第三回　演習問題（抽象クラス，インタフェース）

諸注意

- 「Arms.java」，「Item1.java」，「Usable.java」，「Item2.java」，「Sword.java」，「Cane.java」，「Harb.java」，「JokeTester.java」を Web から提出する（８ファイルを提出する場合，８回に分けて提出処理を行う）．
- コピペ発覚時は見せた側も見せてもらった側も両方０点とする．
- 必ずコンパイルエラーのない状態で提出すること（自動採点したいのでコンパイルエラーがあると，全て０点になってしまう）．
- 課題の途中で提出することになった場合，コンパイルエラーさえ出なければ，課題の途中の状態で提出してくれて構わない．一部のメソッドだけが実現できていない場合，コンパイルエラー出ないならばそのままの状態で提出してくれてよい．
- 主にコンソール出力で評価しているため，デバッグに用いたようなコンソール出力が残っていないように気をつけること．基本的にコンソール出力を指定しない限りは，課題内でコンソール出力はないものとする．
- Package は使わないこと（デフォルトパッケージで実装する）．Package で実装すると，自動採点がうまくいきません．

課題 1

1-1	問題設定	ロールプレイングゲームの武器をイメージしてほしい．色々な武器を考えた結果，抽象クラス Arms を作成し，それぞれの武器がこれを継承することで，統一性のある武器クラスを開発できると考えた．以下の条件で Arms を開発せよ．ただしフィールドは全て隠蔽（自身を継承した子クラスからはアクセスできるように）し，メソッドは他のパッケージからでもアクセスできるようにせよ．
	フィールド	name: 武器の名称(文字列型) ap: AttackPoint: 攻撃力（数値型） map: MagicAttackPoint: 魔法攻撃力（数値型）
	コンストラクタ	1．全てのフィールドの初期値を設定する．
	メソッド	int getAp(): 攻撃力を取得するメソッド 　引数　：なし 　戻り値：攻撃力 int getMap(): 魔法攻撃力を取得するメソッド 　引数　：なし 　戻り値：魔法攻撃力 String toString(): 武器の詳細を確認するメソッド 　引数　：なし 　戻り値：「武器名(攻撃力,魔法攻撃力)」 　※括弧（）とカンマ，は半角を用いること． 　※不要なスペース，半角スペースは表示しないこと． void attackEffect(): 攻撃のエフェクトを表示するメソッド 　引数，戻り値：なし 　※抽象メソッドとする．

1-1：解答例

```
// 抽象メソッドattackEffect()を含む抽象クラス
// なのでabstractをつける.
public abstract class Arms {
	// フィールドは自分を継承した子クラスから
	// アクセスできるようにprotectedとする.
	protected String name = "";
	protected int ap = 0;
	protected int map = 0;

	// コンストラクタは例に従って全てを初期化する.
	public Arms(String name, int ap, int map){
		this.name = name;
		this.ap = ap;
		this.map = map;
	}

	// ただのgetter()
	public int getAp(){
		return this.ap;
	}

	// ただのgetter()
	public int getMap(){
		return this.map;
	}

	// 本インスタンスの概要を示すためのtoString()
	public String toString(){
		// 文字列はとりあえず+で連結する.
		return this.name + "(" + this.ap + "," + this.map + ")";
	}

	// 抽象メソッドとして継承先に実装させるため
	// abstractをつける.
	public abstract void attackEffect();
}
```

1-2	問題設定	ロールプレイングゲームの道具を管理する抽象クラスItem1を開発せよ．ただしフィールドは全て隠蔽（自身を継承した子クラスからはアクセスできるように）し，メソッドは他のパッケージからでもアクセスできるようにせよ．
	フィールド	name: 道具の名称(文字列型) description: 道具の説明(文字列型)
	コンストラクタ	1．全てのフィールドの初期値を設定する．
	メソッド	String toString(): 道具の詳細を確認するメソッド 引数　：なし 戻り値：「道具名(道具の説明)」 ※括弧（）は半角を用いること． ※不要なスペース，半角スペースは表示しないこと． void useEffect(): 道具使用のエフェクトを表示するメソッド 引数，戻り値：なし ※抽象メソッドとする．

1-2：解答例

```
// 考え方はArmsと同じなのでコメントは省略する．
public abstract class Item1 {
	protected String name = "";
	protected String description = "";

	public Item1(String name, String description){
		this.name = name;
		this.description = description;
	}

	public String toString(){
		return this.name + "(" + this.description + ")";
	}

	public abstract void useEffect();
}
```

1-3	問題設定	よくよく考えると，道具として使用できる武器というものが存在することに気づいた．すなわち，Arms を継承してできる武器の中に Item クラス同様に useEffect()が使用できるクラスを開発する可能性がある．これを実現するためのインタフェース Usable を定義せよ． また，Usable を定義したことで，Item1 の一部を Usable で定義できることになる．修正したヴァージョンである Item2 を定義せよ．（Item1 をそのまま修正されると，課題 1-2 が採点できないので，改めて Item2 として修正版を定義せよ．） 要するに，Item1 を Usable と Item2 に分離せよ．

1-3：解答例（Usable.java）

```
// 抽象メソッドのみなのでインタフェースとして定義
public interface Usable {
	// インタフェース内では自動的にpublic abstractとなる．
	void useEffect();
}
```

1-3：解答例（Item2.java）

```
// Item1をUsableとItem2に分割し，道具として使用可能であるか
// という点をインタフェースに分離した．従って，Item2はItem1
// の残りの部分の実装を行いつつ，道具としても使用可能なので
// implements Usableを忘れずに行う．
public abstract class Item2 implements Usable{
	protected String name = "";
	protected String description = "";

	public Item2(String name, String description){
		this.name = name;
		this.description = description;
	}

	public String toString(){
		return this.name + "(" + this.description + ")";
	}

	// ここでuseEffect()の実装は行わない．
	// useEffect()は道具の種類ごとに異なるということで，
	// Item1のときも抽象メソッドだったので，最終的なItem
	// においても抽象メソッドのママとするため何も書かない
	// が正解である．従って，本クラスも抽象クラスとなる．
}
```

<table>
<tr><td rowspan="11">1-4</td><td colspan="2">問題設定</td><td>課題 1-3 までで開発の準備が整った．後はインスタンス化できるクラスの実装を行う．次の３つのクラスを実装せよ．クラス名欄の括弧書きを参照し，適切な extends と implements を実装せよ．extends と implements には Arms，Item2，Usable のいずれかを用いることとする．</td></tr>
<tr><td rowspan="3">クラス１</td><td>クラス名</td><td>Sword　（武器：剣）</td></tr>
<tr><td>コンストラクタ</td><td>１．全てのフィールドの初期値を設定する．</td></tr>
<tr><td>メソッド</td><td>attackEffect()を実装する．
　処理　：「グサッ」とコンソール出力する．</td></tr>
<tr><td rowspan="3">クラス２</td><td>クラス名</td><td>Cane　（武器：杖，　道具としても使用可能）</td></tr>
<tr><td>コンストラクタ</td><td>１．全てのフィールドの初期値を設定する．</td></tr>
<tr><td>メソッド</td><td>attackEffect()を実装する．
　処理　：「ゴスッ」とコンソール出力する．
useEffect()を実装する．
　処理　：「○○を振りかざした」とコンソール出力する．
　　※○○には武器名が入る．</td></tr>
<tr><td rowspan="3">クラス３</td><td>クラス名</td><td>Harb　（道具：やくそう，　道具として使用可能）</td></tr>
<tr><td>コンストラクタ</td><td>１．全てのフィールドの初期値を設定する．</td></tr>
<tr><td>メソッド</td><td>useEffect()を実装する．
　処理　：「モシャモシャ」とコンソール出力する．</td></tr>
<tr><td>テスト例</td><td colspan="2">Main.java の main メソッドにて．
<pre>
public static void main(String[] args){
        Sword sword = new Sword("はがね剣", 30, 10);
        Cane cane = new Cane("輝石の杖", 10, 30);
        Harb harb = new Harb("お洒落薬草", "モリモリ栄養補給");

        // まず1-1で実装されているメソッドの動作確認を行う
        System.out.println(sword.getAp());
        System.out.println(sword.getMap());
        System.out.println(sword.toString());
        sword.attackEffect();

        // 1-2で実装されているメソッドの動作確認
        System.out.println(harb.toString());
        harb.useEffect();

        // 1-3はテストは特にない

        // 1-4で実装されているメソッドの動作確認
        sword.attackEffect();
        cane.attackEffect();
        cane.useEffect();   // Caneは道具としても使える
        harb.useEffect();
}
</pre></td></tr>
</table>

1-4：解答例（Sword.java）

```
// Swordは武器なのでArmsを継承する.
public class Sword extends Arms{
	public Sword(String name, int ap, int map) {
		// 1行目で親クラスのコンストラクタで初期化を行う.
		super(name, ap, map);
	}
	// Armsでは抽象メソッドだったが，Swordでは実装する.
	@Override	// @Override注釈は忘れずに書くこと
	public void attackEffect(){
		System.out.println("グサッ");
	}
}
```

1-4：解答例（Cane.java）

```
// Caneは武器であり，かつ道具としても使える.
// したがって，Armsを継承し，Usableを実装する.
// これによってCaneはArmsとしてもUsableとしても振る舞える.
public class Cane extends Arms implements Usable{
	public Cane(String name, int ap, int map) {
		// 1行目で親クラスのコンストラクタで初期化を行う.
		super(name, ap, map);
	}

	// Armsでは抽象メソッドだったが，Caneでは実装する.
	@Override	// @Override注釈は忘れずに書くこと
	public void attackEffect() {
		System.out.println("ゴスッ");
	}

	// Usableでは抽象メソッドだったが，Caneでは実装する.
	@Override	// @Override注釈は忘れずに書くこと
	public void useEffect() {
		System.out.println(this.name + "を振りかざした");
	}
}
```

1-4：解答例（Harb.java）

```
// Harbは道具なのでItem2を継承する.
// Item2自体がUsableを持っているので再度implementsする必要はない.
public class Harb extends Item2{
	public Harb(String name, String description) {
		// 1行目で親クラスのコンストラクタで初期化を行う.
		super(name, description);
	}
	// Item2では抽象メソッドだったが，Caneでは実装する.
	// 正確にはUsableでは抽象メソッドで，Item2で実装されていないので
	@Override	// @Override注釈は忘れずに書くこと
	public void useEffect() {
		System.out.println("モシャモシャ");
	}
}
```

課題 2

2-1

問題設定

数秒悩み，ダジャレをコールバックする Joke クラスを誰かが開発したようだ．コールバックを受け取るための JokeListener インタフェースも準備されている．

joke.play() でダジャレを依頼し，ダジャレを待っている間，プログレスバーのように進捗が表示される図 1 の機能を実装したい．使い方は次の通りである．

1. Joke をインスタンス化する．コンストラクタには JokeListener を実装したインスタンスを引数とする．
2. Joke インスタンスに対し，int play() を実行する．play() は数秒間悩んだ後にダジャレをコールバックメソッド（void jokePlayed (String joke)）に送る．
3. int play() の戻り値は何秒悩むかを示している．

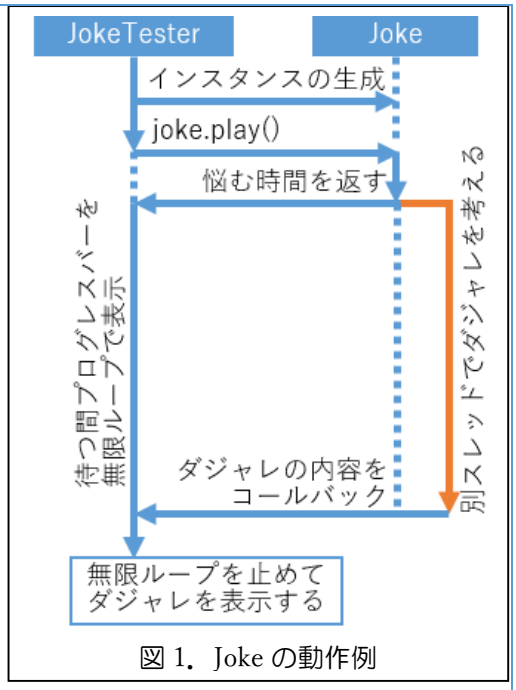

図 1．Joke の動作例

Joke を試すため以下の条件を満たすクラス JokeTester を開発せよ．Joke と JokeListener は Web から DL できる．

継承	なし	実装	JokeListener

フィールド
　joke: Joke 型のインスタンス変数
　flag: boolean 型（Joke のコールバックを待っていることを示すフラグ）

コンストラクタ
　引数なし
　処理：フィールド joke をインスタンス化する．

まずは，ここまでを実装したクラス JokeTester を開発せよ．Joke と JokeListener は Web から DL した後，eclipse のソースフォルダにドラッグアンドドロップでインポートするとよい．

<table>
<tr><td rowspan="4">2-2</td><td>メソッド</td><td>void jokeTest()：
Joke のテストを行う．
引数，戻り値：なし

処理（複雑なので手順を示す）
1. Joke を依頼するため joke.play()を実行し，戻り値（ダジャレ生成にかかる秒数）を得る（適当な変数に格納する）．
2. コールバックメソッド jokePlayed()が実行されるまでの間，0.5 秒で 1 ループする無限ループを実装する．0.5 秒待機する処理はヒントをコピペするとよい．無限ループ中には「ダジャレ待機中」とコンソール出力し，フリーズしているわけではないことを示してほしい．
3. コールバックメソッド jokePlayed()が実行されたら，無限ループが終わるように処理を修正する．boolean 型のフラグを用いるとよい．
4. 無限ループを止めると同時に，コールバックで送られてきたダジャレをコンソール出力し，処理を終了する．</td></tr>
<tr><td>ヒント</td><td>0.5 秒待つという処理は以下のように実装する．
try{
　　Thread.sleep(500)；　//500ms 待つ
}catch(Exception e){
}</td></tr>
<tr><td>テスト例</td><td>Main.java の main メソッドにて
JokeTester jt = new JokeTester()；
jt.jokeTest()；</td></tr>
<tr><td>出力例</td><td>ダジャレ待機中
ダジャレ待機中
ダジャレ待機中
ダジャレ待機中
フトンがフットンだ</td></tr>
</table>

2-3	メソッド	void jokeTestExp()： Joke のテストを行う（発展）． 引数，戻り値：なし 処理（複雑なので手順を示す） 1. Joke を依頼するため joke.play()を実行し，戻り値（ダジャレ生成にかかる秒数）を得る（適当な変数に格納する）． 2. コールバックメソッド jokePlayed()が実行されるまでの間，0.5 秒で 1 ループする無限ループを実装する．0.5 秒待機する処理はヒントをコピペするとよい．無限ループ中にはプログレスバーのような表示をコンソール出力し，フリーズしているわけではないことを示してほしい． 3. コールバックメソッド jokePlayed()が実行されたら，無限ループが終わるように処理を修正する．boolean 型のフラグを用いるとよい． 4. 無限ループを止めると同時に，コールバックで送られてきたダジャレをコンソール出力し，処理を終了する． プログレスバーのような表示とは，テスト例のように，0.5 秒毎に■の数が増えていく．したがって■と□の合計数はダジャレを待つ時間の 2 倍となる（テスト例は 4 秒待機の例）．なお括弧は半角，■と□は「しかく」を日本語変換してでてくる記号である． 【参考】長い処理がかかる Web 通信のような処理を実装する際，シングルスレッドで Web 通信を行うと，利用者はフリーズしたように感じてしまう．今回のようにマルチスレッド＋コールバックによってプログレスバーを実現してあげることでユーザビリティの向上につながる．

	ヒント	0.5 秒待つという処理は以下のように実装する．
		try{
		　Thread.sleep(500)；　//500ms 待つ
		}catch(Exception e){
		}

	テスト例	Main.java の main メソッドにて JokeTester jt = new JokeTester()； jt.jokeTestExp()；
	出力例	[■□□□□□□□] [■■□□□□□□] [■■■□□□□□] [■■■■□□□□] [■■■■■□□□] [■■■■■■□□] [■■■■■■■□] [■■■■■■■■] フトンがフットンだ

2-1, 2-2, 2-3：解答例（JokeTester.java）

```
// JokeListenerを実装と書かれているのでとりあえずimplementsする.
// JokeListenerを実装することで，JokeTesterインスタンスは必ず
// jokePlayed(String)メソッドを持っていることが保証される.
// 従って，Jokeにインスタンスを送ったときにコールバックしてもらえる.
public class JokeTester implements JokeListener{
	// フィールドはJokeインスタンスとループ用のフラグ
	private Joke joke;
	private boolean flag = true;

	public JokeTester(){
		// コンストラクタではjokeをインスタンス化する.
		// Jokeをnewする際の引数にはJokeListenerを実装した
		// インスタンスを送る必要があるので，自分自身thisを送る
		// これによって，以下に実装するjokePlayed()にコールバック
		// することができるようになる.
		this.joke = new Joke(this);
	}

	public void jokeTest(){
		// ここでflagをtrueにしておくことで，再利用可能となる.
		this.flag = true;

		// jokeにダジャレの生成を依頼する.
		// 戻り値はかかる時間だが今回は使わない.
		int wait = this.joke.play();

		// flagでループ継続を判断する
		while(this.flag){
			// 基本的にはダジャレ待機中とコンソール出力し
			System.out.println("ダジャレ待機中");
			try{
				// 500ms待機する
				Thread.sleep(500);
			}catch(Exception e){
				e.printStackTrace();
			}
		}
	}

	@Override   // @Override注釈を忘れずに
	public void jokePlayed(String joke) {
		// ループを止めるためflagを変更する
		this.flag = false;
		// ダジャレをコンソール出力する.
		System.out.println(joke);
	}
```

```
        public void jokeTestExp(){
                // ここでflagをtrueにしておくことで，再利用可能となる.
                this.flag = true;

                // jokeにダジャレの生成を依頼する.
                // 戻り値はかかる時間なので□の個数を決めるのに利用する.
                int wait = this.joke.play();

                // カウント用
                int counter = 0;
                // StringBufferは文字列を連結するために使う（いずれ紹介）.
                // インスタンスにappend("文字列")をしていくと連結されていく.
                // +で文字列を連結するよりも圧倒的に高速である.
                StringBuffer buf = new StringBuffer();

                while(this.flag){
                        buf.append("[");
                        // 500msでループするのでwait*2が□の総数
                        for(int i=0;i<wait*2;i++){
                                // 数え終えている分は黒塗り，他白塗り
                                if(i<=counter)
                                        buf.append("■");
                                else{
                                        buf.append("□");
                                }
                        }
                        buf.append("]");
                        counter++;

                        // StringBufferはtoString()で出力できる.
                        System.out.println(buf.toString());

                        // StringBufferは以下で初期化できる.
                        buf.setLength(0);

                        try{
                                Thread.sleep(500);
                        }catch(Exception e){
                                e.printStackTrace();
                        }
                }
        }
}
```