y;
}
```

자바에서 메소드 오버로딩은 같은 이름의 메소드를 여러 개 정의하는 것으로, 메소드 이름을 일관성 있게 유지하면서도, 서로 다른 매개 변수 타입 또는 개수에 따라 다른 동작을 수행할 수 있습니다.

자바에서 메소드 오버로딩(overloading)은 같은 이름의 메소드를 여러 개 정의하되, 매개변수의 개수나 타입이 다르도록 하는 것을 말합니다. 따라서 메소드 오버로딩은 다형성(polymorphism)을 구현하는 방법 중 하나입니다.

```java
public class Calculator {
    public int add(int x, int y) {
        return x + y;
    }

    public double add(double x, double y) {
        return x + y;
    }

    public int add(int x, int y, int z) {
        return x + y + z;
    }

    public static void main(String[] args) {
        Calculator calculator = new Calculator();
        System.out.println(calculator.add(1, 2)); // 3
        System.out.println(calculator.add(2.5, 3.7)); // 6.2
        System.out.println(calculator.add(1, 2, 3)); // 6
    }
}
```

같은 이름의 메소드를 다양한 매개변수로 오버로딩하여 다양한 상황에서 호출할 수 있도록 하였습니다.

## 1-5. 클래스 상속

자바에서 클래스 상속은 이미 존재하는 클래스를 기반으로 새로운 클래스를 만드는 기능입니다.
상속을 사용하면 기존 클래스의 속성과 메소드를 재사용하면서 새로운 기능을 추가할 수 있습니다.

```
[접근 제어자] class [새 클래스 이름] extends [상속할 클래스 이름] {
// 클래스 멤버 (변수, 메소드 등)
}
```

위의 구문에서 대괄호는 생략 가능한 부분을 나타냅니다.

- 접근 제어자: 클래스의 접근 가능 범위를 지정합니다. public, private, protected, 또는 default 중 하나를 사용할 수 있습니다.
- 새 클래스 이름: 새로운 클래스의 이름을 지정합니다.
- 상속할 클래스 이름: 상속할 기존 클래스의 이름을 지정합니다.

다음은 Person 클래스를 상속받아 Student 클래스를 만드는 예시입니다.

```
public class Student extends Person {
        private int studentID;

        public int getStudentID() {
            return studentID;
        }

        public void setStudentID(int studentID) {
            this.studentID = studentID;
        }
}
```

자바에서 클래스 상속은 이미 존재하는 클래스를 기반으로 새로운 클래스를 만드는 기능으로, 기존 클래스의 속성과 메소드를 재사용하면서 새로운 기능을 추가할 수 있습니다.

예제)

```
// 부모 클래스
class Animal {
    private String name;

    public Animal(String name) {
        this.name = name;
    }

    public String getName() {
        return this.name;
    }

    public void makeSound() {
        System.out.println("Animal sound");
    }
}
```

```java
// 자식 클래스
class Dog extends Animal {
    public Dog(String name) {
        super(name);
    }

    @Override
    public void makeSound() {
        System.out.println("Bark");
    }
}
```

```java
// 실행 클래스
public class Main {
    public static void main(String[] args) {
        Animal animal = new Animal("Generic Animal");
        System.out.println(animal.getName());
        animal.makeSound();

        Dog dog = new Dog("Fido");
        System.out.println(dog.getName());
        dog.makeSound();
    }
}
```

## 1-6. 메소드 오버라이딩(**Overriding)

자바에서 메소드 오버라이딩은 부모 클래스에 정의된 메소드를 자식 클래스에서 재정의하는 기능입니다. 이를 통해 자식 클래스에서 부모 클래스의 동작을 수정하거나 확장할 수 있습니다.

Java에서 메소드 오버라이딩은 다음과 같이 선언됩니다.

```
[접근 제어자] [리턴 타입] [메소드 이름]([매개 변수]) {
// 메소드 실행 내용
}
```

다음은 Person 클래스의 메소드를 오버라이딩하여 Student 클래스에서 재정의한 예시입니다.

```
public class Person {
        public void print() {
                System.out.println("I am a person.");
        }
}

public class Student extends Person {
        @Override
        public void print() {
                System.out.println("I am a student.");
        }
}
```

자바에서 메소드 오버라이딩은 부모 클래스에 정의된 메소드를 자식 클래스에서 재정의하는 기능으로, 자식 클래스에서 부모 클래스의 동작을 수정하거나 확장할 수 있습니다.

예제)

```
class Animal {
    public void move() {
        System.out.println("동물이 움직입니다.");
    }
}

class Dog extends Animal {
    @Override
    public void move() {
        System.out.println("개가 뛰어갑니다.");
    }
}
```

```java
class Cat extends Animal {
    @Override
    public void move() {
        System.out.println("고양이가 네 발로 걷습니다.");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal animal1 = new Animal();
        Animal dog1 = new Dog();
        Animal cat1 = new Cat();

        animal1.move(); // "동물이 움직입니다." 출력
        dog1.move(); // "개가 뛰어갑니다." 출력
        cat1.move(); // "고양이가 네 발로 걷습니다." 출력
    }
}
```

## 1-7. 패키지(package)

자바에서 패키지는 클래스를 논리적으로 그룹화하는 방법입니다. 패키지를 사용하면 클래스의 이름 충돌을 방지하고 클래스를 관리하기 쉽게 할 수 있습니다.

Java에서 패키지는 다음과 같은 구문으로 정의됩니다.

```
package [패키지 이름];
```

위의 구문에서 패키지 이름은 소문자로 작성되어야 하며, "."으로 구분된 계층적인 이름으로 지정됩니다.

예를 들어, 다음은 mypackage 패키지에 속하는 Person 클래스의 예시입니다.

```
package mypackage;

public class Person {
// 클래스 멤버 (변수, 메소드 등)
}
```

Person 클래스는 mypackage 패키지에 속해 있습니다. 이렇게 패키지를 사용하여 클래스를 논리적으로 그룹화하면, 클래스 이름 충돌을 방지할 수 있으며, 클래스를 관리하기 쉽게 할 수 있습니다.

또한 Java에서는 import 문을 사용하여 다른 패키지에 있는 클래스를 현재 패키지에서 사용할 수 있습니다.

다음은 mypackage 패키지에서 Person 클래스를 사용할 때 다른 패키지의 클래스를 import하여 사용하는 예시입니다.

```
package mypackage;
import otherpackage.OtherClass;

public class Person {
private OtherClass otherClass;

// 클래스 멤버 (변수, 메소드 등)

}
```

## 1-8. 접근 제한자

자바에서 접근 제한자는 클래스, 메소드, 변수 등의 멤버에 대한 접근 가능 범위를 지정하는 기능입니다. Java에서는 네 가지의 접근 제한자가 있습니다.

1. private
   private 접근 제한자는 멤버를 정의한 클래스에서만 접근 가능하도록 제한합니다. 즉, 다른 클래스에서는 해당 멤버에 접근할 수 없습니다.
2. protected
   protected 접근 제한자는 멤버를 정의한 클래스와 그 클래스를 상속받은 자식 클래스에서 접근 가능하도록 제한합니다. 즉, 다른 클래스에서는 해당 멤버에 접근할 수 없지만, 자식 클래스에서는 접근할 수 있습니다.

3. public

   public 접근 제한자는 모든 클래스에서 멤버에 접근 가능하도록 제한을 해제합니다. 즉, 모든 클래스에서 해당 멤버에 접근할 수 있습니다.

4. default

   default 접근 제한자는 접근 제한자를 지정하지 않았을 때의 기본값으로, 같은 패키지 내의 클래스에서만 멤버에 접근 가능하도록 제한합니다. 다른 패키지에 있는 클래스에서는 해당 멤버에 접근할 수 없습니다.

다음은 Person 클래스에 여러 가지 접근 제한자를 사용하여 멤버에 대한 접근 범위를 제한한 예시입니다.

```
public class Person {
        private String name; // 해당 멤버는 Person 클래스 내부에서만 접근 가능합니다.
        protected int age; // 해당 멤버는 Person 클래스와 Person 클래스를 상속받은 자식 클래스에서 접근 가능합니다.
        public String address; // 해당 멤버는 모든 클래스에서 접근 가능합니다.
        String gender; // 해당 멤버는 default 접근 제한자를 사용하고 있으며, 같은 패키지 내의 클래스에서만 접근 가능합니다.
}
```

자바에서 접근 제한자는 멤버에 대한 접근 범위를 제한하는 기능으로, private, protected, public, default 네 가지의 접근 제한자가 있습니다.

예제)

아래는 package와 접근 제한자에 대한 예제 코드입니다.

```java
// PackageTest.java

package com.example;

import com.example.package2.AccessTest;

public class PackageTest {
    public static void main(String[] args) {
        AccessTest obj = new AccessTest();

        // 아래 두 줄의 주석을 바꿔가며 실행해보세요.
        // obj.privateVar = 10; // 에러 발생 - 접근 불가
        // obj.defaultVar = 20; // 에러 발생 - 접근 불가
        obj.protectedVar = 30; // 접근 가능
        obj.publicVar = 40; // 접근 가능
    }
}
```

```java
// AccessTest.java

package com.example.package2;

public class AccessTest {
    private int privateVar;
    int defaultVar;
    protected int protectedVar;
    public int publicVar;
}
```

위 예제는 패키지와 접근 제한자에 대한 이해를 돕기 위한 간단한 예제입니다. 보다 복잡한 예제에서는 패키지와 접근 제한자를 효율적으로 사용하여 코드를 구성할 수 있습니다.

## 1-9. 캡슐화(Encapsulation)

자바에서 캡슐화(Encapsulation)란 객체지향 프로그래밍에서 중요한 개념 중 하나로, 객체의 필드와 메소드를 하나로 묶어 캡슐화하여 외부로부터의 접근을 제한하는 것을 말합니다. 즉, 객체의 내부 상태를 보호하고 외부에서의 직접적인 접근을 제한함으로써 객체의 무결성을 유지하고 객체 간의 결합도를 낮추는 역할을 합니다.

자바에서 캡슐화를 구현하는 방법은 다음과 같습니다.

1. 필드를 private으로 선언하여 외부에서 직접적인 접근을 제한합니다.
2. 메소드를 public으로 선언하여 필드에 대한 간접적인 접근을 허용합니다.
3. Getter와 Setter 메소드를 사용하여 필드에 대한 값을 읽고 쓰는 방법을 제공합니다.

다음은 Person 클래스에서 캡슐화를 적용한 예시입니다.

```
public class Person {
        private String name;
        private int age;

        public String getName() {
            return name;
        }

        public void setName(String name) {
            this.name = name;
        }

        public int getAge() {
            return age;
        }

        public void setAge(int age) {
            if (age >= 0) {
                this.age = age;
            } else {
                System.out.println("나이는 0 이상이어야 합니다.");
            }
        }

}
```

자바에서 캡슐화는 객체의 필드와 메소드를 하나로 묶어 캡슐화하여 외부로부터의 접근을 제한하는 것으로, 객체의 무결성을 유지하고 객체 간의 결합도를 낮추는 역할을 합니다.

## 1-10. 다형성(**Polymorphism)**

자바에서 다형성(Polymorphism)은 객체지향 프로그래밍의 중요한 개념 중 하나로, 하나의 코드가 여러 가지 형태를 가질 수 있는 것을 의미합니다. 즉, 다형성을 사용하면 같은 이름의 메소드나 클래스가 다양한 상황에서 다른 동작을 하도록 구현할 수 있습니다.

자바에서 다형성을 구현하는 방법으로는 메소드 오버로딩과 메소드 오버라이딩이 있습니다.

1. 메소드 오버로딩
   메소드 오버로딩은 같은 이름의 메소드를 여러 개 정의하는 것을 말합니다. 메소드 오버로딩을 사용하면 같은 이름의 메소드를 다른 매개 변수를 사용하여 여러 개 만들 수 있습니다.
2. 메소드 오버라이딩
   메소드 오버라이딩은 부모 클래스에서 정의된 메소드를 자식 클래스에서 재정의하는 것을 말합니다. 즉, 같은 이름의 메소드를 자식 클래스에서 다시 구현하여 부모 클래스에서 정의된 메소드와 다른 동작을 수행하도록 할 수 있습니다.

다음은 간단한 예시입니다.

```
public class Animal {
        public void makeSound() {
        System.out.println("동물이 소리를 냅니다.");
        }
}

public class Dog extends Animal {
        @Override
        public void makeSound() {
        System.out.println("강아지가 멍멍 짖습니다.");
        }
}
```

자바에서 다형성은 하나의 코드가 여러 가지 형태를 가질 수 있는 것으로, 메소드 오버로딩과 메소드 오버라이딩을 사용하여 같은 이름의 메소드나 클래스가 다양한 상황에서 다른 동작을 하도록 구현할 수 있습니다.

## 1-11. 인터페이스(Interface)

자바에서 인터페이스(Interface)는 추상 클래스보다 더 추상화된 개념으로, 메소드와 상수 필드만으로 이루어진 일종의 템플릿(Template)입니다. 인터페이스는 다른 클래스와 마찬가지로 선언되지만, 클래스와 달리 인스턴스를 생성할 수 없으며, 구현할 때는 implements 키워드를 사용합니다.

인터페이스는 다음과 같은 특징을 가지고 있습니다.

1. 추상 메소드만을 포함하며, 메소드 몸체를 가질 수 없습니다.
2. 상수 필드만을 가질 수 있으며, 값을 변경할 수 없습니다.
3. 다중 상속을 지원하며, 여러 개의 인터페이스를 상속받을 수 있습니다.

- 인터페이스를 구현하는 클래스는 반드시 인터페이스에서 정의한 모든 메소드를 구현해야 합니다.
- 인터페이스는 다형성을 구현하는 데 매우 유용하며, 여러 클래스가 공통적으로 구현해야 하는 기능을 표준화하여 일관성 있는 개발을 가능하게 합니다.

자바 인터페이스(Interface)는 다음과 같은 문법으로 정의됩니다.

```
[public] interface 인터페이스명 [extends 부모인터페이스명, ...] {
    // 상수 필드 선언
    [public static final] 데이터타입 상수필드명 = 값;

    // 추상 메소드 선언
    [public] abstract 반환타입 메소드명(매개변수, ...);

    // 디폴트 메소드 선언 (Java 8 이후)
    [public] default 반환타입 메소드명(매개변수, ...) {
        // 메소드 구현
    }

    // 정적 메소드 선언 (Java 8 이후)
    [public] static 반환타입 메소드명(매개변수, ...) {
        // 메소드 구현
    }
}
```

- 인터페이스는 클래스와 마찬가지로 접근 제한자(public, protected, private)를 가질 수 있습니다.
- 상수 필드와 추상 메소드를 선언할 수 있으며, Java 8 이후부터는 디폴트 메소드와 정적 메소드도 선언할 수 있습니다.

상수 필드는 public static final 키워드를 사용하여 선언하며, 값은 선언과 동시에 초기화되어야 합니다.

추상 메소드는 public abstract 키워드를 사용하여 선언하며, 메소드 몸체를 가질 수 없습니다. 따라서 인터페이스를 구현하는 클래스에서는 추상 메소드를 반드시 구현해야 합니다.

디폴트 메소드는 인터페이스에서 기본 구현을 제공하는 메소드로, 구현하는 클래스에서는 이를 그대로 사용하거나 필요에 따라 재정의할 수 있습니다.

정적 메소드는 인터페이스에서 클래스 수준의 유틸리티 메소드를 제공하는데 사용되며, 인터페이스를 구현하는 클래스에서는 이를 그대로 사용할 수 있습니다.

다음은 Shape 인터페이스와 이를 구현하는 Rectangle 클래스의 예시입니다.

```
public interface Shape {
    double getArea();
}

public class Rectangle implements Shape {
    private double width;
    private double height;

    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    @Override
    public double getArea() {
        return width * height;
    }
}
```

위의 예시에서는 Shape 인터페이스에서 getArea() 메소드를 선언하고, Rectangle 클래스에서 이를 구현하여 면적을 계산하도록 구현하였습니다.

예제)

다음은 자바에서 인터페이스의 실행가능한 예제입니다.

```
public interface Flyable {
    void fly();
}

public class Bird implements Flyable {
    @Override
    public void fly() {
        System.out.println("I'm flying!");
    }

    public static void main(String[] args) {
        Flyable f = new Bird();
        f.fly();
    }
}
```

## 1-11. 내부 클래스

자바에서 내부 클래스(Inner Class)는 클래스 내부에 정의된 클래스로, 다른 클래스에서는 접근할 수 없습니다. 내부 클래스는 다른 클래스와 밀접하게 연관되어 있으며, 외부 클래스의 멤버를 쉽게 접근할 수 있기 때문에 유용하게 사용됩니다.

자바에서는 내부 클래스를 다음과 같이 네 가지 유형으로 구분할 수 있습니다.

1. 인스턴스 내부 클래스
   인스턴스 내부 클래스는 외부 클래스의 인스턴스에 종속적이며, 인스턴스를 생성해야만 사용할 수 있습니다. 인스턴스 내부 클래스는 외부 클래스의 멤버에 쉽게 접근할 수 있기 때문에, 외부 클래스의 기능을 확장하는 데 유용하게 사용됩니다.
2. 정적 내부 클래스
   정적 내부 클래스는 외부 클래스의 인스턴스와 독립적이며, 인스턴스를 생성하지 않고도 사용할 수 있습니다. 정적 내부 클래스는 외부 클래스의 정적 멤버에만 접근할 수 있기 때문에, 외부 클래스와 관련된 유틸리티 클래스를 만들 때 유용하게 사용됩니다.

3. 지역 내부 클래스
   지역 내부 클래스는 메소드나 초기화 블록과 같은 지역 범위에 정의된 클래스로, 해당 지역에서만 사용할 수 있습니다. 지역 내부 클래스는 외부 클래스의 멤버에 접근할 수 있지만, 해당 지역 내에서만 사용할 수 있기 때문에 지역 변수와 마찬가지로 지역 내부 클래스의 생명 주기는 해당 지역과 같습니다.
4. 익명 내부 클래스
   익명 내부 클래스는 이름이 없는 내부 클래스로, 일반적으로 인터페이스나 추상 클래스를 구현하는 클래스를 생성할 때 사용됩니다. 익명 내부 클래스는 생성과 동시에 인스턴스가 생성되며, 해당 클래스의 인스턴스를 생성한 코드 블록 내에서만 사용할 수 있습니다.

1. 인스턴스 내부 클래스

인스턴스 내부 클래스(Instance Inner Class)는 클래스 내부에서 다른 클래스를 선언하는 것으로, 특정 인스턴스에 속하는 클래스입니다. 인스턴스 내부 클래스는 외부 클래스의 인스턴스 멤버와 동일한 범위를 가지며, 외부 클래스의 인스턴스 멤버를 자유롭게 사용할 수 있습니다.

다음은 인스턴스 내부 클래스의 예제 코드입니다.

```
public class OuterClass {
    private int outerField = 10;

    public void outerMethod() {
        InnerClass inner = new InnerClass();
        inner.innerMethod();
    }

    public class InnerClass {
        public void innerMethod() {
            System.out.println("outerField = " + outerField);
        }
    }

    public static void main(String[] args) {
        OuterClass outer = new OuterClass();
        outer.outerMethod();
    }
}
```

2. 정적 내부 클래스

정적 내부 클래스(Static Inner Class)는 클래스 내부에서 다른 클래스를 선언하는 것으로, 특정 클래스에 속하는 클래스입니다. 정적 내부 클래스는 외부 클래스의 인스턴스 멤버에 접근할 수 없습니다.

다음은 정적 내부 클래스의 예제 코드입니다.

```
public class OuterClass {
    private static int outerField = 10;

    public static void outerMethod() {
        InnerClass inner = new InnerClass();
        inner.innerMethod();
    }

    public static class InnerClass {
        public void innerMethod() {
            System.out.println("outerField = " + outerField);
        }
    }

    public static void main(String[] args) {
        OuterClass.outerMethod();
    }
}
```

### 3. 지역 내부 클래스

지역 내부 클래스(Local Inner Class)는 메소드나 블록 내부에서 선언된 클래스로, 해당 메소드나 블록 내부에서만 사용할 수 있습니다.

다음은 지역 내부 클래스의 예제 코드입니다.

```
public class OuterClass {
    private int outerField = 10;

    public void outerMethod() {
        class InnerClass {
            public void innerMethod() {
                System.out.println("outerField = " + outerField);
            }
        }

        InnerClass inner = new InnerClass();
        inner.innerMethod();
    }

    public static void main(String[] args) {
        OuterClass outer = new OuterClass();
        outer.outerMethod();
    }
}
```

4. 익명 내부 클래스

익명 내부 클래스(Anonymous Inner Class)는 이름이 없는 클래스로, 클래스 선언과 객체 생성을 동시에 하는 것입니다. 주로 인터페이스를 구현하거나 추상 클래스를 상속하는데 사용됩니다.

다음은 익명 내부 클래스의 예제 코드입니다.

```
public class OuterClass {
    private int outerField = 10;

    public void outerMethod() {
        Runnable runnable = new Runnable() {
            @Override
            public void run() {
                System.out.println("outerField = " + outerField);
            }
        };
        Thread thread = new Thread(runnable);
        thread.start();
    }

    public static void main(String[] args) {
        OuterClass outer = new OuterClass();
        outer.outerMethod();
    }
}
```

# 2. 기본 API

## 2.1. java.lang.Object

Java에서 Object 클래스는 모든 클래스의 최상위 클래스로, 모든 클래스는 Object 클래스에서 상속받은 메소드와 필드를 가지고 있습니다. Object 클래스에서는 다음과 같은 메소드들이 자주 사용됩니다.

1. equals() : 객체의 동등성 비교를 위한 메소드입니다. 객체의 내용이 같은지 비교할 때 사용됩니다.
2. hashCode() : 객체의 해시 코드를 반환하는 메소드입니다. 객체의 내용이 같으면 같은 해시 코드를 반환합니다.
3. toString() : 객체의 문자열 표현을 반환하는 메소드입니다. 디버깅이나 로그 작성 등에 사용됩니다.
4. getClass() : 객체가 속한 클래스의 정보를 반환하는 메소드입니다.
5. clone() : 객체의 복사본을 생성하는 메소드입니다. 객체의 깊은 복사(Deep Copy)를 구현할 때 사용됩니다. 객체의 깊은 복사를 위한 다양한 방법이 있습니다. 예를 들어 객체를 json 문자열로 변환한 후 json문자열을 이용해 다시 인스턴스를 생성하는 방법들이 많이 사용하는 방법입니다. 이때 오픈소스인 Jackson등이 많이 사용됩니다.

예)

다음은 Object 클래스의 메소드를 오버라이딩하는 실행 가능한 완전한 예제입니다.

```
public class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public String toString() {
        return "Person [name=" + name + ", age=" + age + "]";
    }
```

```java
@Override
public boolean equals(Object obj) {
    if (this == obj) return true;
    if (!(obj instanceof Person)) return false;
    Person other = (Person) obj;
    return age == other.age && Objects.equals(name, other.name);
}

@Override
public int hashCode() {
    return Objects.hash(name, age);
}
```

```
    public static void main(String[] args) {
        Person p1 = new Person("John", 30);
        Person p2 = new Person("Mary", 25);
        Person p3 = new Person("John", 30);

        System.out.println(p1); // Person [name=John, age=30]
        System.out.println(p2); // Person [name=Mary, age=25]
        System.out.println(p3); // Person [name=John, age=30]

        System.out.println(p1.equals(p2)); // false
        System.out.println(p1.equals(p3)); // true

        System.out.println(p1.hashCode()); // 23270643
        System.out.println(p2.hashCode()); // 3220237
        System.out.println(p3.hashCode()); // 23270643
    }
}
```

## 2.2. String

Java에서 String 클래스는 문자열을 다루기 위한 클래스입니다. String 클래스는 불변(Immutable) 객체이기 때문에, 한번 생성된 문자열은 변경할 수 없습니다. 이러한 특성은 다음과 같은 장점을 가집니다.

1. 안정성 : 문자열은 불변하기 때문에, 한번 생성된 문자열은 보호되며, 다른 코드에서 무작위로 변경될 수 없습니다.
2. 쓰레드 안전성 : String 클래스는 멀티스레드 환경에서 안전합니다. 여러 스레드에서 동시에 사용해도 문자열 값이 변하지 않기 때문입니다.
3. 성능 : 문자열을 생성할 때마다 새로운 객체를 생성하지 않아도 되기 때문에, 메모리를 절약하고 성능을 향상시킵니다.

String 클래스는 다양한 생성자와 메소드를 제공합니다. 그 중 일부는 다음과 같습니다.

1. 생성자 : String 클래스는 다양한 생성자를 제공합니다. 가장 많이 사용되는 생성자는 문자열을 매개변수로 받아 String 객체를 생성하는 생성자입니다.
2. length() : 문자열의 길이를 반환합니다.
3. charAt() : 문자열에서 지정된 위치의 문자를 반환합니다.
4. substring() : 문자열의 일부분을 추출합니다.
5. equals() : 문자열이 같은지 비교합니다.
6. indexOf() : 문자열에서 지정된 문자열이 처음으로 나타나는 인덱스를 반환합니다.
7. toUpperCase() : 문자열을 대문자로 변환합니다.
8. toLowerCase() : 문자열을 소문자로 변환합니다.

위와 같은 메소드를 활용하여 문자열을 다룰 수 있습니다.

예제)

다음은 String 클래스의 다양한 메소드를 사용하는 실행 가능한 예제입니다.

```
public class StringMethodsExample {
    public static void main(String[] args) {
        // substring() 메소드
        String str1 = "Hello, world!";
        String substr1 = str1.substring(7); // "world!"
        String substr2 = str1.substring(0, 5); // "Hello"

        System.out.println(substr1);
        System.out.println(substr2);
```

```java
// replace() 메소드
String str2 = "Java is awesome!";
String replacedStr = str2.replace("Java", "Python"); // "Python is awesome!"
System.out.println(replacedStr);

// toUpperCase() 메소드
String str3 = "lowercase";
String upperStr = str3.toUpperCase(); // "LOWERCASE"
System.out.println(upperStr);

// split() 메소드
String str4 = "apple,banana,kiwi";
String[] fruits = str4.split(","); // {"apple", "banana", "kiwi"}

for (String fruit : fruits) {
    System.out.println(fruit);
}
```

```
        // trim() 메소드
        String str5 = "  Hello, world!  ";
        String trimmedStr = str5.trim(); // "Hello, world!"
        System.out.println(trimmedStr);

        // equals() 메소드
        String str6 = "Hello";
        String str7 = "hello";

        System.out.println(str6.equals(str7)); // false
        System.out.println(str6.equalsIgnoreCase(str7)); // true
    }
}
```

## 2.3. StringBuilder와 StringBuffer

Java에서 StringBuilder와 StringBuffer 클래스는 문자열을 더하기 위한 클래스입니다. 이들 클래스는 String 클래스와는 달리 가변(Mutable) 객체이기 때문에, 문자열을 추가하거나 수정할 수 있습니다. StringBuilder와 StringBuffer는 서로 비슷하지만, 다음과 같은 차이점이 있습니다.

1. StringBuilder : 단일 스레드 환경에서 사용하기 적합합니다. 동기화 처리가 되어 있지 않기 때문에, 멀티스레드 환경에서는 사용하지 않는 것이 좋습니다.
2. StringBuffer : 멀티스레드 환경에서 안전하게 사용할 수 있습니다. 동기화 처리가 되어 있어서, 여러 스레드에서 동시에 사용해도 안전합니다.

StringBuilder와 StringBuffer는 String 클래스와 유사한 메소드를 제공합니다. 그 중 일부는 다음과 같습니다.

1. append() : 문자열을 덧붙입니다.
2. insert() : 문자열을 삽입합니다.
3. delete() : 문자열을 삭제합니다.
4. reverse() : 문자열을 뒤집습니다.

```java
public class StringBuilderExample {
    public static void main(String[] args) {
        StringBuilder sb = new StringBuilder("Hello");
        System.out.println("Original StringBuilder: " + sb);
        // append 메소드를 사용하여 문자열 추가
        sb.append(" World!");
        System.out.println("After appending: " + sb);
        // insert 메소드를 사용하여 특정 위치에 문자열 삽입
        sb.insert(5, ", ");
        System.out.println("After inserting: " + sb);
        // replace 메소드를 사용하여 특정 범위 내의 문자열 교체
        sb.replace(0, 5, "Hi");
        System.out.println("After replacing: " + sb);
        // delete 메소드를 사용하여 특정 범위 내의 문자열 삭제
        sb.delete(0, 3);
        System.out.println("After deleting: " + sb);
        // reverse 메소드를 사용하여 문자열 뒤집기
        sb.reverse();
        System.out.println("After reversing: " + sb);

        // charAt 메소드를 사용하여 특정 위치의 문자 반환
        System.out.println("Character at index 5: " + sb.charAt(5));

        // length 메소드를 사용하여 문자열 길이 반환
        System.out.println("Length of StringBuilder: " + sb.length());
    }
}
```

```java
public class StringBufferExample {
    public static void main(String[] args) {
        // StringBuffer 객체 생성
        StringBuffer sb = new StringBuffer();

        // append() 메소드를 이용해 문자열 추가
        sb.append("Hello");
        sb.append(" ");
        sb.append("world!");
        System.out.println(sb); // 출력: Hello world!

        // insert() 메소드를 이용해 문자열 삽입
        sb.insert(6, "Java ");
        System.out.println(sb); // 출력: Hello Java world!

        // replace() 메소드를 이용해 문자열 교체
        sb.replace(6, 10, "C++");
        System.out.println(sb); // 출력: Hello C++ world!

        // delete() 메소드를 이용해 문자열 삭제
        sb.delete(6, 9);
        System.out.println(sb); // 출력: Hello world!

        // reverse() 메소드를 이용해 문자열 뒤집기
        sb.reverse();
        System.out.println(sb); // 출력: !dlrow olleH
    }
}
```

## 2.4. Math

Java에서 Math 클래스는 수학적인 계산을 위한 클래스입니다. Math 클래스는 다양한 수학 함수를 제공하며, 주요 메소드는 다음과 같습니다.

1. abs() : 절댓값을 반환합니다.
2. ceil() : 주어진 실수보다 크거나 같은 가장 작은 정수를 반환합니다.
3. floor() : 주어진 실수보다 작거나 같은 가장 큰 정수를 반환합니다.
4. round() : 주어진 실수를 반올림한 정수를 반환합니다.
5. max() : 두 수 중 큰 값을 반환합니다.
6. min() : 두 수 중 작은 값을 반환합니다.
7. pow() : 주어진 수의 거듭제곱 값을 반환합니다.
8. sqrt() : 주어진 수의 제곱근 값을 반환합니다.
9. random() : 0부터 1사이의 랜덤한 실수 값을 반환합니다.

Math 클래스는 이 외에도 삼각함수(sin, cos, tan 등), 로그(log10, log 등), 지수(exponential) 함수 등 다양한 수학 함수를 제공합니다.

Math 클래스는 정적(static) 메소드만을 가지고 있기 때문에, Math 클래스의 객체를 생성하지 않고 바로 사용할 수 있습니다.

Math 클래스는 자주 사용되는 수학 함수들을 제공하는 유틸리티 클래스입니다. 다양한 메소드가 존재하지만 여기서는 일부만 예제로 설명하겠습니다.

1. Math.abs(): 입력값의 절대값을 반환합니다.

```
int a = -10;
int b = Math.abs(a); // b = 10
```

1. Math.max(): 입력값 중에서 가장 큰 값을 반환합니다.

```
int a = 10;
int b = 20;
int c = Math.max(a, b); // c = 20
```

1. Math.min(): 입력값 중에서 가장 작은 값을 반환합니다.

```
int a = 10;
int b = 20;
int c = Math.min(a, b); // c = 10
```

1. Math.sqrt(): 입력값의 제곱근을 반환합니다.

```
double a = 9;
double b = Math.sqrt(a); // b = 3.0
```

1. Math.random(): 0.0 이상 1.0 미만의 난수를 반환합니다.

```
double randomNum = Math.random(); // 0.0 <= randomNum < 1.0
```

위 예제들은 Math 클래스의 일부 메소드를 사용한 것입니다. Math 클래스의 다른 메소드들도 비슷한 방식으로 사용할 수 있습니다.

## 2.5. Wrapper class

Java에서 Wrapper 클래스는 기본 데이터 타입(primitive type)을 객체로 감싸주는 클래스입니다. Wrapper 클래스는 다음과 같은 클래스들이 있습니다.

1. Boolean : boolean 타입을 감싸는 클래스입니다.
2. Byte : byte 타입을 감싸는 클래스입니다.
3. Character : char 타입을 감싸는 클래스입니다.
4. Short : short 타입을 감싸는 클래스입니다.
5. Integer : int 타입을 감싸는 클래스입니다.
6. Long : long 타입을 감싸는 클래스입니다.
7. Float : float 타입을 감싸는 클래스입니다.
8. Double : double 타입을 감싸는 클래스입니다.

Wrapper 클래스를 사용하면, 기본 데이터 타입을 객체로 다룰 수 있습니다. 이를 통해 다양한 기능을 활용할 수 있습니다. Wrapper 클래스는 다음과 같은 메소드를 제공합니다.

1. parseXXX() : 문자열을 해당 타입으로 변환합니다.
2. valueOf() : 기본 데이터 타입의 값을 해당 타입의 Wrapper 객체로 변환합니다.
3. xxxValue() : Wrapper 객체를 해당 타입의 기본 데이터 타입 값으로 변환합니다.
4. compareTo() : 두 값을 비교합니다.

Integer와 Double 클래스는 자바의 Wrapper 클래스 중에서 기본 타입인 int와 double을 객체로 감싸주는 역할을 합니다.

Integer 클래스의 메소드 예제:

```
public class IntegerExample {
    public static void main(String[] args) {
        // parseInt 메소드 사용 예제
        String str = "123";
        int num = Integer.parseInt(str);
        System.out.println(num); // 출력 결과: 123
        // toBinaryString 메소드 사용 예제
        int num2 = 15;
        String binaryStr = Integer.toBinaryString(num2);
        System.out.println(binaryStr); // 출력 결과: 1111
```

```
        // compareTo 메소드 사용 예제
        Integer num3 = 10;
        Integer num4 = 20;
        int result = num3.compareTo(num4);
        if (result < 0) {
            System.out.println(num3 + " is less than " + num4); // 출력 결과: 10 is less than 20
        } else if (result == 0) {
            System.out.println(num3 + " is equal to " + num4);
        } else {
            System.out.println(num3 + " is greater than " + num4);
        }
    }
}
```

## Double 클래스의 메소드 예제:

```
public class DoubleExample {
    public static void main(String[] args) {
        // parseDouble 메소드 사용 예제
        String str = "3.14";
        double num = Double.parseDouble(str);
        System.out.println(num); // 출력 결과: 3.14

        // isNaN 메소드 사용 예제
        Double num2 = Double.NaN;
        System.out.println(Double.isNaN(num2)); // 출력 결과: true

        // compare 메소드 사용 예제
        Double num3 = 10.5;
        Double num4 = 20.3;
        int result = Double.compare(num3, num4);
        if (result < 0) {
            System.out.println(num3 + " is less than " + num4); // 출력 결과: 10.5 is less than 20.3
        } else if (result == 0) {
            System.out.println(num3 + " is equal to " + num4);
        } else {
            System.out.println(num3 + " is greater than " + num4);
        }
    }
}
```

# 3. 예외 처리

자바에서 예외 처리(Exception Handling)란, 예외 상황이 발생할 경우 이를 처리하기 위한 기술입니다. 예외(Exception)란, 프로그램 실행 중에 발생할 수 있는 예상치 못한 상황을 말합니다. 예외는 크게 두 가지로 나뉩니다.

1. 일반 예외 (Checked Exception) : 컴파일러가 체크해주는 예외로, 예측 가능한 예외입니다. 일반 예외가 발생할 가능성이 있는 코드를 사용할 때는 반드시 예외 처리를 해주어야 합니다. 일반 예외는 Exception을 상속받는 Exception이 throw하는 곳에서 발생합니다.
2. 실행 예외 (Unchecked Exception) : 컴파일러가 체크해주지 않는 예외로, 런타임 시 발생합니다. 실행 예외는 예측 불가능한 예외이기 때문에, 예외 처리를 하지 않아도 컴파일이 가능합니다. 실행 예외는 RuntimeException을 상속받는 Exception이 throw하는 곳에서 발생합니다.

예외 처리는 try-catch 문을 사용하여 처리합니다. try 블록 내에서 예외가 발생하면, 해당 예외를 처리하는 catch 블록으로 제어가 이동합니다. 예외 처리를 통해 프로그램이 강제 종료되는 상황을 방지하고, 안정적으로 동작할 수 있도록 합니다.

다음은 예외 처리를 사용하는 예시입니다.

```
try {
    int num1 = 10;
    int num2 = 0;
    int result = num1 / num2;
} catch (ArithmeticException e) {
    System.out.println("0으로 나눌 수 없습니다.");
}
```

try-catch 블록 외에도, finally 블록을 사용하여 예외 발생 여부와 상관없이 항상 실행되는 코드를 작성할 수 있습니다.

```
try {
    // 예외 발생 가능한 코드
} catch (Exception e) {
    // 예외 처리
} finally {
    // 예외 발생 여부와 상관없이 항상 실행되는 코드
}
```

1. catch여러개 사용되기

자바에서는 하나의 try 블록에 여러 개의 catch 블록을 사용하여, 다양한 예외를 처리할 수 있습니다. 이를 catch 다중화(multi-catch)라고 합니다.

catch 다중화는 다음과 같은 문법으로 작성됩니다.

```
csharpCopy code
try {
    // 예외 발생 가능한 코드
} catch (ExceptionType1 e1) {
    // 예외 처리1
} catch (ExceptionType2 e2) {
    // 예외 처리2
} catch (ExceptionType3 e3) {
    // 예외 처리3
}
```

catch 다중화는 코드의 가독성을 높일 수 있고, 예외 처리를 더욱 세밀하게 제어할 수 있도록 합니다. 하지만 예외 처리를 과도하게 다중화하면 코드의 복잡도가 증가할 수 있으므로, 적절한 예외 처리를 수행하는 것이 중요합니다.

주의할 점은 윗쪽에 있는 ExceptionType이 아랫쪽에 있는 ExceptionType보다 상위 Exception이 있으면 안됩니다.

## 2. throw와 throws

자바 예외처리에서 throw와 throws는 예외를 처리하는 방법 중 하나입니다.

1. throw : 예외를 강제로 발생시키는 키워드입니다. throw 키워드를 사용하면, 원하는 시점에 예외를 발생시킬 수 있습니다.

```
public void divide(int num1, int num2) throws ArithmeticException {
    if (num2 == 0) {
        throw new ArithmeticException("0으로 나눌 수 없습니다.");
    }
    int result = num1 / num2;
}
```

2. throws : 메소드 선언부에 사용되는 키워드입니다. throws 키워드를 사용하면, 해당 메소드에서 발생할 수 있는 예외를 호출하는 쪽에서 처리하도록 미룰 수 있습니다.

```
public void divide(int num1, int num2) throws ArithmeticException {
    if (num2 == 0) {
        throw new ArithmeticException("0으로 나눌 수 없습니다.");
    }
    int result = num1 / num2;
}

public static void main(String[] args) {
    try {
        divide(10, 0);
    } catch (ArithmeticException e) {
        System.out.println("0으로 나눌 수 없습니다.");
    }
}
```

throw와 throws는 예외를 처리하는 방법 중 일부입니다. 적절한 예외 처리를 통해 프로그램의 안정성을 높이고, 예상치 못한 문제를 방지할 수 있습니다.

## 2. 사용자 정의 Exception

자바에서는 기본적으로 제공되는 예외 클래스 외에도, 사용자가 직접 예외 클래스를 정의할 수 있습니다. 이를 사용자 정의 예외(User-defined Exception)라고 합니다.

사용자 정의 예외는 기존의 예외 클래스를 상속받아 새로운 예외 클래스를 만드는 것으로, 다음과 같은 방법으로 정의됩니다.

```
public class CustomException extends Exception {
    public CustomException() {
        super();
    }

    public CustomException(String message) {
        super(message);
    }
}
```

사용자 정의 예외를 사용하는 방법은 다음과 같습니다.

```
public void divide(int num1, int num2) throws CustomException {
    if (num2 == 0) {
        throw new CustomException("0으로 나눌 수 없습니다.");
    }
    int result = num1 / num2;
}
```

사용자 정의 예외는 프로그램의 안정성을 높일 수 있습니다. 기존 예외 클래스로는 표현하기 어려운 예외 상황에 대해서, 직접 새로운 예외 클래스를 정의하여 처리할 수 있습니다.

예)

```
import java.util.Scanner;

public class ExceptionHandlingExample {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the first number: ");
        int num1 = scanner.nextInt();
        System.out.print("Enter the second number: ");
        int num2 = scanner.nextInt();
        try {
            int result = divide(num1, num2);
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
            System.out.println("Cannot divide by zero.");
        }
    }
```

```
    public static int divide(int num1, int num2) throws ArithmeticException {
        if (num2 == 0) {
            throw new ArithmeticException("Cannot divide by zero.");
        }
        return num1 / num2;
    }
}
```

이 예제 코드는 Scanner를 사용하여 두 개의 정수를 입력받고, 입력된 두 개의 숫자를 divide() 메소드로 전달하여 나눗셈 연산을 수행합니다. divide() 메소드에서는 두 번째 인자가 0인 경우 예외를 발생시키도록 처리하고 있습니다. 이 예제에서는 try-catch 문을 사용하여 예외 처리를 하고 있습니다. 만약 예외가 발생하면 catch 블록에서 해당 예외를 처리합니다. 이 예제에서는 ArithmeticException 예외가 발생하면 "Cannot divide by zero."라는 메시지를 출력합니다.

ArithmeticException은 RuntimeException이기 때문에 throws ArithmeticExceptions는 생략 가능합니다.

# 4. 컬렉션 프레임워크

자바 컬렉션 프레임워크(Collection Framework)는 자바에서 데이터를 저장하는 자료구조를 표준화한 것입니다. 컬렉션 프레임워크는 다양한 자료구조를 제공하여, 데이터를 보다 효율적으로 처리할 수 있도록 합니다.

컬렉션 프레임워크에는 다음과 같은 인터페이스와 클래스들이 포함되어 있습니다.

1. Collection 인터페이스 : 자료구조의 기본적인 동작을 정의한 인터페이스입니다. List, Set, Queue 등의 인터페이스가 이를 상속받아 구현됩니다.
2. List 인터페이스 : 순서가 있는 데이터의 집합을 나타내는 인터페이스입니다. ArrayList, LinkedList, Vector 등이 이를 구현합니다.
3. Set 인터페이스 : 중복을 허용하지 않는 데이터의 집합을 나타내는 인터페이스입니다. HashSet, TreeSet 등이 이를 구현합니다.
4. Map 인터페이스 : 키-값 쌍으로 데이터를 관리하는 자료구조를 나타내는 인터페이스입니다. HashMap, TreeMap 등이 이를 구현합니다.

컬렉션 프레임워크를 사용하면, 데이터를 효율적으로 관리할 수 있으며, 불필요한 코드를 줄일 수 있습니다. 또한, 다양한 알고리즘과 연동하여 사용할 수 있어, 보다 복잡한 데이터 처리도 가능해집니다.

## 4-1. Collection

Collection 인터페이스는 자바에서 제공하는 컬렉션 프레임워크의 기본 인터페이스 중 하나입니다. Collection 인터페이스는 다음과 같은 메소드를 정의하고 있습니다.

1. boolean add(Object o) : 컬렉션에 객체를 추가합니다.
2. boolean addAll(Collection c) : 다른 컬렉션의 모든 객체를 현재 컬렉션에 추가합니다.
3. void clear() : 컬렉션의 모든 객체를 제거합니다.
4. boolean contains(Object o) : 컬렉션에 지정된 객체가 포함되어 있는지 여부를 반환합니다.

5. boolean isEmpty() : 컬렉션이 비어있는지 여부를 반환합니다.
6. Iterator iterator() : 컬렉션에 저장된 객체를 순회하는 Iterator 객체를 반환합니다.
7. boolean remove(Object o) : 컬렉션에서 지정된 객체를 제거합니다.
8. boolean removeAll(Collection c) : 컬렉션에서 다른 컬렉션에 포함된 모든 객체를 제거합니다.
9. boolean retainAll(Collection c) : 컬렉션에서 다른 컬렉션에 포함된 객체만 유지하고 나머지는 제거합니다.
10. int size() : 컬렉션에 저장된 객체의 수를 반환합니다.
11. Object[] toArray() : 컬렉션에 저장된 객체를 배열로 반환합니다.

Collection 인터페이스를 구현하는 대표적인 클래스로는 다음과 같은 것이 있습니다.

1. ArrayList : 순서가 있는 객체의 리스트를 구현하는 클래스입니다.
2. LinkedList : 이중 연결 리스트를 구현하는 클래스입니다.
3. HashSet : 중복되지 않은 객체들을 저장하는 클래스입니다.
4. TreeSet : 객체를 정렬하여 저장하는 클래스입니다.
5. PriorityQueue : 우선순위 큐를 구현하는 클래스입니다.

Collection 인터페이스를 구현하는 클래스들은 다양한 자료구조를 구현하고 있으며, 필요에 따라 다른 클래스로 대체할 수도 있습니다. 이를 통해 유연한 자료구조 구현이 가능해집니다.

## 4-2. Iterator

Iterator 인터페이스는 컬렉션의 요소를 순회하면서, 각 요소에 접근할 수 있는 메소드를 정의하는 인터페이스입니다. Iterator 인터페이스는 다음과 같은 메소드를 정의하고 있습니다.

1. boolean hasNext() : 순회할 다음 요소가 있는지 여부를 반환합니다.
2. Object next() : 다음 요소를 반환하고, 커서를 다음 위치로 이동합니다.
3. void remove() : 마지막으로 반환된 요소를 제거합니다.

Iterator 인터페이스를 구현하는 대표적인 클래스로는 다음과 같은 것이 있습니다.

1. ArrayList의 Iterator : ArrayList에서 Iterator를 반환하는 메소드인 iterator()를 호출하여 사용합니다.

```
ArrayList<String> list = new ArrayList<String>();
// 리스트에 요소 추가
Iterator<String> iter = list.iterator(); // Iterator 객체 생성
while(iter.hasNext()) { // 다음 요소가 있는지 확인
    String str = iter.next(); // 다음 요소 반환
    System.out.println(str);
}
```

1. HashSet의 Iterator : HashSet에서 Iterator를 반환하는 메소드인 iterator()를 호출하여 사용합니다.

```
HashSet<String> set = new HashSet<String>();
// set에 요소 추가
Iterator<String> iter = set.iterator(); // Iterator 객체 생성
while(iter.hasNext()) { // 다음 요소가 있는지 확인
    String str = iter.next(); // 다음 요소 반환
    System.out.println(str);
}
```

Iterator 인터페이스는 Collection 인터페이스와 함께 사용하여, 컬렉션의 모든 요소에 접근하고 처리하는 데 유용합니다. Iterator를 사용하면 컬렉션 내부의 데이터 구조에 관계없이 요소에 접근할 수 있으므로, 코드의 유연성과 확장성이 높아집니다.

## 4-3. List

List 인터페이스는 순서가 있는 데이터의 집합을 나타내는 인터페이스 중 하나입니다. List는 Collection 인터페이스를 상속받으며, 다음과 같은 메소드를 추가로 정의하고 있습니다.

1. void add(int index, Object element) : 지정된 위치에 객체를 추가합니다.
2. boolean addAll(int index, Collection c) : 지정된 위치에 컬렉션의 모든 요소를 추가합니다.

3. Object get(int index) : 지정된 위치에 있는 객체를 반환합니다.
4. int indexOf(Object o) : 지정된 객체의 인덱스를 반환합니다.
5. int lastIndexOf(Object o) : 지정된 객체가 마지막으로 나타나는 인덱스를 반환합니다.
6. ListIterator listIterator() : List의 요소를 반복하는 ListIterator 객체를 반환합니다.
7. ListIterator listIterator(int index) : 지정된 위치부터 List의 요소를 반복하는 ListIterator 객체를 반환합니다.
8. Object remove(int index) : 지정된 위치에 있는 객체를 제거합니다.
9. Object set(int index, Object element) : 지정된 위치에 있는 객체를 새로운 객체로 대체합니다.
10. List subList(int fromIndex, int toIndex) : 지정된 범위 내의 List를 반환합니다.

List 인터페이스를 구현하는 대표적인 클래스로는 다음과 같은 것이 있습니다.

1. ArrayList : 배열을 기반으로 구현된 동적 크기의 리스트입니다. 임의의 위치에 객체를 삽입하거나 삭제할 때는 빈번한 객체 복사가 일어나기 때문에 속도가 느리지만, 인덱스를 이용한 접근이 빠르고, 순차적인 접근과 수정에는 우수한 성능을 보입니다.
2. LinkedList : 이중 연결 리스트를 기반으로 구현된 리스트입니다. 임의의 위치에 객체를 삽입하거나 삭제할 때는 객체 복사가 없으므로 빠른 속도를 보입니다. 하지만 인덱스를 이용한 접근은 느리기 때문에 순차적인 접근과 수정에 적합합니다.

List 인터페이스는 자바에서 가장 많이 사용되는 자료구조 중 하나입니다. 다른 자료구조와 달리 순서가 있고, 중복을 허용하는 특징을 가지고 있습니다. 이를 통해 다양한 데이터 처리에 활용할 수 있습니다.

```java
import java.util.ArrayList;
import java.util.List;

public class ListExample {
    public static void main(String[] args) {
        // ArrayList 생성
        List<String> list = new ArrayList<>();

        // 요소 추가
        list.add("Java");
        list.add("Python");
        list.add("C++");
        list.add("JavaScript");

        // 요소 접근
        System.out.println(list.get(0)); // Java

        // 요소 수정
        list.set(1, "Ruby");
        System.out.println(list); // [Java, Ruby, C++, JavaScript]

        // 요소 삭제
        list.remove(2);
        System.out.println(list); // [Java, Ruby, JavaScript]

        // 리스트 크기 확인
        System.out.println(list.size()); // 3

        // 리스트 순회
        for (String str : list) {
            System.out.println(str);
        }
    }
}
```

## 4-4. Set

Set 인터페이스는 순서가 없고, 중복을 허용하지 않는 데이터의 집합을 나타내는 인터페이스 중 하나입니다. Set은 Collection 인터페이스를 상속받으며, 다음과 같은 메소드를 정의하고 있습니다.

1. boolean add(Object o) : Set에 객체를 추가합니다.
2. boolean addAll(Collection c) : 다른 컬렉션의 모든 객체를 현재 Set에 추가합니다.
3. void clear() : Set의 모든 객체를 제거합니다.
4. boolean contains(Object o) : Set에 지정된 객체가 포함되어 있는지 여부를 반환합니다.

5. boolean isEmpty() : Set이 비어있는지 여부를 반환합니다.
6. Iterator iterator() : Set에 저장된 객체를 순회하는 Iterator 객체를 반환합니다.
7. boolean remove(Object o) : Set에서 지정된 객체를 제거합니다.
8. boolean removeAll(Collection c) : Set에서 다른 컬렉션에 포함된 모든 객체를 제거합니다.
9. boolean retainAll(Collection c) : Set에서 다른 컬렉션에 포함된 객체만 유지하고 나머지는 제거합니다.
10. int size() : Set에 저장된 객체의 수를 반환합니다.
11. Object[] toArray() : Set에 저장된 객체를 배열로 반환합니다.

Set 인터페이스를 구현하는 대표적인 클래스로는 다음과 같은 것이 있습니다.

1. HashSet : 해시 테이블을 이용하여 Set을 구현하는 클래스입니다. 내부적으로는 HashMap을 사용하며, 순서가 없고 중복을 허용하지 않습니다.
2. TreeSet : 이진 검색 트리를 이용하여 Set을 구현하는 클래스입니다. 객체를 정렬하여 저장하기 때문에, 저장된 객체들이 오름차순으로 정렬되어 저장됩니다.

Set은 중복된 데이터를 허용하지 않기 때문에 데이터 처리의 효율성을 높일 수 있습니다. 또한 순서가 없는 특성을 이용하여, 데이터를 순회하거나 검색하는 데 높은 성능을 보입니다.

```java
import java.util.HashSet;
import java.util.Iterator;

public class SetExample {
    public static void main(String[] args) {
        // HashSet 객체 생성
        HashSet<String> set = new HashSet<String>();

        // 요소 추가
        set.add("apple");
        set.add("banana");
        set.add("orange");

        // 요소 개수 출력
        System.out.println("Size of set: " + set.size());

        // 요소 순회
        System.out.println("Elements of set:");
        Iterator<String> iterator = set.iterator();
        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }
```

```
        // 요소 검색
        System.out.println("Contains banana: " + set.contains("banana"));
        System.out.println("Contains grape: " + set.contains("grape"));

        // 요소 삭제
        set.remove("banana");
        System.out.println("Size of set after removing banana: " + set.size());
    }
}
```

HashSet 클래스를 이용하여 HashSet 객체를 생성합니다. 그리고 add() 메소드를 이용하여 요소를 추가하고, size() 메소드를 이용하여 요소 개수를 출력합니다. 또한, iterator() 메소드를 이용하여 요소를 순회하고, contains() 메소드를 이용하여 요소를 검색하며, remove() 메소드를 이용하여 요소를 삭제합니다.

## 4-5. Map

Map 인터페이스는 키-값(key-value) 쌍의 데이터를 저장하는 자료구조를 나타내는 인터페이스입니다. Map은 Collection 인터페이스를 상속받지 않으며, 다음과 같은 메소드를 정의하고 있습니다.

1. void clear() : Map의 모든 객체를 제거합니다.
2. boolean containsKey(Object key) : 지정된 키가 Map에 포함되어 있는지 여부를 반환합니다.
3. boolean containsValue(Object value) : 지정된 값이 Map에 포함되어 있는지 여부를 반환합니다.
4. Set entrySet() : Map에 포함된 키-값 쌍을 Set 형태로 반환합니다.

5. Object get(Object key) : 지정된 키에 해당하는 값(value)을 반환합니다.
6. boolean isEmpty() : Map이 비어있는지 여부를 반환합니다.
7. Set keySet() : Map에 포함된 모든 키를 Set 형태로 반환합니다.
8. Object put(Object key, Object value) : 지정된 키-값 쌍을 Map에 추가합니다.
9. void putAll(Map t) : 다른 Map에 있는 모든 키-값 쌍을 현재 Map에 추가합니다.
10. Object remove(Object key) : 지정된 키에 해당하는 키-값 쌍을 Map에서 제거합니다.
11. int size() : Map에 포함된 키-값 쌍의 수를 반환합니다.
12. Collection values() : Map에 포함된 모든 값(value)을 Collection 형태로 반환합니다.

Map 인터페이스를 구현하는 대표적인 클래스로는 다음과 같은 것이 있습니다.

1. HashMap : 해시 테이블을 기반으로 구현된 클래스로, 순서가 없고 중복된 키는 허용하지 않습니다.
2. TreeMap : 이진 검색 트리를 기반으로 구현된 클래스로, 키를 기준으로 정렬됩니다.
3. LinkedHashMap : 해시 테이블과 연결 리스트를 기반으로 구현된 클래스로, 순서가 유지됩니다.

Map 인터페이스를 사용하여 키-값 쌍의 데이터를 처리할 수 있으며, 각각의 구현 클래스는 다양한 상황에 적합한 성격을 가지고 있습니다.

```java
import java.util.HashMap;
import java.util.Map;

public class MapExample {
    public static void main(String[] args) {
        // HashMap 인스턴스 생성
        Map<String, Integer> hashMap = new HashMap<>();

        // put 메소드를 사용하여 key-value pair 추가
        hashMap.put("John", 25);
        hashMap.put("Jane", 30);
        hashMap.put("Steve", 27);

        // get 메소드를 사용하여 value 검색
        System.out.println("Age of John: " + hashMap.get("John"));
```

```
        // containsKey 메소드를 사용하여 key 존재 여부 확인
        if (hashMap.containsKey("Jane")) {
            System.out.println("Jane is in the map!");
        }

        // remove 메소드를 사용하여 key-value pair 삭제
        hashMap.remove("Steve");

        // entrySet 메소드를 사용하여 모든 key-value pair 출력
        for (Map.Entry<String, Integer> entry : hashMap.entrySet()) {
            System.out.println("Key: " + entry.getKey() + ", Value: " + entry.getValue());
        }

        // clear 메소드를 사용하여 모든 key-value pair 삭제
        hashMap.clear();

        // size 메소드를 사용하여 현재 Map의 크기 출력
        System.out.println("Size of the Map: " + hashMap.size());
    }
}
```

# 5. 람다 표현식(Lambda Expression)

## 5-1. 자바의 함수형 프로그래밍

함수형 프로그래밍(Functional Programming)은 프로그래밍 패러다임 중 하나로, 함수를 일급 객체로 취급하여 프로그래밍하는 것을 의미합니다. 이는 명령형 프로그래밍(Imperative Programming)에서는 변수와 상태를 중심으로 프로그래밍하는 것과 대조적입니다.

자바에서 함수형 프로그래밍은 자바 8부터 지원되는 람다 표현식(Lambda Expression)을 통해 구현할 수 있습니다. 람다 표현식은 함수를 변수에 할당하거나 다른 함수의 인자로 전달할 수 있게 해줍니다.

함수형 프로그래밍은 부수 효과(Side Effect)를 최소화하여 코드의 가독성과 유지보수성을 높이는 장점이 있습니다. 또한 병렬 처리에 적합하다는 장점도 있습니다.

반면, 명령형 프로그래밍은 변수와 상태를 중심으로 코드를 작성하기 때문에 부수 효과가 발생할 가능성이 높아 코드의 예측성이 떨어지는 단점이 있습니다. 하지만 명령형 프로그래밍은 효율적인 코드를 작성할 수 있다는 장점도 있습니다.

두 프로그래밍 패러다임은 각각의 특징과 장단점이 있으며, 상황에 따라 적합한 패러다임을 선택하여 프로그래밍하는 것이 중요합니다.

1급 객체(First-class Object 또는 First-class Citizen)란, 프로그래밍 언어에서 일반적인 값과 같이 다룰 수 있는 객체를 말합니다. 즉, 변수에 할당하거나 함수의 인자로 전달할 수 있으며, 함수의 반환 값이 될 수 있습니다.

자바에서는 모든 객체가 1급 객체입니다. 이는 객체를 변수에 할당하거나 다른 객체의 필드로 사용할 수 있다는 것을 의미합니다. 또한 객체를 함수의 인자로 전달하거나 반환 값으로 사용할 수 있습니다. 자바의 객체는 1급 객체이지만, 함수만 별도로 존재할 수 없습니다. 함수만 따로 만들어서 함수를 변수에 할당하거나 메소드의 인자로 전달할 수 없습니다. 이런 문제를 해결하기 위해 등장한 것이 람다 표현식입니다.

1급 객체의 특징은 다음과 같습니다.

1. 변수에 할당할 수 있어야 합니다.
2. 함수의 인자로 전달할 수 있어야 합니다.
3. 함수의 반환 값이 될 수 있어야 합니다.

이러한 특징으로 인해, 1급 객체는 함수형 프로그래밍에서 매우 중요한 개념입니다.

## 5-2. 람다식이란?

자바의 람다식(Lambda Expression)은 함수형 프로그래밍에서 사용되는 개념으로, 익명 함수(Anonymous Function)를 만들어서 사용하는 것을 말합니다.

람다식은 인터페이스에 대한 구현을 간결하게 표현할 수 있도록 하는 것이 주요한 목적입니다. 기존에는 인터페이스를 구현하려면 클래스를 정의하고 메소드를 구현하는 등 번거로운 작업이 필요했습니다. 하지만 람다식을 사용하면 인터페이스를 구현할 때 불필요한 코드를 줄일 수 있습니다.

람다식은 다음과 같은 형태로 사용됩니다.

```
(parameter) -> { code }
```

(parameter)는 메소드의 파라미터를 나타내며, ->는 람다 연산자입니다. 그리고 { code }는 메소드의 구현 코드를 나타냅니다.

예를 들어, Runnable 인터페이스를 구현하는 람다식은 다음과 같이 표현할 수 있습니다.

```
Runnable runnable = () -> { System.out.println("Hello World"); };
```

위의 람다식을 기존의 익명 클래스 방식으로 작성하면 다음과 같습니다.

```
Runnable runnable = new Runnable(){
        public void run(){
      System.out.println("Hello World");
  }
};
```

new Runnable(){ public void run 까지 삭제하고, () 뒤에 → 를 붙이고, 맨 뒤쪽의 }; 를 삭제한 형태라고 생각하면 됩니다.

## 5-3. 자바가 기본으로 제공하는 람다 인터페이스

자바에서 기본적으로 제공하는 람다 인터페이스는 java.util.function 패키지 내에 위치하고 있습니다. 대표적으로 사용되는 인터페이스는 다음과 같습니다.

1. Consumer<T> : 인자를 받아들이고 반환하지 않는 함수형 인터페이스입니다.
2. Supplier<T> : 인자를 받지 않고 값을 반환하는 함수형 인터페이스입니다.
3. Function<T, R> : 인자를 받아들이고 반환값이 있는 함수형 인터페이스입니다.

4. Predicate<T> : 인자를 받아들이고 boolean 값을 반환하는 함수형 인터페이스입니다.
5. UnaryOperator<T> : 인자를 받아들이고 같은 타입의 값을 반환하는 함수형 인터페이스입니다.
6. BinaryOperator<T> : 인자를 두 개 받아들이고 같은 타입의 값을 반환하는 함수형 인터페이스입니다.

이 외에도 많은 함수형 인터페이스가 존재하며, 필요에 따라 직접 함수형 인터페이스를 정의하여 사용할 수도 있습니다.

## 5-4. Consumer<T>

Consumer<T> 인터페이스는 인자를 받아들이고 반환하지 않는 함수형 인터페이스입니다. accept(T t) 메소드를 가지고 있으며, 이를 구현하여 사용합니다.

다음은 Consumer<T> 인터페이스를 사용한 예제 코드입니다.

```
import java.util.ArrayList;
import java.util.List;
import java.util.function.Consumer;

public class Main {
    public static void main(String[] args) {
        List<String> names = new ArrayList<>();
        names.add("Alice");
        names.add("Bob");
        names.add("Charlie");

        // Consumer를 구현하여 forEach() 메소드에 전달합니다.
        Consumer<String> printName = (String name) -> System.out.println(name);
        names.forEach(printName);
    }
}
```

## 5-5. Supplier<T>

Supplier<T> 인터페이스는 인자를 받지 않고 값을 반환하는 함수형 인터페이스입니다. get() 메소드를 가지고 있으며, 이를 구현하여 사용합니다.

다음은 Supplier<T> 인터페이스를 사용한 예제 코드입니다.

```
import java.util.function.Supplier;

public class Main {
    public static void main(String[] args) {
        // Supplier를 구현하여 get() 메소드에서 값을 반환합니다.
        Supplier<String> getMessage = () -> "Hello, World!";
        System.out.println(getMessage.get());
    }
}
```

## 5-6. Function<T, R>

Function<T, R> 인터페이스는 인자를 받아들이고 반환값이 있는 함수형 인터페이스입니다. apply(T t) 메소드를 가지고 있으며, 이를 구현하여 사용합니다.

다음은 Function<T, R> 인터페이스를 사용한 예제 코드입니다.

```
import java.util.function.Function;

public class Main {
    public static void main(String[] args) {
        // Function을 구현하여 apply() 메소드에서 값을 변환합니다.
        Function<Integer, Integer> doubleNumber = (Integer x) -> x * 2;
        System.out.println(doubleNumber.apply(5));
    }
}
```

## 5-7. Predicate<T>

Predicate<T> 인터페이스는 인자를 받아들이고 boolean 값을 반환하는 함수형 인터페이스입니다. test(T t) 메소드를 가지고 있으며, 이를 구현하여 사용합니다.

다음은 Predicate<T> 인터페이스를 사용한 예제 코드입니다.

```
import java.util.ArrayList;
import java.util.List;
import java.util.function.Predicate;

public class Main {
    public static void main(String[] args) {
        List<String> names = new ArrayList<>();
        names.add("Alice");
        names.add("Bob");
        names.add("Charlie");

        // Predicate를 구현하여 removeIf() 메소드에 전달합니다.
        Predicate<String> isLongName = (String name) -> name.length() > 5;
        names.removeIf(isLongName);

        System.out.println(names);
    }
}
```

## 5-8. UnaryOperator<T>

naryOperator<T> 인터페이스는 하나의 인자를 받아들이고 동일한 타입의 결과를 반환하는 함수형 인터페이스입니다. Function<T, T>와 동일하지만 입력과 출력의 타입이 같습니다.

다음은 UnaryOperator<T> 인터페이스를 사용한 예제 코드입니다.

```
import java.util.function.UnaryOperator;

public class Main {
    public static void main(String[] args) {
        // UnaryOperator를 구현하여 apply() 메소드에서 값을 변환합니다.
        UnaryOperator<Integer> squareNumber = (Integer x) -> x * x;
        System.out.println(squareNumber.apply(5));
    }
}
```

위 코드에서는 UnaryOperator<Integer> 인터페이스를 구현하여 apply() 메소드에서 인자를 제곱하고 있습니다. squareNumber 객체를 생성하고 squareNumber.apply(5)를 호출하여 값을 출력합니다.

위 코드를 실행하면 다음과 같은 결과가 출력됩니다.

```
Copy code
25
```

## 5-9. BinaryOperator<T>

BinaryOperator<T> 인터페이스는 두 개의 인자를 받아들이고 동일한 타입의 결과를 반환하는 함수형 인터페이스입니다. 두 인자와 결과가 모두 같은 타입일 때 사용합니다.

다음은 BinaryOperator<T> 인터페이스를 사용한 예제 코드입니다.

```
import java.util.function.BinaryOperator;

public class Main {
    public static void main(String[] args) {
        // BinaryOperator를 구현하여 apply() 메소드에서 값을 변환합니다.
        BinaryOperator<Integer> sumNumbers = (Integer x, Integer y) -> x + y;
        System.out.println(sumNumbers.apply(5, 7));
    }
}
```

## 5-10. 사용자 정의 람다 인터페이스

자바에서 사용자 정의 람다 인터페이스란, 개발자가 직접 정의한 함수형 인터페이스를 의미합니다. 이를 통해 개발자는 필요에 따라 자신만의 함수형 인터페이스를 정의하고 람다식을 이용하여 구현할 수 있습니다.

다음은 사용자 정의 람다 인터페이스를 정의하고 사용하는 예제 코드입니다.

```
interface Calculator {
    int calculate(int x, int y);
}

public class Main {
    public static void main(String[] args) {
        // 사용자 정의 람다 인터페이스를 구현하여 람다식으로 사용합니다.
        Calculator add = (int x, int y) -> x + y;
        Calculator subtract = (int x, int y) -> x - y;

        int result1 = add.calculate(5, 3);
        int result2 = subtract.calculate(5, 3);

        System.out.println(result1);
        System.out.println(result2);
    }
}
```

Java에서 기본적으로 제공하는 함수형 인터페이스 중 BiFunction<T, U, R> 인터페이스는 두 개의 입력 인자를 받아들이고 하나의 결과를 반환하는 함수형 인터페이스입니다. 이를 이용하여 위 예제를 수정할 수 있습니다.

다음은 BiFunction<T, U, R> 인터페이스를 사용한 예제 코드입니다.

```java
import java.util.function.BiFunction;

public class Main {
    public static void main(String[] args) {
        // BiFunction<T, U, R> 인터페이스를 구현하여 람다식으로 사용합니다.
        BiFunction<Integer, Integer, Integer> add = (x, y) -> x + y;
        BiFunction<Integer, Integer, Integer> subtract = (x, y) -> x - y;

        int result1 = add.apply(5, 3);
        int result2 = subtract.apply(5, 3);

        System.out.println(result1);
        System.out.println(result2);
    }
}
```

이렇게 기본적으로 제공하는 함수형 인터페이스가 있기 때문에, 새롭게 함수형 인터페이스를 만들일은 많지는 않습니다.

## 6. 자바 스트림(Stream)

Java 8부터 도입된 스트림(Stream)은 데이터의 처리를 추상화한 것으로, 컬렉션, 배열 등의 데이터 소스를 추상화하여 데이터를 다루는데 유용하게 사용됩니다. 스트림은 다양한 연산을 지원하며, 병렬 처리가 가능하기 때문에 대용량 데이터 처리에 유용합니다.

스트림은 다음과 같은 특징을 가지고 있습니다.

- 소스 데이터를 변경하지 않습니다.
- 중간 연산과 최종 연산으로 구성됩니다.
- 지연 연산을 수행합니다.
- 병렬 처리가 가능합니다.

스트림은 다음과 같은 예제 코드를 통해 사용될 수 있습니다.

```
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

        int sum = numbers.stream()
                         .filter(n -> n % 2 == 0)  // 중간 연산: 짝수만 선택
                         .mapToInt(n -> n)         // 중간 연산: Integer를 int로 변환
                         .sum();                    // 최종 연산: 합계 계산

        System.out.println(sum);
    }
}
```

스트림은 데이터 처리를 편리하게 하기 위해 도입된 기능으로, 복잡한 데이터 처리를 쉽게 수행할 수 있습니다.

## 6-1. 중간 연산 메소드

스트림에서 중간 연산은 스트림 요소들을 변환, 필터링, 정렬 등의 연산을 수행하는 것입니다. 중간 연산 메소드들은 스트림을 반환하므로, 연속적으로 여러 개의 중간 연산을 수행할 수 있습니다.

스트림에서 제공하는 중간 연산 관련 메소드는 다음과 같습니다.

- filter(Predicate<T>): 조건에 맞는 요소만 남깁니다.
- map(Function<T, R>): 요소들을 변환합니다.
- flatMap(Function<T, Stream<R>>): 스트림을 평면화합니다.
- distinct(): 중복된 요소를 제거합니다.
- sorted(): 요소들을 정렬합니다.
- peek(Consumer<T>): 요소들을 소비하면서 중간 결과를 출력합니다.
- limit(long): 요소 개수를 제한합니다.
- skip(long): 처음 요소부터 지정한 개수만큼 생략합니다.

각각의 메소드들은 다양한 기능을 수행하며, 이를 이용하여 스트림 요소들을 변환하거나 필터링하는 등 다양한 데이터 처리를 수행할 수 있습니다.

다음은 filter() 메소드를 이용하여 리스트에서 짝수인 요소들만 남기는 예제 코드입니다.

```
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

        List<Integer> evenNumbers = numbers.stream()
                                           .filter(n -> n % 2 == 0)
                                           .collect(Collectors.toList());

        System.out.println(evenNumbers);
    }
}
```

중간 연산 메소드들은 스트림 요소들을 변환, 필터링, 정렬 등 다양한 연산을 수행할 수 있으므로, 적절히 사용하여 데이터 처리를 쉽게 수행할 수 있습니다.

## 6-2. 지연 연산 메소드

스트림에서 지연 연산은 중간 연산으로 스트림 요소들을 변환, 필터링, 정렬 등의 연산을 수행하는 것을 말합니다. 지연 연산 메소드들은 중간 연산 메소드들과 마찬가지로 스트림을 반환하므로, 연속적으로 여러 개의 지연 연산을 수행할 수 있습니다.

스트림에서 제공하는 지연 연산 관련 메소드는 다음과 같습니다.

- mapToInt(ToIntFunction<T>): int 스트림으로 변환합니다.
- mapToLong(ToLongFunction<T>): long 스트림으로 변환합니다.
- mapToDouble(ToDoubleFunction<T>): double 스트림으로 변환합니다.
- boxed(): 박싱된 스트림으로 변환합니다.
- flatMapToInt(Function<T, IntStream>): int 스트림을 평면화합니다.
- flatMapToLong(Function<T, LongStream>): long 스트림을 평면화합니다.
- flatMapToDouble(Function<T, DoubleStream>): double 스트림을 평면화합니다.
- asLongStream(): long 스트림으로 변환합니다.
- asDoubleStream(): double 스트림으로 변환합니다.

지연 연산 메소드들은 지연 연산을 수행하며, 반환값으로 스트림을 반환합니다. 이러한 지연 연산 메소드들은 스트림의 데이터를 변환, 필터링, 정렬 등의 연산을 지연시켜 처리할 수 있으며, 이를 이용하여 메모리를 효율적으로 사용할 수 있습니다.

다음은 mapToInt() 메소드를 이용하여 리스트에서 정수 요소들을 int 형으로 변환하고, 이를 합산하는 예제 코드입니다.

```
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<String> numbers = Arrays.asList("1", "2", "3", "4", "5");

        int sum = numbers.stream()
                         .mapToInt(Integer::parseInt)
                         .sum();

        System.out.println(sum);
    }
}
```

지연 연산 메소드들은 스트림의 데이터를 변환, 필터링, 정렬 등의 연산을 지연시켜 처리할 수 있으며, 이를 이용하여 메모리를 효율적으로 사용할 수 있습니다.

## 6-3. 최종 연산 메소드

스트림에서 최종 연산은 스트림을 처리하고 결과를 도출하는 연산입니다. 최종 연산은 한 번만 수행할 수 있으며, 스트림을 닫는 효과가 있습니다.

스트림에서 제공하는 최종 연산 관련 메소드는 다음과 같습니다.

- forEach(): 요소를 하나씩 처리합니다.
- count(): 요소의 개수를 반환합니다.
- min(), max(): 최소, 최대 값을 반환합니다.
- findFirst(), findAny(): 첫 번째 요소, 임의의 요소를 반환합니다.
- anyMatch(), allMatch(), noneMatch(): 요소들 중 조건에 맞는 요소를 찾습니다.
- reduce(): 요소들을 하나의 값으로 축소합니다.
- collect(): 요소들을 수집하여 새로운 컬렉션을 생성합니다.

각각의 메소드들은 다양한 기능을 수행하며, 이를 이용하여 복잡한 데이터 처리를 쉽게 수행할 수 있습니다.

다음은 reduce() 메소드를 이용하여 리스트의 모든 요소의 합을 구하는 예제 코드입니다.

```
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

        int sum = numbers.stream()
                         .reduce(0, (acc, n) -> acc + n);

        System.out.println(sum);
    }
}
```

최종 연산 관련 메소드들은 스트림의 처리 결과를 도출하는 데 유용한 다양한 기능을 제공하므로, 적절히 사용하여 데이터 처리를 쉽게 수행할 수 있습니다.

## 6-4. 병렬처리

자바 스트림에서 병렬 처리는 스트림의 요소를 병렬로 처리하여 처리 시간을 단축시키는 기법입니다. 병렬 처리는 Stream API에서 제공하는 parallel() 메소드를 이용하여 스트림을 병렬 스트림으로 변환하여 수행됩니다. 병렬 스트림은 여러 개의 스레드를 이용하여 요소를 처리하므로, 대용량의 데이터를 처리하는 경우에는 처리 시간을 크게 단축할 수 있습니다.

스트림에서 병렬 처리를 수행할 때는 몇 가지 주의사항이 있습니다. 병렬 스트림은 스레드를 사용하기 때문에 멀티코어 환경에서 최적의 성능을 발휘합니다. 하지만 작은 데이터에 대해서는 오히려 성능이 더 떨어질 수 있습니다. 따라서 병렬 처리를 사용할 때는 데이터 크기에 대한 적절한 판단이 필요합니다.

또한, 스트림의 연산 중 일부는 병렬 처리에 적합하지 않은 연산이 있습니다. 예를 들어, forEach() 메소드는 스트림의 모든 요소를 소비하는 연산으로, 병렬 처리를 수행할 때의 성능 개선 효과가 없습니다. 따라서 병렬 처리를 수행할 때는 이러한 연산을 피해야 합니다.

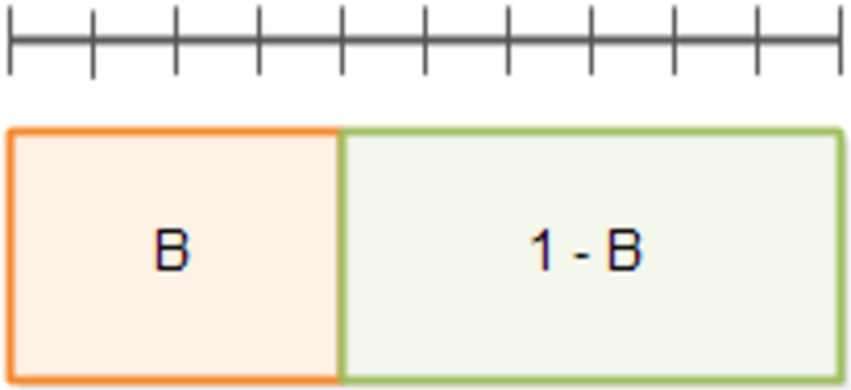
다음은 병렬 처리를 수행하는 스트림 예제 코드입니다.

```
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

        int sum = numbers.parallelStream()
                         .mapToInt(Integer::intValue)
                         .sum();

        System.out.println(sum);
    }
}
```

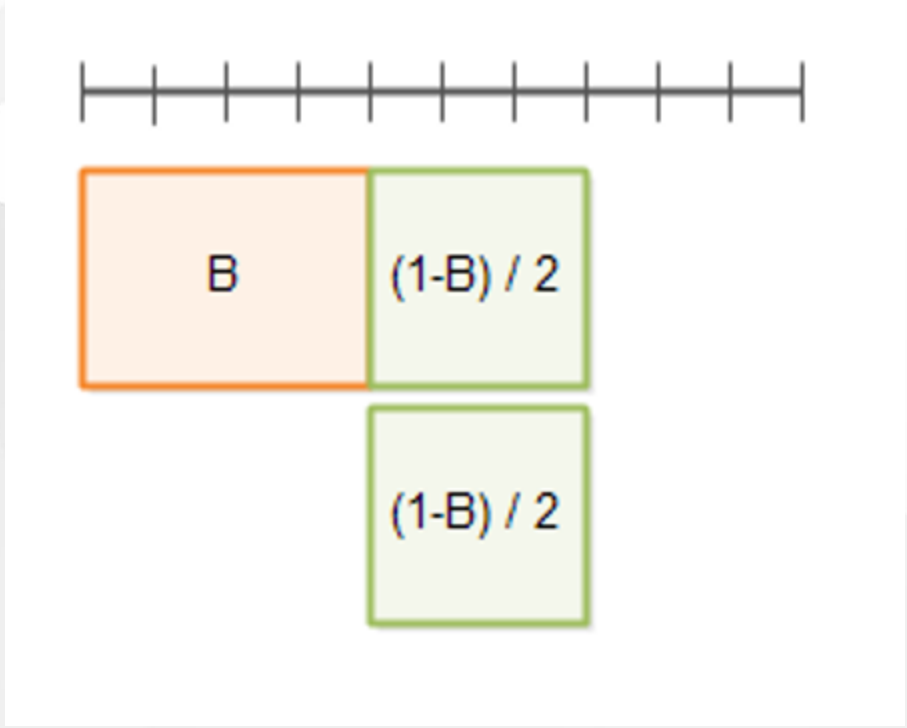
위 예제에서는 스트림을 병렬로 처리하여 처리 시간을 단축시키는 것을 보여주고 있습니다. 하지만 병렬 처리를 수행할 때는 데이터 크기와 처리 시간 등을 고려하여 적절한 판단이 필요합니다.

# 멀티 쓰레딩

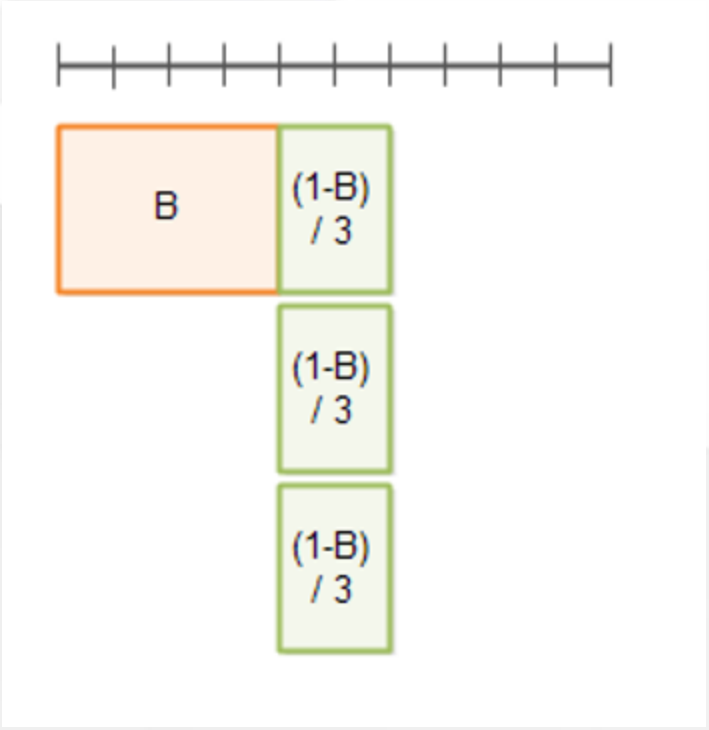
# 암달의 법칙(Amdahl's Law)



# 암달의 법칙(Amdahl's Law)



# 암달의 법칙(Amdahl's Law)



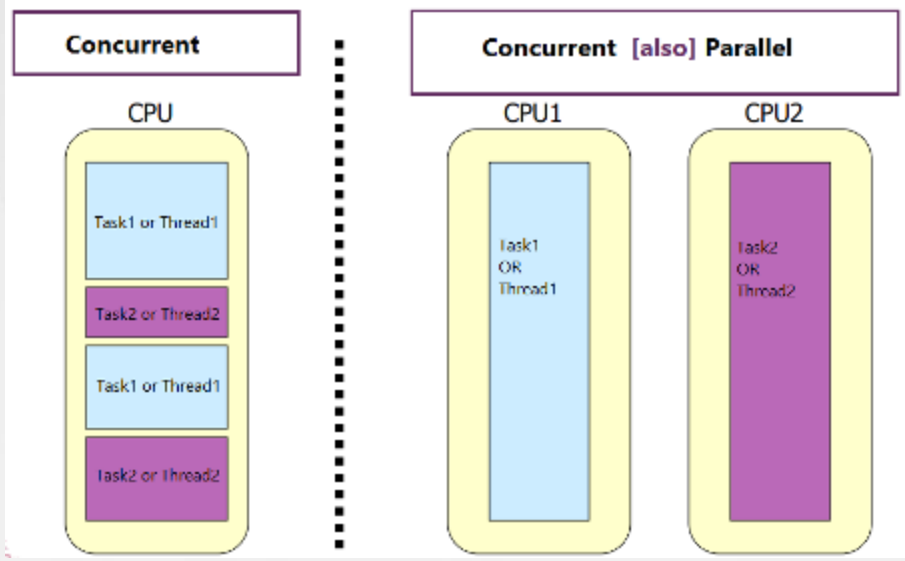
# 암달의 법칙(Amdahl's Law)

**병렬화 할 때 고려해야 할 것들.**

- 메모리의 속도
- CPU 캐시 메모리
- 디스크
- 네트워크
- 컨넥션
- 순차적 실행이 병렬실행보다 빠른 경우도 있다. 동시 실행에 따르는 오버헤드가 없고, 단일 CPU알고리즘은 하드웨어 작업에 더 친화적일 수 있기 때문이다.
- .....

# 병렬 vs 병행

- 병행(Concurrent)은 멀티스레드 프로그래밍을 의미한다.
- 병렬(Parallel)은 멀티코어 프로그래밍을 의미한다.
- 우리가 살펴볼 것은 병행 프로그래밍(동시성 프로그래밍, 멀티스레드 프로그래밍)



## Process

- 각각의 프로세스는 메모리 공간에서 독립적으로 존재한다.
- 각각의 프로세스는 다음 페이지 그림과 같이 자신만의 메모리 구조를 가진다.
- 프로세스 A,B,C가 있을 경우 각각 프로세스는 모두 같은 구조의 메모리 공간을 가진다.
- 독립적인 만큼 다른 프로세스의 메모리 공간에 접근할 수 있다.

# Process

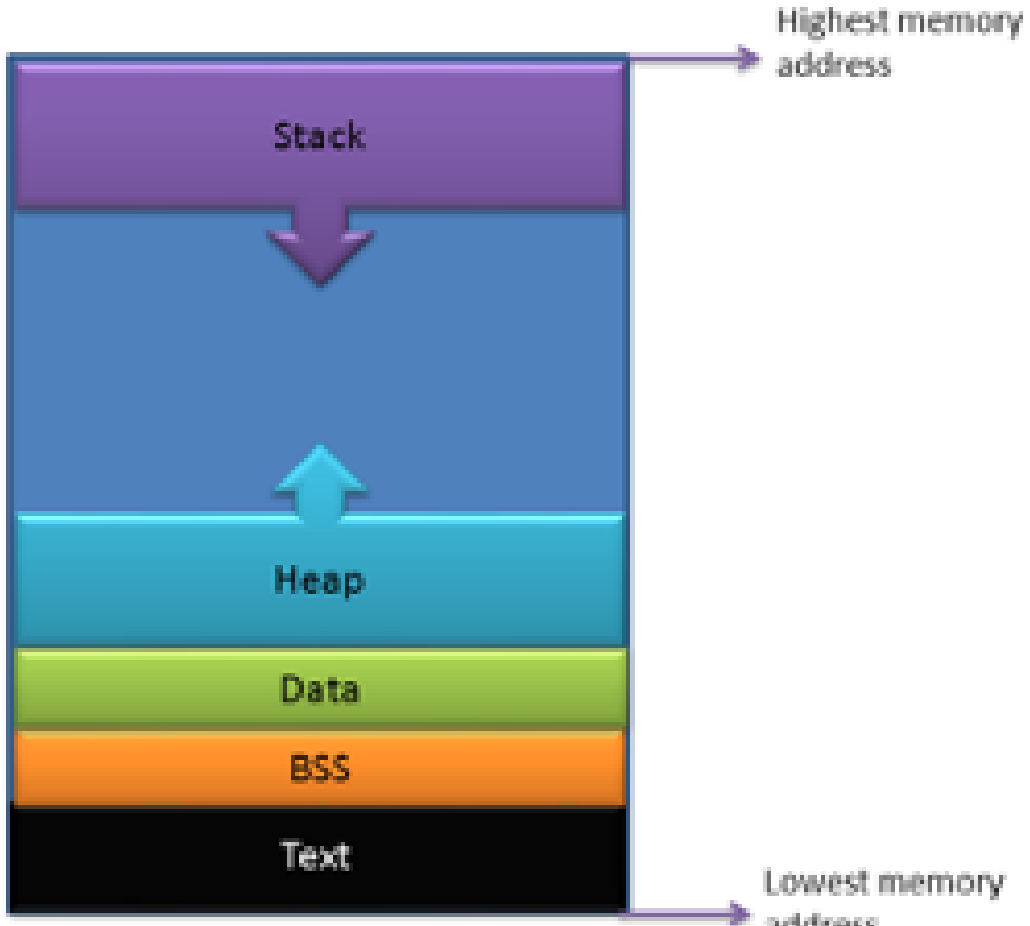
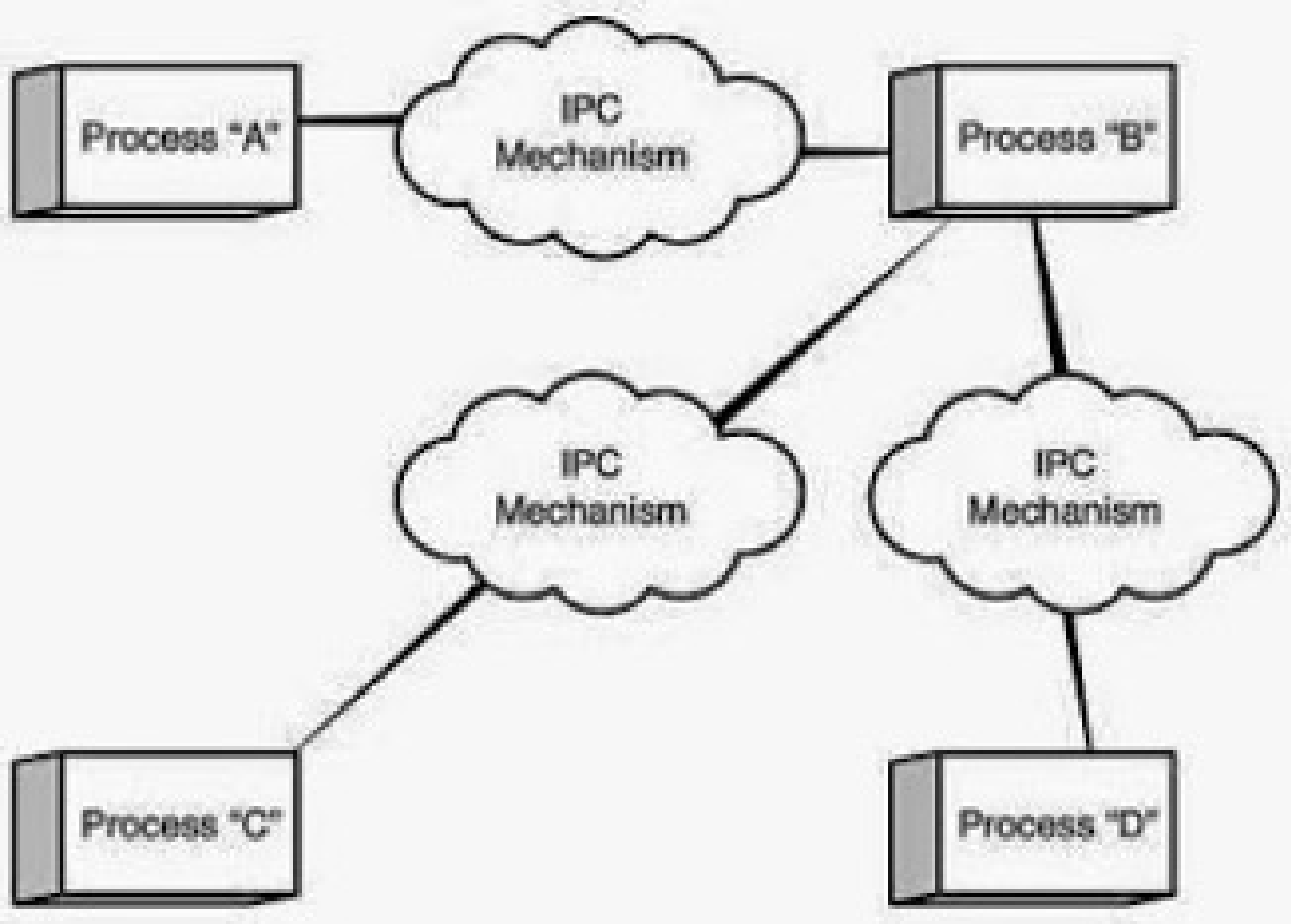



**Figure : Process memory organization**

## IPC

- 프로세스 A에서 프로세스 B를 직접 접근할 수 없기 때문에 프로세스 간의 통신을 하는 특별한 방식이 필요하다. 메일슬록(mailslot), 파이프(pipe)등이 바로 프로세스 간의 통신 즉, IPC의 예들이다.
- 프로세스는 독립적인 메모리 공간을 지니기 때문에 IP를 통하지 않고 통신할 수 없다.
- 프로세스가 여럿이 병렬적으로 실행되기 위해서는 필연적으로 컨텍스트 스위칭이 발생할 수 밖에 없다. 이것을 해결할 수 있는 것이 Thread이다.

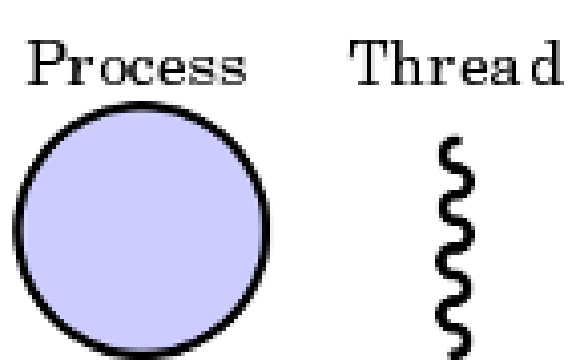
# IPC

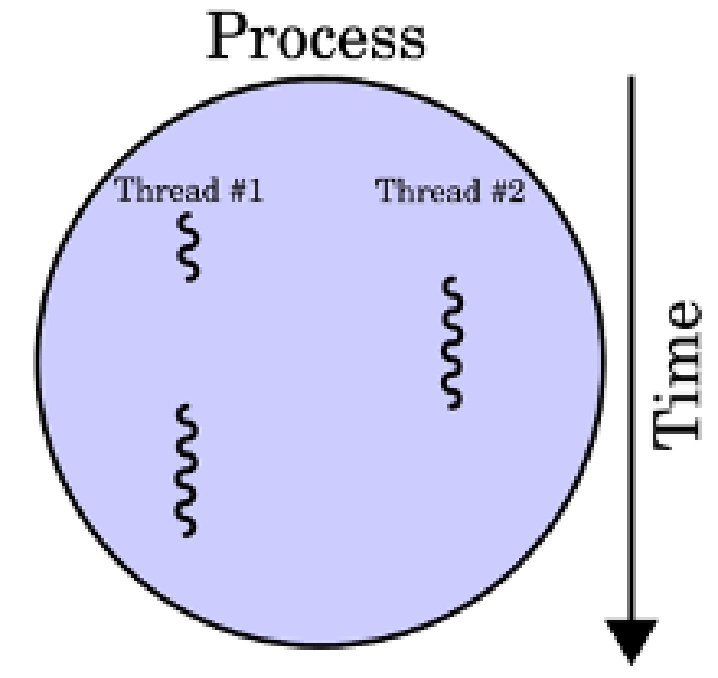

Process "A"
IPC Mechanism
Process "B"
IPC Mechanism
IPC Mechanism
Process "C"
Process "D"

## Thread

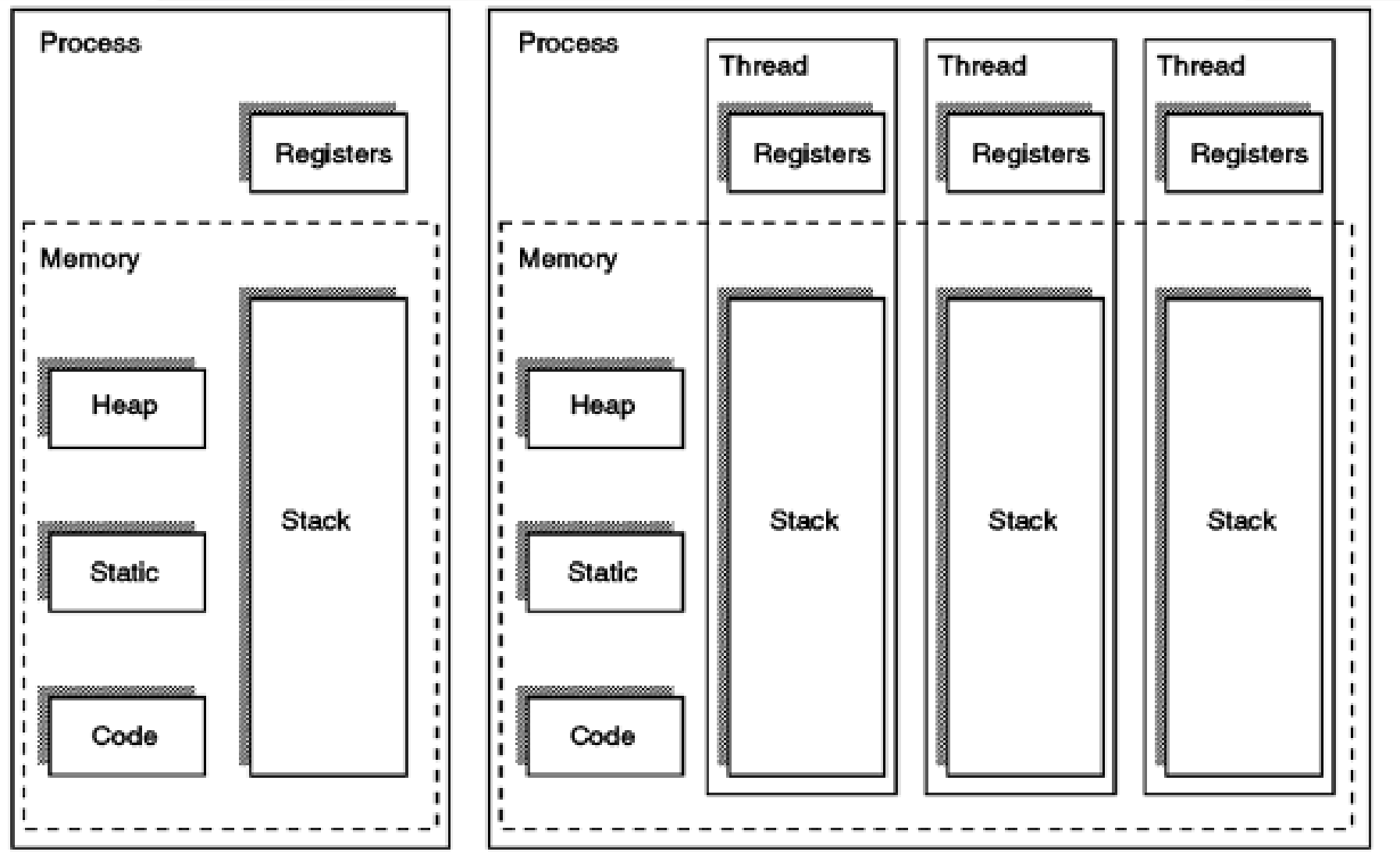
- 스레드는 하나의 프로그램 내에 존재하는 여러 개의 실행 흐름을 위한 모델이다.
- 우리가 생각하는 프로그램이 실행되기 위해서 하나의 실행흐름으로 처리할 수도 있지만 다수의 실행흐름으로 처리할 수도 있다.
- 뒤에 나오는 그림과 같이 스레드는 프로세스와 별개가 아닌 프로세스를 구성하고 실행하는 흐림이다.

# Thread



# 메모리 공간에서의 스레드



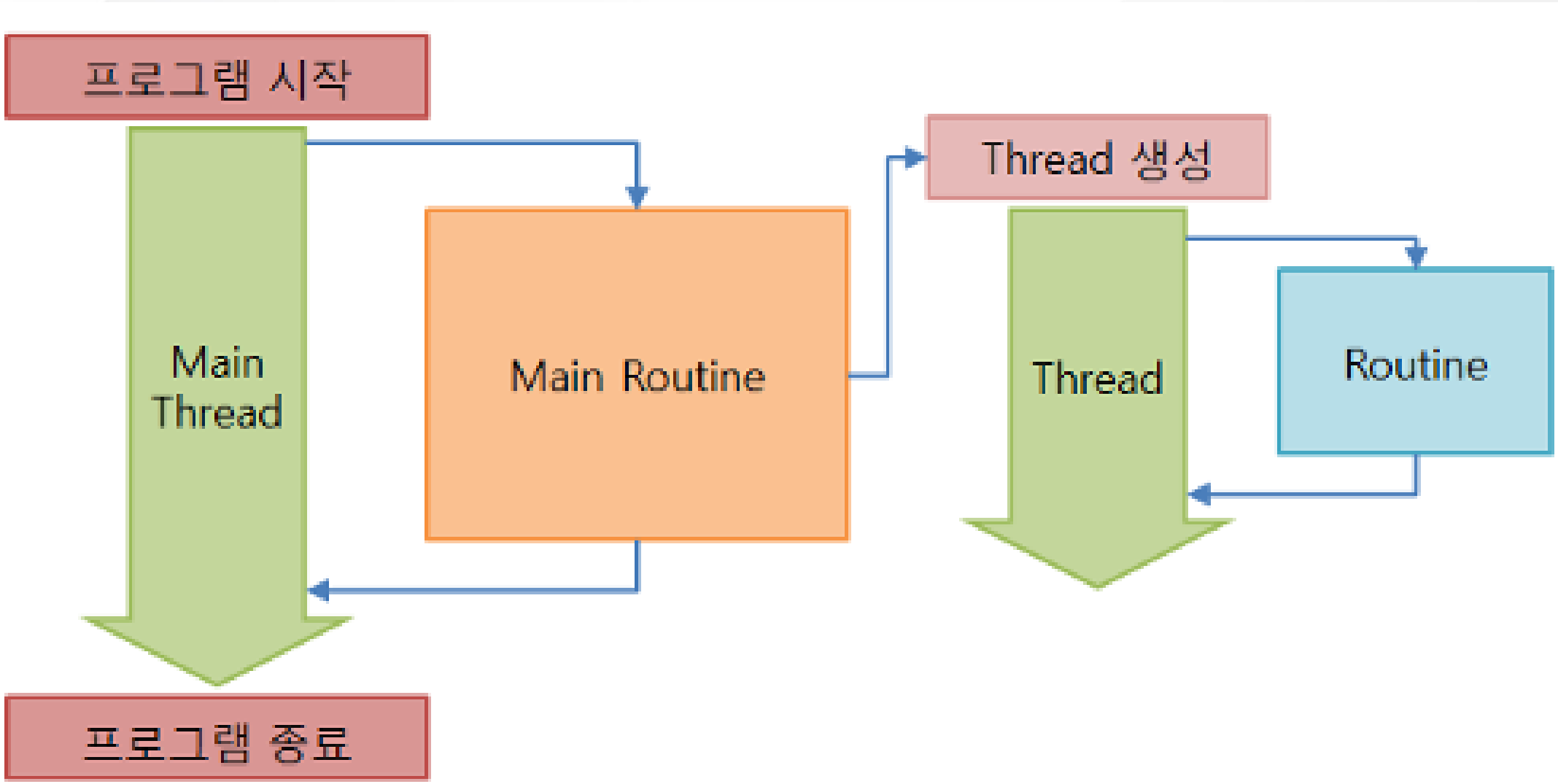
# 스레드 vs 프로세스

- 스레드는 프로세스 안에 존재하는 실행흐름이다.
- 스레드는 프로세스의 heap, static, code영역등을 공유한다.
- 스레드는 staack영역을 제외한 메모리 영역은 공유한다.
- 스레드가 code영역을 공유하기 때문에 프로세스 내부의 스레드들은 프로세스가 가지고 있는 함수를 자연스럽게 모두 호출할 수 있다.
- 스레드는 IPC없이도 스레드 간의 통신이 가능하다. A,B스레드는 통신하기 위해 heap영역에 메모리 공간을 할당하고, 두 스레드가 자유롭게 접근할 수 있다.

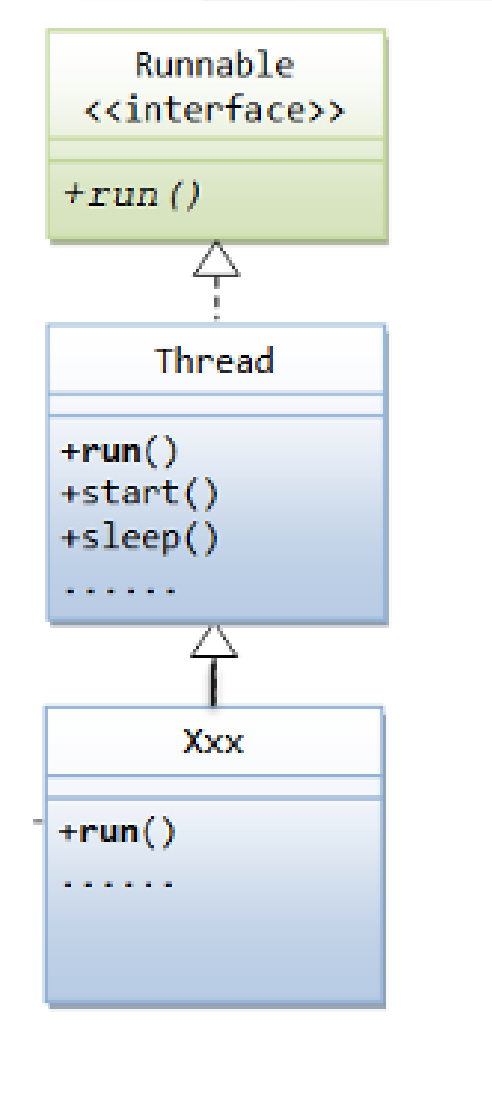
# 스레드 vs 프로세스

- 스레드는 프로세스처럼 스케쥴링의 대상이다. 이 과정에서 컨텍스트 스위칭이 발생한다. 하지만 스레드는 공유하고 있는 메모리 영역 덕분에 컨텍스트 스위칭 때문에 발생하는 오버헤드가 프로세스에 비해 작다.
    - 동작중인 프로세스가 바뀔 때 프로세스는 현재 자신의 상태(contex정보)를 일단 보존한 후, 새롭게 동작 개시하는 프로세스는 이전에 보존해 두었던 자신의 컨텍스트 정보를 다시 복구한다. 이와 같은 현상을 컨텍스트 스위칭이라 말한다.
    - 스레드의 컨텍스트 정보는 프로세스보다 적기 때문에 스레드의 컨텍스트 스위칭은 가볍게 행해지는 것이 보통이다.
    - 하지만, 실제로 스레드와 프로세스의 관계는 JVM구현에 크게 의존한다.
- 참고로 플랫폼이 같아도 JVM의 구현방법에 따라 프로세스와 스레드의 관계는 달라질 수 있다.

# 멀티쓰레드(Multithread) 실행방식



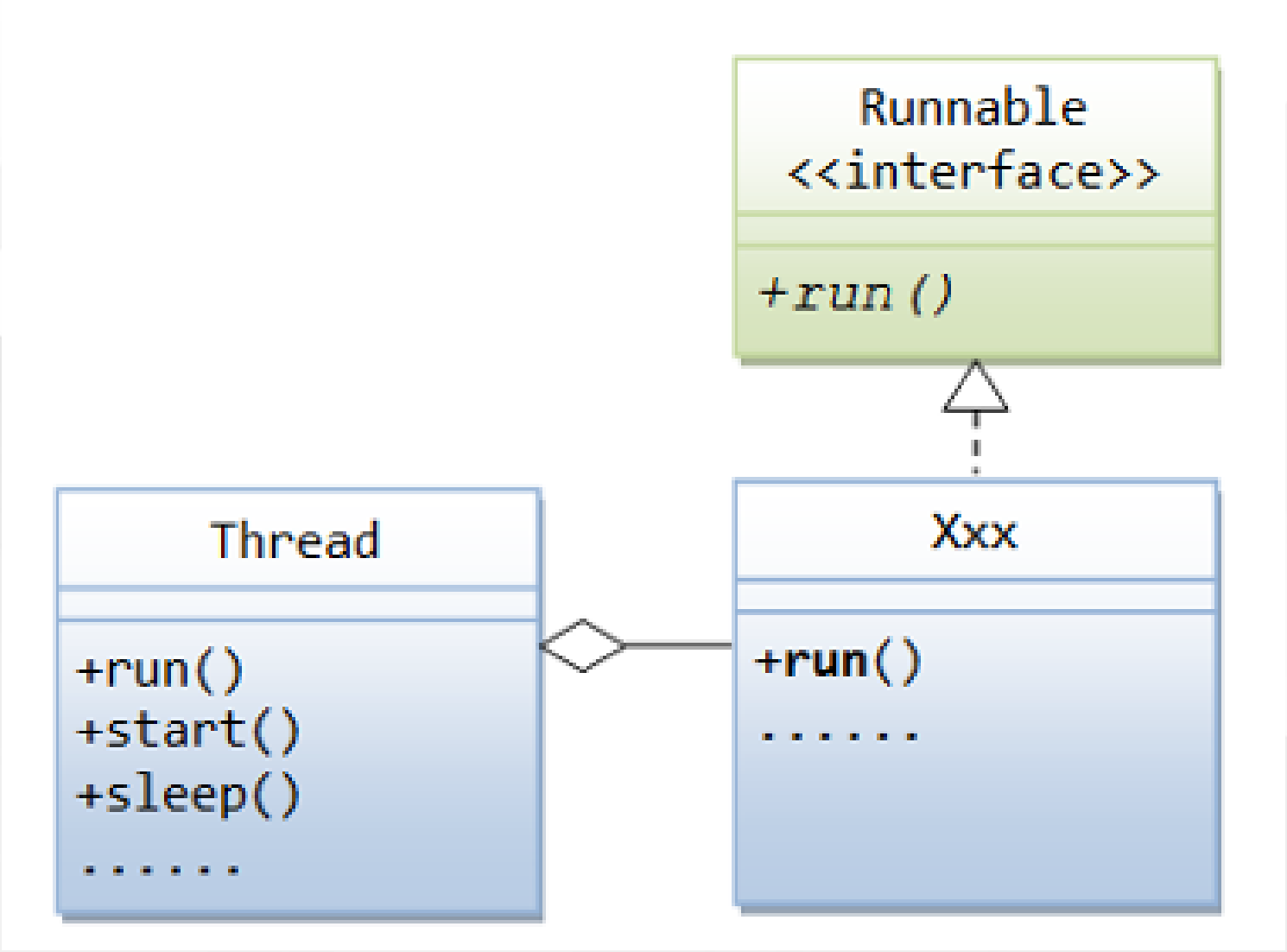
# Thread 클래스를 상속받아 쓰레드 작성하기



```
class Xxx extends Thread{
    public void run(){
        // 동시에 실행될 코드 작성
    }
}
```

```
Xxx x = new Xxx();
x.start();
```

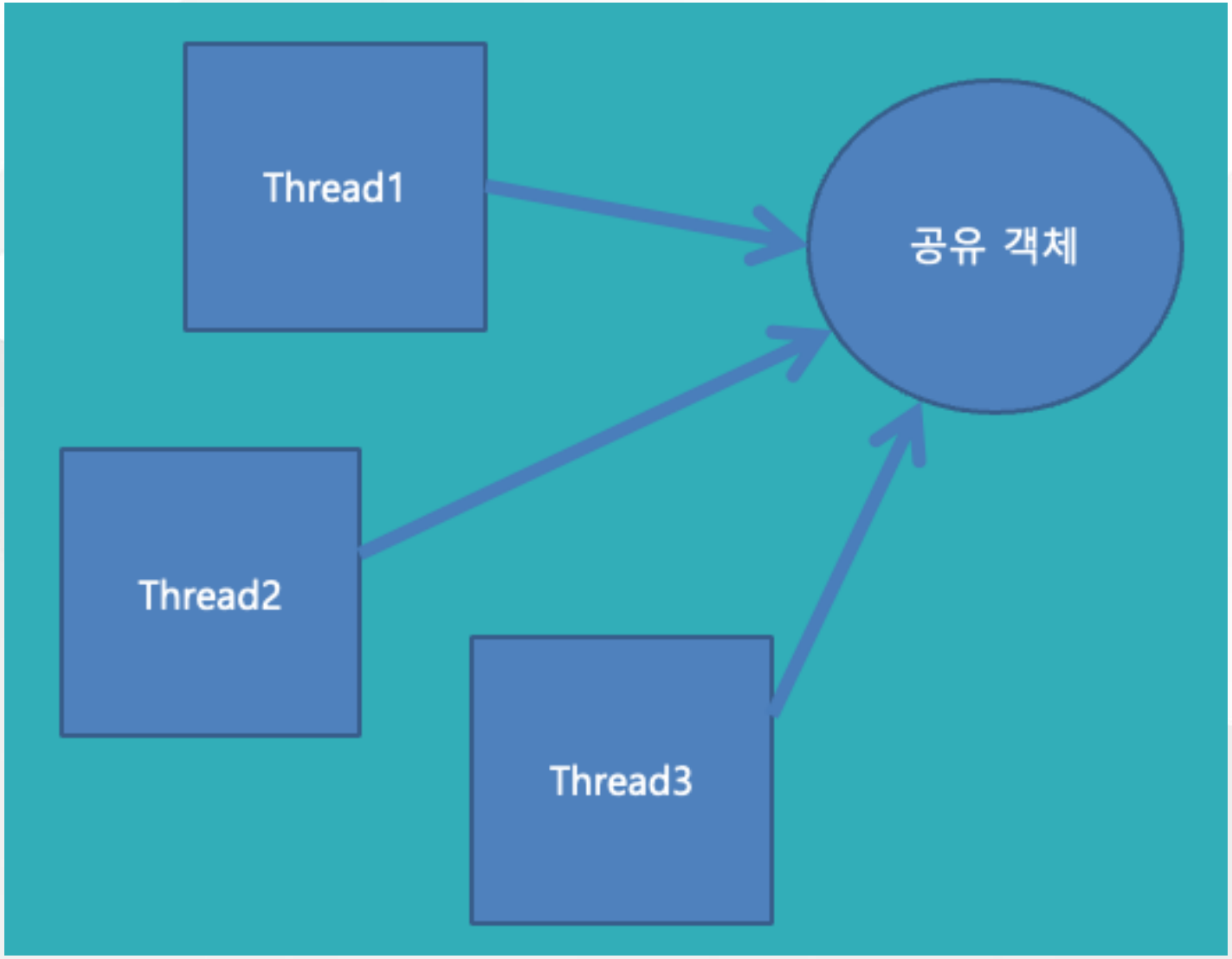
# Runnable 인터페이스를 구현하여 쓰레드 작성하기



```
class Xxx implements Runnable{
    public void run(){
        // 동시에 실행될 코드 작성
    }
}
```

```
Xxx x = new Xxx();
Thread t = new Thread(x);
t.start();
```

# 공유 객체



```
class SharedObject{

}
// Thread를 상속받고 run메소드를 구현한다고 가정하자.
class Thread1{
    private SharedObject so;
    public Thread1(SharedObject so){
        this.so = so;
    }

}

class Thread2{
    private SharedObject so;
    public Thread1(SharedObject so){
        this.so = so;
    }
}

class Thread3{
    private SharedObject so;
    public Thread1(SharedObject so){
        this.so = so;
    }
}
```

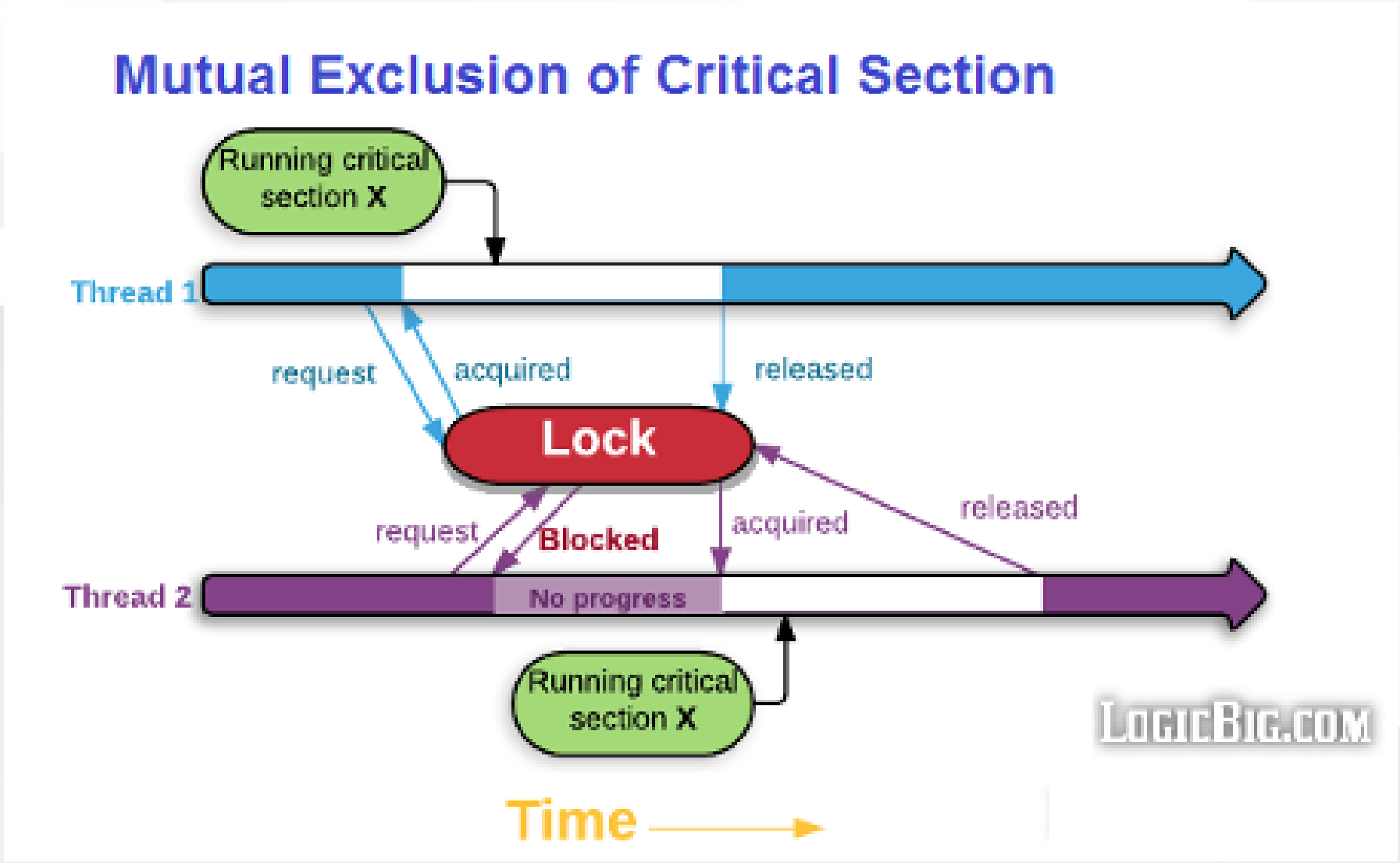
```
SharedObject so = new SharedObject();
Thread1 t1 = new Thread1(so);
Thread1 t2 = new Thread1(so);
Thread1 t3 = new Thread1(so);
```

하나의 인스턴스를 t1, t2, t3가 가지도록 하였다. (같은 객체를 참조한다.)

## Java의 고유 락(intrinsic lock)

- 자바의 모든 객체는 락(lock)을 가지고 있다.
- 고유 락(intrinsic lock) 또는 모니터 락(monitor lock) 또는 모니터(monitor)라고 한다.

# Java의 고유 락(intrinsic lock)



# Synchronized

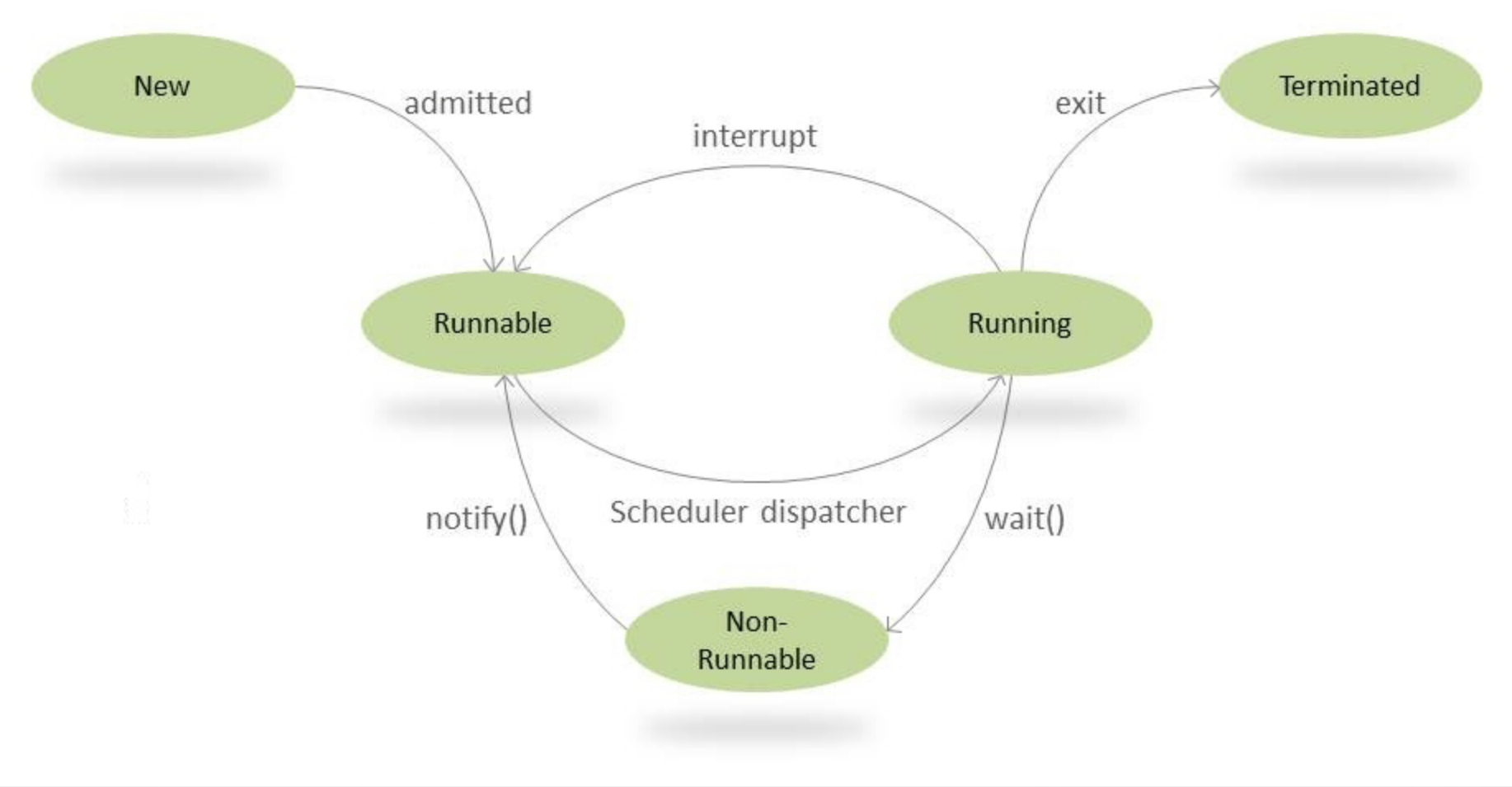
- Java에서는 키워드 synchronized 를 이용해 공유 객체에 동시에 실행되면 안되는 메소드나 블록을 보호할 수 있다.
- 공유 객체에서 동시에 실행되면 안되는 부분과 동시에 실행 되도 되는 부분을 잘 구분한 후, 동시에 실행되면 안되는 부분에 synchronized 키워드를 이용해 제어할 수 있다.
- synchronize는 메소드에 붙여 사용하거나 메소드 안의 특정 부분을 블록으로 감싸 사용할 수 있다.

```
synchronize( 객체의 필드 ) {
    // 동시에 실행되면 안되는 코드
}

synchronized 리턴타입 메소드명(파라미터){
      // 동시에 실행되면 안되는 코드
}
```

## 가정

- 마술 상자(Magic Box)가 있다.
- 돌릴 수 있는 핸들(handle)이 3개 달려 있다.
- 2개의 핸들은 함께 돌릴 경우 고장이 날 수 있다.
- 1개의 핸들은 함께 돌려도 아무런 문제가 없다.
- 3명의 사람이 핸들을 하나씩 잡고 10번씩 돌려보자.

# 쓰레드 라이프사이클(Life cycle)



# wait() & notifyAll()

```
public class Data {
    private String packet;

    // True if receiver should wait
    // False if sender should wait
    private boolean transfer = true;

    public synchronized String receive() {
        while (transfer) {
            try {
                wait();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                System.out.println("Thread Interrupted");
            }
        }
        transfer = true;

        String returnPacket = packet;
        notifyAll();
        return returnPacket;
    }

    public synchronized void send(String packet) {
        while (!transfer) {
            try {
                wait();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                System.out.println("Thread Interrupted");
            }
        }
        transfer = false;

        this.packet = packet;
        notifyAll();
    }
}
```

```
public class Sender implements Runnable {
    private Data data;

    // standard constructors

    public void run() {
        String packets[] = {
          "First packet",
          "Second packet",
          "Third packet",
          "Fourth packet",
          "End"
        };

        for (String packet : packets) {
            data.send(packet);

            // Thread.sleep() to mimic heavy server-side processing
            try {
                Thread.sleep(ThreadLocalRandom.current().nextInt(1000, 5000));
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                Log.error("Thread interrupted", e);
            }
        }
    }
}
```

```
public class Receiver implements Runnable {
    private Data load;

    // standard constructors

    public void run() {
        for(String receivedMessage = load.receive();
          !"End".equals(receivedMessage);
          receivedMessage = load.receive()) {

            System.out.println(receivedMessage);

            // ...
            try {
                Thread.sleep(ThreadLocalRandom.current().nextInt(1000, 5000));
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                Log.error("Thread interrupted", e);
            }
        }
    }
}
```

```
public static void main(String[] args) {
    Data data = new Data();
    Thread sender = new Thread(new Sender(data));
    Thread receiver = new Thread(new Receiver(data));

    sender.start();
    receiver.start();
}
```

# 감사합니다.

강경미(carami@nate.com)