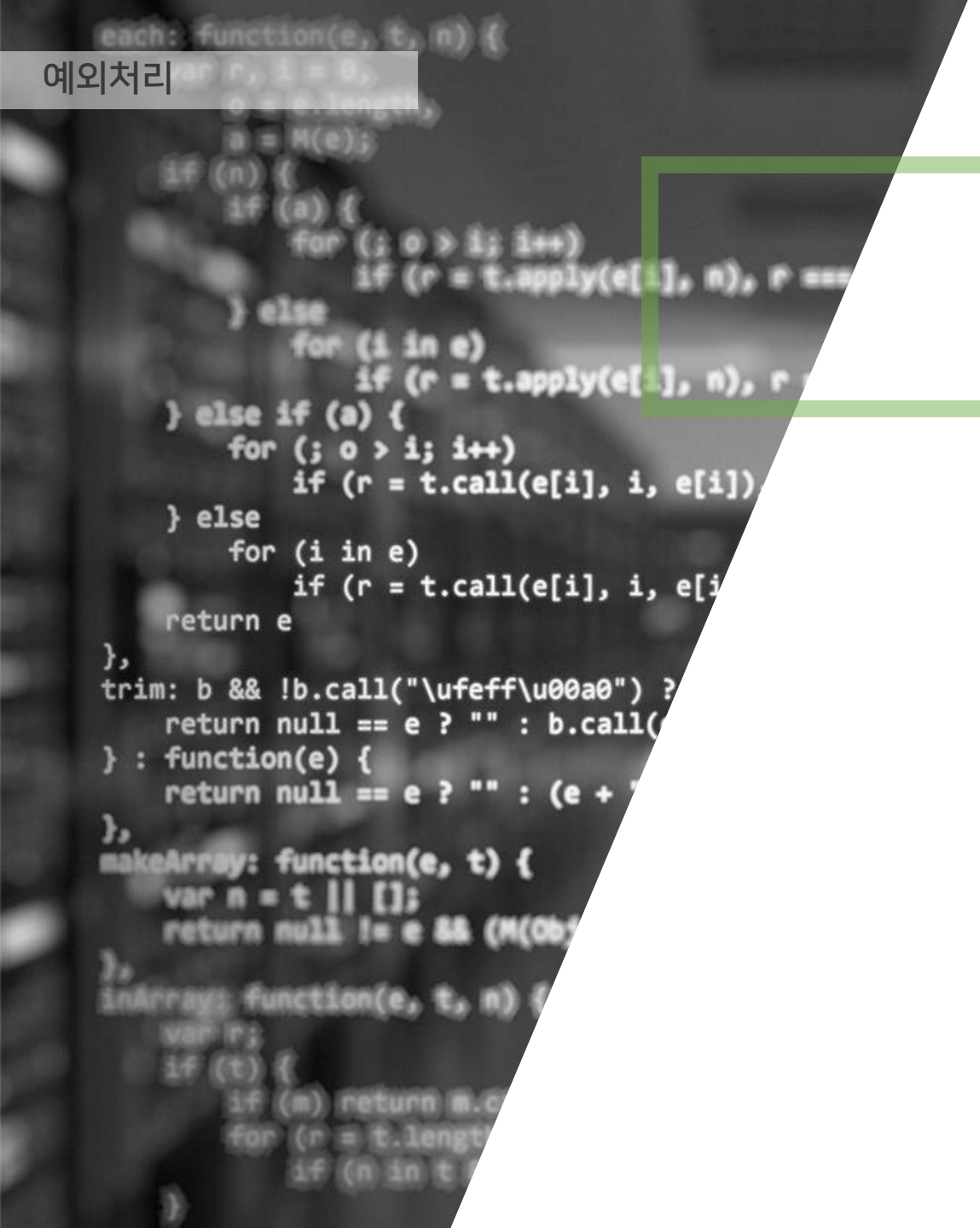
제11장

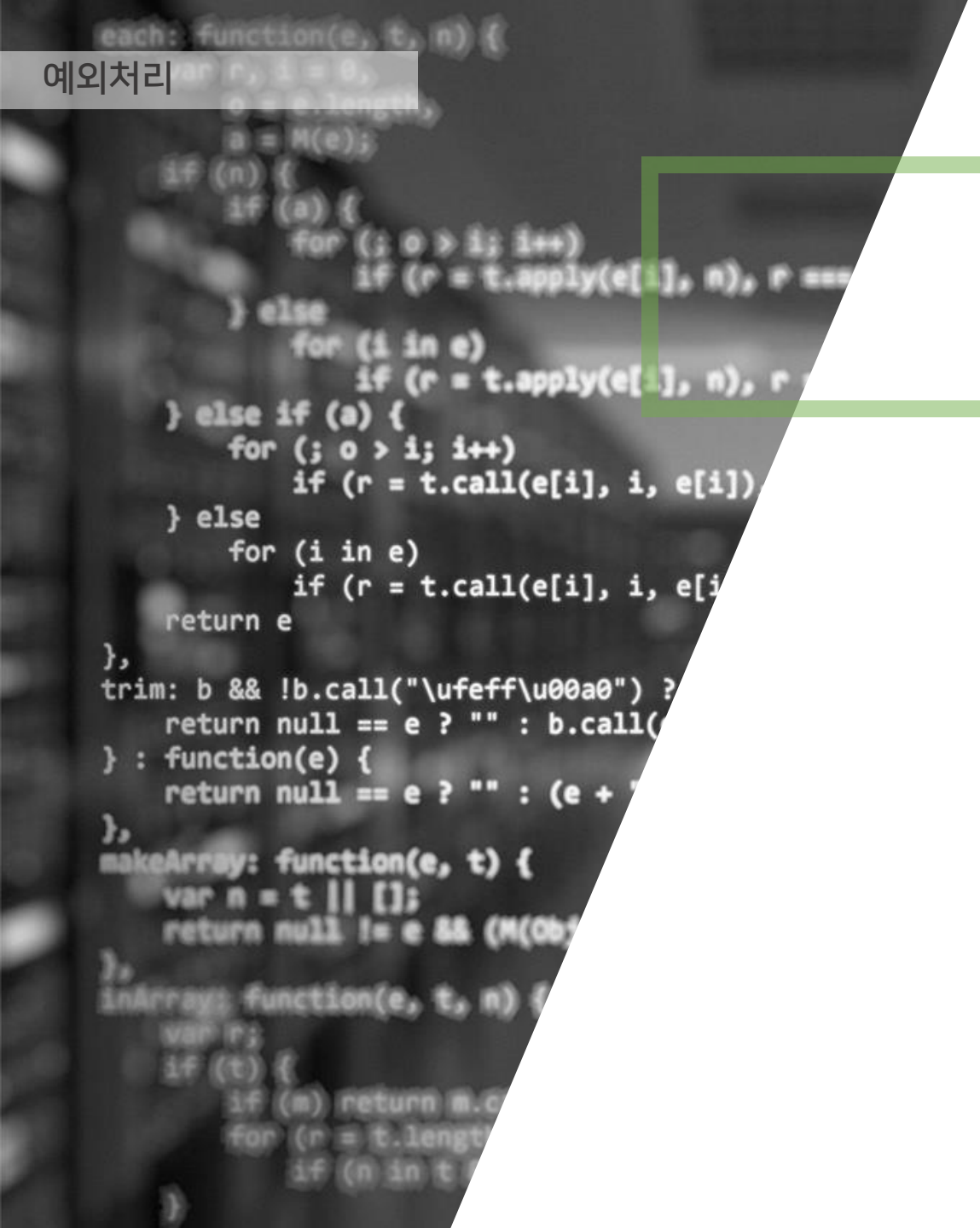
예외처리

구디아카데미 ▷ 민경태 강사

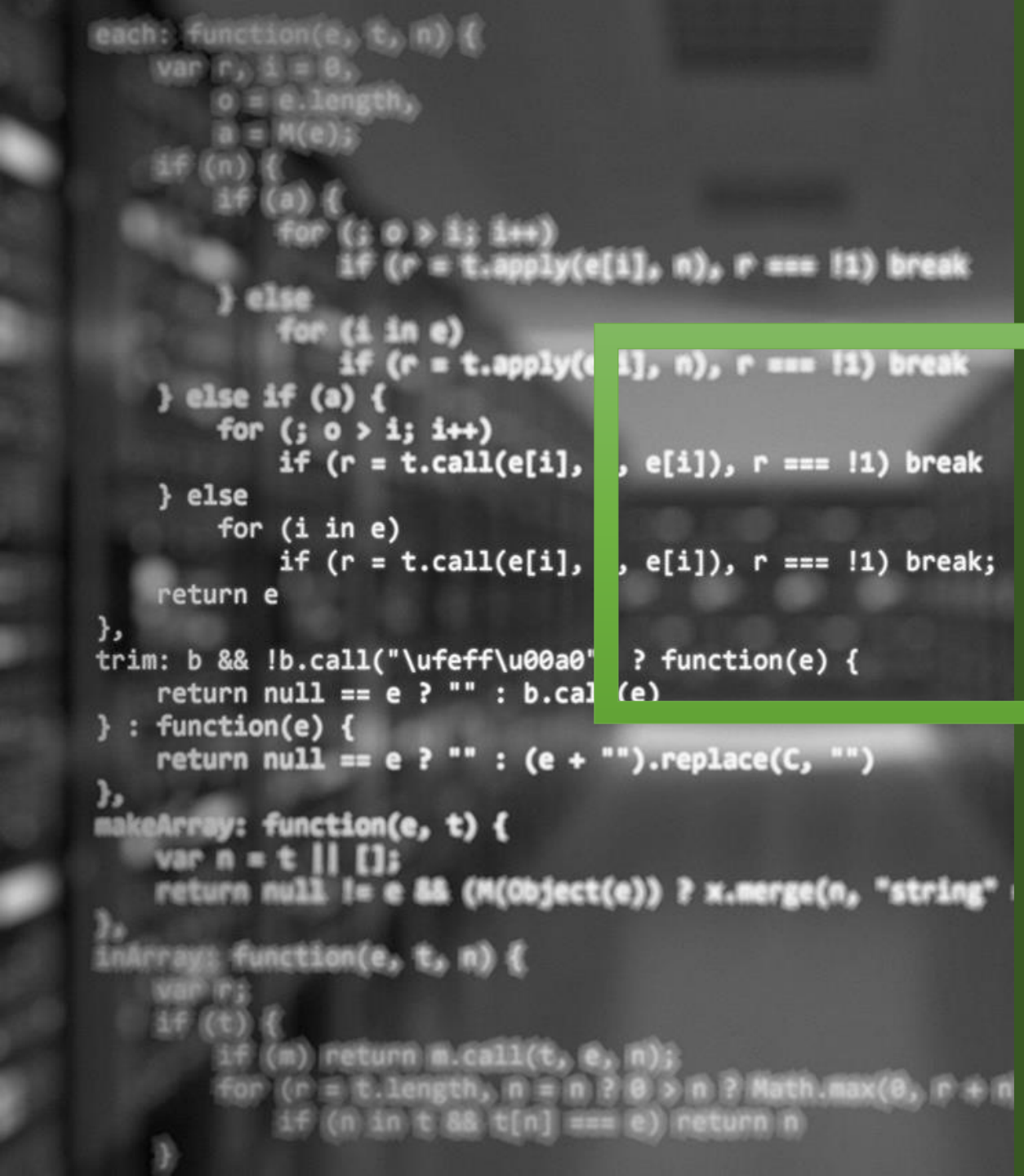


학습목표

1. 예외의 개념과 예외 클래스에 대해서 알 수 있다.
2. try catch문에 대해서 알 수 있다.
3. throw문과 throws문에 대해서 알 수 있다.
4. 사용자 정의 예외 클래스에 대해서 알 수 있다.



목차



1. 예외의 개념과 예외 클래스

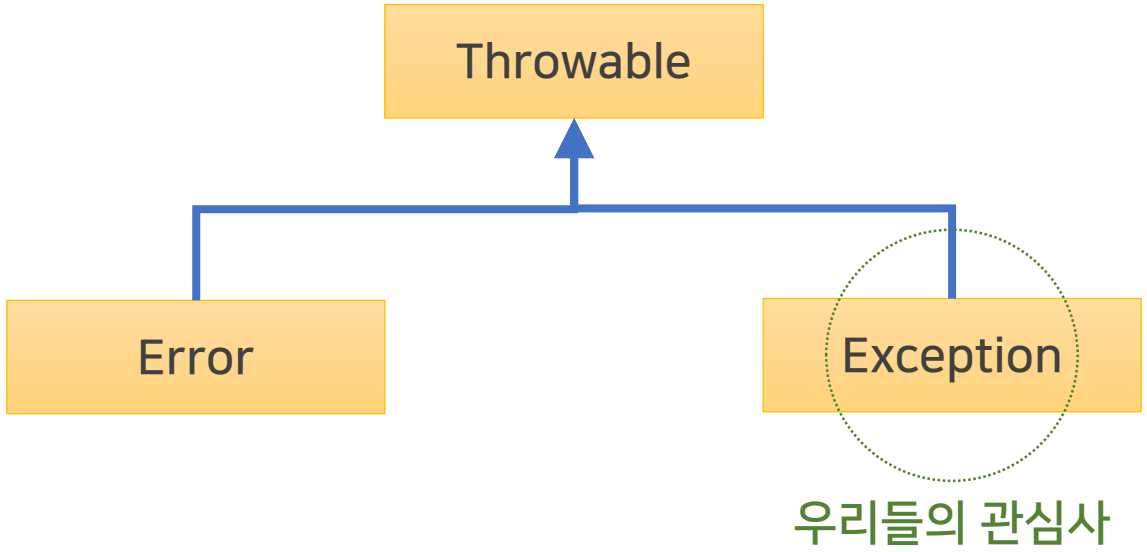
오류와 예외

- 오류(Error)
  - 시스템 레벨의 심각한 문제점
  - 외부 요인으로 발생하기 때문에 개발자의 대처가 사실상 불가능
- 예외(Exception)
  - 프로그램 레벨의 일반적인 문제점
  - 코드 수정으로 문제점 해결
    - ✓ 널(Null) 처리
    - ✓ 공백(Empty) 검사
    - ✓ 예외 처리 등

일반적으로 발생하는 문제점은 예외(Exception)이다.

Error 클래스와 Exception 클래스

- Error 클래스
  - 오류 처리를 위한 자바 클래스
  - 오류 클래스들의 슈퍼 클래스
- Exception 클래스
  - 예외 처리를 위한 자바 클래스
  - 예외 클래스들의 슈퍼 클래스



예외

- Exception

- 개발자가 코드 수정 등을 통해 처리할 수 있는 수준의 문제점을 의미함

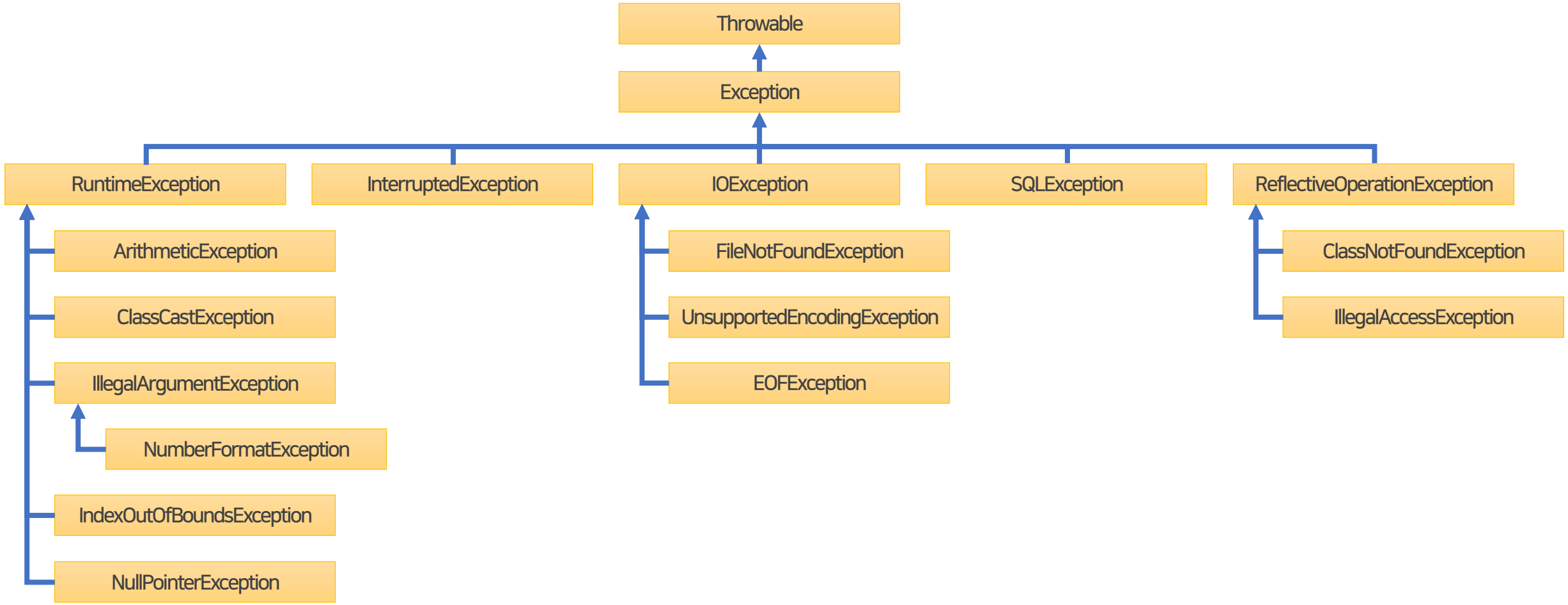
- 주요 예외 발생 원인
  1. 코드 자체의 문제
  2. 사용자들의 비정상적인 사용

```
Exception in thread "main" java.lang.ClassCastException: class Person cannot be cast to class Student at
MainClass.main(MainClass.java:10)
```

예시) ClassCaseException 예외가 발생한 모습

예외 클래스 구조

- 주요 Exception 클래스 구조



예외 클래스 종류

▪ 주요 예외 클래스

예외 클래스	역할
Throwable	Error 클래스와 Exception 클래스의 슈퍼 클래스
Exception	모든 예외 클래스들의 슈퍼 클래스
RuntimeException	JVM이 실행하는 도중에 발생하는 모든 예외 클래스들의 슈퍼 클래스
ArithmeticException	산술 연산의 문제로 인해 발생(0으로 나눈 몫을 구하는 경우)
ClassCastException	어떤 객체를 변환할 수 없는 클래스타입으로 변환(Casting)하는 경우에 발생
IndexOutOfBoundsException	배열의 인덱스가 가용 범위를 벗어난 경우에 발생
NullPointerException	null값을 참조하는 경우에 발생(null값으로 메소드를 호출할 때)
NumberFormatException	String을 Number타입으로 변환하지 못하는 경우에 발생
IOException	실패하거나 중단된 입출력(I/O) 작업으로 인해 발생
FileNotFoundException	지정된 파일을 찾을 수 없는 경우에 발생
UnsupportedEncodingException	지원하지 않는 인코딩 방식인 경우에 발생
InterruptedException	스레드(Thread)가 대기/휴면/다른 스레드에 의해 점유 중일때 발생
SQLException	데이터베이스 처리가 실패하는 경우에 발생

Throwable 클래스 - 생성자

- Throwable 클래스 주요 생성자

생성자	역할
Throwable()	예외 메시지를 null로 생성함
Throwable(String message)	예외 메시지를 message로 생성함
Throwable(String message, Throwable cause)	예외 메시지를 message로 생성하고 예외 원인 객체를 저장함
Throwable(Throwable cause)	예외 원인 객체를 저장함

Throwable 클래스 - 메소드

- Throwable 클래스 주요 메소드

메소드	역할
Throwable getCause()	예외가 발생한 근본적인 원인 예외 객체 반환. 없는 경우 null을 반환
String getMessage()	예외 메시지를 반환
void printStackTrace()	어떤 부분에서 예외가 발생했는지 알기 위해 현재 작업(스레드)에 대한 스택 정보를 표준 출력 스트림으로 출력

예외 처리

- 예외가 발생하면 프로그램이 비정상적으로 종료될 수 있음

- 예외 처리(Exception Handling)란 프로그램이 동작할 때 발생 가능한 상황에 대비하여 프로그램이 정상적인 실행상태를 유지할 수 있도록 미리 코드를 작성하는 것을 의미함

- 예외 처리 목적
  - 예외 발생 자체를 막는 것은 어려움
  - 예외가 발생하더라도 프로그램이 비정상적으로 종료되는 것을 막기 위해서 예외 처리를 해야 함

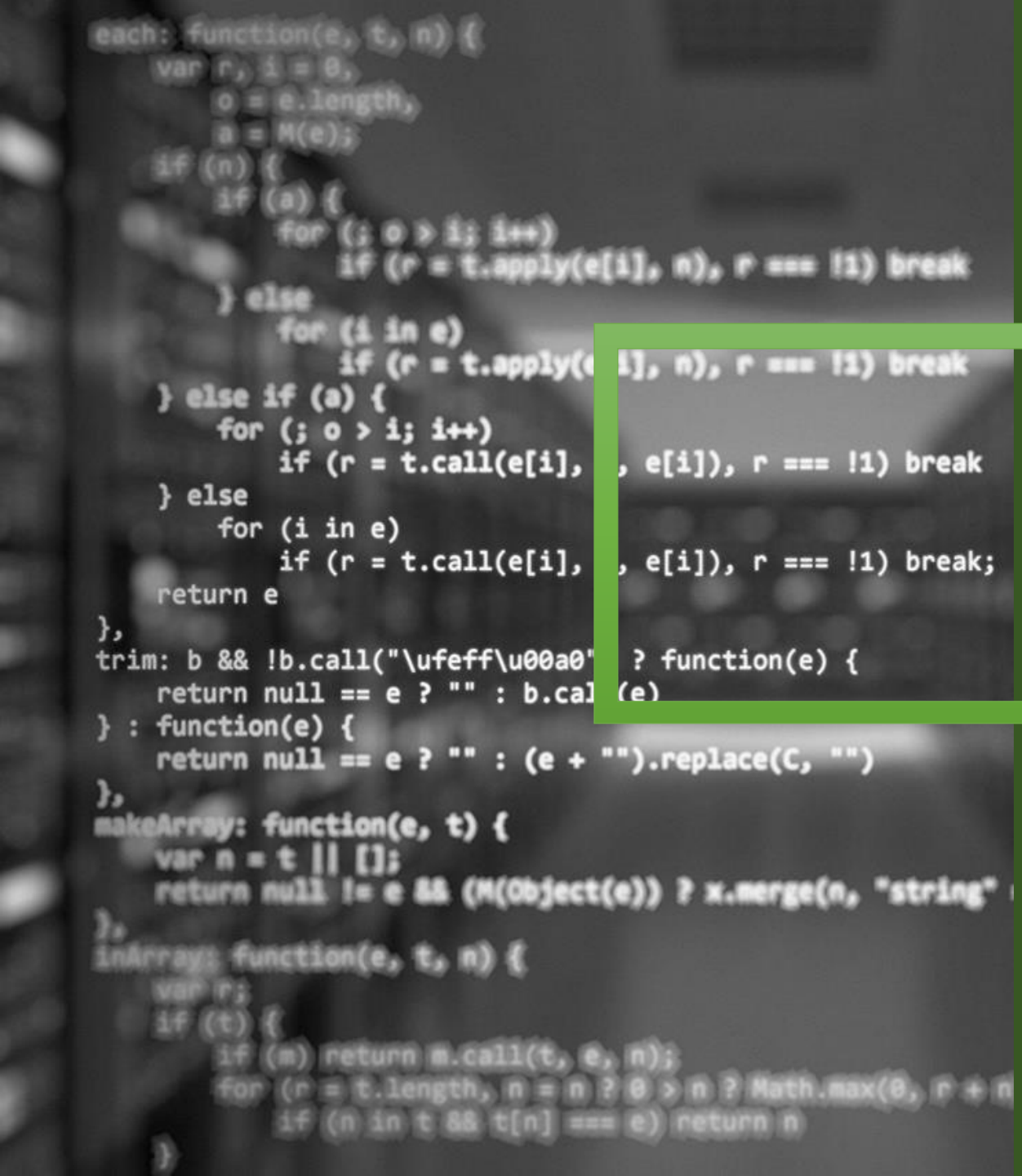
예외 구분

- 예외 처리의 필수 여부에 따라 Checked Exception과 Unchecked Exception으로 구분함
- Checked Exception은 명시적으로 예외 처리를 하지 않으면 실행이 되지 않기 때문에 반드시 예외 처리를 해야 함
- Unchecked Exception은 예외 처리를 하지 않아도 실행할 수 있으나 실행 중에 예외가 발생할 가능성이 있음

Checked Exception과 Unchecked Exception

- 주요 차이점 비교

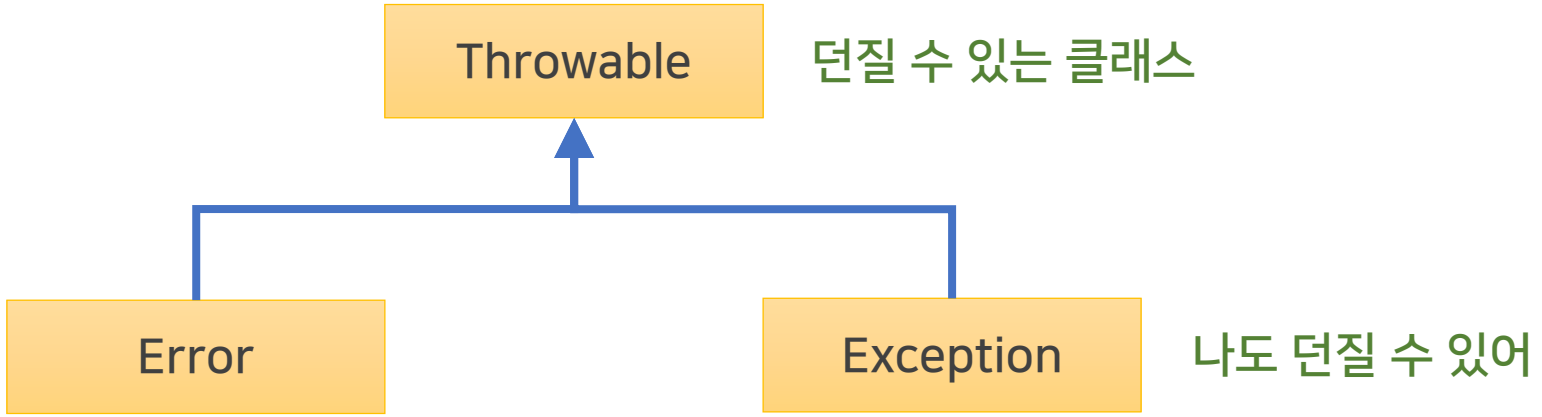
구분	Checked Exception	Unchecked Exception
예외 처리	필수	선택
예외 확인 시점	컴파일할 때	실행할 때
트랜잭션	rollback 처리 안 함	rollback 처리함
종류	RuntimeException을 제외한 모든 예외 클래스 IOException FileNotFoundException UnsupportedEncodingException UnknownHostException MalformedURLException InterruptedException SQLException ...	RuntimeException과 그 하위 클래스 ArithmeticException ClassCaseException IllegalArgumentException IndexOutOfBoundsException NoSuchElementException NullPointerException NumberFormatException ...


each: function(e, t, n) {
var r, i = 0,
o = e.length,
a = M(e);
if (n) {
if (a) {
for (; o > i; i++)
if (r = t.apply(e[i], n), r === !1) break
} else
for (i in e)
if (r = t.apply(e[i], n), r === !1) break
} else if (a) {
for (; o > i; i++)
if (r = t.call(e[i], e[i]), r === !1) break
} else
for (i in e)
if (r = t.call(e[i], e[i]), r === !1) break;
return e
},
trim: b && !b.call("\ufeff\u00a0") ? function(e) {
return null == e ? "" : b.call(e)
} : function(e) {
return null == e ? "" : (e + "").replace(C, "")
},
makeArray: function(e, t) {
var n = t || [];
return null != e && (M(Object(e)) ? x.merge(n, "string"
},
inArray: function(e, t, n) {
var r;
if (t) {
if (m) return m.call(t, e, n);
for (r = t.length, n = n ? 0 > n ? Math.max(0, r + n
if (n in t && t[n] === e) return n
}

2. try catch문

예외를 던진다

- Throwable 클래스는 왜 이름이 Throwable인가?



자바는 "예외를 던진다"고 표현한다.

try catch문

- 예외를 처리할 때 사용하는 코드
- 실행할 코드는 try 블록에 두고 예외를 처리할 코드는 catch 블록에 두는 방식
- try 블록에서 예외가 발생하면 해당 예외는 자동으로 catch 블록으로 전달됨

try 블록에서 예외를 던지면 catch 블록이 예외를 받는다.

try catch문 형식

- try catch문 형식

```
try {
    실행 코드
} catch(예외 객체 선언) {
    예외 처리 코드
}
```

try catch문 동작 원리

- 예외가 없는 경우
  ① ② ③ ⑤ 순으로 진행

```
try {
    ① 실행 코드
    ② 실행 코드
    ③ 실행 코드
} catch(예외 객체 선언) {
    ④ 예외 처리 코드
}
⑤ try catch 이후 코드
```

- 예외가 발생한 경우
  ① ④ ⑤ 순으로 진행

```
try {
    ① 예외 발생 코드
    ② 실행 코드
    ③ 실행 코드
} catch(예외 객체 선언) {
    ④ 예외 처리 코드
}
⑤ try catch 이후 코드
```

다중 catch 블록

- 하나의 try 블록에 여러 개의 catch 블록을 추가할 수 있음
- 예외 발생 시 작성된 catch 블록 순서대로 예외 처리를 시도함
- 형식

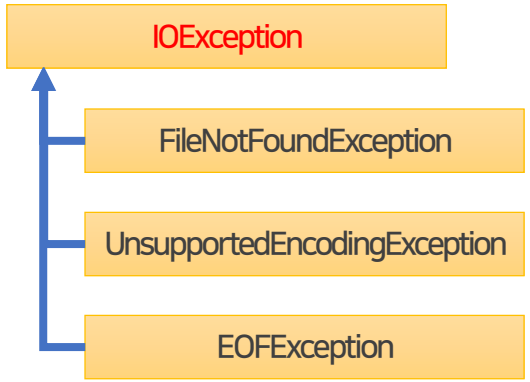
```
try {
    실행 코드
} catch(예외 객체 선언) {
    예외 처리 코드
} catch(예외 객체 선언) {
    예외 처리 코드
}
```

① 예외가 발생하면 첫 번째 catch 블록이 예외 처리를 시도한다.

② 첫 번째 catch 블록이 예외 처리를 못하면 두 번째 catch 블록이 예외 처리를 시도한다.

다중 catch 블록 주의사항

- 예외 클래스의 상속 구조를 참고하여 자식 타입을 먼저 catch 블록으로 배치하고 부모 타입을 나중에 배치해야 함



상속 구조 예시

```
try {
    실행 코드
} catch(FileNotFoundException e) {
    예외 처리 코드
} catch(UnsupportedEncodingException e) {
    예외 처리 코드
} catch(EOFException e) {
    예외 처리 코드
} catch(IOException e) {
    예외 처리 코드
}
```

자식 타입을 위에 둔다.

부모 타입을 아래에 둔다.

finally 블록

- try catch문 마지막에 추가할 수 있는 선택 블록
- 예외 발생 여부와 상관없이 항상 마지막에 실행되는 블록임
- 일반적으로 사용한 자원을 반납(close)하는 용도로 사용함

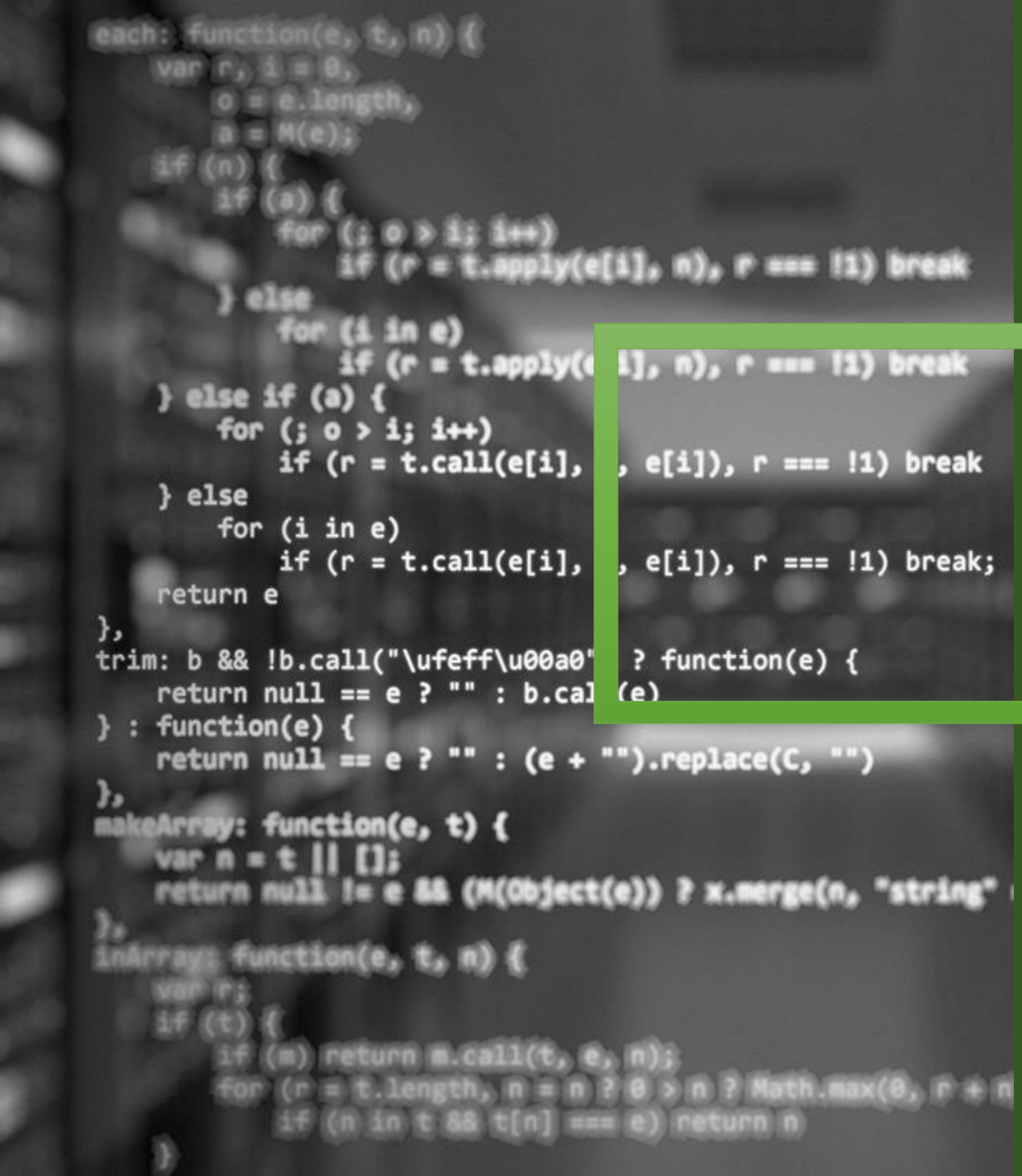
finally 블록 형식

- finally 블록 형식

```
try {
    실행 코드
} catch(예외 객체 선언) {
    예외 처리 코드
} finally {
    항상 마지막에 실행되는 코드
}
```

- finally 블록 내부에 try catch문

```
try {
    실행 코드
} catch(예외 객체 선언) {
    예외 처리 코드
} finally {
    try {
        실행 코드
    } catch(예외 객체 선언) {
        예외 처리 코드
    }
}
```



3. throw문과 throws문

예외 던지기

- throw

- 개발자가 직접 예외 객체를 만들어서 던질 때 사용

- 예외 원인 객체를 전달하고 싶은 경우 또는 자바는 예외로 인식하지 못하지만 실제로는 예외인 경우에 주로 사용

- 자바가 예외로 인식 못하는 예시
  - 상황 ▶ 점수(0-100) 저장을 위한 변수 int score를 선언하고 999를 입력함
  - 자바 ▶ 999는 정수이므로 문제 없음
  - 실제 ▶ 존재할 수 없는 점수이므로 예외 처리해야 함

throw 예시

- 변수 int score에 999가 저장된 상황

```
public class MainClass {
    public static void main(String[] args) {
        try {
            int score = 999;
            if(score < 0 || score > 100) {
                throw new RuntimeException(score + "점은 존재할 수 없습니다.");
            }
            System.out.println(score);
        } catch( RuntimeException e ){
            System.out.println(e.getMessage());
        }
    }
}
```

예외를 던지면 catch 블록이 받는다.

실행결과

"999점은 존재할 수 없습니다."

예외 처리 회피

- 예외 처리가 필요한 메소드에서 예외를 처리하지 않는 방식
- 메소드를 호출한 곳으로 예외를 던지는 방식
- throws 키워드를 이용해서 예외 처리를 회피할 수 있음
- 형식

```
반환타입 메소드명(매개변수) throws 예외클래스1, 예외클래스2, ... {

}
```

throws 예시

- 나누기 상황, 0으로 나누면 ArithmeticException이 발생함

```
public class Calculator {

    public void div(int a, int b) throws ArithmeticException {
        System.out.println(a / b);
    }

}
```

① 매개변수 int a, int b로 5, 0을 전달한다.

② 5 / 0 을 하면 예외가 발생한다.

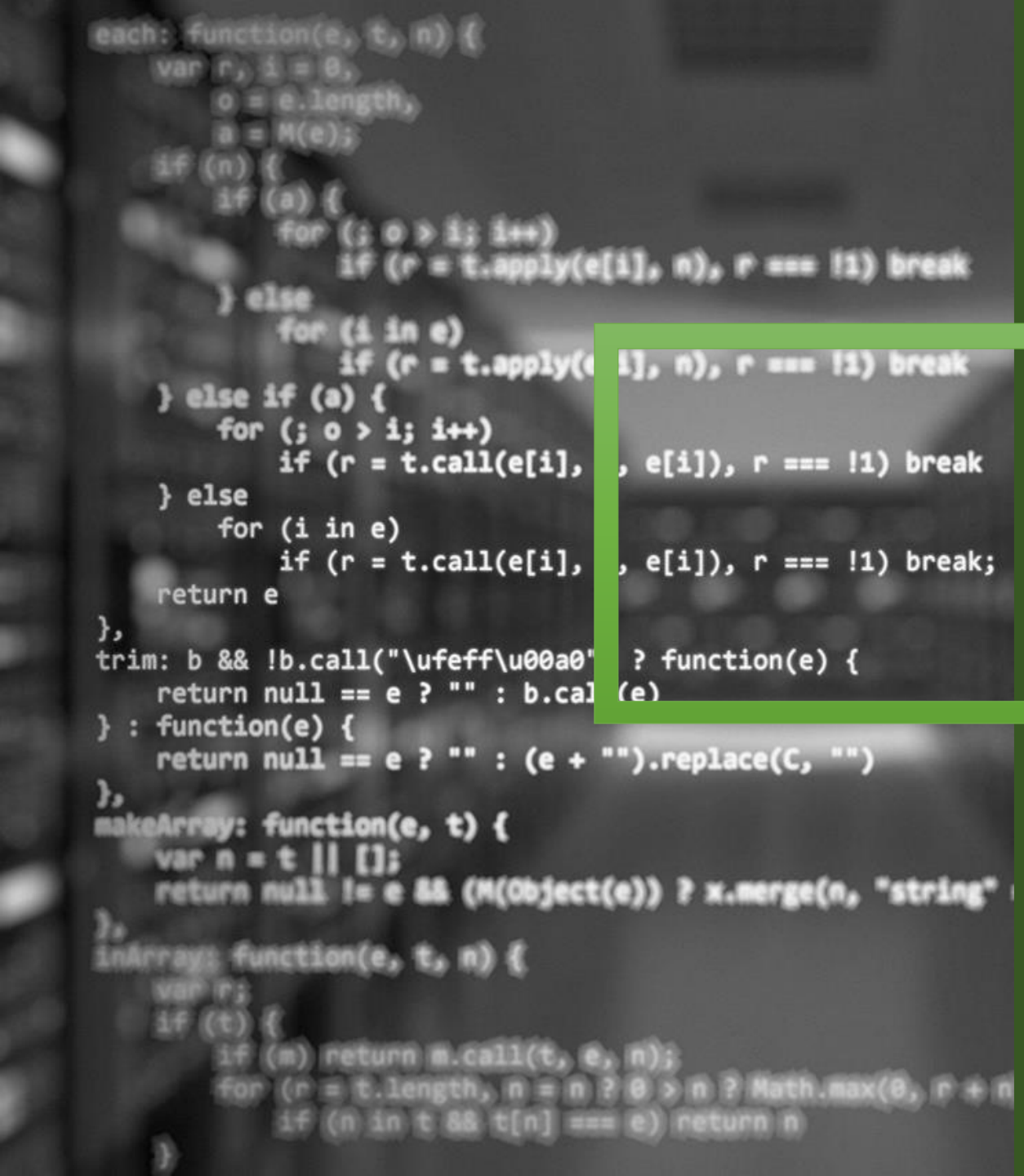
③ 발생한 예외를 호출한 곳으로 던진다.

```
public class MainClass {

    public static void main(String[] args) {
        Calculator calc = new Calculator();
        try {
            calc.div( 5, 0 );
        } catch( ArithmeticException e ) {
            System.out.println(e.getMessage());
        }
    }

}
```

④ 예외를 처리한다.

실행결과

```
"/ by zero"
```



4. 사용자 정의 예외 클래스

사용자 정의 예외 클래스

- 자바 표준 API에서 제공하는 예외 클래스를 이용해서 처리할 수 없는 예외의 경우 직접 예외 클래스를 만들어서 사용할 수 있음

- Exception 클래스나 RuntimeException 클래스를 상속 받는 방식으로 사용자 정의 예외 클래스를 만들 수 있음

- 형식

```
class 예외클래스 extends Exception {

}
```

사용자 정의 예외 클래스 특징

- 개발자가 직접 예외 메시지를 정의할 수 있음
- 자바 표준 API 예외 클래스가 지원하지 않는 필드나 메소드를 새롭게 정의할 수 있음
- JVM에 의해서 자동으로 예외가 발생하지 않기 때문에 개발자가 직접 throw문을 이용해서 예외를 발생시켜야함

사용자 정의 예외 메시지

- 생성자를 정의하고 Exception 클래스에 예외 메시지를 전달함

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

② 전달 받은 예외 메시지를 다시 Exception 클래스에 전달함

```
public class MainClass {
    public static void main(String[] args) {
        new MyException("사용자 정의 예외 메시지");
    }
}
```

① 예외 객체 생성 시 예외 메시지를 전달함

사용자 정의 예외 메시지 처리 과정

- Exception 클래스가 자신에게 전달된 예외 메시지를 Throwable 클래스에 전달하면 Throwable 클래스가 예외 메시지를 저장함

```
public class Throwable {
    private String detailMessage;
    public Throwable(String message) {
        detailMessage = message;
    }
}
```

② 전달 받은 예외 메시지를 String detailMessage에 저장함

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

① 전달 받은 예외 메시지를 다시 Throwable 클래스에 전달함

사용자 정의 예외 던지기

- throw문을 활용해 예외 던지기

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

```
public class MainClass {
    public static void main(String[] args) {
        try {
            throw new MyException("사용자 정의 예외 메시지");
        } catch (MyException e) {
            System.out.println(e.getMessage());
        }
    }
}
```