**자바프로그래밍2 실습과제**

**제7주 상속**

이클립스에서 자신의 이름으로 프로젝트를 만드세요. 그 아래에 w7라는 이름의 패키지를 만들고 (7주차) 그 아래에 아래 문제의 클래스들을 작성하세요.



상속에 대한 기본적인 연습문제입니다. [dog.zip](src\dog.zip) 파일의 압축을 풀어 이클립스(w7.dog 패키지)에 넣으세요. 코드를 잘 읽어 보세요. 컴파일 에러가 있지만 아래에서 해결할 것입니다.

* 1. Dog가 기본 클래스이고 그것을 확장하여 Jindo와 Yorkiee 클래스를 만들었습니다.

Diagram

Description automatically generated

Jindo는 Dog의 서브클래스입니다. 그런데 Jindo.java의 생성자에 컴파일 에러가 있습니다. 컴파일에러가 왜 발생했는지 설명이 Dog와 Jindo 클래스에 주석 형태로 적혀 있으니 읽어보세요.

Jindo의 생성이자 코드를 수정하여 Jindo의 필드들이 적절히 초기화되도록 하세요. Jindo는 수퍼클래스인 Dog로부터 name 필드를 상속해 갖고 있고, 자기가 선언한 color 필드를 갖습니다. name 필드는 수퍼클래스의 생성자가 초기화하도록 위임하고 자기가 선언한 필드는 자신의 생성자에서 초기화하도록 하세요. Jindo의 생성자만 수정하면 됩니다.

Jindo의 컴파일 에러가 없어지면 DogTest를 실행해 보세요. 아래 그림과 같은 화면이 나타날 것입니다.

A screenshot of a computer

Description automatically generated

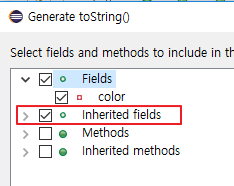
DogTest 코드 중의 dog는 **Dog 타입 참조변수**입니다. 프로그램이 실행되면 dog는 처음에는 Dog 인스턴스를, 두 번째로는 Jindo 인스턴스를, 다음으로는 Yorkie 인스턴스를 가리킵니다. 두 번째로 dog.speak() 메소드가 실행될 때, **Jindo 클래스에서 엎어쓰기한 speak** 메소드가 실행됐다는 점에 유의하세요. dog의 형식적인 타입은 Dog이지만, 실행할 때는 dog가 가리키는 객체의 실제 타입인 Jindo의 speak 메소드가 선택됩니다. Yorkie는 speak 메소드를 엎어쓰기하지 않았기 때문에, 세 번째로 dog.speak() 메소드가 실행될 때는 Dog 클래스로부터 상속한 speak 메소드가 실행됐습니다.

* 1. System.out.println(dog) 문장에 의한 출력은 Object 클래스로부터 상속한 toString() 메소드가 작동한 결과입니다. System.out.println(dog) 문장에 의한 출력이 아래 그림과 같이 되도록 Dog, Jindo, Yorkie 클래스에 toString 메소드를 엎어쓰기하세요. Yorkie 클래스에도 speak 메소드를 엎어쓰기하세요.

A white background with black text

Description automatically generated

toString 메소드를 엎어쓰기할 때는 이클립스 메뉴의 Source - Generate toString... 기능을 이용하면 편리합니다. 이 때 나타나는 아래 대화창에서 Inherited fields가 포함되도록 해야 합니다. (수퍼클래스인 Dog의 name 필드를 protected로 선언했기 때문에 서브클래스에서 이 필드를 읽을 수 있습니다! Dog 클래스의 필드를 다시 확인해 보세요.)



* 1. Jindo는 진돗개의 평균 무게를 필드로 갖고 있습니다. 진돗개의 평균 무게라는 것은 진돗개 전체의 특성이므로 개별 진돗개가 각각 그 값을 인스턴스 필드로 갖고 있을 필요가 없습니다. 그래서 이 필드는 클래스 공통으로 하나만 존재하는 **static** 필드로 선언되어 있습니다. 이 값을 읽는 데 사용하는 메소드는 getAverageWeight입니다. 이 메소드는 static 필드를 읽어 반환하는 메소드이지만 이 메소드가 반드시 static 메소드이어야 하는 것은 아닙니다. 인스턴스 메소드도 static 필드를 읽을 수 있습니다.

DogTest.java에 진돗개의 평균 무게를 출력하는 코드를 추가하세요. 출력은 아래와 같습니다.

A screenshot of a computer code

Description automatically generated

<힌트>

DogTest 안에 있는 dog는 Dog 타입 참조변수로 선언되어 있습니다. 따라서 dog에게는 Dog 타입의 메소드만을 호출할 수 있습니다. Dog 타입에는 getAverageWeight라는 메소드가 없으므로 dog.getAverageWeight()라고 호출할 수 없습니다. dog가 Dog 타입 참조변수이지만 실제로는 Dog의 일종인 Jindo 인스턴스를 가리키고 있다면, Jindo jin = (Jindo)dog 문장으로 새로운 Jindo 타입 참조변수 jin을 선언하고 이 참조변수가 Jindo 인스턴스를 가리키게 한 후, jin.getAverageWeight()라고 호출하면 됩니다.

만약, Jindo 클래스에서 getAverageWeight 메소드가 static 메소드로 선언되었다면, 클래스이름.메소드이름 형식, 그러니까 Jindo.getAverageWeight()라고 호출할 수 있습니다.

강온실 w7p1\_Dog1 문제의 답으로 Dog, Jindo, Yorkie, DogTest 등 네 개의 클래스를 제출하세요. DogTest만 pubic으로 선언하세요.

* 1. w7.dog 패키지를 복사하여 w7.dog2 패키지를 만들고 w7.dog2 패키지에서 작업하세요.

모든 개 종자들은 평균 무게가 있습니다. 그래서 개 품종별로 그 품종의 평균 무게가 얼마인지 알아보는 메소드를 넣고 싶습니다. 현재 Jindo에는 진돗개의 평균 무게 정보가 static 필드에 적혀 있고, 이 필드를 읽어 값을 알려주는 getAverageWeight 메소드가 있습니다. 그런데 Yorkie에는 그런 것이 없습니다.

Dog의 서브클래스를 만들 때 반드시 그 종자의 평균 무게 정보를 적어 넣고 getAverageWeight 메소드도 작성하도록 강제하려면 어떻게 하면 될까요?

수퍼클래스인 Dog에 getAverageWeight 메소드를 만들어 넣으면 됩니다. 그러면 모든 서브클래스가 이 메소드를 갖게 됩니다. 그러나, 평균 무게는 각 견종별로 정해지는 값이므로 Dog 클래스는 이 값을 하나로 갖고 있을 수 없고, 그래서 Dog 클래스에는 getAverageWeight 메소드를 구현할 수 없습니다. 그러니까 Dog 클래스에서는 getAverageWeight 메소드를 **abstract** 메소드로 선언합니다. abstact 메소드를 갖는 클래스는 반드시 abstract 클래스라고 선언해야 합니다. abstract 클래스는 인스턴스를 생성할 수 없습니다. 아직 구현되지 않은 메소드가 있기 때문입니다.

아래 그림과 같이 Dog 클래스를 수정하세요.

A screenshot of a computer program

Description automatically generated

이렇게 하고 나면 Yorkie 클래스에 컴파일 에러가 납니다. Dog의 서브클래스인 Yorkie는 Dog로부터 상속한 abstract 메소드인 getAverageWeight를 반드시 구현해야 하는데 아직 구현하지 않았기 때문입니다. Poodle, Chihuahua, Maltes, bulldog 등 Dog의 다른 서브클래스를 만들 때도 마찬가지입니다. 이것들도 반드시 getAverageWeight 메소드를 구현해야 합니다. (이런 이유로 abstract 클래스가 일종의 “양식(form)”으로 사용된다고 말합니다.)

Yorkie에 getAverageWeight 메소드를 적절히 구현하세요. 물론 아래와 같은 필드도 추가되어야 합니다.

private static int averageWeight = 4; // Yorkie 클래스 공통의 값이므로 static으로 선언함.

DogTest.java 클래스에도 컴파일 에러가 보입니다. Dog 클래스를 abstract 클래스로 만들었기 때문에 Dog 인스턴스를 생성할 수 없기 때문입니다. Dog 인스턴스를 생성하고 사용하는 코드를 삭제하고 (Dog 타입 참조변수 dog를 선언하는 부분은 남겨두세요), 요크셔의 평균무게를 출력하는 코드를 추가한 후 프로그램을 실행해 보세요. 출력은 아래와 같습니다.

A screenshot of a computer code

Description automatically generated

강온실 w7p1\_Dog2 문제의 답으로 Dog, Jindo, Yorkie, DogTest 등 네 개의 클래스를 제출하세요. DogTest만 pubic으로 선언하세요.

* 1. w7.dog2 패키지를 복사하여 w7.dog3 패키지를 만들고 w7.dog3 패키지에서 작업하세요.

DogTest.main을 아래와 같이 바꾸고 실행해 보세요. List<Dog>에 여러 마리의 Jindo와 Yorkie들을 넣고 이들에게 차례로 toString과 speak 메소드를 호출하면 각 Dog가 Jindo인지, Yorkie인지에 따라 다르게 행동합니다. Dog들이 이렇게 행동하는 것을 다형성(Polymorphism)이라고 합니다.

A screenshot of a computer code

Description automatically generated

A screenshot of a computer

Description automatically generated

강온실 w7p1\_Dog3 문제의 답으로 Dog, Jindo, Yorkie, DogTest 등 네 개의 클래스를 제출하세요. DogTest만 pubic으로 선언하세요.

* 1. w7.dog3 패키지를 복사하여 w7.dog4 패키지를 만들고 새 패키지에서 아래 작업을 하세요.

Dog, Jindo 각각에 equals, hashcode 메소드를 추가하세요. Dog 클래스를 제일 먼저 수정해야 합니다. Jindo는 Dog의 서브클래스이므로 Dog의 수정에 영향을 받기 때문입니다. 이클립스 메뉴 중 Source - Generate hashcode() and equals()... 기능을 사용하세요. Yorkie는 필드를 새로 선언하지 않았으므로 Dog로부터 상속한 필드(name)만을 갖습니다. 그러니까 Yorkie는 Dog로부터 상속한 hashcode, equals 메소드를 엎어쓰기할 필요가 없습니다.

이클립스가 만들어 준 Jindo의 hashcode 코드는 아래와 같습니다.

A computer code with red text

Description automatically generated

빨간 네모 부분을 눈여겨 보세요. super.hashCode()는 Jindo 자신에게 수퍼클래스인 Dog의 hashCode() 메소드를 호출하는 것입니다.

Dog의 hashcode 메소드는 아래와 같습니다.

텍스트, 스크린샷, 폰트, 라인이(가) 표시된 사진

자동 생성된 설명

그러니까, 위 빨간 네모 코드는 Jindo 자신의 상태변수인 name의 hash code 값을 계산합니다. 그 아래 줄에서는 color의 hash code를 얻어 그 정보를 result에 추가합니다. 결국, Jindo의 hashCode 메소드는 Jindo의 name과 color 등 두 가지 정보로부터 hashCode 값을 (곱하기, 더하기 등 지지고 볶아) 계산해 냅니다.

@Override 줄은 지우지 말고 놔 두세요. 이것은 코드에 오류가 있는지 컴파일러가 점검하는 데 도움을 주는 “아노테이션(표식)”입니다.

아래는 Jindo의 equals 메소드입니다.

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명

Jindo의 equals 메소드는 얼핏 보면 두 Jindo 인스턴스가 color만 같으면 두 객체가 같다고 판단하는 것처럼 보이지만(파란 네모), 빨간 네모 코드가 name이 같은지 확인하므로 실제로는 name과 color이 모두 같은 경우에만 두 객체가 같다고 판단하게 됩니다. 빨간 메모 부분은 이 Jindo에게 Dog의 equals 메소드를 호출하면서 obj를 인자로 주는 것입니다. obj가 Dog인데 그 name이 이 Jindo의 name과 같은지 판별합니다.

Set은 List처럼 여러 원소들을 넣는 Collection입니다. List는 원소들의 순서를 관리하고 같은 원소를 여러 개 넣는 경우 그들을 모두 다 보관합니다. 반면, Set은 원소의 중복을 허용하지 않습니다. 같은 원소 두 개를 넣으면 하나만 들어갑니다. 또, Set은 원소들의 순서를 관리하지 않습니다. Set에 원소를 넣는 순서와 Set에서 원소를 하나씩 읽을 때 나오는 순서는 서로 관계없습니다. HashSet은 Set 인터페이스를 구현한 대표적인 클래스입니다.

DogTest.main의 기존 코드 대신 아래 코드를 넣고 실행하면 어떤 출력이 나올지 생각해 보고, 프로그램을 실행하여 확인하세요. HashSet은 원소의 중복을 허용하지 않습니다. Jindo 두 마리가 같은지 여부는 Jindo의 equals 메소드에 의해 결정됩니다.

A computer code with text

Description automatically generated

강온실 w7p1\_Dog4 문제의 답으로 Dog, Jindo, Yorkie, DogTest 등 네 개의 클래스를 제출하세요. DogTest만 pubic으로 선언하세요. 아래 세 개의 import 문장이 맨 위에 있어야 합니다.

import java.util.HashSet;

import java.util.Set;

import java.util.Objects;

**아래 p2, p3 문제는 기능이 복잡하므로 클래스와 메소드에 주석을 달아야 합니다. 주석을 형식에 맞춰 작성해야 하며, 의미가 분명히 전달되도록 요령껏 작성해야 합니다.**

w7.stutterlist 패키지에 아래 프로그램을 작성하세요.

원소를 추가하면 그 원소가 여러 개 추가되는 리스트 클래스를 작성하고자 합니다. 클래스 이름은 StutterIntList입니다. stutter라는 단어는 "말 더듬다"는 의미를 갖습니다. 아래 그림은 원소를 추가하면 주어진 원소 세 개가 추가되는 StutterIntList의 사용 예입니다. ([UseStutter.java](src\UseStutter.java)) 아래 코드를 잘 읽어 보세요.

A screenshot of a computer program

Description automatically generated

Figure 1

위 프로그램을 실행한 결과 화면은 아래와 같습니다.

Text

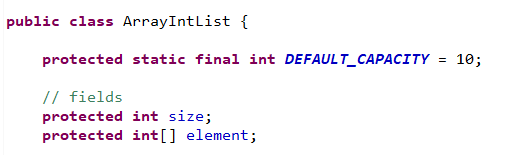
Description automatically generated

위 화면에서는 예외처리 결과(빨간색과 파란색 글씨)가 화면의 끝에 보이지만, 예외처리 결과가 화면의 첫 부분이나 중간에 나타날 수도 있습니다. 정상적인 출력은 System.out 채널로, 에러 출력은 System.err 채널로 나갑니다. 이 두 채널이 모두 콘솔에 연결되어 있습니다. 그래서 두 채널로부터 오는 출력들이 상황에 따라 (프로그램에서의 출력 순서와 달리) 어느 한쪽이 먼저 나타나거나 서로 섞일 수 있습니다.

반복도는 영어로 stretch factor(늘이는 정도)라고 부를 수 있습니다. 리스트에 원소를 add할 때 얼마만큼이나 늘여서 넣을 것인지를 지정하는 숫자이기 때문입니다. 위 코드에서 사용된 StutterIntList는 반복도(stretch factor)가 3인 StutterIntList입니다.

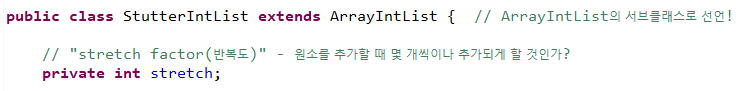
StutterIntList 클래스는 지난 주에 작성한 ArrayIntList의 서브클래스로 선언합니다. StutterIntList는 ArrayIntList와 매우 유사하기 때문에 ArrayIntList의 속성(필드)과 메소드를 상속하고, **상속한 메소드 중 일부를 적절히 엎어쓰기(override)하거나 속성과 메소드를 한 두 개 추가하여 구현할** 수 있기 때문입니다. 그러니까, 복잡한 기능들을 처음부터 새로 만드는 대신, ArrayIntList에 구현된 대부분의 기능을 재사용하는 것이지요.

[ArrayIntList](src\ArrayIntList.java) 를 w7.stutterlist 패키지에 붙여 넣으세요. 이 클래스는 지난 주에 작성한 ArrayIntList와 같은 것입니다. 단, 아래 그림의 필드들이 private에서 protected로 변경되었습니다.



서브클래스인 StutterIntList 클래스 내에서 size와 element 필드에 접근할 수 있도록 하기 위함입니다. 이들을 ArrayIntList 클래스의 private 필드로 유지하는 경우, StutterIntList 객체는, 자신이 size와 element 필드를 갖고 있음에도 불구하고, 자신의 이 필드에 접근할 수 없습니다. DEFAULT\_CAPACITY는 static 필드입니다. static 필드는 “클래스 필드”라고도 부르는 데서 알 수 있듯이 ArrayIntList 클래스 고유의 필드입니다. static 필드는 상속되지 않습니다. (서브클래스는 베이스클래스의 static 필드를 복사해서 갖지 않습니다.) 그렇지만 서브클래스에서 베이스클래스(수퍼클래스)의 static 필드에 접근할 수는 있습니다. 즉, **베이스클래스는 자신의 static 필드를 서브클래스들과 "공유(share)"합니다.**

StutterIntList 클래스는 원소를 추가할 때 그 원소를 몇 번이나 넣어야 할지 알아야 하기 때문에 stretch factor를 상태변수(필드)에 기억해야 합니다. 이 상태변수 이름을 stretch로 지읍시다. 아래와 같이 StutterIntList 클래스를 작성하세요.

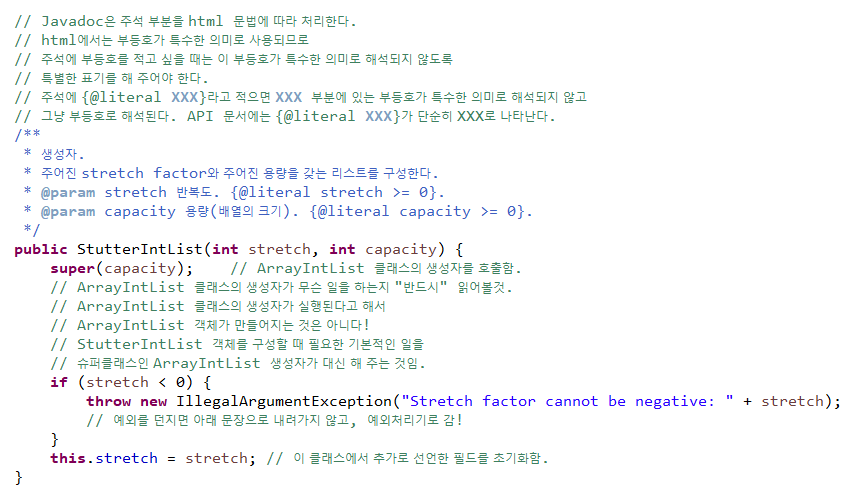


이렇게 선언하면 StutterIntList 객체는 elements, size, stretch 등 세 개의 필드를 갖게 됩니다.

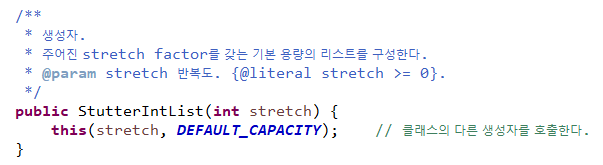
A diagram of a computer

Description automatically generated

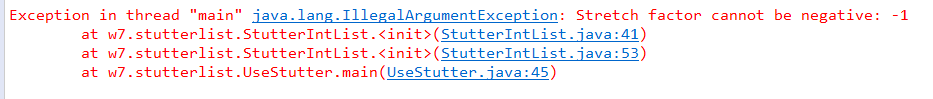
다음으로는 생성자를 작성합니다. 우선 두 개의 파라미터를 갖는 생성자를 작성해 봅니다. 아래 녹색 주석은 실습코드에 입력하지 않아도 됩니다. 잘 읽어 보고 이해하면 됩니다.



반복도 하나만을 파라미터로 갖는 생성자는 방금 만든 파라미터 두 개짜리 생성자를 호출함으로써 쉽게 구현할 수 있습니다.



이제 Figure 1 [UseStutter](src\UseStutter.java) 코드를 실행해 보세요. UserStutter에서 컴파일 에러가 나는 부분(아직 구현하지 않은 SttutterIntList 메소드를 호출하는 문장들)은 일단 주석처리되어 있는데 이것을 해제하지 말고 실행하세요.



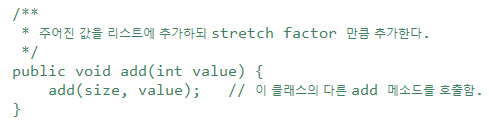
Stretch factor가 음수로 주어지면 (Figure 1의 마지막 줄) 생성자에서 IllegalArgumentException을 던집니다. 이 예외는 프로그램 안에서 잡아 처리하지 않았기 때문에 예외가 시스템까지 전파되고 시스템의 예외처리기가 작동하여 위 그림과 같은 메세지를 표준에러출력(콘솔)으로 내보냅니다. IllegalArgumentException은 비확인예외이기 때문에 예외처리를 해 주지 않아도 컴파일에 문제가 없습니다. 만약 IllegalArgumentException이 확인예외였다면 StutterIntList 생성자에 헤더에 **throws** IllegalArgumentException 표기를 해야 했을 것이고, UseStutter.main에도 같은 표기를 붙이거나, 생성자를 호출하는 문장을 try블록으로 감싸고 catch 블록을 붙여 예외처리를 해야 했을 것입니다.

다음으로는 add(int index, int value) 메소드를 엎어쓰기합니다. 수퍼클래스(ArrayIntList)로부터 상속한 add는 주어진 값을 한 개만 삽입하므로 이것을 수정하여 stretch 개만큼 삽입하도록 합니다. 아래 녹색 주석은 실습코드에 입력하지 않아도 됩니다. 잘 읽어 보고 이해하면 됩니다.

A screenshot of a computer code

Description automatically generated

파라미터가 하나인 add 메소드는 아래 그림과 같이 방금 구현한 파라미터 두 개짜리 add를 이용하여 쉽게 구현할 수 있습니다.



그러나, 위 코드는 수퍼클래스로부터 상속한 파라미터 하나짜리 add 메소드의 코드와 완전히 같습니다. 따라서 이 코드는 새로 작성할 필요가 없습니다.

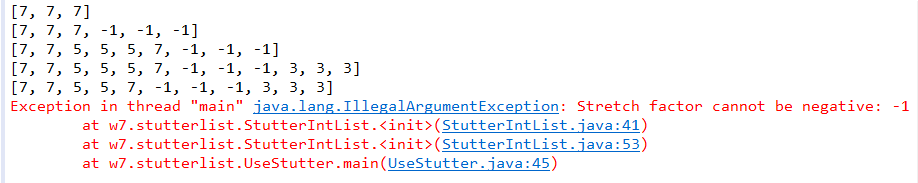
<ArrayIntList의 파라미터 하나짜리 add 메소드>

A close up of text

Description automatically generated

우리는 이제 add 메소드를 사용할 수 있습니다. remove 메소드 호출도 사용할 수 있습니다. remove는 우리가 엎어쓰기하지 않았으므로 수퍼클래스인 ArrayIntList로부터 상속한 remove 메소드가 작동합니다.

두 개의 add 메소드(파라미터 하나짜리와 두 개짜리)와 remove를 테스트합니다. UseStutter의 10번 줄부터 24번 줄까지의 주석을 해제하고 실행하세요. 여러 줄을 주석처리하려면 해당 줄들을 선택(Shift + 아래/위 화살표 여러 번)한 후, "control-/" 합니다. 주석처리를 해제하려면 해당 줄들을 선택한 후 다시 한 번 "control-/" 합니다. 프로그램을 실행하면 아래 그림과 같은 출력이 나타날 것입니다.

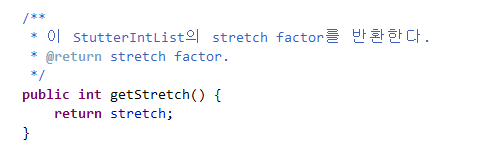


아래 그림과 같은 출력문장(11번줄)에서는 toString 메소드가 묵시적으로 호출됩니다.

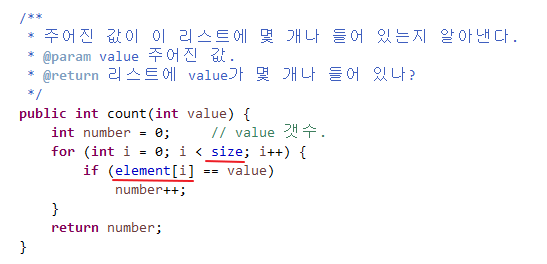


toString 메소드도 역시 ArrayIntList로부터 상속한 것을 그대로 사용한 것입니다.

다음으로 stretch factor을 알아내는 데 사용하는 getter 메소드를 작성합니다.



또, 특정 값이 몇 개나 들어 있는지 알아내는 아래 메소드를 추가합니다.



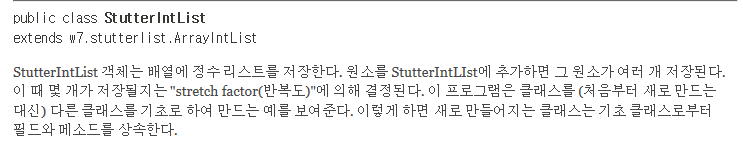
위 코드 중의 element와 size는 수퍼클래스인 ArrayIntList에 선언된 필드로서 StutterIntList는 이 필드를 상속합니다. 그런데 이 필드들은 ArrayIntList에 private 필드로 선언되어 있었습니다. private 필드는 같은 클래스에서만 접근할 수 있는데 ArrayIntList와 StutterIntList는 같은 클래스가 아니므로 그 상태로는 위 코드가 컴파일되지 않습니다. StutterIntList 객체는 내부에 세 개의 필드(size, element, stretch)를 갖고 있는데 이 중 두 개(size, element)에는 접근할 수 없기 때문입니다. 이 문제를 해결하기 위해 우리는 이미 수퍼클래스인 ArrayIntList의 element와 size 등 두 개의 필드를 protected로 바꾸어 선언해 주었습니다. protected로 선언된 멤버에는 같은 패키지에 속하는 클래스들과 서브클래스들로부터의 접근이 허용됩니다. 이렇게 하면 StutterIntList 객체가 (ArrayIntList로부터 상속한) **자신의** 세 필드 모두에 접근할 수 있게 됩니다.

StutterIntList를 javadoc으로 처리하여 얻은 [api 문서](doc/index.html)의 일부를 발췌하면 아래와 같습니다.

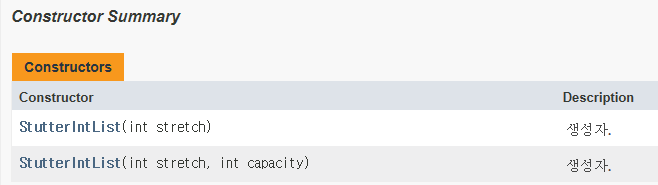


StutterIntList 클래스의 수퍼클래스를 모두 보여줍니다. 맨 위에 Object 클래스가 있습니다.

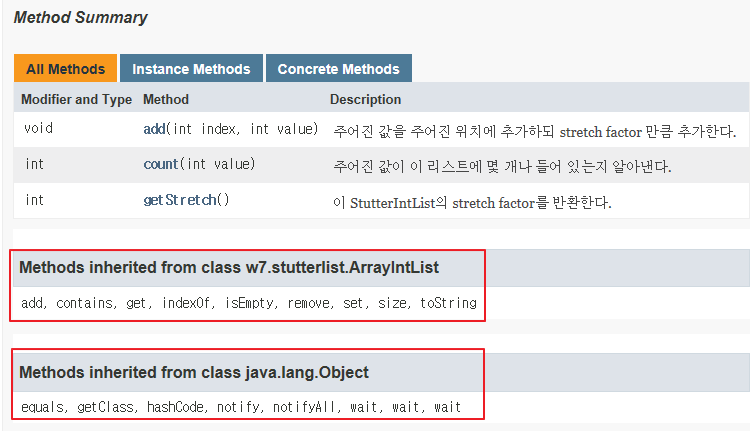
아래는 StutterIntList클래스에 대한 설명입니다.



그리고 두 개의 생성자를 간단히 보여줍니다.

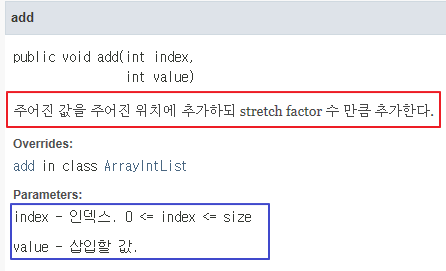


아래에서는 이 클래스에 추가되거나 엎어쓰기된 세 개의 public 메소드를 간단히 보여줍니다. 이 세 개의 메소드 중 getStretch와 count는 이 클래스에 새로 추가한 메소드이고, add 메소드는 수퍼클래스인 ArrayIntList로부터 상속한 것을 엎어쓰기(기능변경)한 것입니다.

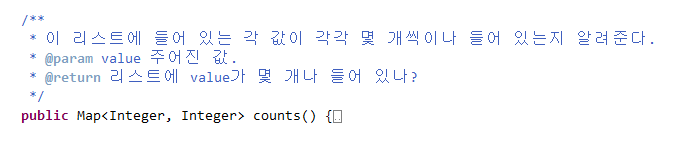


위 그림은 또, 수퍼클래스들로부터 단순히 상속한 메소드들도 함께 보여줍니다(빨간 네모 부분). 사용자는 StutterIntList 객체에게 이 메소드들도 호출할 수 있습니다. 이 메소드들의 사용법을 보려면 해당 메소드 이름을 클릭합니다. 그러면 그 메소드를 구현한 수퍼클래스의 api 문서로 이동합니다.

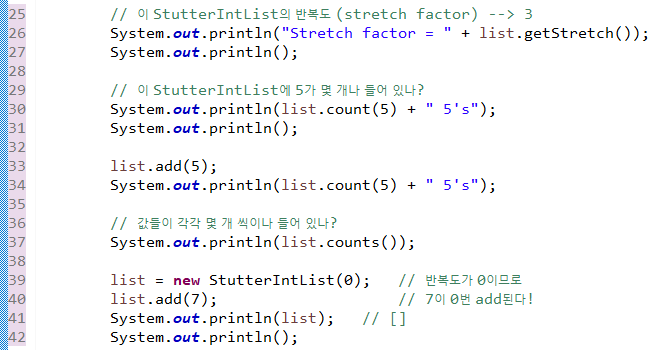
아래는 add 메소드에 대한 상세한 설명입니다. 빨간 부분은 StutterIntList에서 적어 준(엎어쓰기한) 주석이고 파란 부분은 수퍼클래스 주석에 적힌 내용을 그대로 상속한 것입니다. 메소드를 상속하면 수퍼클래스 메소드의 주석까지도 상속합니다. 주석은 설명, 파라미터, 반환값 등으로 구성되는데 상속한 메소드에 엎어쓰기 하면서 상속한 주석도 그중 일부를 엎어쓰기할 수 있습니다. 엎어쓰기하면서 메소드에 대한 설명만 주석으로 적으면 메소드 설명은 그 설명으로 엎어쓰기 되고 나머지 파라미터와 반환값은 상속한 내용이 유지됩니다.



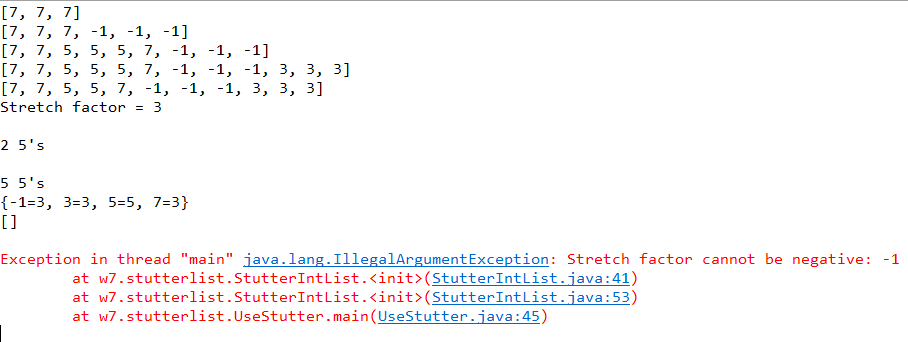
* 1. StutterIntList 클래스에 counts 메소드를 추가하세요. 이 메소드는 리스트에 들어 있는 각 값이 몇 개씩이나 들어 있는지를 Map<Integer, Integer>에 넣고 그 Map을 반환합니다. 가령, 4가 두 개, 7이 세 개 들어 있다면 {4=2, 7=3}과 같은 내용의 Map을 반환합니다.



counts 메소드를 완성한 후, UseStutter의 주석을 아래 그림과 같이 모두 해제하고 새로 작성한 StutterIntList 메소드들이 올바로 작동하는지 테스트하세요.



이제 UseStutter의 모든 주석처리를 해제하고 실행하면 아래와 같은 출력이 나와야 합니다.



**강온실 w7p2 StutterIntList 문제의 답으로 완성된 StutterIntList와 ArrayIntList를 입력하세요. UseStutter은 입력하지 않습니다. Import 문장은 넣지 마세요. java.util.HashMap과 java.util.Map, java.util.Arrays이 문제 자체에 이미 import되어 있습니다. StutterIntList와 ArrayIntList 중 어느 하나만 public으로 선언하세요.**

상속 연습으로, 이번에는 정렬 기능을 갖는 ArrayIntList를 만들어봅니다. 아래에 있는 **문제 설명을 일단 끝까지 다 읽고 나서** 작업을 시작하세요.

ArrayIntList의 서브클래스인 SortedIntList 클래스를 작성하세요. SortedIntList는 ArrayIntList의 일종으로서 배열의 원소들을 오름차순으로 유지하는 특징을 갖습니다. SortedIntList는 ArrayIntList의 기존 기능 중 일부를 변경하고 새 기능을 몇 개 추가하여 구현할 수 있습니다. SortedIntList와 기존의 ArrayIntList간의 차이점 중 중요한 것은 아래 두 가지입니다.

* SortedIntList는 원소들을 오름차순으로 저장합니다.
* SortedIntList는 원소의 중복을 금지할지 여부를 설정할 수 있습니다. 이를 위해 SortedIntList는 boolean 타입인 uniqueness(원소중복금지)를 인스턴스 필드로 갖습니다. uniqueness가 true인 SortedIntList는 원소를 중복해서 갖지 않습니다. uniqueness가 false인 SortedIntList에는 원소를 중복해 넣을 수 있습니다.

기존의 [ArrayIntList](doc\index.html)는 아래와 같은 생성자와 메소드를 갖고 있습니다.

A screenshot of a computer

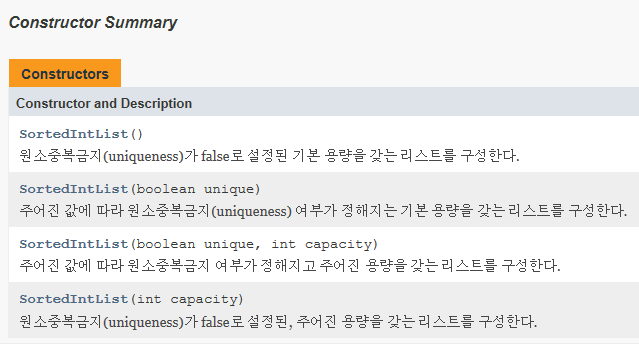
Description automatically generated

A screenshot of a computer

Description automatically generated

새로 작성하는 SortedIntList는 ArrayIntList를 확장합니다. 즉, 아래에서 보듯 생성자와 메소드들을 추가하고 몇 개의 기존 메소드에 엎어쓰기를 해야 합니다. ArrayIntList 클래스 자체는 변경하지 말아야 합니다.

작성할 생성자



엎어쓰기하거나 추가할 메소드



SortedIntList를 작성하는 데 있어서 중요한 사항을 요약하면 아래와 같습니다.

* SortedIntList는 원소의 중복을 금지하는지 여부를 나타내는 필드를 하나 가져야 합니다.
* add(int value) 메소드를 엎어쓰기해야 합니다. 원소를 추가할 때 원소를 리스트의 맨 끝에 추가하는 대신, 오름차순이 유지되도록 적당한 위치에 삽입해야 합니다.
* add(int index, int value) 메소드는 작동하지 않도록 엎어쓰기해야 합니다. add할 때 위치를 임의로 지정하면 오름차순 정렬이 안되기 때문입니다.
* indexOf 메소드도 엎어쓰기해야 합니다. 원소들이 오름차순으로 정렬되어 있어 원래의 방법(앞에서부터 차례로 하나하나 비교해 나가는 방법, linear search)보다 효율적인 작업(binary search)이 가능하므로 **성능을 개선하기** 위해 엎어쓰기를 합니다.
* toString 메소드도 엎어쓰기해야 합니다. 반환하는 문자열의 형식이 다릅니다.
* 위 그림의 다른 메소드들(max, min, setUnique, getUnique)은 새로 추가해야 합니다.
* ArrayIntList로부터 상속한 다른 메소드들은 엎어쓰기할 필요가 없습니다.

**상세한 구현 방법을 아래에서 설명합니다.**

**import w7.stutterlist.ArrayIntList;**

**public SortedIntList()**

기본용량(10)을 갖는, uniqueness가 false로 설정된 리스트를 생성합니다. 원소의 중복을 허용합니다.

**public SortedIntList(boolean unique)**

기본용량(10)을 갖는, uniqueness가 주어진 값으로 설정된 리스트를 생성합니다. unique가 true이면 원소의 중복을 금지하고 false이면 중복을 허용합니다.

**public SortedIntList(int capacity)**

주어진 용량을 갖는, uniqueness가 false로 설정된 리스트를 생성합니다(원소의 중복 허용). 만약, capacity가 음수라면 IllegalArgumentException이 발생해야 합니다(ArrayIntList의 생성자가 그랬듯이).

**public SortedIntList(boolean unique, int capacity)**

주어진 용량을 갖는, uniqueness가 주어진 값으로 설정된 리스트를 생성합니다. 만약, capacity가 음수라면 IllegalArgumentException이 발생해야 합니다(ArrayIntList의 생성자가 그랬듯이).

**public void add(int value)**

value를 리스트의 끝에 추가하는 대신 오름차순이 유지되도록 하는 위치에 삽입합니다. 예를 들어 현재 리스트가 [-3, 7, 18, 42]인 경우 27을 추가한다면, 추가 후에는 리스트가 [-3, 7, 18, 27, 42]가 되어야 합니다.

add 메소드는 원소중복금지(uniqueness)가 설정됐는지 여부를 살펴야 합니다. 만약 중복 금지라면 이미 리스트에 들어 있는 숫자는 추가하지 말아야 합니다.

원소를 리스트의 끝에 추가한 후 리스트를 새로 정렬하는 식으로 구현하지는 말아야 합니다. 처음부터 원소가 삽입될 위치를 찾아 그 곳에 삽입하는 것이 리스트 전체를 다시 정렬하는 것보다 훨씬 효율적이기 때문입니다.

원소를 삽입할 위치를 정하고 나면 이미 구현한 add(index, value) 메소드를 호출하면 됩니다.

원소가 삽입될 위치를 찾을 때는 원소들이 이미 정렬되어 있다는 점을 잘 이용해야 빠르게 찾을 수 있습니다. ArrayIntList에서 사용한 순차탐색(sequential search) 대신 그보다 훨씬 빠른 이분탐색(binary search) 알고리즘을 사용합니다. (<https://blog.penjee.com/binary-vs-linear-search-animated-gifs/>), 이분탐색은 원소들이 정렬되어 있을 때만 사용할 수 있습니다.

이분탐색 알고리즘을 직접 구현할 필요는 없습니다. 자바 표준라이브러리의 Arrays 클래스의 static 메소드인 [binarySearch](https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Arrays.html#binarySearch(int%5B%5D,int,int,int)) 메소드를 사용하면 됩니다.



예를 들어, 아래와 같은 int 배열에서 42가 몇 번 방에 있는지 찾는다고 해 봅시다.

int[] data = [-4, 2, 7, 10, 15, 20, 22, 25, 30, 36, 42, 50, 56, 68, 85, 92, 103, 0, 0];

0번 방부터 16번 방까지 17개의 숫자가 오름차순으로 정렬되어 있습니다. 맨 뒤의 0 두 개는 빈 공간입니다. (크기가 19인 int 배열을 만든 후 앞 부분에 17개의 숫자를 넣은 상태)

42의 인덱스(42가 있는 방의 번호)를 찾으려면 아래와 같이 합니다.

// data 배열의 0번 방 이상(0번 포함)부터 17번 방 미만(17번 미포함)에서 42를 찾음.

int index1 = Arrays.binarySearch(data, 0, 17, 42);

위 문장의 실행 결과로 index1은 10이 됩니다. 42가 10번 방에 있기 때문입니다. 그럼 새로운 42를 이곳에 끼워 넣으면 됩니다.

찾는 값이 리스트에 들어 있지 않은 경우에는, 그 값이 리스트에 있었다면 차지했을 방의 번호(삽입지점)에 마이너스 기호를 붙여 음수로 만들고 그 음수 값에서 1을 뺀 값을 반환합니다. 예를 들어, 아래와 같이 66을 찾으면,

int index2 = Arrays.binarySearch(data, 0, 17, 66);

index2는 -14가 됩니다. 일단 66은 리스트에 들어 있지 않으므로 반환값이 음수입니다. 또, 66을 리스트에 삽입한다면 13번 방에 넣어야 합니다. 즉, 삽입지점이 13입니다, 그러니까 -13-1=-14의 계산에 의해 -14가 반환됩니다. 그러니까, 반환값에 1을 더한 후 부호를 바꾸면 그 수가 곧 삽입지점입니다.

(참고: 1은 왜 빼주나? 🡪 1을 빼주지 않는다고 해 봅시다. 찾는 값이 0번 방에 이미 들어 있어서 인덱스 0이 반환된 경우와, 찾는 값이 리스트에 없는데 삽입지점이 0인 경우가 구분되지 않습니다! 0 == -0)

**public void add(int index, int value)**

정렬 상태를 유지하는 리스트에 원소를 추가할 때 특정 값을 특정 위치에 추가하라고 요구하는 것은 타당하지 않습니다. 이 메소드는 필요하지 않습니다. 그러나 우리는 이 메소드를 수퍼클래스로부터 이미 상속했으로 그냥 놔둘 수는 없습니다. 대신 이 메소드가 UnsupportedOperationException을 던지고 아무 일도 하지 않도록 엎어쓰기해 줍니다. 이 메소드 호출에 의해서는 리스트에 어떤 변경도 일어나지 않아야 합니다.

**public boolean getUnique()**

이 메소드는 현재의 uniqueness 설정 내역을 반환합니다. 즉, 원소중복금지 상태이면 true를, 그렇지 않으면 false를 반환합니다.

**public void setUnique(boolean value)**

이 메소드는 리스트의 원소중복금지 여부를 설정해 줍니다. value가 true이면 원소 중복이 금지되며 false이면 중복이 허용됩니다. 리스트가 원소중복금지로 설정되어 있는 상태에서는, 리스트에 이미 들어 있는 정수를 add하는 경우, 리스트에 아무런 변화가 생기지 말아야 합니다. 리스트가 비어 있는 상태에서 42를 세 번 add하는 경우, 원소중복금지가 해제되어 있다면 [42 42 42]가 되어야 하며, 원소중복금지가 설정되어 있다면 [42]가 되어야 합니다.

리스트가 이미 중복된 원소들을 갖고 있는 상태에서 사용자가 setUnique(true)를 호출함으로써 원소중복금지상태로 전환하면, 중복된 여분의 원소들이 즉시 삭제되어야 하며, 그 이후로는 원소중복금지가 다시 해제되지 않는 한 중복 원소의 추가가 허용되지 않아야 합니다.

**public int max(), public int min()**

이 메소드들은 각각 리스트에서 가장 큰 값과 가장 작은 값을 반환합니다. 예를 들어 리스트가 [4, 4, 17, 39, 58]이라면, 최대값은 58이고 최소값은 4입니다. 리스트에 원소가 하나만 있는 경우에는 최대값과 최소값이 같습니다. 리스트가 비어 있는 경우에는 최대값과 최소값의 개념이 없으므로 NoSuchElementException을 던져야 합니다.

**public String toString()**

이 메소드는 아래와 같은 형식의 문자열을 반환합니다.

"S:[4, 4, 17, 29, 102]", 혹은 "S:[-5, 7, 19]U"

원소중복금지가 해제된 상태라면 첫 번째 형식의 문자열을, 원소중복금지가 설정된 상태라면 두 번째 형식의 문자열을 반환합니다. U는 Unique를 의미합니다. 첫머리의 S는 Sorted를 의미하며 이 S는 항상 붙습니다.

**public int indexOf(int value)**

만약 value가 리스트에 하나만 들어 있다면 그 위치를 반환합니다. 만약 value가 리스트에 여러 개 들어 있으면 그 중 어느 하나(어느 것이라도 좋음)의 방 번호를 반환합니다. 만약 value가 리스트에 들어 있지 않으면 음수를 반환합니다.

이 메소드는 리스트가 정렬된 상태라는 점을 잘 이용하도록 엎어쓰기해야 합니다. 이 때 역시 표준라이브러리의 Arrays.binarySearch 메소드를 사용하면 됩니다.

**<아래는 코드를 작성할 때의 유의사항입니다.>**

* SortedIntList 클래스를 구현할 때 **Arrays.sort 메소드를 사용하지 마세요.** 사용할 필요도 없습니다.
* 코드의 반복을 피하도록 노력해야 합니다. 프로그램 내에 유사한 코드가 여기 저기 반복되는 것은 매우 좋지 않습니다. 그런 경우 반복되는 코드를 private helper 메소드로 작성하고 이 메소드를 호출하여 사용하도록 함으로써 코드 반복을 피해야 합니다. 이렇게 하면 로직을 추상화하는 효과도 얻게 됩니다. 복잡한 코드를 직접 적어 넣는 것 보다는 복잡한 코드를 private helper메소드로 선언하면서 그 메소드에 적절한 이름을 붙여줍니다. 이 메소드가 필요한 곳에서는 이 메소드 이름을 이용하여 메소드를 호출합니다. 복잡한 로직을 적당한 메소드 이름으로 추상화하는 것입니다.
* 두 개 이상의 생성자들이 서로 비슷한 코드를 갖고 있을 때 한 생성자가 다른 생성자를 호출하게 하는 것도 코드 반복을 줄이는 하나의 방법입니다. 메소드들 간에도 마찬가지로 한 메소드가 유사한 코드를 갖는 다른 메소드를 호출하게 함으로써 코드 반복을 줄일 수 있습니다.
* 불필요한 코드를 줄이는 또 다른 방법은 클래스의 새 메소드를 구현할 때 수퍼클래스인 ArrayIntList로부터 상속한 메소드를 활용하는 것입니다. 예를 들어 ArrayIntList는 리스트의 특정 위치에 원소를 추가하거나 삭제하는 기능을 갖고 있습니다. 만약 SortedIntList의 메소드를 구현할 때 이런 기능이 필요하면, 새로 코드를 작성하는 대신 기존 메소드를 호출해야 합니다. 수퍼클래스로부터 단순히 상속한 메소드를 호출하려면 메소드 이름만 적으면 되지만, 이 클래스에서 엎어쓰기된 수퍼클래스의 원래 메소드를 호출하려면 super 키워드를 이용해야 합니다. 가령 ArrayIntList에 add 메소드가 있고 서브클래스인 SortedIntList가 add 메소드를 엎어쓰기한 상황에서, SortedIntList의 어떤 메소드를 구현하는 과정에서 수퍼클래스인 ArrayIntList의 add 메소드를 호출하고 싶다면 supper.add라고 호출해야 합니다.
* 클래스와 메소드에 표준 형식에 맞도록 주석을 반드시 적어 주어야 합니다. 주석 내의 설명문은 간단하게, 그리고 읽는 사람이 그 뜻을 분명히 알 수 있게 완전한 문장으로 적어야 합니다.

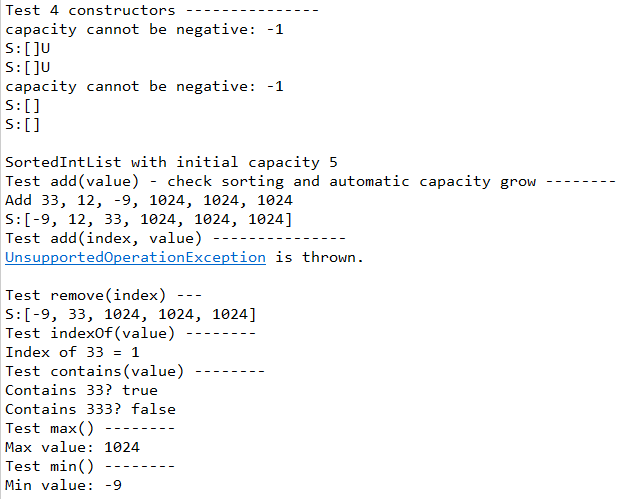
**단계적 프로그램 개발 – 이 문제는 다소 복잡하므로 세 단계에 걸쳐 개발할 것을 권장합니다.**

1. 첫 버전에서는 원소중복금지 여부에 대해 신경 쓰지 않고 중복이 허용되도록 코드를 작성합니다. 일단 원소들이 오름차순으로 정렬되도록 하고, 이분탐색 알고리즘을 사용하여 탐색 속도를 올리는 데 집중합니다. 이를 위해 아래와 같은 작업이 필요합니다.

* w7.sortedlist1 패키지를 만들고 그 아래에 SortedIntList 클래스를 작성한다. SortedIntList클래스에 아래와 같은 코드를 작성한다.
  + import w7.stutterlist.ArrayIntList;
    - 원소의 중복을 금지하는지 여부를 나타내는 필드를 하나 선언합니다. 이름은 uniqueness로 지읍시다.
  + 생성자를 작성한다.
  + 항상 오름차순을 유지하도록 add(int value) 메소드를 엎어쓰기한다. 원소를 어느 위치에 추가할지 결정하고 그 위치에 원소를 삽입한다. 원소를 추가할 위치를 결정할 때 이분탐색이 필요하다.
    - public void add(int index, int value) 메소드를 엎어쓰기한다.정렬 상태를 유지하는 리스트에 원소를 추가할 때 특정 값을 특정 위치에 추가하라고 요구하는 것은 타당하지 않으므로 UnsupportedOperationException을 던지고 아무 일도 하지 않도록 엎어쓰기합니다. 이 메소드 호출에 의해서는 리스트에 어떤 변경도 일어나지 않아야 합니다.
  + indexOf 메소드를 엎어쓰기한다. 이 메소드도 이분탐색을 사용하여 구현한다.
  + max, min 메소드를 작성한다.
  + toString 메소드를 엎어쓰기한다.
* SimpleTest.java를 만들고 main 메소드를 두어 SortedIntList의 기본 동작을 적절히 테스트한다.

[SimpleTest 예](src\w7.sortedlist1.SimpleTest.java)

위 SimpleTest의 실행 결과 화면을 예로 보이면 아래와 같습니다. 학생의 테스트 코드와 출력은 아래 그림과 같지 않아도 됩니다. 나름대로 테스트를 해 보세요.



위 작업을 완료하면 아래 작업을 추가로 수행합니다. 우선 **단위테스트**에 대해 설명합니다.

**<단위테스트 (Unit Test)>**

단위테스트(unit test)라는 개념이 있습니다. 애플리케이션 전체를 테스트하기 전에 작은 각 단위(메소드나 클래스)를 먼저 테스트한다는 의미입니다. 단위테스트 방식으로 클래스의 메소드를 테스트하는 것을 main 메소드에서 클래스의 각 메소드를 테스트하는 것과 비교하면 두 가지 차이가 있습니다.

(1) 메인 메소드에서 여러 가지 경우를 테스트하게 되면 테스트 중간에 메소드 동작에 문제가 발견되는 경우 프로그램 실행이 끝나 버립니다. 그래서 문제가 발견된 테스트 문장 아래에 있는 다른 테스트 문장들은 실행되지 않습니다. 반면 단위테스트 방식으로 테스트를 하면 테스트 중간에 문제가 발견되더라도 거기서 중단하지 않고 테스트 프로그램이 끝까지 실행됩니다. 프로그램 실행이 끝나고 나면 전체 테스트 중 어떤 곳에 문제가 있었는지를 알 수 있습니다.

(2) 단위테스트를 이용하면 테스트를 자동화할 수 있습니다. 큰 애플리케이션을 개발할 때, 애플리케이션을 구성하는 클래스들을 하나하나 작성해 나가면서 각 클래스의 메소드를 차례로 작성하는데, 클래스를 작성할 때마다 그 클래스의 메소드를 테스트하는 단위테스트 프로그램을 함께 작성하고 테스트합니다. 개발 진행에 따라 클래스와 그 클래스들에 대한 단위테스트 프로그램들이 많이 만들어지면, 이 단위테스트 프로그램들을 한꺼번에 실행시킬 수 있습니다. 새로운 클래스가 추가로 개발되거나 코드 중 일부에 수정이 있을 때마다 단위테스트 프로그램 전체를 돌려 봄으로써 클래스 추가나 일부 수정이 기존 프로그램에 새로운 에러를 유발하지 않는지 확인할 수 있습니다. 코드 일부 변경에 의해 기존 프로그램에 새로운 에러가 생기는 경우, 어느 부분에서 어떤 문제가 생겼는지도 쉽게 확인할 수 있게 됩니다.

단위테스트를 돕기 위해 미리 만들어 놓은 라이브러리 클래스들이 있습니다. 이 클래스들은 표준라이브러리에 포함되어 있지 않기 때문에 프로젝트에서 단위테스트를 하려면 라이브러리를 프로젝트에 추가해 주어야 합니다. 라이브러리 이름은 JUnit입니다. 아래와 같은 절차를 따라 JUnit 라이브러리를 프로젝트에 추가하세요.

Package Explorer에서 프로젝트를 오른쪽 버튼 클릭 – Build Path – Add Libraries… - JUnit 선택 – Next – Finish.

이렇게 하고 나면 프로젝트에 아래 그림과 같이 JUnit5 라이브러리가 추가된 것을 Package Explorer에서 확인할 수 있습니다.

Text

Description automatically generated with medium confidence

이렇게 해서 단위테스트를 할 수 있는 환경을 갖추었습니다.

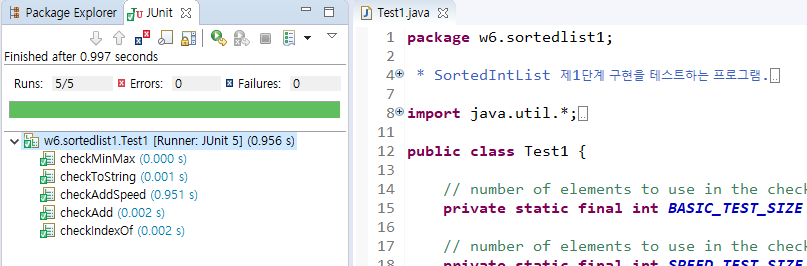
단위테스트를 하려면 테스트를 수행하는 테스트 클래스를 작성해야 합니다. 단위테스트에서는 테스트 메소드 여러 개를 묶어 테스트 클래스로 작성합니다. SortedIntList에 대해서도 테스트 클래스를 작성해야 하지만 테스트 클래스까지 작성하려면 시간이 너무 많이 걸릴 것이므로 이미 완성된 테스트 클래스를 제공해 주겠습니다. [Test1.java](src\Test1.java)는 위에서 일 단계로 작성한 SortedIntList 클래스의 여러 메소드들이 제대로 작동하는지 테스트하는 테스트 클래스 파일입니다. 이 클래스를 w7.sortedlist1 패키지에 추가하세요.

Test1 클래스를 Junit 테스트로 실행해 보세요. 아래 그림의 방법으로 실행할 수 있습니다.

Graphical user interface, text, application

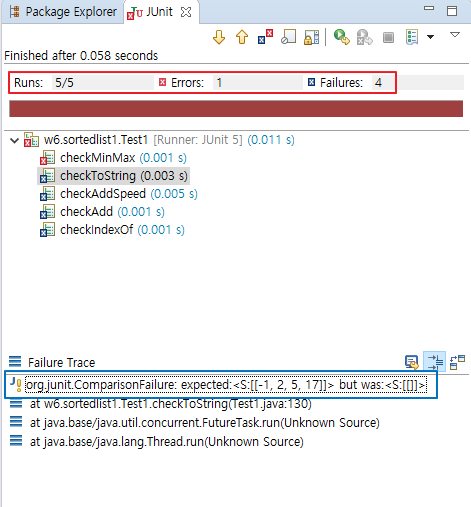
Description automatically generated

SortedIntList 클래스의 메소드들을 모두 제대로 완성했으면 아래 그림과 같이 녹색 줄이 나타납니다.



녹색 줄 아래에는 Test1 클래스에 들어 있는 다섯 개의 테스트 메소드들이 보입니다. 녹색 줄은 SortedIntList 메소드들이 이 다섯 개의 테스트를 무사히 “통과”했다는 뜻입니다. 다섯 개의 테스트 메소드에도 녹색 체크 표시가 보입니다.

테스트를 통과하지 못한 경우의 예를 들면 아래와 같습니다.



위 그림의 빨간 네모 부분은, Test1 클래스에 있는 다섯 개의 테스트 메소드가 실행되었는데, 그 중 한 개에서는 Error가, 나머지 네 개에서는 Failure가 나왔다는 의미입니다. Failure는 코드를 테스트한 결과 코드의 기능에 문제가 있어 테스트를 통과하지 못한 것이고, Error는 테스트 메소드 자체가 제대로 실행될 수 없었다는 의미입니다. Error는 주로 테스트 메소드 자체의 오류에 의해 나타나지만 SortedIntList 메소드에 문제가 있어 발생할 수도 있습니다. 다섯 개의 Failure 혹은 Error 중 하나를 선택하면 그림의 아랫부분에 그 Failure나 Error가 발생하게 된 과정을 보여줍니다. 위 그림의 경우, checkToString 테스트에서 Failure가 나왔는데 그 이유가 파란 네모 부분에 설명되어 있습니다. SortedIntList의 toString 메소드 실행 결과 S:[[-1, 2, 5, 17’]이 나와야 하는데 (expected), 실제로는 S[[]]가 나왔(was)기 때문에 테스트를 통과하지 못했다는 설명입니다. 그리고 그 아래에는 stack trace가 보입니다.

1. 위 1에서 만든 코드를 수정하여 원소중복금지 여부에 따라 작동이 달라지게 합니다. 우선 w7.sortedlist1 패키지를 복사하여 w7.sortedlist2 패키지를 만들고 새 패키지에서 작업하세요. 원소중복금지가 설정되어 있는 경우에는 중복된 원소를 추가하지 않도록 add 메소드를 수정합니다. 그리고 SimpleTest.java에 필요한 문장을 추가하여 기본 동작을 테스트합니다.

[SimpleTest 예](src\w7.sortedlist2.SimpleTest.java)

위 SimpleTest의 실행 결과 화면을 예로 보이면 아래와 같습니다. 학생의 테스트 코드와 출력은 아래 그림과 같지 않아도 됩니다. 나름대로 테스트를 해 보세요. (패키지를 복사하는 과정에서 SimpleTest.java 파일의 첫 부분에 import w7.sortedlist1.SortedIntList; 문장이 자동으로 삽입되는 수가 있습니다. 이 문장이 있는지 확인하고 있으면 지우세요.)

텍스트이(가) 표시된 사진

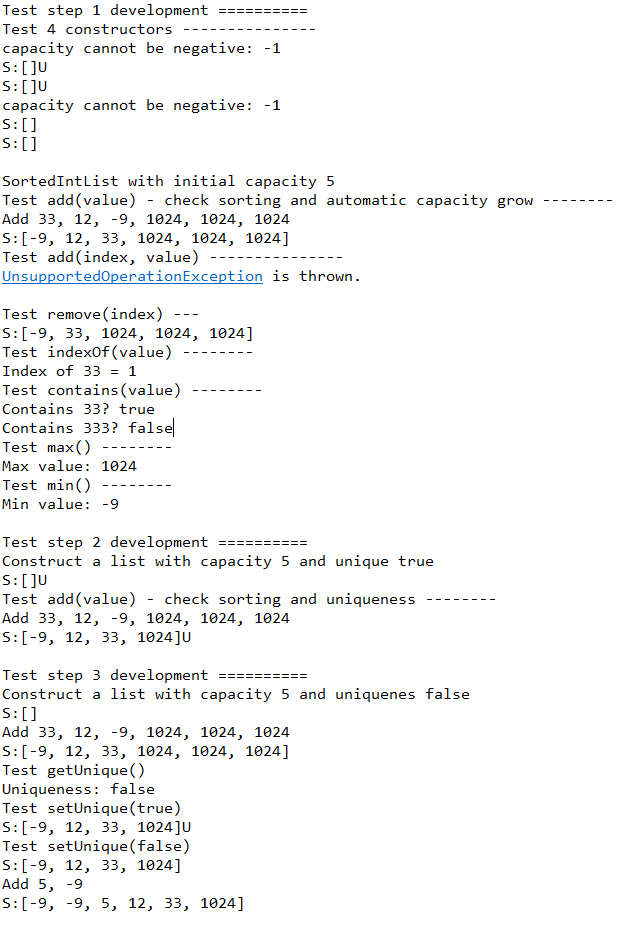
자동 생성된 설명

위 작업을 통해 기본적인 확인을 했으면 이제 Junit 테스트 클래스인 [Test2.java](src\Test2.java) 파일을 패키지에 붙여 넣고 단위테스트를 실행해 보세요. 기존의 Test1과 추가된 Test2를 모두 통과해야 합니다.

1. 위 2에서 만든 SortedIntList에 getUnique, setUnique 메소드를 추가합니다. w7.sortedlist2 패키지를 복사하여 w7.sortedlist3 패키지를 만들고 새 패키지에서 작업하세요. 사용자가 setUnique 메소드를 이용하여 원소중복금지를 설정하는 경우 현재 리스트에 있는 모든 원소 중복을 해소하도록 해야 합니다. 그리고 SimpleTest.java에 필요한 문장을 추가하여 기본 동작을 테스트합니다.

[SimpleTest 예](src\w7.sortedlist3.SimpleTest.java)

위 SimpleTest의 실행 결과 화면을 예로 보이면 아래와 같습니다. 학생의 테스트 코드와 출력은 아래 그림과 같지 않아도 됩니다. 나름대로 테스트를 해 보세요. (패키지를 복사하는 과정에서 SimpleTest.java 파일의 첫 부분에 import w7.sortedlist2.SortedIntList; 문장이 자동으로 삽입되는 수가 있습니다. 이 문장이 있는지 확인하고 있으면 지우세요.)



위 작업을 통해 기본적인 확인을 했으면 이제 Junit 테스트 클래스인 [Test3.java](src\Test3.java) 파일을 패키지에 붙여 넣고 단위테스트를 실행해 보세요. 기존의 Test1, Test2와 새로 추가된 Test3를 모두 통과해야 합니다.

소프트웨어를 개발할 때 기능을 추가하거나 변경하고 나면 그에 대한 테스트를 해야 합니다. 그런데 그런 기능 추가나 변경이 기존에 있던 소프트웨어에 예기치 않은 문제를 야기할 수도 있습니다. 그래서, 추가/변경된 기능에 대한 테스트 뿐만 아니라 그 동안 처음부터 개발하면서 만들어 놓았던 테스트 프로그램들을 모두 실행하여 기존에 잘 되던 소프트웨어에 여전히 문제가 없는지 확인을 해야 합니다. 이렇게 추가/변경이 있을 때마다 그 동안 작성해 놓았던 모든 테스트 프로그램들을 돌려 문제가 없는 확인하는 작업을 리그레션 테스트(regression test)라고 부릅니다. 처음으로 돌아가 다시 테스트를 수행한다는 의미입니다.

위에서 우리가 사용한 테스트 케이스(Test1, Test2, Test3)들은 꽤 심도 있게 테스트를 하도록 개발됐지만 이 테스트를 통과한다고 해도 여전히 프로그램이 완전하다고 보장하지는 않습니다. 어떤 테스트도 완벽한 테스트를 하지는 못합니다.

**강온실 w7p3\_SortedIntList 문제의 답으로 완성된 SortedIntList와 ArrayIntList를 입력하세요. ArrayIntList는 w7.stutterlist 패키지에 있는 것을 입력하세요. Import 문장은 넣지 마세요. java.util.Arrays, java.util.NoSuchElementException이 문제 자체에 이미 import되어 있습니다. UnsupportedOperationException은 java.lang 패키지에 들어 있기 때문에 import하지 않아도 됩니다.**

끝.