9주차 강의 - (05.02)

추상 클래스

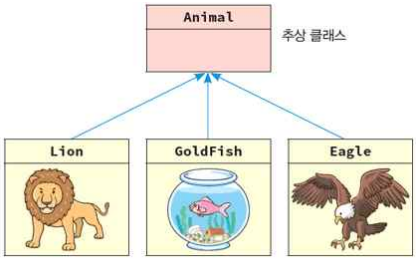


그림 7.1 추상 클래스의 개념

- 정의 : 클래스의 구현되어 있지않은 메소드를 가진 클래스
- 추상 클래스로는 **객체를 생성할 수 없다**.
- 상속 계층에서 추상적인 개념을 나타내기 위한 용도로 사용한다.

=> 필수 클래스를 오버라이딩 해라!

추상 클래스 만들기

```
public abstract class Animal  {
   public abtract void move();
   ...
};
```

추상 메소드 정의, ;으로 종료됨을 유의!

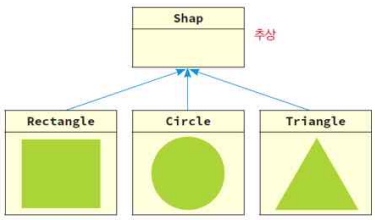
```
public class Lion extends Animal  {
   public void move() {
      System.out.println("사자의 move() 메소드입니다.");
   }
};
```

추상 클래스를 상속받으면 추상 메소드를 구현하여야 한다.

- 만들기 : 클래스 이름 앞에 **abstract**만 붙이면 된다.
- 객체는 못 만들어도 **상속은 할 수 있다.**

* 이거보단 인터페이스 많이 써요 => 추클은 다중상속이 안되기 때문

추상 클래스 만들기 : 도형




```
abstract class Shape {
   int x, y;
   public void translate(int x, int y) {
      this.x = x;
      this.y = y;
   }
   public abstract void draw();
};
```

추상 클래스라고 하더라도 추상 메소드가 아닌 보통의 메소드도 가질 수 있음을 유의하라.

추상 메소드를 선언한다. 추상 메소드를 하나라도 가지면 추상 클래스가 된다. 추상 메소드를 가지고 있는데도 abstract를 class 앞에 붙이지 않으면 컴파일 오류가 발생한다.

- 업캐스팅 : 부모의 것을 자식이 가져온다.

추상 클래스를 쓰는 이유

```
abstract class Shape {
   public abstract void draw();
};
```

```
class Shape {
   public void draw() { }
};
```

```
class Circle extends Shape {
   @Override
   public void draw() {
      System.out.println("Circle draw()");
   }
};
```

추상 메소드로 정의되면 자식 클래스에서 반드시 오버라이드하여야 한다. 하지 않으면 오류가 발생한다.

```
class Circle extends Shape {
   @Override
   public void draw() {
      System.out.println("Circle draw()");
   }
};
```

일반 메소드로 정의되면 자식 클래스에서 오버라이드하지 않아도 컴파일러가 체크할 방법이 없다.

- **왜 쓰나?**

  > 일반 메소드는 자식 클래스에서 오버라이드 하지 않을 때, 컴파일러가 체크할 방법이 없다.

  ex) 함수이름 오타내서 drew()로 해버리면 왼쪽=오류 오른쪽=동작

- **단점**

  > 진짜 쩌는 슈퍼 클래스가 있는데 이런걸 추상 클래스에 써야함

* @Override 어노테이션을 안 붙인 상황이다.

인터페이스

- **정의** : 클래스 간의 상호작용을 기술하는 일종의 규격
- 진짜 쩌는 카드를 소지하지 않아도 추상 메소드를 정의할 수 있다.
- 얘는 딴 건 다 못가지고, 딱 **추상 메소드만 가질 수 있다**.

- **예시**

  > 사람(Human)과 자동차(Car)는 둘 다 달릴 수 있다
  > 그렇다고 부모 클래스로 Runner를 작성 후
  > Human과 Car를 Runner 클래스의 자식 클래스로 하면 선넘은 거

  => Runnable 인터페이스를 생성 후 양쪽 클래스가 구현

인터페이스 정의

```
public interface 인터페이스_이름 {
   반환형    추상메소드1(...);
   반환형    추상메소드2(...);
   ...
}
```

- 변수는 선언될 수 없지만 **상수(열거형 같은 거)는 선언 가능**

인터페이스 구현

```
class Television implements RemoteControl {
   boolean on;
   public void turnOn() {
      on = true;
      System.out.println("TV가 켜졌습니다.");
   }
   public void turnOff() {
      on = false;
      System.out.println("TV가 꺼졌습니다.");
   }
}
```

> **implements + 상속할 인터페이스 이름**
> 하나라도 구현 안 하면 바로 에러남

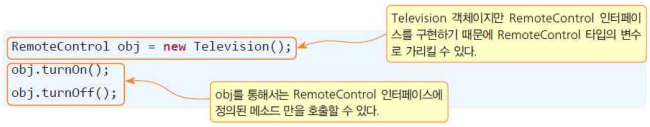
추상 클래스 VS 인터페이스

- **추상클래스**
  - 〉 클래스들 사이에서 코드를 공유하고 싶다면
  - 〉 **공통적인 필드나 메소드의 수가 많은 경우**
  - 〉 public 이외의 접근 지정자를 사용해야 하는 경우
  - 〉 정적이 아닌 필드나 상수가 아닌 필드를 선언하기를 원할 때

- **인터페이스**
  - 〉 관련 없는 클래스들이 동일한 동작을 구현하기를 원할 때
  - 〉 특정 자료형 동작을 지정하고 싶지만 누가 구현해도 상관X 일 때
  - 〉 **다중 상속이 필요할 때**

* 교수님 취향 : 인터페이스

인터페이스와 타입



- 왜 쓰지...?

* 잘 안 쓴다고 하신다.

인터페이스의 default

```
interface RemoteControl {
   void turnOn();
   void turnOff();
   public default void printBrand() { System.out.println("Remote Control 가능 TV"); }
}
```

- default를 이용해 기본 구현을 정의할 수 있는 것 같다.
- 일반적으로 오버라이딩 없이 그냥 써라라는 의미
- 그래도 오버라이드 해서 구현해도 되긴 한다.

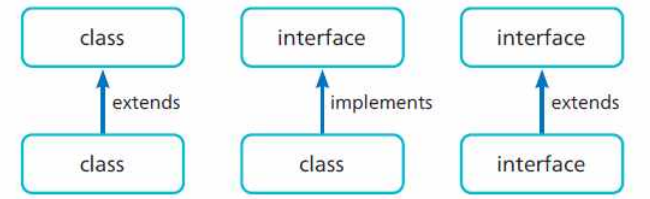
인터페이스의 상속

```
public interface RemoteControl {
   public void turnOn();  // 가전 제품을 켠다.
   public void turnOff(); // 가전 제품을 끈다.
}
```

```
public interface AdvancedRemoteControl extends RemoteControl {
   public void volumeUp();   // 가전 제품의 볼륨을 높인다.
   public void volumeDown(); // 가전 제품의 볼륨을 낮춘다.
}
```

인터페이스도 다른 인터페이스를 상속 받을 수 있다.

- 상속도 가능!



다중상속

TestInterface2.java

```
01 class Shape {
02    protected int x, y;
03 }
04 interface Drawable{
05   void draw();
06 }
07 class Circle extends Shape implements Drawable {
08    int radius;
09    public void draw() {  System.out.println("Circle Draw at ("+x+", "+y+")");  }
10 }
11 public class TestInterface2 {
12    public static void main(String args[]){
13       Drawable obj = new Circle();
14       obj.draw();
15    }
16 }
```

상속과 동시에 인터페이스를 구현하고 있다. 이것이 가장 일반적인 클래스 정의 형태이다.

```
Circle Draw at (0, 0)
```

정적 메소드

StaticMethodTest2.java

```
01 interface Employable {
02    String getName();
03
04    static boolean isEmpty(String str) {
05       if (str == null || str.trim().length() == 0) {
06          return true;
07       } else {
08          return false;
09       }
10    }
11 }
```

이름을 검사하는 유틸리티 메소드를 만들어서 인터페이스 안에 정적 메소드로 추가할 수 있다.

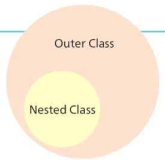
팩토리 메소드

- 팩토리 메소드는 공장처럼 객체를 생성하는 정적 메소드이다.
- 디자인 패턴의 하나
- **객체를 만드는 부분을 부모 클래스에 위임하는** 패턴
- new를 호출해 객체를 생성하는 코드를 부모 클래스에 위임

중첩 클래스

Syntax: 중첩 클래스 정의

```
class OuterClass {
   ...
   class NestedClass {
      ...
   }
}
```



- **외부 클래스(outer class)** : 내부에 클래스를 가지고 있는 클래스
- **중첩 클래스(nested class)** : 클래스 내부에 포함되는 클래스

중첩 클래스의 분류

- **정적 중첩 클래스**: 앞에 static이 붙어서 내장되는 클래스
- **비정적 중첩 클래스**: static이 붙지 않은 일반적인 중첩 클래스
  - 〉 **내부 클래스**(inner class): 클래스 멤버처럼 선언되는 중첩 클래스
  - 〉 **지역 클래스**(local class): 메소드의 몸체 안에 선언된 중첩 클래스
  - 〉 **익명 클래스**(anonymouse class): 수식의 중간에서 선언되고 바로 객체화되는 클래스

내부 클래스

InnerClassTest.java

```java
class OuterClass {
    private int value = 10;

    class InnerClass {
        public void myMethod() {
            System.out.println("외부 클래스의 private 변수 값: " + value);
        }
    }

    OuterClass() {
        InnerClass obj = new InnerClass();
        obj.myMethod();
    }
}

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

이것이 바로 내부 클래스이다. 내부 클래스 안에서는 외부 클래스의 private 변수들을 참조할 수 있다.

내부 클래스를 사용한다.

```
외부 클래스의 private 변수 값: 10
```

지역 클래스

```java
class localInner {
    private int data = 30;   // 인스턴스 변수

    void display() {
        final String msg = "현재의 데이터값은 ";

        class Local {
            void printMsg() {
                System.out.println(msg + data);
            }
        }
        Local obj = new Local();
        obj.printMsg();
    }
}

public class localInnerTest {
    public static void main(String args[]) {
        localInner obj = new localInner();
        obj.display();
    }
}
```

메소드 display() 안에 클래스 Local이 정의되어 있다. 지역 클래스는 메소드 안에서만 사용이 가능하다. 외부 클래스의 private 변수에 접근할 수 있다.

```
현재의 데이터값은 30
```

익명 클래스

AnonymousClassTest.java

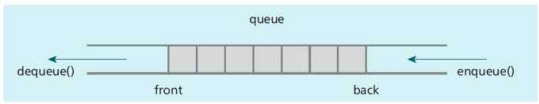
```java
public interface RemoteControl {
    void turnOn();
    void turnOff();
}

public class AnonymousClassTest {
    public static void main(String args[]) {
        RemoteControl ac = new RemoteControl() {        // 익명 클래스 정의
            public void turnOn() {
                System.out.println("TV turnOn()");
            }
            public void turnOff() {
                System.out.println("TV turnOff()");
            }
        };
        ac.turnOn();
        ac.turnOff();
    }
}
```

익명 클래스가 정의되면서 동시에 객체도 생성된다.

```
TV turnOn()
TV turnOff()
```

해보세요 : 큐 구현하기

- 큐(queue)는 요소가 선입선출 방식으로 추가되고 제거되는 자료구조이다.



- 인터페이스 Queue를 설계해보자.
  - q.dequeue() : Queue에서 하나의 항목을 삭제하고 반환한다.
  - q.enqueue(item) : Queue에서 하나의 항목을 추가한다.
  - q.isEmpty() : Queue가 비어 있는지를 검사한다.
- 인터페이스 Queue를 구현하는 클래스 MyQueue 클래스를 작성해보자.

05월 16일 - 월

패키지

- **정의** : 관련있는 클래스들을 하나로 묶은 것
- **사용하는 이유**

〉 서로 관련된 클래스들을 하나의 단위로 모을 수 있다.
〉 중요한 이유 중 하나는 “이름공간(name space)” 때문이다.
〉 세밀한 접근 제어를 구현할 수 있다.

*패키지 내부 클래스는 패키지 안에서만 쓸 수 있도록 선언 가능

- **패키지 선언**

Circle.java

```java
package graphics;

public class Circle {
    double radius;
}
```

- **패키지 포함**

  〉 특정 패키지 : `import graphics.Rectangle;`

  〉 전체 패키지 : `import graphics.*;`

  〉 포함 범위 :
```
import java.awt.*;          // java.awt 패키지의 클래스 포함
import java.awt.font.*;     // java.awt.font 패키지의 클래스 포함
```
  ** 저기에 '*' 붙였다고 하위 '폴더'는 포함되지 않는다.
  ** '*'에 포함되는 전체란 '패키지 파일' 뿐이다.

클래스 파일 로드 시기

1) 자바 소스 파일(확장자 **.java)이 컴파일**
2) .class 확장자의 **클래스 파일로 변환**되고 **파일 시스템에 저장**
3) 클래스 파일은 자바 가상 기계**(JVM)에 의하여 로드**

- 클래스 파일은 요청 시 JVM에 로드된다.
- 이것이 기본적인 방법, **지연 클래스 로드(Lazy Class Loading)**라 함
- 애플리케이션 코드를 구성하는 기본 클래스 파일은 시작 시 로드

- **장점** : 가상 머신의 메모리 절약 가능

- **예시**

```
if(result != null) {
   MyClass object = new MyClass();
   // 어떤 작업을 한다.
}
```

코드에서 result이 null이면 내부의 코드 블록은 절대 실행되지 않는다.
이러한 상황에서 가상 기계는 MyClass를 메모리에 로드하지 않는다.

JVM의 클래스를 찾는 순서

1) 부트스트랩 클래스
〉 **자바 플랫폼을 구성**하는 핵심적인 클래스
〉 디렉터리 jre/lib에 있는 rt.jar 및 기타 여러 jar 파일
〉 Java 9부터는 모듈화되어 있는 jmods 파일들이 로드

2) 확장 클래스
〉 **자바 확장 메커니즘**을 사용하는 클래스
〉 확장 디렉터리에 있는 jar 파일들
〉 디렉터리 jre/lib/ext의 jar 파일은 확장으로 간주되어 로드

3) 사용자 클래스
〉 확장 메커니즘을 활용하지 않는 **개발자 및 타사에서 정의한 클래스**
〉 가상 머신 명령줄에서 -classpath 옵션을 사용하거나,
CLASSPATH 환경 변수를 사용하여 이러한 클래스의 위치를 식별

JVM의 클래스를 찾는 방법

1) 현재 디렉터리 탐색

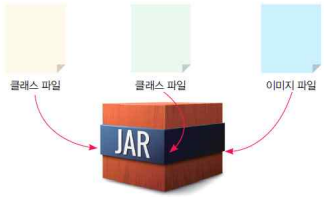
2) 일반적으로 환경 변수인 **CLASSPATH에 설정된 디렉터리**에서 탐색
〉 CLASSPATH 설정 : C:₩〉 set CLASSPATH=C:₩classes;C:₩lib;.

3) 가상 머신 **실행 옵션 -classpath** 이용
〉 클래스 경로를 JVM 실행 시 알림, 제일 권장
〉 C:₩〉 java -classpath C:₩classes;C:₩lib;. Main

CLASS PATH(클래스 경로)

사용자가 직접 작성하였거나 외부에서 다운로드 받은 클래스를 찾기 위하여 가상 머신이 살펴보는 디렉터리들을 모아둔 경로

JAR 압축 파일



- 자바 애플리케이션을 한 파일로 만들어 사용자에게 전달할 방법
- 자바 파일들을 압축하여 하나의 파일로 만드는 데 사용
- .JAR : "Java Archive"

- **생성법**

  - JDK 안에 포함된 jar 도구를 이용하여서 JAR 파일을 생성할 수 있다.

```
C:\> jar cvf Game.jar  *.class icon.png
```

  - 만약 실행 가능한 JAR 파일을 생성하려면 다음과 같이 e를 추가하여야 한다.

```
C:\> jar cvfe Game.jar Main *.class icon.png
```

  - JAR 파일로 압축된 파일을 실행하려면 어떻게 하면 될까?

```
C:\> java -jar Game.jar
```

JAVA API 패키지

패키지	설명
java.applet	애플릿을 생성하는 데 필요한 클래스
java.awt	그래픽과 이미지를 위한 클래스
java.io	입력과 출력 스트림을 위한 클래스
java.lang	자바 프로그래밍 언어에 필수적인 클래스
java.math	수학에 관련된 클래스
java.net	네트워킹 클래스
java.nio	새로운 네트워킹 클래스
java.security	보안 프레임워크를 위한 클래스와 인터페이스
java.sql	데이터베이스에 저장된 데이터를 접근하기 위한 클래스
java.util	날짜, 난수 생성기 등의 유틸리티 클래스
javax.imageio	자바 이미지 I/O API
javax.net	네트워킹 애플리케이션을 위한 클래스
javax.swing	스윙 컴포넌트를 위한 클래스
javax.xml	XML을 지원하는 패키지

Object 클래스

- Object 클래스는 java.lang 패키지에 포함
- 자바 클래스 계층 구조의 최상위 클래스

- **Object 메서드**

  - public boolean equals(Object obj) : obj가 이 객체와 같은지를 검사한다.
  - public String toString() : 객체의 문자열 표현을 반환한다.
  - protected Object clone() : 객체 자신의 복사본을 생성하여 반환한다.
  - public int hashCode() : 객체에 대한 해시 코드를 반환한다.
  - protected void finalize() : 가비지 콜렉터에 의하여 호출된다.
  - public final Class getClass() : 객체의 클래스 정보를 반환한다.

- getClass() 예제

CircleTest.java

```
01  class Circle {   }
02  public class CircleTest {
03     public static void main(String[] args) {
04        Circle obj = new Circle();
05        System.out.println("obj is of type " + obj.getClass().getName());
06        System.out.println("obj의 해쉬코드=" + obj.hashCode());
07     }
08  }
```

```
obj is of type test.Circle
obj의 해쉬코드=1554547125
```

- toString() 예제

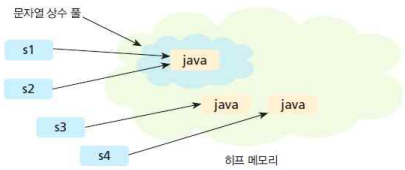
CircleTest.java

```
01  class Circle {
02     int radius;
03     public Circle(int radius) {          this.radius = radius;       }
04     public String toString() {           return "Circle(r="+radius+")";       }
05  }
06  public class CircleTest {
07     public static void main(String[] args) {
08        Circle obj = new Circle(10);
09        System.out.println(obj);
10     }
11  }
```

```
Circle(r=10)
```

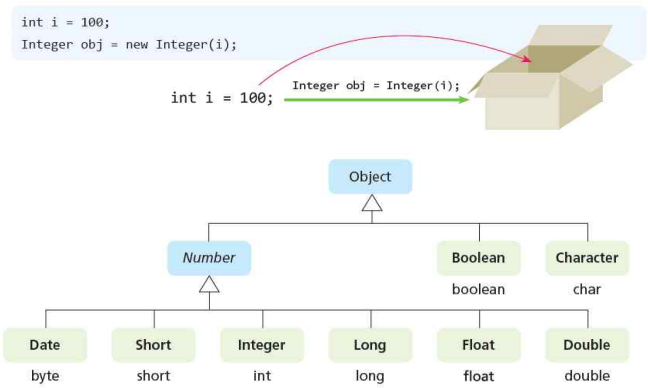
> Circle 클래스에서 toString을 오버라이딩 해주고 있다.
> Object 클래스에 이미 toString이 있기 때문에 오버라이딩이다.

- string



> string s1 = "hi", s2 = "hi" : 스택영역
> string s3 = new string("hi") : 힙영역
> s1 == s2 : ture
> s3 == s4 : false // 주소값의 동일함을 보기 때문

랩퍼 클래스(Wrapper Class)



- 정수와 같은 기초 자료형도 객체로 포장하고 싶은 경우 사용

반환값	메소드 이름	설명
static int	intValue()	int형으로 반환한다.
static double	doubleValue()	double형으로 반환한다.
static float	floatValue()	float형으로 반환한다.
static int	parseInt(String s)	문자열을 int형으로 변환한다.
static String	toBinaryString(int i)	int형의 정수를 2진수 형태의 문자열로 변환한다.
static String	toString(int i)	int형의 정수를 10진수 형태의 문자열로 변환한다.
static Integer	valueOf(String s)	문자열 s를 Integer 객체로 변환한다.
static Integer	valueOf(String s, in radix)	문자열 s를 radix진법의 Integer 객체로 변환한다.

오토박싱

```
Integer obj;

obj = 10;                       // 정수 -> Integer 객체
System.out.println(obj + 1);  // Integer 객체 -> 정수
```

- : 랩퍼 객체, 기초 자료형 변환을 자동으로 해주는 기능

split 동작 신기해서 넣음

```
String[] tokens = "I am a boy.".split(" ");
for (String token : tokens)
   System.out.println(token);
```

```
I
am
a
boy.
```

String Buffer 클래스

```
StringBuffer sb = new StringBuffer("Hello"); // 16바이트의 공간이 할당된다.
int length = sb.length();            // 5
int capacity = sb.capacity();        // 21
```

```
StringBuffer sb = new StringBuffer("10+20=");
sb.append(10+20);
sb.insert(0, "수식 ");                  // sb= "수식 10+20=30"
```

- **String 클래스의 비효율**
  > **상황** 빈번하게 문자열을 변경할 때
  > **이유** : 새 String 객체를 생성 후 기존의 내용을 복사 하기 때문

- 비교

	String	StringBuffer
메모리 위치	String pool	Heep
수정 여부	No(immutable)	Yes(mutable)
스레드 안전성	Yes	Yes
동기화 여부	Yes	Yes
성능	Fast	Slow

예외처리

- 예외 : 잘못된 코드, 부정확 데이터, 예외 상황에 발생하는 오류

```
try {
   // 예외가 발생할 수 있는 코드
} catch (예외클래스 변수) {
   // 예외를 처리하는 코드
}
} finally {
   // 여기 있는 코드는 try 블록이 끝나면 무조건 실행된다.
}
```

생략이 가능

예외

- 그림 너무 귀엽당 희—히

분류	예외	설명
Runtime-Exception	ArithmeticException	어떤 수를 0으로 나누는 경우
	NullPointerException	널 객체를 참조할 때 발생하는 경우
	ClassCastException	적절치 못하게 클래스를 형변환하는 경우
	NegativeArraySizeException	배열의 크기가 음수값인 경우
	OutOfMemoryException	사용 가능한 메모리가 없는 경우
	NoClassDefFoundException	원하는 클래스를 찾지 못하였을 경우
	ArrayIndexOutOfBoundsException	배열을 참조하는 인덱스가 잘못된 경우

분류	예외	설명
Checked-Exception	ClassNotFoundException	클래스가 발견되지 않을 때
	IOException	입출력 오류
	illegalAccessException	클래스의 접근이 금지되었을 때
	NoSuchMethodException	메소드가 발견되지 않을 때
	NoSuchFieldException	필드가 발견되지 않을 때
	InterruptedException	스레드가 다른 스레드에 의하여 중단되었을 때
	FileNotFoundException	파일을 찾지 못했을 때

try-with-resources 문장

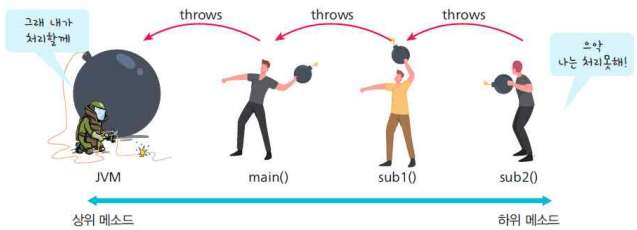
: 문장의 끝에서 리소스들이 자동으로 닫혀지게 함

TryTest.java

```
01  import java.io.*;
02
03  public class TryTest {
04     public static void main(String args[]) {
05        try (FileReader fr = new FileReader("test.txt")) {
06           char[] a = new char[50];
07           fr.read(a);
08           for (char c : a)
09              System.out.print(c);
10        } catch (IOException e) {
11           e.printStackTrace();
12        }
13     }
14  }
```

괄호가 있으면 자원으로 취급한다.

Throw

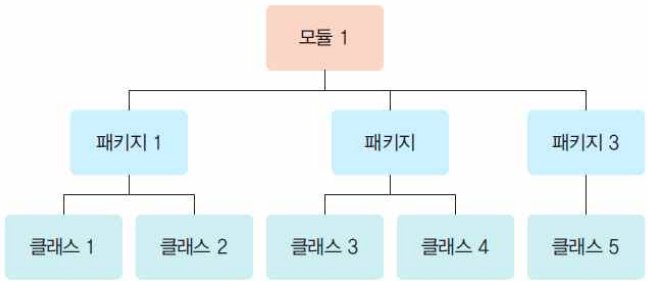


- 상위 메소드로 넘겨가며 폭탄 넘기기
- 전담 메소드를 하나 만들어서 에러를 처리하겠다.
- try-catch가 더 좋다고 생각하신다.

직쏘(Jigsaw) 프로젝트

- 2008~2017에 완료된 OpenJDK 프로젝트
- 자바 API를 모듈로 분리해 소형 컴퓨팅 장치도 동작가능하게 축소

모듈화 장점



- 모듈은 하나 이상의 자바 패키지로 이루어진다.
- 모듈은 패키지의 상위 개념이다

* 9장에서 12장 건너뛰고 13장 시작함

제네릭 프로그래밍(generic programming)

- **정의** : 다양한 타입의 데이터를 처리 가능한 클래스/함수 작성 기법

```
class Box<T> {
   private T data;
   public void set(T data)  {    this.data = data;     }
   public T get()           {    return data;     }
}
```

T는 타입을 의미한다.

```
Box<String> b = new Box<String>();
b.set("Hello World!");     // 문자열 저장
String s = Box.get();
```

```
Box<String> stringBox = new Box<String>();
stringBox.set(new Integer(10));    // 정수 타입을 저장하려고 하면 컴파일 오류!
```

제네릭 메소드

```
public class MyArrayAlg {

   public static <T> T getLast(T[] a) {
      return a[a.length - 1];
   }
}
```

제네릭 메소드 정의

```
public class MyArrayAlgTest {

   public static void main(String[] args) {
      String[] language = { "C++", "C#", "JAVA" };
      String last = MyArrayAlg.getLast(language);    // last는 "JAVA"
      System.out.println(last);
   }
}
```

```
JAVA
```

컬렉션

- **정의** : 자바에서 자료 구조를 구현한 클래스
- **종류** : 리스트, 스택, 큐, 집합(set), 해시 테이블(hash table)

표 13.1 컬렉션 인터페이스

인터페이스	설명
Collection	모든 자료구조의 부모 인터페이스로서 객체의 모임을 나타낸다.
Set	집합(중복된 원소를 가지지 않는)을 나타내는 자료구조
List	순서가 있는 자료구조로 중복된 원소를 가질 수 있다.
Map	키와 값들이 연관되어 있는 사전과 같은 자료구조
Queue	극장에서의 대기줄과 같이 들어온 순서대로 나가는 자료구조

- **특징**
  - 〉 컬렉션 인터페이스와 컬렉션 클래스로 나누어서 제공
  - 〉 제네릭을 사용한다.
  - 〉 int같은 기본 자료형은 저장 불가능
  - 〉 but 기초 자료형을 클래스로 감싼 랩퍼 클래스인 Integer는 가능

  ex) Vector<int> list = new Vector<int>(); // 컴파일 오류!

  ex) Vector<Integer> list = new Vector<Integer>(); // OK
  - 〉 **오토박싱** : 기본자료형 저장 시 자동으로 랩퍼 클래스 객체로 변환

컬렉션 요소 방문

```
String a[] = new String[] { "A", "B", "C", "D", "E" };
List<String> list = Arrays.asList(a);
```

① for문 이용

```
for (int i=0; i<list.size(); i++)
    System.out.println(list.get(i));
```

② for-each 이용

```
for (String s: list)
    System.out.println(s);
```

③ 반복자 이용

메소드	설명
hasNext()	아직 방문하지 않은 원소가 있으면 true를 반환
next()	다음 원소를 반환
remove()	최근에 반환된 원소를 삭제

```
String s;
Iterator e = list.iterator();
while(e.hasNext())
{
   s = (String)e.next();              // 반복자는 Object 타입을 반환!
   System.out.println(s);
}
```

④ Stream 라이브러리를 이용

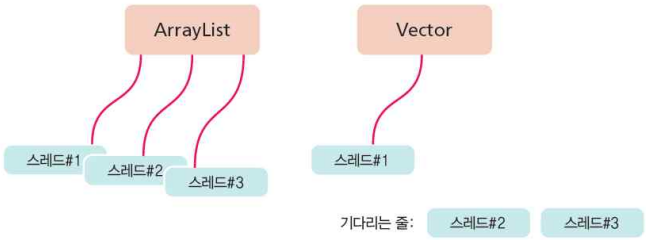
```
list.forEach((n) -> System.out.println(n));
```

* 반복자 이용 이거 기억해주세요

05월 23일 - 월

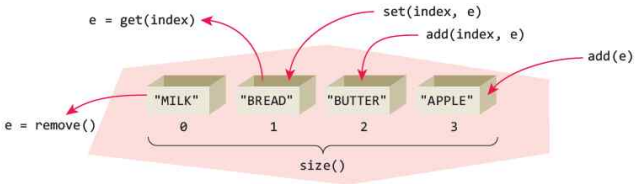
Vector vs ArrayList

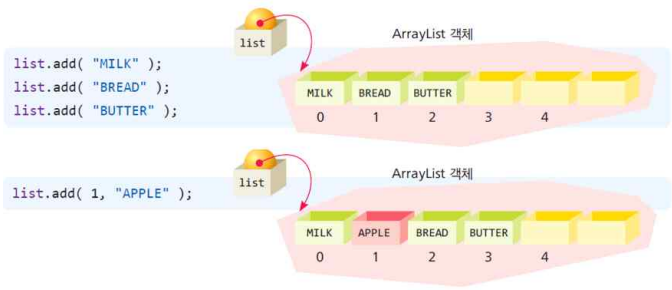
- Vector는 스레드 간의 동기화를 지원
- ArrayList는 동기화를 하지 않아 Vector보다 성능은 우수
  but 스케쥴링에서 오류가 생길 가능성 존재



* 벡터랑 어레이리스트의 차이! 이게 중요하지요!

- **ArrayList**
  - 〉 배열(Array)의 향상된 버전, 가변 크기의 배열
  - 〉 **생성** : ArrayList<String> list = new ArrayList<String>();





불행하게도 자바에서는 배열, ArrayList, 문자열 객체의 크기를 알아내는 방법이 약간 다르다.

- 배열: array.length
- ArrayList: arrayList.size()
- 문자열: string.length()

LinkedList

- 빈번하게 삽입과 삭제가 일어나는 경우에 사용

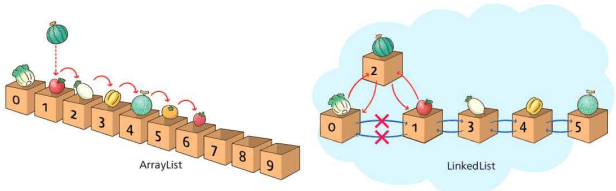


그림 13.6 배열의 중간에 데이터를 삽입하려면 원소들을 이동하여야 한다. 연결 리스트 중간에 삽입하려면 링크만 수정하면 된다.

ArrayList vs LinkedList

- **ArrayList**
  - 〉인덱스를 가지고 원소에 접근할 경우, 항상 일정한 시간만 소요
  - 〉리스트의 각각의 원소를 위하여 노드 객체를 할당할 필요가 없다.
  - 〉동시에 많은 원소를 이동하는 경우 System.arraycopy()를 사용

- **LinkedList**
  - 〉앞에 빈번하게 원소를 추가하거나, 삭제를 반복하는 경우에는 LinkedList를 사용하는 것이 낫다.
  - 〉원소의 개수에 비례하는 시간이 소요

Set

- 원소의 중복을 허용하지 않음

```
import java.util.*;

public class SetTest {
   public static void main(String args[]) {
      HashSet<String> set = new HashSet<String>();

      set.add("Milk");
      set.add("Bread");
      set.add("Butter");
      set.add("Cheese");
      set.add("Ham");
      set.add("Ham");

      System.out.println(set);

      if (set.contains("Ham")) {
         System.out.println("Ham도 포함되어 있음");
      }
   }
}
```

- **HashSet**
  - 〉해쉬 테이블에 원소를 저장하기 때문에 성능면에서 가장 우수하다.
  - 〉**원소들의 순서가 일정하지 않은** 단점이 있다.

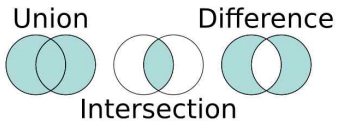
- **TreeSet**
  - 〉레드-블랙 트리(red-black tree)에 원소를 저장한다.
  - 〉값에 따라서 순서가 결정되며, HashSet보다는 느리다.

- **LinkedHashSet**
  - 〉해쉬 테이블과 연결 리스트를 결합한 것
  - 〉원소들의 순서는 삽입되었던 순서와 같다.

- 파일로 데이터 관리 시 유용

- **List와의 차이점**
  - 〉Set은 순서가 없다.
  - 〉중복을 허용하지 않는다.

Set : 대량 연산 메소드



- s1.**containsAll**(s2) — 만약 s2가 s1의 부분 집합이면 참
- s1.**addAll**(s2) — s1을 s1과 s2의 합집합으로
- s1.**retainAll**(s2) — s1을 s1과 s2의 교집합으로
- s1.**removeAll**(s2) — s1을 s1과 s2의 차집합으로

```
Set<Integer> s1 = new HashSet<>(Arrays.asList(1, 2, 3, 4, 5, 7, 9));
Set<Integer> s2 = new HashSet<>(Arrays.asList(2, 4, 6, 8));

s1.retainAll(s2);      // 교집합을 계산한다.
System.out.println(s1);
```

```
[2, 4]
```

Set : 중복 단어 검출하기

FindDupplication.java

```
21  import java.util.*;
22
23  public class FindDupplication {
24     public static void main(String[] args) {
25        Set<String> s = new HashSet<String>();
26        String[] sample = { "사과", "사과", "바나나", "토마토" };
27        for (String a : sample)
28           if (!s.add(a))
29              System.out.println("중복된 단어: " + a);
30
31        System.out.println(s.size() + " 중복되지 않은 단어: " + s);
32     }
33  }
```

집합에 추가되지 않으면 중복된 단어이다.

```
중복된 단어: 사과
3 중복되지 않은 단어: [토마토, 사과, 바나나]
```

- Set은 중복을 허용하지 않음, add의 성공여부로 중복을 판단

Map

- 원하는 데이터를 빠르게 찾을 수 있는 자료구조
- 사전과 같은 형태

```
Map<String, String> map = Map.of("kim", "1234", "park", "pass", "lee", "word");
```

키(key)	값(value)
"kim"	"1234"
"park"	"pass"
"lee"	"word"

그림 13.8 Map의 개념

- put(key, value), get(key)

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

public class MapTest {
	public static void main(String[] args) {
		Map<String, String> map = new HashMap<String, String>();

		map.put("kim", "1234");
		map.put("park", "pass");
		map.put("lee", "word");

		System.out.println(map.get("lee"));		// 키를 가지고 값을 참조한다.

		for (String key: map.keySet()) {		// 모든 항목을 방문한다.
			String value = map.get(key);
			System.out.println("key=" + key + ", value=" + value);
		}
		map.remove(3);			// 하나의 항목을 삭제한다.
		map.put("choi", "password");		// 하나의 항목을 대치한다.
		System.out.println(map);
	}
}
```

keySet()은 키들의 집합을 변환한다.

- map의 모든 요소 가져오기 : map.keySet()
- **삭제 시 주의** : remove() Key값이 3인걸 삭제하는 것

* 의심하고 돌려보자, 의심할 수 있어 행복하다
* 순서가 왜 뒤죽박죽? :
https://www.scaler.com/topics/hashcode-in-java/#introduction

우선순위 큐

- **remove**를 호출 할 때마다 **가장 작은 원소 호출**
- 내부적으로 **힙(heap)** 자료구조 이용

PriorityQueueTest.java

```java
import java.util.*;

public class PriorityQueueTest {
   public static void main(String[] args) {
      PriorityQueue<Integer> pq = new PriorityQueue<Integer>();
      pq.add(30);
      pq.add(80);
      pq.add(20);

      System.out.print(pq);
      System.out.println("삭제된 원소: " + pq.remove());
   }
}
```

```
[20, 80, 30]
삭제된 원소: 20
```

- 정렬보단, 제일 작은게 제일 앞으로 오는 구나!

Collections 클래스

- 여러 유용한 알고리즘을 구현한 메소드들을 제공한다.
  - 〉 정렬(Sorting)
  - 〉 섞기(Shuffling)
  - 〉 탐색(Searching)
- 정렬 예제

```java
import java.util.*;

public class Sort {
   public static void main(String[] args) {
      String[] sample = { "i", "walk", "the", "line" };
      List<String> list = Arrays.asList(sample);		// 배열을 리스트로 변경
      Collections.sort(list);
      System.out.println(list);
   }
}
[i, line, the, walk]
      }
}
```

정렬 알고리즘을 실행하기 위하여 asList() 메소드를 이용하여 배열을 리스트로 변환한다.

Collections 인터페이스가 가지고 있는 정적 메소드인 sort()을 호출하여서 정렬을 수행한다.

```
[i, line, the, walk]
```

스트림

그림 15.2 스트림의 분류

- **정의** : 순서가 있는 데이터의 연속적인 흐름

- **바이트 스트림**
  〉 바이트 단위로 입출력 하는 스트림

- **문자스트림**
  〉 문자 단위로 입출력 하는 스트림
  〉 왜 16 바이트? : 자바 String은 2바이트라서

문자 스트림

- 모든 문자 스트림 클래스는 Reader와 Writer 클래스에서 상속된다.
- 모든 문자 스트림은 Reader와 Writer로부터 파생된다.

반환형	메소드	설명
int	read()	입력 스트림에서 한개의 문자를 읽는다. 반환값은 0에서 65535 범위(0x0000에서 0xFFFF)의 유니코드 값이다. 스트림이 종료되면 -1을 반환한다.
int	read(char[] cbuf)	입력 스트림에서 문자를 읽어서 cbuf[]에 저장하고 읽은 개수를 반환한다.
int	skip(long n)	입력 스트림에서 n만큼의 문자를 건너뛴다
void	close()	입력 스트림을 닫는다.

Reader 클래스의 주요 메소드

- **읽어오기**

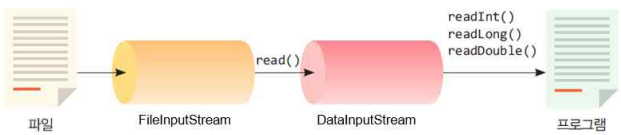
```java
int ch;
while ((ch = fr.read()) != -1)
   System.out.print((char) ch +" ");
```

〉 -1 : EOF(End of File)

* ANSI : 윈도우는 UTF-8로 처리하지 않는다. ANSI로 처리한다.

이진 파일에서 정수 읽기

```java
DataInputStream dataSt = new DataInputStream(new FileInputStream("data.bin"));
int i = dataSt.readInt();
```



버퍼 스트림

〉 입력 장치에서 한번에 많이 읽어서 버퍼에 저장
〉 프로그램이 입력을 요구하면 버퍼에서 꺼내서 반환
〉 버퍼가 비었을 때만 입력 장치에서 읽는다
=> **왜 버퍼씀?** 물리적(디스크)/전자적(프로그램) 처리 시간의 차이.

* 버퍼는 메모리에 있대
* 양쪽 시스템간의 성능 차이를 최소화하기 위해

프로세스와 스레드

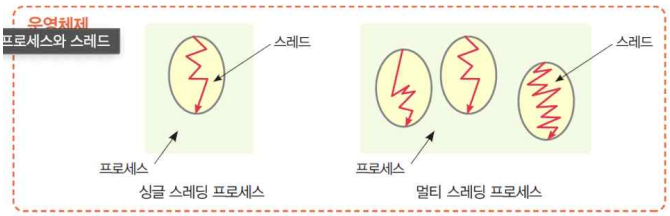


그림 16.2 스레드는 하나의 프로세스 안에 존재한다.

- **프로세스** : 자신만의 작업을 가진다. 실행되는 최소 프로그램 단위
  > ex. 크롬, ppt 등
- **스레드** : 동일한 데이터 공유
  > ex. ftp로 다운로드 받으면서, 스트림도 받고, 등등

스레드를 사용하는 이유

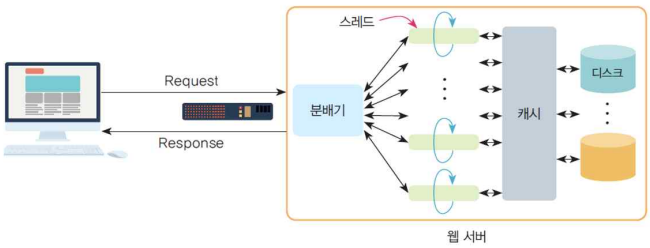


그림 16.3 웹 서버에서의 스레드 이용

- 프로그램을 보다 빠르게 실행하기 위해.
- CPU는 빠르고 코어가 많기 때문에 하나 스레드로는 코어 다 못 씀
- 멀티스레딩을 쓰면 여러 코어를 최대한 활용할 수 있다.

- SW가 필요로 하는 성능이 낮아지고 있다.

멀티스레딩의 문제점

- 같은 데이터를 공유하게 되면 동기화 문제가 발생한다.
- 많은 대처법 : 락킹
  > 일단 무조건 먼저 도착하는 작업 이후는 락 걸어놓음
  > 다른 스레드에서 락 해제할 때까지 못 쓴다.

스레드 생성과 실행

```
Thread t = new Tread();
t.start();
```

- 스레드 생성은 Tread 클래스가 담당하지!

```
Thread t = new Tread();
Thread t1 = new Tread();
Thread t2 = new Tread();
Thread t3 = new Tread();

t.start();    // 이거 실행하면 run메소드 실행하러 가고 분기됨
t1.start();   // 분기분기
t2.start();   // 분기분기분기
t3.start();   // 분기분기분기분기
```

Thread 클래스

메소드	설명
Thread()	매개 변수가 없는 기본 생성자
Thread(String name)	이름이 name인 Thread 객체를 생성한다
static int activeCount()	현재 활동 중인 스레드의 개수를 반환한다.
String getName()	스레드의 이름을 반환한다.
intgtPriority()	스레드의 우선 순위를 반환한다.
void interrupt()	현재의 스레드를 중단한다.
boolean isInterrupted()	현재의 스레드가 중단될 수 있는지를 검사한다.
void setPriority(int priority)	스레드의 우선 순위를 지정한다.
void setName(String name)	스레드의 이름을 지정한다.
static void sleep(int milliseconds)	현재의 스레드를 지정된 시간만큼 재운다.
void run()	스레드가 해야 하는 작업을 이 메소드 안에 위치시킨다. 스레드가 시작될 때 호출된다.
void start()	스레드를 시작한다.
static void yield()	현재 스레드를 다른 스레드에 양보하게 만든다.

Thread 생성법

- **Thread 클래스를 상속하는 방법:**
  1) Thread 클래스를 상속받은 후
  2) run() 메소드를 재정의한다.
  3) run() 메소드 안에 작업을 기술한다.

* 선호하지 않는 방법 왜? 다중 상속이 지원되지 않기 때문

- **Runnable 인터페이스를 구현하는 방법:**
  > run() 메소드를 가지고 있는 클래스를 작성
  > 이 클래스의 객체를 Thread 클래스의 생성자를 호출 시 전달

Thread 클래스 상속하는 방법

MyThreadTest.java

```
01 class MyThread implements Thread {        // ①
02    public void run() {                    // ②
03       for (int i = 0; i <= 10; i++)
04          System.out.print(i + " ");
05    }
06 }
07
08 public class MyThreadTest {
09    public static void main(String args[]) {
10       Thread t = new MyThread(); // ③
11       t.start();                 // ④
12    }
13 }
0 1 2 3 4 5 6 7 8 9 10
```

MyThread 클래스는 Thread를 상속받는다. MyThread 클래스는 하나의 메소드 run()만을 가지고 있는데 run()은 이 스레드가 시작되면 자바 런타임 시스템에 의하여 호출된다. 스레드가 실행하는 모든 작업은 이 run() 메소드 안에 있어야 한다. 현재는 단순히 0부터 10까지를 화면에 출력한다.

스레드를 실행시키려면 Thread에서 파생된 클래스 MyThread의 인스턴스를 생성한 후 start()를 호출한다. Thread 타입의 변수 t가 선언되고 MyThread의 객체를 생성하였다. 객체가 생성되었다고 스레드가 바로 시작되는 것은 아니다. start() 메소드를 호출해야만 스레드가 실행된다.

- 저기에 t, t1 스레도 있고 둘 다 돌리면 결과 뭐게?
  > 예측 못 해요 정확히 맞추는 분 있으면 시험 면제 해줄게

```
static int x = 0;
public void run() {
      incx(); // x++
      Thread.sleep(100);
      decx(); // y++
}
```

- 돌리면 0만 나올 것 같지만 실제로는 2 3 4 3 이럼
- 왜? 다른 함수가 접근해서 증가하고 막 그럼

Runnable 인터페이스 구현하는 방법

MyRunnableTest.java

```
01  class MyRunnable implements Runnable {  // ①
02     public void run() {                   // ②
03        for (int i = 0; i <= 10; i++)
04           System.out.print(i + " ");
05     }
06  }
07
08  public class MyRunnableTest {
09     public static void main(String args[]) {
10        Thread t = new Thread(new MyRunnable());  // ③
11        t.start();                                // ④
12     }
13  }
0 1 2 3 4 5 6 7 8 9 10
```

Runnable을 구현하는 클래스를 작성한다. run() 메소드를 재정의하여 작업에 필요한 코드를 넣는다.

Thread 클래스의 인스턴스를 생성하고, Runnable 객체를 Thread 생성자의 매개 변수로 넘긴다. Thread 객체의 start() 메소드를 호출하여야 한다.

① Runnable 인터페이스를 구현한 클래스를 작성
② run() 메소드를 작성
③ Thread 객체를 생성하고 이때 MyRunnable 객체를 인수로 전달
④ start()를 호출하여서 스레드를 시작

스레드 상태

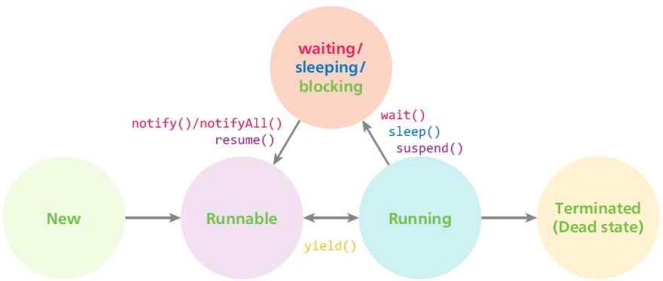


그림 16.5 스레드의 상태

- **New 상태**
  > Thread 클래스의 인스턴스는 생성되었지만
  > start() 메소드를 호출하기 전이라면 스레드는 New 상태에 있다.

- **Runnable 상태**
  > start() 메소드가 호출되면 스레드는 실행 가능한 상태가 된다.
  > 하지만 아직 스레드 스케줄러가 선택하지 않았으므로
  > 실행 상태는 아니다.

- **Running 상태**
  > 스레드 스케줄러가 스레드를 선택하면,
  > 스레드는 실행 중인 상태가 된다.

- **Blocking 상태**
  > 스레드가 아직 살아 있지만,
  > 여러 가지 이유로 현재 실행할 수 없는 상태이다.

- **Terminated 상태**
  > 스레드가 종료된 상태이다.
  > 스레드의 run() 메소드가 종료되면 스레드도 종료된다.

* 갑자기 다 넘기고 스레드 상태
* 시험에 잘 나오는 내용임니다.

교수님의 알쓸신잡

- **인터럽트(interrupt)**
  > 하나의 스레드가 실행하고 있는 작업을 중지하도록 하는 메커니즘
  ex) usb 꼽기, 전원 뽑기

- 콤푸타 업구레이드 하고 싶으면
  1) 메인 메모리부터 올려라
  2) SSD를 올려라
  3) 그래픽 카드를 올려라
  ...
  맨마지막) CPU를 올려라

- **Disk의 성능 단위는** : rpm단위(분당 회전수) 이다.

생성 상태와 실행 가능 상태

- **생성 상태**
  > Thread 클래스를 이용하여 새로운 스레드를 생성
  > start()는 생성된 스레드를 시작
  > stop()은 생성된 스레드를 멈추게 한다.

- **실행 가능 상태**
  > 스레드가 스케줄링 큐에 넣어지고
  > 스케줄러에 의해 우선순위에 따라 실행

* **최우선 인터럽트** : 셧다운 - 모든 잔류 전원을 저장에 쓴다.
* **라운드 로빙** : 많은 큐를 사용한다. 최상위 큐의 작업 수행

실행 중지 상태

- 발생 : 실행 가능한 상태에서 다음의 이벤트가 발생 시
- 스레드나 다른 스레드가 suspend()를 호출하는 경우
- 스레드가 wait()를 호출하는 경우
- 스레드가 sleep()을 호출하는 경우
- 스레드가 입출력 작업을 하기 위해 대기하는 경우

실행 중지 상태

- 어떤 스케줄링을 선택하느냐는 자바 가상 머신에 의하여 결정된다.

- **선점형 스케쥴링** : 우선순위를 부여하여 우선순위대로 진행
  > 이게 타이트해질수록 직렬화로 변화된다.

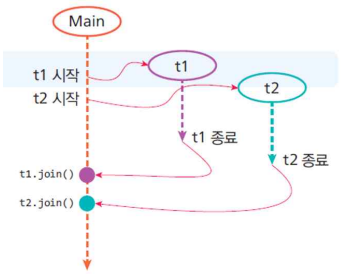
- **타임 슬라이싱** : 시간 잘라서 씀

스레드 우선순위

- 우선순위는 1에서 10 사이의 숫자로 표시된다. 스레드의 기본 우선 순위는 NORM_PRIORITY이다. MIN_PRIORITY의 값은 1이고 MAX_PRIORITY의 값은 10이다.
  - 정적 정수 MIN_PRIORITY(1)
  - 정적 정수 NORM_PRIORITY(5)
  - 정적 정수 MAX_PRIORITY(10)

- void setPriority(int newPriority): 현재 스레드의 우선 순위를 변경한다.
- getPriority(): 현재 스레드의 우선 순위를 반환한다.

join()

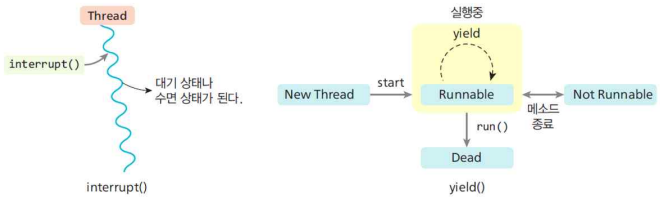


- 스레드가 종료될 때까지 기다리는 메소드
- 특정 스레드 종료될 때까지 현재 스레드 실행을 중지하고 기다림
- 어떻게 보면 직렬화 기법이다.

```java
public class JoinTest extends Thread {
   public void run() {
      for (int i = 1; i <= 3; i++) {
         System.out.println(getName() + " "+i);
      }
   }

   public static void main(String args[]) {
      JoinTest t1 = new JoinTest();
      JoinTest t2 = new JoinTest();
      t1.start();
      try {
         t1.join();
      } catch (Exception e) {
         System.out.println(e);
      }
      t2.start();
   }
}
```

```
Thread-0 1
Thread-0 2
Thread-0 3
Thread-1 1
Thread-1 2
Thread-1 3
```

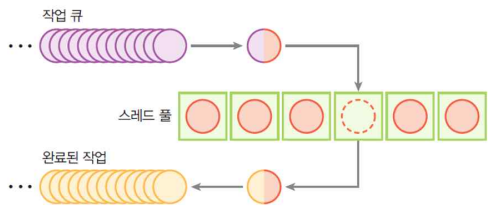
인터럽트(interrupt)와 yield()



- 인터럽트 : 하나의 스레드가 실행하는 작업을 중지하는 메커니즘
- yield() : cpu를 다른 스레드에게 양보하는 메소드

* 메소드 종료 -> run(), run() -> " "
* yield : 일드라고 읽네

자바 스레드 풀



- **정의** : 미리 초기화된 스레드들이 모여 있는 곳
- **일반적인 스레드가 저장된 컬렉션의 크기** : 고정, 반드시 그런 거x
- 스레드 풀의 동일한 스레드로 N개의 작업을 쉽게 실행 가능
- 스레드의 개수보다 작업의 개수가 더 많은 경우
  > 작업은 FIFO 큐에서 기다려야 한다.

스레드 사용 시 주의점

- 동일한 메모리를 사용하기 때문에 2가지의 문제가 발생할 수 있다.

1) **스레드 간섭**(thread interference)

2) **메모리 불일치 문제**(consistency problem)

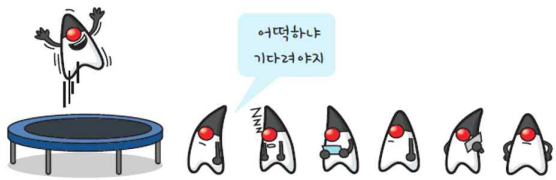
* 서술형으로 나올 듯

동기화(synchronization)

- 한 번에 하나의 스레드 만이 공유 데이터를 접근할 수 있도록 제어
- **기본 해법** : 하나의 스레드만 방문을 열고 사용할 수 있게 하는 것

- **자바 제공 방법**
  > **동기화 메소드**(synchronized method)
  > **동기화 블록**(synchronized block)
  > **정적 동기화**(static synchronization)

- **동기화 방법**
  > 락(lock) or 모니터(monitor)로 알려진 방법을 이용

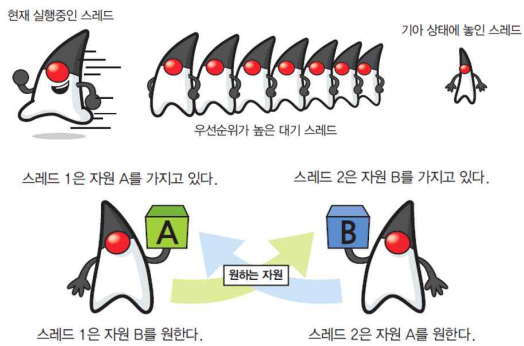


* 자바 캐릭터 너무 귀여워 하심
* 기다리다가 자꾸 누가 낚아채면 화나지?
* 오래 기다리는 큐, 프로세스에 대해서 우선순위를 빡빡 높여준다.
* 인터럽트 발생 시 우선순위 무시 ex) 정전

교착 상태

- 동일한 자원을 접근하려고,
- 동기화를 기다리면서 **대기하는 스레드들이 많아지면**
- 자바 가상 머신이 느려지거나 일시 중단되기도 한다.
  => **교착 상태**(deadlock), **기아**(starvation), **라이브락**(livelock)라 함

- **해결법** : 잠금 순서 변경 or sleep



* 이미지는 기아상태와 라이브락 상태
* 0.1초면 1억개의 명령어를 처리할 시간

스레드간의 조정

- **발생** : 2개의 스레드가 데이터를 주고 받을 때
- **예제 상황** : 한 스레드는 계속 값생성, 다른 스레드는 계속 값 출력

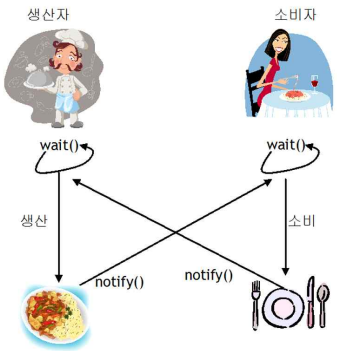
```
static int x = 0;
public void run() {
      incx(); // x += 1
      Thread.sleep(100);
      decx(); // y -= 1
}
```

- 이 결과가 이쁘게 0으로 안 끝나는게 많다.

- **해결법**
  - 〉 **polling** : 계속 대기한다.
  - 〉 **wait & notify** : 이벤트 처리하듯이

좋지 못한 방법(polling) 좋은 방법(wait & notify)

* 프로그램의 공유 데이터 처리를 위해선 직렬화가 거의 이용됨
* 게임 레이드 뛸 때 체력 깎는 것도 직렬로 됨



시험범위

- 6, 7, 8, 13, 15, 16
- 상속이나 스레드를 많이 보라
- 이 책 내용만 나오진 않아용
- 10시부터 11시까지
- 단답형 중심, 프로그램 결과도 있당
- 성적처리 1주일 걸려용, A꽉꽉 눌러 채울게용
- 12027
- 스레드 동기화 예제 실습 문제 풀이 과제 올라와용