SIMON ROBERTS

クイズに挑戦

Java認定試験の作者による出題の第2回

解答の詳しい説明が付くようになった新しいクイズの形式は気に入っていただけたでしょうか。今回も、1Z0-809 Programmer II試験の練習となる問題を出題いたします。

設問1次のコードについて：

```
public enum Suit {                    // line n1
  HEART, DIAMOND, SPADE, CLUB;        // line n2
  private Color color;                // line n3
  public final Suit(Color color) { // line n4
    this.color = color;
  }
}
```

このコードをコンパイルできるようにするために必要な変更はどれですか。2つ選んでください。

a. line n4のpublicキーワードを削除する。

b. line n4のpublicキーワードとfinalキーワードの両方を削除する。

c. line n1のpublicキーワードを削除する。

d. line n2を次のように変更する。

```
HEART(Color.RED), DIAMOND(Color.RED),
    SPADE(Color.BLACK), CLUB(Color.BLACK);
```

e. line n2を次のように変更する。

```
new HEART(Color.RED), new DIAMOND(Color.RED),
    new SPADE(Color.BLACK), new CLUB(Color.BLACK);
```

設問2IOExceptionをスローする可能性があるI/O操作を実行するメソッドを作成しています。実際のI/O処理部分は省略されておりますが、この操作は次のコード部分の// ...で始まる部分で行われていると仮定します。

FileReaderのコンストラクタでは、throws FileNotFoundExceptionが宣言されています。FileNotFoundExceptionは、IOExceptionのサブクラスです。このコードでは、どちらかの例外をログに記録し、呼出し元に再スローする必要があります。

以上の要件を満たすのはどのコードですか。

a.

```
try (BufferedReader br =
  new BufferedReader(new FileReader(fName));) {
  // ...
} catch (IOException e) {
  LOG.warning(()->e.getLocalizedMessage());
  throw e;
} catch (FileNotFoundException e) {
  LOG.warning(()->e.getLocalizedMessage());
  throw e;
}
```

b.

```
try (BufferedReader br =
  new BufferedReader(new FileReader(fName));) {
  // ...
} catch (FileNotFoundException e) {
  LOG.warning(()->e.getLocalizedMessage());
  throw e;
} catch (IOException e) {
  LOG.warning(()->e.getLocalizedMessage());
  throw e;
} finally {
  br.close();
}
```

c.

```
try (BufferedReader br =
  new BufferedReader(new FileReader(fName));) {
```

```
  // ...
} catch (IOException e) {
  LOG.warning(()->e.getLocalizedMessage());
 throw e;
}
```

d.

```
try (BufferedReader br =
  new BufferedReader(new FileReader(fName));) {
  // ...
} catch (IOException | FileNotFoundException e) {
  LOG.warning(()->e.getLocalizedMessage());
  throw e;
}
```

e.

```
try (BufferedReader br =
  new BufferedReader(new FileReader(fName));) {
  // ...
} catch (IOException | FileNotFoundException e) {
  LOG.warning(()->e.getLocalizedMessage());
  throw e;
} finally {
  br.close();
}
```

設問3 アクセス可能な、果物を示すFruitクラスがあります。このクラスに次のプロトタイプを持つJavaBeansスタイルのアクセッサ・メソッドが定義されているものとします。

public String getColor()

また、sfが空でないStream<Fruit>であり、次のコード部分があるものとします。

```
System.out.println(
  // line n1
  .filter("yellow"::equalsIgnoreCase)
  .count());
```

ストリーム内の黄色い果物の合計数を出力するためには、次のうちどれをline n1に挿入すればよいですか。

a. `sf.filter(f->f.getColor()`
`.equalsIgnoreCase("yellow"))`

b. `sf.flatMap(f->f.getColor())`

c. `sf.reduce(f->f.getColor())`

d. `sf.map(Fruit::getColor)`

e. いずれでもない

設問1 正解は選択肢BとDです。列挙型の重要な目的の1つは、コンパイル時に特定のインスタンスの集合を定義し、プログラムの実行中にその集合が存在し続けるようにすることです。コンストラクタにアクセスを許可するということは、新しいインスタンスを作成できるようにするために他なりません。列挙型の新しいインスタンスを作成できてしまうと、列挙型の前提が崩れてしまうため、コンストラクタにアクセスを許可する理由はありません。したがって、そのような矛盾を防ぐ手段の1つとして、列挙型のコンストラクタはprivateでなければならないことになっています。列挙型のデフォルトのコンストラクタは常にprivateであり、明示的にアクセス可能性を定義するキーワードが指定されていないコンストラクタはすべてprivateとなります。アクセス制御キーワードを指定する場合は、必ずprivateである必要があります。

『Java言語仕様』のセクション8.9には、「列挙型はそのenum定数で定義したインスタンス以外は存在しない。列挙型を明示的にインスタンス化しようとした場合はコンパイル時エラーとなる」と記載されています。セクション8.9.2には、「列挙型の宣言において、アクセス修飾子がないコンストラクタの宣言はprivateとなる」と書かれています。さらにセクション8.9.3には、「列挙型の宣言でのコンストラクタの宣言がpublicまたはprotectedの場合、コンパイル時エラーとなる」とあります。

以上のことを踏まえると、コンストラクタからpublic修飾子を外さなければならないのは明らかです。これは必要なことである一方で、これだけでは十分ではありません。理由を考えてみます。一般的に考えるなら、コンストラクタはfinalではありません（コンストラクタは継承されないため、この修飾子を使うのは意味をなしません）。

セクション8.8.3には、「メソッドとは異なり、コンストラクタをabstract、static、final、native、strictfp、synchronizedにすることはできない」と記載されています。

そのため、コンストラクタからpublicとfinalの両方のキーワードを削除する必要があります。この要件を満たすのは選択肢Bのみであるため、この選択肢が正解に含まれることになります。また、選択肢Aが除外されることもわかります。

選択肢Cは、列挙型クラス全体からpublicキーワードを削除すると言っています。これは必要ではありません。publicな列挙型を作ることには何の問題もありません。

セクション8.1.1には、通常のクラスの修飾子にpublicを含めることができると記述されています。その後のセクション8.9には、列挙型固有の情報が記載されています。abstractとfinalは禁止されていますが、その他の修飾子は禁止されていないため、publicは許可されています。そのため、選択肢Cは誤りです。

また、クラスに明示的なコンストラクタが定義されている場合、コンパイラはデフォルトのコンストラクタを生成しません（セクション8.8.9）。そのため、元のソースのline n2はコンパイルされません。この行で使われているフォーマットには括弧と引数リストがないため、存在しないデフォルト・コンストラクタを呼び出そうとして失敗するからです。セクション8.9.1には、引数リストを記載した場合は引数が「列挙型のコンストラクタに渡される」と書かれています。また、通常のオーバーロードのルールが適用されることも書かれています。この点とこのセクションで規定されているフォーマットを踏まえると、選択肢Dの呼出しフォーマットは正しいことがわかります。newキーワードを使用している選択肢Eの形式は構文エラーです。そのため、選択肢Eは誤りで、もう1つの正解は選択肢Dです。

認定試験受験者の方へ：この設問では、コンストラクタの引数でColor.REDやColor.BLACKを参照しているにもかかわらず、Colorクラスの定義やインポートはありません。そのため、なぜこれが正しいのかと疑問に思う方もいらっしゃるかもしれません。

検討しなければならないコードの量を合理的な限度内に収めるために、オラクルは設問を解くにあたっての前提事項をいくつか定めています。その内容は、試験情報のページに掲載されていますが、特に次のような前提事項が明示されています。「package文やimport文の欠落：サンプル・コードにpackage文やimport文が含まれておらず、省略されている文について設問で明示的に言及されていない場合は、すべてのサンプル・コードが同じパッケージにあり、それが正しく動作するimport文が含まれていると仮定します」。この点を考慮すると、Colorクラスが存在しており、そこで上記の定数が定義されていると仮定するのが適切であることは明らかです。よくわからない場合は、これをオリジナルのjava.awt.Colorクラスと考えることもできますが、そのような細かい点は重要ではありません。

オラクルによる注意事項のほとんどはわかりきったことかもしれません。つまり、このような前提を持って考えない限り、多くの設問で正答がなくなってしまいます。とはいえ、注意事項が明示され、不適切な仮定を行っていないかと心配する必要がないのはよいことです。通常、「**Exam Topics**」タブ（各試験のページを半分ほど下にスクロールしたところにあります）を選択すると、この注意事項を確認できます。

設問2正解は選択肢Cです。FileNotFoundExceptionは、IOExceptionのサブクラスです。選択肢Aで行おうとしているように、クラス階層を形成する2つの例外の両方を明示的にキャッチする場合、詳細な例外を汎用的な例外よりも先に配置する必要があります。この順番のルールが存在するのは、これが実行される際に、適用可能な最初のcatchブロックにジャンプする（『Java言語仕様』のセクション14.20.1）ため、汎用的な例外が最初にあった場合、詳細な例外をキャッチする部分が実行されなくなってしまうためです。選択肢Aでは、このルールが守られていないため、コンパイルに失敗します。つまり、選択肢Aは誤りです。

ちなみに、以前のバージョンのJavaでは実際にこのようなことが発生し、非常に見つけづらいバグになることがありました。現在の仕様では、このような状態はセクション14.21の不達コードとして扱われ、コンパイル・エラーとなるように定められています。

選択肢Bでは、catchブロックの順番は正しくなっていますが、明示的

なfinallyブロックも追加されています。finallyブロックがあること自体は問題ありませんが、リソース変数br（BufferedReader）のスコープはtry文に続く括弧でくくられたブロック内に限定されています。したがって、finallyブロックで明示的にbrをクローズしようとした場合、コンパイルに失敗します。もちろん、try-with-resource構造の本質はリソースを暗黙的にクローズする点にあるため、この選択肢のようにbrを明示的にクローズする操作は的外れでもあります。

選択肢Cは正しく動作し、この選択肢が正解です。FileNotFoundExceptionはIOExceptionのサブクラスであるため、1つのcatchブロックで両方ともキャッチでき、仕様どおりに動作します。ちなみに、この2つの例外に対して別々の処理を行いたい場合や、親子関係にない例外に対して同じ処理を行いたい場合、この選択肢の方法は適切ではありません。しかし、これはこの設問に関わることではありません。

新しい（Java 7）マルチcatch構文を使った選択肢Dも一見よいアイデアのように思えるかもしれません。しかし、2つの例外には親子関係があるため、このコードはコンパイル・エラーになります。セクション14.20には、「別々の型DiとDj（i ≠ j）があり、DiがDjのサブタイプであるとき、この2つを組み合わせるとコンパイル時エラーとなる」と書かれています。少し回りくどい表現ですが、この意味を端的に述べると、マルチcatch構文はIOExceptionとFileNotFoundExceptionのような継承関係がある例外型には使用できないということです。

選択肢Eは、選択肢DおよびBと同じ理由で失敗します。

2つ補足します。まず、前述の2つの例外はかなり一般的であり、多くの試験目的でこの種の知識が前提とされていることを考えると、試験でこれらの例外の継承関係が明示的に伝えられないことも十分考えられると思います。

また、この例にはかなり大量のコードが記載されていることにお気づきでしょう。ここまで長い設問は一般的ではありませんが、ときどきこのような設問に遭遇することもあります。頭のさえた状態を保ち、特定のポイントについて系統的に調べる能力を鍛えておくとよいでしょう。いずれにせよ、本番環境で動作しているコードは常にきれいなコードを作成するための表記規則に従って書かれ、メンテナンスされているとは限りません。

設問3正解は選択肢Dです。このコード部分は黄色い果物の数を出力するものです。countメソッドに渡すストリーム（つまりfilterメソッドの出力）には、元のストリームの中にある黄色い果物1つにつき1つの要素が含まれている必要があります。もちろん、その要素が何であるかは特段重要ではありません。

すでにコードに書かれているfilterメソッドは、"yellow"というテキストと（大文字、小文字を無視して）一致する文字列の要素だけを通過させます。つまり、filterメソッドに入るストリームには、元のFruitオブジェクト自体ではなく、Fruitオブジェクトの色名のみが含まれている必要があります。

入力項目と出力項目の1対1の対応付けを維持したままストリームの内容（値またはデータ型のいずれか）を変換できるStreamインタフェースのメソッドは、mapです。選択肢Dはこれを行っており、形式も正しいものです。具体的には、getColorメソッドへのメソッド参照を使って入力ストリームからFruitオブジェクトを取り出し、色名を抽出しています。今説明したように、要求されたとおりに動作するために必要なのは、このコードです。そのため、選択肢Dは正解です。

選択肢Aは、既存のfilter文を置き換えるのであれば動作します。この文を実行すると、黄色いFruitオブジェクトが次のストリームに渡されます。これを直接数えるのであれば望む結果が得られます。しかし、コードからわかるように、下流のストリーム・フィルタは、色名ではなくFruitオブジェクト全体を受け取ることになるため、コンパイルに失敗します。

選択肢Bはコンパイルされません。flatMapに引数として渡す動作の戻り値の型は何らかのストリームでなければならないためです。なお、この選択肢がsf.flatMap(f->Stream.of(f.getColor()))であった場合、非効率的で回りくどい方法ではあるものの、正常に動作します。

選択肢Cもコンパイルに失敗します。さらに、この選択肢は意味をなしていません。Streamのreduceメソッドは、ストリームを通ってきたすべての値を収集する終端操作で、1つの出力値を生成します。reduceメソッドの動作を指定する引数は、この単一の出力値を生成する計算を実行するものです。とにかく、ここで必要とされるものではありません。なお、単一引数のreduceメソッドの動作引数はBinaryOperatorです。ラムダ式f->f.getColor()は実質的にFunction<Fruit, String>であるため、ここで使用することはできません。**</article>**

Simon Roberts：Sun Microsystemsがイギリスで初めてJavaの研修を行う少し前にSunに入社し、Sun認定JavaプログラマーとSun認定Java開発者の試験の作成に携わる。何冊かのJava認定ガイドを執筆し、現在はフリーランスでシリコン・バレーや世界中のたくさんの大企業の研修を行う。OracleのJava認定プロジェクトにも継続的に関わっている。

learn more

- 列挙型を活用する
- オラクルによるJavaのFileReaderに関するチュートリアル
- オラクルによるJavaのストリームと集計操作に関するチュートリアル

注目のJDK強化提案

JEP 283およびJEP 263：LinuxでのGTK+ 3への移行

この件はJavaの内部が中心であるため、LinuxでJavaがどのようにユーザー・インタフェース要素を扱っているかを見ていきます。現在、Linux版のJREの一部は、Cで記述されているポータブルUIライブラリGIMP Toolkit（GTK+）を使用しています。しかし、JavaはそのライブラリのGTK+ 2を使用しています。JEP 283では、GTK+ 3への移行が提案されています。このメリットは、GTK+ 3が開発ブランチであることです（GTK+ 2はすでにアクティブではありません）。この変更が行われるもう1つの理由は、今後生じるおそれがある問題、すなわち今後のLinuxディストリビューションでGTK+ 2がサポートされなくなる可能性があるという懸念の存在です。現在、ほとんどのバージョンのLinuxにはGTK+ 2と3の両方が搭載されています。しかし、バージョン3の誕生は2011年であるため、さまざまなディストリビューションでいつまでGTK+ 2のバンドルが継続されるのかは不透明です。そのため、この強化提案によって、AWT/SwingやJavaFX（場合によってはSWTも）でのバージョン3のサポートをどのように始めていけるかを検討します。JDK強化提案（JEP）ドキュメントはとりわけ興味深い内容になっています。というのも、ここには単一プラットフォームの単一コンポーネントでかなり小さな変更を行うためにどの程度の作業が必要になるかについての概要が記載されているからです。利点や欠点がうまくまとめられており、さまざまな移行パスについても書かれています。

初期の提案であるJEP 263では、高解像度ディスプレイのサポートが向上したライブラリとしてのGTK+ 3のメリットが指摘されています。

このトピックに興味があり、専門知識をお持ちの方は、ぜひExpert Committeesにご参加ください。