

객체지향 프로그래밍 및 실습

7주차. Polymorphism & Interfaces

1. Abstract Classes & methods

■ 추상 클래스 & 메소드

- 추상 클래스는 오직 super class로 동작하기 위해서 존재한다.
- 따라서 추상 클래스를 인스턴스화 할 수는 없다.

1. 객체를 만들 수 있는 실체 클래스들 내 공통적인 특성(필드, 메소드)의 이름을 통일할 목적으로 사용.

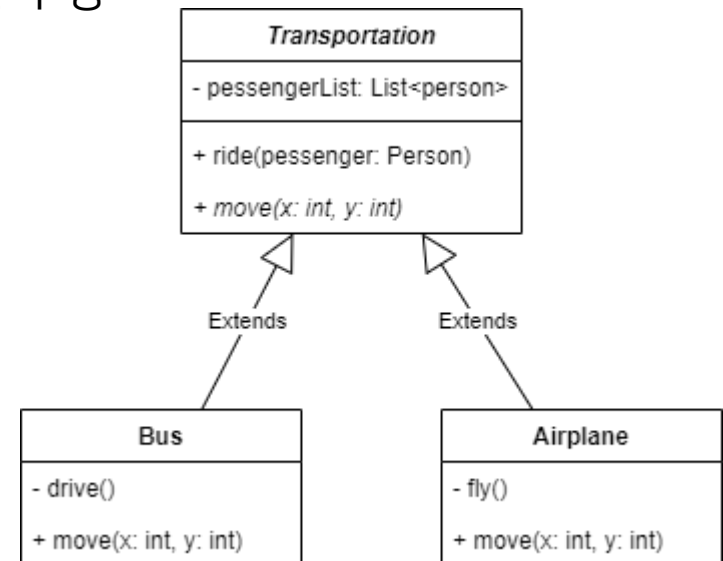
Ex) Smart Phone이라는 class를 정의하고자 한다. 개발자 입장에서 전원을 켤 때 호출되는 메소드의 이름을 powerOn()이거나 turnOn()로 설계할 수 있다. 이렇게 동일한 기능을 수행하는 메소드에 대하여 이름이 달라 사용방법이 달라질 수 있는데 이를 방지하고자 한다.

2. 설계자는 개발자에게 코딩해야하는 부분을 직접적으로 알려줄 수 있으며 개발자는 클래스를 작성할 때 시간을 절약할 수 있다.

1. Abstract Classes & methods

■ 추상 클래스 & 메소드

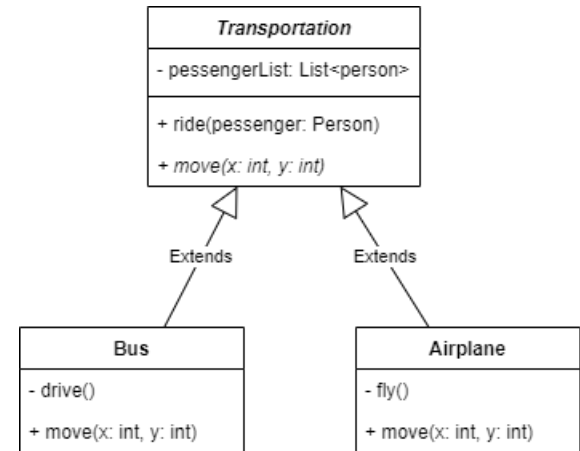
- 추상 메소드는 하위 클래스에서 무조건 override 해야 한다.
 - 그렇지 않으면, 컴파일되지 않는다. → 컴파일러의 강제사항
- 추상 메소드를 포함하는 클래스는 무조건 추상 클래스로 선언되어야 한다.
- UML에서는 추상 클래스 & 메소드를 기울임꼴로 표현
- Transportation은 상위 클래스로서의 역할만 수행
 - 인스턴스화 될 필요가 없음
 - 따라서 추상 클래스로 선언되는 것이 옳다.
- ride()는 모든 클래스가 동일하므로,
- move()만 추상 메소드로 선언
- Bus, Airplane은 무조건 move()를 Override해야 한다.



1. Abstract Classes & methods

- 추상 클래스 & 메소드
 - 추상 메소드는 method body가 필요 없음.

```
1 package abstractclass;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 public abstract class Transportation {
7     private List<Person> pessengerList = new ArrayList<>();
8
9     public void ride(Person pessenger) {
10         this.pessengerList.add(pessenger);
11     }
12
13     public abstract void move(int x, int y);
14 }
```



```
1 package abstractclass;
2
3 public class Bus extends Transportation {
4
5     private void drive() {
6         System.out.println("버스는 운전해서 간다.");
7     }
8
9     @Override
10    public void move(int x, int y) {
11        drive();
12        System.out.printf("%d, %d로 이동한다!\n", x, y);
13    }
14
15 }
```

1. Abstract Classes & methods

■ 추상 클래스의 생성자

- 추상 클래스의 생성자는 **protected**로 선언하는 것을 권장
- Code Smell: 코드에서 더 심오한 문제를 일으킬 가능성이 있는 프로그램 소스코드의 특징을 가리킨다.(위키피디아)

Constructors of an "abstract" class should not be declared "public" (java:S5993)

🔍 Code Smell ⬆ Major

Abstract classes should not have public constructors. Constructors of abstract classes can only be called in constructors of their subclasses. So there is no point in making them public. The `protected` modifier should be enough.

Noncompliant Code Example

```
public abstract class AbstractClass1 {
    public AbstractClass1 () { // Noncompliant, has public modifier
        // do something here
    }
}
```

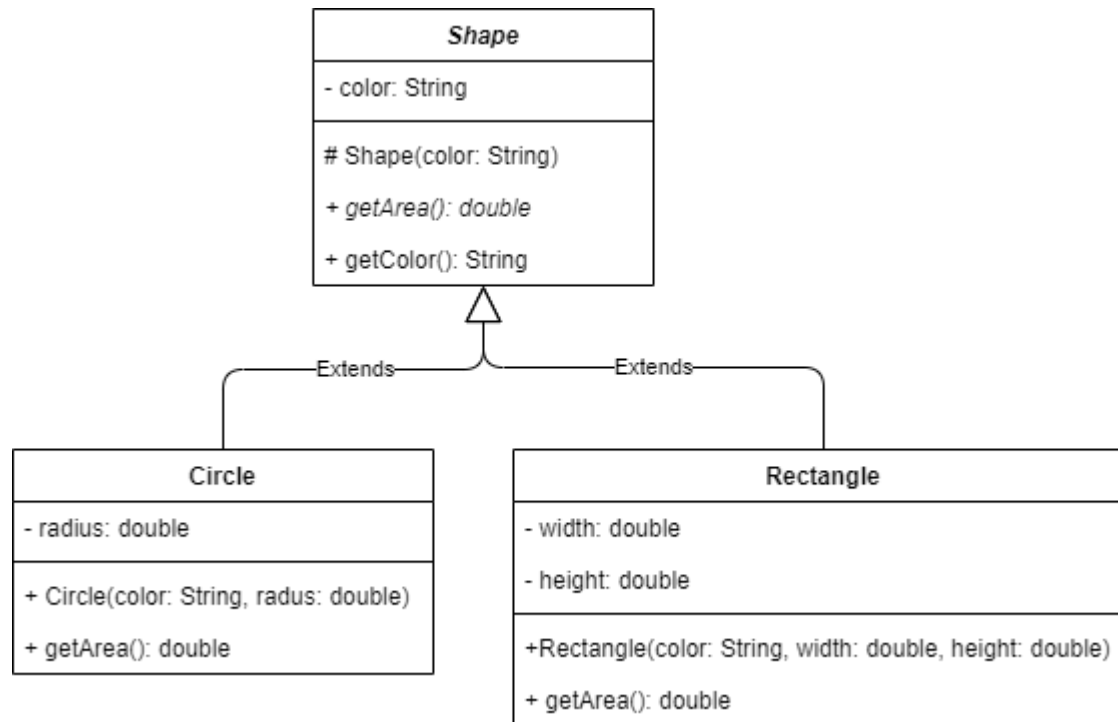
Compliant Solution

```
public abstract class AbstractClass2 {
    protected AbstractClass2 () {
        // do something here
    }
}
```

1. Abstract Classes & methods

■ 실습 문제 1

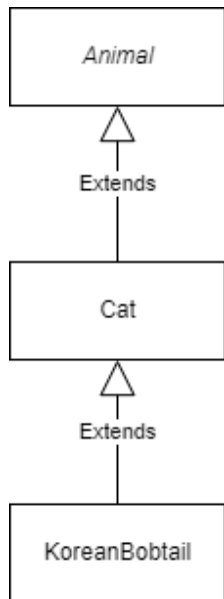
- 아래의 클래스 다이어그램을 참조하여 코드를 구현해보자.
- ShapeTest 클래스에 main 메서드를 생성하고, Circle과 Rectangle 객체를 생성하여 getArea(), getColor()의 반환 값을 출력해보자.



2. Polymorphism

■ 다형성

- Java에서는 어느 객체가 여러가지의 자료형을 가질 수 있는 능력을 의미
- KoreanBobtail(한국 고양이 품종)의 사례
- KoreanBobtail is-a Cat.
- But, Cat is-not-a KoreanBobtail.



```
1 Animal cat1 = new Cat();
2 Animal cat2 = new KoreanBobtail();
3
4 Cat cat3 = (Cat) cat1;
5 Cat cat4 = (Cat) cat2;
6
7 KoreanBobtail cat6 = (KoreanBobtail) cat2;
8
9
10 // 불가능한 코드
11 KoreanBobtail cat7 = new Cat();
```

2. Polymorphism

■ 메서드 인자에서의 다형성

- 메서드의 argument에서도 활용 가능
- print, println 메서드에서의 활용 사례
- 참고사항: Java의 모든 클래스는 Object라는 클래스를 상속받고 있다.
- <https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Object.html>

```
/**
 * Prints an Object and then terminate the line. This method calls
 * at first String.valueOf(x) to get the printed object's string value,
 * then behaves as
 * though it invokes {@link #print(String)} and then
 * {@link #println()}.
 *
 * @param x The {@code Object} to be printed.
 */
public void println(Object x) {
    String s = String.valueOf(x);
    synchronized (this) {
        print(s);
        newline();
    }
}
```

Method Summary

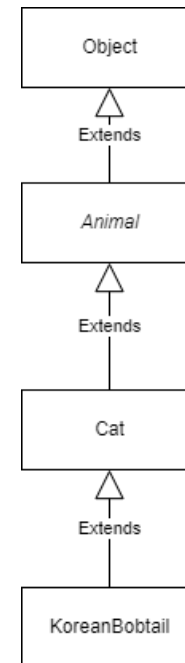
All Methods	Instance Methods	Concrete Methods	Depr
Modifier and Type	Method		
protected Object	clone()		
boolean	equals(Object obj)		
protected void	finalize()		
Class<?>	getClass()		
int	hashCode()		
void	notify()		
void	notifyAll()		
String	toString()		
void	wait()		
void	wait(long timeoutMillis)		
void	wait(long timeoutMillis, int nanos)		

2. Polymorphism

■ 메서드 인자에서의 다형성

- print, println 메서드는 Object 타입의 객체를 받아, toString()의 반환 값을 출력해준다.
- 따라서 아래 코드가 작동 가능
- Cat is-a Object 이기 때문에 Subtyping-polymorphism 사용 가능

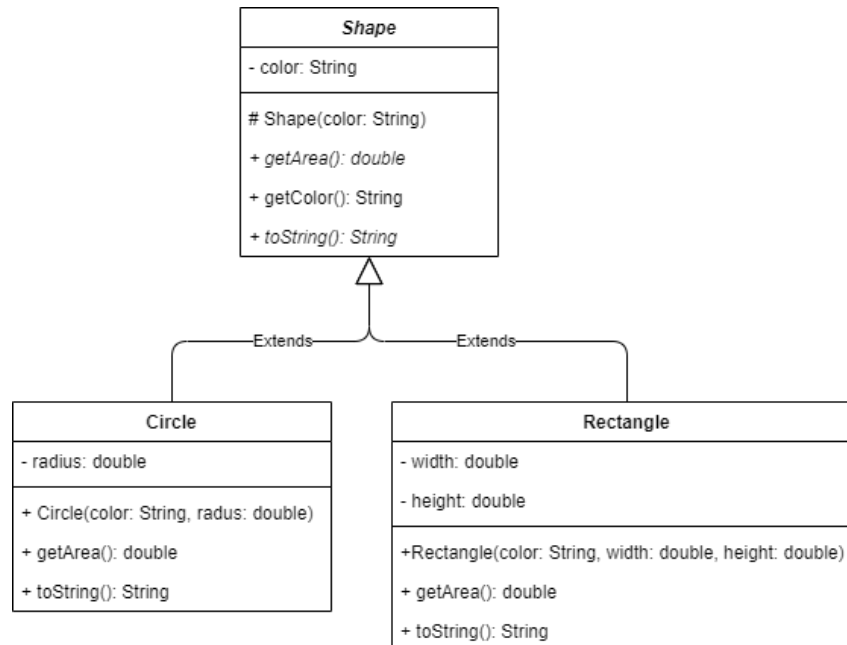
```
1 System.out.println(cat1);  
2 System.out.println(cat2);
```



2. Polymorphism

■ 실습 문제 2

- 아래의 클래스 다이어그램을 참조하여 코드를 구현해보자.
- ToStringTest 클래스에 main메서드를 생성하고 print, println에 Circle, Rectangle 객체를 넣어보자.
- Circle.toString() => "Circle r=<radius 값>" 반환
- Rectangle.toString() => "Rectangle w=<width 값> h=<height 값>" 반환



2. Polymorphism

■ 어느 클래스의 객체일까?

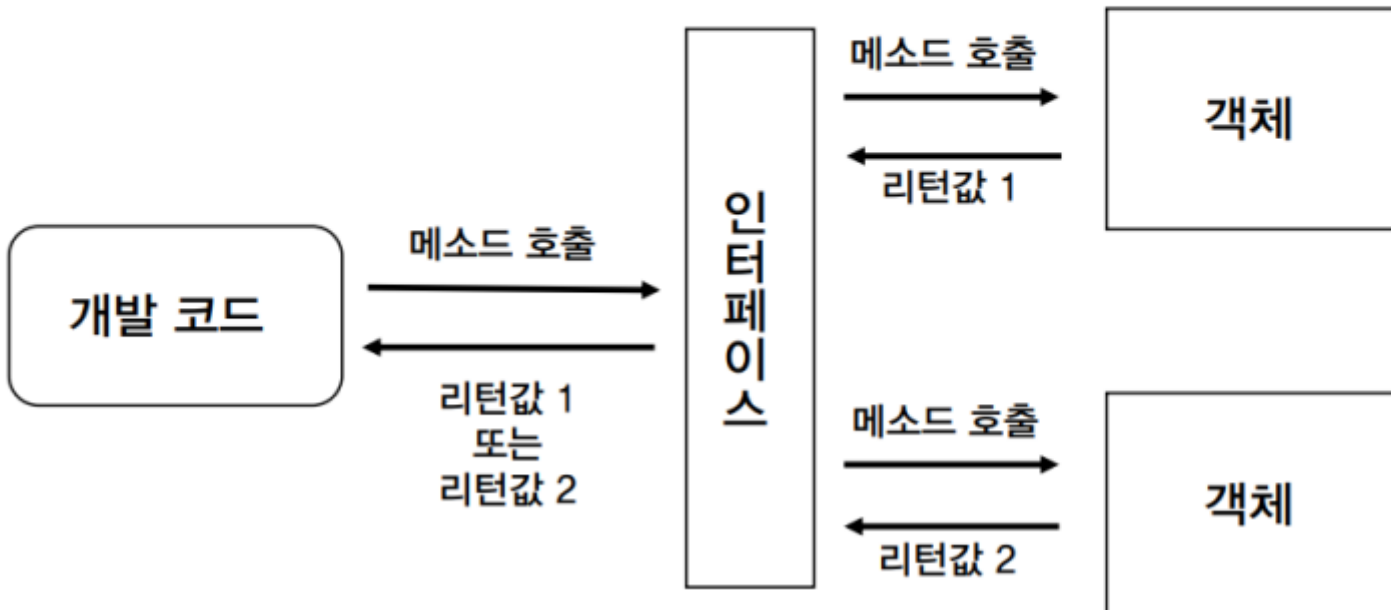
- `Animal cat1 = new Cat();`
- `Animal cat2 = new KoreanBobtail();`
- `someMethod(cat1, cat2);`
- └ `someMethod` 내부에서 `cat1`과 `cat2`가 어떤 클래스의 객체인지 알 수 있는 방법은?
 - `InstanceOf` 키워드
 - `cat1.getClass().getName()`
 - `cat1.getClass() == KoreanBobtail.class`
 - 등등

```
1 boolean bool = cat1 instanceof KoreanBobtail
```

3. Interfaces

■ 인터페이스란?

- 서로 다른 장치들이 연결되어서 상호 데이터를 주고받는 **규격**을 의미한다.
- 즉, 자바에서는 객체의 사용 방법을 정의한 클래스이다.
- 하나의 메소드 호출로 실행 내용과 리턴 값을 다양화 할 수 있다.



3. Interfaces

■ 2. 인터페이스 선언

- class 키워드 대신에 interface 키워드를 사용한다.
- 상수와 method만을 구성 멤버로 가진다.

■ 3-1. 상수 선언

- 고정된 값으로 런타임 시에 데이터를 바꿀 수 없다.
- 반드시 초기값을 대입해야 한다.

```
// 3-1. 상수의 선언 : public static final type 상수명 = 값;  
public static final int constant = 2021;
```

■ 3-2. 정적 메소드(Static method) 선언

- 클래스의 정적 메소드와 완전 동일하다.
- 따라서 객체 생성 없이 메소드를 호출할 수 있다.

```
// 3-2. 정적 메소드의 선언 : public static 리턴타입 메소드명 (매개변수 ...){};  
public static void staticMethod(int para1, String para2){};
```

3. Interfaces

■ 3-3. 추상 메소드(Abstract method) 선언

- 인터페이스를 통해 호출된 메소드는 최종적으로 객체에서 실행된다.
- 따라서 인터페이스의 메소드는 추상 메소드로 선언한다.

```
// 3-3. 추상 메소드의 선언 : (public abstract) 리턴타입 메소드명 (매개변수 ...);  
public abstract void abstractMethod(double para3, char para4);  
void abstractMethod2(double para5, char para6);
```

■ 3-4. 디폴트 메소드(Default method) 선언

- 메소드 내용을 명시할 수 있다. 단, 객체가 있어야 호출할 수 있다.
- 기존 인터페이스를 확장해서 새로운 기능을 추가하기 위해 허용되었다.

```
// 3-4. 디폴트 메소드의 선언 : public default 리턴타입 메소드명 (매개변수 ...){};  
public default void defaultMethod(Object para){};
```

3. Interfaces

■ 4. 인터페이스 작성 예시

```
public interface InterfaceTest {  
  
    // 3-1. 상수의 선언 : public static final type 상수명 = 값;  
    public static final int constant = 2021;  
  
    // 3-2. 정적 메소드의 선언 : public static 리턴타입 메소드명 (매개변수 ...){};  
    public static void staticMethod(int para1, String para2){};  
  
    // 3-3. 추상 메소드의 선언 : (public abstract) 리턴타입 메소드명 (매개변수 ...);  
    public abstract void abstractMethod(double para3, char para4);  
    void abstractMethod2(double para5, char para6);  
  
    // 3-4. 디폴트 메소드의 선언 : public default 리턴타입 메소드명 (매개변수 ...){};  
    public default void defaultMethod(Object para){};  
}
```

3. Interfaces

■ 인터페이스 사용 사례 1

- 번역기 개발 팀과제를 진행한다!
- JapaneseTranslator는 동원이가 맡기로 했다.
- EnglishTranslator는 정규가 맡기로 했다.
- 어라 메서드 이름이랑 인자에 통일성이 없고 중구난방이네?
- 나는 이미 코드 다 짰는데...

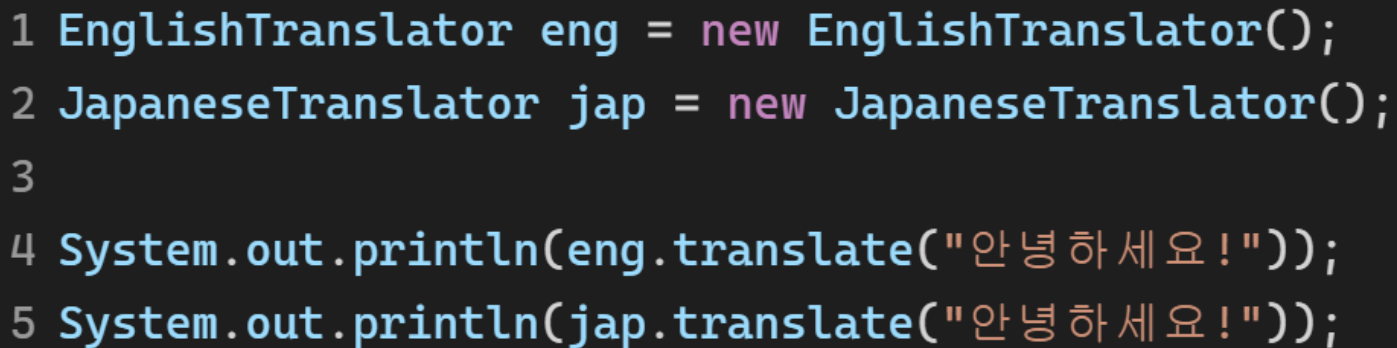
```
1 public class JapaneseTranslator {
2     public String translate(String korean) {
3         return "일본어로 번역된 문장";
4     }
5
6     public String inverseTranslate(String english) {
7         return "한국어로 번역된 문장";
8     }
9 }
```

```
1 public class EnglishTranslator {
2     boolean mode;
3
4     public String koreanToEnglish(String korean, int n) {
5         return "영어로 번역된 문장" + n;
6     }
7
8     public String EnglishToKorean(String english) {
9         if (mode) {
10             return "한국어로 번역된 문장";
11         } else {
12             return null;
13         }
14     }
15
16     public void setMode(boolean mode) {
17         this.mode = mode;
18     }
19 }
```


3. Interfaces

■ 인터페이스 사용 사례 1

- 내가 짠 코드는 아래와 같은데, 동원이가 짠 코드는 잘 작동하고 정규가 짠 코드는 작동을 안한다.
- 정규는 열심히 해보겠다고 이상한 기능들까지 추가해 놓아서, 사용하기에 번잡하다.



```
1 EnglishTranslator eng = new EnglishTranslator();
2 JapaneseTranslator jap = new JapaneseTranslator();
3
4 System.out.println(eng.translate("안녕하세요!"));
5 System.out.println(jap.translate("안녕하세요!"));
```

3. Interfaces

■ 인터페이스 사용 사례 1

- 그렇다면, 코드를 구현하기 전에 다같이 규칙을 세우자!
- 규칙은 인터페이스로!!

```
1 public interface Translatable {
2     /**
3      * 한국어를 외국어로 변환
4      * @param korean 한국어 문장
5      * @return 번역된 외국어 문장
6      */
7     public String translate(String korean);
8
9     /**
10    * 외국어를 한국어로 변환
11    * @param foreignLang 외국어 문장
12    * @return 번역된 한국어 문장
13    */
14    public String inverseTranslate(String foreignLang);
15 }
```

3. Interfaces

■ 인터페이스 사용 사례 1

- 인터페이스를 implements를 하게 되면, 컴파일러 수준에서 필수 구현 메서드를 강제할 수 있다.

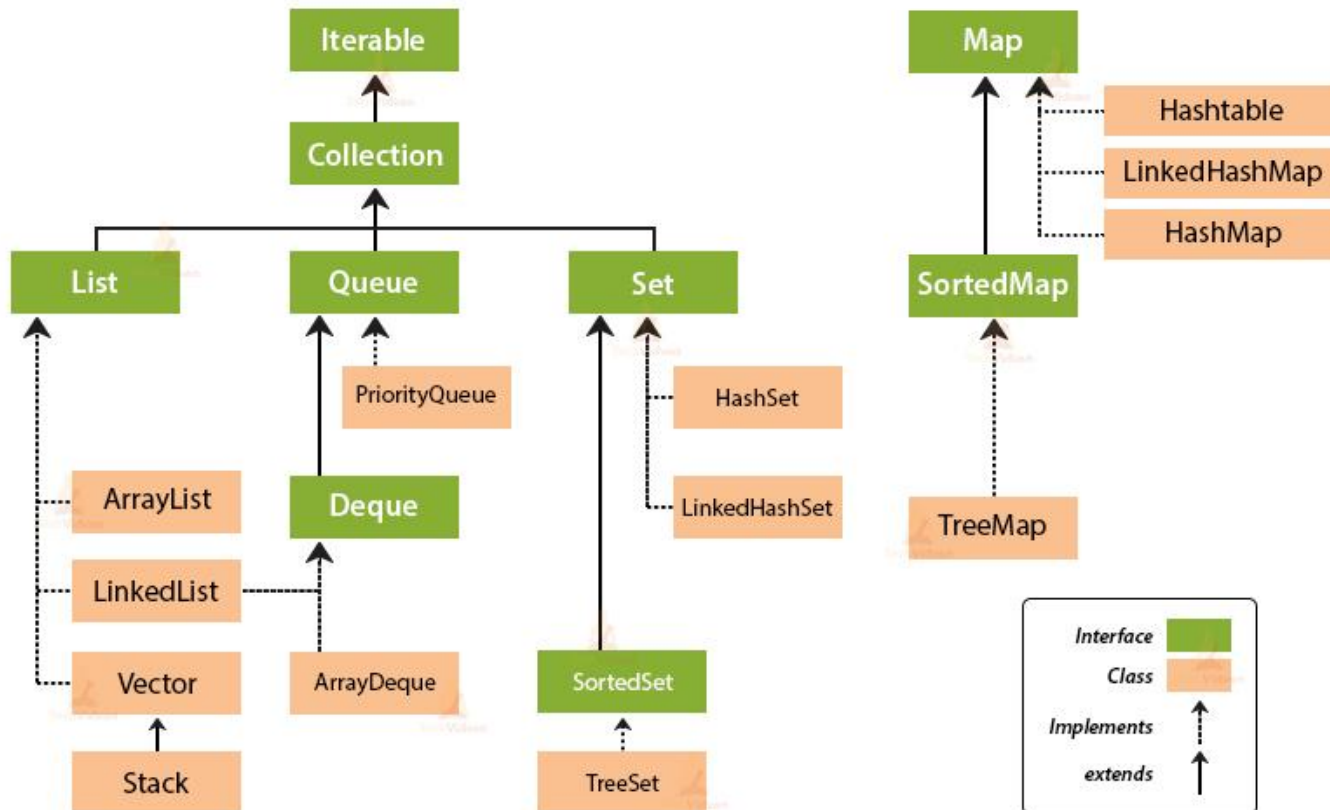
```
1 public class EnglishTranslator implements Translatable {
2     @Override
3     public String translate(String korean) {
4         return "영어로 번역된 문장";
5     }
6
7     @Override
8     public String inverseTranslate(String english) {
9         return "한국어로 번역된 문장";
10    }
11 }
```

```
1 public class JapaneseTranslator implements Translatable {
2     @Override
3     public String translate(String korean) {
4         return "일본어로 번역된 문장";
5     }
6
7     @Override
8     public String inverseTranslate(String japanese) {
9         return "한국어로 번역된 문장";
10    }
11 }
```

3. Interfaces

- 인터페이스 사용 사례 2
 - Collection 프레임워크에서의 활용

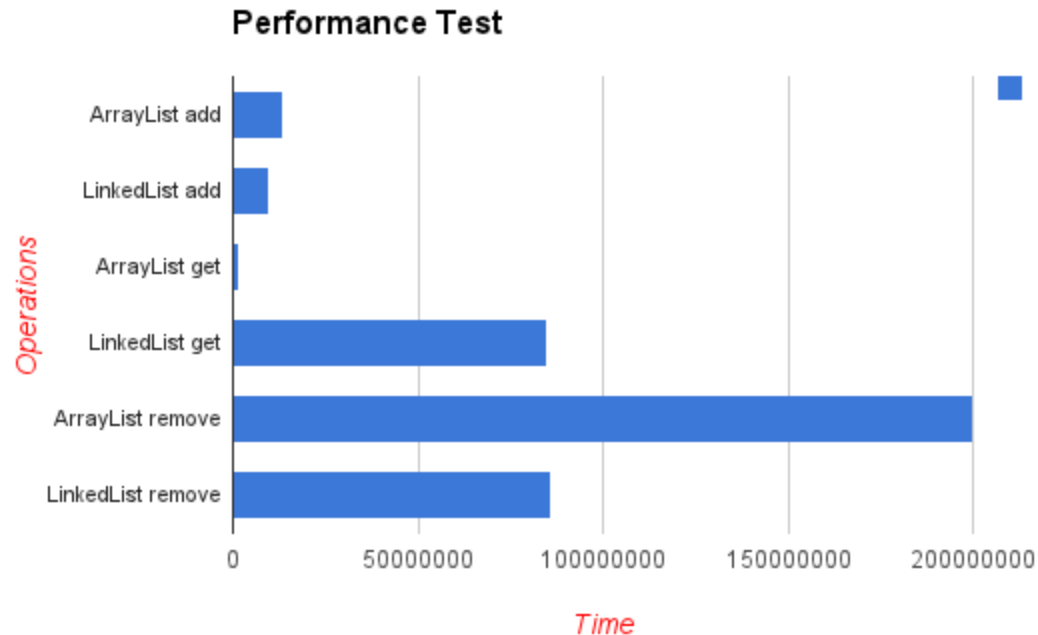
Collection Framework Hierarchy in Java



3. Interfaces

■ 인터페이스 사용 사례 2

- ArrayList, LinkedList, Vector 클래스는 다 똑같이 List 기능을 수행할 수 있다.
- 다만 순회 속도, 참조 속도, 삽입 속도, 세부적인 기능에서 약간의 차이가 있다.



3. Interfaces

■ 인터페이스 사용 사례 2

- 정렬하는 메서드에서 굳이 ArrayList, LinkedList, Vector를 가려서 받을 필요가 있을까?
- 세 클래스 모두 다 List로써의 역할을 수행할 수 있기 때문에 저마다의 정렬 메서드가 필요하지는 않다.

```
1 public class ListJob {  
2     public void sort(ArrayList<?> list) {  
3         // do sorting ...  
4     }  
5 }
```

3. Interfaces

■ 인터페이스 사용 사례 2

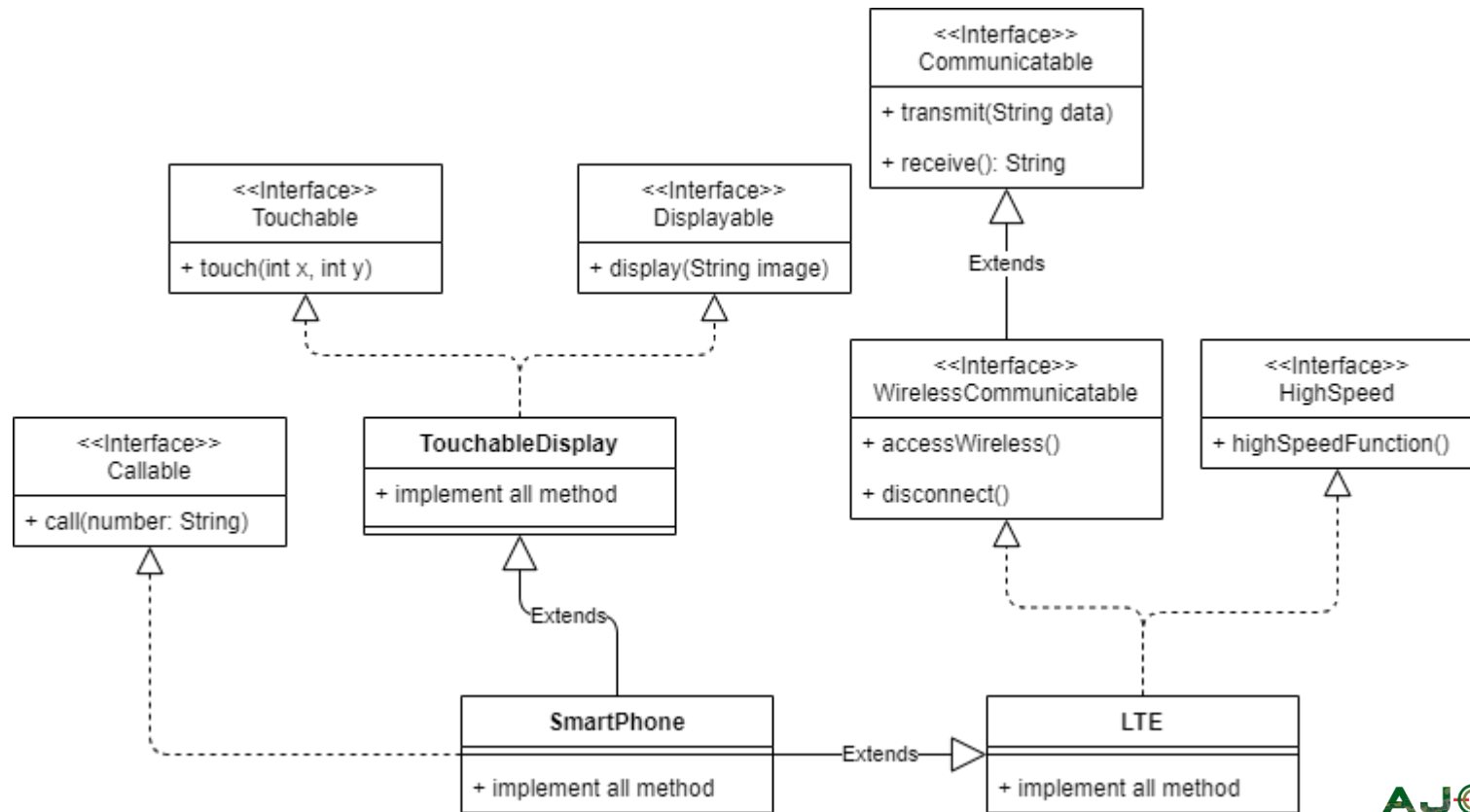
- 굳이 가려서 받을 필요가 없는 경우에는 super type을 받도록 하는 것이 좋다.
- 알면 좋은 것: 리스코프 치환 원칙
 - <https://steady-coding.tistory.com/383>

```
1 public class ListJob {  
2     public void sort(List<?> list) {  
3         // do sorting ...  
4     }  
5 }
```

3. Interfaces

■ 실습 문제 3

- 아래의 클래스 다이어그램을 참조하여 코드를 구현해보자.
- 각 메서드는 print 이용하여 간략하게 구현
- `transmit(String data) => printf("%d를 전송했습니다.\n", data);`



4. Final method & classes

■ Final 키워드

- 필드에 붙으면? 재할당 불가능
- 메서드에 붙으면? 오버라이딩 불가능
- 클래스에 붙으면? 상속 불가능

```
1 public class ChessSimulator {
2     private Player white;
3     private Player black;
4
5     // some code ...
6
7     public final Player getFirstPlayer() {
8         return white;
9     }
10 }
```

체스에서는 무조건 흰색이 첫수를 둔다.

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

    /**
     * The value is used for character storage.
     *
     * @implNote This field is trusted by the VM, and is a subject to
     * constant folding if String instance is constant. Overwriting this
     * field after construction will cause problems.
     *
     * Additionally, it is marked with {@link Stable} to trust the contents
     * of the array. No other facility in JDK provides this functionality (yet).
     * {@link Stable} is safe here, because value is never null.
     */
    @Stable
    private final byte[] value;

    /**
     * The identifier of the encoding used to encode the bytes in
     * {@code value}. The supported values in this implementation are
     *
     * LATIN1
```

String은 immutable 속성을 가지고 있다.