

09-1. 중첩 클래스와 중첩 인터페이스 소개

혼자 공부하는 자바 (신용권 저)

- 시작하기 전에
- 중첩 클래스
- 중첩 클래스의 접근 제한
- 중첩 인터페이스
- 키워드로 끝내는 핵심 포인트
- 확인문제



시작하기 전에

[핵심 키워드] : 중첩 클래스, 멤버 클래스, 로컬 클래스, 중첩 인터페이스

[핵심 포인트]

객체 지향 프로그래밍에서 클래스들은 서로 긴밀한 관계를 맺고 상호작용을 한다. 그 중 특정한 클래스와 관계를 맺을 경우에는 클래스 내부에 선언하는 것이 좋다. 중첩 클래스와 중첩 인터페이스에 대해 알아본다.



시작하기 전에

❖ 중첩 클래스 (nested class)

- 클래스 내부에 선언한 클래스
- 두 클래스의 멤버들을 서로 쉽게 접근하게 하고, 외부에는 불필요한 관계 클래스 감춤
- 코드 복잡성 줄임

```
class ClassName {  
    class NestedClassName {  
    }  
}
```

← 중첩 클래스

❖ 중첩 인터페이스 (nested interface)

- 인터페이스 역시 클래스 내부에 선언 가능
- 해당 클래스와 긴밀한 관계 갖는 구현 클래스 만들기 위함

```
class ClassName {  
    interface NestedInterfaceName {  
    }  
}
```

← 중첩 인터페이스



중첩 클래스

❖ 멤버 클래스

- 클래스의 멤버로서 선언되는 중첩 클래스

❖ 로컬 클래스

- 메소드 내부에서 선언되는 중첩 클래스
- 메소드 실행할 때만 사용되고 메소드 종료되면 사라짐

선언 위치에 따른 분류		선언 위치	설명
멤버 클래스	인스턴스 멤버 클래스	<pre>class A { class B { ... } }</pre>	A 객체를 생성해야만 사용할 수 있는 B 클래스
	정적 멤버 클래스	<pre>class A { static class B { ... } }</pre>	A 클래스로 바로 접근할 수 있는 B 클래스
로컬 클래스		<pre>class A { void method() { class B { ... } } }</pre>	method()가 실행할 때만 사용할 수 있는 B 클래스



중첩 클래스

❖ 중첩 클래스를 컴파일하면 바이트 코드 파일(.class)이 별도로 생성

■ 멤버 클래스 경우

A \$ B .class
↑ ↑
바깥 클래스 멤버 클래스

■ 로컬 클래스 경우

A \$1 B .class
↑ ↑
바깥 클래스 로컬 클래스



❖ 인스턴스 멤버 클래스

- static 키워드 없이 중첩 선언된 클래스
- 인스턴스 필드와 메소드만 선언 가능하고 정적 필드와 메소드는 선언할 수 없음

```
class A {  
    /**인스턴스 멤버 클래스**/  
    class B {  
        B() { } ← 생성자  
        int field1; ← 인스턴스 필드  
        //static int field2; ← 정적 필드 (x)  
        void method1() { } ← 인스턴스 메소드  
        //static void method2() { } ← 정적 메소드 (x)  
    }  
}
```



중첩 클래스

- A 클래스 외부에서 B 객체 생성하려면 먼저 A 객체 생성 후 B 객체 생성 필요

A 클래스 외부

```
A a = new A();  
A.B b = a.new B();  
b.field1 = 3;  
b.method1();
```

A 클래스 내부

```
class A {  
    class B { ... }  
  
    void methodA() {  
        B b = new B();  
        b.field = 3;  
        b.method1();  
    }  
}
```



❖ 정적 멤버 클래스

- static 키워드로 선언된 클래스
- 모든 종류의 필드와 메소드 선언 가능

```
class A {  
    /**정적 멤버 클래스**/  
    static class C {  
        C() { } ← 생성자  
        int field1; ← 인스턴스 필드  
        static int field2; ← 정적 필드  
        void method1() { } ← 인스턴스 메소드  
        static void method2() { } ← 정적 메소드  
    }  
}
```



중첩 클래스

- A 클래스 외부에서 정적 멤버 클래스 C 객체 생성할 경우 A 객체 생성 필요하지 않음

```
A.C c = new A.C();  
c.field1 = 3;    //인스턴스 필드 사용  
c.method1();     //인스턴스 메소드 호출  
A.C.field2 = 3;  //정적 필드 사용  
A.C.method2();   //정적 메소드 호출
```



❖ 로컬 클래스

- 중첩 클래스를 메소드 내에서 선언할 수 있음
- 접근 제한자 및 static 붙일 수 없음
- 인스턴스 필드와 메소드만 선언할 수 있고 정적 필드와 메소드는 선언 불가

```
void method() {  
    /**로컬 클래스**/  
    class D {  
        D() { } ← 생성자  
        int field1; ← 인스턴스 필드  
        //static int field2; ← 정적 필드 (x)  
        void method1() { } ← 인스턴스 메소드  
        //static void method2() { } ← 정적 메소드 (x)  
    }  
    D d = new D();  
    d.field1 = 3;  
    d.method1();  
}
```



❖ 예시 - 중첩 클래스

```
01 package sec01.exam01;
02
03 /**바깥 클래스**/
04 class A {
05     A() { System.out.println("A 객체가 생성됨"); }
06
07     /**인스턴스 멤버 클래스**/
08     class B {
09         B() { System.out.println("B 객체가 생성됨"); }
10         int field1;
11         //static int field2;
12         void method1() { }
13         //static void method2() { }
14     }
15
16     /**정적 멤버 클래스**/
17     static class C {
18         C() { System.out.println("C 객체가 생성됨"); }
19         int field1;
20         static int field2;
21         void method1() { }
22         static void method2() { }
23     }
```



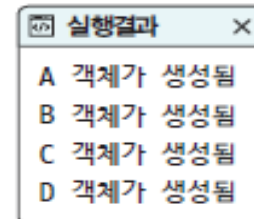
중첩 클래스

```
24
25 void method() {
26     /**로컬 클래스**/
27     class D {
28         D() { System.out.println("D 객체가 생성됨"); }
29         int field1;
30         //static int field2;
31         void method1() { }
32         //static void method2() { }
33     }
34     D d = new D();
35     d.field1 = 3;
36     d.method1();
37 }
38 }
```



❖ 예시 - 중첩 클래스의 객체 생성

```
01 package sec01.exam01;
02
03 public class Main {
04     public static void main(String[] args) {
05         A a = new A();
06
07         //인스턴스 멤버 클래스 객체 생성
08         A.B b = a.new B();
09         b.field1 = 3;
10         b.method1();
11
12         //정적 멤버 클래스 객체 생성
13         A.C c = new A.C();
14         c.field1 = 3;
15         c.method1();
16         A.C.field2 = 3;
17         A.C.method2();
18
19         //로컬 클래스 객체 생성을 위한 메소드 호출
20         a.method();
21     }
22 }
```



중첩 클래스의 접근 제한

❖ 바깥 필드와 메소드에서 사용 제한

- 바깥 클래스에서 인스턴스 멤버 클래스 사용하는 경우

```
01 package sec01.exam02;
02
03 public class A {
04     //인스턴스 필드
05     B field1 = new B(); ← (o)
06     C field2 = new C();
07
08     //인스턴스 메소드
09     void method1() {
10         B var1 = new B(); ← (o)
11         C var2 = new C();
12     }
13
14     //정적 필드 초기화
15     //static B field3 = new B(); ← (o)
16     static C field4 = new C();
```



중첩 클래스의 접근 제한

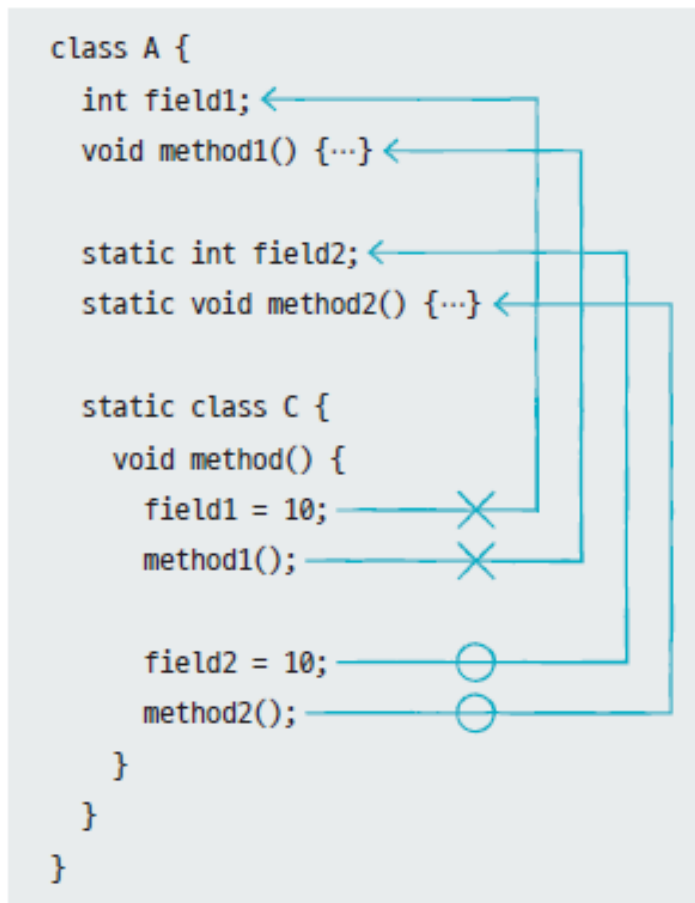
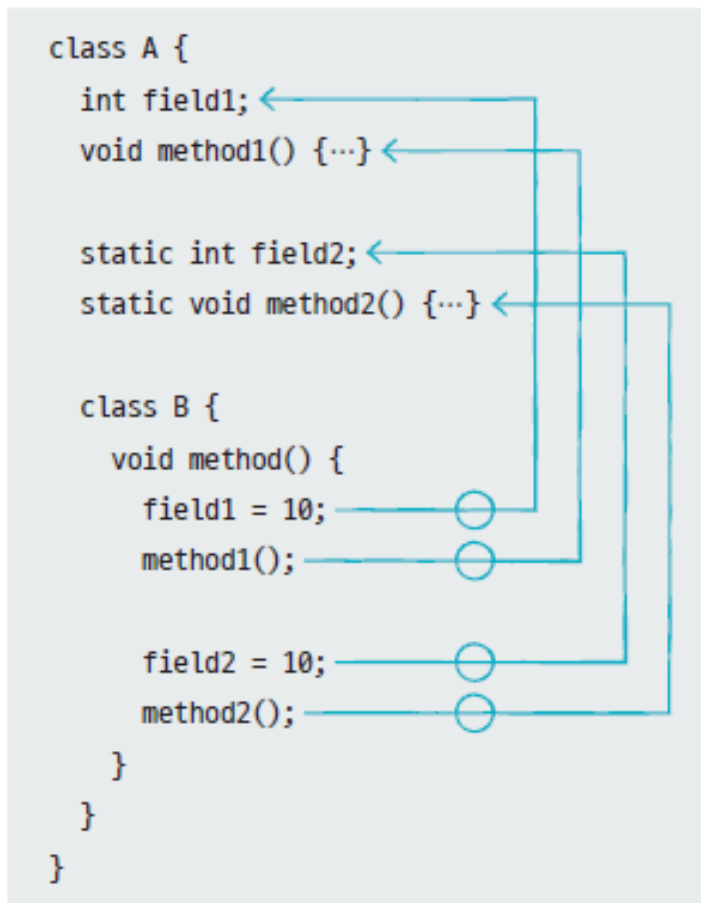
```
17
18     //정적 메소드
19     static void method2() {
20         //B var1 = new B(); ← (x)
21         C var2 = new C(); ← (o)
22     }
23
24     //인스턴스 멤버 클래스
25     class B {}
26
27     //정적 멤버 클래스
28     static class C {}
29 }
```



중첩 클래스의 접근 제한

❖ 멤버 클래스에서 사용 제한

- 멤버 클래스 내부에서 바깥 클래스의 필드와 메소드에 접근하는 경우



중첩 클래스의 접근 제한

```
01 package sec01.exam03;  
02  
03 public class A {  
04     int field1;  
05     void method1() { }  
06  
07     static int field2;  
08     static void method2() { }  
09  
10     class B {  
11         void method() {
```



중첩 클래스의 접근 제한

```
12     field1 = 10;  
13     method1();  
14  
15     field2 = 10;  
16     method2();  
17 }  
18 }  
19  
20 static class C {  
21     void method() {  
22         //field1 = 10;  
23         //method1();  
24  
25         field2 = 10;  
26         method2();  
27     }  
28 }  
29 }
```

모든 필드와 메소드에 접근할 수 있음

인스턴스 필드와 메소드는 접근할 수 없음



중첩 클래스의 접근 제한

❖ 로컬 클래스에서 사용 제한

- 메소드의 매개 변수나 로컬 변수를 로컬 클래스에서 사용할 때의 제한
- 메소드가 종료되어도 계속 실행 상태로 존재하는 로컬 스레드 객체의 경우 등
- 매개 변수나 로컬 변수를 final 키워드로 선언해야 함
- 자바 8부터는 final 선언 하지 않아도 final 특성 부여되어 있음

```
01 package sec01.exam04;
02
03 public class Outer {
04     //자바 7 이전
05     public void method1(final int arg) {
06         final int localVariable = 1;
07         //arg = 100;
08         //localVariable = 100;
09         class Inner {
10             public void method() {
11                 int result = arg + localVariable;
```

← (x)

중첩 클래스의 접근 제한

```
12     }  
13 }  
14 }  
15  
16 //자바 8 이후  
17 public void method2(int arg) {  
18     int localVariable = 1;  
19     //arg = 100;  
20     //localVariable = 100; ← (X)  
21     class Inner {  
22         public void method() {  
23             int result = arg + localVariable;  
24         }  
25     }  
26 }  
27 }
```



중첩 클래스의 접근 제한

❖ 중첩 클래스에서 바깥 클래스 참조 얻기

- 바깥 클래스의 이름을 this 앞에 붙임

바깥클래스.this.필드

바깥클래스.this.메소드();

```
01 package sec01.exam05;
02
03 public class Outer {
04     String field = "Outer-field";
05     void method() {
06         System.out.println("Outer-method");
07     }
08 }
```



중첩 클래스의 접근 제한

```
09  class Nested {  
10      String field = "Nested-field";  
11      void method() {  
12          System.out.println("Nested-method");  
13      }  
14      void print() {  
15          System.out.println(this.field);  
16          this.method();  
17          System.out.println(Outer.this.field);  
18          Outer.this.method();  
19      }  
20  }  
21  }
```

← 중첩 객체 참조

← 바깥 객체 참조



중첩 클래스의 접근 제한

```
01 package sec01.exam05;
02
03 public class OuterExample {
04     public static void main(String[] args) {
05         Outer outer = new Outer();
06         Outer.Nested nested = outer.new Nested();
07         nested.print();
08     }
09 }
```

실행결과

- Nested-field
- Nested-method
- Outer-field
- Outer-method



중첩 인터페이스

❖ 중첩 인터페이스

- 클래스의 멤버로 선언된 인터페이스
- 해당 클래스와 긴밀한 관계 맺는 구현 클래스 만들기 위함

```
class A {  
    [static] interface I {  
        void method();  
    }  
}
```

← 중첩 인터페이스

- 인스턴스 멤버 인터페이스와 정적 멤버 인터페이스 모두 가능함



❖ 예시 - 중첩 인터페이스

```
01 package sec01.exam06;
02
03 public class Button {
04     OnClickListener listener; ← 인터페이스 타입 필드
05
06     void setOnClickListener(OnClickListener listener) {
07         this.listener = listener; ← 매개 변수의 다형성
08     }
09
10     void touch() {
11         listener.onClick(); ← 구현 객체의 onClick() 메소드 호출
12     }
13
14     static interface OnClickListener {
15         void onClick(); ← 중첩 인터페이스
16     }
17 }
```

❖ 예시 - 구현 클래스

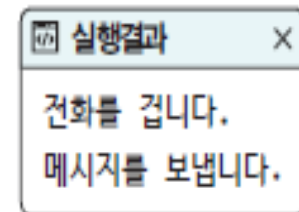
```
01 package sec01.exam06;
02
03 public class CallListener implements Button.OnClickListener {
04     @Override
05     public void onClick() {
06         System.out.println("전화를 겁니다.");
07     }
08 }
```

```
01 package sec01.exam06;
02
03 public class MessageListener implements Button.OnClickListener {
04     @Override
05     public void onClick() {
06         System.out.println("메시지를 보냅니다.");
07     }
08 }
```



❖ 버튼 이벤트 처리

```
01 package sec01.exam06;
02
03 public class ButtonExample {
04     public static void main(String[] args) {
05         Button btn = new Button();
06
07         btn.setOnClickListener( new CallListener() );
08         btn.touch();
09
10         btn.setOnClickListener( new MessageListener() );
11         btn.touch();
12     }
13 }
```



키워드로 끝내는 핵심 포인트

- **중첩 클래스**: 클래스 내부에 선언한 클래스. 두 클래스의 멤버들을 서로 쉽게 접근하게 하고 외부에는 불필요한 관계 클래스 감추어 코드의 복잡성 줄임
- **멤버 클래스**: 클래스의 멤버로서 선언되는 중첩 클래스. 멤버 클래스는 바깥 객체의 필요 여부에 따라 인스턴스 멤버 클래스와 정적 멤버 클래스로 구분
- **로컬 클래스**: 생성자 또는 메소드 블록 내부에 선언된 중첩 클래스
- **중첩 인터페이스**: 클래스의 멤버로 선언된 인터페이스. 인스턴스 멤버 인터페이스와 정적 멤버 인터페이스 모두 가능함. 주로 정적 멤버 인터페이스를 UI 프로그래밍에서 이벤트 처리 목적으로 자주 활용함.



확인문제

- ❖ 중첩 멤버 클래스에 대한 설명으로 맞는 것에 O, 틀린 것에 X 하세요
 - 인스턴스 멤버 클래스는 바깥 클래스의 객체가 있어야 사용될 수 있다 ()
 - 정적 멤버 클래스는 바깥 클래스의 객체가 없어도 사용될 수 있다 ()
 - 인스턴스 멤버 클래스 내부에는 바깥 클래스의 필드와 메소드를 사용할 수 있다 ()
 - 정적 멤버 클래스 내부에는 바깥 클래스의 인스턴스 필드를 사용할 수 있다 ()
- ❖ 다음과 같이 Car 클래스 내부에 Tire와 Engine이 멤버 클래스로 선언되어 있습니다. 바깥 클래스에서 멤버 클래스의 객체를 생성하는 코드를 빈 칸에 작성해 보세요.

소스 코드 Car.java

```
01 package sec01.verify.exam03;  
02  
03 public class Car {  
04     class Tire { }  
05     static class Engine { }  
06 }
```



소스 코드 NestedClassExample.java

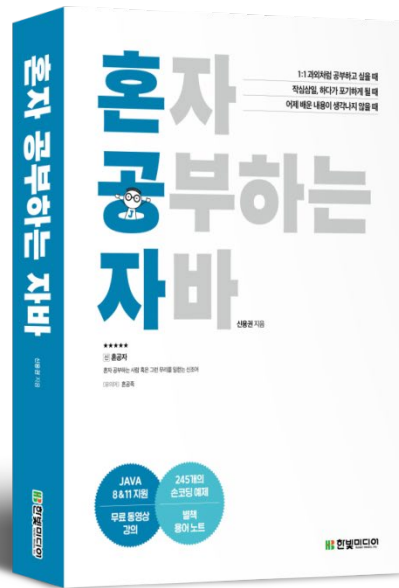
```
01 package sec01.verify.exam03;
02
03 public class NestedClassExample {
04     public static void main(String[] args) {
05         Car myCar = new Car();
06
07         Car.Tire tire = 
08
09         Car.Engine engine = 
10     }
11 }
```



❖ 다음 Chatting 클래스에서 컴파일 에러가 발생하는 이유는 무엇입니까?

소스 코드 Chatting.java

```
01 package sec01.verify.exam04;
02
03 public class Chatting {
04     void startChat(String chatId) {
05         String nickName = null;
06         nickName = chatId;
07
08         class Chat {
09             public void start() {
10                 while (true) {
11                     String inputData = "안녕하세요";
12                     String message = "[" + nickName + "]" + inputData;
13                     sendMessage(message);
14                 }
15             }
16
17             void sendMessage(String message) {
18             }
19         }
20
21         Chat chat = new Chat();
22         chat.start();
23     }
24 }
```

09-2. 익명 객체

혼자 공부하는 자바 (신용권 저)

- 시작하기 전에
- 익명 자식 객체 생성
- 익명 구현 객체 생성
- 익명 객체의 로컬 변수 사용
- 키워드로 끝내는 핵심 포인트
- 확인문제



시작하기 전에

[핵심 키워드] : 익명 자식 객체, 익명 구현 객체

[핵심 포인트]

클래스 선언 시 일반적으로 클래스 이름과 동일한 소스 파일 생성하고 클래스를 선언한다. 그런데 클래스 이름이 없는 객체가 있고, 이를 익명 객체라고 한다. 익명 객체에 대해 알아본다.



시작하기 전에

❖ 익명 (anonymous) 객체

- 이름이 없는 객체
- 어떤 클래스를 상속하거나 인터페이스를 구현하여야 함

[상속]

```
class 클래스이름1 extends 부모클래스 { ... }  
부모클래스 변수 = new 클래스이름1();
```

[구현]

```
class 클래스이름2 implements 인터페이스 { ... }  
인터페이스 변수 = new 클래스이름2();
```

[상속]

```
부모클래스 변수 = new 부모클래스() { ... };
```

[구현]

```
인터페이스 변수 = new 인터페이스() { ... };
```



익명 자식 객체 생성

❖ 익명 자식 객체 생성

- 일반적인 경우 부모 타입의 필드나 변수 선언하고 자식 객체를 초기값으로 대입하는 경우
부모 클래스 상속하여 자식 클래스 선언
new 연산자 이용하여 자식 객체 생성 후 부모 타입의 필드나 변수에 대입

```
class Child extends Parent { } ← 자식 클래스 선언

class A {
    Parent field = new Child(); ← 필드에 자식 객체를 대입
    void method() {
        Parent localVar = new Child(); ← 로컬 변수에 자식 객체를 대입
    }
}
```



익명 자식 객체 생성

- 자식 클래스를 재사용하지 않고 특정 위치에서만 사용하려는 경우
익명 자식 객체 생성하여 사용

```
부모클래스 [필드|변수] = new 부모클래스(매개값, ...) {  
    //필드  
    //메소드  
};
```

필드 선언할 때 초기값으로 익명 자식 객체 생성하여 대입

```
class A {  
    Parent field = new Parent() {  
        int childField;  
        void childMethod() { }  
        @Override  
        void parentMethod() { }  
    };  
}
```

A 클래스의 필드 선언

Parent의 메소드를 재정의



익명 자식 객체 생성

메소드 내에서 로컬 변수 선언 시 초기값으로 익명 자식 객체 생성하여 대입

```
class A {  
    void method() {  
        Parent localVar = new Parent() {  
            int childField;  
            void childMethod() { }  
            @Override  
            void parentMethod() { }  
        };  
    }  
}
```

로컬 변수 선언

Parent의 메소드를 재정의



익명 자식 객체 생성

메소드 매개 변수가 부모 타입일 경우 메소드 호출하는 코드에서 익명 자식 객체 생성하여 매개값으로 대입

```
class A {  
    void method1(Parent parent) { }
```

```
    void method2() {
```

```
        method1(  
            new Parent() {  
                int childField;  
                void childMethod() { }                @Override  
                void parentMethod() { }            }  
        );  
    }  
}
```

method1() 메소드 호출

method1()의 매개값으로
익명 자식 객체를 대입



익명 자식 객체 생성

- 익명 자식 객체에 새롭게 정의된 필드 및 메소드는 익명 자식 객체 내부에서만 사용되고 외부에서는 접근할 수 없음

```
class A {  
    Parent field = new Parent() {  
        int childField; ←  
        void childMethod() { } ←  
        @Override  
        void parentMethod() { ←  
            childField = 3;  
            childMethod();  
        }  
    };  
  
    void method() {  
        field.childField = 3; ←  
        field.childMethod(); ←  
        field.parentMethod(); ←  
    }  
}
```

익명 자식 객체 생성

❖ 예시 - 자식 객체 생성

```
01 package sec02.exam01;
02
03 public class Anonymous {
04     //필드 초기값으로 대입
05     Person field = new Person() {
06         void work() {
07             System.out.println("출근합니다.");
08         }
09         @Override
10         void wake() {
11             System.out.println("6시에 일어납니다.")
12             work();
13         }
14     };
15
16     void method1() {
```

```
17     //로컬 변수값으로 대입
18     Person localVar = new Person() {
19         void walk() {
20             System.out.println("산책합니다.");
21         }
22         @Override
23         void wake() {
24             System.out.println("7시에 일어납니다.");
25             walk();
26         }
27     };
28     //로컬 변수 사용
29     localVar.wake();
30 }
31
32 void method2(Person person) {
33     person.wake();
34 }
35 }
```

부모 클래스

```
01 package sec02.exam01;
02
03 public class Person {
04     void wake() {
05         System.out.println("7시에 일어납니다.");
06     }
07 }
```

익명 자식 객체 생성

```
01 package sec02.exam01;
02
03 public class AnonymousExample {
04     public static void main(String[] args) {
05         Anonymous anony = new Anonymous();
06         //익명 객체 필드 사용
07         anony.field.wake();
08         //익명 객체 로컬 변수 사용
09         anony.method1();
10         //익명 객체 매개값 사용
11         anony.method2(
12             new Person() {
13                 void study() {
14                     System.out.println("공부합니다.");
15                 }
16                 @Override
17                 void wake() {
18                     System.out.println("8시에 일어납니다.");
19                     study();
20                 }
21             }
22         );
23     }
24 }
```

← 매개값으로 익명 객체 대입

실행결과

6시에 일어납니다.
출근합니다.
7시에 일어납니다.
산책합니다.
8시에 일어납니다.
공부합니다.

익명 구현 객체 생성

- ❖ 인터페이스 타입의 필드 혹은 변수 선언 후 구현 객체를 초기값으로 대입하는 경우

```
class TV implements RemoteControl { }

class A {
    RemoteControl field = new TV(); ← 필드에 구현 객체를 대입
    void method() {
        RemoteControl localVar = new TV(); ← 로컬 변수에 구현 객체를 대입
    }
}
```

- 구현 클래스가 재사용되지 않고 특정 위치에서만 사용되는 경우 - 익명 구현 객체 생성

```
인터페이스 [필드|변수] = new 인터페이스() {
    //인터페이스에 선언된 추상 메소드의 실제 메소드 선언
    //필드
    //메소드
};
```

익명 구현 객체 생성

- 필드 선언 시 초기값으로 익명 구현 객체 생성하여 대입하는 경우

```
class A {  
    RemoteControl field = new RemoteControl() {  
        @Override  
        void turnOn() { }  
    };  
}
```

클래스 A의 필드 선언

RemoteControl 인터페이스의
추상 메소드에 대한 실제 메소드

- 메소드 내에서 로컬 변수 선언 시 초기값으로 익명 구현 객체 생성하여 대입

```
void method() {  
    RemoteControl localVar = new RemoteControl() {  
        @Override  
        void turnOn() { }  
    };  
}
```

로컬 변수 선언

RemoteControl 인터페이스의
추상 메소드에 대한 실제 메소드



익명 구현 객체 생성

- 메소드의 매개 변수가 인터페이스 타입일 때 메소드 호출하는 코드에서 익명 구현 객체 생성하여 매개값으로 대입하는 경우

```
class A {  
    void method1(RemoteControl rc) { }  
  
    void method2() {  
        method1(  
            new RemoteControl() {  
                @Override  
                void turnOn() { }  
            }  
        );  
    }  
}
```

method1() 메소드 호출

method1()의 매개값으로
익명 구현 객체를 대입



익명 구현 객체 생성

```
01 package sec02.exam02;
02
03 public interface RemoteControl {
04     public void turnOn();
05     public void turnOff();
06 } ❖ 예시 - 인터페이스
```

```
01 package sec02.exam02;
02
03 public class AnonymousExample {
04     public static void main(String[] args) {
05         Anonymous anony = new Anonymous();
06         //익명 객체 필드 사용
07         anony.field.turnOn();
08         //익명 객체 로컬 변수 사용
09         anony.method1();
10         //익명 객체 매개값 사용
11         anony.method2(
12             new RemoteControl() {
13                 @Override
14                 public void turnOn() {
15                     System.out.println("SmartTV를 켭니다.");
16                 }
17                 @Override
18                 public void turnOff() {
19                     System.out.println("SmartTV를 끕니다.");
20                 }
21             }
22         );
23     }
24 }
```

실행결과

```
TV를 켭니다.
Audio를 켭니다.
SmartTV를 켭니다.
```

```
01 package sec02.exam02;
02
03 public class Anonymous {
04     //필드 초기값으로 대입
05     RemoteControl field = new RemoteControl() {
06         @Override
07         public void turnOn() {
08             System.out.println("TV를 켭니다.");
09         }
10         @Override
11         public void turnOff() {
12             System.out.println("TV를 끕니다.");
13         }
14     };
15 }
```

```
16 void method1() {
17     //로컬 변수값으로 대입
18     RemoteControl localVar = new RemoteControl() {
19         @Override
20         public void turnOn() {
21             System.out.println("Audio를 켭니다.");
22         }
23         @Override
24         public void turnOff() {
25             System.out.println("Audio를 끕니다.");
26         }
27     };
28     //로컬 변수 사용
29     localVar.turnOn();
30 }
31
32 void method2(RemoteControl rc) {
33     rc.turnOn();
34 }
35 }
```

❖ 예시 - 익명 구현 객체 생성

익명 구현 객체 생성

❖ 예시 - UI 클래스

```
01 package sec02.exam03;
02
03 public class Button {
04     OnClickListener listener; ← 인터페이스 타입 필드
05
06     void setOnClickListener(OnClickListener listener) {
07         this.listener = listener;
08     }
09
10     void touch() {
11         listener.onClick(); ← 구현 객체의 onClick() 메소드 호출
12     }
13
14     static interface OnClickListener {
15         void onClick(); ← 추상 인터페이스
16     }
17 }
```

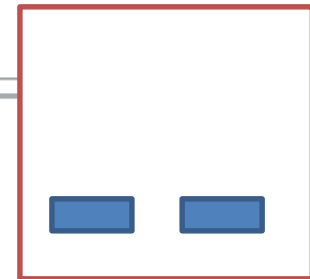
```
01 package sec02.exam03;
02
03 public class Main {
04     public static void main(String[] args) {
05         Window w = new Window();
06         w.button1.touch(); ← 버튼 클릭
07         w.button2.touch();
08     }
09 }
```

```
01 package sec02.exam03;
02
03 public class Window {
04     Button button1 = new Button();
05     Button button2 = new Button();
06
07     //필드 초기값으로 대입
08     Button.OnClickListener listener = new Button.OnClickListener() {
09         @Override
10         public void onClick() {
11             System.out.println("전화를 겁니다.");
12         }
13     };
14
15     Window() {
16         button1.setOnClickListener( listener ); ← 매개값
17         button2.setOnClickListener(new Button.OnClickListener() {
18             @Override
19             public void onClick() {
20                 System.out.println("메시지를 보냅니다.");
21             }
22         }); ← Window 클래스를 2개의 Button 객체 가진 창이라 가정
23     }
24 }
```

첫 번째 button1 클릭 이벤트 처리는 필드로 선언한 익명 구현 객체가 담당
두 번째 button2 클릭 이벤트 처리는 setOnClickListener() 호출할 때 매개값으로 준 익명 구현 객체

실행결과

전화를 겁니다.
메시지를 보냅니다.



익명 객체의 로컬 변수 사용

- ❖ 메소드의 매개 변수나 로컬 변수를 익명 객체 내부에서 사용할 때의 제한
 - 메소드가 종료되어도 익명 객체가 계속 실행 상태로 존재할 수 있음
 - 메소드의 매개 및 로컬 변수를 익명 객체 내부에서 사용할 경우에는 지속 사용 불가
 - 컴파일 시 익명 객체에서 사용하는 매개 변수나 로컬 변수의 값을 익명 객체 내부에 복사해두고 사용
매개 및 로컬 변수가 수정되어 값 변경되면 매개 및 로컬 변수를 final로 선언할 것을 요구



익명 객체의 로컬 변수 사용

```
01 package sec02.exam04;
02
03 public class Anonymous {
04     private int field;
05
06     public void method(final int arg1, int arg2) {
07         final int var1 = 0;
08         int var2 = 0;
09
10         field = 10;
11
12         //arg1 = 20;
13         //arg2 = 20;
14
15         //var1 = 30;
16         //var2 = 30;
17
18         Calculatable calc = new Calculatable() {
19             @Override
20             public int sum() {
21                 int result = field + arg1 + arg2 + var1 + var2;
22                 return result;
23             }
24         };
25
26         System.out.println(calc.sum());
27     }
28 }
```

주요 0x10

class

❖ 예시 - 익명 객체의 로컬 변수 사용

```
01 package sec02.exam04;
02
03 public interface Calculatable {
04     public int sum();
05 }
```

❖ 예시 - 인터페이스

❖ 예시 - 익명 객체의 로컬 변수 사용

```
01 package sec02.exam04;
02
03 public class AnonymousExample {
04     public static void main(String[] args) {
05         Anonymous anony = new Anonymous();
06         anony.method(0, 0);
07     }
08 }
```



키워드로 끝내는 핵심 포인트

- **익명 자식 객체**: 자식 클래스가 재사용되지 않고 오로지 특정 위치에서 사용되는 경우라면 익명 자식 객체 생성하여 사용하는 것이 편리함

```
부모클래스 [필드|변수] = new 부모클래스(매개값, ...) {  
    //필드  
    //메소드  
};
```

- **익명 구현 객체**: 구현 객체 클래스가 재사용되지 않고 오로지 특정 위치에서 사용되는 경우라면 익명 구현 객체 생성하여 사용하는 것이 편리함

```
인터페이스 [필드|변수] = new 인터페이스() {  
    //인터페이스에 선언된 추상 메소드의 실제 메소드 선언  
    //필드  
    //메소드  
};
```



확인문제

- ❖ AnonymousExample 클래스의 실행결과를 보고 Worker 클래스의 익명 자식 객체를 이 용해서 필드, 로컬 변수의 초기값과 메소드의 매개값을 대입해보세요

인터페이스

소스 코드 Vehicle.java

```
01 package sec02.verify.exam01;
02
03 public class Worker {
04     public void start() {
05         System.out.println("쉬고 있습니다.");
06     }
07 }
```



익명 구현 클래스와 객체 생성

소스 코드 Anonymous.java

```
01 package sec02.verify.exam01;
02
03 public class Anonymous {
04     Worker field = 
05
06
07     void method1() {
08         Worker localVar = 
09
10
11         localVar.start();
12     }
13
14     void method2(Worker worker) {
15         worker.start();
16     }
17 }
```



익명 구현 클래스와 객체 생성

소스 코드 AnonymousExample.java

```
01 package sec02.verify.exam01;
02
03 public class AnonymousExample {
04     public static void main(String[] args) {
05         Anonymous anony = new Anonymous();
06         anony.field.start();
07         anony.method1();
08         anony.method2(
09             
10
11     );
12 }
13 }
```

실행결과

디자인을 합니다.
개발을 합니다.
테스트를 합니다.



- ❖ CheckBox 클래스 내용을 보면 중첩 인터페이스 타입으로 필드를 선언하고 Setter 메소드로 외부에서 구현 객체를 받아 필드에 대입합니다. 선택 이벤트가 발생했을 때 인터페이스를 통해 구현 객체의 메소드를 호출합니다.

```
package sec02.verify.exam03;

public class CheckBox {
    OnSelectListener listener;

    void setOnSelectListener(OnSelectListener listener) {
        this.listener = listener;
    }

    void select() {
        listener.onSelect();
    }

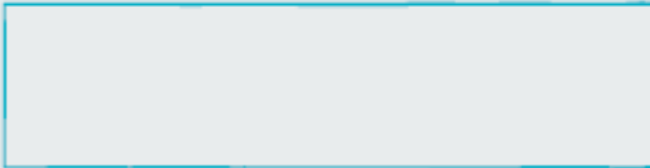
    static interface OnSelectListener {
        void onSelect();
    }
}
```

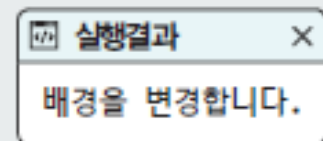


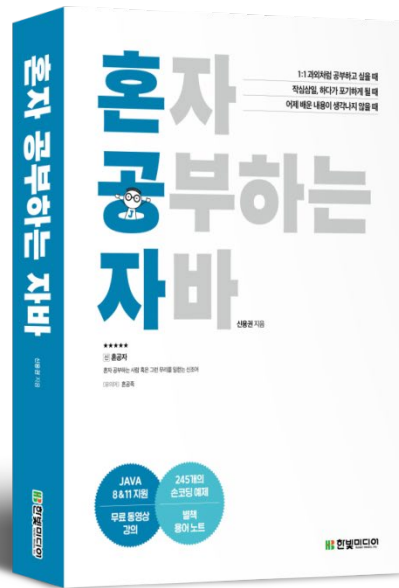
확인문제

- 다음 CheckBoxExample 클래스를 실행했을 때 다음과 같은 실행결과가 출력되도록 익명 구현 객체를 작성해보세요

```
package sec02.verify.exam03;

public class CheckBoxExample {
    public static void main(String[] args) {
        CheckBox checkBox = new CheckBox();
        checkBox.setOnSelectListener(
            
        );
        checkBox.select();
    }
}
```





10-1. 예외 클래스

혼자 공부하는 자바 (신용권 저)

- 시작하기 전에
- 예외와 예외 클래스
- 실행 예외
- 키워드로 끝내는 핵심 포인트
- 확인문제



시작하기 전에

[핵심 키워드] : 예외, 예외 클래스, 일반 예외, 실행 예외

[핵심 포인트]

자바에서 컴퓨터 하드웨어 관련 고장으로 인해 응용프로그램 실행 오류가 발생하는 것을 에러라 하고, 그 외 프로그램 자체에서 발생하는 오류를 예외라고 한다. 예외의 종류와 발생 경우를 알아본다.



시작하기 전에

❖ 예외 (Exception)

- 사용자의 잘못된 조작 또는 개발자의 잘못된 코딩으로 인해 발생하는 프로그램 오류
- 예외 처리 프로그램 통해 정상 실행상태 유지 가능
- 예외 발생 가능성이 높은 코드 컴파일할 때 예외 처리 유무 확인

이클립스 : 코드 편집 뷰의 toString 클래스 선택 후 F1 키 클릭 - Help 뷰로 이동

<https://docs.oracle.com/en/java/javase/index.html>



예외와 예외 클래스

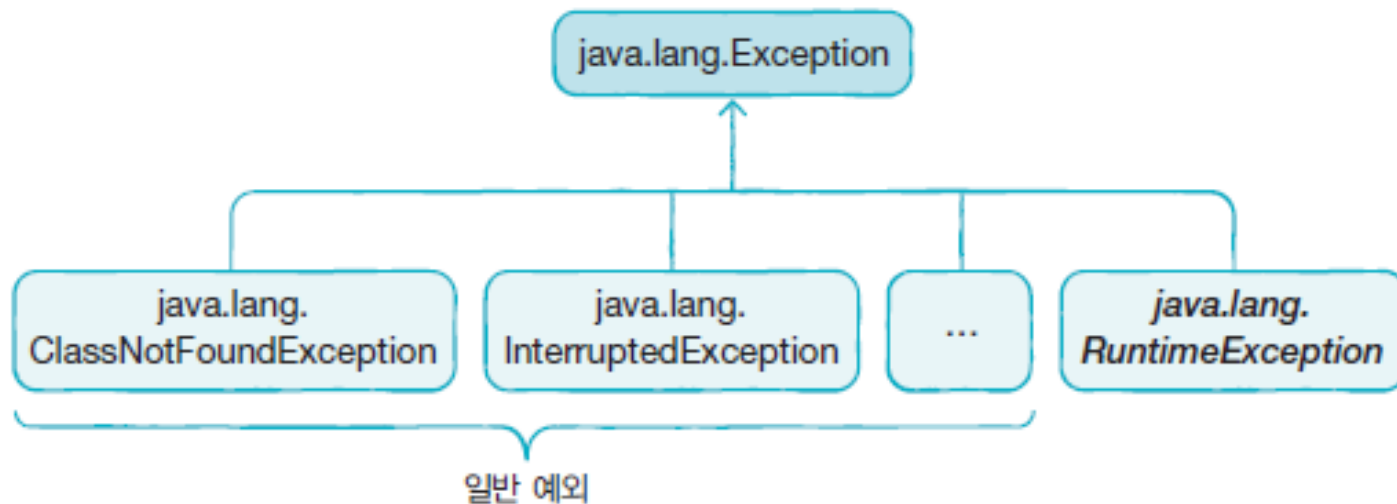
❖ 일반 예외 (exception)

- 컴파일러 체크 예외
- 자바 소스 컴파일 과정에서 해당 예외 처리 코드 있는지 검사하게 됨

❖ 실행 예외 (runtime exception)

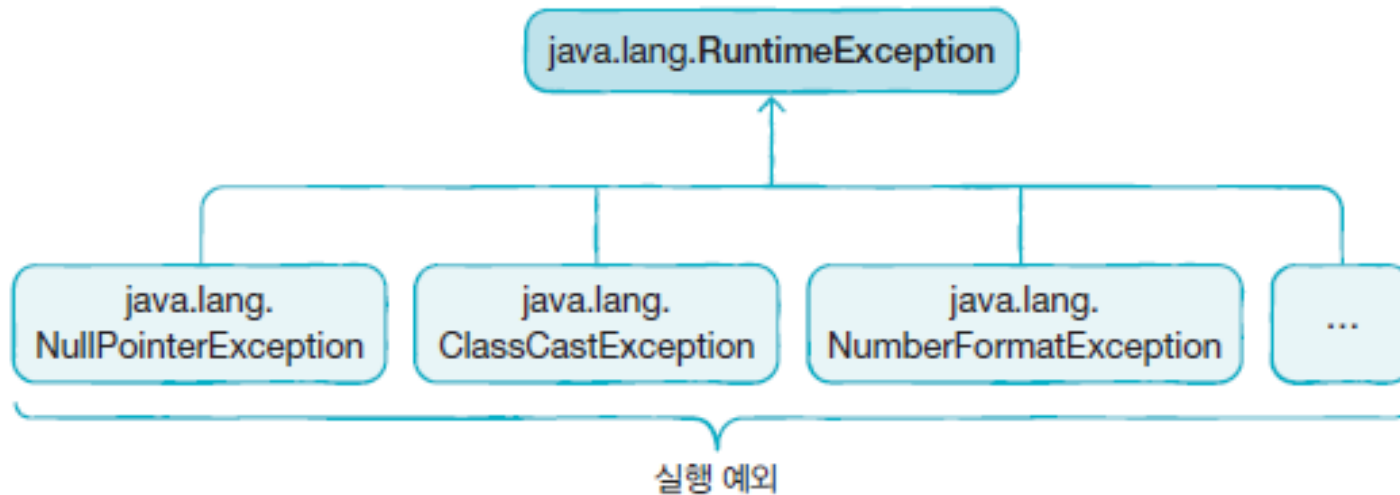
- 컴파일러 런 체크 예외
- 실행 시 예측할 수 없이 갑자기 발생하기에 컴파일 과정에서 예외처리코드 검사하지 않음

❖ 자바에서는 예외를 클래스로 관리



예외와 예외 클래스

- RuntimeException 클래스 기준으로 일반 및 실행 예외 클래스 구분



❖ 개발자의 경험에 의해서 예외 처리 코드 작성해야 함

- 예외처리코드 없을 경우 해당 예외 발생 시 프로그램 종료

❖ NullPointerException

- 가장 빈번하게 발생하는 실행 예외
- `java.lang.NullPointerException`
- 객체 참조가 없는 상태의 참조 변수로 객체 접근 연산자 도트를 사용할 경우 발생

```
01 package sec01.exam01;
02
03 public class NullPointerExceptionExample {
04     public static void main(String[] args) {
05         String data = null;
06         System.out.println(data.toString());
07     }
08 }
```

실행결과

```
Exception in thread "main" java.lang.NullPointerException
    at NullPointerExceptionExample.main(NullPointerExceptionExample.java:6)
```



❖ ArrayIndexOutOfBoundsException

- 배열에서 인덱스 범위를 초과할 경우
- `java.lang.ArrayIndexOutOfBoundsException`

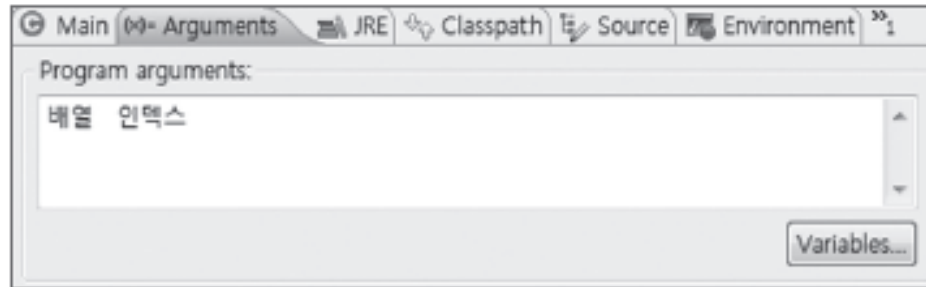
```
01 package sec01.exam02;  
02  
03 public class ArrayIndexOutOfBoundsExceptionExample {  
04     public static void main(String[] args) {  
05         String data1 = args[0];  
06         String data2 = args[1];  
07  
08         System.out.println("args[0]: " + data1);  
09         System.out.println("args[1]: " + data2);  
10     }  
11 }
```

실행결과

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 0 out of bounds  
for length 0  
    at ArrayIndexOutOfBoundsExceptionExample.main(ArrayIndexOutOfBoundsExceptionExample.  
    java:5)
```


실행 예외

- 이클립스 - [Run] - [Run Configuration] - [Arguments]탭 - [Program arguments]
아래와 같이 입력하여 해결



```
01 package sec01.exam03;
02
03 public class ArrayIndexOutOfBoundsExceptionExample {
04     public static void main(String[] args) {
05         if(args.length == 2) {
06             String data1 = args[0];
07             String data2 = args[1];
08             System.out.println("args[0]: " + data1);
09             System.out.println("args[1]: " + data2);
10         } else {
11             System.out.println("두 개의 실행 매개값이 필요합니다.");
12         }
13     }
14 }
15 }
16 }
```



❖ NumberFormatException

- 문자열을 숫자로 변환하는 경우

리턴 타입	메소드 이름(매개 변수)	설명
int	<code>Integer.parseInt(String s)</code>	주어진 문자열을 정수로 변환해서 리턴
double	<code>Double.parseDouble(String s)</code>	주어진 문자열을 실수로 변환해서 리턴

- 숫자가 변환될 수 없는 문자가 포함된 경우 `java.lang.NumberFormatException` 발생



실행 예외

```
01 package sec01.exam04;
02
03 public class NumberFormatExceptionExample {
04     public static void main(String[] args) {
05         String data1 = "100";
06         String data2 = "a100";
07
08         int value1 = Integer.parseInt(data1);
09         int value2 = Integer.parseInt(data2);    //NumberFormatException 발생
10
11         int result = value1 + value2;
12         System.out.println(data1 + "+" + data2 + "=" + result);
13     }
14 }
```

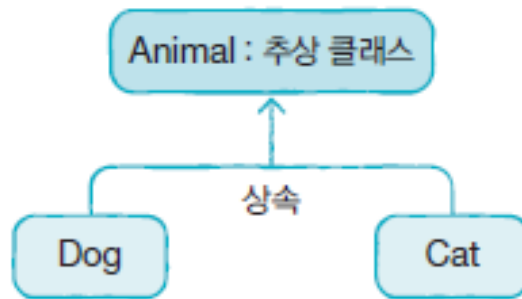
실행결과

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "a100"
    at java.base/java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
    at java.base/java.lang.Integer.parseInt(Integer.java:652)
    at java.base/java.lang.Integer.parseInt(Integer.java:770)
    at NumberFormatExceptionExample.main(NumberFormatExceptionExample.java:9)
```



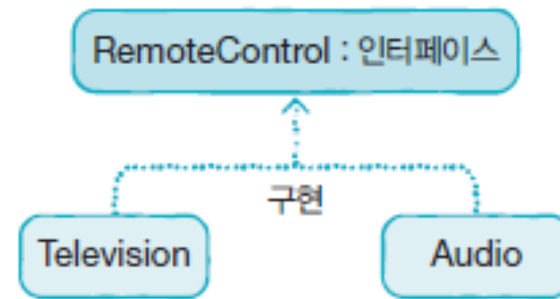
❖ ClassCastException

- 상위 및 하위 클래스 그리고 구현 클래스와 인터페이스 간 타입 변환 가능
- 위 관계가 아닌 경우 ClassCastException 발생



```
Animal animal = new Dog();
Dog dog = (Dog) animal;
```

```
Animal animal = new Dog();
Cat cat = (Cat) animal;
```



```
RemoteControl rc = new Television();
Television tv = (Television) rc;
```

```
RemoteControl rc = new Television();
Audio audio = (Audio) rc;
```

실행 예외

- instanceof 연산자로 타입 변환 가능 여부를 미리 확인

```
Animal animal = new Dog() ;  
if(animal instanceof Dog) {  
    Dog dog = (Dog) animal;  
} else if(animal instanceof Cat) {  
    Cat cat = (Cat) animal;  
}
```

```
Remocon rc = new Audio();  
if(rc instanceof Television) {  
    Television tv = (Television) rc;  
} else if(rc instanceof Audio) {  
    Audio audio = (Audio) rc;  
}
```



❖ 예시 - ClassCastException

```
01 package sec01.exam05;
02
03 public class ClassCastExceptionExample {
04     public static void main(String[] args) {
05         Dog dog = new Dog();
06         changeDog(dog);
07
08         Cat cat = new Cat();
09         changeDog(cat);
10     }
11
12     public static void changeDog(Animal animal) {
13         //if(animal instanceof Dog) {
14             Dog dog = (Dog) animal;    //ClassCastException 발생 가능
15         //}
16     }
17 }
18
19 class Animal {}
20 class Dog extends Animal {}
21 class Cat extends Animal {}
```



실행 예외

실행결과

```
Exception in thread "main" java.lang.ClassCastException: class Cat cannot be cast to class
Dog (Cat and Dog are in unnamed module of loader 'app')
    at ClassCastExceptionExample.changeDog(ClassCastExceptionExample.java:14)
    at ClassCastExceptionExample.main(ClassCastExceptionExample.java:9)
```



키워드로 끝내는 핵심 포인트

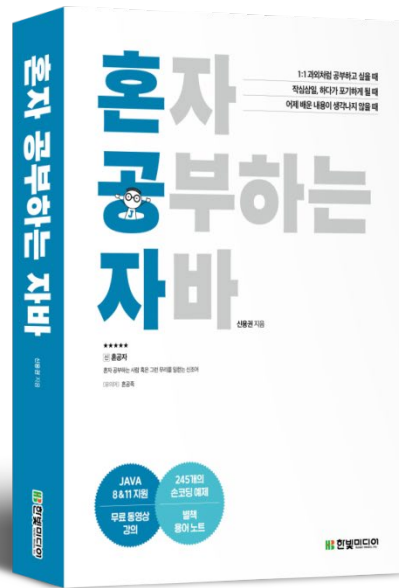
- **예외** : 사용자의 잘못된 조작 또는 개발자의 잘못된 코딩으로 인해 발생하는 프로그램 오류. 예외 발생 시 프로그램이 곧바로 종료되나, 예외 처리 통해 정상 실행상태를 유지할 수 있음
- **예외 클래스** : 자바에서는 예외를 클래스로 관리함. 프로그램 실행 중 예외가 발생하면 해당 예외 클래스로 객체를 생성하고 예외 처리 코드에서 예외 객체를 이용할 수 있도록 해줌.
- **일반 예외** : 컴파일러 체크 예외. 프로그램 실행 시 예외 발생 가능성 높기 때문에 자바 소스 컴파일 과정에서 해당 예외 처리 코드 있는지 검사함.
- **실행 예외** : 컴파일러 런 체크 예외. 실행 시 예측할 수 없이 갑자기 발생하기 때문에 컴파일 과정에서 예외 처리 코드 존재 여부를 검사하지 않음



❖ 예외에 대한 아래 설명 중 틀린 것을 고르세요

- 예외는 사용자의 잘못된 조작, 개발자의 잘못된 코딩으로 인한 프로그램 오류를 말한다.
- RuntimeException의 하위 클래스는 컴파일러가 예외 처리 코드를 체크하지 않는다.
- 예외는 클래스로 관리된다
- Exception의 하위 클래스는 모두 일반 예외에 해당한다.





10-2. 예외 처리

혼자 공부하는 자바 (신용권 저)

- 시작하기 전에
- 예외 처리 코드
- 예외 종류에 따른 처리 코드
- 예외 떠넘기기
- 키워드로 끝내는 핵심 포인트
- 확인문제



시작하기 전에

[핵심 키워드] : 예외 처리, try-catch-finally 블록, 다중 catch 블록, throws 키워드

[핵심 포인트]

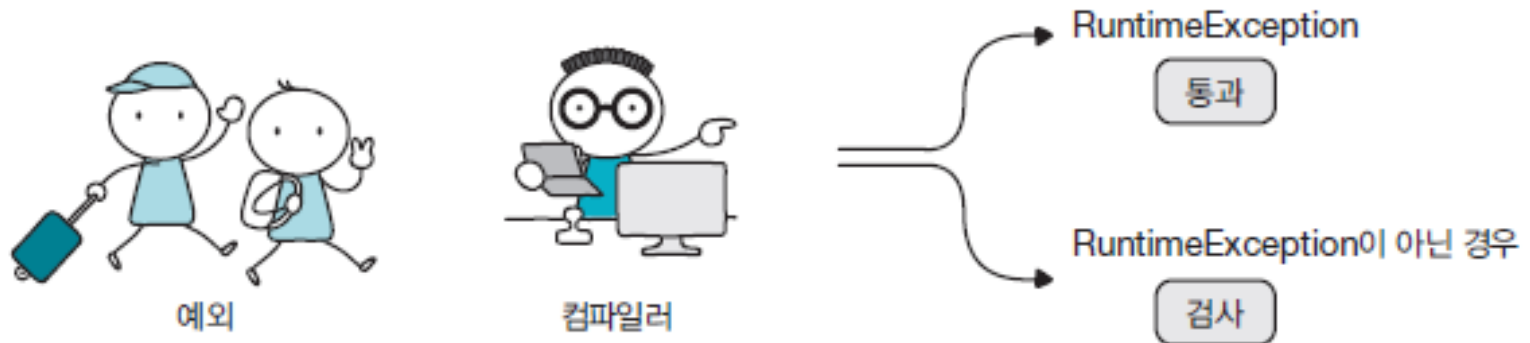
프로그램에서 예외가 발생했을 경우 프로그램의 갑작스러운 종료를 막고, 정상 실행을 유지할 수 있도록 예외 처리를 해야 한다. 예외 처리를 하는 방법에 대해 알아본다.



시작하기 전에

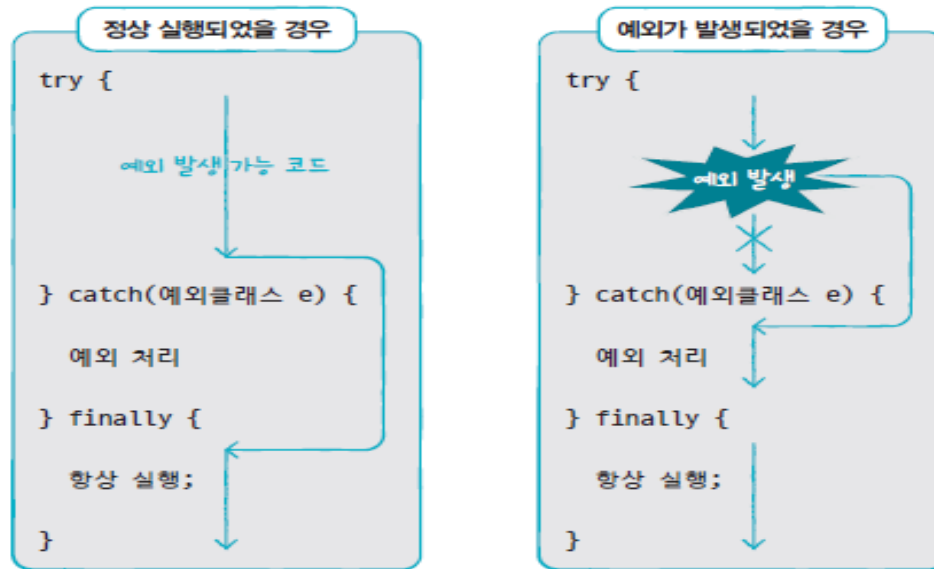
❖ 예외 처리 코드

- 자바 컴파일러는 소스 파일 컴파일 시 일반 예외 발생할 가능성이 있는 코드를 발견하면 컴파일 에러를 발생시켜 개발자에게 예외 처리 코드 작성을 요구
- 실행 예외의 경우 컴파일러가 체크하지 않으므로 개발자가 경험을 바탕으로 작성해야 함



❖ try-catch-finally 블록

- 생성자 및 메소드 내부에서 작성되어 일반예외와 실행예외가 발생할 경우 예외 처리 가능하게 함

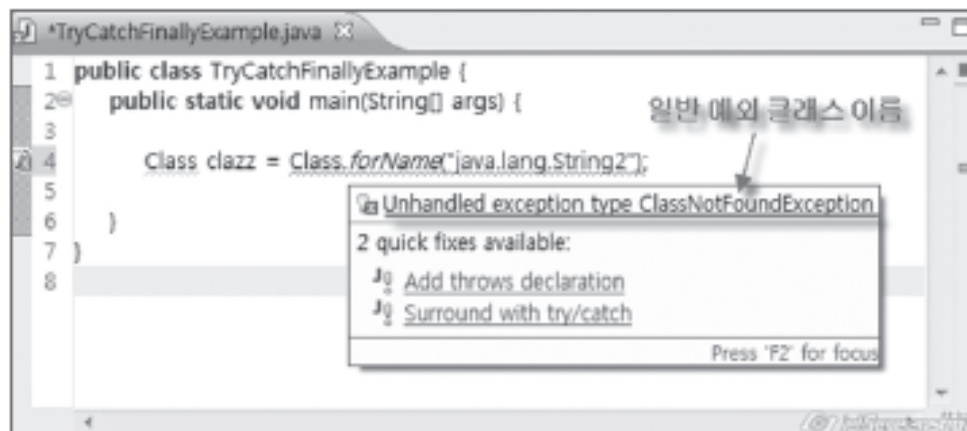


- try 블록에는 예외 발생 가능 코드가 위치
- try 블록 코드가 예외발생 없이 정상실행되면 catch 블록의 코드는 실행되지 않고 finally 블록의 코드를 실행. try 블록의 코드에서 예외가 발생한다면 실행 멈추고 catch 블록으로 이동하여 예외 처리 코드 실행. 이후 finally 블록 코드 실행
- finally 블록은 생략 가능하며, 예외와 무관하게 항상 실행할 내용이 있을 경우에만 작성.



예외 처리 코드

- 빨간색 밑줄로 예외 처리 코드 필요성 알림



❖ 예시 - 일반 예외 처리

```
01 package sec02.exam01;
02
03 public class TryCatchFinallyExample {
04     public static void main(String[] args) {
05         try {
06             Class clazz = Class.forName("java.lang.String2");
07         } catch(ClassNotFoundException e) {
08             System.out.println("클래스가 존재하지 않습니다.");
09         }
10     }
11 }
```

실행결과

클래스가 존재하지 않습니다.



❖ 예시 - 실행 예외 처리

```
01 package sec02.exam02;
02
03 public class TryCatchFinallyRuntimeExceptionExample {
04     public static void main(String[] args) {
05         String data1 = null;
06         String data2 = null;
07         try {
08             data1 = args[0];
09             data2 = args[1];
10         } catch (ArrayIndexOutOfBoundsException e) {
11             System.out.println("실행 매개값의 수가 부족합니다.");
12             return;
13         }
14
15         try {
16             int value1 = Integer.parseInt(data1);
17             int value2 = Integer.parseInt(data2);
18             int result = value1 + value2;
19             System.out.println(data1 + "+" + data2 + "=" + result);
20         } catch (NumberFormatException e) {
21             System.out.println("숫자로 변환할 수 없습니다.");
22         } finally {
23             System.out.println("다시 실행하세요.");
24         }
25     }
26 }
```

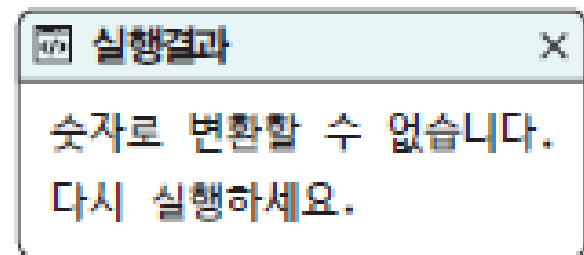
 실행결과

X

실행 매개값의 수가 부족합니다.

예외 처리 코드

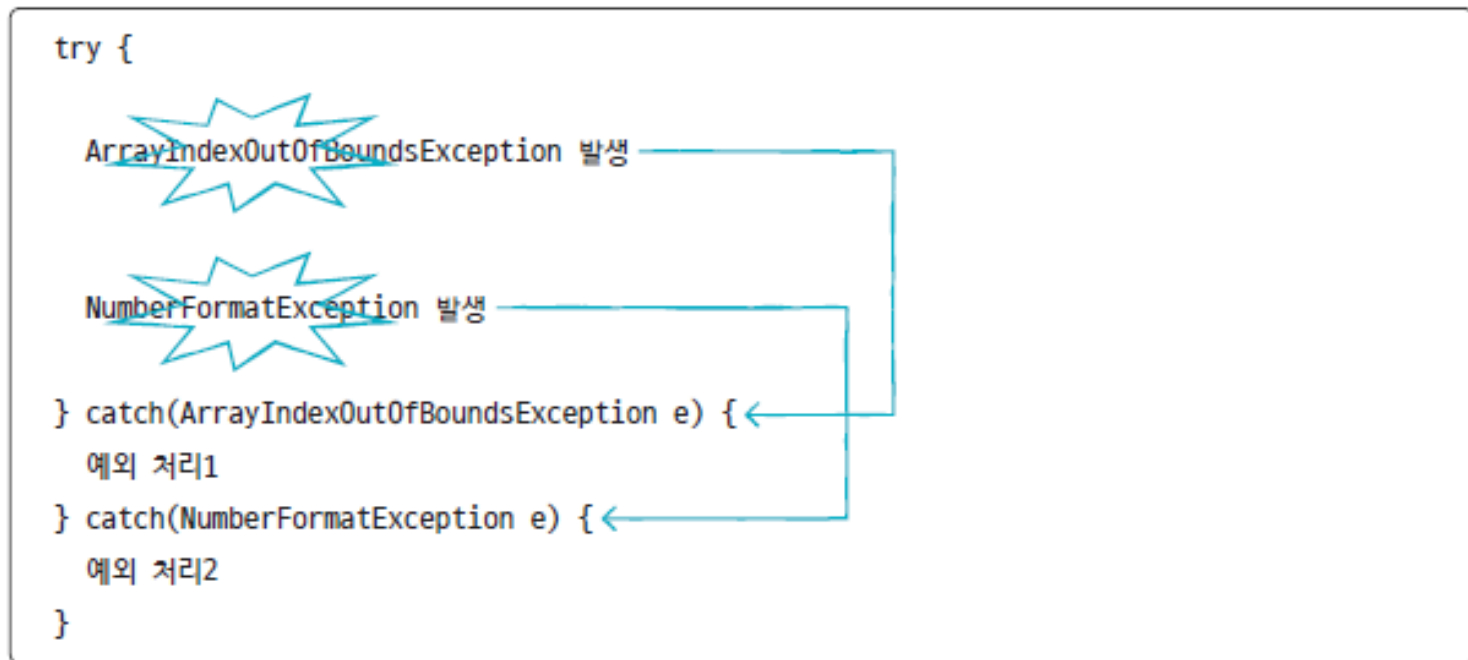
- 이클립스 - [Run] - [Run Configuration] 메뉴 선택
2개의 실행 매개값 주되 첫 번째 실행 매개값에 숫자가 아닌 문자 넣고 실행
16라인에서 예외 발생
21라인에서 예외처리 후 마지막으로 23라인 실행



예외 종류에 따른 처리 코드

❖ 다중 catch

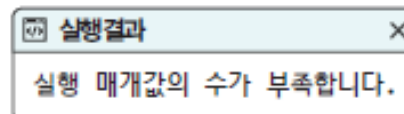
- 발생하는 예외별로 예외 처리 코드를 다르게 하는 다중 catch 블록
- catch 블록의 예외 클래스 타입은 try 블록에서 발생한 예외의 종류 말함
- try 블록에서 해당 타입 예외가 발생하면 catch 블록을 실행



예외 종류에 따른 처리 코드

❖ 예시



```
01 package sec02.exam03;
02
03 public class CatchByExceptionKindExample {
04     public static void main(String[] args) {
05         try {
06             String data1 = args[0];
07             String data2 = args[1];
08             int value1 = Integer.parseInt(data1);
09             int value2 = Integer.parseInt(data2);
10             int result = value1 + value2;
11             System.out.println(data1 + "+" + data2 + "=" + result);
12         } catch (ArrayIndexOutOfBoundsException e) {
13             System.out.println("실행 매개값의 수가 부족합니다.");
14         } catch (NumberFormatException e) {
15             System.out.println("숫자로 변환할 수 없습니다.");
16         } finally {
17             System.out.println("다시 실행하세요.");
18         }
19     }
20 }
```

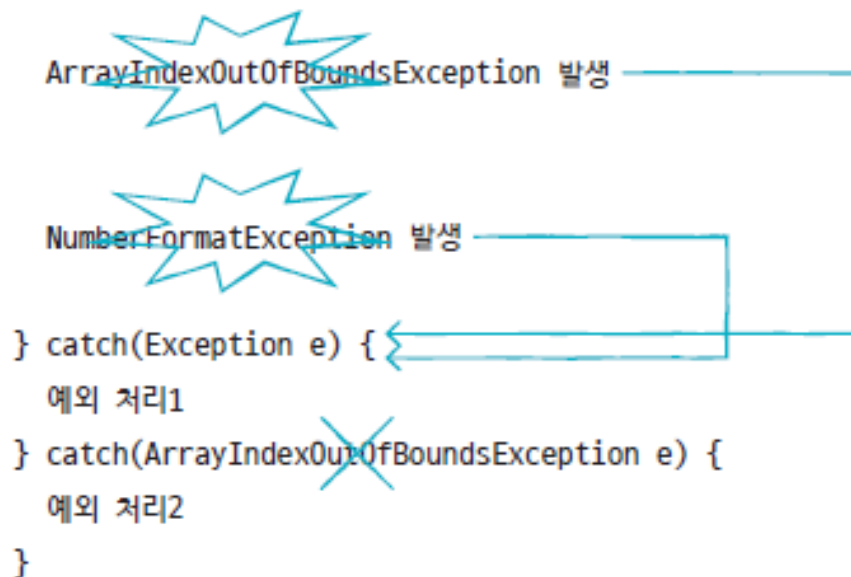


예외 종류에 따른 처리 코드

❖ catch 순서

- 다중 catch 블록 작성 시 상위 예외 클래스가 하위 예외 클래스보다 아래 위치해야 함
- 잘못된 예

```
try {  
     ArrayIndexOutOfBoundsException 발생  
     NumberFormatException 발생  
} catch (Exception e) {  
    예외 처리1  
} catch (ArrayIndexOutOfBoundsException e) {  
    예외 처리2  
}
```




예외 종류에 따른 처리 코드

■ 올바른 예

```
try {
```

 `ArrayIndexOutOfBoundsException` 발생

 다른 `Exception` 발생

```
} catch(ArrayIndexOutOfBoundsException e) { <
```

예외 처리1

```
} catch(Exception e) { <
```

예외 처리2

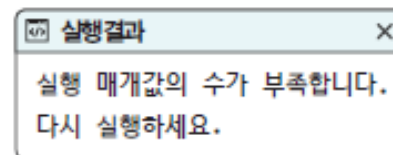
```
}
```



예외 종류에 따른 처리 코드

❖ 예시 - catch 블록의 순서

```
01 package sec02.exam01;
02
03 public class CatchOrderExample {
04     public static void main(String[] args) {
05         try {
06             String data1 = args[0];
07             String data2 = args[1];
08             int value1 = Integer.parseInt(data1);
09             int value2 = Integer.parseInt(data2);
10             int result = value1 + value2;
11             System.out.println(data1 + "+" + data2 + "=" + result);
12         } catch (ArrayIndexOutOfBoundsException e) {
13             System.out.println("실행 매개값의 수가 부족합니다.");
14         } catch (Exception e) {
15             System.out.println("실행에 문제가 있습니다.");
16         } finally {
17             System.out.println("다시 실행하세요.");
18         }
19     }
20 }
```



❖ throws 키워드

- 메소드 선언부 끝에 작성되어 메소드에서 처리하지 않은 예외를 호출한 곳으로 넘기는 역할
- throws 키워드 뒤에는 떠넘길 예외 클래스를 쉼표로 구분하여 나열

```
리턴타입 메소드이름(매개변수,...) throws 예외클래스1, 예외클래스2, ... {  
}
```

```
리턴타입 메소드이름(매개변수,...) throws Exception {  
}
```



예외 떠넘기기

```
public void method1() {  
    try {  
        method2();  
    } catch(ClassNotFoundException e) {  
        //예외 처리 코드  
        System.out.println("클래스가 존재하지 않습니다.");  
    }  
}
```

호출한 곳에서 예외 처리

```
public void method2() throws ClassNotFoundException {  
    Class clazz = Class.forName("java.lang.String2");  
}
```

method1()에서 try-catch 블록으로 예외 처리하지 않고 throws 키워드로 다시 예외 떠넘기는 경우

```
public void method1() throws ClassNotFoundException {  
    method2();  
}
```



❖ 예시 - 예외 처리 떠넘기기

```
01 package sec02.exam02;
02
03 public class ThrowsExample {
04     public static void main(String[] args) {
05         try {
06             findClass();
07         } catch(ClassNotFoundException e) {
08             System.out.println("클래스가 존재하지 않습니다.");
09         }
10     }
11
12     public static void findClass() throws ClassNotFoundException {
13         Class clazz = Class.forName("java.lang.String2");
14     }
15 }
```

- main() 메소드에서 throws 키워드 사용하여 예외 떠넘기는 경우

```
public static void main(String[] args) throws ClassNotFoundException {
    findClass();
}
```

키워드로 끝내는 핵심 포인트

- **예외 처리** : 프로그램에서 예외 발생하는 경우 프로그램의 갑작스러운 종료 막고 정상 실행상태 유지할 수 있도록 처리하는 것.
- **try-catch-finally 블록** : 생성자 내부와 메소드 내부에서 작성되어 일반 예외와 실행 예외 발생하는 경우 예외 처리 할 수 있도록 함
- **다중 catch 블록** : catch 블록이 여러 개이더라도 하나의 catch 블록만 실행함. try 블록에서 동시다발적으로 예외가 발생하지 않고, 하나의 예외 발생했을 때 즉시 실행 멈추고 해당 catch 블록으로 이동하기 때문.
- **throws 키워드** : 메소드 선언부 끝에 작성되어 메소드에서 처리하지 않은 예외를 호출한 곳으로 떠넘기는 역할.



- ❖ try-catch-finally 블록에 대한 설명 중 틀린 것을 고르세요
 - try {} 블록에는 예외가 발생할 수 있는 코드를 작성한다
 - catch {} 블록은 try{} 블록에서 발생한 예외를 처리하는 블록이다
 - try {} 블록에서 return문을 사용하면 finally{} 블록은 실행되지 않는다
 - catch {} 블록은 예외의 종류별로 여러 개를 작성할 수 있다
- ❖ 다음 코드가 실행되었을 때 출력 결과는 무엇입니까?

소스 코드 TryCatchFinallyExample.java

```
01 String[] strArray = { "10", "2a" };
02 int value = 0;
03 for(int i=0; i<=2; i++) {
04     try {
05         value = Integer.parseInt(strArray[i]);
06     } catch(ArrayIndexOutOfBoundsException e) {
07         System.out.println("인덱스를 초과했음");
08     } catch(NumberFormatException e) {
09         System.out.println("숫자로 변환할 수 없음");
10     } finally {
11         System.out.println(value);
12     }
13 }
```

