yonggyo1981 강의 동영상 URL 추가 4b14b4b · 2 years ago History

Name	Name	Last commit date
..		
day11	예외처리, java.lang패키지, 유용...	2 years ago
images	기본 클래스 강의 작성	2 years ago
README.md	강의 동영상 URL 추가	2 years ago

README.md

강의 동영상 링크

동영상 링크

예외처리

예외 클래스

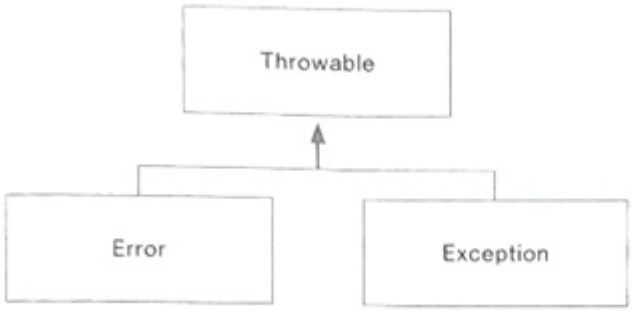
오류란 무인가요?

- **컴파일 오류(compile error)**
  - 프로그램 코드 작성 중 실수로 발생하는 오류
  - 발생한 컴파일 오류를 모두 수정해야 프로그램이 정상적으로 실행 되므로, 문법적으로 오류가 있다는 것을 바로 알 수 있습니다.
- **실행 오류(runtime error)**

- 실행 중인 프로그램이 의도하지 않은 동작을 하거나 프로그램이 중지되는 오류 입니다. 실행 오류 중 프로그램을 잘목 수현하여 의도한 바와 다르게 실행되어 생기는 오류를 **버그(bug)**라고 합니다.
- 프로그램 실행 중에 발생하는 오류는 예측하기 어려운 경우가 많고, 프로그램이 비정상 종료되면서 갑자기 멈춰 버립니다.
- 자바은 이러한 비정상 종료를 최대한 줄이기 위해 다양한 예외에 대한 처리방법을 가지고 있습니다.
- 예외 처리를 하는 목적은 프로그램이 비정상 종료 되는 것을 방지하기 위한 것 입니다.

오류와 예외

- **시스템 오류(error)**
  - 자바 가상 머신에서 발생
  - 시스템 오류의 예로는 사용 가능한 동적 메모리가 없는 경우나 스택 메모리의 오버플로가 발생한 경우 등이 있습니다.
  - 이러한 시스템 오류는 프로그램에서 제어할 수 없습니다.
- **예외(exception)**
  - 프로그램에서 제어할 수 있습니다.
  - 예를 들어 프로그램에서 파일을 읽어 사용하려는데 파일이 없는 경우, 네트워크로 데이터를 전송하려는데 연결이 안된 경우, 배열 값을 출력하는데 배열 요소가 없는 경우 등



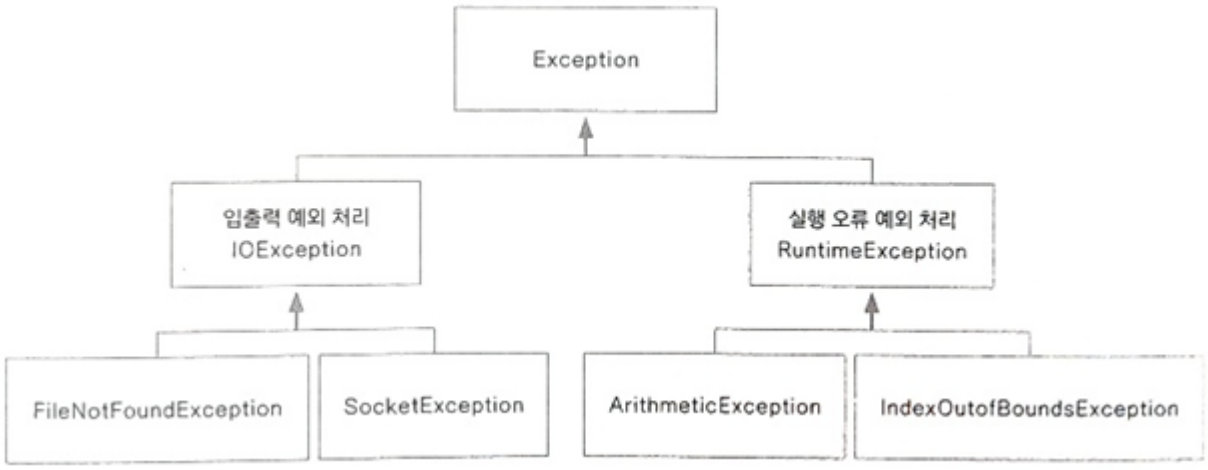
- 오류 클래스는 모두 **Throwable 클래스**에서 상속받습니다. **Error 클래스**의 하위 클래스는 시스템에서 발생하는 오류를 다루며 프로그램에서 제어하지 않습니다. 프로그램에서 제어하는 부분은 **Exception 클래스**와 그 하위에 있는 예외 클래스 입니다.

예외 클래스의 종류

- 프로그램에서 처리하는 예외 클래스의 최상위 클래스는 Exception 클래스 입니다. Exception 클래스의 내용을 살펴보면 다음과 같습니다.

```
Class Exception

java.lang.Object
        java.lang.Throwable
                java.lang.Exception
```

- 다음 그림은 Exception 하위 클래스 중 사용 빈도가 높은 클래스 위주로 계층도를 표현한 것 입니다.



- Exception 클래스 하위에는 이 외에도 많은 클래스가 있습니다. 계층도에서 IOException 클래스는 입출력에 대한 예외를 처리하고, RuntimeException은 프로그램 실행 중 발생할 수 있는 오류에 대한 예외를 처리합니다.
- 이클립스 같은 개발환경에서는 예외가 발생하면 대부분 처리하라는 컴파일 오류 메세지를 띄웁니다. 곧 배우게될 try ~ catch문을 사용하여 예외 처리를 해야 합니다.
- Exception 하위 클래스 중 **RuntimeException**은 try ~ catch문을 사용하여 **예외 처리를 하지 않아도 컴파일 오류가 나지 않습니다**. 곧 배우게될 예외 전가도 처리하지 않아도 컴파일 오류가 발생하지 않습니다.
- 예를 들어 RuntimeException 하위 클래스 중 ArithmeticException은 산술 연산 중 발생할 수 있는 예외, 즉 '0'으로 숫자 나누기와 같은 경우 발생하는 예외 입니다. 이러한 **컴파일러에 의해 체크되지 않는 예외는 프로그래머가 알아서 처리해야 하므로 주의해야 합니다.**

예외 처리하기

try ~ catch문

```
try {
        예외가 발생할 수 있는 코드 부분
} catch (처리할 예외 타입 e) {
        try 블록 안에서 예외가 발생했을 때 예외를 처리하는 부분
}
```

- try 블록에는 **예외가 발생할 가능성이 있는 코드를 작성**합니다.
- try 블록 안에 예외가 발생하면 바로 catch 블록이 수행됩니다.
- catch문의 괄호()안에 쓰는 예외 타입은 예외 상황에 따라 달라집니ㅏㄷ.

day11/exception/ArrayExceptionHandling.java - try ~ catch문 사용하

```
package day11.exception;
```

```
public class ArrayExceptionHandling {
	public static void main(String[] args) {
		int[] arr = new int[5];

		try {
			// 예외가 발생할 수 있으므로 try 블록에 작성
			for (int i = 0; i <= 5; i++) {
				arr[i] = i;
				System.out.println(arr[i]);
			}
		} catch (ArrayIndexOutOfBoundsException e) {
			// 예외가 발생하면 catch 블록 수행
			System.out.println(e);
			System.out.println("예외처리 부분");
		}
	}
}
```

실행결과

```
0
1
2
3
4
java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 5
예외처리 부분
```

- 배열에 저장하려는 값의 개수가 배열 범위를 벗어났기 때문에 예외가 발생한 것 입니다.
- 참고로 이 예외는 **RuntimeException의 하위 클래스**인 ArrayIndexOutOfBoundsException으로 처리하는에, 이 클래스는 **예외처리를 하지 않아도 컴파일 오류가 나지 않습니다.**
- 따라서 프로그래머가 직접 예외 처리를 하지 않으면 예외가 잡히지 않아서 예외가 발생하는 순간(이 예제에서는 i가 5가 되는 순간) 프로그램이 갑자기 멈춥니다. 그러므로 예외가 발생한 순간 프로그램이 비정상 종료되지 않도록 예외처리를 해주어야 합니다.
- **예외 처리는 프로그램이 비정상 종료되는 것을 방지할 수 있으므로 매주 중요**합니다.

컴파일러에 의해 예외가 처리되는 경우

- 자바에서 제공하는 많은 예외 클래스들은 컴파일러에 의해 처리됩니다. 이런 경우 자바에서는 예외 처리를 하지 않으면 컴파일 오류가 계속 남습니다.
- 컴파일오류가 발생하므로 적절한 예외처리가 이뤄지지 않는다면 컴파일이 발생하지 않아 실행되지 않습니다.

day11/exception/ExceptionHandling1.java

```
package day11.exception;

import java.io.FileInputStream;
import java.io.FileNotFoundException;

public class ExceptionHandling1 {
	public static void main(String[] args) {
```

```
                try {
                        FileInputStream fis = new FileInputStream("a.txt");
                } catch (FileNotFoundException e) {
                        System.out.println(e); // 예외 클래스의 toString() 메서드 호
출
                }

                System.out.println("여기도 수행됩니다."); // 정상 출력
        }
}
```

실행 결과

```
java.io.FileNotFoundException: a.txt (지정된 파일을 찾을 수 없습니다)
여기도 수행됩니다.
```

> 예외 처리를 한다고 해서 프로그램의 예외 상황 자체를 막을 수는 없습니다. 하지만 예외 처리를 하면 예외 상황을 알려 주는 메시지를 볼 수 있고, 프로그램이 비정상 종료되지 않고 계속 수행되도록 만들 수 있습니다.

try-catch-finally문

- try 블록이 수행되면 finally 블록은 어떤 경우에도 반드시 실행됩니다.
- try나 catch에 return문이 있어도 수행됩니다.

```
try {
        예외가 발생할 수 있는 부분
} catch (처리할 예외 타입 e) {
        예외를 처리하는 부분
} finally {
        항상 수행되는 부분
}
```

day11/exception/ExceptionHandling2.java

```
package day11.exception;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

public class ExceptionHandling2 {
        public static void main(String[] args) {
                FileInputStream fis = null;

                try {
                        fis = new FileInputStream("a.txt");
                } catch (FileNotFoundException e) {
                        System.out.println(e);
                        return;
                } finally {
                        if (fis != null) {
```

```
                                try {
                                        fis.close();  // 파일 입력 스트림 닫기
                                } catch (IOException e) {
                                        e.printStackTrace();
                                }
                        }
                        System.out.println("항상 수행됩니다.");
                }
                System.out.println("여기도 수행됩니다.");
        }
}
```

실행결과

```
java.io.FileNotFoundException: a.txt (지정된 파일을 찾을 수 없습니다)
항상 수행됩니다.
```

- 프로그램에서 사용항 리소스는 프로그램이 종료되면 자동으로 해제됩니다. 예를 들어 네트워크가 연결되었을 경우에 채팅 프로그램이 종료될 때 연결도 닫힙니다.
- 하지만 끝나지 않고 계속 수행되는 서비스 같은 경우에 리소스를 여러 번 반복해서 열기만 하고 닫지 않는다면 문제가 발생합니다. **시스템에서 허용하는 자원은 한계가 있습니다.**
- 사용한 시스템 리소스는 사용 후 반드시 close() 메서드로 닫아 주어야 합니다.
- 입력받은 파일이 없는 경우에 대해 try - catch 문을 사용해 FileNotFoundException예외 처리를 하였습니다. 프로그램을 실행하면 a.txt 파일이 없으므로 예외가 발생하여 catch 블록이 수행될 것 입니다.
- 예외를 출력하고 강제로 return을 해보았지만 **return문과 상관 없이 finally 블록이 수행**되어 '항상 수행됩니다.' 문장이 출력 됩니다.
- 파일 접근을 위해 열린 리소스는 **예외 발생과 상관없이 항상 닫혀야 하므로 finally 블록에 파일 리소스를 닫는 코드를 구현**하였습니다.

try-with-resources문

- 시스템 리소스를 사용하고 해제하는 코드는 다소 복잡합니다.
- JDK1.7 부터 try-with-resources문을 제공하여 close() 메서드를 명시적으로 호출하지 않아도 try 블록 내에서 열린 리소스를 자동으로 닫도록 만들 수 있습니다.
- try-with-resources 문법을 사용하려면 해당 리소스가 **AutoCloseable 인터페이스를 구현**해야 합니다.
- AutoCloseable 인터페이스에는 close() 메서드가 있고 이를 구현한 클래스는 close()를 명시적으로 호출하지 않아도 close() 메서드 부분이 호출됩니다.

FileInputStream의 JavaDoc 예시

Module java.base
Package java.io

Class FileInputStream

java.lang.Object
    java.io.InputStream
        java.io.FileInputStream

All Implemented Interfaces:
Closeable, AutoCloseable

- FileInputStream 클래스는 Closeable과 AutoCloseable 인터페이스를 구현했습니다.
- JDK1.7부터는 try-with-resources 문법을 사용하면 FileInputStream을 사용할 때 close()를 명시적으로 호출하지 않아도 정상적인 경우와 예외가 발생한 경우 모두 close()메서드가 호출됩니다.
- FileInputStream 이외에 네트워크(Socket)와 데이터베이스(Connection) 관련 클래스도 AutoCloseable 인터페이스를 구현하고 있습니다.

> AutoCloseable 인터페이스를 구현한 클래스가 무엇이 있는지 궁금하다면 JavaDoc에서 AutoCloseable Interface를 찾아보세요.

day11/exception/AutoCloseObj.java

```
package day11.exception;

public class AutoCloseObj implements AutoCloseable {
        @Override
        public void close() throws Exception {
                System.out.println("리소스가 close() 되었습니다.");
        }
}
```

- AutoCloseable 인터페이스는 반드시 close() 메서드를 구현해야 합니다.

day11/exception/AutoCloseTest.java

```
package day11.exception;

public class AutoCloseTest {
        public static void main(String[] args) {
                try (AutoCloseObj obj = new AutoCloseObj()) { // 사용할 리소스 선언

                } catch(Exception e) {
                        System.out.println("예외 부분 입니다.");
                }
        }
}
```

실행결과

```
리소스가 close() 되었습니다.
```

- try-with-resources문을 사용할 때 try문의 괄호() 안에 리소스를 선언합니다.
- 이 예제는 예외가 발생하지 않고 정상 종료 되는데 출력 결과를 보면 close() 메서드가 호출되어 **리소스가 close() 되었습니다**. 문장이 출력 됩ㄴ디ㅏ.
- 리소스를 여러개 생성해야 한다면 세미 콜론(;)으로 구분합니다.

```
try(A a = new A(); B b = new B()) {
        ...
} catch(Exception e) {
        ...
}
```

day11/exception/AutoCloseObjTest.java

```
package day11.exception;

public class AutoCloseObjTest {
        public static void main(String[] args) {
                try (AutoCloseObj obj = new AutoCloseObj()) { // 사용할 리소스 선언
                        throw new Exception(); // 강제 예외 발생
                } catch(Exception e) {
                        System.out.println("예외 부분 입니다.");
                }
        }
}
```

```
실행결과

리소스가 close() 되었습니다.
예외 부분 입니다.
```

- 강제로 예외를 발생시키면 catch 블록이 수행됩니다.
- 출력 결과를 보면 리소스의 close() 메서드가 먼저 호출되고 예외 블록 부분이 수행되는 것을 알 수 있습니다.
- 이처럼 try-with-resources문을 사용하면 close() 메서드를 명시적으로 호출하지 않아도 정상 종료된 경우와 예외가 발생한 경우 모두 리소스가 잘 해제 됩니다.

예외 처리 미루기

예외 처리를 미루는 throws 사용하기

- 예외를 해당 메서드에서 처리하지 않고 미룬 후 메서드를 호출하여 사용하는 부분에서 예외를 처리하는 방법

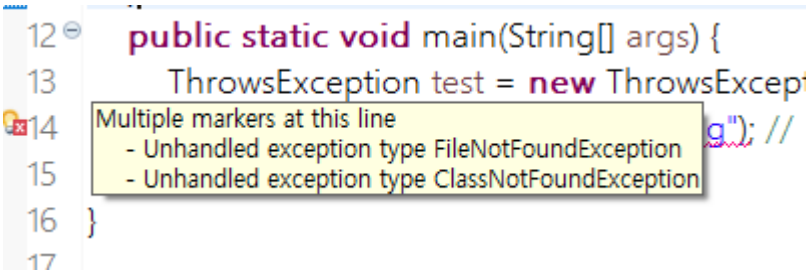
day11/exception/ThrowsException.java

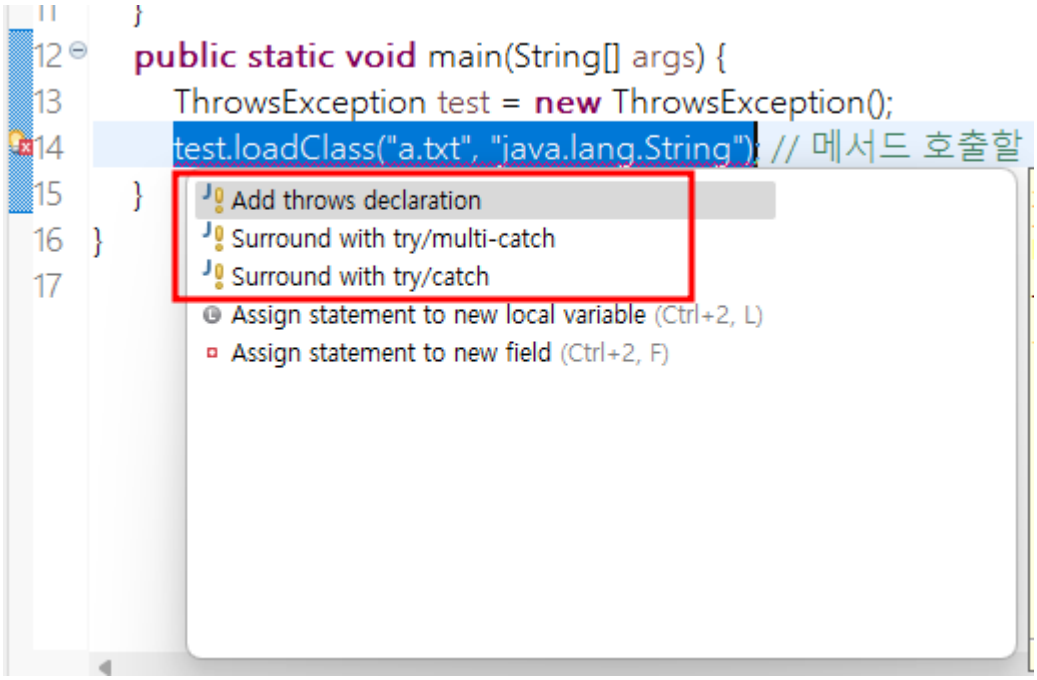
```
package day11.exception;
```

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;

public class ThrowsException {
	public Class loadClass(String fileName, String className) throws
FileNotFoundException, ClassNotFoundException {
		FileInputStream fis = new FileInputStream(fileName); //
FileNotFoundException 발생 가능
		Class c = Class.forName(className); // ClassNotFoundException 발생
가능

		return c;
	}
	public static void main(String[] args) {
		ThrowsException test = new ThrowsException();
		test.loadClass("a.txt", "java.lang.String"); // 메서드 호출할 때 예
외처리함
	}
}
```

throws를 활용하여 예외처리 미루기

- 예외를 처리하지 않고 미룬다고 선언하면, 그 메서드를 호출하여 사용하는 부분에서 예외 처리를 해야 합니다.





- **Add throws declaration**
  - main() 함수 선언 부분에 throws FileNotFoundException, ClassNotFoundException을 추가하고 예외 처리를 미룬다는 뜻 입니다.

    - main() 함수에서 미룬 예외 처리는 main() 함수를 호출하는 자바 가상 머신으로 보내집니다.즉, 예외를 처리하는 것이 아니라 대부분의 프로그램이 비정사아 종료 됩니다.

- **Surround with try/multi-catch** - 여러 예외를 한꺼번에 처리하기

```
public static void main(String[] args) {
	ThrowsException test = new ThrowsException();
	try {
		test.loadClass("a.txt", "java.lang.String");
	} catch (FileNotFoundException | ClassNotFoundException e) {
		// TODO Auto-generated catch block
		e.printStackTrace();
	}
}
```

- **Surround with try/catch** - 예외를 상황마다 처리하기

```
...
public static void main(String[] args) {
	ThrowsException test = new ThrowsException();
	try {
		test.loadClass("a.txt", "java.lang.String");
	} catch (FileNotFoundException e) {
		// TODO Auto-generated catch block
		e.printStackTrace();
	} catch (ClassNotFoundException e) {
		// TODO Auto-generated catch block
		e.printStackTrace();
	}
}
...
```

- 예외가 발생한 메서드에서 그 예외를 바로 처리할 것인지, 아니면 미루어서 그 메서드를 호출하여 사용하는 부분에서 처리할 것인지는 만들고자 하는 프로그램 상황에 따라 다를 수 있습니다.

- 만약 어떤 메서드가 다른 여러 코드에서 호출되어 사용된다면 메서드를 호출하는 부분에서 예외처리를 하도록 미루는 것이 합리적입니다.

다중 예외 처리

- 어떤 예외가 발생할지 미리 알수 없지만 모든 예외 상황을 처리하고자 한다면 맨 마지막 부분에 Exception 클래스를 활용하여 catch 블록을 추가합니다.

day11/exception/ThrowsException.java

```
package day11.exception;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
```

```
public class ThrowsException {
	public Class loadClass(String fileName, String className) throws
FileNotFoundException, ClassNotFoundException {
		FileInputStream fis = new FileInputStream(fileName); //
FileNotFoundException 발생 가능
		Class c = Class.forName(className); // ClassNotFoundException 발생
가능
		return c;
	}
	public static void main(String[] args) {
		ThrowsException test = new ThrowsException();
		try {
			test.loadClass("a.txt", "java.lang.String");
		} catch (FileNotFoundException e) {
			e.printStackTrace();
		} catch (ClassNotFoundException e) {
			e.printStackTrace();
		} catch (Exception e) { // Exception  클래스로 그 외 예외 상황 처리
			e.printStackTrace();
		}
	}
}
```

- Exception 클래스는 모든 예외 클래스의 최상위 클래스 입니다. 따라서 다른 catch 블록에서 선언한 것 이외의 예외가 발생하더라도 Exception 클래스로 자동 형 변환 됩니다(다형성).
- 가장 처음 Exception이 catch 구간에 있다면 모든 예외가 이 구간으로 유입이 되어 적절한 처리가 되지 않습니다. 따라서 기본 예외 처리를 하는 **Exception 클래스 블록은 여러 예외 처리 블록의 가장 아래 놓여야 합니다.**

사용자 정의 예외

- 사용자 정의 예외 클래스를 구현할 때는 기존 JDK에서 제공하는 예외 클래스 중 가장 유사한 클래스를 상속받는 것이 좋습니다.
- 유사한 예외 클래스를 잘 모르겠다면 **가장 상위 클래스인 Exception 클래스**를 상속받으시면 됩니다.

day11/exception/IDFormatException.java

```
package day11.exception;

public class IDFormatException extends Exception {
	// 생성자의 매개변수로 예외 상황 메시지를 받음
	public IDFormatException(String message) {
		super(message);
	}
}
```

- 위 코드는 Exception 클래스에서 상속받아 구현했습니다.
- 예외 상황 메세지를 생성자에 입력 받습니다.

- Exception 클래스에서 메세지 생성자, 멤버 변수와 메서드를 이미 제공하고 있으므로 super(message)를 사용하여 메시지를 설정합니다.
- 나중에 getMessage() 메서드를 호출하면 메시지 내용을 볼 수 있습니다.

day11/exception/IDFormatTest.java

```
package day11.exception;

public class IDFormatTest {
	private String userID;

	public String getUserID() {
		return userID;
	}

	public void setUserID(String userID) throws IDFormatException {
		if (userID == null) {
			// 강제로 예외 발생
			throw new IDFormatException("아이디는 null일 수 없습니다.");
		} else if (userID.length() > 0 || userID.length() > 20) {
			// 강제로 예외 발생
			throw new IDFormatException("아이디는 8자 이상 20자 이하로
쓰세요");
		}
		this.userID = userID;
	}

	public static void main(String[] args) {
		IDFormatTest test = new IDFormatTest();

		String userID = null; // 아이디 값이 null인 경우
		try {
			test.setUserID(userID);
		} catch (IDFormatException e) {
			System.out.println(e.getMessage());
		}

		userID = "1234567"; // 아이디 값이 8자 이하인 경우
		try {
			test.setUserID(userID);
		} catch (IDFormatException e) {
			System.out.println(e.getMessage());
		}
	}
}
```

실행결과

```
아이디는 null일 수 없습니다.
아이디는 8자 이상 20자 이하로 쓰세요
```

- 여기에서 발생하는 예외는 자바에서 제공하는 예외가 아니므로 예외 클래스를 직접 생성하여 예외를 발생시켜야 합니다.

- 예외 메시지를 생성자에 넣어 예외 클래스를 생성한 후 **throw문으로 직접 예외를 발생**시킵니다.

java.lang 패키지

- java.lang 패키지에는 기본적으로 많이 사용하는 클래스들이 포함되어 있습니다.
- 예를 들면 String, Integer와 같은 클래스는 java.lang 패키지에 속해 있고, String 클래스의 전체 이름은 java.lang.String이고 Integer 클래스의 전체 이름은 java.lang.Integer입니다.
- 외부 패키지에 선언한 클래스를 사용할 때는 import문으로 클래스가 어느 패키지에 속해 있는지 선언해야 하나 String 클래스를 쓰면서 import java.lang.String; 문장을 쓴 적이 없습니다.
- java.lang 패키지는 컴파일 할 때 import java.lang.*; 문장이 자동으로 추가되어 java.lang 패키지 하위 클래스를 모두 사용할 수 있으므로 직접 써 줄 필요가 없습니다.
- 즉, import문을 직접 쓰지 않아도 java.lang 패키지의 모든 하위 클래스를 참조할 수 있습니다.

Object 클래스

- Object 클래스는 **모든 자바 클래스의 최상위 클래스** 입니다.
- 다시 말하면 모든 클래스는 Object 클래스로 부터 상속을 받습니다.
- 우리가 클래스를 만들 때 Object를 상속받는 코드를 작성한 적이 없는데, Object 클래스의 상속을 의미하는 **extends Object가 컴파일 과정에서 자동으로 쓰입니다.**
- 코드를 작성할 때

```
class Student {
        int studentID;
        String studentName;
}
```

- 컴파일러가 변환

```
class Student extends Object {
        int studentID;
        String studentName;
}
```

String 클래스 JavaDoc 예시

Module java.base
Package java.lang

Class String

java.lang.Object
    java.lang.String

- **java.lang.Object** : 최상위 클래스
- java.lang.String : JavaDoc에서 검색한 클래스
- JavaDoc을 보니 String 클래스 역시 Object 클래스를 상속받았음을 알 수 있습니다.
- **모든 클래스가 Object 클래스를 상속받았으므로 Object의 메서드를 사용할 수 있고, 재정의 할 수도 있고, Object형으로 변환할 수 도 있습니다.**
- 자바로 프로그래밍을 하다 보면 클래스가 Object형으로 변환되는 경우도 있고, Object에서 원래 클래스형으로 다운 캐스팅되는 경우도 있습니다.

Object 클래스에 정의된 메서드

메서드	설명
String toString()	객체를 문자열로 표현하여 반환합니다. 재정의하여 객체에 대한 설명이나 특정 멤버 변수 값을 반환합니다.
boolean equals(Object obj)	두 인스턴스가 동일한지 여부를 반환합니다. 재정의하여 논리적으로 동일한 인스턴스임을 정의할 수 있습니다.
int hashCode()	객체의 해시 코드 값을 반환합니다.
Object clone()	객체를 복제하여 동일한 멤버 변수 값을 가진 새로운 인스턴스를 생성합니다.
Class getClass()	객체의 Class 클래스를 반환합니다.
void finalize()	인스턴스가 힙 메모리에서 제거될 때 가비지 컬렉터(GC)에 의해 호출되는 메서드입니다. 네트워크 연결 해제, 열려 있는 파일 스트림 해제 등을 구현합니다.
void wait()	멀티스레드 프로그램에서 사용하는 메서드입니다. 쓰레드를 **기다리는 상태**(non runnable)로 만듭니다.
void notify()	wait() 메서드에 의해 기다리고 있는 쓰레드(non runnable 상태)를 **실행 가능한 상태**(runnable)로 가져옵니다.

- Object 메서드 중에는 재정의 할 수 있는 메서드도 있고 그렇지 않은 메서드도 있습니다.
- Object 메서드 중에서 final 예약어로 선언된 메서드는 자바 쓰레드에서 사용하거나 클래스를 로딩하는 등 자바 가상 머신과 관련된 메서드이기 때문에 재정의할 수 없습니다.|

toString() 메서드

- Object 클래스에서 기본으로 제공하는 toString() 메서드는 이름 처럼 객체 정보를 문자열(String)로 바꾸어 줍니다.
- Object 클래스를 상속받은 모든 클래스는 toString()을 재정의할 수 있습니다.
- String이나 Integer 등 여러 JDK 클래스에는 toString() 메서드가 이미 재정의 되어 있습니다.

Object 클래스의 toString() 메서드

- toString() 메서드는 인스턴스 정보를 문자열로 변환하는 메서드 입니다.
- toString() 메서드의 원형은 생성된 인스턴스 클래스의 이름과 주소 값을 보여줍니다.

day11/object/ToStringEx.java

```
package day11.object;

class Book {
	int bookNumber;
	String bookTitle;

	Book(int bookNumber, String bookTitle) {
		this.bookNumber = bookNumber;
		this.bookTitle = bookTitle;
	}
}

public class ToStringEx {
	public static void main(String[] args) {
		Book book1 = new Book(200, "개미");

		 // 클래스 정보(클래스 이름, 주소 값)
		System.out.println(book1);

		// toString())) 메서드로 인스턴스 정보(클래스 이름, 주소 값)를 보여
줌
		System.out.println(book1.toString());
	}
}
```

실행결과

```
day11.object.Book@5e91993f
day11.object.Book@5e91993f
```

- System.out.println() 출력문에 참조 변수를 넣으면 인스턴스 정보가 출력되는데, 이 때 자동으로 호출되는 메서드가 toString() 입니다.
- 여기에서 호출되는 toString()은 Book클래스의 메서드가 아닌 **Object 클래스의 메서드** 입니다.
- Object 클래스의 toString()메서드 원형은 다음과 같습니다.

```
getClass().getName() + '@' + Integer.toHexString(hashCode())
```

- **클래스 이름@해시 코드 값** 입니다. 즉, 클래스의 이름과 16진수 해시 코드 값이 출력됩니다.

String과 Integer클래스의 toString() 메서드

```
String str = new String("test");
System.out.println(str); // test 출력됨
Integer i1 = new Integer(100);
System.out.println(i1); // 100 출력됨
```

- 두 클래스의 출력 결과는 **클래스이름@해시코드 값**이 아니라 문자열 값 test, 정수값 100이 각각 출력됩니다.
- 그 이유는 String과 Integer 클래스는 toString() 메서드를 미리 재정의해 두었기 때문입니다.
- JDK에서 제공하는 클래스 중에는 toString() 메서드를 미리 재정의한 클래스가 많습니다.
- toString() 메서드가 재정의된 클래스는 '클래스의 이름@해시코드 값'을 출력하는 toString() 메서드의 원형이 아닌 **재정의된 메서드**가 호출되는 것입니다.

day11/object/ToStringEx.java

```
package day11.object;

class Book {
	int bookNumber;
	String bookTitle;

	Book(int bookNumber, String bookTitle) {
		this.bookNumber = bookNumber;
		this.bookTitle = bookTitle;
	}

	@Override
	public String toString() { // toString() 메서드 재정의
		return bookTitle + "," + bookNumber;
	}
}

public class ToStringEx {
	public static void main(String[] args) {
		Book book1 = new Book(200, "개미");

		 // 클래스 정보(클래스 이름, 주소 값)
		System.out.println(book1);

		// toString())) 메서드로 인스턴스 정보(클래스 이름, 주소 값)를 보여
줌
		System.out.println(book1.toString());
	}
}

실행결과

개미,200
개미,200
```

equals() 메서드

- equals() 메서드의 원래 기능은 두 인스턴스의 주소 값을 비교하여 boolean 값(true/false)을 반환해 주는 것입니다.
- 주소 값이 같다면 당연히 같은 인스턴스 입니다. 그런데 서로 다른 주소 값을 가질 때도 같은 인스턴스라고 정의할 수 있는 경우가 있습니다.

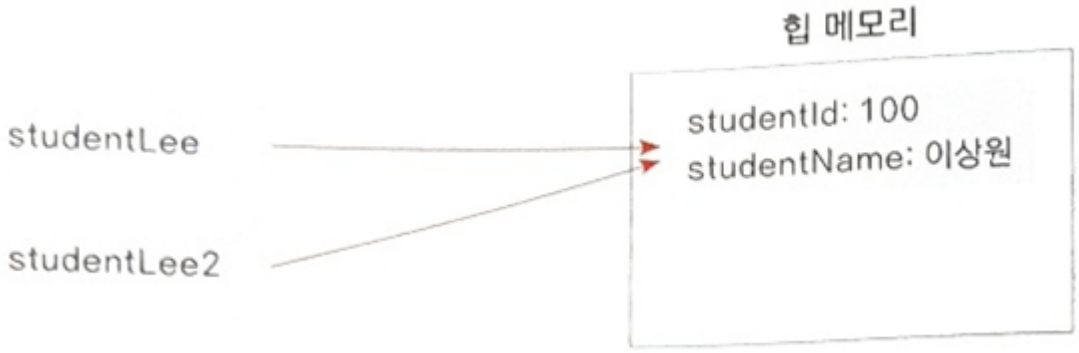
- 물리적 동일성(인스턴스 메모리 주소가 같음)뿐 아니라 **논리적 동일성(논리적으로 두 인스턴스가 같음)을 구현할 때도 equals() 메서드를 재정의**하여 사용합니다.

Object 클래스의 equals() 메서드

- 두 인스턴스가 물리적으로 같다는 것은, 두 인스턴스의 주소 값이 같은 경우를 말합니다. 즉, 두 변수가 같은 메모리 주소를 가리키고 있다는 뜻 입니다.

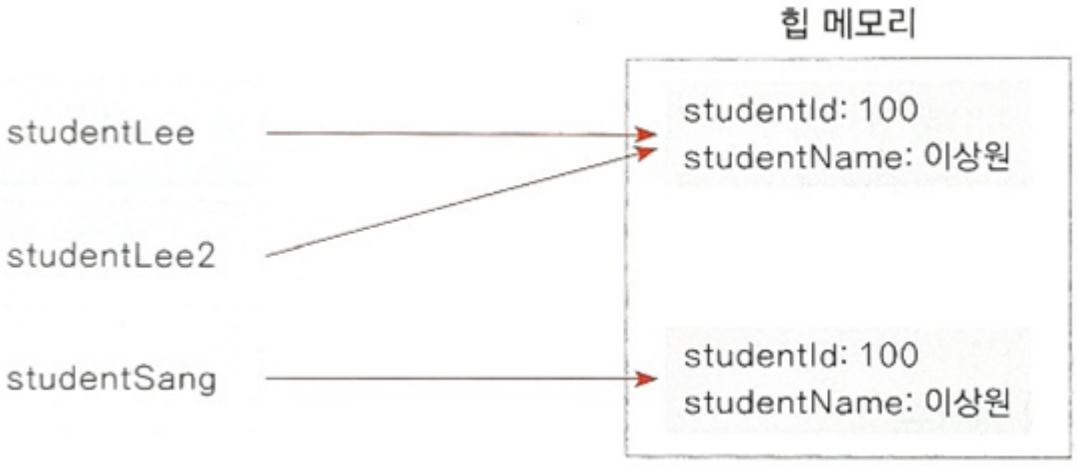
```
Student studentLee = new Student(100, "이상원");
Student studentLee2 = studentLee; // 주소 복사
```

- 두 변수는 다음 그림과 같이 동일한 인스턴스를 가리킵니다. 이 때 equals()메서드를 이용해 두 변수를 비교하면 동일하다는 결과가 나옵니다.




```
Student studentLee = new Student(100, "이상원");
Student studentLee2 = studentLee;
Student studentSang = new Student(100, "이상원");
```

- 위 코드를 표현한 그림은 다음과 같습니다.



- studentLee, studentLee2가 가리키는 인스턴스와 studentSang이 가리키는 인스턴스는 서로 다른 주소를 가지고 있지만, 저장된 학생의 정보는 같습니다. 이런 경우 논리적으로는 studentLee, studentLee2와 studentSang을 같은 학생으로 처리하는 것이 맞을 것 입니다.

day11/object/EqualsTest.java

```
package day11.object;

class Student{

	int studentId;
	String studentName;

	public Student(int studentId, String studentName){
		this.studentId = studentId;
		this.studentName = studentName;
	}

	public String toString(){
		return studentId + "," + studentName;
	}
}

public class EqualsTest {

	public static void main(String[] args) {

		Student studentLee = new Student(100, "이상원");
		Student studentLee2 = studentLee; // 주소 복사
		Student studentSang = new Student(100, "이상원");

		// 동일한 주소의 두 인스턴스 비교
		if(studentLee == studentLee2) // == 기호로 비교
			System.out.println("studentLee와 studentLee2의 주소는 같습
니다.");
		else
			System.out.println("studentLee와 studentLee2의 주소는 다릅
니다.");

		if(studentLee.equals(studentLee2)) // equals() 메서드로 비교
			System.out.println("studentLee와 studentLee2는 동일합니
다.");
		else
			System.out.println("studentLee와 studentLee2는 동일하지 않
습니다.");


		// 동일인이지만 인스턴스의 주소가 다른 경우
		if(studentLee == studentSang) // == 기호로 비교
			System.out.println("studentLee와 studentSang의 주소는 같습
니다.");
		else
			System.out.println("studentLee와 studentSang의 주소는 다릅
니다.");

		if(studentLee.equals(studentSang)) // equals() 메서드로 비교ㅕ
			System.out.println("studentLee와 studentSang은 동일합니
다.");
		else
			System.out.println("studentLee와 studentSang은 동일하지 않
습니다.");
```

```
        }
}
```

실행결과

```
studentLee와 studentLee2의 주소는 같습니다.
studentLee와 studentLee2는 동일합니다.
studentLee와 studentSang의 주소는 다릅니다.
studentLee와 studentSang은 동일하지 않습니다.
```

- Object의 equals() 메서드의 원래 기능은 두 인스턴스의 주소를 비교하는 것입니다. 따라서 같은 주소인 경우만 equals() 메서드의 결과가 true가 됩니다.
- 두 인스턴스가 있을 때 ==는 단순히 물리적으로 같은 메모리 주소인지 여부를 확인할 수 있고, Object의 equals() 메서드는 재정의를 하여 논리적으로 같은 인스턴스인지(메모리 주소가 다르더라도 같은 학생인지) 확인하도록 구현할 수 있습니다.

String과 Integer 클래스의 equals() 메서드

- JDK에서 제공하는 String 클래스와 Integer 클래스에는 equals() 메서드가 이미 재정의 되어 있습니다.

day11/object/StringEquals.java

```
package day11.object;

public class StringEquals {
        public static void main(String[] args) {

                String str1 = new String("abc");
                String str2 = new String("abc");

                System.out.println(str1 == str2);  // 두 스트링 인스턴스의 주소 값은 다름
                System.out.println(str1.equals(str2)); // String 클래스의 equals 메소드가 재정의 됨

                Integer i1 = new Integer(100); // Integer 보다는 Integer.valueOf 사용 -> 인스턴스 주소는 같은 값에 대해 동일
                Integer i2 = new Integer(100);

                System.out.println(i1 == i2);   // 두 정수 인스턴스의 주소 값은 다름
                System.out.println(i1.equals(i2)); // Integer 클래스의 equals 메소드가 재정의 됨
        }
}
```

실행결과

```
false
true
false
true
```

- 코드의 내용을 보면 str1과 str2는 서로 다른 인스턴스를 가리키기 때문에 str1 == str2의 결과는 false 입니다. 하지만 String 클래스의 equals() 메서드는 같은 문자열의 경우 true를, 그렇지 않은 경우 false를 반환하도록 **재정의**되어 있습니다.
- 두 문자열은 "abc"로 같은 값을 가지므로 str1.equals(str2)의 반환 값은 true입니다.
- Integer 클래스의 경우도 정수 값이 같은 경우 true를 반환하도록 equals() 메서드가 재정의되어 있습니다.

day11/object/EqualsTest.java - Student 클래스에서 equals() 메서드 직접 재정의 하기

```
package day11.object;

class Student{
        ...
        @Override
        public boolean equals(Object obj) {
                if(obj instanceof Student){
                        Student std = (Student)obj;
                        // 학생의 학번이 같으면 true 반환
                        if( studentId == std.studentId)
                                return true;
                        else return false;
                }
                return false;
        }
}
...

실행결과

studentLee와 studentLee2의 주소는 같습니다.
studentLee와 studentLee2는 동일합니다.
studentLee와 studentSang의 주소는 다릅니다.
studentLee와 studentSang은 동일합니다.
```

- equals() 메서드를 재정의하였습니다.
- equals() 메서드의 매개변수는 Object형 입니다. 비교될 객체가 Object형으로 전달되면 instanceof를 사용하여 매개변수의 원래 자료형이 Student인지 확인합니다.
- this의 학번과 매개변수로 전달된 객체의 학번이 같으면 true를 반환합니다.
- equals() 메서드를 재정의한 후 출력 결과를 보면 studentLee와 studentSang은 서로 다른 메모리 주소에 존재하는 인스턴스이므로 == 연산의 결과 값은 false를 반환하지만, 학번이 같으므로 equals()는 true를 반환합니다.

hashCode() 메서드

- 해시(hash)는 정보를 저장하거나 검색할 때 사용하는 자료구조 입니다.
- 정보를 어디에 저장할 것인지, 어디서 가져올 것인지 해시 함수를 사용하여 구현합니다.

- 해시 함수는 객체의 특정 정보(키 값)를 매개변수 값으로 넣으면 그 객체가 저장되어야 할 위치나 저장된 해시 테이블 주소(위치)를 반환합니다.
- 따라서 객체 정보를 알면 해당 객체의 위치를 빠르게 검색할 수 있습니다.
- 해시 함수(java.util.Objects.hash(key))는 개발하는 프로그램에 따라 다르게 구현됩니다.
- 자바에서는 인스턴스를 힙 메모리에 생성하여 관리할 때 해시 알고리즘을 사용합니다.

```
hashCode = java.util.Objects.hash(key); // 객체의 해시 코드 값(메모리 위치 값)이 반환됨
```

- 자바에서는 두 인스턴스가 같다면 hashCode() 메서드에서 반환하는 해시 코드 값이 같아야 합니다. 따라서 논리적으로 같은 두 객체도 같은 해시 코드 값을 반환하도록 hashCode() 메서드를 재정의해야 합니다. 다시 말해, equals() 메서드를 재정의 했다면 hashCode() 메서드도 재정의해야 합니다.

String과 Integer 클래스의 hashCode() 메서드

- String 클래스와 Integer 클래스의 equals() 메서드는 재정의되어 있습니다. 마찬가지로 hashCode() 메서드도 함께 재정의되어 있습니다.

```
package day11.object;

public class HashCodeTest {
	public static void main(String[] args) {

		String str1 = new String("abc");
		String str2 = new String("abc");

		// abc 문자열의 해시 코드 값 출력
		System.out.println(str1.hashCode());
		System.out.println(str2.hashCode());

		Integer i1 = new Integer(100);
		Integer i2 = new Integer(100);

		// Integer(100)의 해시코드 값 출력
		System.out.println(i1.hashCode());
		System.out.println(i2.hashCode());
	}
}
```

실행결과

```
96354
96354
100
100
```

- String 클래스는 같은 문자열을 가진 경우 즉, equals() 메서드의 결과 같이 true인 경우 hashCode() 메서드는 동일한 해시 코드 값을 반환합니다.

- Integer 클래스의 hashCode() 메서드는 정수 값을 반환하도록 재정의되어 있습니다.

day11/object/EqualsTest.java - Student 클래스에서 hashCode()메서드 재정의

```
package day11.object;

class Student{
	...
	@Override
	public int hashCode() {
		return studentId;
	}
}

public class EqualsTest {
	public static void main(String[] args) {
		...
		System.out.println("studentLee의 hashCode :"
+studentLee.hashCode());
		System.out.println("studentSang의 hashCode :"
+studentSang.hashCode());

		System.out.println("studentLee의 실제 주소값 :"+
System.identityHashCode(studentLee));
		System.out.println("studentSang의 실제 주소값 :"+
System.identityHashCode(studentSang));
	}
}

실행결과

studentLee와 studentLee2의 주소는 같습니다.
studentLee와 studentLee2는 동일합니다.
studentLee와 studentSang의 주소는 다릅니다.
studentLee와 studentSang은 동일합니다.
studentLee의 hashCode :100
studentSang의 hashCode :100
studentLee의 실제 주소값 :1586600255
studentSang의 실제 주소값 :474675244
```

- 출력 결과를 보면 studentLee, studentSang은 학번이 같기 때문에 논리적으로 같은지 확인하는 equals() 메서드 출력 값이 true 입니다.
- 또한 같은 해시 코드 값을 반환하고 있습니다.
- hashCode() 메서드를 재정의했을 때 실제 인스턴스 주소 값은 System.indentityHashCode() 메서드를 사용하면 알 수 있습니다. studentLee와 studentSang은 실제 메모리 주소 값은 다릅니다. 즉, 논리적으로는 같지만 실제로는 다른 인스턴스 입니다.

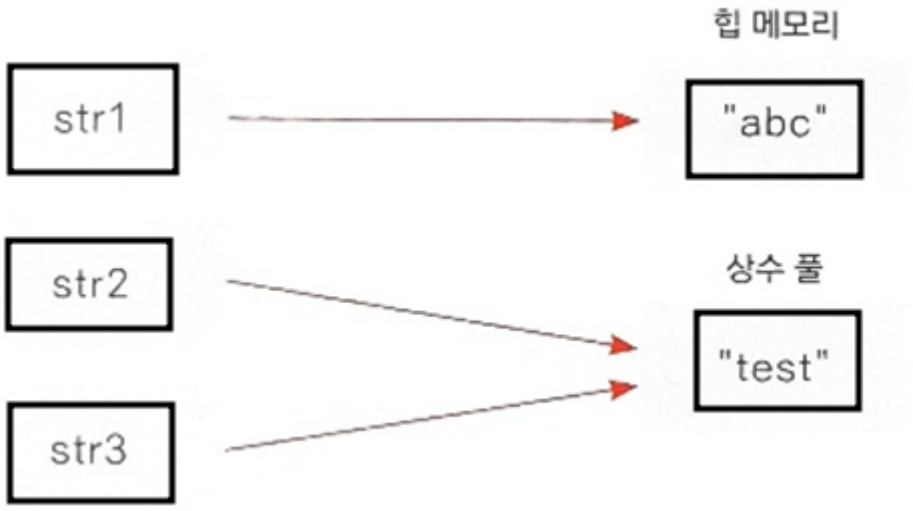
String 클래스

String을 선언하는 두 가지 방법

- 자바에서는 문자열을 사용할 수 있도록 String 클래스를 제공합니다.
- String을 사용할 때 문자열을 생성자의 매개변수로 하여 생성하는 방식과 이미 생성된 문자열 상수를 가리키는 방식이 있습니다.

```
String str1 = new String("abc"); // 생성자의 매개변수로 문자열 생성
String str2 = "test"; // 문자열 상수를 가리키는 방식
```

- 내부적으로 두 가지 방식은 큰 차이가 있습니다.
- new 예약어를 사용하여 객체를 생성하는 경우는 "abc" 문자열을 위한 메모리가 할당되고 새로운 객체가 생성됩니다.
- 하지만 str2 = "test"와 같이 생성자를 이용하지 않고 바로 문자열 상수를 가리키는 경우에는 str2가 기존에 만들어져 있던 "test"라는 문자열 상수의 메모리 주소를 가리키게 됩니다.
- 따라서 String str3 = "test" 코드를 작성하면 str2와 str3는 주소 값이 같게 됩니다.



- test나 10, 20 등과 같이 프로그램에서 사용되는 상수값을 저장하는 공간을 **상수 풀(constant pool)** 이라고 합니다. [상수와 리터럴 참조](https://github.com/yonggyo1125/curriculum300H/tree/main/1.JAVA(84%EC%8B%9C%EA%B0%84)/1%EC%9D%BC%EC%B0%A8(3h)%20-%20%EC%8B%A4%EC%8A%B5%ED%99%98%EA%B2%BD%20%EA%B5%AC%EC%B6%95%2C%EB%B3%80%EC%88%98%EC%99%80%20%EC%9E%90%EB%A3%8C%ED%98%95#%EC%83%81%EC%88%98%EC%99%80-%EB%A6%AC%ED%84%B0%EB%9F%B4)

day11/string/StringTest1.java

```
package day11.string;

public class StringTest1 {
        public static void main(String[] args) {

                String str1 = new String("abc");
                String str2 = new String("abc");

                System.out.println(str1 == str2);    //false
                System.out.println(str1.equals(str2)); //true
```

```
                String str3 = "abc";
                String str4 = "abc";

                System.out.println(str3 == str4);  //true
                System.out.println(str3.equals(str4)); //true
        }
}
```

실행결과

```
false
true
true
true
```

- 문자열 상수를 바로 가리키는 경우에는 주소 값이 같음을 알 수 있습니다.

String 클래스의 final char[] 변수

- 다른 프로그래밍 언어는 문자열을 구현할 때 일반적으로 char[] 배열을 사용합니다.
- 자바는 String클래스를 제공해 char[] 배열을 직접 구현하지 않고도 편리하게 문자열을 사용할 수 있습니다.
  (JDK12 이후 byte[] 배열로 변경됨)

```
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence,
               Constable, ConstantDesc {

        ...
    @Stable
    private final byte[] value;
        ...
```

- String 클래스의 구현 내용을 보면 private final char value[]라고 선언된 char형 배열이 있습니다.
- 프로그램에서 String s = new String("abc")라고 쓰면 abc는 String 클래스의 value 변수에 저장됩니다. 그런데 이 변수는 **final로 선언**되어 있습니다.
- **final은 문자열을 변경할 수 없다**는 뜻입니다. 따라서 **한번 생성된 문자열은 변경되지 않습니다.**
- 이런 문자열의 특징을 **문자열은 불변(immutable)한다**라고 합니다.
- 프로그램에서 두 개의 문자열을 연결하면 둘 중에 하나의 문자열이 변경되는 것이 아니라 두 문자열이 연결된 새로운 문자열이 생성됩니다.

day11/string/StringTest2.java

```
package day11.string;

public class StringTest2 {
        public static void main(String[] args) {

                String javaStr = new String("java");
```

```
                String androidStr = new String("android");
                System.out.println(javaStr);
                System.out.println("처음 문자열 주소 값: "+
System.identityHashCode(javaStr));

                javaStr = javaStr.concat(androidStr); //java 와 android 문자열의 연
결

                System.out.println(javaStr);
                System.out.println("연결된 문자열 주소 값: "
+System.identityHashCode(javaStr));
        }
}
```

실행결과

```
java
처음 문자열 주소 값: 1651191114
javaandroid
연결된 문자열 주소 값: 1586600255
```

- 두 개의 문자열 "java"와 "android"를 생성했습니다. 그리고 두 문자열을 연결하는 concat() 메서드를 호출했습니다.
- javaStr 변수 출력 결과를 보면 "javaandroid"로 연결되어 잘 출력되고 있습니다.
- 이 결과만 보면 "java" 문자열에 "android"문자열이 연결된 것 같지만, **문자열은 불편(immutable)하므로 javaStr 변수 값 자체가 변하는 것이 아니라 새로운 문자열이 생성**된 것입니다.

StringBuffer와 StringBuilder 클래스 활용하기

- 프로그램을 만들다 보면 문자열을 변경하거나 연결해야 할 때가 많습니다.
- 그런데 String 클래스는 한번 생성되면 그 내부의 문자열이 변경되지 않기 때문에 String 클래스를 사용하여 문자열을 계속 연결하거나 변경하는 프로그램을 작성하면 메모리가 많이 낭비됩니다.
- 이 문제를 해결하는 것이 바로 **StringBuffer**와 **StringBuilder** 클래스 입니다.
- StringBuffer와 StringBuilder는 **내부에 변경가능한 (final이 아닌) char[](JDK12에서 byte[]로 변경)를 변수**로 가지고 있습니다.
- 이 두 클래스를 사용하여 문자열을 연결하면 기존에 사용하던 char[] 배열이 확장되므로 추가 메모리를 사용하지 않습니다.

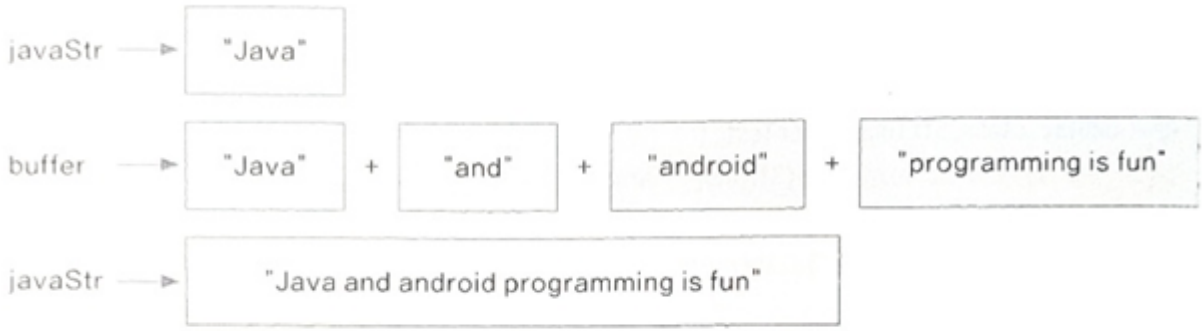
- 따라서 문자열을 연결하거나 변경할 경우 두 클래스 중 하나를 사용하면 됩니다.
- 두 클래스의 차이는 여러 작업(쓰레드)이 동시에 문자열을 변경하려 할 때 문자열이 안전하게 변경되도록 보장해 주는가 그렇지 않은가의 차이 입니다.
- StringBuffer 클래스는 문자열이 안전하게 변경되도록 보장하지만, StringBuilder 클래스는 보장되지 않습니다.
- 프로그램에서 따로 쓰레드를 생성하는 멀티쓰레드 프로그램이 아니라면 StringBuilder를 사용하는 것이 싱행 속도가 좀 더 빠릅니다.

day11/string/StringBuilderTest.java

```
package day11.string;

public class StringBuilderTest {
        public static void main(String[] args) {

                String javaStr = new String("Java");
                System.out.println("javaStr 문자열 주소 :"
+System.identityHashCode(javaStr));//처음 생성된 메모리 주소

                StringBuilder buffer = new StringBuilder(javaStr); //String으로 부
터 StringBuilder생성
                System.out.println("연산 전 buffer 메모리 주소:" +
System.identityHashCode(buffer));//buffer 메모리 주소
                buffer.append(" and");                // 문자열 추가
                buffer.append(" android");            // 문자열 추가
                buffer.append(" programming is fun!!!"); //문자열 추가
                System.out.println("연산 후 buffer 메모리 주소:" +
System.identityHashCode(buffer));//buffer 메모리 주소

                javaStr = buffer.toString(); //String 클래스로 반환
                System.out.println(javaStr);
                System.out.println("새로 만들어진 javaStr 문자열 주소 :"
+System.identityHashCode(javaStr)); //새로 생성된 메모리 주소

        }
}
```

실행결과

```
javaStr 문자열 주소 :1651191114
연산 전 buffer 메모리 주소:1586600255
연산 후 buffer 메모리 주소:1586600255
Java and android programming is fun!!!
새로 만들어진 javaStr 문자열 주소 :474675244
```



- StringBuilder 클래스를 새엇ㅇ하고 여기에 문자열을 추가(append) 합니다.
- 그러면 append() 메러드가 실행될 때마다 **메모리가 새로 생성되는 것이 아니라, 하나의 메모리에 계속 연결되는 것**을 해시 코드 값을 통해 알 수 있습니다.
- 연산 전 메모리 주소와 연산 후 메모리 주소가 같기 때문입니다.
- 문자열을 변경한 후에 buffer에 toString() 메서드를 호출하면 다시 문자열로 반환할 수 있습니다.

Wrapper 클래스

기본 자료형을 위한 클래스

- 지금까지 정수를 사용할 때 기본형인 int를 사용했습니다. 그런데 정수를 객체형으로 사용해야 하는 경우가 있습니다. 예) 매개변수가 객체거나 반환값이 객체인 경우

```
public void setValue(Integer i) { ... } // 객체를 매개변수로 받는 경우
public Integer returnValue() { ... } // 반환 값이 객체인 경우
```

- 이를 위해 자바에서는 **기본 자료형 처럼 사용할 수 있는 클래스를 제공**합니다. 이러한 클래스를 기본 자료형을 감쌌다는 의미로 **Wrapper 클래스**라고 합니다.

Wrapper 클래스의 종류

기본형	Wrapper 클래스
boolean	Boolean
byte	Byte
char	Character
short	Short
int	Integer
long	Long
float	Float

기본형	Wrapper 클래스
double	Double

- 가장 대표적인 Integer 클래스를 통해 Wrapper 클래스의 사용법을 알아봅니다. 다른 클래스도 사용 방법이 크게 다르지 않으므로 Integer클래스 기준으로 익혀 두시면 됩니다.

Integer 클래스 사용하기

- int 자료형을 감싼 클래스
- Integer 클래스의 생성자는 다름과 같이 특정 정수를 매개변수로 받는 경우와 문자열을 받는 경우 두 가지가 있습니다.

```
Integer(int value) { ... } // 특정 정수를 매개변수로 받는 경우
Integer(String s) { ... } // 특정 문자열을 매개변수로 받는 경우
```

- Integer 클래스는 int 자료형과 특성이 그대로 구현되어 있습니다. 사용 가능한 최대값과 최솟값이 static 변수로 정의되어 있습니다.
- 대부분의 Wrapper 클래스가 위 Integer 클래스의 정의와 크게 다르지 않습니다.
- Integer 클래스는 멤버 변수로 기본 자료형 int를 가지고 있고, int 값을 객체로 활용할 수 있는 여러 메서드를 제공합니다.
- int value는 final 변수 이며, 한번 생성되면 변경할 수 없습니다.

Integer 클래스의 메서드

- intValue() : Integer 클래스 내부의 int 자료형 값을 가져오기 위해 사용

```
Integer iValue = new Integer(100);
int myValue = iValue.intValue(); // int값 가져오기 myValue 값을 출력하면 100이 출력
됨
```

- valueOf() 정적 메서드 : 정수나 문자열을 바로 Integer 클래스로 반환받을 수 있습니다.

```
Integer number1 = Integer.valueOf("100");
Integer number2 = Integer.valueOf(100);
```

- parseInt() : 문자열이 어떤 숫자를 나타낼 때, 문자열에서 int 값을 바로 가져와서 반환할 수 있습니다.

```
int num = Integer.parseInt("100");
```

- 다른 Wrapper 클래스의 사용방법도 비슷합니다.

오토박싱과 언박싱

- 어떤 정수 값을 사용할 때 int로 선언하는 경우와 Integer로 선언하는 경우는 전혀 다릅니다.

- int는 기본 자료형 4바이트지만 Integer는 클래스이기 때문에 인스턴로 생성하려면 생성자를 호출하고 정수 값을 인수로 넣어야 합니다. 이처럼 기본 자료형과 Wrapper 클래스는 같은 값을 나타내지만, 그 쓰임새과 특성이 전혀 다릅니다.
- JDK1.5 이전에는 기본 자료형과 Wrapper 클래스형을 함께 연산하기 위해 둘 중 하나의 형태로 일치시켜야 했습니다. 예를 들어 Integer와 int형으로 선언한 두 값을 더한다면 Integer에서 intValue() 메서드를 사용해 정수 값을 꺼내거나 int형으로 선언된 변수의 값을 Integer로 만들어 연산해야 했습니다.
- 하지만 JDK1.5부터는 다음과 같이 변환 없이 사용할 수 있습니다.

```
Integer num1 = new Integer(100);
int num2 = 200;
int sum = num + num2;  // num은 num.intValue()로 변환(언박싱)
Integer num3 = num2;  // num2는 Integer.valueOf(num2)로 변환(오토박싱)
```

- **오토박싱(autoboxing)** : 기본형을 객체형으로 바꾸는 것
- **언박싱(unboxing)** : 객체를 기본형으로 꺼내는 것
- 이는 자바의 연산 방식이 변경된 것이 아니라 컴파일러가 변경하는 것입니다. 따라서 객체의 형 변환에 신경 쓰지 않고 편리하게 프로그래밍 할 수 있습니다.

Class 클래스

- 자바의 모든 클래스와 인터페이스는 컴파일되고 나면 class 파일로 생성됩니다.
- 예를 들어 a.java 파일이 컴파일되면 a.class 파일이 생성되고, 이 class 파일에는 클래스나 인터페이스에 대한 변수, 메서드, 생성자 등의 정보가 들어 있습니다.
- Class 클래스는 컴파일된 class파일에 저장된 클래스나 인터페이스 정보를 가져오는 데 사용합니다.

Class 클래스란?

모르는 클래스의 정보를 사용할 경우 우리가 클래스의 정보를 직접 찾아야 합니다. 이때 Class 클래스를 활용합니다.

Class 클래스를 선언하고 클래스 정보를 가져오는 방법

1. Object 클래스의 getClass() 메서드 사용하기

```
String s = new String();
Class c = s.getClass(); // getClass() 메서드 반환형은 Class
```

2. 클래스 파일 이름을 Class 변수에 직접 대입하기

```
Class c = String.class;
```

3. Class.forName("클래스 이름") 메서드 사용하기

```
Class c = Class.forName("java.lang.String");
```

- 1번의 경우 Object에 선언한 getClass() 메서드는 모든 클래스가 사용할 수 있는 메서드입니다. 이 메서드를 사용하려면 이미 생성된 인스턴스가 있어야 합니다.
- 2,3번의 경우에는 컴파일된 클래스 파일이 있다면 클래스 이름만으로 Class 클래스를 반환받습니다.

day11/classex/Person.java

```
package day11.classex;

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


	public Person(){}

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

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

	public String getName() {
		return name;
	}

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

	public int getAge() {
		return age;
	}

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

day11/classex/ClassTest.java

```
package day11.classex;

public class ClassTest {
	public static void main(String[] args) throws ClassNotFoundException {
```

```
                Person person = new Person();
                Class pClass1 = person.getClass();  //Object 의 getClass() 메소드
사용
                System.out.println(pClass1.getName());

                Class pClass2 = Person.class;    // 직접 class 파일 대입
                System.out.println(pClass2.getName());

                Class pClass3 = Class.forName("day11.classex.Person"); // 클래스 이
름으로 가져오기
                System.out.println(pClass3.getName());          //이름과 일치하는
클래스가 없는경우 ClassNotFoundException 발생함
        }
}
```

실행결과

```
day11.classex.Person
day11.classex.Person
day11.classex.Person
```

- Class 클래스를 통하여 클래스 정보를 알 수 있습니다.

Class 클래스를 활용해 클래스 정보 알아보기

day11/classex/StringClassTest.java

```
package day11.classex;

import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.lang.reflect.Method;

public class StringClassTest {
        public static void main(String[] args) throws ClassNotFoundException {

                Class strClass = Class.forName("java.lang.String");

                Constructor[] cons = strClass.getConstructors();
                for(Constructor c : cons){
                        System.out.println(c);
                }
                System.out.println();
                Field[] fields = strClass.getFields();
                for(Field f : fields){
                        System.out.println(f);
                }
                System.out.println();
                Method[] methods = strClass.getMethods();
                for(Method m : methods){
                        System.out.println(m);
                }
        }
}
```

실행결과

```
public java.lang.String(java.lang.StringBuffer)
public java.lang.String(java.lang.StringBuilder)
public java.lang.String(byte[],int,int,java.nio.charset.Charset)
public java.lang.String(byte[],java.lang.String) throws
java.io.UnsupportedEncodingException
public java.lang.String(byte[],java.nio.charset.Charset)
public java.lang.String(byte[],int,int)
public java.lang.String(byte[])
public java.lang.String(char[],int,int)
public java.lang.String(char[])
public java.lang.String(java.lang.String)
public java.lang.String()
public java.lang.String(byte[],int,int,java.lang.String) throws
java.io.UnsupportedEncodingException
public java.lang.String(byte[],int)
public java.lang.String(byte[],int,int,int)
public java.lang.String(int[],int,int)

...
```

- Class 클래스와 java.lang.reflect 패키지에 있는 클래스를 활용하면 클래스 이름만 알아도 클래스의 생성자, 메서드 등의 정보를 알 수 있습니다.

Class.forName()을 사용해 동적 로딩 하기

- 대부분의 클래스 정보는 프로그램이 로딩될 때 이미 메모리에 있습니다.
- 예를들면 어떤 회사에서 개발한 시스템이 있는데, 그 시스템은 여러 종류의 데이터베이스를 지원합니다. 오라클, MySQL, MS-SQL 등등 여러 데이터베이스를 연동할 수 있습니다. 그렇다고 이 시스템을 구동할 때 어떤 데이터 베이스와 연결할지만 결정되면 해당 드라이버만 로딩하면 됩니다. 회사에서 사용하는 데이터베이스 정보는 환경파일에서 읽어올 수 있고 다른 변수 값으로 받을 수도 있습니다.
- 즉, 프로그램 실행 이후 클래스의 로딩이 필요한 경우 **동적 로딩(dynamic loading)** 방식을 사용합니다.
- 자바는 Class.forName() 메서드를 동적 로딩으로 제공합니다.

```
Class pClass = Class.forName("classex.Person");
```

- forName() 메서드를 살펴보면 매개변수로 문자열을 입력받습니다. 이때 입력받는 문자열을 변수로 선언하여 변수 값만 바꾸면 다른 클래스를 로딩할 수 있습니다.
- 앞에서 설명했듯이 여러 데이터베이스 드라이버 중 필요한 드라이버의 값을 설정 파일에서 읽어 문자열 변수로 저장한다면, 설정 파일을 변경함으로써 필요한 드라이버를 간단하게 로딩합니다.

```
String className = "classex.Person";
Class pClass = Class.forName(className);
```

- 위와 같이 작성하고 className 변수에 다른 문자열을 대입하면 필요에 따라 로딩되는 클래스를 동적으로 변경할 수 있습니다.

forName() 메서드를 사용할 때 유의할 점

- forName("클래스 이름")의 클래스 이름이 문자열 값이므로 **문자열에 오류가 있어도(Person의 P가 소문자라든가) 컴파일할 때는 그 오류를 알 수 없습니다.** 결국 프로그램이 실행되고 메서드가 호출될 때 클래스 이름에 해당하는 클래스가 없다면 ClassNotFoundException이 발생합니다. 따라서 **동적 로딩 방식은 컴파일 할 때 오류를 알 수 없습니다.**

유용한 클래스

java.lang.Math 클래스

- 임의의 수
  - random() : 0~1사이의 임의의 실수를 반환
- 올림, 버림 반올림
  - ceil(double a) : 올림
  - floor(double a) : 버림
  - round(double a) : 반올림
- 절대값 : abs()

day11/utils/RandomEx1.java

```
package day11.utils;

public class RandomEx1 {
        final static int RECORD_NUM = 10; // 생성할 레코드의 수를 정한다.
        final static String TABLE_NAME = "TEST_TABLE";
        final static String[] CODE1 = {"010", "011", "017", "018", "019"};
        final static String[] CODE2 = {"남자", "여자"};
        final static String[] CODE3 = {"10대", "20대", "30대", "40대", "50대"};

        public static void main(String[] args) {
                for (int i = 0; i < RECORD_NUM; i++) {
                        System.out.println(" INSERT INTO " + TABLE_NAME + " VALUES
('"
                                                        + getRandArr(CODE1)+ "','"
                                                        + getRandArr(CODE2) +
"','"
                                                        + getRandArr(CODE3) +
"','"
                                                        + getRand(100, 200) //
100~200 사이의 값을 얻는다.
                                                        + "');");
```

```
                }
        }

        public static String getRandArr(String[] arr) {
                return arr[getRand(arr.length - 1)]; // 배열에 저장된 값 중 하나를
반환한다.
        }

        public static int getRand(int n) {
                return getRand(0, n);
        }

        public static int getRand(int from, int to) {
                return (int)(Math.random() * (Math.abs(to-from))) + Math.min(from,
to);
        }
}
```

실행결과

```
 INSERT INTO TEST_TABLE VALUES ('018','남자','20대','135');
 INSERT INTO TEST_TABLE VALUES ('018','남자','20대','105');
 INSERT INTO TEST_TABLE VALUES ('017','남자','30대','141');
 INSERT INTO TEST_TABLE VALUES ('010','남자','30대','135');
 INSERT INTO TEST_TABLE VALUES ('011','남자','30대','135');
 INSERT INTO TEST_TABLE VALUES ('017','남자','20대','109');
 INSERT INTO TEST_TABLE VALUES ('010','남자','20대','136');
 INSERT INTO TEST_TABLE VALUES ('018','남자','30대','112');
 INSERT INTO TEST_TABLE VALUES ('010','남자','30대','130');
 INSERT INTO TEST_TABLE VALUES ('010','남자','10대','147');
```

java.util.Objects 클래스

Object 클래스의 보조 클래스로 Object를 다루는 유용한 메서드를 제공합니다.

```
        static int hashCode(Object o) // 객체의 해시코드 조회 및 생성
        static int hash(Object... values) // 해시코드 생성
```

```
        static boolean equals(Object a, Object b); // 두 객체를 비교
        static boolean deepEquals(Object a, Object b); // 두 객체를 재귀적으로 비교
(다차원 배열 비교 가능 )
```

java.util.Random 클래스

- 난수를 얻는 방법은 Math.random()을 사용하는 방법이 있습니다. 이 외에도 Random 클래스를 사용하면 난수를 얻을 수 있습니다.
- Math.random()은 내부적으로 Random의 인스턴스를 생성해서 사용하는 것이므로 둘 중에서 편한 것을 사용하면 된다.

- 다음 두 코드는 동일합니다.

```
double randNum = Math.random();
double randNum = new Random().nextDouble(); // 위 문장과 동일
```

- 만약 1~6 사이의 정수를 난수로 얻고자 할 때는 다음과 같다.

```
int num = (int)(Math.random() \* 6) + 1;
int num = new Random().nextInt(6) + 1; //
```

- Math.random()과 Random의 가장 큰 차이점은 종자값(seed)을 설정할 수 있다는 것이다. 종자값이 같은 Random인스턴스들은 항상 같은 난수를 같은 순서대로 반환하다.
- 종자값은 난수를 만드는 공식에 사용되는 값으로 같은 공식에 같은 값을 넣으면 같은 결과를 얻는 것처럼 같은 종자값을 넣으면 같은 난수를 얻게된다.

Random 클래스의 생성자와 메서드

- 생성자 Random()은 아래와 같이 종자값을 System.currentTimeMillis()로 하기 때문에 실행할 때마다 얻는 난수가 달라진다.

> System.cyrrentTimeMillis()는 현재시간을 천분의 1초단위로 변환해서 반환한다.

```
public Random() {
        this(System.currentTimeMillis());  // Random(long seed)를 호출한다.
}
```

Random의 생성자와 메서드

메서드	설명
Random()	System 현재 시간을 종자값(seed)로 이용하는 Random인스턴스를 생성한다.
Random(long seed)	매개변수 seed를 종자값으로 하는 Random인스턴스를 생성한다.
boolean nextBoolean()	boolean 타입의 난수를 반환한다.
void nextBytes(byte[] bytes)	bytes배열에 byte타입의 난수를 채워서 반환한다.
double nextDouble()	double타입의 난수를 반환한다.(0.0 <= x < 1.0)
float nextFloat()	float 타입의 난수를 반환한다.(0.0 <= x <1.0)
int nextInt()	int타입의 난수를 반환한다(int의 범위)
long nextLong()	long타입의 난수를 반환한다.(long의 범위)
void setSeed(long seed)	종자값을 주어진 값(seed)으로 변경한다.

day11/utils/RandomEx2.java

```
package day11.utils;

import java.util.Random;

public class RandomEx2 {
	public static void main(String[] args) {
		Random rand = new Random(1);
		Random rand2 = new Random(1);

		System.out.println("= rand =");
		for (int i = 0; i < 5; i++) {
			System.out.println(i + ":" + rand.nextInt());
		}

		System.out.println();
		System.out.println("= rand2 =");
		for (int i = 0; i < 5; i++) {
			System.out.println(i + ":" + rand2.nextInt());
		}
	}
}
```

실행결과

```
= rand =
0:-1155869325
1:431529176
2:1761283695
3:1749940626
4:892128508

= rand2 =
0:-1155869325
1:431529176
2:1761283695
3:1749940626
4:892128508
```

정규식(Regular Expression) - java.util.regex패키지

- 정규식이란 텍스트 데이터 중에서 원하는 조건(패턴, pattern)과 일치하는 문자열을 찾아 내기 위해 사용하는 것으로 미리 정의된 기호와 문자를 이용해서 작성한 문자열을 말한다.
- 원래 Unix에서 사용하던 것이고 Perl의 강력한 기능이었는데, 요즘은 Java를 비롯해 다양한 언어에서 지원하고 있다.
- 정규식을 이용하면 많은 양의 텍스트 파일 중에서 원하는 데이터를 손쉽게 뽑아낼 수도 있고 입력된 데이터가 형식에 맞는지 체크할 수 도 있다. 예를 들면 html문서에서 전화번호나 이메일 주소만을 바로 추출한다던가, 입력한 비밀번호가 숫자와 영문자의 조합으로 되어 있는지 확인할 수도 있다.

정규표현식 패턴

정규식 패턴	설명
abc	문자열을 포함한다
[abc]	문자클래스 - 문자집합안에 특정 문자 한개
[^abc]	부정문자클래스 : 문자 집합안의 특정 문자 한개
[a-z]	두 문자 사이의 모든 문자
.	줄 바꿈 문자를 제외한 문자 한개
\d	모든 숫자[0-9]와 같음
\D	숫자를 제외한 모든 문자 한개 [^0-9]와 같음
\w	임의의 영어 단어 문자(알파벳, 숫자, 언더스코어) 한개
\W	영어단어 문자(알파벳, 숫자, 언더스코어)를 제외한 문자 한개
\s	모든 공백 문자 한 개
\S	공백문자가 아닌 문자 한개
x{2,4}	x를 최소 2번, 최대 4번 반복
x{2,}	x를 2번 잇ㅇ 반복
x?	x를 한번 이하 반복
x+	x를 한번 이상 반복
x*	x를 0번 이상 반복
(x)	x를 그룹화(부분 정규 표현식)
^	문자열의 시작 위치
$	문자열의 마지막 위치
x|y|z	x,y,z 중 하나(선택)

day11/utils/RegularEx1.java

```
package day11.utils;

import java.util.regex.*;

public class RegularEx1 {
        public static void main(String[] args) {
                String[] data = {"bat", "baby", "bonus", "cA", "ca", "co", "c.",
"c0", "car", "combat", "date", "disc"};
                Pattern p = Pattern.compile("c[a-z]*");

                for (int i = 0; i < data.length; i++) {
```

```
                        Matcher m = p.matcher(data[i]);
                        if (m.matches()) {
                                System.out.print(data[i] + ",");
                        }
                }
        }
}
```

실행결과

```
ca,co,car,combat,
```

1. 정규식을 매개변수로 Pattern클래스의 static 메서드인 Pattern compile(String regex)을 호출하여 Pattern인스턴스를 얻는다.

```
Pattern p = Pattern.compile("c[a-z]*");
```

2. 정규식으로 비교할 대상을 매개변수로 Pattern클래스의 Matcher matcher(CharSequence input)를 호출해서 Matcher인스턴스를 얻는다.

```
Matcher m = p.matcher(data[i]);
```

3. Matcher인스턴스에 boolean matches()를 호출해서 정규식에 부합하는지 확인한다.

```
if (m.matches()) {
        ...
}
```

> CharSequence는 인터페이스로 이를 구현한 클래스는 CharBuffer, String, StringBuffer가 있다.

day11/utils/RegularEx2.java

```
package day11.utils;

import java.util.regex.*;

public class RegularEx2 {
        public static void main(String[] args) {
                String source = "HP:011-1111-1111, HOME:02-999-9999 ";
                String pattern = "(0\\d{1,2})-(\\d{3,4})-(\\d{4})";

                Pattern p = Pattern.compile(pattern);
                Matcher m = p.matcher(source);

                int i = 0;
                while(m.find()) {
                        System.out.println(++i + ": " + m.group() + " -> " +
m.group(1) + ", " + m.group(2) + ", " + m.group(3));
                }
        }
```

```
}
```

실행결과

```
1: 011-1111-1111 -> 011, 1111, 1111
2: 02-999-9999 -> 02, 999, 9999
```

java.util.Scanner 클래스

- Scanner는 화면, 파일, 문자열과 같은 입력소스로 부터 문자데이터를 읽어오는데 도움을 줄 목적으로 JDK1.5부터 추가되었다.
- Scanner에는 다음과 같은 생성자를 지원하기 때문에 다양한 입력소스로 부터 데이터를 읽을 수 있다.

```
Scanner(String source)
Scanner(File source)
Scanner(InputStream source)
Scanner(Readable source)
Scanner(ReadableByteChannel source)
Scanner(Path source)
```

- 또한 Scanner는 정규식 표현(Regular expression)을 이용한 라인단위 검색을 지원하며 구분자(delimiter)에도 정규식 표현을 사용할 수 있어서 복잡한 형태의 구분자도 처리가 가능하다.

```
Scanner useDelimiter(Pattern pattern)
Scanner useDelimiter(String pattern)
```

- 입력받을 값이 숫자라면 nextLine() 대신 nextInt() 또는 nextLong()과 같은 메서드를 사용할 수 있다.
- Scanner에서는 이와 같은 메서드를 제공함으로써 입력받은 문자를 다시 변환하는 수고를 덜어준다.

```
boolean nextBoolean()
byte nextByte()
short nextShort()
int nextInt()
long nextLong()
double nextDouble()
float nextFloat()
String nextLine()
```

day11/utils/ScannerEx1.java

```
package day11.utils;

import java.util.*;
```

```
public class ScannerEx1 {
	public static void main(String[] args) {
		Scanner s = new Scanner(System.in);
		String[] argArr = null;

		while(true) {
			String prompt = ">>";
			System.out.print(prompt);

			// 화면으로부터 라인단위로 입력받는다.
			String input = s.nextLine();

			input = input.trim(); // 입력받은 값에서 불필요한 앞뒤 공백
을 제거한다.
			argArr = input.split(" +"); // 입력받은 내용을 공백(1개 이상
의) 으로 구분자로 자른다

			String command = argArr[0].trim();

			if ("".equals(command))
				continue;

			// 명령어를 소문자로 바꾼다.
			command = command.toLowerCase();

			// q또는 Q를 입력하면 실행종료한다.
			if (command.equals("q")) {
				System.exit(0);
			} else {
				for(int i = 0; i < argArr.length; i++) {
					System.out.println(argArr[i]);
				}
			}
		}
	}
}
```

day11/utils/ScannerEx2.java

```
data2.txt

100
200
300
400
500
```

```
package day11.utils;

import java.util.Scanner;
import java.io.File;
import java.io.IOException;
```

```
public class ScannerEx2 {
	public static void main(String[] args) throws IOException {
		Scanner sc = new Scanner(new File("data2.txt"));
		int sum = 0;
		int cnt = 0;

		while(sc.hasNextInt()) {
			sum += sc.nextInt();
			cnt++;
		}

		System.out.println("sum=" + sum);
		System.out.println("average=" + (double)sum/cnt);
	}
}
```

실행결과

```
sum=1500
average=300.0
```

java.util.StringTokenizer 클래스

- StringTokenizer는 긴 문자열을 지정된 구분자(delimiter)를 기준으로 토큰(token)이라는 여러 개의 문자열로 잘라내는 데 사용됩니다.
- StringTokenizer를 이용하는 방법 이외에는 아래와 같이 String의 split(String regex)이나 Scanner의 useDelimiter(String pattern)를 사용할 수도 있지만, 이 두가지 방법은 정규식 표현(Regular expression)을 사용해야하므로 정규식 표현에 익숙하지 않은 경우 StringTokenizer를 사용하는 것이 간단하면서도 명확한 결과를 얻을 수 있다.
- 그러나 StringTokenizer는 구분자로 단 하나의 문자 밖에 사용하지 못하기 때문에 보다 복잡한 형태의 구분자로 문자열을 나누어야 할 때는 어쩔 수 없이 정규식을 사용하는 메서드를 사용해야 할 것이다.

StringTokenizer의 생성자와 메서드

생성자 / 메서드	설명
StringTokenizer(String str, String delim)	문자열(str)을 지정된 구분자(delim)로 나누는 StringTokenizer를 생성한다. returnDelims의 값을 true로 하면 구분자도 토큰으로 간주된다.
StringTokenizer(String str, String delimi, boolean return Delims)	문자열(str)을 지정된 구분자(delim)로 나누는 StringTokenizer를 생성한다. returnDelims의 값을 true로 하면 구분자도 토큰으로 간주된다.
int countTokens()	전체 토큰의 수를 반환한다.
boolean hasMoreTokens()	토큰이 남아있는지 알려준다.
String nextToken()	다음 토큰을 반환한다.

day11/utils/StringTokenizerEx1.java

```
package day11.utils;

import java.util.*;
public class StringTokenizerEx1 {
	public static void main(String[] args) {
		String source = "100,200,300,400";
		StringTokenizer st = new StringTokenizer(source, ",");

		while(st.hasMoreTokens()) {
			System.out.println(st.nextToken());
		}
	}
}
```

실행결과

```
100
200
300
400
```

day11/utils/StringTokenizerEx2.java

```
package day11.utils;

import java.util.*;

public class StringTokenizerEx2 {
	public static void main(String[] args) {
		String expression = "x=100+(200+100)/2";
		StringTokenizer st = new StringTokenizer(expression, "+-*/=()",
true);

		while(st.hasMoreTokens()) {
			System.out.println(st.nextToken());
		}
	}
}
```

실행결과

```
x
=
100
+
(
200
+
100
)
```

/
2

단 한 문자의 구분자만 사용할 수 있기 때문에, "+-*/=()"전체가 하나의 구분자가 아니라 각각의 문자가 모두 구분자라는 것에 주의해야한다.

> 만일 구분자가 두 문자 이상이라면, Scanner나 String 클래스의 split메서드를 사용해야 한다.