



Java



중첩 클래스 (Nested Class)

- 클래스 간 서로 긴밀한 관계를 맺고 상호 작용하면서 객체지향 프로그램은 동작한다.
- 클래스가 여러 클래스와 관계를 맺는 경우에는 독립적으로 선언하는 것이 좋으나, 특정 클래스만 관계를 맺어야 하는 경우에는 중첩 클래스로 선언하는 것이 유지보수에 도움이 되는 경우가 많다.
- 중첩 클래스란 선언하는 위치에 따라 두 가지로 분류된다.
 - 멤버 클래스: 클래스의 멤버로서 선언되는 중첩 클래스 (인스턴스 멤버 클래스, 정적 멤버 클래스)
 - 로컬 클래스: 메소드 내부에서 선언되는 중첩 클래스

TIP

중첩 클래스도 컴파일 시에 별도의 바이트코드 파일이 생성된다.

- 바깥클래스\$멤버클래스.class
- 바깥클래스\$1중첩클래스.class



중첩 클래스 (Nested Class) – 인스턴스 멤버 클래스

- 인스턴스 멤버 클래스는 아래와 같이 선언된다.

```
package com.nestedclass.member;

public class A {
    public class B {
        // 다른 패키지에서 B 클래스 사용가능
    }
    class C {
        // 같은 패키지에서만 C 클래스 사용가능
    }
    private class D {
        // A 클래스 내부에서만 D 클래스 사용가능
    }
}
```

- 인스턴스 멤버 클래스는 주로 클래스 내부에서 사용되기 때문에 D처럼 private으로 선언하는 것이 일반적이다.



중첩 클래스 (Nested Class) – 인스턴스 멤버 클래스

- 인스턴스 멤버 클래스의 생성은 아래와 같은 위치에서 생성 가능하다.

```
package com.nestedclass.member;

public class A {
    class NestedClass {

        // 필드값으로 생성
        NestedClass nc = new NestedClass();

        // 생성자에서 생성
        A() {
            NestedClass nc = new NestedClass();
        }

        // 메소드에서 생성
        void method() {
            NestedClass nc = new NestedClass();
        }
    }
}
```



중첩 클래스 (Nested Class) – 인스턴스 멤버 클래스

- 외부에서 중첩 클래스 NestedClass를 생성하려면, A 객체를 먼저 생성한 다음 NestedClass를 생성해야 한다.

```
package com.nestedclass.member;
```

```
public class AExample {  
    public static void main(String[] args) {  
        A a = new A();  
        A.NestedClass nc = a.new NestedClass();  
    }  
}
```

- 인스턴스 멤버 클래스 내부에는 필드, 생성자, 메소드 선언이 올 수 있다. Java17부터 정적 필드와 정적 메소드의 선언도 가능하다.



중첩 클래스 (Nested Class) – 정적 멤버 클래스

- 정적 멤버 클래스는 아래와 같이 클래스 내부에 static 키워드와 함께 선언된 클래스를 말한다.

```
package com.nestedclass.member2;  
  
public class A {  
    public static class B {  
        // 다른 패키지에서 B 클래스 사용가능  
    }  
    static class C {  
        // 같은 패키지에서만 C 클래스 사용가능  
    }  
    private static class D {  
        // A 클래스 내부에서만 D 클래스 사용가능  
    }  
}
```

- 정적 멤버 클래스는 주로 클래스 외부에서 A와 함께 사용되기 때문에 default 또는 public으로 선언하는 것이 일반적이다.



중첩 클래스 (Nested Class) – 정적 멤버 클래스

- 정적 멤버 클래스의 생성은 아래와 같은 위치에서 생성 가능하다.

```
package com.nestedclass.member2;

public class A {
    static class NestedClass { }

    // 필드값으로 생성
    NestedClass nc1 = new NestedClass();

    // 정적필드값으로 생성
    static NestedClass nc2 = new NestedClass();

    // 생성자에서 생성
    A() {
        NestedClass nc = new NestedClass();
    }

    // 메소드에서 생성
    void method1() {
        NestedClass nc = new NestedClass();
    }

    // 정적메소드에서 생성
    static void method2() {
        NestedClass nc = new NestedClass();
    }
}
```



중첩 클래스 (Nested Class) – 정적 멤버 클래스

- 외부에서 중첩 클래스 NestedClass를 생성할 때, A 객체 생성없이 NestedClass를 생성한다.

```
package com.nestedclass.member2;
```

```
public class AExample {  
    public static void main(String[] args) {  
        A.NestedClass nc = new A.NestedClass();  
    }  
}
```

- 정적 멤버 클래스 내부에도 필드, 생성자, 메소드 선언이 올 수 있다. Java17부터 정적 필드와 정적 메소드의 선언도 가능하다.



중첩 클래스 (Nested Class) – 로컬 클래스

- 생성자 또는 메소드 내부에서 아래와 같이 선언된 클래스를 로컬 클래스라 한다.

```
package com.nestedclass.local;
```

```
public class A {  
    public A() {  
        // 생성자 내부에서 선언된 로컬클래스  
        class B {}  
  
        // 생성자 실행동안만 객체 생성 가능  
        B b = new B();  
    }  
    public void method() {  
        // 메서드 내부에서 선언된 로컬클래스  
        class B {}  
  
        // 메소드 실행동안만 객체 생성 가능  
        B b = new B();  
    }  
}
```



중첩 클래스 (Nested Class) - 로컬 클래스

- 생성자 또는 메소드의 매개변수나 내부에서 선언된 변수를 로컬 클래스에서 사용할 경우에는 로컬 클래스 내부에서 값을 변경하지 못하도록 제한되어 있다. 즉, final 특성을 갖게 된다.

TIP

Java 7 이전에는 final 키워드를 반드시 붙여야했는데,
Java 8 부터는 final을 붙이지 않아도 final 특성을 갖는다.

- 로컬 클래스 내부에도 필드, 생성자, 메소드 선언이 올 수 있다. Java17부터 정적 필드와 정적 메소드의 선언도 가능하다.



중첩 클래스 (Nested Class) – 바깥 클래스에 접근

- 인스턴스 멤버 클래스는 바깥 클래스가 생성되어야 생성이 가능하다는 특징이 있다.
- 따라서 바깥 클래스의 모든 필드와 메소드에 접근 가능하다.
- 하지만 정적 멤버 클래스는 바깥 클래스가 없어도 사용 가능해야 한다.
- 따라서 바깥 클래스의 인스턴스 필드와 인스턴스 메소드 접근은 불가능하고, 정적 필드와 정적 메소드만 접근할 수 있다.
- 만약 중첩 클래스 안에서 바깥클래스의 객체를 얻기 위해서는 바깥클래스 이름에 `this`를 붙여주면 된다.



```
package com.nestedclass;

public class Outer {
    String name = "바깥쪽";
    class Inner {
        String name = "안쪽";
        void method() {
            System.out.println("\t안쪽 메소드");
        }
        void useInner() {
            System.out.println("안에서 안쪽 필드와 메소드 사용");
            System.out.println("\t" + this.name);
            this.method();
        }
        void useOuter() {
            System.out.println("안에서 바깥쪽 필드와 메소드 사용");
            System.out.println("\t" + Outer.this.name);
            Outer.this.method();
        }
    }
    void method() {
        System.out.println("\t바깥쪽 메소드");
    }
    void useOuter() {
        System.out.println("바깥에서 바깥쪽 필드와 메소드 사용");
        System.out.println("\t" + this.name);
        this.method();
    }
    void useInner() {
        System.out.println("바깥에서 안쪽 필드와 메소드 사용");
        Inner i = new Inner();
        System.out.println("\t" + i.name);
        i.method();
    }
}
```



```
package com.nestedclass;
```

```
public class OuterExample {  
    public static void main(String[] args) {  
        Outer o = new Outer();  
        o.useInner();  
        o.useOuter();  
  
        Outer.Inner i = o.new Inner();  
        i.useInner();  
        i.useOuter();  
    }  
}
```



중첩 인터페이스 (Nested Interface)

- 클래스의 멤버로 선언된 인터페이스를 중첩 인터페이스라고 한다.
- 특정 클래스와 긴밀한 관계를 맺는 구현 객체를 만들기 위해서 아래와 같이 중첩 인터페이스를 선언한다.

```
package com.nestedinterface;

public class A {
    public interface B {}
    // public static interface B {}

    interface C {}
    // static interface C {}

    private interface D {}
    // private static interface D {}
}
```

- 외부의 접근을 막지 않으려면 public을 붙이고, 클래스 내부에서만 사용하려면 private를 붙인다. 접근 제한자를 붙이지 않으면 같은 패키지 안에서만 접근이 가능하다.
- 중첩 인터페이스는 암시적으로 static이므로 생략해도 항상 A 객체 없이 인터페이스 사용이 가능하다.



중첩 인터페이스 (Nested Interface)

- 중첩 인터페이스는 UI 프로그램에서 이벤트를 처리할 목적으로 많이 사용된다.

```
package com.nestedinterface2;

public class Button {
    // 중첩 인터페이스 (static) 생략
    public static interface ClickListener {
        void onClick();
    }

    // 필드
    private ClickListener clickListener;

    // 메소드
    public void setClickListener(ClickListener clickListener) {
        this.clickListener = clickListener;
    }
    public void click() {
        this.clickListener.onClick();
    }
}
```



중첩 인터페이스 (Nested Interface)

```
package com.nestedinterface2;
```

```
public class ButtonExample {  
    public static void main(String[] args) {  
        Button btnOk = new Button();  
        class OkListener implements Button.ClickListener {  
            @Override  
            public void onClick() {  
                System.out.println("OK 버튼을 눌렀습니다.");  
            }  
        }  
        btnOk.setClickListener(new OkListener());  
        btnOk.click();  
  
        Button btnCancel = new Button();  
        class CancelListener implements Button.ClickListener {  
            @Override  
            public void onClick() {  
                System.out.println("취소 버튼 눌렀습니다.");  
            }  
        }  
        btnCancel.setClickListener(new CancelListener());  
        btnCancel.click();  
    }  
}
```




익명 객체 (Anonymous Object)

- 익명 객체는 명시적으로 클래스를 선언하지 않은 이름이 없는 객체를 의미한다.
- 명시적으로 클래스를 선언하지 않았기 때문에 쉽게 객체 생성이 가능하다.
- 익명 객체는 필드값, 로컬 변수값, 매개변수값으로 주로 사용된다.
- 익명 객체를 생성하기 위해서는 클래스를 상속하거나 인터페이스를 구현해야 한다.
 - 클래스를 상속해서 만들면 익명 자식 객체라고 한다.
 - 인터페이스를 구현해서 만들면 익명 구현 객체라고 한다.



익명 객체 (Anonymous Object) – 익명 자식 객체

- 부모 클래스를 상속받아 생성되는 익명 자식 객체는 아래와 같이 생성된다.

```
new 부모생성자(매개값, ...) {  
    // 필드  
    // 메소드  
};
```

- 익명 자식 객체는 부모 타입의 필드, 로컬 변수, 매개변수의 값으로 대입할 수 있다.



익명 객체 (Anonymous Object) – 익명 자식 객체

- 부모 타입의 필드로 대입되는 익명 자식 객체

```
package com.anonymous.inheritance;

public class Tire {
    public void roll() {
        System.out.println("0. 일반 타이어가 굴러갑니다.");
    }
}

public class Car {
    // 필드에 Tire 객체 대입
    private Tire tire1 = new Tire();

    // 필드에 익명자식객체 대입
    private Tire tire2 = new Tire() {
        @Override
        public void roll() {
            System.out.println("1. 익명 자식 객체 Tire가 굴러갑니다.");
        }
    };

    // 필드 사용 메소드
    public void run1() {
        tire1.roll();
        tire2.roll();
    }
}
```



익명 객체 (Anonymous Object) – 익명 자식 객체

- 로컬 변수로 대입되는 익명 자식 객체

```
package com.anonymous.inheritance;

public class Car {
    .....
    // 로컬변수 사용 메소드
    public void run2() {
        // 로컬변수에 익명 자식 객체 대입
        Tire tire = new Tire() {
            @Override
            public void roll() {
                System.out.println("2. 익명 자식 객체 Tire가 굴러갑니다.");
            }
        };
        tire.roll();
    }
}
```



익명 객체 (Anonymous Object) – 익명 자식 객체

- 메소드의 매개변수 값으로 대입되는 익명 자식 객체

```
package com.anonymous.inheritance;  
  
public class Car {  
    .....  
    // 매개변수 사용 메소드  
    public void run3(Tire tire) {  
        tire.roll();  
    }  
}
```



익명 객체 (Anonymous Object) – 익명 자식 객체

```
package com.anonymous.inheritance;

public class CarExample {
    public static void main(String[] args) {
        Car car = new Car();
        car.run1();
        car.run2();
        // 매개변수에 익명 자식 객체 대입
        car.run3(new Tire() {
            @Override
            public void roll() {
                System.out.println("익명 자식 객체 Tire가 굴러갑니다.");
            }
        });
    }
}
```



익명 객체 (Anonymous Object) – 익명 구현 객체

- 인터페이스를 구현해서 생성되는 익명 구현 객체는 아래와 같이 생성된다.

```
new 인터페이스() {  
    // 필드  
    // 메소드  
};
```

- 익명 구현 객체는 인터페이스 타입의 필드, 로컬 변수, 매개변수의 값으로 대입할 수 있다.



익명 객체 (Anonymous Object) – 익명 구현 객체

- 인터페이스 타입의 필드로 대입되는 익명 구현 객체

```
package com.anonymous.implement;
```

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

```
public class Home {  
    // 필드에 익명 구현 객체 대입  
    private RemoteControl rc = new RemoteControl() {  
        @Override  
        public void turnOn() {  
            System.out.println("TV를 켭니다.");  
        }  
        @Override  
        public void turnOff() {  
            System.out.println("TV를 끕니다.");  
        }  
    };  
    // 필드 사용 메소드  
    public void use1() {  
        rc.turnOn();  
        rc.turnOff();  
    }  
}
```




익명 객체 (Anonymous Object) – 익명 구현 객체

- 로컬 변수로 대입되는 익명 구현 객체

```
public class Home {  
    .....  
    // 로컬변수 사용 메소드  
    public void use2() {  
        // 로컬 변수에 익명 구현 객체 대입  
        RemoteControl rc = new RemoteControl() {  
            @Override  
            public void turnOn() {  
                System.out.println("에어컨을 켭니다.");  
            }  
            @Override  
            public void turnOff() {  
                System.out.println("에어컨을 끕니다.");  
            }  
        };  
        rc.turnOn();  
        rc.turnOff();  
    }  
}
```



익명 객체 (Anonymous Object) – 익명 구현 객체

- 메소드의 매개변수 값으로 대입되는 익명 구현 객체

```
public class Home {  
    .....  
    // 매개변수 사용 메소드  
    public void use3(RemoteControl rc) {  
        rc.turnOn();  
        rc.turnOff();  
    }  
}
```



익명 객체 (Anonymous Object) – 익명 구현 객체

```
public class HomeExample {  
    public static void main(String[] args) {  
        Home home = new Home();  
  
        home.use1();  
  
        home.use2();  
  
        // 매개변수에 익명 구현 객체 대입  
        home.use3(new RemoteControl() {  
            @Override  
            public void turnOn() {  
                System.out.println("선풍기를 켭니다.");  
            }  
            @Override  
            public void turnOff() {  
                System.out.println("선풍기를 끕니다.");  
            }  
        });  
    }  
}
```



라이브러리(Library)와 모듈(Module)

- 라이브러리
 - 라이브러리는 프로그램 개발 시 활용할 수 있는 클래스와 인터페이스들을 모아놓은 것을 의미한다.
 - 일반적으로 JAR(Java Archive) 압축 파일 형태로 존재한다.
- 모듈
 - 모듈은 패키지 관리 기능까지 포함된 라이브러리로, Java 9 부터 지원한다.
 - 모듈도 라이브러리의 일종이므로 JAR 파일 형태로 배포 가능하다.

TIP

대규모 프로젝트에서는 모듈 시스템을 이해하고 활용하는 것이 유리하지만, 필수적인 내용은 아니기 때문에 라이브러리만 가볍게 알아볼 것이다.



라이브러리

- 특정 클래스와 인터페이스가 응용프로그램을 개발할 때 공통으로 자주 사용된다면, JAR 파일로 압축해서 라이브러리로 관리하는 것이 좋다.
- 프로그램 개발 시 라이브러리를 이용하려면 JAR 파일을 ClassPath에 추가해야 한다.

TIP

ClassPath는 말 그대로 클래스를 찾기 위한 경로를 의미한다.

- ClassPath에 라이브러리를 추가하는 방법은 다음과 같다.
 - 콘솔(명령 프롬프트, 터미널 등)에서 프로그램을 실행할 경우
 - java 명령어를 실행할 때 -classpath로 제공하거나 CLASSPATH 환경 변수에 경로를 추가
 - 이클립스 프로젝트에서 실행할 경우
 - 프로젝트의 Build Path에 추가



라이브러리

- 프로젝트 생성 (File > New > Java Project)
 - Project Name : my_lib
 - Module : [체크 안함] Create module-info.java file
- Package Explorer 뷰에서 src 폴더를 선택 후 마우스 우클릭 [New > Package로 pack1과 pack2 패키지 생성]
- A와 B 클래스 작성

```
package pack1;
```

```
public class A {  
    public void method() {  
        System.out.println("A 메소드 실행");  
    }  
}
```

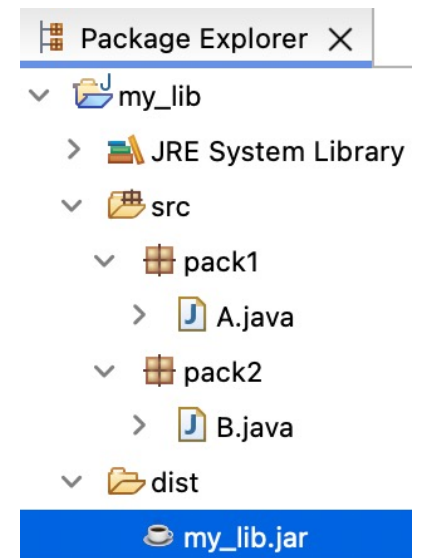
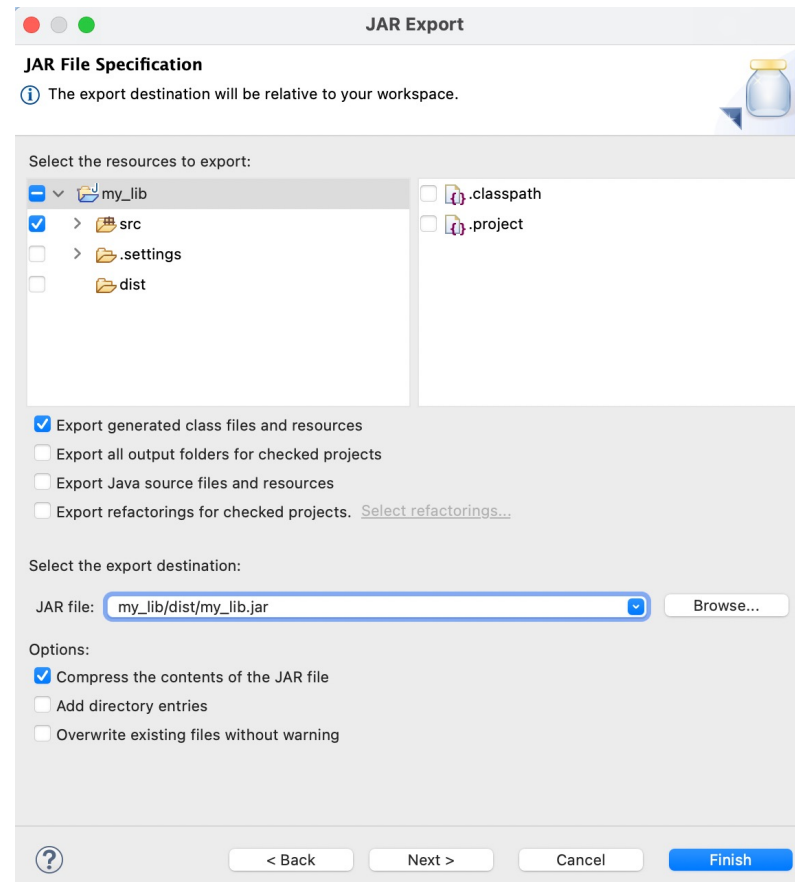
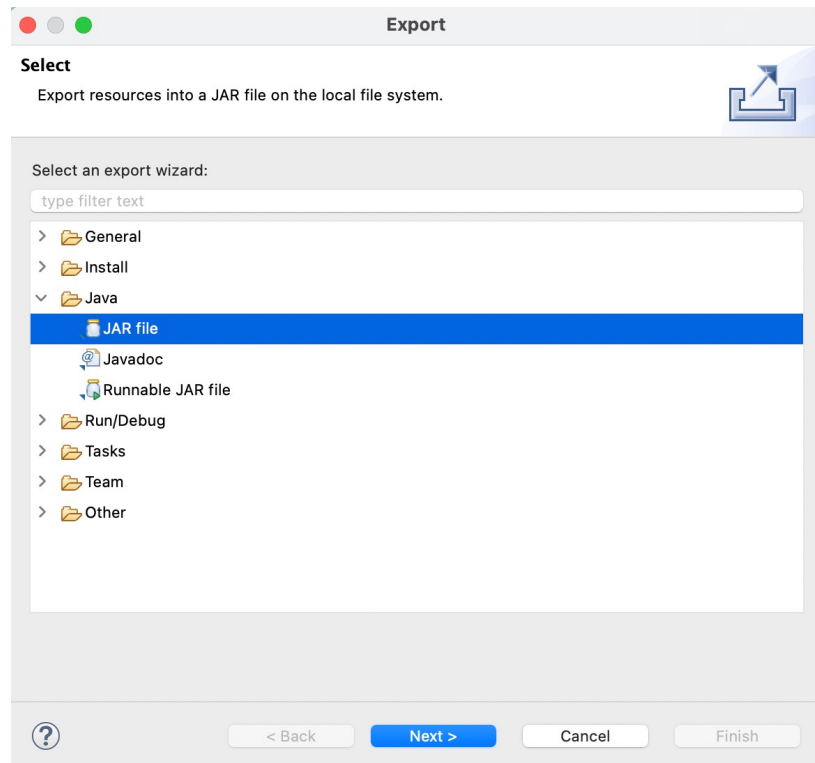
```
package pack2;
```

```
public class B {  
    public void method() {  
        System.out.println("B 메소드 실행");  
    }  
}
```



라이브러리

- my_lib 프로젝트 선택 후, 마우스 우클릭 [New > Folder로 dist 폴더 생성]
- my_lib 프로젝트 선택 후, 마우스 우클릭 [Export 선택]






라이브러리

- 프로젝트 생성 (File > New > Java Project)
 - Project Name : my_application
 - Module : [체크 안함] Create module-info.java file
- Package Explorer 뷰에서 my_application 프로젝트 선택 후, 마우스 우클릭 [Build Path > Configure Build Path]
- Libraries 탭에 JARs and class folders on the build path에서 Classpath 항목을 선택하고, Add External JARs 버튼 클릭

▼ ⬇️⬆️ Classpath

>  my_lib.jar - /Users/inkyu/Documents/java/workspace/my_lib/dist



라이브러리

- Package Explorer 뷰에서src 폴더를 선택 후 마우스 우클릭 [New > Package로 app 패키지 생성]
- Main 클래스 작성

```
package my_application;

import pack1.A;
import pack2.B;

public class Main {
    public static void main(String[] args) {
        A a = new A();
        a.method();
        B b = new B();
        b.method();
    }
}
```



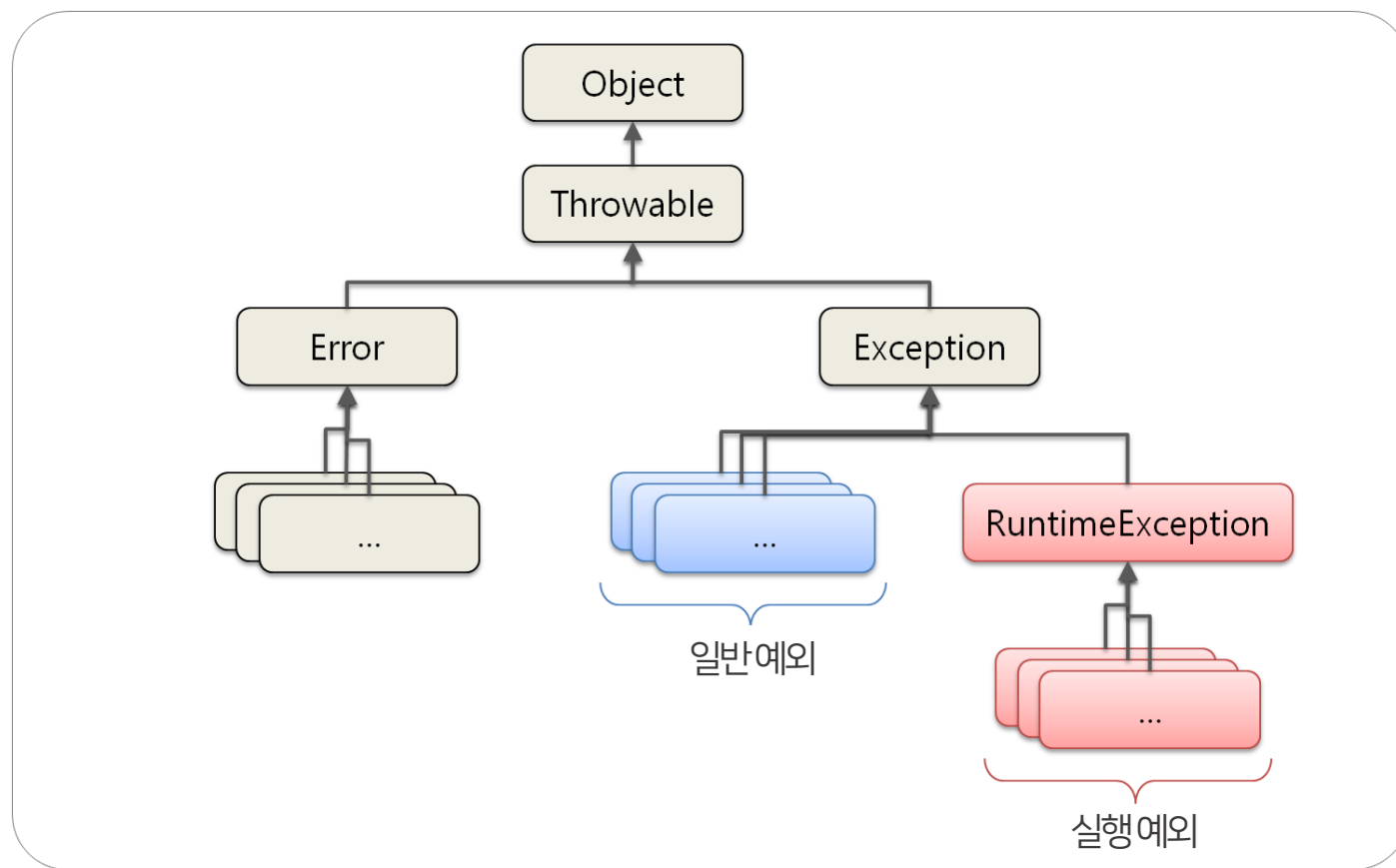
예외 처리 - 예외와 예외 클래스

- 컴퓨터 하드웨어의 고장으로 인해 프로그램 실행에 오류가 발생하는 것을 Java에서는 에러(error)라고 한다.
- 프로그램을 아무리 견고하게 잘 만들어도, 이런 에러는 대처할 방법이 없다.
- Java에서는 예외(Exception)라고 부르는 오류가 있다. 예외란 잘못된 사용 또는 코딩으로 인한 프로그램 오류를 의미한다.
- 예외가 발생하면 프로그램이 곧바로 종료된다는 점에서 에러와 동일하지만, 예외는 처리를 통해 프로그램의 실행 상태를 유지할 수 있다.
- 예외에는 두 가지가 있다.
 - 일반 예외 (Exception): 컴파일러가 예외 처리 코드 여부를 검사하는 예외
 - 실행 예외 (Runtime Exception): 컴파일러가 예외 처리 코드 여부를 검사하지 않는 예외



예외 처리 - 예외와 예외 클래스

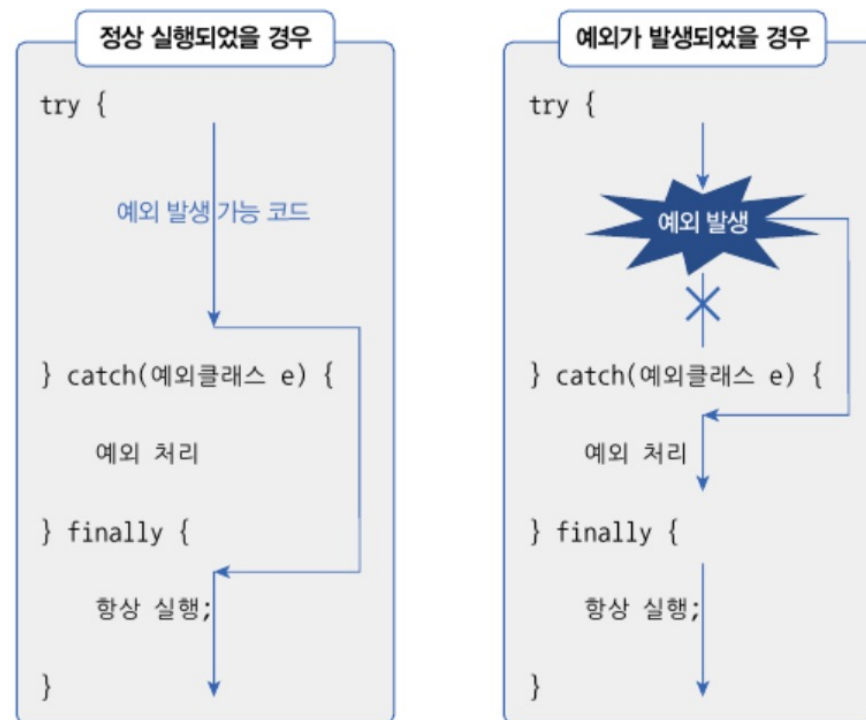
- Java는 예외가 발생하면 예외 클래스로부터 객체를 생성하고, 해당 객체는 예외 처리 시 사용된다.
- Java의 모든 에러와 예외 클래스는 Throwable을 상속받아 만들어지고, 추가적으로 예외 클래스는 java.lang.Exception 클래스를 상속 받는다.





예외 처리 - 예외 처리 코드

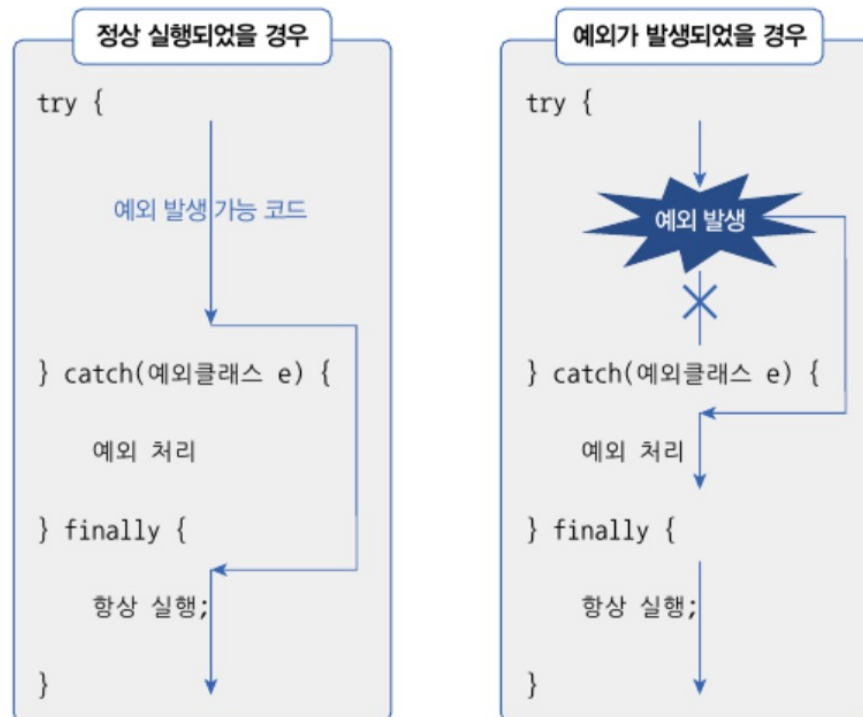
- 예외가 발생했을 때 프로그램의 갑작스러운 종료를 막고 정상 실행을 유지할 수 있도록 처리하는 코드를 예외 처리 코드라고 한다.
- 예외 처리 코드는 try - catch - finally 블록으로 구성된다.
- try - catch - finally 블록은 생성자 내부와 메소드 내부에서 작성된다.





예외 처리 - 예외 처리 코드

- try 블록에서 작성한 코드가 예외 없이 정상 실행되면, catch 블록은 실행되지 않고 finally 블록이 실행된다.
- 하지만 try 블록에서 예외가 발생하면 catch 블록이 바로 실행되고, 연이어 finally 블록이 실행된다.
- 즉, finally 블록은 예외 발생 여부와 상관없이 실행된다. 심지어 try나 catch 블록에 return이 있더라도 finally 블록은 항상 실행된다.
- finally는 옵션으로 생략이 가능하다.





예외 처리 - 예외 처리 코드

```
package com.exception;

public class ExceptionExample1 {
    public static void main(String[] args) {
        System.out.println("프로그램 시작");
        printLength("exception");
        printLength(null);
        System.out.println("프로그램 종료");
    }
    public static void printLength(String data) {
        int result = data.length();
        System.out.println("글자 수 : " + result);
    }
}
```

프로그램 시작

글자 수 : 9

Exception in thread "main" [java.lang.NullPointerException](#): Cannot invoke "String.length()" because "data" is null
at com.exception.ExceptionExample1.printLength([ExceptionExample1.java:12](#))
at com.exception.ExceptionExample1.main([ExceptionExample1.java:7](#))



예외 처리 - 예외 처리 코드

```
package com.exception;

public class ExceptionExample1 {
    ...
    public static void printLength(String data) {
        try {
            int result = data.length();
            System.out.println("글자 수 : " +
                result);
        } catch (NullPointerException e) {
            e.printStackTrace();
            // System.out.println(e.getMessage());
            // System.out.println(e.toString());
        } finally {
            System.out.println("끝");
        }
    }
}
```

프로그램 시작

글자 수 : 9

끝

[java.lang.NullPointerException](#): Can not invoke "String.length()" because "data" is null at com.exception.ExceptionExample1.printLength([ExceptionExample1.java:13](#)) at com.exception.ExceptionExample1.main([ExceptionExample1.java:7](#))

끝

프로그램 종료



예외 처리 - 예외 처리 코드

```
package com.exception;

public class ExceptionExample1 {
    ...
    public static void printLength(String data) {
        try {
            int result = data.length();
            System.out.println("글자 수 : " +
                result);
        } catch (NullPointerException e) {
            e.printStackTrace();
            // System.out.println(e.getMessage());
            // System.out.println(e.toString());
        } finally {
            System.out.println("끝");
        }
    }
}
```

프로그램 시작

글자 수 : 9

끝

[java.lang.NullPointerException](#): Can not invoke "String.length()" because "data" is null at com.exception.ExceptionExample1.printLength([ExceptionExample1.java:13](#)) at com.exception.ExceptionExample1.main([ExceptionExample1.java:7](#))

끝

프로그램 종료



예외 처리 - 예외 처리 코드

```
package com.exception;
```

```
public class ExceptionExample2 {  
    public static void main(String[] args) {  
        try {  
            String className1 = "java.lang.String";  
            Class.forName(className1);  
            System.out.println(className1 + "이 존재합니다.");  
        } catch (ClassNotFoundException e) {  
            e.printStackTrace();  
        }  
        System.out.println();  
        try {  
            String className2 = "java.lang.String2";  
            Class.forName(className2);  
            System.out.println(className2 + "이 존재합니다.");  
        } catch (ClassNotFoundException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

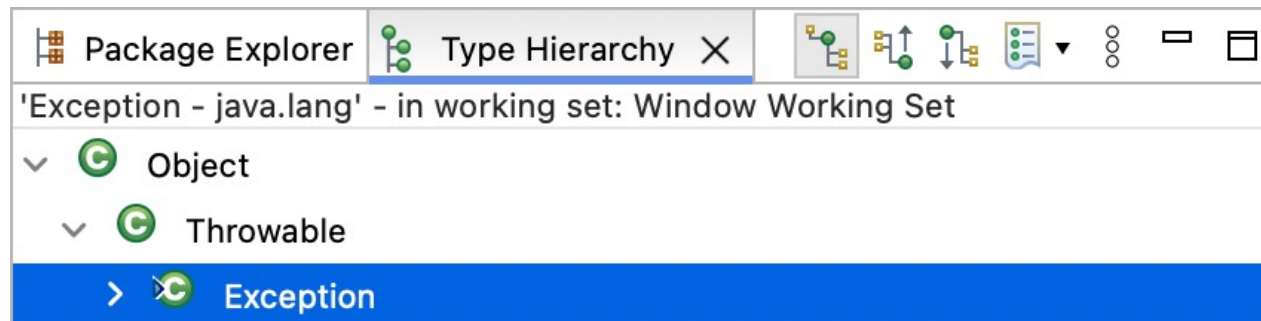
java.lang.String이 존재합니다.

java.lang.ClassNotFoundException: java.lang.String2 ...



예외 처리 - 예외 종류에 따른 처리

- try 블록에서는 다양한 종류의 예외가 발생할 수 있다. 이 경우에는 다중 catch를 이용해 발생하는 예외에 따라 처리를 다르게 할 수 있다.
- catch 블록의 예외 클래스는 try 블록에서 발생한 예외의 종류를 말하는데, 해당 타입의 예외가 발생하면 catch 블록이 선택되어 실행된다.
- catch 블록이 여러 개일지라도 catch 블록은 단 하나만 실행된다.
- 그 이유는 try 블록에서 동시에 예외가 발생하지 않으며, 하나의 예외가 발생하면 즉시 catch 블록으로 이동하기 때문이다.
- catch 블록은 예외가 발생하면 위에서부터 차례대로 검사 대상이 되기 때문에 처리해야 할 예외 클래스들이 서로 상속 관계에 있는 경우에는 하위 클래스의 catch 블록을 먼저 적고, 상위 클래스의 catch 블록을 나중에 적어야 한다.





예외 처리 - 예외 종류에 따른 처리

```
package com.exception;
```

```
public class ExceptionExample3 {  
    public static void main(String[] args) {  
        String[] strArr = {"80", "90", "100"};  
        for (int i = 0; i <= strArr.length; i++) {  
            try {  
                String str = strArr[i];  
                int value = Integer.parseInt(str);  
                System.out.println("strArr[" + i + "]: " + value);  
            } catch (ArrayIndexOutOfBoundsException e) {  
                System.out.println("배열 인덱스가 초과됨: " + e.getMessage());  
            } catch (NumberFormatException e) {  
                System.out.println("숫자로 변환할 수 없음: " + e.getMessage());  
            } catch (Exception e) {  
                System.out.println("실행에 문제가 있습니다.");  
            }  
        }  
    }  
}
```



예외 처리 - 예외 종류에 따른 처리

- 만약 두 개 이상의 예외를 하나의 catch 블록으로 동일하게 예외 처리하고 싶다면, catch 블록에 예외 클래스를 기호(|)로 연결하면 된다.

```
package com.exception;
```

```
public class ExceptionExample3 {  
    public static void main(String[] args) {  
        String[] strArr = {"80", "90", null, "100"};  
        for (int i = 0; i <= strArr.length; i++) {  
            try {  
                String str = strArr[i];  
                int value = Integer.parseInt(str);  
                System.out.println("strArr[" + i + "]: " + value);  
            } catch (ArrayIndexOutOfBoundsException e) {  
                System.out.println("배열 인덱스가 초과됨: " + e.getMessage());  
            } catch (NumberFormatException | NullPointerException e) {  
                System.out.println("데이터에 이상이 있음: " + e.getMessage());  
            } catch (Exception e) {  
                System.out.println("실행에 문제가 있습니다.");  
            }  
        }  
    }  
}
```



예외 처리 - 리소스 자동 닫기

- 리소스(resource)란 데이터를 제공하는 객체를 말한다.
- 리소스는 사용하기 위해 열고(Open), 사용을 한 뒤에는 닫아야(Close) 한다. [파일을 읽기 위해 파일을 열고, 다 읽은 후에는 닫는다]
- 리소스를 사용하다가 예외가 발생한 경우에도 안전하게 리소스를 닫아주는 것이 중요하다. 그렇지 않으면 리소스가 불안정한 상태로 남아있게 되기 때문이다.

```
FileInputStream fis = null;
try {
    fis = new FileInputStream("file.txt");
    // ...
} catch(IOException e) {
    // ...
} finally {
    fis.close();
}
```

TIP

리소스 관련 에러처리는 뒤에서 알아본다.
눈으로만 익혀둘 것!



예외 처리 - 리소스 자동 닫기

- 더 편리한 방법은 try - with -resources 블록을 사용하는 것인데, 예외 발생 여부와 상관없이 리소스를 자동으로 닫아준다.
- try - with -resources 블록을 사용하기 위해서는 AutoCloseable 인터페이스를 구현해서 close() 메소드를 재정의해야 한다.

```
try(FileInputStream fis = new FileInputStream("file.txt")) {  
    // ...  
} catch(IOException e) {  
    // ...  
}
```

- 만약 두 개 이상의 리소스를 사용해야 한다면, 세미콜론(;)으로 구분해서 리소스를 여는 코드를 작성하면 된다.

```
try(  
    FileInputStream fis1 = new FileInputStream("file1.txt");  
    FileInputStream fis2 = new FileInputStream("file2.txt")) {  
    // ...  
} catch(IOException e) {  
    // ...  
}
```

TIP

리소스 관련 예러처리는 뒤에서 알아본다.
눈으로만 익혀둘 것!



예외 처리 - 리소스 자동 닫기

- Java 9 이상부터는 외부 리소스 변수를 try 블록에 사용할 수 있다.

```
FileInputStream fis1 = new FileInputStream("file1.txt");
FileInputStream fis2 = new FileInputStream("file2.txt");

try(fis1; fis2) {
    // ...
} catch(IOException e) {
    // ...
}
```

TIP

리소스 관련 에러 처리는 뒤에서 알아본다.
눈으로만 익혀둘 것!



예외 처리 - 예외 던지기

- 메소드 내부에서 예외가 발생할 때 try - catch 블록으로 예외를 처리하는 것이 기본이지만, 메소드를 호출한 곳으로 예외를 던질 수도 있다.
- 이때 사용하는 키워드가 throws이다.
- throws는 메소드 선언부 끝에 작성하는데, 던질 예외 클래스를 쉼표로 구분해서 나열해 주면 된다.

리턴타입 메소드이름 (매개 변수타입 매개 변수명) **throws** 예외클래스 { }

- throws 키워드가 붙어 있는 메소드는 해당 예외를 처리하지 않고 던졌기 때문에 이 메소드를 호출하는 곳에서 예외를 받아 처리해야 한다.

```
package com.exception;
```

```
public class ExceptionExample4 {  
    public static void main(String[] args) {  
        try {  
            findClass();  
        } catch (ClassNotFoundException e) {  
            e.printStackTrace();  
        }  
    }  
    public static void findClass() throws ClassNotFoundException {  
        Class.forName("java.lang.String2");  
    }  
}
```




예외 처리 - 예외 떠넘기기

- 만약 예외를 떠넘기려 할 때, 나열해야 할 예외 클래스가 많은 경우에는 쉼표(,)로 연결해서 작성하거나, throws Exception이나 throws Throwable 만으로 모든 예외를 간단히 떠넘길 수 있다.
- main() 메소드에서도 예외를 떠넘길 수 있는데, 그렇게 하게 되면 결국 JVM에서 최종적으로 예외를 처리하게 된다. JVM은 예외의 내용을 콘솔에 출력하는 것으로 처리한다.

```
package com.exception;

public class ExceptionExample4 {
    public static void main(String[] args) throws Exception {
        findClass();
    }
    public static void findClass() throws ClassNotFoundException {
        Class.forName("java.lang.String2");
    }
}
```



예외 처리 - 사용자 정의 예외

- 존재하지 않는 예외를 직접 예외 클래스로 정의해서 사용하는 것을 사용자 정의 예외라고 한다.
- 사용자 정의 예외는 컴파일러가 체크하는 일반 예외로 선언할 수도 있고, 컴파일러가 체크하지 않는 실행 예외로 선언할 수도 있다.
- 통상적으로 일반 예외는 Exception의 자식 클래스로 선언하고, 실행 예외는 RuntimeException의 자식 클래스로 선언한다.

// 사용자 정의 예외 (일반예외)

```
public class MyException extends Exception {  
    public MyException() { }  
  
    public MyException(String message) {  
        super(message);  
    }  
}
```

// 사용자 정의 예외 (실행예외)

```
public class MyException extends RuntimeException {  
    public MyException() { }  
  
    public MyException(String message) {  
        super(message);  
    }  
}
```



예외 처리 - 사용자 정의 예외 [실습]

- 은행계좌(Account) 클래스의 출금(withdraw) 메소드에서 잔고(balance) 필드와 출금액(매개값)을 비교해 잔고가 부족하면 사용자 정의 예외(MyException)을 발생시키고 throws한다.
- 그리고 AccountExample 클래스의 main() 메소드에서 withdraw() 메소드를 호출할 때 예외 처리를 한다.

```
package com.bank;

import com.exception.MyException;

public class Account {
    private long balance;
    public Account() {}
    public long getBalance() {
        return balance;
    }
    public void deposit(int money) {
        balance += money;
    }
    public void withdraw(int money) throws MyException {
        if (balance < money) {
            throw new MyException("잔고 부족 : " + (money - balance) + "원이 부족합니다.");
        }
        balance -= money;
    }
}
```



예외 처리 - 사용자 정의 예외 [실습]

```
package com.bank;

import com.exception.MyException;

public class AccountExample {
    public static void main(String[] args) {
        Account acc = new Account();

        // 입금
        acc.deposit(10000);
        System.out.println("예금액 : " + acc.getBalace());

        // 출금
        try {
            acc.withdraw(30000);
        } catch (MyException e) {
            System.out.println(e.getMessage());
        }
    }
}
```