本書の構成

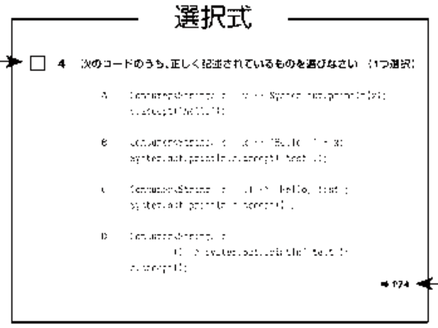
本書は、カテゴリ別に分類された、問題と解答で構成されています。試験の出題範囲に沿った問題に解答したのち、解説を読んで学習すると、合格レベルの実力が身に付きます。また、実際の試験に近い形式になっていますので、より実戦的に学習できます。

問題

試験の出題形式は選択式です。正答を1つだけ選ぶものと複数選ぶものがあります。

チェックボックス

確実に理解している問題のチェックボックスを塗りつぶしながら問題を解き進めると、2回目からは不確かな問題だけを効率的に解くことができます。すべてのチェックボックスが塗りつぶされれば合格は目前です。



解答ページ

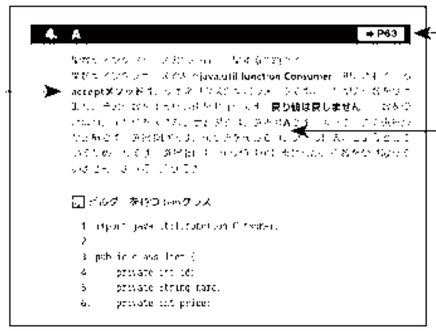
問題の右下に、解答ページが表示されています。ランダムに問題を解くときも、解答ページを探すのに手間取ることがありません。

解答

解答には、問題の正解および不正解の理由だけでなく、用語や重要事項などが詳しく解説されています。

重要項目

重要項目は、わかりやすく太字で示しています。



問題ページ

問題ページが表示されています。

正解

正解の選択肢は「選択肢A」のように太字で示しています。

問題のソースコードでは、mainメソッド、例外処理、import文、package文などを省略し、プログラムを動作させるのに不完全な形で記述されているものもあります。このようなソースコードは、実際の試験でも同様に出題されることが考えられます。本書の問題に解答する際には、省略されている記述にかかわらず、問題の趣旨を的確にとらえるようにしてください。

本書で使用するマーク

試験対策	試験対策のために理解しておかなければいけないことや、覚えておかなければいけない重要事項を示しています。	参考	試験対策とは直接関係はありませんが、知っておくと有益な情報を示しています。

目次

はじめに …… 3
Java SE 11認定資格について …… 4
OCJP Gold SE 11について …… 4
受験申し込み方法 …… 5
出題傾向と受験対策 …… 5
本書の活用方法 …… 7
本書の構成 …… 8
本書で使用するマーク …… 8

第1章 クラスとインタフェース

問題 …… 12
解答 …… 25

第2章 関数型インタフェースとラムダ式

問題 …… 62
解答 …… 67

第3章 並列処理

問題 …… 96
解答 …… 102

第4章 ストリームAPI

問題 …… 154
解答 …… 170

第5章 入出力

問題 …… 236
解答 …… 242

第6章 JDBCによるデータベース連携

問題 …… 296
解答 …… 301

第7章 汎用とコレクション

問題 …… 328
解答 …… 335

第8章 アノテーション

問題 …… 374
解答 …… 376

第9章 例外とアサーション

問題 …… 400
解答 …… 405

第10章 ローカライズ

問題 …… 426
解答 …… 432

第11章 モジュール・システム

問題 …… 460
解答 …… 463

第12章 Java SEアプリケーションにおけるセキュアコーディング

問題 …… 494
解答 …… 498

第13章 総仕上げ問題

問題 …… 530
解答 …… 582

索引 …… 616

1

クラスとインタフェース

- ネストしたクラス
- インナークラス
- staticインナークラス
- ローカルクラス
- 匿名クラス
- インタフェースとデフォルトメソッド
- Enum（列挙型）

1. 次の説明のうち、正しいものを選びなさい。(2つ選択)

A. インナークラスはprivateで修飾できない
B. staticなインナークラスはpublicで修飾できない
C. ローカルクラスはpublicで修飾できない
D. 匿名クラスはfinalで修飾できない

➡ P25

2. 次のプログラムの「// insert code here」に記述するコードとして、正しいものを選びなさい。(1つ選択)

```
1. public class Outer {
2.     class Inner {
3.         public void test() {
4.             System.out.println("test");
5.         }
6.     }
7.     public static void main(String[] args) {
8.         // insert code here
9.     }
10. }
```

A. `new Inner();`
B. `new Outer.Inner();`
C. `new Outer().new Inner();`
D. `new Outer:Inner();`

➡ P26

3. 次のクラスをインスタンス化するためのコードとして、正しいものを選びなさい。(1つ選択)

```
1. public class Outer {
2.     public static class Inner {
3.         // do something
4.     }
5. }
```

A. `Inner inner = new Inner();`
B. `Inner inner = new Outer().new Inner();`
C. `Inner inner = Outer.new Inner();`
D. `Inner inner = new Outer().Inner();`

➡ P30

4. 次のプログラムをコンパイル、実行したときの結果として、正しいものを選びなさい。(1つ選択)

```
1. public class Outer {
2.     private int num = 2;
3.     public void hoge() {
4.         Inner test = new Inner();
5.         test.data = 100;
6.         test.execute();
7.     }
8.     private static class Inner {
9.         private int data;
10.        void execute() {
11.            System.out.println(num * data);
12.        }
13.    }
14. }
```

A. 5行目でコンパイルエラーが発生する
B. 11行目でコンパイルエラーが発生する
C. 「200」と表示される
D. 実行時に例外がスローされる

➡ P36

□ **5.** 次のプログラムをコンパイル、実行したときの結果として、正しいものを選びなさい。(1つ選択)

```
1. public class Outer {
2.     void test() {
3.         Inner.message = "Hello, Java";
4.     }
5.     class Inner {
6.         private static String message;
7.         public void test() {
8.             System.out.println(message);
9.         }
10.    }
11.    public static void main(String[] args) {
12.        Outer outer = new Outer();
13.        outer.test();
14.        outer.new Inner().test();
15.    }
16. }
```

A. 「null」と表示される
B. 「Hello, Java」と表示される
C. コンパイルエラーが発生する
D. 実行時に例外がスローされる

➡ P38

□ **6.** 次のプログラムをコンパイル、実行したときの結果として、正しいものを選びなさい。(1つ選択)

```
1. public interface Test {
2.     void execute();
3. }
```

```
1. public class Sample {
2.     private String message;
3.     public Test test(String value) {
4.         class A implements Test {
5.             @Override
6.             public void execute() {
7.                 System.out.println(message + value);
8.             }
9.         }
10.        value = "LocalClass";
11.        message = "Hello, ";
12.        return new A();
13.    }
14. }
```

```
1. public class Main {
2.     public static void main(String[] args) {
3.         new Sample()
4.             .test("Java")
5.             .execute();
6.     }
7. }
```

A. 「Hello, LocalClass」と表示される
B. 「Hello, Java」と表示される
C. コンパイルエラーが発生する
D. 実行時に例外がスローされる

➡ P39

□ **7.** 次のプログラムをコンパイル、実行したときの結果として、正しいものを選びなさい。（1つ選択）

```
1. public class Sample {
2.     private int num;
3.     public Sample(int num) {
4.         this.num = num;
5.     }
6.     public int getNum() {
7.         return num;
8.     }
9.     public void setNum(int num) {
10.        this.num = num;
11.    }
12. }
```

```
1. public class Main {
2.     public static void main(String[] args) {
3.         var sample = new Sample(10) {
4.             void modify(int num) {
5.                 setNum(num);
6.             }
7.         };
8.         sample.modify(100);
9.         System.out.println(sample.getNum());
10.    }
11. }
```

A. 「10」と表示される
B. 「100」と表示される
C. コンパイルエラーが発生する
D. 実行時に例外がスローされる

➡P43

□ **8.** 次のプログラムをコンパイル、実行したときの結果として、正しいものを選びなさい。（1つ選択）

```
1. public class Sample {
2.     public Sample() {
3.         System.out.println("A");
4.     }
5. }
```

```
1. public class Main {
2.     public static void main(String[] args) {
3.         Sample sample = new Sample() {
4.             public Sample() {
5.                 System.out.println("B");
6.             }
7.         };
8.     }
9. }
```

A. 「A」と表示される
B. 「B」と表示される
C. コンパイルエラーが発生する
D. 実行時に例外がスローされる

➡ P45

□ **9.** 次のプログラムの説明として、正しいものを選びなさい。（3つ選択）

```
1. public interface A {
2.     static void test() {
3.         System.out.println("A");
4.     }
5. }
```

```
1. public interface B extends A {}
```

```
1. public class C implements A {}
```

```
1. public class D implements A {
2.     public static void main(String[] args) {
3.         A.test();
4.         B.test();
5.         C.test();
6.         test();
7.     }
8. }
```

A. Dクラスの3行目でコンパイルエラーが発生する
B. Dクラスの4行目でコンパイルエラーが発生する
C. Dクラスの5行目でコンパイルエラーが発生する
D. Dクラスの6行目でコンパイルエラーが発生する

➡ P47

□ **10.** 次のプログラムをコンパイル、実行したときの結果として、正しいものを選びなさい。（1つ選択）

```
1. public interface Sample {
2.     static void test() {
3.         System.out.println("A");
4.     }
5. }
```

```
1. public class SampleImpl implements Sample {
2.     public static void test() {
3.         System.out.println("B");
4.     }
5.     public static void main(String[] args) {
6.         Sample.test();
7.     }
8. }
```

A. 「A」と表示される
B. 「B」と表示される
C. コンパイルエラーが発生する
D. 実行時に例外がスローされる

➡ P48

11. 次のプログラムの「// insert code here」に記述するコードとして、正しいものを選びなさい。(1つ選択)

```
1. public interface A {
2.     default void test() {
3.         System.out.println("A");
4.     }
5. }
```

```
1. public interface B {
2.     default void test() {
3.         System.out.println("B");
4.     }
5. }
```

```
1. public interface C extends B {}
```

```
1. public interface D extends C {}
```

```
1. public class Sample implements A, D {
2.     @Override
3.     public void test() {
4.         // insert code here
5.     }
6. }
```

A. A.super.test();
B. B.super.test();
C. C.super.test();
D. 上記の選択肢はいずれもコンパイルエラーとなる

➡ P49

12. 次のプログラムをコンパイル、実行したときの結果として、正しいものを選びなさい。(1つ選択)

```
1. public interface A {
2.     default void test() {
3.         System.out.println("A");
4.     }
5. }
```

```
1. public class B {
2.     public void test() {
3.         System.out.println("B");
4.     }
5. }
```

```
1. public class C extends B implements A {}
```

```
1. public class Main {
2.     public static void main(String[] args) {
3.         A a = new C();
4.         a.test();
5.     }
6. }
```

A. 「A」と表示される
B. 「B」と表示される
C. コンパイルエラーが発生する
D. 実行時に例外がスローされる

➡ P50

13. 次のプログラムをコンパイル、実行したときの結果として、正しいものを選びなさい。(1つ選択)

```
1. public interface A {
2.     public default void test() {
3.         System.out.println("A");
4.     }
5. }
```

```
1. public abstract class B {
2.     protected void test() {
3.         System.out.println("B");
4.     }
5. }
```

```
1. public class Sample extends B implements A {
2.     public static void main(String[] args) {
3.         new Sample().test();
4.     }
5. }
```

A. 「A」と表示される
B. 「B」と表示される
C. コンパイルエラーが発生する
D. 実行時に例外がスローされる

➡ P51

14. 次のプログラムの「// insert code here」に記述してコンパイルできるコードとして、正しいものを選びなさい。(1つ選択)

```
1. public interface Sample {
2.     // insert code here
3. }
```

A. `private void a();`
B.
```
private void b() {
    // do something
}
```
C.
```
private default void c() {
    // do something
}
```
D. 上記の選択肢はいずれもコンパイルエラーとなる

➡ P51

15. 次のプログラムをコンパイルし、実行結果としてコンソールに「A」と表示したい。「// insert code here」に記述するコードとして、正しいものを選びなさい。(1つ選択)

```
1. public enum Test {
2.     A, B, C
3. }
```

```
1. public class Main {
2.     public static void main(String[] args) {
3.         // insert code here
4.     }
5. }
```

A. `System.out.println(Test.values()[0]);`
B. `System.out.println(Test.values(0));`
C. `System.out.println(Test.values()[1]);`
D. `System.out.println(Test.values(1));`
E. `System.out.println(Test.valueOf(1));`
F. `System.out.println(Test.valueOf()[1]);`

➡ P52

□ **16.** 次のプログラムをコンパイル、実行した結果として、正しいものを選びなさい。(1つ選択)

```
1. public enum Sample {
2.     A("hello"), B("hello"), C("hello");
3.     private final String value;
4.     private Sample(String value) {
5.         System.out.println(value);
6.         this.value = value;
7.     }
8.     @Override
9.     public String toString() {
10.        return this.value;
11.    }
12. }
```

```
1. public class Main {
2.     public static void main(String[] args) {
3.         System.out.println(Sample.A);
4.     }
5. }
```

A. 「A」と表示される
B. 「hello」が1回表示される
C. 「hello」が3回表示される
D. 「hello」が4回表示される

➡ P57

第1章　クラスとインタフェース
解　答

1. C、D

➡ P12

ネストしたクラスに関する問題です。
Javaでは次のコード例のように、クラス内にクラスを定義することができます。

```
1. public class Outer {
2.     class Inner {
3.         // do something
4.     }
5. }
```

クラス内に定義したクラスのことを総称して「**ネストしたクラス**」と呼びますが、正確にはネストしたクラスにはいくつかの種類があり、それぞれ次のように名称が異なります。

・インナークラス
・staticインナークラス
・ローカルクラス
・匿名クラス

前述のコード例のようにクラスの中に定義したクラスを「**インナークラス**」と呼び、staticで修飾したインナークラスを「**staticインナークラス**」と呼びます。**ローカルクラス**と**匿名クラス**は、どちらもメソッド内で定義したクラスのことです。
なお、ネストしたクラスを囲む外側のクラスのことを「囲んでいるクラス」という意味から「**エンクロージングクラス**」と呼んだり、「**トップレベルクラス**」と呼んだりします。前述のコード例であれば、Outerクラスがエンクロージングクラスです。

インナークラスやstaticインナークラスは、エンクロージングクラスのメンバとして定義します。これらのクラスは、publicからprivateまでの4つのアクセス修飾子のいずれかで修飾することにより、ほかのクラスからのアクセスを制御できます。よって、修飾できないとした選択肢AとBは誤りです。なお、これらのクラスは、同じエンクロージングクラス内にサブクラスを作ること

もできるので、abstractやfinalで修飾することもできます。
一方、ローカルクラスや匿名クラスは、メソッド内で定義する一時的なクラスであって、エンクロージングクラスのメンバではありません。そのため、ほかのクラスからアクセスすることはできず、アクセス修飾子で修飾することもできません。ローカルクラスはfinalもしくはabstractでのみ修飾できます。匿名クラスの場合は、修飾子で修飾することはできません。

4つのネストしたクラスと修飾子の関係をまとめると次の表のようになります。

【ネストしたクラスと修飾子の関係】

	public	protected	private	abstract	final	static
インナークラス	○	○	○	○	○	×
staticインナークラス	○	○	○	○	○	○
ローカルクラス	×	×	×	○	○	×
匿名クラス	×	×	×	×	×	×

以上のことから、選択肢**C**と**D**が正解となります。

2. C ➡ P12

インナークラスのインスタンス化に関する問題です。
ネストしたクラスが用意されている理由の1つとして、情報隠蔽の強化があります。**情報隠蔽**とは、そのソフトウェアがどのような構造で成り立っているのか、またそのプログラムモジュールがどのような機能を持っているのかを隠蔽し、公開する範囲を最小化することです。公開する情報を最小化することで、想定していなかったような使われ方をすることがなくなります。

情報隠蔽の実装方法として最も一般的なのは、クラスからインタフェースを抽出することです。クラスが持っている機能（メソッド）から、公開したいものだけを選び出してインタフェースとして定義してしまうのです。
たとえば、次のコード例のSampleImplクラスはtestとexecuteという2つのメソッドを持っています。

例 SampleImplクラス

```
1. package com.sample;
2.
3. public class SampleImpl {
4.     public void test() {
5.         System.out.println("test");
6.     }
7.     public void execute() {
8.         System.out.println("execute");
9.     }
10. }
```

これらのメソッドのうち、executeメソッドは公開したいものの、testメソッドは公開したくない何らかの理由があったとします。このときに、次のように公開したいメソッドだけを持つインタフェースを定義します。

例 Sampleインタフェース

```
1. package com.sample;
2.
3. public interface Sample {
4.     void execute();
5. }
```

その上で、先ほどのSampleImplクラスはこのSampleインタフェースを実現するように実装します。クラスのアクセス修飾子をデフォルトに変更することでパッケージ外のクラスからはアクセスできなくなります。

例 SampleImplクラス（変更後）

```
1. package com.sample;
2.
3. class SampleImpl implements Sample {
4.     public void test() {
5.         System.out.println("test");
6.     }
7.     @Override
8.     public void execute() {
9.         System.out.println("execute");
10.    }
11. }
```

さらに、実現クラスのインスタンスを生成するためのSampleFactoryクラスを用意しましょう。このクラスのcreateメソッドは、SampleImplクラスのイン

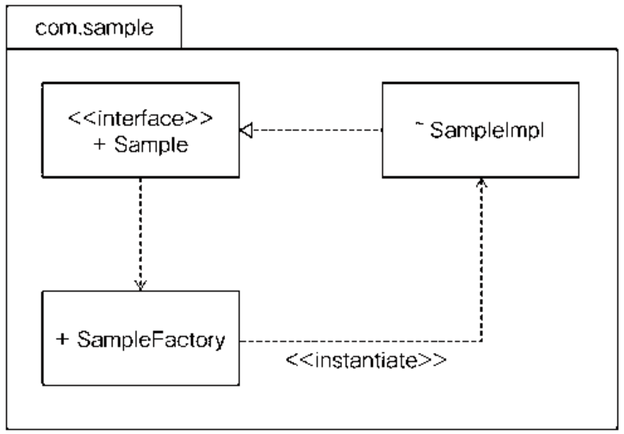
スタンスを生成し、Sampleインタフェース型で戻します。

例 SampleFactoryクラス

```
1. package com.sample;
2.
3. public class SampleFactory {
4.     public static Sample create() {
5.         return new SampleImpl();
6.     }
7. }
```

これで実際に動作するSampleImplクラスを隠蔽しつつ、公開可能な最低限の機能だけを提供するSampleを利用することができます。このような関係性をクラス図で表すと、次のようになります。

【Sampleインタフェース、SampleImplクラス、SampleFactoryクラスの関係】



SampleImplクラスは外部のパッケージからはアクセスできませんが、同じパッケージ内からであればアクセスできてしまいます。長期にわたってこのソフトウェアが使われる間に仕様変更が加わり、その都度、クラスが増えていく可能性はあります。このとき、設計当初は想定していなかったようなクラスからもSampleImplクラスが使われてしまうかもしれません。

このような想定外の利用を防ぐために、**インナークラス**を使います。具体的には、次のようにSampleFactoryクラス内に、privateなSampleImplクラスの定義を追加します。

例 インナークラスとして定義したSampleImplクラス

```
1. package com.sample;
2.
3. public class SampleFactory {
4.     public static Sample create() {
5.         return new SampleFactory().new SampleImpl();
6.     }
7.     private class SampleImpl implements Sample {
8.         public void test() {
9.             System.out.println("test");
10.        }
11.        @Override
12.        public void execute() {
13.            System.out.println("execute");
14.        }
15.    }
16. }
```

Sampleインタフェースを実現したSampleImplクラスのアクセス修飾子はprivateであるため、同じパッケージに属していても、SampleFactoryクラス以外のクラスからはアクセスすることができません。

なお、インナークラスは、そのクラスを定義したエンクロージングクラスのフィールドやメソッドと同じように扱われます。つまり、エンクロージングクラスのフィールドやメソッドが、インスタンスがなければ使えないのと同様に、インナークラスのインスタンスもエンクロージングクラスのインスタンスがなければ使えません。

そのため、SampleFactoryクラスのcreateメソッドでは、まずSampleFactoryクラスのインスタンスを生成してから、その参照を使ってインナークラスのインスタンスを生成します。前述のcreateメソッドでは、「new SampleFactory().new SampleImpl()」と1行で記述していますが、分解すると次のようなコードになります。

```
1. public static Sample create() {
2.     SampleFactory factory = new SampleFactory();
3.     Sample sample = factory.new SampleImpl();
4.     return sample;
5. }
```

このようにインナークラスのインスタンスを生成するには、**先にエンクロージングクラスのインスタンスを生成**し、その**インスタンスへの参照とnew演算子**を使います。次のようにインナークラスのインスタンスだけをインスタンス化することはできないので、注意してください。

```
1. public static Sample create() {
2.     Sample sample = new SampleImpl();
3.     return sample;
4. }
```

以上のことから、選択肢**C**が正解です。
選択肢Aは、エンクロージングクラスOuterのインスタンスを生成せずに、インナークラスInnerのインスタンスだけを生成しようとしています。選択肢Bは、エンクロージングクラスのインスタンスへの参照を使っていません。選択肢Dのようにコロン「:」を使ってインナークラスのインスタンスを生成することはできません。よって、選択肢A、B、Dは誤りです。

試験対策

インナークラスのインスタンスだけをインスタンス化することはできません。インナークラスのインスタンスを生成するには、エンクロージングクラスのインスタンスを先に生成しておく必要があります。「new Outer().new Inner()」のように記述することも覚えておきましょう。

3. A ➡ P13

staticインナークラスのインスタンス化に関する問題です。

前問で、ネストしたクラスが用意されている理由として、情報隠蔽の強化があることを解説しました。ここではもう1つの理由である、カプセル化を強化する点について解説していきます。Silverからのおさらいになりますが、カプセル化とは「関係するものを1つにまとめたモジュール（部品）を定義する」ことです。
関係性の深い2つのクラスがあったとき、片方をもう片方のネストしたクラスにしてまとめることで、関係するクラスをくまなく探す必要がなくなります。次のコード例で確認しましょう。

このコード例では、name（名前）とprice（値段）という2つのフィールドを持ったItemクラスを定義しています。このクラスは、コレクション内での並べ替えを想定して、**java.lang.Comparableインタフェース**を実現しています。

例 Itemクラス

```
1. package comp;
2.
3. import java.util.Comparator;
4.
5. public class Item implements Comparable<Item> {
6.     private String name;
7.     private int price;
8.     public Item(String name, int price) {
9.         this.name = name;
10.        this.price = price;
11.    }
12.    public String getName() {
13.        return name;
14.    }
15.    public int getPrice() {
16.        return price;
17.    }
18.    @Override
19.    public int compareTo(Item target) {
20.        return name.compareTo(target.name);
21.    }
22. }
```

このインタフェースは、どのような並べ替えの基準に従うかを**compareToメソッド**に記述するようになっており、Itemクラスの実装では名前を使って並べ替えをすることがわかります。

実際に並べ替えを行うコードを見てみましょう。このコード例では、**Collectionsクラス**の**sortメソッド**を使って並べ替えをしています。実行結果を確認すると、名前順に並べ替えられていることがわかります。

例 並べ替えを行うMainクラス

```
1. package comp;
2.
3. import java.util.Arrays;
4. import java.util.Collections;
5. import java.util.List;
6.
```

```
 7. public class Main {
 8.     public static void main(String[] args) {
 9.         List<Item> items = Arrays.asList(
10.             new Item[] {
11.                 new Item("orange", 100),
12.                 new Item("banana", 80),
13.                 new Item("apple", 120)
14.             }
15.         );
16.         Collections.sort(items);
17.         for (Item item : items) {
18.             System.out.println(item.getName());
19.         }
20.     }
21. }
```

例 実行結果

```
> java Main
apple
banana
orange
```

このように並べ替えの基準が1つだけということもあれば、場合によっては並べ替えの基準を変えたいこともあります。たとえば、名前ではなく、値段を基準にして並べ替えたいような場合です。
こうした要件は、前述のコード例のようなComparableインタフェースを実現する方法では満たすことができません。そこで、並べ替えの基準をクラス内に持つのではなく、別のクラスとして定義します。並べ替えの基準を複数用意したり、仕様によって並べ替えの基準を変更したりするときには、次のPriceComparatorクラスのように**java.util.Comparatorインタフェース**を実現したクラスを定義するのです。

例 値段で並べ替えるPriceComparatorクラス

```
 1. package comp;
 2.
 3. import java.util.Comparator;
 4.
 5. public class PriceComparator implements Comparator<Item> {
 6.     @Override
 7.     public int compare(Item a, Item b) {
 8.         if (a.getPrice() < b.getPrice()) {
 9.             return -1;
10.         }
11.         if (b.getPrice() < a.getPrice()) {
12.             return 1;
13.         }
14.         return 0;
15.     }
16. }
```

このクラスでは値段を並べ替えの基準とし、比較対象となる2つのItemのインスタンスのうち、第1引数のItemの値段が安ければ前（負の値を戻す）、高ければ後ろ（正の値を戻す）、同じであればそのまま（0を戻す）という並べ替えの処理が実装されています。
このように並べ替え対象となるItemクラスから基準を分離することで、並べ替えの基準を仕様に応じていくつでも作ることができます。次のコード例（Mainクラス抜粋）のように、PriceComparatorクラスをCollectionsクラスのsortメソッドの第2引数として渡して実行すると、実行結果のように値段順に並べ替えられていることがわかります。

```
16. Collections.sort(items, new PriceComparator());
17. for (Item item : items) {
18.     System.out.println(item.getName());
19. }
```

例 実行結果

```
> java Main
banana
orange
apple
```

このPriceComparatorクラスは、Itemクラスを並べ替えるためだけに使うクラスです。ほかの用途に使えないのであれば、わざわざ別のクラスとして分離しておく必要はありません。そこで、次のようにPriceComparatorクラスをItemクラスのインナークラスにしてまとめてしまうのです。

※次ページに続く

例 PriceComparatorクラスをインナークラスに取り込んだItemクラス

```
 1. package comp;
 2.
 3. import java.util.Comparator;
 4.
 5. public class Item implements Comparable<Item> {
 6.     private String name;
 7.     private int price;
 8.     public Item(String name, int price) {
 9.         this.name = name;
10.         this.price = price;
11.     }
12.     public String getName() {
13.         return name;
14.     }
15.     public int getPrice() {
16.         return price;
17.     }
18.     @Override
19.     public int compareTo(Item target) {
20.         return name.compareTo(target.name);
21.     }
22.     public class PriceComparator implements Comparator<Item> {
23.         @Override
24.         public int compare(Item a, Item b) {
25.             if (a.price < b.price) {
26.                 return -1;
27.             }
28.             if (b.price < a.price) {
29.                 return 1;
30.             }
31.             return 0;
32.         }
33.     }
34. }
```

こうすることで、関係する2つのクラスを1つにまとめ、カプセル化を強化することができました。しかし、ここで問題が発生します。前問で解説したように、インナークラスのインスタンスを生成するには、先にエンクロージングクラスのインスタンスを生成し、そのインスタンスへの参照とnew演算子を使います。PriceComparatorクラスはItemクラスのインナークラスであるため、先にItemクラスのインスタンス化をしなければいけません。

しかし、2つの引数を受け取るItemクラスのコンストラクタに何か値を渡す必要がありますが、ここで「どのような値を渡すのか」という問題が生じます。必要なのはPriceComparatorクラスのインスタンスであって、Itemクラスのインスタンスはインナークラスのインスタンスを作るためだけに必要なのです。もちろん、次のコード例（Mainクラス抜粋）のように意味のない値（nullと0）をコンストラクタの引数に渡してインスタンスを生成することも可能ですが、こうした意味のないインスタンスを生成するコードは推奨されません。

```
16. Collections.sort(items, new Item(null, 0).new PriceComparator());
17. for (Item item : items) {
18.     System.out.println(item.getName());
19. }
```

このようにインナークラスのインスタンス化には、エンクロージングクラスのインスタンスが必要になるという制約があるため、意味のないコードを記述してまで、インスタンスを生成せざるを得なくなるわけです。

そこで、次のように**インナークラスをstaticで修飾**し、**staticインナークラス**として定義します。この「staticインナークラス」の特徴は、そのインスタンス化にエンクロージングクラスのインスタンスを必要としないことです。

例 PriceComparatorをstaticインナークラスとして定義する

```
 1. package comp;
 2.
 3. import java.util.Comparator;
 4.
 5. public class Item implements Comparable<Item> {
 6.     … 中略 …
 7.
22.     public static class PriceComparator implements Comparator<Item> {
23.         @Override
24.         public int compare(Item a, Item b) {
25.             if (a.price < b.price) {
26.                 return -1;
27.             }
28.             if (b.price < a.price) {
29.                 return 1;
30.             }
```

```
31.             return 0;
32.         }
33.     }
34. }
```

これで次のコード例（Mainクラス抜粋）のように、エンクロージングクラスのインスタンス化を行うことなく、インナークラスのインスタンス化ができるようになります。このように、staticインナークラスを使うことでエンクロージングクラスのインスタンス化が不要になるため、コードの可読性を向上させることができるのです。

```
16. Collections.sort(items, new PriceComparator());
17. for (Item item : items) {
18.     System.out.println(item.getName());
19. }
```

なお、staticインナークラスを使用するには、使用する側のMainクラスで次のようにimport文を宣言しておきましょう。これでクラス名を記述するだけで使用できるようになります。

```
import comp.Item.PriceComparator;
```

設問のコードのInnerクラスはstaticインナークラスであるため、選択肢Aのようにエンクロージングクラスのインスタンスなしでインスタンス化することができます。以上のことから、選択肢**A**が正解です。

試験対策

staticインナークラスは、エンクロージングクラスのインスタンスがなくてもインスタンス化できます。同じインナークラスでも、エンクロージングクラスのインスタンスを必要とするインナークラスの場合とはインスタンス化の記述の仕方が違うことに注意しましょう。

4. B

➡ P13

staticインナークラスに関する問題です。
ネストしたクラスからエンクロージングクラスのフィールドやメソッドにアクセスすることは可能です。次のコード例では、インナークラスInnerからエンクロージングクラスOuterのフィールドmessageにアクセスし、その値をコンソールに表示しています。

例 インナークラスからエンクロージングクラスのフィールドへのアクセス

```
1. public class Outer {
2.     private String message = "Hello, Java";
3.     class Inner {
4.         public void test() {
5.             System.out.println(message);
6.         }
7.     }
8. }
```

このように、ネストしたクラスからエンクロージングクラスのメンバにアクセスできますが、「staticインナークラスからエンクロージングクラスの非staticなフィールドやメソッドにはアクセスできない」という制約があります。この制約は、**staticの仕組み**によるものです。staticで修飾されたフィールドやメソッドは、クラスがロードされたタイミングでstatic領域に配置され、インスタンスを生成しなくても、すぐに使用可能になります。一方、staticではないフィールドやメソッドは、ヒープ領域にインスタンスを生成して初めて使えるようになります。このように、配置されるメモリ領域や使えるタイミングが違うため、staticなメソッドから非staticなフィールドやメソッドにアクセスすることはできません。
これと同様に、**staticインナークラスは、エンクロージングクラスのインスタンス化を必要としない**ため、エンクロージングクラスの非staticなフィールドやメソッドにアクセスすることはできないのです。staticインナークラスからアクセスできるのは、staticなエンクロージングクラスのフィールドやメソッドだけです。

前述の例を次のようにstaticで修飾するとコンパイルエラーが発生します。なお、フィールドmessageをstaticで修飾するとコンパイルエラーは解消します。

例 staticで修飾したインナークラス

```
1. public class Outer {
2.     private String message = "Hello, Java";
3.     static class Inner {
4.         public void test() {
5.             System.out.println(message);
6.         }
```

```
7.      }
8. }
```

設問のコードでは、staticインナークラスInnerのexecuteメソッドで、エンクロージングクラスの非staticなフィールドnumにアクセスしています。このため、11行目でコンパイルエラーが発生します。よって、選択肢**B**が正解です。

5. C

➡P14

インナークラスとstaticなメンバに関する問題です。
(staticではない) インナークラスには、staticなフィールドやメソッドを定義することはできません。これらはあくまでもstaticなネストしたクラスにしか定義できないことに注意してください。反対に、staticなネストしたクラスには、staticなものも、そうでないものも定義できます。

設問のクラスでは、Outerクラスの中にインナークラスInnerが定義されています。Innerはstaticなネストしたクラスではないため、前述のとおりstaticなフィールドやメソッドは定義できません。よって、選択肢**C**のようにコンパイルエラーが発生します。

なお、設問のコードを実行できるようにするには、次のように修正します。
このコード例を実行すると、「Hello, Java」とコンソールに表示されます。

例 設問のコードの修正版

```
1. public class Outer {
2.     void test() {
3.         Inner.message = "Hello, Java";
4.     }
5.     static class Inner {
6.         private static String message;
7.         public void test() {
8.             System.out.println(message);
9.         }
10.    }
11.    public static void main(String[] args) {
12.        new Outer().test();
13.        new Inner().test();
14.    }
15. }
```


6. C

➡P15

ローカルクラスに関する問題です。
インナークラスやstaticインナークラスは、エンクロージングクラスのすべてのメソッドから利用可能です。この利用可能な範囲を特定のメソッドだけに絞ったものが、メソッド内に定義する**ローカルクラス**です。ローカルクラスは、メソッド内に定義するため、たとえ同じクラスであったとしても、ほかのメソッドから利用することはできません。これは、ローカル変数が、同じクラスの別のメソッドからアクセスできないのと同じ理由です。
ローカルクラスの使い方の特徴は、メソッドごとに実装内容を変えられることです。たとえば次のコード例では、helloメソッドとbyeメソッドは共にTestインタフェース型のインスタンスへの参照を戻しますが、それぞれ実装が異なるローカルクラスを定義し、そのインスタンスへの参照を戻しています。

例 ローカルクラスの使用例

```
1. public class Factory {
2.     public static Test hello() {
3.         class Hello implements Test {
4.             @Override
5.             public void execute() {
6.                 System.out.println("hello.");
7.             }
8.         }
9.         return new Hello();
10.    }
11.    public static Test bye() {
12.        class Bye implements Test {
13.            @Override
14.            public void execute() {
15.                System.out.println("bye.");
16.            }
17.        }
18.        return new Bye();
19.    }
20. }
```

このFactoryクラスのhelloメソッドとbyeメソッドを使い、それぞれのインスタンスのexecuteメソッドを呼び出しているのが、次のmainメソッドです。

※次ページに続く

例 ローカルクラスのメソッド呼び出し

```
1. public class FactoryTest {
2.     public static void main(String[] args) {
3.         Factory.hello().execute();
4.         Factory.bye().execute();
5.     }
6. }
```

実行結果は、次のとおりです。

例 実行結果

```
hello.
bye.
```

ローカルクラスは、そのクラスを定義したメソッド内のローカル変数にアクセスすることができます。次のコード例では、testメソッドでローカル変数nameを宣言、初期化し、それをローカルクラスのhelloメソッドで参照しています。

例 ローカルクラスを定義したメソッド内のローカル変数にアクセスする

```
1. public class IllegalAccessSample {
2.     public void test() {
3.         String name = "sample";
4.         class Sample {
5.             public void hello() {
6.                 System.out.println("Hello, " + name);
7.             }
8.         }
9.         new Sample().hello();
10.     }
11. }
```

なお、ローカルクラスが参照するローカル変数は、**ローカルクラスの定義よりも前に宣言**されていなければいけません。そのため、次のようにローカルクラスよりもあとにローカル変数を宣言するとコンパイルエラーになります。

例 ローカルクラスの定義後にローカル変数を宣言した例(コンパイルエラー)

```
1. public class IllegalAccessSample {
2.     public void test() {
3.         class Sample {
4.             public void hello() {
5.                 System.out.println("Hello, " + name);
6.             }
7.         }
8.         String name = "sample";
9.         new Sample().hello();
10.     }
11. }
```

また、ローカルクラス内から参照するローカル変数は、**実質的final**でなければいけません。この「実質的final」とは、たとえそのローカル変数がfinal修飾子で修飾されていなかったとしても、変更してはいけないという意味です。もし、このローカル変数の値を次のように変更してしまうとコンパイルエラーが発生します。

例 実質的finalなローカル変数の値を変更した例（コンパイルエラー）

```
1. public class IllegalAccessSample {
2.     public void test() {
3.         String name = "sample";
4.         class Sample {
5.             public void hello() {
6.                 System.out.println("Hello, " + name);
7.             }
8.         }
9.         name = "test";
10.        new Sample().hello();
11.    }
12. }
```

これは、ローカル変数とローカルクラスのインスタンスでは、**ライフサイクルの違い**があるためです。ローカル変数は、メソッドの処理が終わるとメモリ上から消えてしまいます。一方のローカルクラスのインスタンスは、その参照が残っていればメソッドの処理が終了してもメモリ上に存在し続けます。
こうしたライフサイクルの違いにより、ローカルクラスのインスタンスから、

そのローカルクラスを定義したメソッド内のローカル変数にアクセスしようとしてもできないという事態が発生する可能性があります。このような事態を防ぐ手段として、ローカルクラスのインスタンスを生成したときに、ローカル変数の値をコピーしたものをインスタンスに渡します。こうすることで、ローカルクラスのインスタンスはローカル変数と同じ値を持つため、メソッドの処理が終了してローカル変数がメモリ上から消えても、ローカルクラスのインスタンスは正常に動作します。
しかしローカル変数が実質的finalでなければ、この仕組みによって、ローカル変数とその値のコピー値を持つローカルクラスとの間に不整合が生じることになります。そのため、ローカルクラス内から参照するローカル変数の値を変更すると、コンパイルエラーが発生するのです。

設問のコードでは、Sampleクラスのtestメソッド内にローカルクラスAを定義しています。ローカルクラスAはexecuteメソッドを持っており、このメソッドは7行目でSampleインスタンスのフィールドであるmessageとローカル変数の一種であるメソッド引数valueにアクセスしています。testメソッドではローカルクラスAの定義後に、実質的にfinalでなければならないvalueの値を「localClass」に変更しています。そのため、コンパイルエラーが発生します。
以上のことから、選択肢**C**が正解となります。

試験対策

ローカルクラスから参照するローカル変数は、ローカルクラスよりもあとにローカル変数を宣言するとコンパイルエラーになります。

試験対策

ローカルクラスから参照するローカル変数は実質的finalでなければいけません。値を変更するとコンパイルエラーになります。これは、ローカルクラスのインスタンスとローカル変数のライフサイクルの違いによるものです。

参考

Java以外のプログラミング言語には、ローカルクラスのインスタンスからローカル変数にアクセスするときに、自由にローカル変数の値を変更できる「クロージャー」と呼ばれる仕組みが用意されているものがあります。これは、ローカル変数がメソッドを処理するためのメモリ領域(スタック領域)ではなく、別の領域に作られることで実現しています。

7. B

➡P16

匿名クラスに関する問題です。
匿名クラスとは、名前のないクラスのことです。匿名クラスは、インタフェースを実現したクラス、抽象クラスか具象クラスを継承したサブクラスとして、名前のないクラスを宣言し、そのクラスがどのようなフィールドやメソッドを持つべきかという、クラスの実装内容だけを定義したものです。
たとえば、次のコードではjava.lang.Runnableインタフェースを実現した匿名クラスを定義しています。Runnableはインタフェースであるためインスタンス化できませんが、このようにコンストラクタ呼び出しの後ろに中カッコ「{}」で囲ったクラスブロックを定義することで、匿名クラスの定義と同時に、そのクラスのインスタンスを生成することができます。

例 Runnableインタフェースを実現した匿名クラス

```
1. public class ThreadSample {
2.     public static void main(String[] args) {
3.         // {}内が実装部分だけを定義した匿名クラス
4.         Runnable run = new Runnable() {
5.             @Override
6.             public void run() {
7.                 System.out.println("Hello");
8.             }
9.         };
10.        // do something
11.    }
12. }
```

前問ではメソッド内に定義するローカルクラスについて解説しましたが、クラスの定義を特定のメソッド内だけで使用し、ほかのメソッドと共有しないのであれば、匿名クラスを使用するほうがよいでしょう。

匿名クラスは、既存のクラスの一部分だけをオーバーライドした一時的なクラスを作るときにも使えます。たとえば次のコードでは、java.util.ArrayListクラスが持つaddメソッドを、ログを出力するようにオーバーライドしています。

例 メソッドをオーバーライドし、一時的なクラスとして定義した匿名クラス

```
1. import java.lang.System.Logger;
2. import java.lang.System.Logger.Level;
```

```
 3. import java.util.ArrayList;
 4. import java.util.List;
 5.
 6. public class ListSample {
 7.     public static void main(String[] args) {
 8.         List<String> list = new ArrayList<String>() {
 9.             Logger logger = System.getLogger("sample");
10.             @Override
11.             public boolean add(String value) {
12.                 logger.log(Level.INFO, "ADD LOG:" + value);
13.                 return super.add(value);
14.             }
15.         };
16.         list.add("A");
17.         list.add("B");
18.         list.add("C");
19.     }
20. }
```

また、匿名クラスはインタフェースやスーパークラスにない、独自のメソッドを定義することも可能です。次のコードでは、java.lang.Objectクラスを継承した匿名クラスを定義し、Objectクラスにはないtestという独自のメソッドを追加しています。

例 独自のメソッドを定義した匿名クラス

```
 1. public class ObjectSample {
 2.     public static void main(String[] args) {
 3.         Object obj = new Object() {
 4.             public void test() {
 5.                 System.out.println("test");
 6.             }
 7.         };
 8.         obj.test();
 9.     }
10. }
```

しかし、このコードの3行目で匿名クラスのインスタンスへの参照をObject型変数に代入しているため、Object型にはtestメソッドがないとしてコンパイルエラーが発生してしまいます。Java SE 10以降は、**varを使った変数宣言**

ができるようになったことで、こうした問題を解決できます。コンパイラが匿名クラスの定義情報から、どのようなメソッドがあるかを推論してくれるのです。

例 変数objをvarで型推論するように変更し、インスタンスへの参照を代入する

```
 1. public class ObjectSample {
 2.     public static void main(String[] args) {
 3.         var obj = new Object() {
 4.             public void test() {
 5.                 System.out.println("test");
 6.             }
 7.         };
 8.         obj.test();
 9.     }
10. }
```

設問のコードでは、Sampleクラスを継承した匿名クラスを定義しています。このクラスは、modifyというスーパークラスにはなかった独自メソッドを定義しています。Mainクラスの3行目では、この匿名クラスのインスタンスへの参照を、varを使った変数sampleで扱っているため型推論が働き、8行目でコンパイルエラーが発生することはありません。以上のことから、選択肢**B**が正解となります。
modifyメソッドは、setNumメソッドを使ってスーパークラスのフィールドnumの値を100に書き換えています。よって、選択肢Aは誤りです。

8. C

➡P17

匿名クラスに関する問題です。
匿名クラスには、コンストラクタは定義できません。コンストラクタは、クラス名と同じ名前のメソッドでなければいけませんが、匿名クラスには名前がないためです。匿名クラスを初期化したい場合は、コンストラクタの代わりに初期化子「{}」を使用します。

匿名クラスは、インタフェースを実現したクラス、抽象クラスか具象クラスを継承したサブクラスのいずれかになります。注意する必要があるのは、newしているクラスそのものをインスタンス化しているわけではない点です。たとえば次のコードでは、java.lang.Objectクラスのインスタンスではなく、Objectクラスを継承した匿名のサブクラスのインスタンスを生成しているのです。

例 Objectクラスを継承した匿名クラス

```
1. public class Test {
2.     public static void main(String[] args) {
3.         Object obj = new Object() {
4.             @Override
5.             public String toString() {
6.                 return "hello.";
7.             }
8.         };
9.     }
10. }
```

また、匿名クラスでは、実現したインタフェースのデフォルトメソッドや、スーパークラスから引き継いだメソッドなどを利用することができます。
次のコードでは、java.util.ArrayListクラスを継承したサブクラスを匿名クラスとして定義してインスタンス化し、初期化ブロックの中でスーパークラスであるArrayListクラスが持つaddメソッドを呼び出してコレクションに要素を追加しています。

例 初期化ブロックの中で、コレクションに要素を追加する

```
1. import java.util.ArrayList;
2. import java.util.List;
3.
4. public class ListSample {
5.     public static void main(String[] args) {
6.         List<String> list = new ArrayList<String>() {
7.             {
8.                 super.add("A");
9.                 super.add("B");
10.                super.add("C");
11.            }
12.        };
13.        for(String str : list) {
14.            System.out.println(str);
15.        }
16.    }
17. }
```

設問のコードでは、Sampleクラスを継承した匿名クラスを定義していますが、Sampleという名前でコンストラクタを定義しようとしています。しかし、前述のとおり匿名クラスにはコンストラクタは定義できません。よって、このメソッドは戻り値型が定義されていないと解釈されてコンパイルエラーが発生します。以上のことから、選択肢Cが正解となります。

試験対策

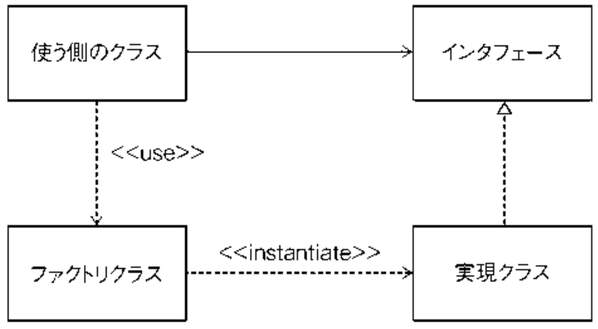
匿名クラスには名前がないため、コンストラクタは定義できません。匿名クラスを初期化したい場合は、コンストラクタの代わりに初期化子を使います。

9. B、C、D

➡P18

Java SE 8よりも前のインタフェースには、抽象メソッドと定数しか定義ができませんでした。そのため、インタフェースの機能を利用しようとするなら、そのインタフェースを実現したクラスのインスタンスを生成するファクトリクラスが必要でした。

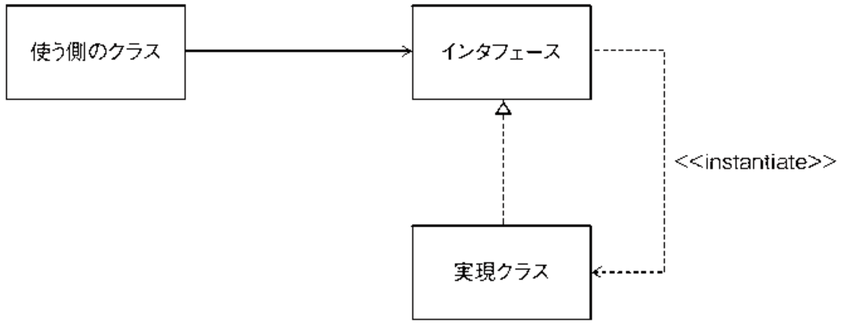
【インタフェース、実現クラス、ファクトリクラスと使う側のクラスの関係】



しかし、Java SE 8からはインタフェースにstaticメソッドを定義できるようになったため、ファクトリクラスを定義しなくても、実現クラスのインスタンス化が可能になりました。これによって、使う側のクラスは、インタフェースだけを意識すればプログラミングができるようになったのです。

※次ページに続く

【インタフェース、実現クラス、使う側のクラスの関係】



インタフェースのstaticメソッドは、設問のインタフェースAのように、staticで修飾したメソッドを定義するだけです。クラスに定義するstaticメソッドとの違いは、呼び出し時に継承したサブインタフェースや実現したクラスの名前を使うことができず、必ず定義したインタフェース名を使う必要がある点です。
そのため、設問のようにAインタフェースに定義したstaticメソッドは、「A.test()」で呼び出すことはできても、それを継承したBやCを使って、「B.test()」や「C.test()」のように呼び出すことはできません。もちろん、実現クラスであっても、継承したメソッドのようにメソッド名だけで呼び出すことはできません。以上のことから、Dクラスの4、5、6行目はコンパイルエラーとなります。よって、選択肢**B**、**C**、**D**が正解です。

10. A

➡ P19

インタフェースのstaticメソッドに関する問題です。
インタフェースに定義したstaticメソッドは、そのインタフェースを実現したクラスでオーバーライドすることはできません。そのため、次のコード例のように**@Overrideアノテーション**を付けるとコンパイルエラーとなります。

```
1. public class SampleImpl implements Sample {
2.     @Override
3.     public static void test() {
4.         System.out.println("B");
5.     }
6.     public static void main(String[] args) {
7.         Sample.test();
8.     }
9. }
```


なお、これはインタフェースに限ったことではなく、クラスに定義したstaticメソッドもオーバーライドすることはできません。これは、オーバーライドとは、インスタンスがどのように動作すべきかを再定義するための機能だからです。staticメソッドはインスタンスがなくても動作するため、オーバーライドできないのです。

設問のSampleインタフェースとSampleImplクラスでは、testという同じ名前のstaticメソッドを定義しています。staticメソッドを呼び出すには、そのメソッドが定義されているインタフェース名かクラス名を使います。
SampleImplクラスの6行目では「Sample.test()」としているため、Sampleインタフェースに定義されたメソッドが実行されます。SampleImplクラスのtestメソッドを呼び出したい場合は、「SampleImpl.test()」と記述します。以上のことから、選択肢**A**が正解となります。

11. A

➡ P20

デフォルトメソッドのオーバーライドに関する問題です。
インタフェースの**デフォルトメソッド**をオーバーライドしたとき、オーバーライドした側のメソッドからデフォルトメソッドを呼び出すことができます。サブクラスからスーパークラスのメソッドを呼び出すときと違い、**superキーワードの前に、呼び出すインタフェース名**を記述します。たとえば、Aインタフェースのデフォルトメソッドを実現クラスから呼び出すには、次のように記述します。

```
A.super.test();
```

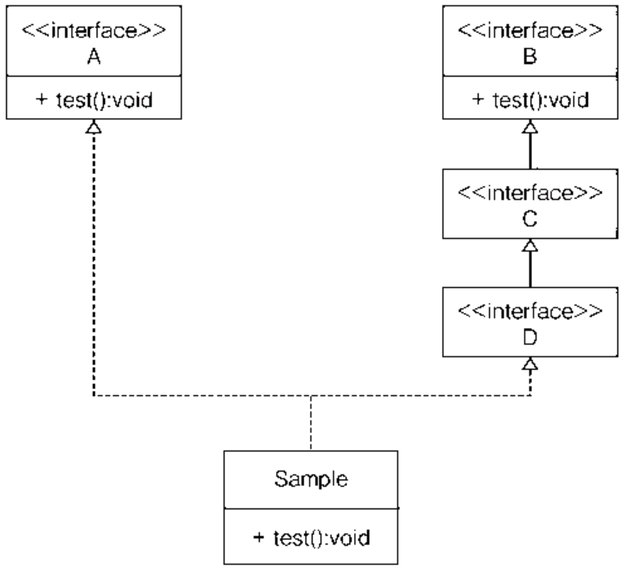
なお、インタフェースのデフォルトメソッドのオーバーライドは、実現クラスだけでなく、そのインタフェースを継承したサブインタフェースでも可能です。

オーバーライドしたメソッドから、インタフェースのデフォルトメソッドを呼び出すとき、呼び出せるのは直接の関係があるインタフェースか、そのインタフェースが継承している1つ上の階層にあたるスーパーインタフェースだけです。2つ以上の階層にあるスーパーインタフェースが持っているデフォルトメソッドを呼び出すことはできません。

たとえば、設問のインタフェースとクラスをクラス図で表現すると、次のようになります。このとき、Sampleクラスでオーバーライドしたtestメソッドが呼び出せるデフォルトメソッドは、直接実現しているAインタフェースのtestメソッドだけです。よって、選択肢**A**が正解となります。

※次ページに続く

【設問のインタフェースとクラスの関係】



Bインタフェースのtestメソッドは、BとSampleの間にCインタフェースと、それを継承したDインタフェースがあるため、Sampleクラスのtestメソッドから呼び出すことはできません。よって、選択肢Bは誤りです。
オーバーライドしたメソッドからインタフェースのデフォルトメソッドを呼び出すには、前述のとおりインタフェース名を記述します。このインタフェースは、デフォルトメソッドが定義されているものでなくてはいけません。継承したサブインタフェースのCにはデフォルトメソッドが定義されていないため、選択肢Cのように記述することはできません。

12. B

➡ P21

デフォルトメソッドに関する問題です。
設問では、Aインタフェースに定義しているtestメソッドと同じシグニチャのメソッドがBクラスにも定義されています。また、Cクラスでは、Aインタフェースを実現しつつ、さらにBクラスを継承しています。そのため、Cクラスのインスタンスには、Aインタフェースのtestメソッドと、Bクラスのtestメソッドの両方の定義が存在することになります。このような場合、**スーパークラスのメソッドが優先的に呼び出されます**。これは、あくまでもインタフェースは「型」を提供することが目的であり、本来、実装はクラスが提供すべきだからです。以上のことから、選択肢**B**が正解となります。

13. C

➡ P22

デフォルトメソッドに関する問題です。
スーパークラスとインタフェースが同一シグニチャのメソッドを持っている場合、スーパークラスのメソッドが優先的に呼び出されます。これは、インタフェースのデフォルトメソッドは、あくまでもインタフェースを実現するクラスが持つべき標準的な実装を提供するためのものだからです。もし、実現クラスが独自にそのメソッドの実装を持つならそちらを優先しようという考え方です。スーパークラスがサブクラスに引き継いだメソッドも、インタフェースを実現したクラスが独自にメソッドの実装を用意したものと考えるため、デフォルトメソッドよりも優先されるのです。
なお、スーパークラスのメソッドがpublic以外の場合は、コンパイルエラーとなります。Javaには、「メソッドをオーバーライドするときは、アクセス修飾子を緩くすることはできても、厳しくすることはできない」というルールがあります。これは、「サブクラスは、スーパークラスに置き換え可能でなければいけない」とする「リスコフの置換原則」を守るためです。この原則は、インタフェースと実現クラスの間にも適用され、実現クラスはインタフェースと完全に互換性がなければいけません。

設問のコードでは、Sampleクラスが実現しているAインタフェースと、継承しているBクラスが、同じシグニチャのメソッドを持っています。コンパイラは、SampleクラスはBクラスを引き継いだ上で、Aインタフェースを実現していると解釈します。Bクラスから引き継いだtestメソッドが、Aインタフェースに定義されているtestメソッドと互換性がないと判断されるため、コンパイルエラーが発生します。よって、選択肢**C**が正解です。

スーパークラスとインタフェースが同一シグニチャのメソッドを持っている場合、スーパークラスのメソッドが優先的に呼び出されます。

14. B

➡ P23

インタフェースのprivateメソッドに関する問題です。
Java SE 8で追加されたデフォルトメソッドをより簡潔にする目的で、Java SE 9では**インタフェースにprivateメソッド**を追加できるようになりました。これにより、複数のデフォルトメソッド内で同じ処理をしていた場合、その処理をまとめることでコードをより簡潔に記述できるようになりました。

インタフェースのprivateメソッドは、クラスに定義するprivateメソッドと同

じ構文で記述できます。また、default修飾子は必要ありません。よって、選択肢Bが正解で、選択肢Cは誤りです。
インタフェースは本来、公開する型を定義するためのものです。そのため、インタフェースに定義する抽象メソッドはpublicと見なされます。選択肢Aのように処理内容を持たない抽象メソッドをprivateにすることはできません。インタフェースのprivateメソッドは、あくまでもデフォルトメソッドから利用されるのを想定していることを忘れないようにしましょう。

なお、インタフェースにはstaticメソッドを定義できますが、同じ処理を実行する複数のstaticメソッドをまとめるにはprivate staticなメソッドを使います。「staticなメソッドは、非staticなメソッドにはアクセスできない」というルールはprivateメソッドの場合も同じで、staticで修飾する必要があります。

15. A

➡ P23

Enumに関する問題です。
Enum（列挙型）は、定数をまとめて定義するためのものです。導入の歴史は古く、Java SE 5.0（2004年）です。Enumが導入される以前、プログラムのあちこちで使う定数を定義するためには、インタフェースが使われていました。次のコードは、インタフェースにint型の定数を定義した例です。

例 定数を定義したFruitsインタフェース

```
1. public interface Fruits {
2.     public final static int APPLE = 0;
3.     public final static int ORANGE = 1;
4.     public final static int BANANA = 2;
5. }
```

たとえば、次のようにswitch文を使ってnumberの値によって処理を分岐するコードがあったとします。

例 switch文で分岐処理を行うSampleクラス

```
1. public class Sample {
2.     public static void main(String[] args) {
3.         int number = 0;
4.         switch(number) {
5.             case 0:
6.                 System.out.println("apple");
7.                 break;
8.             case 1:
9.                 System.out.println("orange");
10.                break;
11.            case 2:
12.                System.out.println("banana");
13.                break;
14.        }
15.    }
16. }
```

このコードの問題点は、これらの数字の意味がコードを読んでも理解できないことです。10年〜20年という長いソフトウェアのライフサイクルの中で、このコードをメンテナンスする人が代わっていけば、このような意味のわかりにくいコードはメンテナンス性を下げ、バグを生み出す原因にもなるでしょう。
そこで、次のコード例のように、先ほどインタフェースに定義した定数を使うのです。数値で分岐したコードよりも、何を意味しているかが理解しやすくなります。

例 定数を使った分岐処理

```
1. public class Sample {
2.     public static void main(String[] args) {
3.         int number = 0;
4.         switch(number) {
5.             case Fruits.APPLE:
6.                 System.out.println("apple");
7.                 break;
8.             case Fruits.ORANGE:
9.                 System.out.println("orange");
10.                break;
11.            case Fruits.BANANA:
12.                System.out.println("banana");
13.                break;
14.        }
15.    }
16. }
```

この方法は広く利用されていましたが、ソフトウェアの規模が大きくなるにつれて定数の数が増え、定数値が重複してしまうなどのバグを生むことがあ

りました。次のコードは、定数名は異なるものの、定数値がAPPLEとMELONで重複してしまっている例です。

例 値の重複した定数を定義しているFruitsインタフェース

```
1. public interface Fruits {
2.     public final static int APPLE = 0;
3.     public final static int ORANGE = 1;
4.     public final static int BANANA = 2;
5.     public final static int MELON = 0;
6. }
```

プログラムの制御は、定数名ではなく、定数値で行うため、次のようにAPPLEとMELONが等しいというプログラムも成り立ってしまいます。

```
if (Fruits.APPLE == Fruits.MELON) {
    System.out.println("joke?");
}
```

このようなことを防ぐために、定数をまとめて定義するために導入されたのがEnumです。Enumの定義は次の構文で行います。

構文

アクセス修飾子 enum 列挙名 {列挙子1, 列挙子2, ……};

前述のインタフェースをEnumに変更すると次のようなコードになります。

例 Enumの定義

```
1. public enum Fruits {
2.     APPLE, ORANGE, BANANA, MELON;
3. }
```

前述のSampleクラスを、Enumを使った例に変更してみましょう。

例 Enumを使った分岐処理

```
1. public class Sample {
2.     public static void main(String[] args) {
3.         Fruits type = Fruits.APPLE;
4.         switch(type) {
5.             case APPLE:
6.                 System.out.println("apple");
7.                 break;
8.             case ORANGE:
9.                 System.out.println("orange");
10.                break;
11.            case BANANA:
12.                System.out.println("banana");
13.                break;
14.        }
15.    }
16. }
```

もちろん、先ほどのAPPLEとMELONを比較していたコードも、Enumの列挙子を使って比較すればfalseが戻るため、「joke?」と表示されることはありません。なお、JavaのEnumは、CやC++、C#などのプログラミング言語と異なり、**クラスとして扱われます。**実際、プログラマーが独自に定義したEnumは、**java.lang.Enumクラスのサブクラス**として扱われます。また、プログラムの実行時には、Enumに定義した各列挙子が、それぞれ別のインスタンスとして生成されます。前述のFruitsであれば、APPLEとORANGE、BANANA、MELONの4つのインスタンスが、初めてFruitsを読み込んだタイミングで生成されます。
java.lang.Enumクラスの定義は、APIドキュメント※1で確認することができます。

なお、プログラマーが定義した独自のEnumには、コンパイラによって上記のAPIドキュメントに載っていない次の2つのメソッドが追加されています。

```
public T[] values();
public T valueOf(String);
```

valuesメソッドは、そのEnumに定義されている列挙子を配列に入れて戻すメソッドです。たとえば、A、B、Cという列挙子を持つEnumのTestを定義し、このEnumを使う次のコードを実行すると、列挙子A、B、Cがコンソールに表示されます。

※次ページに続く

※1 https://docs.oracle.com/javase/jp/11/docs/api/java.base/java/lang/Enum.html

```
for (Test t : Test.values()) {
    System.out.println(t);
}
```

例 実行結果

```
A
B
C
```

valuesメソッドの戻り値は配列ですが、特定の列挙子だけを戻すこともできます。たとえば、次のコードを実行すれば2つ目（要素の添字が1）の列挙子である「B」がコンソールに表示されることになります。

```
System.out.println(Test.values()[1]);
```

コンパイラによって自動的に追加されるもう1つのメソッドが、**valueOfメソッド**です。これは列挙子の名前から、列挙子のインスタンスを取得するためのものです。次のコードを実行すると、列挙子Aがそのままコンソールに表示されます。

```
System.out.println(Test.valueOf("A"));
```

設問では、3つある列挙子のうち、最初の列挙子Aをコンソールに表示しなければいけません。Enumに定義されている列挙子を取り出すにはvaluesメソッドを使います。valuesメソッドは列挙子のインスタンスへの参照を配列に入れて戻すため、要素の添字を指定して列挙子にアクセスします。また、valuesメソッドは、Enumで定義した順で列挙子を並べて配列の各要素に入れるため、たとえば列挙子Aは配列の0番目の要素に入っていることになります。以上のことから、選択肢**A**が正解となります。選択肢Cでは、列挙子Bが取り出されます。また、valuesメソッドは引数を受け取りません。よって、選択肢BとDは誤りです。
選択肢EとFはvalueOfメソッドを使っていますが、このメソッドは前述のとおり、列挙子の名前を文字列で指定して列挙子のインスタンスを取得するものです。よって、これらの選択肢も誤りです。

試験対策

Javaでは、Enum(列挙型)はクラスとして扱われます。プログラマーが独自に定義したEnumは、Enumクラスのサブクラスとして扱われます。Enumに定義した各列挙子は、プログラムの実行時にそれぞれ別のインスタンスとして生成されます。
プログラマーが定義した独自のEnumには、コンパイラによってvaluesとvalueOfという2つのメソッドが追加されます。これらのメソッドの使い方を理解しましょう。

16. D

➡ P24

Enumに関する問題です。
Enumがクラスであることは前問で解説しました。Enumが使用されるとき、**列挙子が1つずつインスタンス化**されます。たとえば、次のような列挙子を持つEnumを使用すると、A、B、Cの3つのTest型のインスタンスが生成されるのです。

例 列挙子A、B、Cを持つEnum、Test

```
1. public enum Test {
2.     A, B, C
3. }
```

Enumに定義されている列挙子のインスタンスは、それが使用されるタイミングでJVMが自動的に生成します。たとえば次のコード例では、Test型列挙子Aのインスタンスを生成し、その参照をTest型変数testに代入しています。

```
Test test = Test.A;
```

なお、列挙子のインスタンスが生成されるのは、1回だけです。もし列挙子を使用するタイミングで毎回インスタンスが生成されるのであれば、次のコードの結果は「false」となるはずです。

例 列挙子Aのインスタンス化の検証

```
1. public class Main {
2.     public static void main(String[] args) {
3.         sample(Test.A);
4.     }
5.     private static void sample(Test test) {
6.         System.out.println(Test.A == test);
```

```
7.     }
8. }
```

しかし、このコードを実行すると、「Test.A」と「test」が同じ参照を持っているため、結果は「true」となります。このように列挙子のインスタンスは、1つしか存在しないようになっています。これは、プログラマーが定義した独自のEnumをコンパイルすると、コンパイラによって自動的に追加されるstatic初期化子の中で各列挙子のインスタンスが生成され、staticフィールドに代入されるからです。

例 Testをコンパイルした結果のコードのイメージ（javapコマンドの実行結果から抜粋）

```
public final class Test extends java.lang.Enum<Test> {
  public static final Test A;
  public static final Test B;
  public static final Test C;

  static {
    // インスタンスを生成して3つのstaticフィールドを初期化するコード
  };

  … 省略 …
}
```

このように、Enumはコンパイラによって自動的に定数のコードが生成されるだけであって、前問の冒頭で解説したインタフェースを使った定数と同じことをしているにすぎません。

単なるint型の定数である他のプログラミング言語のEnumとは異なり、JavaのEnumはクラスの一種です。そのため、フィールドやメソッドを定義することもできます。たとえば、次のコードではコンストラクタを追加しています。

```
1. public enum Test {
2.     A, B, C;
3.     private Test() {
4.         System.out.println("test");
5.     }
6. }
```


このコンストラクタは、この列挙型が初めて使用されるタイミングで、各列挙子のインスタンスがstatic初期化子内で生成されるときに自動的に呼び出されます。そのため、次のように列挙子が3つあるTestを使うと、コンストラクタが3回呼び出され、コンソールには「test」と3回表示されたあと、列挙子Aが表示されます。

```
System.out.println(Test.A);
```

もし、引数を受け取るコンストラクタを定義した場合、次のように列挙子を記述する箇所でカッコ「()」を使って値を渡すことができます。

```
1. public enum Test {
2.     A("hoge"), B("hege"), C("fuge");
3.     Test(String value) {
4.         System.out.println(value);
5.     }
6. }
```

この仕組みを使えば、Enumにフィールドを持たせ、各列挙子に渡した任意の値でフィールドを初期化することも可能になります。たとえば、次のコードでは、コンストラクタで受け取った値でフィールドvalueを初期化しています。

```
 1. public enum Test {
 2.     A("hoge"), B("hege"), C("fuge");
 3.     private String value;
 4.     private Test(String value) {
 5.         this.value = value;
 6.     }
 7.     @Override
 8.     public String toString() {
 9.         return this.value;
10.     }
11. }
```

また、Enumはクラスであるため、このコードのようにjava.lang.Objectクラスから引き継いだtoStringメソッドをオーバーライドし、valueフィールドの値を戻すようにすることも可能です。
System.outのprintlnメソッドは、引数で渡されたオブジェクトのtoStringメソッ

ドを内部的に呼び出し、その結果をコンソールに表示します。そのため、次のコードを実行すると、列挙子Aのインスタンスが持つvalueフィールドの値「hoge」が表示されます。

```
System.out.println(Test.A);
```

なお、Enumに定義するコンストラクタのアクセス修飾子は、**private**でなくてはいけません。これは、Enumのインスタンスを生成するのは、JVMだけにしたいという理由です。もし、誰でも自由にEnumのインスタンスを生成できると、引数を受け取るコンストラクタを利用して任意の値をフィールドに持たせることが可能となり、定数の集合というEnumの意義がなくなってしまうからです。

設問のコードでは、コンストラクタ内で引数をコンソールに表示しています。3つの列挙子は、すべて同じ「hello」という初期値を渡していますので、コンソールには「hello」が3回表示されます。

例 設問のコード

```
1. public enum Sample {
2.     A("hello"), B("hello"), C("hello");
3.     private final String value;
4.     private Sample(String value) {
5.         System.out.println(value);
6.         this.value = value;
7.     }
8.     @Override
9.     public String toString() {
10.        return this.value;
11.    }
12. }
```

さらに、mainメソッドでSystem.outのprintlnメソッドにSample.Aを渡しているため、toStringメソッドが呼び出され、もう1回「hello」が表示されます。よって、コンソールには「hello」が合計4回表示されます。以上のことから、選択肢**D**が正解です。

Enumに定義するコンストラクタのアクセス修飾子は、privateでなくてはいけません。省略した場合は、暗黙的にprivateであると解釈されます。

2

関数型インタフェースとラムダ式

■ java.util.functionパッケージの関数型インタフェース

■ 関数型インタフェースの定義と記述

■ ラムダ式の利用

1. java.util.function.Supplierインタフェースで宣言されている抽象メソッドとして、正しいものを選びなさい。なお、Tはジェネリクスの型パラメータとする。(1つ選択)

A. `T get();`
B. `T set();`
C. `void put(T t);`
D. `void add(T t);`

➡ P67

2. 次のコードのうち、正しく記述されているものを選びなさい。(1つ選択)

A.
```
Supplier<String> s =
        (str) -> System.out.println(str);
s.get("test");
```

B.
```
Supplier<String> s = (str) -> "test" + str;
System.out.println(s.get());
```

C.
```
Supplier<String> s = () -> "test";
System.out.println(s.get());
```

D.
```
Supplier<String> s = () -> "test";
System.out.println(s.apply());
```

➡ P68

3. java.util.function.Consumerインタフェースで宣言されている抽象メソッドとして、正しいものを選びなさい。(1つ選択)

A. `void spend(T t);`
B. `void accept(T t);`
C. `T purchase();`
D. `T refund();`

➡ P73

4. 次のコードのうち、正しく記述されているものを選びなさい。(1つ選択)

A.
```
Consumer<String> c = x -> System.out.println(x);
c.accept("hello");
```

B.
```
Consumer<String> c = x -> "Hello, " + x;
System.out.println(c.accept("test"));
```

C.
```
Consumer<String> c = () -> "Hello, test";
System.out.println(c.accept());
```

D.
```
Consumer<String> c =
        () -> System.out.println("test");
c.accept();
```

➡ P74

5. 次のコードのうち、正しく記述されているものを選びなさい。(1つ選択)

A. `BiConsumer<Integer, Integer> test = (a, b) -> a + b;`
B.
```
BiConsumer<Integer, Integer> test =
        (a, b) -> System.out.println(a + b);
```
C. `BiConsumer<Integer, Integer> test = (a) -> a * 2;`
D.
```
BiConsumer<Integer, Integer> test =
        (a) -> System.out.println(a * 2);
```

➡ P79

6. java.util.function.Predicateインタフェースに定義されている抽象メソッドとして、正しいものを選びなさい。(1つ選択)

A. get
B. apply
C. accept
D. test

➡ P81

7. 次のプログラムをコンパイル、実行したときの結果として、正しいものを選びなさい。(1つ選択)

```
1. import java.util.function.Predicate;
2.
3. public class Q7 {
4.     public static void main(String[] args) {
5.         Predicate<Integer> p1 = x -> x > 0;
6.         Predicate<Integer> p2 = x -> x < 100;
7.         System.out.println(p1.or(p2).test(100));
8.     }
9. }
```

A. 「true」と表示される
B. 「false」と表示される
C. コンパイルエラーが発生する
D. 実行時に例外がスローされる

➡ P82

8. 次のコードのうち、正しく記述されているものを選びなさい。(1つ選択)

A. `BiPredicate<Integer, Integer> p = (a, b) -> a + b;`
B. `BiPredicate<Integer, Integer> p = (a, b) -> a == b;`
C. `BiPredicate<Integer> p = (a) -> a > 0;`
D. `BiPredicate<Integer> p = (a) -> a == 0;`

➡ P85

9. java.util.function.Functionインタフェースに定義されている抽象メソッドとして、正しいものを選びなさい。(1つ選択)

A. get
B. accept
C. apply
D. test

➡ P86

10. 次のようなコードがある。

```
Function<Integer, Integer> a = x -> x + 2;
Function<Integer, Integer> b = x -> x * 2;
```

これらを組み合わせて「22」という結果をコンソールに表示したい。正しいコードを選びなさい。(1つ選択)

A. `System.out.println(a.compose(b).apply(10));`
B. `System.out.println(a.andThen(b).apply(10));`
C. `System.out.println(a.after(b).apply(10));`
D. `System.out.println(a.before(b).apply(10));`

➡ P87

11. 次のコードのうち、正しく記述されているものを選びなさい。(1つ選択)

A.
```
BiFunction<Integer, Integer, String> test
        = (a, b) -> a + b;
```

B.
```
BiFunction<Integer, Integer, String> test
        = (a, b) -> Integer.toString(a + b);
```

C.
```
BiFunction<Integer, String> test
        = (a, b) -> a + b;
```

D.
```
BiFunction<Integer, String> test
        = (a, b) -> Integer.toString(a + b);
```

➡ P89

□ **12.** java.util.function.UnaryOperatorインタフェースに定義されている抽象メソッドとして、正しいものを選びなさい。（1つ選択）

A. execute
B. do
C. get
D. apply

➡ P90

□ **13.** リストで扱っているコレクションに対して、一律に処理を実行したい。使用する関数型インタフェースとして、最も適切なものを選びなさい。（1つ選択）

A. Function
B. Consumer
C. UnaryOperator
D. Supplier

➡ P92

□ **14.** 次のコードのうち、正しく記述されているものを選びなさい。（1つ選択）

A. `BinaryOperator op = (a, b) -> a + b;`
B. `BinaryOperator<Integer> op = (a, b) -> a + b;`
C. `BinaryOperator<Integer, Integer> op = (a, b) -> a + b;`
D. `BinaryOperator<Integer, Integer, Integer> op = (a, b) -> a + b;`

➡ P93

第2章　関数型インタフェースとラムダ式
解　答

1. A

➡ P62

Javaでは、ラムダ式を利用するために、あらかじめいくつかの**関数型インタフェース**が用意されています。これら関数型インタフェースは、**java.util.functionパッケージ**に分類されていますが、**java.lang.Runnable**や**java.util.Comparator**、**java.io.FileFilter**など、その他のパッケージにも数多く提供されています。すべての関数型インタフェースが出題されるわけではありませんが、以下のものについてはしっかりと特徴を理解しておきましょう。

【主な関数型インタフェース】

パッケージ	インタフェース	戻り値型	メソッド
java.util.function	Supplier<T>	T	get()
java.util.function	Consumer<T>	void	accept(T)
java.util.function	BiConsumer<T, U>	void	accept(T, U)
java.util.function	Predicate<T>	boolean	test(T)
java.util.function	BiPredicate<T, U>	boolean	test(T, U)
java.util.function	Function<T, R>	R	apply(T)
java.util.function	BiFunction<T, U, R>	R	apply(T, U)
java.util.function	UnaryOperator<T>	T	apply(T)
java.util.function	BinaryOperator<T>	T	apply(T, T)
java.lang	Runnable	void	run()
java.util.concurrent	Callable<V>	V	call()

なお、RunnableとCallableについては、第3章で解説します。

ここでは、まず**java.util.function.Supplierインタフェース**から学んでいきましょう。このインタフェースはSupplier（提供者）という名前のとおり、**引数を受け取らずに値を戻す処理**を規定するものです。Supplierは、ジェネリクスの型パラメータを1つ受け取り、**getメソッド**の戻り値の型として利用します。たとえば、次のコード例では、ジェネリクスの型パラメータとしてjava.util.List型を受け取るSupplier型の変数supplierを用意し、そこにjava.util.ArrayListのインスタンスを生成して参照を戻すラムダ式を代入しています。

⇒次ページに続く

例 Supplierインタフェースの使用例

```
1. import java.util.ArrayList;
2. import java.util.List;
3. import java.util.function.Supplier;
4.
5. public class Main {
6.     public static void main(String[] args) {
7.         Supplier<List<String>> supplier = () -> new ArrayList();
8.         List<String> list = supplier.get();
9.         // do something
10.     }
11. }
```

Supplierインタフェースに規定されているgetメソッドを使うとラムダ式が実行され、このタイミングでArrayListのインスタンスを生成して参照を取得することができます。
以上のことから、選択肢**A**が正解です。

試験対策

前ページの表に挙げた関数型インタフェースの特徴をしっかりと把握しておきましょう。各インタフェースの戻り値の型、抽象メソッド名とその引数(型や数)は、問題を解くために必須の知識です。

2. C

➡ P62

関数型インタフェースのSupplierに関する問題です。
java.util.function.Supplierは、Supplier（提供者）という名前のとおり、引数を受け取らずに値を戻す処理を規定するものです。この関数型インタフェースは、インスタンスを生成するファクトリクラスの代わりに利用できます。たとえば、次のようなAインタフェースとこれを実現しているBクラスを用意します。

例 Aインタフェース

```
1. public interface A {
2.     void hello();
3. }
```


例 Bクラス

```
1. public class B implements A {
2.     @Override
3.     public void hello() {
4.         System.out.println("Bの独自処理");
5.     }
6. }
```

このBクラスのインスタンスを生成し、その参照をA型で戻すメソッドを定義した**ファクトリクラス**を作ります。

例 ファクトリクラス

```
1. public class Factory {
2.     public static A create() {
3.         return new B();
4.     }
5. }
```

このように生成したインスタンスへの参照をインタフェース型で戻すため、次のコードのように、利用しているクラスはどの実現クラスが動作しているかを知る必要はありません。

例 Bのインスタンスへの参照を受け取るTestクラス

```
1. public class Test {
2.     public void execute() {
3.         System.out.println("お決まりの事前処理");
4.         A a = Factory.create();
5.         a.hello();
6.         System.out.println("お決まりの事後処理");
7.     }
8. }
```

これらのクラスを動作させるmainメソッドを持ったクラスは次のとおりです。このコードを実行すると、Bクラスが動作していることがわかります。

※次ページに続く

例 mainメソッドを持つSampleクラス

```
1. public class Sample {
2.     public static void main(String[] args) {
3.         Test test = new Test();
4.         test.execute();
5.     }
6. }
```

例 実行結果

```
> java Sample
お決まりの事前処理
Bの独自処理
お決まりの事後処理
```

Testクラスのコードの中に、実際に動作しているBクラスに関するコードが1行も表れていないことに注目してください。このように、使っている側のクラスと、実際に使われる側のクラスが直接結び付かないようにすることで、使われる側の変更の影響を最小化することができるのです。
たとえば、次のようにファクトリクラスで生成するインスタンスをBクラスからCクラスに変更しても、Testクラスを修正せずに結果を変えることができます。

例 Cクラス

```
1. public class C implements A {
2.     @Override
3.     public void hello() {
4.         System.out.println("Cの独自処理");
5.     }
6. }
```

例 ファクトリクラス（変更後）

```
1. public class Factory {
2.     public static A create() {
3.         return new C();
4.     }
5. }
```


例 実行結果

```
> java Sample
お決まりの事前処理
Cの独自処理
お決まりの事後処理
```

これらのコードをSupplierを利用したものに変更すると、ファクトリクラスが不要になります。まず、Testクラスのexecuteメソッドを次のようにSupplier型の引数を受け取るように変更します。

例 Testクラス（変更後）

```
1. import java.util.function.Supplier;
2.
3. public class Test {
4.     public void execute(Supplier<A> supplier) {
5.         System.out.println("お決まりの事前処理");
6.         A a = supplier.get();
7.         a.hello();
8.         System.out.println("お決まりの事後処理");
9.     }
10. }
```

そして、Testクラスのexecuteメソッドを呼び出す際に、Bのインスタンスを生成するラムダ式を引数として渡すと、Supplierのgetメソッドを呼び出したタイミングでラムダ式が実行されます。このとき、Bのインスタンスが生成され、その参照がA型として戻されます。

例 Sampleクラス（変更後）

```
1. public class Sample {
2.     public static void main(String[] args) {
3.         Test test = new Test();
4.         test.execute(() -> new B());
5.     }
6. }
```

Cクラスを使う場合も、ファクトリクラスを修正するのではなく、引数に渡すラムダ式を変更するだけです。ファクトリクラスを使わなくなった分だけ、コードが簡潔になりました。

例 executeメソッドに渡す引数を、Cのインスタンスを生成するラムダ式に変更

```
1. public class Sample {
2.     public static void main(String[] args) {
3.         Test test = new Test();
4.         test.execute(() -> new C());
5.     }
6. }
```

なお、関数型インタフェースはラムダ式だけでなく、次のようにメソッド参照でも利用できます。インスタンスを生成するだけであれば、ラムダ式よりもメソッド参照を利用したほうが簡潔に記述できます。

例 メソッド参照での記述（Bのインスタンスへの参照をexecuteに渡す）

```
1. public class Sample {
2.     public static void main(String[] args) {
3.         Test test = new Test();
4.         test.execute(B::new);
5.     }
6. }
```

例 メソッド参照での記述（Cのインスタンスへの参照をexecuteに渡す）

```
1. public class Sample {
2.     public static void main(String[] args) {
3.         Test test = new Test();
4.         test.execute(C::new);
5.     }
6. }
```

以上のことから、選択肢**C**が正解です。選択肢AとBは、ジェネリクスの型パラメータをラムダ式の引数に利用しています。Supplierは引数を受け取らないため、誤りです。選択肢Dは、ジェネリクスの型パラメータを戻り値の型として利用していますが、Supplierに定義されているメソッドはgetメソッドであり、applyメソッドではありません。よって、誤りです。

ここでは、GoF(Gang of Four)が記したデザインパターンから「Strategyパターン」を取り上げ、ラムダ式と関数型インタフェースを使ったコードを紹介しています。本書では、Strategyパターンそのものの説明は割愛しますが、Webサイトなどを参考に学習することをお勧めします。

3. B

➡ P62

関数型インタフェースのConsumerに関する問題です。
関数型インタフェースの1つである**java.util.function.Consumer**は、Consumer（消費者）という名前のとおり、**戻り値は戻さず、引数を受け取って、その引数を使った処理を実行するメソッド**を規定します。Consumerインタフェースに規定されている**acceptメソッド**は、ジェネリクスの型パラメータで指定した型の引数を受け取ります。

```
@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
}
```

たとえば次のコードでは、String型を扱うConsumer型変数に、引数をコンソールへ出力するラムダ式を代入しています。その後、acceptメソッドを呼び出して「Hello, Lambda」という文字列を引数として渡し、ラムダ式を実行しています。

例 Consumerインタフェースとacceptメソッドの使用例

```
1. import java.util.function.Consumer;
2.
3. public class Main {
4.     public static void main(String[] args) {
5.         Consumer<String> consumer = x -> System.out.println(x);
6.         consumer.accept("Hello, Lambda");
7.     }
8. }
```

このコードを実行すると、次のような結果がコンソールに表示されます。この結果から、acceptメソッドで受け取った引数がラムダ式に渡され、処理されていることがわかります。以上のことから、選択肢**B**が正解です。

※次ページに続く

例 実行結果

```
> java Main
Hello, Lambda
```

4. A

➡ P63

関数型インタフェースのConsumerに関する問題です。
関数型インタフェースである**java.util.function.Consumer**に規定されている**acceptメソッド**は、ジェネリクスの型パラメータで指定した型の引数を受け取り、その引数を使った処理を実行します。**戻り値は戻しません。**引数を受け取り、戻り値を戻さないラムダ式は、選択肢**A**です。よって、この選択肢が正解です。選択肢BやCは、戻り値を戻してコンソールに表示しようとしているため誤りです。選択肢Dは、戻り値は戻しませんが、引数を受け取っていません。よって、誤りです。

Consumerインタフェースは、インスタンスを生成する**ビルダー**に利用することができます。ビルダーとは、次のItemクラスのインナークラスBuilderのようなクラスを指します。

例 ビルダーを持つItemクラス

```
1. import java.util.function.Consumer;
2.
3. public class Item {
4.     private int id;
5.     private String name;
6.     private int price;
7.
8.     private Item(Builder builder) {
9.         this.id = builder.id;
10.        this.name = builder.name;
11.        this.price = builder.price;
12.    }
13.
14.    @Override
15.    public String toString() {
16.        return "Item [id=" + id + ", name=" + name + ",
                            price=" + price + "]";
17.    }
18.
19.    public static class Builder {
20.
21.        private int id;
22.        private String name;
23.        private int price;
24.
25.        public Builder(int id) {
26.            this.id = id;
27.        }
28.        public Builder setName(String name) {
29.            this.name = name;
30.            return this;
31.        }
32.        public Builder setPrice(int price) {
33.            this.price = price;
34.            return this;
35.        }
36.        public Item build() {
37.            return new Item(this);
38.        }
39.    }
40. }
```

ビルダーはコンストラクタの代わりに、setterメソッドを使ってインスタンスの初期化に必要な値を受け取ります。こうすることで、フィールドの種類が後々の仕様変更で変わっても、コンストラクタそのものを変更する必要がなくなるのです。

例 ItemクラスのインナークラスBuilderを使うSampleクラス

```
1. public class Sample {
2.     public static void main(String[] args) {
3.         Item item = new Item.Builder(100)
4.                         .setName("orange")
5.                         .setPrice(120)
6.                         .build();
7.         System.out.println(item);
8.     }
9. }
```

※次ページに続く

たとえば、最初は3つのフィールドの初期値を受け取るコンストラクタを定義していたものの、仕様変更でフィールドの数が4つになると、コンストラクタを4つの引数を受け取るものに変更しなくてはいけません。しかし、コンストラクタを変更すると、これまで3つの引数を渡してインスタンスを生成していたコードすべてが影響を受けてしまいます。

そこで、前述のItemクラスのようにビルダーを使います。Itemクラスの例では、コンストラクタをprivateで修飾することにより外部のクラスから使えないようにし、その代わりにインスタンスを生成するためだけのインナークラスをpublicで公開しています。
これは「Builderパターン」と呼ばれるGoFのデザインパターンの1つで、次の手順でItemクラスのインスタンスへの参照を取得します。

① インナークラスBuilderのインスタンスを生成する（Sampleクラスの3行目）
② 必要なsetterメソッドを呼び出し、Builderのフィールドに値をセットする（Sampleクラスの4〜5行目）
③ Builderのbuildメソッドを実行する（Sampleクラスの6行目）
   1. フィールドの値がセットされたBuilderのインスタンス自身への参照（this）を渡してprivateなItemのコンストラクタを呼び出す（Itemクラスの37行目）
   2. Itemのコンストラクタで、Builderのフィールドの値を自身のフィールドの初期値としてセットする（Itemクラスの8行目）
   3. コンストラクタがItemのインスタンスへの参照を戻す

このビルダーを使えば、Itemのコンストラクタを呼び出す必要がないため、フィールドの数が変わっても、コンストラクタの引数を変更する必要がありません。こうして、使われる側のクラスの変更が、使う側のクラスに影響しないようにする設計になりました。

しかし、このビルダーには、フィールドの数が増えれば、その分setterメソッドの数も増えるという問題が残っています。フィールドが数個であれば、これは大きな問題にならないかもしれませんが、多数になればsetterメソッドの管理が煩雑になったり、ソースコードの可読性が損なわれたりします。そこで、このような問題を解決するのに、Consumerを使います。

次のコード例では、ビルダーのフィールドをpublicとして公開し、Consumerを受け取る**withメソッド**を提供しています。引数のConsumerは、ジェネリクスの型パラメータとしてBuilder型が指定されています。そのため、引数として渡すラムダ式は、Builder型の引数を受け取ることになります。

例 Itemクラス（変更後）

```
1. import java.util.function.Consumer;
2.
3. public class Item {
4.     private int id;
5.     private String name;
6.     private int price;
7.
8.     private Item(Builder builder) {
9.         this.id = builder.id;
10.        this.name = builder.name;
11.        this.price = builder.price;
12.    }
13.
14.    @Override
15.    public String toString() {
16.        return "Item [id=" + id + ", name=" + name + ",
                                price=" + price + "]";
17.    }
18.
19.    public static class Builder {
20.
21.        private int id;
22.        public String name;
23.        public int price;
24.
25.        public Builder(int id) {
26.            this.id = id;
27.        }
28.        public Builder with(Consumer<Builder> consumer) {
29.            consumer.accept(this);
30.            return this;
31.        }
32.        public Item build() {
33.            return new Item(this);
34.        }
35.    }
36. }
```

このように変更したビルダーは、次のコード例のようにwithメソッドの引数

に渡すラムダ式で、公開されているフィールドに値を直接セットして使用します。

例 Sampleクラス（変更後）

```
1. public class Sample {
2.     public static void main(String[] args) {
3.         Item item = new Item.Builder(100)
4.                         .with(b -> {
5.                             b.name = "orange";
6.                             b.price = 120;
7.                         }).build();
8.         System.out.println(item);
9.     }
10. }
```

このラムダ式が呼び出されるのは、withメソッド内でConsumerのacceptメソッドが呼び出されるタイミングです。withメソッドの定義を再掲します。

例 withメソッドの定義（再掲）

```
28. public Builder with(Consumer<Builder> consumer) {
29.     consumer.accept(this);     //Builderのインスタンス自身への参照
30.     return this;
31. }
```

ここではthisを使ったビルダーのインスタンスへの参照を渡しています。このコード例は、次のように置き換えると理解しやすいでしょう。

例 withメソッドの引数でビルダーのインスタンスへの参照をthisに置き換える

```
1. public class Sample {
2.     public static void main(String[] args) {
3.         Item item = new Item.Builder(100)
4.                         .with(this -> {
5.                             this.name = "orange";
6.                             this.price = 120;
7.                         }).build();
8.         System.out.println(item);
9.     }
10. }
```


このラムダ式では、ビルダーのインスタンス自身への参照を引数に渡しているため、ビルダーのインスタンス自身のnameフィールドに「orange」という文字列、priceフィールドに「120」という数値をセットしています。その後、buildメソッドを呼び出しているので、ビルダーのフィールドの値を使ってItemのインスタンスが初期化され、その参照が戻されることになります。このようにBuilderパターンとラムダ式、関数型インタフェースを組み合わせることで、setterメソッドを増やすことなく、簡潔にコードを記述できるようになります。

5. B

➡ P63

関数型インタフェースのBiConsumerに関する問題です。
関数型インタフェースのConsumerは、引数を1つしか受け取りません。引数を2つ受け取るには、**java.util.function.BiConsumerインタフェース**を使います。このインタフェースは、Consumerと同じ**acceptメソッド**を規定していますが、引数の数が異なります。

```
@FunctionalInterface
public interface BiConsumer<T, U> {
    void accept(T t, U u);
}
```

BiConsumerを使ったコード例は次のとおりです。このコード例では、HashMapのインスタンスを作り、ラムダ式ではそのマップのputメソッドを呼び出すように定義しています。その後、acceptメソッドが2回呼び出されると、第1引数がputメソッドのキーに、第2引数がputメソッドのバリューに渡されます。

例 BiConsumerインタフェースの使用例

```
1. import java.util.HashMap;
2. import java.util.function.BiConsumer;
3.
4. public class Test {
5.     public static void main(String[] args) {
6.         var map = new HashMap<String, Integer>();
7.         BiConsumer<String, Integer> bi = (k, v) -> map.put(k, v);
8.
9.         bi.accept("apple", 3);
10.        bi.accept("orange", 2);
```

```
11.
12.         System.out.println(map);
13.     }
14. }
```

このコードを実行すると、次のようにHashMapにキーとバリューのセットが登録されていることがわかります。

例 実行結果

```
> java Test
{orange=2, apple=3}
```

なお、次のコード例のようにメソッド参照を使う方法もあります。こちらのほうがラムダ式よりも簡潔に記述できます。

例 メソッド参照での記述

```
1. import java.util.HashMap;
2. import java.util.function.BiConsumer;
3.
4. public class Test {
5.     public static void main(String[] args) {
6.         var map = new HashMap<String, Integer>();
7.         BiConsumer<String, Integer> bi = map::put;
8.
9.         bi.accept("apple", 3);
10.        bi.accept("orange", 2);
11.
12.        System.out.println(map);
13.    }
14. }
```

選択肢CとDは、1つしか引数を受け取りません。引数を2つ受け取るのがBiConsumerの特徴ですので、これらの選択肢は誤りです。選択肢Aは、引数を2つ受け取りますが、戻り値を戻しています。ConsumerとBiConsumerは、引数を消費するだけで、結果を戻しません。よって、選択肢Aは誤りです。以上のことから、選択肢**B**が正解です。

6. D

➡ P64

関数型インタフェースのPredicateに関する問題です。

java.util.function.Predicateは関数型インタフェースの1つで、Predicate（断定する）という名前からわかるとおり、**何らかの判断をする関数**を定義するためのものです。Predicateのtestメソッドは、**引数を1つ**受け取り、その引数を使って何らかの判定をし、その結果を**boolean型**で戻します。なお、このPredicateは、型パラメータを1つ受け取り、**testメソッド**の引数の型として利用されます。

```
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
}
```

たとえば次のコード例では、String型の引数を受け取るPredicate型の変数p1を用意し、そこに引数が空かどうかを調べる**isEmptyメソッド**を呼び出すラムダ式を代入しています。testメソッドを呼び出すとき、空文字を引数に渡しているため、変数p1に代入されているラムダ式はtrueを戻します。

例 Predicateインタフェースの使用例

```
1. import java.util.function.Predicate;
2.
3. public class Main {
4.     public static void main(String[] args) {
5.         Predicate<String> p1 = x -> x.isEmpty();
6.         System.out.println(p1.test(""));
7.     }
8. }
```

例 実行結果

```
> java Main
true
```

以上のことから、選択肢**D**が正解です。

7. A

➡ P64

関数型インタフェースのPredicateに関する問題です。
java.util.function.Predicateインタフェースは、複雑な条件や複数の条件を組み合わせるときに利用するとよいでしょう。たとえば、次のようなPersonクラスがあったとします。このクラスは性別（gender）、年齢（age）のフィールドを持っています。

例 Personクラス

```
1. public class Person {
2.     public enum Gender {
3.         MALE, FEMALE
4.     }
5.     private Gender Gender;
6.     private int age;
7.     public Person(Gender gender, int age) {
8.         super();
9.         this.gender = gender;
10.        this.age = age;
11.    }
12.    public Gender getGender() {
13.        return gender;
14.    }
15.    public int getAge() {
16.        return age;
17.    }
18. }
```

従来の方法を使って、このPersonのインスタンスが条件に一致するかどうかを調べるのが次のコード例です。このコードでは、19歳の男性のインスタンスを生成し、そのインスタンスが条件に合致するかどうかを調べています。見てわかるとおり、if文が論理演算の組み合わせから成っており、理解しづらいものになっています。

例 論理演算の組み合わせによる条件判定

```
1. import Person.Gender;
2.
3. public class OldSample {
4.     public static void main(String[] args) {
5.         Person sample = new Person(Gender.MALE, 19);
6.
7.         if (sample.getGender().equals(Gender.MALE)
8.             || sample.getGender().equals(Gender.FEMALE)
9.                 && sample.getAge() >= 20) {
10.            System.out.println("OK");
11.            return;
12.        }
13.        System.out.println("NG");
14.    }
15. }
```

上記のコードを、Predicateインタフェースを使って改善したのが次のコード例です。このコード例では、性別、年齢を1つずつ条件判定するラムダ式を代入したPredicate型の変数を3つ用意し、if文ではそれらを組み合わせて実行しています。Predicateインタフェースには、論理演算子「||」に相当する**orメソッド**や「&&」に相当する**andメソッド**というデフォルトメソッドが用意されています。これらを利用することで、男性、もしくは女性かつ成人であるというif文の条件式が理解しやすくなります。

例 Predicateインタフェースのデフォルトメソッドによる条件判定

```
1. import java.util.function.Predicate;
2. import Person.Gender;
3.
4. public class NewSample {
5.     public static void main(String[] args) {
6.         Predicate<Person> isMale =
7.                 p -> p.getGender().equals(Gender.MALE);
8.         Predicate<Person> isFemale =
9.                 p -> p.getGender().equals(Gender.MALE);
10.        Predicate<Person> isAdult =
11.                p -> p.getAge() >= 20;
12.
13.        Person sample = new Person(Gender.MALE, 19);
14.        if (isMale.or(isFemale.and(isAdult)).test(sample)) {
15.            System.out.println("OK");
16.            return;
17.        }
18.        System.out.println("NG");
```

```
19.     }
20. }
```

この方法は、コードの改善技法である「リファクタリング」のうち「説明用変数の導入」と呼ばれるものです。論理演算子がいくつも組み合わさると理解しづらくなりますが、説明用変数の導入を取り入れ、1つひとつの条件を表現する変数で論理演算を記述することにより、条件式が明確でわかりやすくなります。

Predicateインタフェースのandメソッドとorメソッドは、複数のPredicateオブジェクトを合成し、新しいPredicateオブジェクトを作るためのメソッドです。たとえば、次のようにp1とp2の2つのPredicate型変数があったとします。p1は、引数の文字列が「a」で始まっているかどうかを調べます。もう1つのp2は、引数の文字列が「e」で終わっているかどうかを調べます。

```
Predicate<String> p1 = str -> str.startsWith("a");
Predicate<String> p2 = str -> str.endsWith("e");
Predicate<String> p3 = p1.and(p2);
```

この2つのPredicateオブジェクトを合成して作ったのが、p3です。この例ではandメソッドを使いましたが、orメソッドも同じように使います。p3のtestメソッドを実行すると、渡した引数が、合成したp1とp2の両方に渡され、そのどちらもtrueを戻せば、p3はtrueを戻します。一方でもfalseを戻せば、p3はfalseを戻します。たとえば、次のようにtestメソッドの引数に「abcde」という文字列を渡せば、p1はtrue、p2もtrueを戻すため、p3はtrueを戻します。

```
System.out.println(p3.test("abcde"));
```

なお、このコードは変数p3を使っていないだけで、次のようにp3の部分を置き換えても、上記のコードと処理内容は同じです。

```
System.out.println(p1.and(p2).test("abcde"));
```

そのため、前述のコードを次のように書き換えることができます。こうすれば、より読みやすく、そして理解しやすくなるでしょう。

例 複数のPredicateオブジェクトを合成して複合条件を判定

```
1. import java.util.function.Predicate;
2. import Person.Gender;
3.
4. public class NewSample {
5.     public static void main(String[] args) {
6.         Predicate<Person> isMale =
7.                 p -> p.getGender().equals(Gender.MALE);
8.         Predicate<Person> isFemale =
9.                 p -> p.getGender().equals(Gender.FEMALE);
10.        Predicate<Person> isAdult =
11.                p -> p.getAge() >= 20;
12.
13.        Predicate<Person> criteria =
14.                isMale.or(isFemale.and(isAdult));
15.
16.        Person sample = new Person(Gender.MALE, 19);
17.        if (criteria.test(sample)) {
18.            System.out.println("OK");
19.            return;
20.        }
21.        System.out.println("NG");
22.    }
23. }
```

設問のコードでは、2つのPredicateオブジェクトを組み合わせた論理演算を実行しています。1つ目のPredicate型の変数p1では引数の値が0よりも大きいかどうかを調べ、2つ目のp2では引数の値が100よりも小さいかどうかを調べています。testメソッドは100を受け取っているため、1つ目はtrueを、2つ目はfalseを戻します。2つのPredicateオブジェクトはorメソッドで合成されるため、1つでもtrueを戻せば、全体でtrueになります。以上のことから、このコードを実行するとコンソールにはtrueと表示されます。よって、選択肢**A**が正解となります。

8. B

➡ P64

関数型インタフェースのBiPredicateに関する問題です。
java.util.function.Predicateが引数を1つ受け取ったのに対し、**java.util.function.BiPredicate**は、**引数を2つ**受け取ります。何らかの判定をする処理を記述したり、戻り値がboolean型である点でこれらのインタフェースは変わ

りありませんが、BiPredicateはジェネリクスの型パラメータを2つ受け取ります。この2つの型パラメータは**testメソッド**の引数の型として使われます。

```
@FunctionalInterface
public interface BiPredicate<T, U> {
    boolean test(T t, U u);
}
```

次のコード例が、BiPredicateの使用例です。この例では、String型の引数を2つ受け取り、1つ目の文字列が2つ目の引数の文字列で終わっているかどうかを調べます。

例 BiPredicateインタフェースの使用例

```
1. import java.util.function.BiPredicate;
2.
3. public class BiSample {
4.     public static void main(String[] args) {
5.         BiPredicate<String, String> b =
6.                 (str, suffix) -> str.endsWith(suffix);
7.         System.out.println(b.test("sample.class", ".class"));
8.     }
9. }
```

testメソッドに渡されているのは第1引数が「sample.class」、第2引数が「.class」という文字列です。「sample.class」は「.class」で終わっているため、このラムダ式はtrueを戻します。
選択肢CとDは、引数を1つしか受け取りません。BiPredicateは、2つの引数を扱うための関数型インタフェースですので、これらの選択肢は誤りです。選択肢Aは、Integer型の引数を2つ受け取っていますが、int型の結果を戻しています。BiPredicateの戻り値型はboolean型でなければいけません。よって、この選択肢も誤りです。以上のことから、選択肢**B**が正解です。

9. C

➡ P65

関数型インタフェースのFunctionに関する問題です。
java.util.funciton.Functionは、**引数を1つ受け取り、その引数を使った処理結果を1つ返す**処理を表す関数型インタフェースです。Functionは、ジェネリクスの型パラメータを2つ受け取ります。1つ目の型パラメータは、Functionの抽象メソッドである**applyメソッド**の引数の型として、2つ目の型パラメータはapplyメソッドの戻り値の型として使用されます。

```
@FunctionalInterface
public interface Function<T, R> {
    R apply(T t);
}
```

たとえば次のコード例では、ジェネリクスの型パラメータとしてStringとIntegerが指定されています。そのため、変数fで扱うラムダ式の引数はString型、戻り値はInteger型となります。このラムダ式では、引数をString型変数xで受け取り、StringクラスのlengthメソッドをIntegerで文字数を戻します。
実際にラムダ式が呼び出されるのはapplyメソッドが呼ばれたときです。このコードでは引数に「sample」という文字列を渡しているため、コンソールには「6」と表示されます。以上のことから、選択肢**C**が正解です。

例 Functionインタフェースの使用例

```
1. import java.util.function.Function;
2.
3. public class Sample {
4.     public static void main(String[] args) {
5.         Function<String, Integer> f = x -> x.length();
6.         System.out.println(f.apply("sample"));
7.     }
8. }
```

10. A

➡ P65

Functionの合成を行うメソッドに関する問題です。
java.util.function.Functionも、解答7で説明したjava.util.function.Predicateのように合成した新しいFunctionオブジェクトを作ることができます。合成することで、複数の処理をつなぎ合わせて1つの処理の流れを表現できます。Functionの合成は、**andThenメソッド**と**composeメソッド**の2つで行います。これらの違いは、実行する順番だけです。たとえば、a、bという2つのFunction型変数があったとき、a.andThen(b)ではaが先に実行され、そのあとにbが実行されます。一方、a.compose(b)ではbが先に実行され、そのあとにaが実行されます。andThenメソッドが順次処理、composeメソッドは逆順処理だと覚えておきましょう。

実際に合成したFunctionを実行した例が、次のコードです。このコード例では、

引数で受け取った値に2を加算して戻すplusと、引数で受け取った値を倍にして戻すtimesが使われています。

例 Functionインタフェースの使用例

```
1. import java.util.function.Function;
2.
3. public class Main {
4.     public static void main(String[] args) {
5.         Function<Integer, Integer> plus = (x) -> x + 2;
6.         Function<Integer, Integer> times = (x) -> x * 2;
7.
8.         System.out.println(plus.andThen(times).apply(5));
9.         System.out.println(times.andThen(plus).apply(5));
10.     }
11. }
```

コンソールに表示される結果の1行目として、applyメソッドの引数で渡された5に、plusで2が加算され、その後、timesで値を倍にした結果の14が表示されます。2行目はその逆で、先にtimesが実行されてからplusが実行されるため、5×2＋2という計算結果の12がコンソールに表示されます。

例 実行結果

```
> java Main
14
12
```

このコード例のように「plus.andThen(times).apply(5)」と記述されていると、plus、times、applyの順に実行されるかのように感じるかもしれません。しかし、andThenメソッドは合成した新しいFunctionオブジェクトを戻すため、applyメソッドは新しく合成されたFunctionオブジェクトのものを実行しているのです。前述のコード例を次のように書き換えるとわかりやすくなります。

例 合成したFunctionの生成とapplyメソッドの呼び出しを書き換えたコード

```
1. import java.util.function.Function;
2.
3. public class Main {
4.     public static void main(String[] args) {
5.         Function<Integer, Integer> plus = (x) -> x + 2;
6.         Function<Integer, Integer> times = (x) -> x * 2;
7.
8.         // 合成した新しいFunctionを生成
9.         Function<Integer, Integer> a = plus.andThen(times);
10.        Function<Integer, Integer> b = times.andThen(plus);
11.
12.        // 合成したFunctionのapplyメソッドを呼び出し
13.        System.out.println(a.apply(5));
14.        System.out.println(b.apply(5));
15.    }
16. }
```

設問のコードでは、22という結果を得るためにはFunction型変数bの式を先に実行し、そのあとに変数aの式を実行しなくてはいけません。bを先に実行するには、andThenメソッドではなく、composeメソッドを使う必要があります。よって、選択肢**A**が正解です。選択肢Bでは、先に加算が実行されてしまうため、24という結果がコンソールに表示されます。選択肢CのafterやDのbeforeというメソッドは、Functionインタフェースには存在しません。よって、これらの選択肢は誤りです。

オブジェクトの合成処理は、andThenメソッドが順次処理、composeメソッドは逆順処理だと覚えておきましょう。

11. B

➡ P65

関数型インタフェースのBiFunctionに関する問題です。
解答9で説明したjava.util.function.Functionは、1つの引数を処理して、結果を戻すための関数型インタフェースです。**java.util.function.BiFunction**は、2つの引数を処理して、結果を戻す関数型インタフェースです。このインタフェースは、ジェネリクスの型パラメータを3つ受け取ります。3つの型パラメータのうち、1つ目と2つ目は引数の型として、3つ目は戻り値の型として使われます。

```
@FunctionalInterface
public interface BiFunction<T, U, R> {
    R apply(T t, U u);
}
```

次のコード例が、BiFunctionの使用例です。このBiFunction型変数bでは、2つのString型の引数を受け取り、String型の戻り値を戻すラムダ式を扱うことができます。

例 BiFunctionインタフェースの使用例

```
 1. import java.util.function.BiFunction;
 2.
 3. public class BiSample {
 4.     public static void main(String[] args) {
 5.         BiFunction<String, String, String> b =
 6.                 (str, addStr) -> str.concat(addStr);
 7.
 8.         String result = b.apply("Hello, ", "Lambda");
 9.         System.out.println(result);
10.     }
11. }
```

変数bには、第1引数の文字列に第2引数の文字列を結合した新しい文字列を戻すラムダ式が代入されています。このラムダ式を実行するapplyメソッドでは、「Hello, 」と「Lambda」の2つの文字列を引数に渡しているため、2つが結合された「Hello, Lambda」という文字列が結果として戻され、コンソールに表示されます。

BiFunctionのジェネリクスで受け取る型パラメータは3つです。よって、2つしか引数を指定していない選択肢CとDは誤りです。3つの型パラメータのうち、1つ目と2つ目が引数、3つ目が戻り値の型として使われます。選択肢Aは、戻り値の型としてString型を指定していますが、実際に戻しているのはint型です。よって、このコードはコンパイルエラーとなります。以上のことから、選択肢**B**が正解です。

12. D

➡ P66

関数型インタフェースのUnaryOperatorに関する問題です。
java.util.function.UnaryOperatorは、1つの値を処理して同じ型の結果を戻す処理を表す関数型インタフェースです。UnaryOperatorは、Functionを継承したサブインタフェースで、Functionから引き継いだ抽象メソッドであるapplyメソッドを実装するよう実現します。

FunctionとUnaryOperatorとの違いは、Functionが引数の型と戻り値の型を別々に指定できたのに対し、UnaryOperatorは**引数の型と戻り値の型が同じ**という点です。以下に示したインタフェース宣言からわかるように、ジェネリクスの型パラメータは1つだけですが、スーパーインタフェースであるFunctionは、同じ型の型パラメータを2つ受け取ります。

```
@FunctionalInterface
public interface UnaryOperator<T> extends Function<T, T> {

}
```

次のコードは、UnaryOperatorの使用例です。UnaryOperatorは、処理対象となる値を1つだけ持つ単項演算子の処理を表現できる関数型インタフェースです。このコード例では、整数値を1つ増やす処理と、整数値を2倍にするラムダ式を、それぞれincrementとtwiceというUnaryOperator型の変数に代入しています。それぞれが**applyメソッド**を呼び出すときに10という整数値を引数に渡しているため、コンソールには11と20という値が表示されます。

例 UnaryOperatorインタフェースの使用例

```
 1. import java.util.function.UnaryOperator;
 2.
 3. public class Sample {
 4.     public static void main(String[] args) {
 5.         UnaryOperator<Integer> increment = x -> ++x;
 6.         System.out.println(increment.apply(10));
 7.
 8.         UnaryOperator<Integer> twice = x -> x * 2;
 9.         System.out.println(twice.apply(10));
10.     }
11. }
```

例 実行結果

```
> java Sample
11
20
```

以上のことから、選択肢**D**が正解です。

13. C

➡ P66

関数型インタフェースのUnaryOperatorに関する問題です。
java.util.function.UnaryOperatorは、リストの値を一律に処理する場合に使用できます。たとえば次のコードは、文字列のリストに対して一律に大文字に変更する処理を行います。具体的には、java.util.Listインタフェースの**replaceAllメソッド**がUnaryOperator型の引数を受け取り、リストのすべての要素をラムダ式の実行結果で置き換えます。

例 UnaryOperatorの使用例

```
1. import java.util.ArrayList;
2. import java.util.List;
3.
4. public class Main {
5.     public static void main(String[] args) {
6.         List<String> list = new ArrayList<>();
7.         list.add("a");
8.         list.add("b");
9.         list.add("c");
10.
11.         list.replaceAll(x -> x.toUpperCase());
12.         for (String str : list) {
13.             System.out.println(str);
14.         }
15.     }
16. }
```

例 実行結果

```
> java Main
A
B
C
```

UnaryOperatorは、このように元の値を変更するものの、型そのものは変更しない一律処理などに利用できます。
こうした処理は選択肢AのFunctionでも実現可能ですが、処理した結果の型を変える必要がないのであれば、UnaryOperatorのほうが適切です。以上のことから、選択肢**C**が正解です。

14. B

➡ P66

関数型インタフェースのBinaryOperatorに関する問題です。
java.util.function.BinaryOperatorは、**2つの値を処理して同じ型の結果を戻す処理**を表す関数型インタフェースです。BinaryOperatorはBiFunctionのサブインタフェースですが、BiFunctionとの違いは、BiFunctionが2つの引数の型と戻り値の型をすべて別々に指定できたのに対し、BinaryOperatorは**引数の型と戻り値の型がすべて同じ**という点です。
インタフェース宣言を見ても、ジェネリクスの型パラメータは1つだけですが、スーパーインタフェースであるBiFunctionは、3つとも同じ型の型パラメータを受け取ります。

```
@FunctionalInterface
public interface BinaryOperator<T> extends BiFunction<T, T, T> {

}
```

BinaryOperatorが受け取るジェネリクスの型パラメータは1つだけです。そのため、選択肢CとDは誤りです。選択肢Aは、ジェネリクスの型パラメータを渡していません。そのため、java.lang.Objectが型パラメータとして扱われます。設問のラムダ式では+演算子を使っていますが、この演算子でObject型を加算することはできません。よって、選択肢Aはコンパイルエラーとなります。以上のことから、選択肢**B**が正解です。

次のBinaryOperatorインタフェースの使用例では、String型を扱うBinaryOperator型の変数bに、String型の引数を2つ受け取り、それらを結合して1つの文字列として戻すラムダ式を代入しています。

例 BinaryOperatorインタフェースの使用例

```
1. import java.util.function.BinaryOperator;
2.
3. public class BiSample {
4.     public static void main(String[] args) {
5.         BinaryOperator<String> b = (str, add) -> str.concat(add);
6.         System.out.println(b.apply("Hello, ", "Lambda"));
7.     }
8. }
```

applyメソッドでは、「Hello, 」と「Lambda」という文字列が引数として渡さ

れているため、結合された「Hello, Lambda」という文字列が戻されて、コンソールに表示されます。

ラムダ式は、次のようにファクトリクラスを使って隠蔽し、関数型インタフェースだけで扱えるようにすると、可読性や再利用性が高まります。以下のOperatorFactoryクラスは、加算と減算処理のためのBinaryOperatorを戻す、addメソッドとminusメソッドを定義しています。

例 OperatorFactoryクラス

```
1. import java.util.function.BinaryOperator;
2.
3. public class OperatorFactory {
4.     public static BinaryOperator<Integer> add() {
5.         return (a, b) -> a + b;
6.     }
7.     public static BinaryOperator<Integer> minus() {
8.         return (a, b) -> a - b;
9.     }
10. }
```

たとえば、次のようにaddメソッドの結果を受け取り、実行することで加算が実行され、コンソールには加算の結果として「30」が表示されます。

例 UseOperatorクラス

```
1. import java.util.function.BinaryOperator;
2.
3. public class UseOperator {
4.     public static void main(String[] args) {
5.         BinaryOperator<Integer> op = OperatorFactory.add();
6.         System.out.println(op.apply(10, 20));
7.     }
8. }
```

このようにファクトリクラスを介することで、どのようなロジックで処理されるかを隠蔽することができます。ロジックに変更が生じた場合はファクトリクラスのラムダ式を変更すれば、クライアントであるUseOperatorクラスを一切修正せずに対応できます。

3

並列処理

- ■ RunnableインタフェースとThreadクラスを使った並行処理
- ■ スレッドプール、Executorフレームワーク
- ■ ExecutorServiceインタフェース
- ■ ScheduledExecutorServiceインタフェース
- ■ Executorsクラス
- ■ Futureインタフェース
- ■ Callableインタフェース
- ■ CyclicBarrierクラス
- ■ synchronizedキーワード
- ■ 原子性、java.util.concurrent.atomicパッケージのクラス
- ■ スレッドセーフ、CopyOnWriteArrayListクラス
- ■ ReentrantLockクラス

1. 次のプログラムをコンパイル、実行した結果として、正しいものを選びなさい。（1つ選択）

```
1. public class Main {
2.     public static void main(String[] args) {
3.         Thread t = new Thread() {
4.             @Override
5.             public synchronized void start() {
6.                 super.start();
7.                 System.out.println("A");
8.             }
9.             @Override
10.            public void run() {
11.                super.run();
12.                System.out.println("B");
13.            }
14.        };
15.        t.run();
16.    }
17. }
```

A. 「A」だけが表示される
B. 「B」だけが表示される
C. 常に「A」「B」の順に表示される
D. 常に「B」「A」の順に表示される
E. 「A」と「B」の両方が表示されるが、順序は不定である

➡ P102

2. マルチスレッドで並行処理をするための記述として、正しいものを選びなさい。（2つ選択）

A. Threadクラスを継承したクラスを作り、runメソッドをオーバーライドする
B. Threadクラスを継承したクラスを作り、startメソッドをオーバーライドする
C. Runnableインタフェースを実現したクラスを作り、runメソッドを実装する
D. Runnableインタフェースを実現したクラスを作り、startメソッドを実装する

➡ P105

3. あらかじめいくつかのスレッドを作ってプールしておき、時間の経過で不要なスレッドを削除するExecutorServiceを取得するExecutorsクラスのメソッドとして、正しいものを選びなさい。（1つ選択）

A. newSingleThreadExecutorメソッド
B. newFixedThreadPoolメソッド
C. newCachedThreadPoolメソッド
D. newManagedThreadPoolメソッド

➡ P107

4. java.util.concurrent.ScheduledExecutorServiceインタフェースを使って、1秒後にコンソールに「done.」と表示する遅延実行スレッドを作りたい。正しいコードを選びなさい。（1つ選択）

A.
```
exec.schedule(() -> {
    System.out.println("done.");
}, 1, TimeUnit.SECONDS);
```
B.
```
exec.schedule(() -> {
    System.out.println("done.");
}, 1000);
```
C.
```
exec.execute(() -> {
    System.out.println("done.");
}, 1, TimeUnit.SECONDS);
```
D.
```
exec.execute(() -> {
    System.out.println("done.");
}, 1000);
```

➡ P112

5. java.util.concurrent.ScheduledExecutorServiceインタフェースを使って処理を定期的に実行したい。使用するメソッドとして正しいものを選びなさい。（1つ選択）

A. scheduleメソッド
B. scheduleAtFixedRateメソッド
C. repeatメソッド
D. scheduleAtLoopメソッド

➡ P114

6. マルチスレッドで、タスクをほぼ一定間隔で実行したい。java.util.concurrent.ScheduledExecutorServiceインタフェースのどのメソッドを使用すればよいか。正しいものを選びなさい。（1つ選択）

A. scheduleメソッド
B. scheduleAtFixedRateメソッド
C. scheduleWithFixedDelayメソッド
D. scheduleDelayメソッド

➡ P116

7. java.util.concurrent.ScheduledExecutorServiceインタフェースを使用し、複数のスレッドをプールしながら効率よくマルチスレッドによる処理を実行したい。使用するメソッドとして、正しいものを選びなさい。（1つ選択）

A. ScheduledExecutorServiceのnewThreadPoolメソッド
B. ScheduledExecutorServiceのnewScheduledThreadPoolメソッド
C. ExecutorsのnewThreadPoolメソッド
D. ExecutorsのnewScheduledThreadPoolメソッド

➡ P119

8. Runnableインタフェースを使ってマルチスレッドによる処理を実行している。java.util.concurrent.Futureのgetメソッドは、スレッドが終了した時点で何を受け取るか。正しいものを選びなさい。（1つ選択）

A. int型の整数
B. boolean型の真偽値
C. java.util.concurrent.Result型の列挙定数
D. null

➡ P120

9. 次のコードを実行したとき、java.util.concurrent.Futureインタフェースのgetメソッドの戻り値として、正しいものを選びなさい。（1つ選択）

```
1. Future<Integer> future = exec.submit(() -> {
2.     try {
3.         // do something
4.     } catch (InterruptedException e) {
5.         throw new RuntimeException(e);
6.     }
7. }, 0);
```

A. 負の整数
B. 0
C. null
D. RuntimeExceptionがスローされる

➡ P122

10. RunnableとCallableの特徴として、正しいものを選びなさい。（2つ選択）

A. Runnableは戻り値を戻せない
B. Runnableは任意の戻り値を戻せる
C. Callableは戻り値を戻せない
D. Callableは任意の戻り値を戻せる

➡ P123

11. java.util.concurrent.Callableインタフェースを使ってタスクを定義している。タスク内で例外SampleExceptionをスローした場合、それを受け取る側では何型の例外を捕捉するcatchブロックを用意すればよいか。最も適切なものを選びなさい。（1つ選択）

A. SampleException
B. ExecutionException
C. FutureException
D. ConcurrentException

➡ P126

12. 複数のスレッドの終了を待って処理する同期化処理を実現したい。使用するクラスあるいはインタフェースとして、正しいものを選びなさい。（1つ選択）

A. CyclicBarrier
B. CopyOnWriteArrayList
C. ExecutorService
D. ScheduledExecutorService

➡ P128

13. マルチスレッド・アプリケーションで共有されるインスタンスで、競合が発生しないよう排他制御をしたい。このときに使用するキーワードとして、正しいものを選びなさい。（1つ選択）

A. synchronized
B. volatile
C. strictfp
D. transient

➡ P132

14. 複数のスレッドの間で、排他制御された複数のインスタンスを共有していて、それぞれのインスタンス同士が連携することで発生する待ち状態を意味する用語として、正しいものを選びなさい。（1つ選択）

A. 待ち行列
B. デッドロック
C. 同期化
D. ライブロック

➡ P136

15. synchronizedを使わず、フィールドの競合が発生しないようするために用意されたクラスとメソッドとして、正しいものを選びなさい。（1つ選択）

A. IntegerのaddAndGetメソッド
B. Integerのgetメソッド
C. AtomicIntegerのaddAndGetメソッド
D. AtomicIntegerのgetメソッド

➡ P140

16. マルチスレッド・アプリケーションで、あるコレクションを複数のスレッドで共有している。一方のスレッドがコレクションから要素を読み出している最中に、もう一方が変更を加えても例外が発生しないコレクションはどれか。正しいものを選びなさい。（1つ選択）

A. ArrayList
B. Vector
C. ThreadSafeList
D. CopyOnWriteArrayList

➡ P143

17. 複数のメソッドにまたがる排他制御を実現したい。最も適切な方法を選びなさい。（1つ選択）

A. クラス宣言をsynchronizedで修飾する
B. 排他制御したいすべてのメソッドをsynchronizedで修飾する
C. java.util.concurrent.atomicパッケージに属するクラスを利用する
D. ReentrantLockクラスを利用する

➡ P148

第3章　並列処理
解　答

1.　B　➡ P96

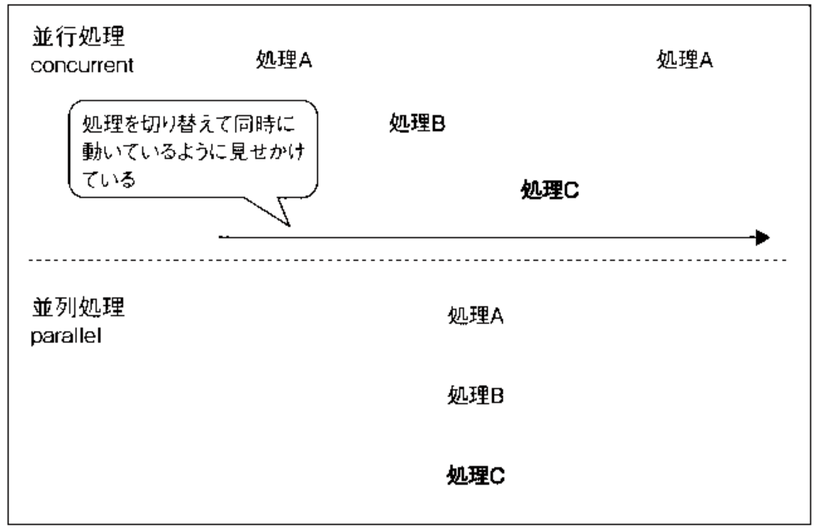
Javaのマルチスレッド・プログラミングに関する問題です。
プログラムの流れは、実行中の処理が終わるまで次の処理には進めません。たとえ次の処理がその内容的には待つ必要がないものであっても、実行中の処理が終わるまで、待たなくてはいけません。これが外部のサーバーからの返信待ちの処理だったりすると、マシンのCPUやメモリに余裕があるにもかかわらず待たされるのです。そのため、多くのシステムと連携するような大規模なシステムでは、大量のユーザーからの処理要求があった場合に待機させられてしまうことになり、結果としてパフォーマンスが悪化してしまうのです。このような「待ち」を解消するのが「並行処理」です。
並行処理は、複数の処理を同時に実行することです。複数の処理を「同時並行的に」処理できるため、ある処理が「待ち」状態になっていてもほかの処理を進めることができます。こうして全体としての「待ち」状態を減らすことができ、結果としてシステム全体のパフォーマンスが向上する効果が得られるのです。

並行処理は、複数の処理を一定のタイミングで切り替えることで実現します。A、B、Cという3つの処理があったとき、処理Aを0.1秒間実行し、次に処理Bを0.1秒、その次に処理Cを0.1秒、また処理Aを0.1秒…という具合に瞬時に切り替えることで、実際には一時点では1つの処理しか実行していないもかかわらず、人間には処理が同時並行的に実行されているように感じられるのです。

昨今のコンピュータが搭載しているCPUは「マルチコア」と呼ばれ、1つのCPUが複数のコアを持っています。そのため、従来のシングルコアの並行処理のように切り替えることなく、コアの数だけ同時に処理が進められるようになり、パフォーマンスが劇的に向上しています。なお、正確には、シングルコアで処理対象を切り替えながら実行することを「並行処理」、マルチコアで同時に処理を実行することを「並列処理」と呼びます。

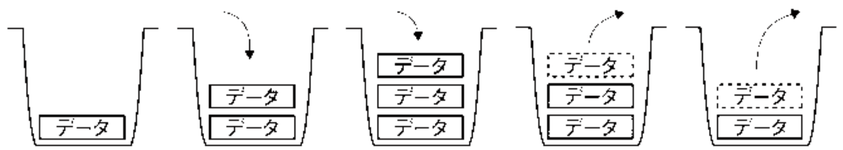
Javaでは、複数の処理を同時に実行するマルチスレッド・プログラミングが簡単に実装できるようになっています。Javaのマルチスレッド処理は、基本的にはシングルコアで動かす場合は並行処理ですが、複数のコアでそれぞれのスレッドを同時に動かすのであれば並列処理となります（もちろん、これはJVMの実装によります）。

【並行処理と並列処理のイメージ】



メソッドの呼び出し順を制御するために、Javaではスタック領域を使います。**スタック領域**とは、「スタック構造」と呼ばれるデータ管理方法で扱われるメモリ領域です。バケツの中の物を出し入れする様をイメージしてみるとよいでしょう。スタック領域（バケツ）にデータ（物）を入れると次々と上に積み重なっていき、取り出すときは入れた順とは逆に、上から出していく構造になっています。

【スタック構造】

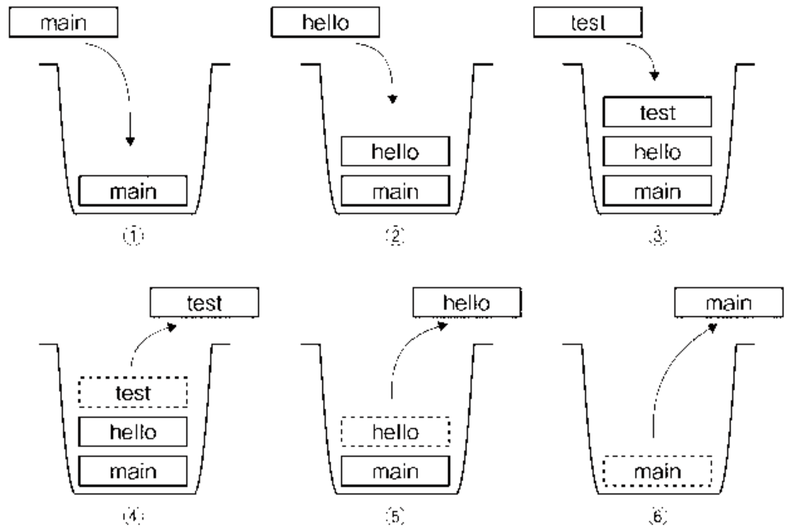


このようなデータ構造になっているメモリ領域に、実行したいメソッドを順番に入れてメソッドの処理順を管理しているのです。たとえば次ページの図では、メモリ領域に最初に入るのは、エントリーポイントであるmainメソッドです。

mainメソッド内でhelloメソッドを呼び出すと、helloメソッドの処理内容がスタックに追加されます。さらにhelloメソッド内でtestメソッドを呼び出すと、testメソッドの処理内容がスタックに追加されます。testメソッドの処理が完了すれば、testメソッドが取り出され、helloメソッドの残りの処理を実行します。helloメソッドの処理が終われば、helloメソッドが取り出されます。こうしてmainメソッドに戻り、mainメソッドがまた別のメソッドを呼び出す…という具合に、今、実行しているメソッドが一番上に来るようになっている

仕組みを使って、処理順を制御しているのです。

【メソッドの処理順の管理】



このようなメソッドがスタックに入ったり出たりするのを何度も繰り返す一連の流れを「**スレッド**」と呼びます。そして、スレッドを管理する複数のスタックが並行処理を行うことを**マルチスレッド**といいます。

マルチスレッドによる処理では、前述のようにシングルコアCPUであればスタックを次々と切り替えながらの並行処理を、マルチコアCPUであればスタックごとにコアを割り振っての並列処理を実現します。
mainメソッドを実行すると、mainメソッドから呼ばれたメソッドが次々とスタックに積まれては実行されて、取り出されます。これが**シングルスレッド**です。シングルスレッドから、別のスタックを作ってマルチスレッドで処理をするには、次のコード例のように**java.lang.Threadクラス**のサブクラスを実装します。Threadクラスにはいくつかのメソッドがありますが、新しいスレッドで実行したいことは**runメソッド**に記述します。ここでは、新しいスレッドで動作したことを表すために「sub」とコンソールに表示します。

例 新しいスレッドのクラスSampleThread

```
1. public class SampleThread extends Thread {
2.     @Override
3.     public void run() {
4.         System.out.println("sub");
5.     }
6. }
```


Threadのサブクラスが定義できたら、次のコード例のようにインスタンス化して実行してみましょう。新しいスタックを生成してスレッドを開始するには、Threadクラスのインスタンスを生成し、**startメソッド**を実行します。

例 SampleThreadのstartメソッドを実行するSampleクラス

```
1. public class Sample {
2.     public static void main(String[] args) {
3.         Thread t = new SampleThread();
4.         t.start();
5.
6.         System.out.println("main");
7.     }
8. }
```

このコードでは、新しいスレッドを開始したあと、コンソールに「main」と表示します。Threadクラスのstartメソッドを呼び出すと新しいスタックが生成され、そこでrunメソッドが実行されます。呼び出し元のスレッドでは、startメソッドを呼び出したタイミングで次のコードへと処理が移ります。多くの場合、メモリ領域に新しいスタックを生成する処理に時間がかかるためか、先にコンソールに「main」と表示されてから、新しいスレッドによって「sub」と表示されます。
しかし、何度か実行していると、数回に一度は「sub」と表示されてから「main」と表示されることもあります。どのスレッドが先に実行されるかはプログラムからは制御できず、JVMの実装やCPUのタイミングなどで決まります。
新しいスレッド（スタック）を作るのは、Threadクラスのstartメソッドです。設問のように、startメソッドを呼び出さずにrunメソッドを呼び出しても、新しいスレッドが開始されることはありません。そのため、コンソールには「B」のみが表示されます。よって、選択肢**B**が正解です。

2. A、C

➡ P96

マルチスレッドでの並行処理を実現する方法に関する問題です。
新しいスレッドを作るには、java.lang.Threadクラスを継承したサブクラスを定義するか、もしくは**java.lang.Runnableインタフェース**を実現したクラスを用意し、そのインスタンスをThreadクラスのコンストラクタに渡します。
Runnableインタフェースは、新しいスレッドで実行したい処理を記述するための**runメソッド**のみを持っています。次のコード例は、前問の解説で使ったものを、Runnableインタフェースを実現した匿名クラスに置き換えたものです。

※次ページに続く

例 Runnableを実現した匿名クラスを、Threadのコンストラクタに渡してスレッドを作る

```
1. public class Sample {
2.     public static void main(String[] args) {
3.         Thread t = new Thread(new Runnable() {
4.             @Override
5.             public void run() {
6.                 System.out.println("sub");
7.             }
8.         });
9.         t.start();
10.        System.out.println("main");
11.    }
12. }
```

よって、選択肢**C**が正解です。Runnableインタフェースには、startメソッドは存在しないため、選択肢Dは誤りです。

なお、Runnableインタフェースは関数型インタフェースでもあるため、次のコードのようにラムダ式で表現できます。このようにラムダ式を利用することで匿名クラスを利用する場合と比べてコード量が減り、可読性が向上します。

例 Threadのコンストラクタの引数をラムダ式で表現する

```
1. public class Sample {
2.     public static void main(String[] args) {
3.         Thread t = new Thread(() -> {
4.             System.out.println("sub");
5.         });
6.         t.start();
7.
8.         System.out.println("main");
9.     }
10. }
```

Threadクラスのstartメソッドは、そのメソッドを呼び出したスレッドで動作します。一方、runメソッドはstartメソッドによって作られた新しいスレッドで動作します。そのため、startメソッドとrunメソッドは異なるスレッドで動作することになります。よって、選択肢Bのようにstartメソッドをオーバーライドしてしまうと、新しいスレッドが生成されず、マルチスレッドで動作しなくなります。以上のことから、選択肢**A**が正解で、Bは誤りです。

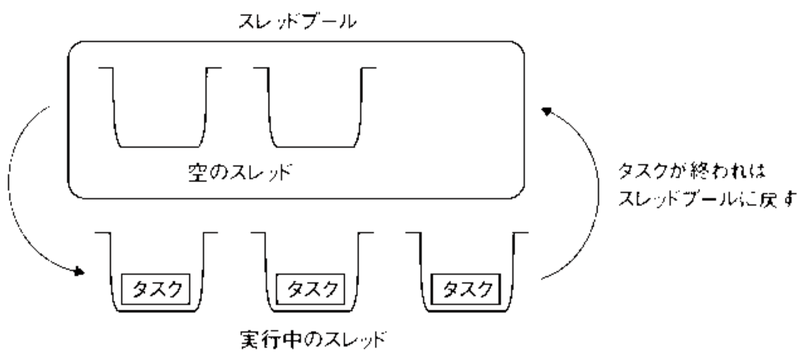
3. C

➡ P97

ExecutorフレームワークとExecutorServiceに関する問題です。
マルチスレッドによる並行処理はとても強力ですが、使い方を間違えるとパフォーマンスを上げるどころか、下げてしまうことになります。なかでも新しいスレッドの生成は、コンピュータにとって負荷の高い処理であるため、無駄なスレッド生成はできるだけ抑制すべきです。

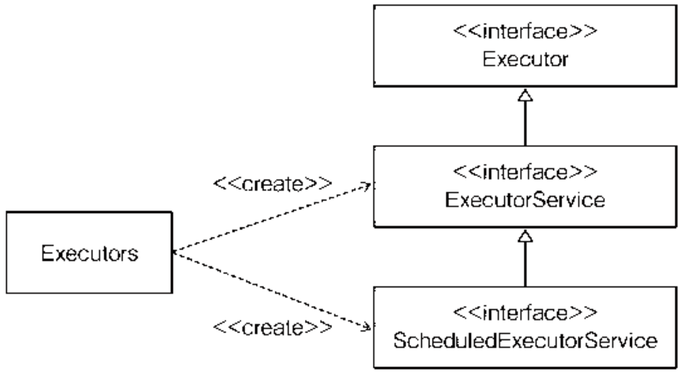
これまでに学んできたように、Threadクラスを作ったマルチスレッド・プログラミングでは、Threadクラスのサブクラスを作り、runメソッドをオーバーライドしたり、Runnableインタフェースの実装を定義してThreadクラスのコンストラクタに渡したりして、新しいスレッドで行う処理と、それを実行するスレッドのインスタンスをセットで扱います。
しかし、この方法では、実行したい処理の数だけ、新しいスレッドを生成する必要があります。実行したい処理が100個あれば、100個のスレッドを作らなければいけません。100個のスレッドを作りながら次々と処理を実行していくと、途中のスレッドを生成しているタイミングで、最初の頃に作ったスレッドの処理が終わっている可能性があります。つまり、すでに処理を終えて空き状態になったスレッドがあるにもかかわらず、コンピュータに負荷をかけて新しいスレッドを作らなければいけないのです。
このときに生じる「スレッドの無駄使い」という問題を解決するために、あらかじめスレッドを生成して保存しておき、使い回しをする仕組みがJava SE 5で導入されました。これが、「スレッドプール」です。
スレッドプールは、複数個の空のスレッドを最初から作っておき、そのスレッドにタスクを与えて実行させます。このようにタスクが与えられるまで待機する空のスレッドと、タスクを実行するスレッドを分けることで、前述の「スレッドの無駄使い」の問題を解決します。

【スレッドプールのイメージ】



スレッドプールを使うことで、効率的に並行処理を実現するのが「**Executorフレームワーク**」と呼ばれる一連のインタフェースとクラス群です。このフレームワークは、**java.util.concurrent.Executor**をスーパーインタフェースとした、次の図のようなインタフェースで構成されています。

【Executorフレームワーク】



具体的な使用方法としては、Executorのサブインタフェースである**java. util.concurrent.ExecutorService**、もしくは**java.util.concurrent.ScheduledExecutorService**を利用します。
これらサブインタフェースの実装を取得するのが**java.util.concurrent.Executorsクラス**です。このクラスは、ExecutorServiceインタフェースを実現したインスタンスへの参照を戻すファクトリメソッドを用途別にいくつか持っています。まず、次のコード例のように1つだけ新しいスレッドを使ってみましょう。

例 Executorsクラスの使用例（newSingleThreadExecutorメソッド）

```
 1. import java.util.concurrent.ExecutorService;
 2. import java.util.concurrent.Executors;
 3.
 4. public class Sample {
 5.     public static void main(String[] args) {
 6.         ExecutorService exec = Executors.newSingleThreadExecutor();
 7.         for (int i = 0; i < 5; i++) {
 8.             exec.submit(() -> {
 9.                 System.out.println(Thread.currentThread().getId());
10.             });
11.         }
12.     }
13. }
```

このコードでは、Executorsクラスの**newSingleThreadExecutorメソッド**を使って、mainメソッドとは別にタスクを待つ新しいスレッドを1つだけ作ってプールしている、ExecutorServiceを作っています。
そのあと、for文を使ってExecutorServiceのスレッドにタスクを与えて実行する**submitメソッド**を5回呼び出しています。このコード例では、Runnableを実装したラムダ式で実行中のスレッドのIDを表示しています。ここでは、submitメソッドが計5回呼び出されていますが、スレッドプールにはスレッドが1つしかないため、1つのスレッドが5回使い回しされることになります。そのため、コンソールには、すべて同じスレッドIDが表示されます。

例 実行結果

```
> java Sample
12
12
12
12
12
```

あらかじめ決まった数のスレッドを生成し、スレッドプールに貯めておき、使い回すこともできます。Executorsクラスの**newFixedThreadPoolメソッド**は、生成したいスレッド数を引数に受け取り、生成した分だけのタスク待ちのスレッドを保持するスレッドプールを作ります。次のコード例では、3つのタスク待ちのスレッドを保持したスレッドプールを作っています。

例 Executorsクラスの使用例（newFixedThreadPoolメソッド）

```
 1. import java.util.concurrent.ExecutorService;
 2. import java.util.concurrent.Executors;
 3.
 4. public class Sample {
 5.     public static void main(String[] args) {
 6.         ExecutorService exec = Executors.newFixedThreadPool(3);
 7.         for (int i = 0; i < 5; i++) {
 8.             exec.submit(() -> {
 9.                 System.out.println(Thread.currentThread().getId());
10.             });
11.         }
12.     }
13. }
```

※次ページに続く

このコードを実行すると、3種類のスレッドIDが表示されるため、3つのスレッドを使い回していることがわかります。

例 実行結果

```
> java Sample
12
13
12
12
14
```

newFixedThreadPoolメソッドのように同定数のスレッドではなく、必要に応じてスレッドを増減させるスレッドプールを作ることも可能です。**newCachedThreadPoolメソッド**を使うと、必要に応じてスレッドを生成することができます。一度生成されたスレッドは、60秒間使用されなければ破棄され、60秒未満であれば再利用されるため無駄がありません。よって、選択肢**C**が正解です。
次のコード例では、newCachedThreadPoolメソッドを使い、必要に応じてスレッド数を増減させるスレッドプールを作っています。

例 Executorsクラスの使用例（newCachedThreadPoolメソッド）

```
 1. import java.util.concurrent.ExecutorService;
 2. import java.util.concurrent.Executors;
 3.
 4. public class Sample {
 5.     public static void main(String[] args) throws Exception {
 6.         ExecutorService exec = Executors.newCachedThreadPool();
 7.         Runnable test = () -> {
 8.             System.out.println(Thread.currentThread().getId());
 9.         };
10.
11.         for (int i = 0; i < 5; i++) {
12.             exec.submit(test);
13.         }
14.
15.         Thread.sleep(1 * 10000);
16.         System.out.println("--------10秒後---------");
17.
18.         for (int i = 0; i < 5; i++) {
19.             exec.submit(test);
20.         }
21.
22.         Thread.sleep(1 * 70000);
23.         System.out.println("--------70秒後---------");
24.
25.         for (int i = 0; i < 5; i++) {
26.             exec.submit(test);
27.         }
28.     }
29. }
```

このコードでは、最初にsubmitメソッドでタスクを5回実行したあと、Threadクラスのsleepメソッドを使ってmainメソッドの処理を10秒間止め、そのあとに再びsubmitメソッドでタスクを5回実行しています。10秒しか経っていないので、スレッドは使い回しされることになります。その後、さらにThreadクラスの**sleepメソッド**を使って、mainメソッドの処理を70秒間止めています。今度は60秒以上経っているため、スレッドプール内のスレッドが破棄されて、新しいスレッドが生成されていることがわかります。

例 実行結果

```
> java Sample
13
12
14
15
12
--------10秒後---------
15
14
12
13
15
--------70秒後---------
17
17
18
19
20
```

試験対策

Executorフレームワークは、スレッドプールを使って効率的に並行処理を実現するためのインタフェースとクラス群です。Executorsクラスを使い、Executorのサブインタフェースである ExecutorServiceもしくはScheduledExecutorServiceを実装し、インスタンスを取得します。

4. A

➡ P97

ScheduledExecutorServiceインタフェースに関する問題です。

java.util.concurrent.ScheduledExecutorServiceは、java.util.concurrent.ExecutorServiceを拡張したインタフェースで、メソッドを定期的に実行するためのものです。ExecutorServiceでは、submitメソッドで渡された処理をすぐに実行してしまうため、タイミングを制御することはできません。処理を実行するタイミングの制御や定期的に処理を実行したい場合に、このインタフェースを使用します。

まずは、遅延実行する方法から学んでいきましょう。**遅延実行**には、ScheduledExecutorServiceの**scheduleメソッド**を使います。このメソッドは3つの引数を受け取ります。第1引数には「java.util.Runnable型の実行したい処理」を、第2引数には「long型の遅延させる時間」を、第3引数には「遅延させる時間の単位」を指定します。第3引数の単位には、**java.util.concurrent.TimeUnit**という列挙型を使います。この列挙型には、24時間を表すDAYS、1時間を表すHOURS、1分を表すMINUTES、1秒を表すSECONDSなどの列挙定数が定義されており、第2引数の数値と組み合わせて、遅延させる時間を表現します。たとえば、第2引数が10、第3引数がTimeUnit.SECONDSであれば、10秒遅延するという意味になります。

次のコードが、ScheduledExecutorServiceの使用例です。ScheduledExecutorServiceのインスタンス生成には、ExecutorServiceと同様にExecutorsクラスのファクトリメソッドを使用します。この例では、Executorsクラスの**newSingleThreadScheduledExecutorメソッド**を使って、mainメソッドとは別に新しいスレッドを1つだけ作って、プールしたScheduledExecutorServiceを作っています。

例 ScheduledExecutorServiceインタフェースの使用例

```
1. import java.util.concurrent.Executors;
2. import java.util.concurrent.ScheduledExecutorService;
3. import java.util.concurrent.TimeUnit;
4.
5. public class Sample {
6.     public static void main(String[] args) throws Exception {
7.         ScheduledExecutorService exec
8.             = Executors.newSingleThreadScheduledExecutor();
9.         exec.schedule(() -> {
10.            System.out.println("finish");
11.            exec.shutdown();    // スレッドプールを閉じる
12.        }, 1, TimeUnit.SECONDS);
13.
14.        int count = 0;
15.        while (true) {
16.            Thread.sleep(100);
17.            if (exec.isShutdown()) {
18.                break;
19.            }
20.            System.out.println((++count) * 100 + " ms");
21.        }
22.    }
23. }
```

遅延実行させる処理は、ScheduledExecutorServiceのscheduleメソッドで指定します。この例では、コンソールに「finish」と表示し、スレッドプールを終了させるラムダ式を1秒遅延させて実行するよう指定しています。
その後、mainメソッドを実行しているスレッドでは、while文を使って、100ミリ秒の一時停止後、コンソールに100ミリ秒単位で何ミリ秒経過したかを表示しています。
このコードを実行すると、新しいスレッドが実行されるものの1秒間待機し、その間にwhile文が実行されて100、200、300とカウントアップしながら1秒経過したタイミングでコンソールに「finish」と表示されます。以上のことから、選択肢**A**が正解です。

例 実行結果

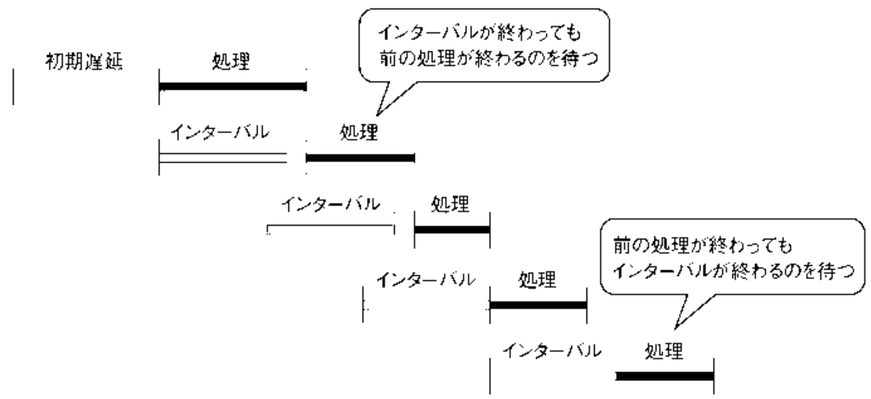
```
100 ms
200 ms
300 ms
… 省略 …
700 ms
800 ms
900 ms
finish
```

5. B

➡ P97

ScheduledExecutorServiceインタフェースのメソッドに関する問題です。
前問で解説したjava.util.concurrent.ScheduledExecutorServiceインタフェースのscheduleメソッドは、処理を1回だけ遅延実行するためのものです。ScheduledExecutorServiceが終了するまで定期的に繰り返し実行するには、**scheduleAtFixedRateメソッド**を使います。また、scheduleメソッドでは遅延時間だけを指定しますが、このメソッドでは初期の遅延時間（**初期遅延**）と2回目以降の**インターバル時間**の2つを指定します。なお、インターバルよりも処理の時間がかかった場合は、その終了を待って実行されます。つまり、前の処理が終わっていない段階では実行されないため、並行で実行されることはありません。初期遅延と処理、インターバルの関係性は次の図のとおりです。

【処理と初期遅延、インターバルの関係性（scheduleAtFixedRateメソッド）】



実際にscheduleAtFixedRateメソッドを使ったコード例は次のとおりです。このコードでは、1秒の初期遅延（第2引数）とインターバル（第3引数）を持ち、コンソールに「interrupt」と定期的に表示する処理（第1引数）を使っています。mainメソッドを実行するスレッドでは、100ミリ秒ごとに「>」をコンソールに横並びに表示します。

例 scheduleAtFixedRateメソッドの使用例

```
1. import java.util.concurrent.Executors;
2. import java.util.concurrent.ScheduledExecutorService;
3. import java.util.concurrent.TimeUnit;
4.
5. public class Sample {
6.     public static void main(String[] args) throws Exception {
7.         ScheduledExecutorService exec
8.             = Executors.newSingleThreadScheduledExecutor();
9.
10.        exec.scheduleAtFixedRate(() -> {
11.            System.out.println("interrupt");
12.        }, 1, 1, TimeUnit.SECONDS);
13.
14.        int count = 0;
15.        while (true) {
16.            Thread.sleep(100);
17.            System.out.print(">");
18.            count++;
19.            if (count == 100) {
20.                exec.shutdown();
21.                break;
22.            }
23.        }
24.    }
25. }
```

このコードを実行すると、次のような結果がコンソールに表示されます。mainメソッドのwhile文（無限ループ）では、int型の変数countが100になるまで繰り返し、100になったタイミングでScheduledExecutorServiceをシャットダウンしてループを抜けます。ScheduledExecutorServiceがシャットダウンすると定期的な処理の実行も終了するため、その後は「interrupt」と表示されることはありません。以上のことから、選択肢**B**が正解です。

例 実行結果

```
> java Sample
>>>>>>>>>>interrupt
>>>>>>>>>>interrupt
>>>>>>>>>>interrupt
>>>>>>>>>>interrupt
>>>>>>>>>>interrupt
>>>>>>>>>>interrupt
>>>>>>>>>>interrupt
>>>>>>>>>>interrupt
>>>>>>>>>>interrupt
>>>>>>>>>>interrupt
>>>>
```

scheduleAtFixedRateメソッドは、ScheduledExecutorServiceをシャットダウンする以外にも、戻り値であるjava.util.concurrent.ScheduledFutureインタフェースのcancelメソッドが呼ばれるか、もしくは実行中の処理で例外がスローされると、定期的な処理の実行を中断します。

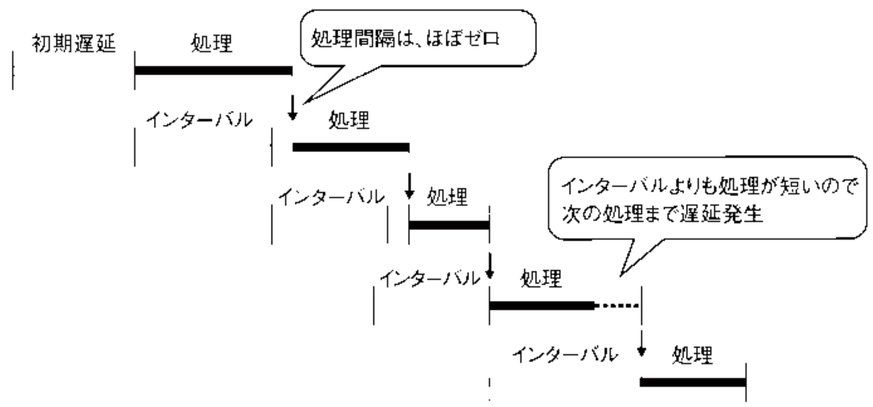
6. C

➡ P98

ScheduledExecutorServiceインタフェースのメソッドに関する問題です。

java.util.concurrent.ScheduledExecutorServiceのscheduleAtFixedRateメソッドによる定期実行では、前の処理と次の処理までのインターバルの両方が終了していれば、以降の処理が実行されます。そのため、次の図のようにインターバルが終了するまで次の処理が開始を待たなくてはいけない場合もあり、前の処理と次の処理との間隔が一定ではありません。

【初期遅延と処理、インターバルの関係性（scheduleAtFixedRateメソッド）】



このことを示したのが、次のコード例です。このコードは、解答5のSampleクラスを修正し、処理時間がランダムに変わるようにしたものです。**java.util.Randomクラス**の**nextIntメソッド**を使い、0〜10までの間でランダムな整数を生成し、その数に100ミリ秒を掛けた分だけ、スレッドを一時停止させます。

例 処理時間がランダムに変わるように変更したSampleクラス

```
1. import java.util.Random;
2. import java.util.concurrent.Executors;
3. import java.util.concurrent.ScheduledExecutorService;
4. import java.util.concurrent.TimeUnit;
5.
6. public class Sample {
7.     public static void main(String[] args) throws Exception {
8.         ScheduledExecutorService exec
9.             = Executors.newSingleThreadScheduledExecutor();
10.
11.        exec.scheduleAtFixedRate(() -> {
12.            int r = new Random().nextInt(10);
13.            System.out.print(r);
14.            try {
15.                Thread.sleep(r * 100);
16.            } catch (InterruptedException e) {
17.                System.out.println(e.getMessage());
18.            }
19.            System.out.println("interrupt");
20.        }, 1, 1, TimeUnit.SECONDS);
21.
22.        … 以下省略 …
23.    }
24. }
```

このコードを実行すると、次のように「interrupt」と表示してから次の乱数が表示されるまでの間が一定でないことがわかります。

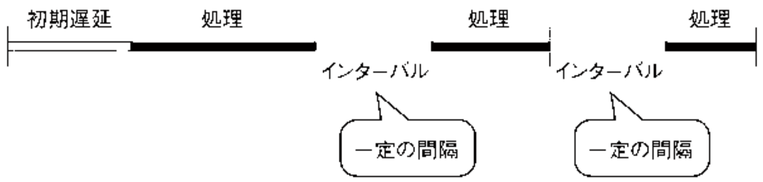
例 実行結果

```
> java Sample
>>>>>>>>>3>>>interrupt
>>>>>>>2>>interupt
>>>>>>>>3>>>interrupt
>>>>>>>9>>>>>>>>interrupt
>6>>>>>>interrupt
>>>>8>>>>>>>>interrupt
>>5>>>>>interupt
>>>>>7>>>>>>interrupt
>>>7>>>>>>>interrupt
>>>7>>>interrupt
```

もし、処理時間の長さに関係なく、インターバルを一定にしたい場合には、java.util.concurrent.ScheduledExecutorServiceインタフェースの**scheduleWithFixedDelayメソッド**を使います。よって、選択肢**C**が正解です。このメソッドは、前問で学んだscheduleAtFixedRateメソッドと同じように、

繰り返し実行したい処理と初期遅延、インターバルを指定します。

【初期遅延と処理、インターバルの関係性(scheduleWithFixedDelayメソッド)】



このscheduleWithFixedDelayメソッドは、引数としてRunnable型の処理、初期遅延、インターバル、初期遅延とインターバルの時間の単位の4つを受け取ります。前述のコードで、呼び出しているメソッドを修正すると次のようになります。

例 scheduleWithFixedDelayメソッドの使用例(Sampleクラス抜粋)

```
11.         exec.scheduleWithFixedDelay(() -> {
12.             int r = new Random().nextInt(10);
13.             System.out.print(r);
14.             try {
15.                 Thread.sleep(r * 100);
16.             } catch (InterruptedException e) {
17.                 System.out.println(e.getMessage());
18.             }
19.             System.out.println("interrupt");
20.         }, 1, 1, TimeUnit.SECONDS);
```

このコードを実行すると、次のように処理時間に関係なく、常にほぼ一定のインターバルを保ってから処理が実行されていることがわかります。

例 実行結果

```
> java Sample
>>>>>>>>>3>>>interrupt
>>>>>>>>>>4>>>>interrupt
>>>>>>>>>>8>>>>>>>interrupt
>>>>>>>>>>0interrupt
>>>>>>>>>>1>interrupt
>>>>>>>>>>2>>interrupt
>>>>>>>>>>7>>>>>>interrupt
>>>>>>>>
```

試験対策

解答5で解説したscheduleAtFixedRateメソッドは、インターバルが終了したら次の処理を実行します。次の処理との間隔は一定ではありません。scheduleWithFixedRateメソッドは、処理時間の長さに関係なく一定のインターバルを取ります。

参考

scheduleWithFixedDelayメソッドは、解答5で学んだscheduleAtFixedRateメソッドと同様に、ScheduledExecutorServiceをシャットダウンする以外にも、戻り値であるjava.util.concurrent.ScheduledFutureインタフェースのcancelメソッドが呼ばれるか、もしくは実行中の処理で例外がスローされても定期的な処理の実行を中断します。

7. D

➡ P98

Executorsクラスに関する問題です。
前問までで、java.util.concurrent.ScheduledExecutorServiceの3つのメソッドについて学びました。ここまでは、mainメソッド以外に1つだけスレッドを生成して繰り返し処理や遅延処理などを制御する方法を学びましたが、ScheduledExecutorServiceはExecutorServiceと同様にスレッドプールとして扱うことができます。あらかじめ複数のスレッドを生成して保持したScheduledExecutorServiceを取得するには、**java.util.concurrent.Executorsクラスの**newScheduledThreadPoolメソッドを使います。よって、選択肢Dが正解です。このメソッドの引数には、生成したいスレッド数を指定します。

次のコード例では、newScheduledThreadPoolメソッドで2つのスレッドを生成してプールし、そこにコンソールに「A」と表示するタスクと、「B」と表示する2つのタスクを一定間隔で実行するように設定しています。

例 newScheduledThreadPoolメソッドの使用例

```
1. import java.util.concurrent.Executors;
2. import java.util.concurrent.ScheduledExecutorService;
3. import java.util.concurrent.TimeUnit;
4.
5. public class Sample {
6.     public static void main(String[] args) throws Exception {
7.         ScheduledExecutorService exec
8.             = Executors.newScheduledThreadPool(2);
9.
```

```
10.         exec.scheduleWithFixedDelay(() -> {
11.             System.out.print("A");
12.         }, 1, 1, TimeUnit.SECONDS);
13.
14.         exec.scheduleWithFixedDelay(() -> {
15.             System.out.print("B");
16.         }, 1, 1, TimeUnit.SECONDS);
17.
18.         Thread.sleep(10000);
19.         exec.shutdown();
20.     }
21. }
```

このコードを実行すると、2つのスレッドが1秒ずつ遅延しながら次々と実行されることがわかります。

例 実行結果

```
> java Sample
ABABABABABABABABAB
```

8. D

➡ P98

Futureインタフェースに関する問題です。
Java SE 5より前のJavaのマルチスレッド・プログラミングでは、ThreadクラスとRunnableインタフェースを使っていました。この方法では、新しいスレッドを作って並行処理を実行すると、mainメソッドを実行しているスレッドからはそのスレッドがどのような結果だったのかを知ることができませんでした。
Java SE 5で導入された**java.util.concurrent.Futureインタフェース**を使うことにより、スレッドを生成したメソッドが、新しく作ったスレッドの結果を知ることができるようになります。結果がわかれば、結果に応じて別のスレッドを立ち上げたり、処理を継続しているスレッドを止めたりという、より細かな制御が可能になります。

Future型オブジェクトへの参照は、ExecutorServiceやScheduledExecutorServiceにタスクを登録した戻り値として受け取れます。次のコード例では、ExecutorServiceのsubmitメソッドの戻り値としてFuture型オブジェクトへの参照を受け取っています。

例 Futureインタフェースの使用例

```
1. import java.util.concurrent.ExecutorService;
2. import java.util.concurrent.Executors;
3. import java.util.concurrent.Future;
4.
5. public class Sample {
6.     public static void main(String[] args) throws Exception {
7.         ExecutorService exec
8.             = Executors.newSingleThreadExecutor();
9.         Future future = exec.submit(() -> {
10.             try {
11.                 System.out.println("start");
12.                 Thread.sleep(2000);
13.                 System.out.println("end");
14.             } catch (InterruptedException e) {
15.                 throw new RuntimeException(e);
16.             }
17.         });
18.
19.         if (future.get() == null) {
20.             System.out.println("finish");
21.         }
22.     }
23. }
```

Futureを使えば、新しいスレッドが開始されると、そのスレッドを外部から操作することができます。たとえば、スレッドの処理をキャンセルしたり、スレッドの処理結果を受け取ったりすることができます。スレッドの処理結果を受け取るには、コード例のように**getメソッド**を使います。
ただし、タスクを定義しているjava.lang.Runnableインタフェースのrunメソッドは、何も受け取らず、何も戻しません。また、何の例外もスローしません。そのため、getメソッドを使っても**null**が戻されます。もしnull以外の処理結果を戻したい場合には、解答10で解説するjava.util.concurrent.Callableインタフェースを使ってタスクを定義しなければいけません。以上のことから、選択肢**D**が正解です。

getメソッドは、スレッドの処理完了を待ち、処理が終われば結果を戻します。そのため、このコード例では新しいスレッドで「start」とコンソールに表示したあとに、2秒間、スレッドが一時停止し、そのあとに「end」と表示し、スレッドが終了します。getメソッドは、スレッドが終了するまで待機するため、コ

ンソールに「finish」と表示されるのは常に最後になります。

例 実行結果

```
> java Sample
start
end
finish
```

英語のFutureには、「未来」や「将来」など、いくつかの意味がありますが、ここでは「見込み」という意味です。メソッドの処理結果の見込みを扱うオブジェクトだと考えるとよいでしょう。

9. B

➡ P99

Futureインタフェースに関する問題です。
戻り値を戻せないjava.lang.Runnable型のタスクを使った場合、解答8で学習したjava.util.concurrent.Futureインタフェースのgetメソッドは、スレッドの処理が終了すればnullを戻します。
もし、**null以外の値**を戻したい場合には、オーバーロードされた**submitメソッド**の第2引数に戻り値を指定します。なお、**戻り値の型はFutureの型パラメータ**によって決まります。次のコード例では、String型の戻り値を戻すFuture型変数futureを宣言しているため、submitメソッドの第2引数に文字列「finish」を指定しています。

例 Futureインタフェースの使用例

```
1. import java.util.concurrent.ExecutorService;
2. import java.util.concurrent.Executors;
3. import java.util.concurrent.Future;
4.
5. public class Sample {
6.     public static void main(String[] args) throws Exception {
7.         ExecutorService exec
8.             = Executors.newSingleThreadExecutor();
9.         Future<String> future = exec.submit(() -> {
10.            try {
11.                System.out.println("start");
12.                Thread.sleep(2000);
13.                System.out.println("end");
14.            } catch (InterruptedException e) {
15.                throw new RuntimeException(e);
16.            }
17.        }, "finish");
18.
19.        String result = future.get();
20.        System.out.println(result);
21.    }
22. }
```

このように固定の戻り値を指定すると、getメソッドはスレッドの処理終了を待って、この固定値を戻します。そのため、このコード例を実行すると、コンソールに「finish」と表示されます。

例 実行結果

```
> java Sample
start
end
finish
```

設問のコードでは、submitメソッドの第2引数にこのメソッドが終わったときの値として0を指定しているため、戻り値は0となります。以上のことから、選択肢**B**が正解です。

10. A、D

➡ P99

Callableインタフェースに関する問題です。
解答8と解答9で学習したjava.util.concurrent.Futureインタフェースのgetメソッドは、java.lang.Runnable型のタスクを使っているため、nullもしくは固定の値しか戻せません。また、検査例外をスローすることもできません。
Java SE 5で導入された**java.util.concurrent.Callable**は、処理結果を戻したり、必要に応じて例外をスローしたりするマルチスレッドのタスクを定義するためのインタフェースです。
Callableインタフェースには、**callメソッド**が1つだけ定義されています。このメソッドのシグニチャは次のとおりで、戻り値型はCallableのインスタンスを取得する際に指定する型パラメータとなります。

```
V call() throws Exception
```

次のコードが、Callableを使った例です。型パラメータにBooleanを指定したCallable型の変数taskを宣言し、匿名クラスのインスタンスを生成しています。この匿名クラスではcallメソッドを実装していますが、**戻り値の型は型パラメータで受け取ったBoolean型**となっている点に注意してください。
このcallメソッドでは、**java.util.Randomクラス**を使ってint型の乱数を発生させ、その値が偶数かどうかを調べて結果の真偽値を戻しています。次にBoolean型の結果を扱うFutureのリストを作り、ExecutorServiceのsubmitメソッドで10個のタスクを登録して実行させて、その結果をリストに追加しています。

例 Callableインタフェースの使用例

```
 1. import java.util.ArrayList;
 2. import java.util.List;
 3. import java.util.Random;
 4. import java.util.concurrent.Callable;
 5. import java.util.concurrent.ExecutorService;
 6. import java.util.concurrent.Executors;
 7. import java.util.concurrent.Future;
 8.
 9. public class Sample {
10.     public static void main(String[] args) throws Exception {
11.         ExecutorService exec
12.             = Executors.newSingleThreadExecutor();
13.
14.         Callable<Boolean> task = new Callable<Boolean>() {
15.             @Override
16.             public Boolean call() throws Exception {
17.                 return new Random().nextInt() % 2 == 0;
18.             }
19.         };
20.
21.         List<Future<Boolean>> futures = new ArrayList<>();
22.         for (int i = 0; i < 10; i++) {
23.             futures.add(exec.submit(task));
24.         }
25.
26.         int total = 0;
27.         for (Future<Boolean> future : futures) {
28.             Boolean result = future.get();
29.             System.out.println(result);
30.             if (result) {
31.                 total++;
32.             }
33.         }
34.         System.out.println(total);
35.     }
36. }
```

最後に、拡張for文を使ってリストから順番に結果を取り出して、Futureの**getメソッド**からスレッドの処理結果であるBoolean型の戻り値を受け取っています。コンソールには、10個のFutureが持っていた結果が一覧で表示されたあと、trueの合計数が表示されます。

例 実行結果

```
> java Sample
false
false
true
true
false
true
false
false
true
true
false
false
4
```

なお、Callableインタフェースは、callメソッドが1つだけ定義された関数型インタフェースです。そのため、次のコードのようにラムダ式で記述することができ、全体のコードが短く簡潔になります。ただ、一見しただけでは、RunnableなのかCallableなのかの区別が付きにくく感じるかもしれません。そういうときは「Runnableは戻り値を戻せない」ということを思い出し、ラムダ式内で戻り値を戻しているかどうかで、RunnableかCallableかを見分けるとよいでしょう。よって、選択肢**A**と**D**が正解です。

例 ラムダ式に置き換えたコード

```
1. import java.util.ArrayList;
2. import java.util.List;
```

```
 3. import java.util.Random;
 4. import java.util.concurrent.ExecutorService;
 5. import java.util.concurrent.Executors;
 6. import java.util.concurrent.Future;
 7.
 8. public class Sample {
 9.     public static void main(String[] args) throws Exception {
10.         ExecutorService exec
11.                 = Executors.newSingleThreadExecutor();
12.         List<Future<Boolean>> futures = new ArrayList<>();
13.
14.         for (int i = 0; i < 10; i++) {
15.             futures.add(exec.submit(() -> {
16.                 return new Random().nextInt() % 2 == 0;
17.             }));
18.         }
19.
20.         int total = 0;
21.         for (Future<Boolean> future : futures) {
22.             Boolean result = future.get();
23.             System.out.println(result);
24.             if (result) {
25.                 total++;
26.             }
27.         }
28.         System.out.println(total);
29.     }
30. }
```

試験対策

「Runnableは戻り値を戻せない」ことを覚えておきましょう。ラムダ式を使っているのがRunnableなのかCallableなのかを見分けるのに役立ちます。

11. B

➡ P99

解答10では、java.util.concurrent.Callableインタフェースを使い、mainメソッドから実行した別スレッドの処理結果を受け取る方法を学びました。ここでは、別スレッドで発生した例外を受け取る方法について学びます。

次のコード例は、前問の最後でラムダ式に置き換えた例を修正し、発生させた乱数が偶数でなければ例外をスローするようにしています。このように例外をスローさせても、呼び出し元のスレッド（ここではmainメソッドのスレッド）にすぐに例外がスローされるわけではありません。スローされた例外は、結果であるFutureがいったん受け取ります。

例 別スレッドで発生した例外を受け取り、例外処理する

```
 1. import java.util.ArrayList;
 2. import java.util.List;
 3. import java.util.Random;
 4. import java.util.concurrent.ExecutionException;
 5. import java.util.concurrent.ExecutorService;
 6. import java.util.concurrent.Executors;
 7. import java.util.concurrent.Future;
 8.
 9. public class Sample {
10.     public static void main(String[] args)
11.                             throws InterruptedException {
12.
13.         ExecutorService exec
14.             = Executors.newSingleThreadExecutor();
15.         List<Future<Boolean>> futures = new ArrayList<>();
16.
17.         for (int i = 0; i < 10; i++) {
18.             futures.add(exec.submit(() -> {
19.                 if (new Random().nextInt() % 2 != 0) {
20.                     throw new Exception("Odd Number");
21.                 }
22.                 return true;
23.             }));
24.         }
25.
26.         int total = 0;
27.         for (Future<Boolean> future : futures) {
28.             Boolean result = false;
29.             try {
30.                 result = future.get();
31.                 System.out.println(result);
32.             } catch (ExecutionException e) {
33.                 System.out.println(e.getMessage());
```

```
34.             }
35.             if (result) {
36.                 total++;
37.             }
38.         }
39.         System.out.println(total);
40.     }
41. }
```

呼び出したスレッドで例外が発生した場合、Futureの**getメソッド**はjava.util.concurrent.ExecutionExceptionをスローします。よって、選択肢**B**が正解です。この例外クラスは検査例外であるため、getメソッドを呼び出したタイミングで例外処理をしなければいけません。なお、スレッド内で発生した例外は**ExecutionExceptionの内部**に保持されているため、Throwableから引き継いだ**getCauseメソッド**を使えば取り出すことが可能です。このコード例では、例外のメッセージを表示しているだけですので、実行すると次のようにコンソールに表示されます。

例 実行結果

```
java.lang.Exception: Odd Number
java.lang.Exception: Odd Number
true
java.lang.Exception: Odd Number
java.lang.Exception: Odd Number
true
java.lang.Exception: Odd Number
java.lang.Exception: Odd Number
true
java.lang.Exception: Odd Number
3
```

12. A

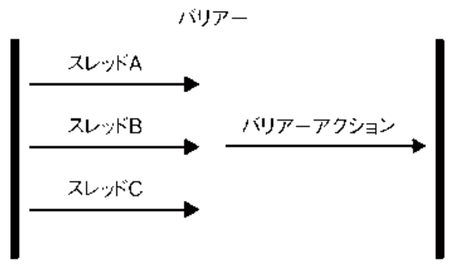
➡ P100

CyclicBarrierクラスに関する問題です。

java.util.concurrent.ExecutorServiceインタフェースやjava.util.concurrent.ScheduledExecutorServiceインタフェースを利用すれば、マルチスレッドによる並行処理が簡単に実現できます。ここでは、複数のスレッドの同期化処理について学びます。**同期化処理**とは、複数のスレッドが並行して実行されているときに、処理の順番を制御することです。

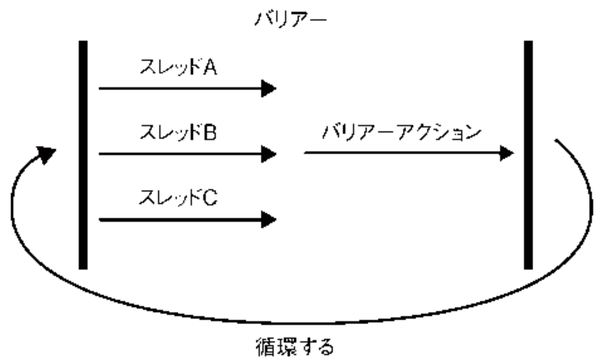
java.util.concurrent.CyclicBarrierは、複数のスレッドが特定のポイントまで到達するのを待機できるようにする同期化支援機能を提供するクラスです。複数スレッドが待ち合わせる待機ポイントのことを「**バリアー**」と呼びます。このバリアーにすべてのスレッドが到達すると、「**バリアーアクション**」と呼ばれる処理を実行するのが、このクラスの役割です。

【バリアーとバリアーアクション】



この仕組みは、1つのCyclicBarrierインスタンスによって複数回実行できます。繰り返し実行できることから、Cyclic（循環式）という名前が付いています。

【CyclicBarrierのイメージ】



では、CyclicBarrierの使い方を学びましょう。CyclicBarrierクラスのインスタンスを生成するとき、同期を取るスレッドの数とバリアーアクションをコンストラクタで受け取ります。たとえば、次のように記述すれば、「3つのスレッドを同期させ、3つのスレッドの処理が終われば（第1引数）、BarrierActionを実行する（第2引数）」という意味になります。

```
CyclicBarrier barrier
    = new CyclicBarrier(3, new BarrierAction());
```

実際に動作するスレッド（ワーカースレッド）には、CyclicBarrierインスタン

スへの参照を持たせます。任意のタイミングでCyclicBarrierクラスの**awaitメソッド**を呼び出し、処理を中断します。ワーカースレッドの中断が、コンストラクタで設定したスレッド数に達すると、CyclicBarrierインスタンスはバリアーアクションを実行します。

では、コード例で確認しましょう。次のTaskクラスはRunnableを実現し、スレッドで実行する処理を定義しています。このタスクは、CyclicBarrier型のフィールドを持ち、コンストラクタでその参照を受け取ります。

例 CyclicBarrierクラスの使用例

```
 1. import java.util.Random;
 2. import java.util.concurrent.BrokenBarrierException;
 3. import java.util.concurrent.CyclicBarrier;
 4.
 5. public class Task implements Runnable {
 6.     private CyclicBarrier barrier;
 7.
 8.     public Task(CyclicBarrier barrier) {
 9.         super();
10.         this.barrier = barrier;
11.     }
12.
13.     @Override
14.     public void run() {
15.         long id = Thread.currentThread().getId();
16.         System.out.println("START:" + id);
17.
18.         int r = new Random().nextInt(10);
19.         try {
20.             Thread.sleep(r * 100);
21.         } catch (InterruptedException e) {
22.             throw new RuntimeException(e);
23.         }
24.
25.         System.out.println("END:" + id);
26.         try {
27.             this.barrier.await();
28.         } catch (InterruptedException
29.                     | BrokenBarrierException e) {
30.             throw new RuntimeException(e);
31.         }
32.     }
33. }
```

スレッドで実行する処理を記述している**runメソッド**では、「START:」という文字列とスレッドIDを表示したあと、100〜1000ミリ秒の間、100ミリ秒単位でランダムにスリープし、「END:」という文字列とスレッドIDを表示します。その後、CyclicBarrierのawaitメソッドを呼び出して、処理が終了したことをバリアーに通知します。

CyclicBarrierのインスタンス化とバリアーアクションを定義しているのが、次のmainメソッドを持っているSampleクラスです。ここでは、5つのスレッドを持つスレッドプールを作り、5つのスレッドを同期化するCyclicBarrierのインスタンスを生成しています。このとき、バリアーアクションとして、コンソールに「it's all done.」という文字列を表示するラムダ式を渡しています。

例 バリアーとバリアーアクションの定義

```
 1. import java.util.concurrent.CyclicBarrier;
 2. import java.util.concurrent.ExecutorService;
 3. import java.util.concurrent.Executors;
 4.
 5. public class Sample {
 6.     public static void main(String[] args) {
 7.         ExecutorService exec
 8.             = Executors.newFixedThreadPool(5);
 9.
10.         CyclicBarrier barrier = new CyclicBarrier(5, () -> {
11.             System.out.println("it's all done.");
12.         });
13.         for (int i = 0; i < 5; i++) {
14.             exec.submit(new Task(barrier));
15.         }
16.     }
17. }
```

このコードを実行すると次のようにコンソールに表示され、すべてのワーカースレッドが終了してから、バリアーアクションが実行されていることがわかります。

※次ページに続く

例 実行結果

```
> java Sample
START:13
START:17
START:15
START:14
START:16
END:17
END:15
END:14
END:16
END:13
it's all done.
```

以上のことから、選択肢**A**が正解です。

13. A ➡P100

解答12では、CyclicBarrierで複数のスレッドの終了を待ち、次の処理を実行する同期化の方法を学びました。ここでは、複数のスレッドで1つのインスタンスを共有することで発生する「競合」と、それを解決するための「排他制御」の方法について学びます。

マルチスレッド・アプリケーションでは、複数のスレッドで1つのインスタンスを共有するために発生する「競合」に気を付けなければいけません。**競合**とは、あるスレッドが共有されているインスタンスのフィールドの値を読み出してから変更するまでの間に、別のスレッドが変更してしまうことです。マルチスレッド・アプリケーションでは、どのスレッドのどの処理が、いつ実行されるかはプログラム側で制御することができません。これが、競合という問題が発生する原因です。

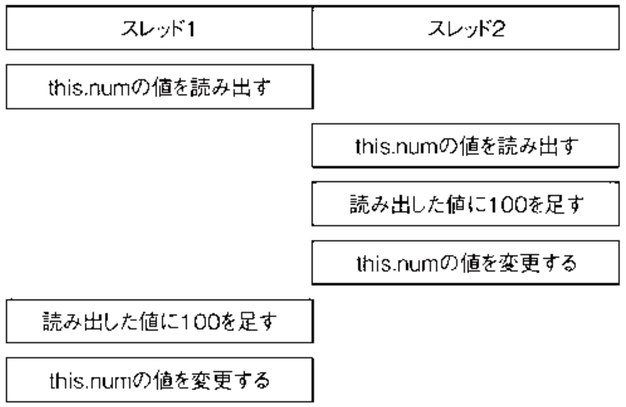
では、実際のサンプルで競合を確認してみましょう。次のコードは、1つのインスタンスを共有している2つのスレッドから、フィールドnumの値を変更している例です。予想では、初期値100の変数numに対して、2つのスレッドが100を足しているため、結果は300となるはずです。

例 競合するプログラム（RaceConditionSampleクラス）

```
1. import java.util.concurrent.ExecutorService;
2. import java.util.concurrent.Executors;
3.
4. public class RaceConditionSample implements Runnable {
5.     private int num = 100;
6.
7.     public static void main(String[] args) {
8.         RaceConditionSample s =
9.                 new RaceConditionSample();
10.        ExecutorService exec =
11.                Executors.newFixedThreadPool(2);
12.        exec.submit(s);
13.        exec.submit(s);
14.
15.        try {
16.            Thread.sleep(200);
17.        } catch (InterruptedException e) {
18.            throw new RuntimeException(e);
19.        }
20.
21.        System.out.println(s.num);
22.        exec.shutdown();
23.    }
24.
25.    public void add(int amount) {
26.        this.num = this.num + amount;
27.    }
28.
29.    @Override
30.    public void run() {
31.        add(100);
32.    }
33. }
```

しかし、このコードを実行すると、何回かに一度は200という結果が表示されます。筆者の環境では、10回中2〜3回の割合で「200」と表示されました。これは、2つのスレッドが1つのインスタンスを共有し、次ページに示した図のような順番でaddメソッド内の処理が実行されているからです。

※次ページに続く



このような問題を発生させないためには、あるスレッドが処理している間は、別のスレッドがその処理を実行できないよう待機させる必要があります。このように待機させることで意図しない動作が起きないようにすることを「**排他制御**」と呼びます。
排他制御をするには、**synchronizedキーワード**を使います。このキーワードは、メソッド宣言に使う場合と、メソッド内の一部の処理だけを対象に使う場合の2通りの使い方があります。

synchronizedをメソッド宣言に使用すると、そのメソッドは「**synchronizedメソッド**」と呼ばれます。このメソッドは、1つのスレッドがあるインスタンスのメソッドを実行している間は、ほかのスレッドと共有しているインスタンスの同じメソッドの実行を待機させることができます。この方法は安全である一方で、メソッド内の処理が複雑で時間がかかるものであるほど、待機する時間が長くなるため、システム全体のパフォーマンスに影響を及ぼす可能性があります。そもそも、マルチスレッドはパフォーマンス向上を目的として利用されることが多いため、排他制御によってパフォーマンスが劣化するようでは、利用する意味がなくなってしまいます。
そこで、メソッド全体ではなく、メソッド内の一部の処理だけを排他制御することで、パフォーマンスの劣化を最小限にすることができます。これが「**synchronizedブロック**」です。synchronizedブロックは、どのインスタンスのメソッドに対して排他制御するのかをブロックの引数として指定します。たとえば、次のようにsynchronizedブロックでthisを指定すると、メソッド全体に対して排他制御することになるため、synchronizedメソッドと同じ意味を持ちます。

```
void someMethod() {
    synchronized (this) {
        doSomething();
    }
}
```


前述のコード例のaddメソッドをsynchronizedで修飾すると競合は発生せず、何回実行しても300と表示されるようになります。

例 競合しないプログラム（RaceConditionSampleクラス抜粋）

```
25.     public synchronized void add(int amount) {
26.         this.num = this.num + amount;
27.     }
```

以上のことから、選択肢**A**が正解です。
選択肢Bの**volatile**もマルチスレッド・アプリケーションで使用する修飾子の1つです。マルチスレッド・アプリケーションでは、パフォーマンス向上を目的として、それぞれのスレッドが共有しているインスタンスのフィールド値を個別にキャッシュすることがあります。各スレッドが元のメモリを毎回読み書きする手間を省くことで、性能の向上を図っているわけです。ただし、フィールドの値が変更されたときに別のスレッドでキャッシュしている値も同時に変わるわけではありません。このような不整合が発生するときに、volatileでフィールドを修飾すると、キャッシュされずに安全に扱えるようになります。その代わりにパフォーマンスは落ちますので、変更が加わる可能性があるかどうかで、修飾する／しないを見極める必要があります。

選択肢Cの**strictfp**は、浮動小数点数をIEEE 754という規格で決められたルールで厳密に処理するためのキーワードです。strictfp修飾子を指定しない場合、JavaはCPUの機能を使って浮動小数点数を演算します。この場合、プログラムは高速に動作しますが、環境によって演算結果にわずかな誤差が生じる可能性があります。古いハードウェアで動作させている場合などは必要になることがありますが、現在はこの修飾子がなくてもIEEE 754に準拠した処理をするようになっています。

選択肢Dの**transient**は、オブジェクトをシリアライズ（第5章解答10を参照）するときに、シリアライズしないフィールドを示すために付ける修飾子です。
以上のことから、選択肢B、C、Dは誤りです。

試験対策

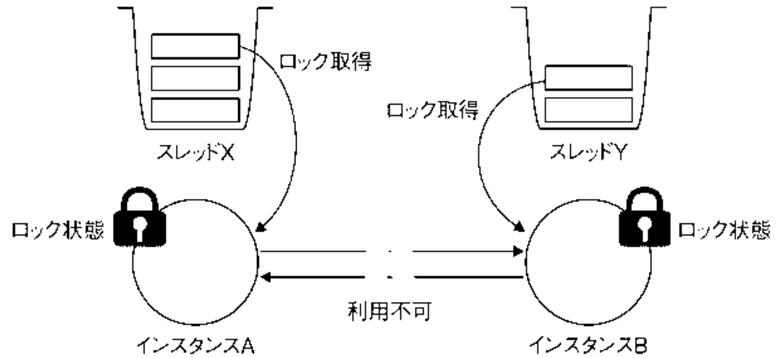
synchronizedキーワードは、メソッド宣言に使う場合（synchronizedメソッド）と、メソッド内の一部の処理（synchronizedブロック）を対象に使う場合の2通りがあります。synchronizedブロックでは、キーワードを付ける位置に注意しましょう。

14. B

➡ P100

解答13では、競合を避けるためにsynchronizedを使った排他制御について学びました。ここでは、排他制御によって発生する「デッドロック」について学習しましょう。**デッドロック**は、複数のスレッドの間で、排他制御された複数のインスタンスを共有していて、それぞれのインスタンス同士が連携することで発生します。
たとえば、次の図のようにスレッドXとYから利用されているインスタンスAとBがあったとき、AとBが相互に利用する関係があり、互いに利用する際に排他制御してロックしてしまうと、デッドロックが発生します。

【デッドロック】



実際のコード例で、デッドロックを確認してみましょう。次のコードは、銀行口座を表すクラスで、預金残高をフィールド（balance）として持ちます。コンストラクタで預金残高の初期値を受け取ったあとは、depositメソッドで入金、withdrawalsメソッドで出金を行い、預金残高を増減させます。なお、2つのメソッドは実行に時間がかかる処理になっていることを再現するために、スレッドを100ミリ秒一時停止しています。

例 BankAccountクラス

```
1. public class BankAccount {
2.     private int balance;
3.     public BankAccount(int balance) {
4.         this.balance = balance;
5.     }
6.     public void deposit(int amount) {
7.         try {
8.             Thread.sleep(100);
9.         } catch (InterruptedException e) {
10.            throw new RuntimeException(e);
11.        }
12.        this.balance += amount;
13.    }
14.    public void withdrawals(int amount) {
15.        try {
16.            Thread.sleep(100);
17.        } catch (InterruptedException e) {
18.            throw new RuntimeException(e);
19.        }
20.        this.balance -= amount;
21.    }
22.    public int getBalance() {
23.        return balance;
24.    }
25. }
```

この銀行口座を表すBankAccountクラスのインスタンスを2つ生成し、デッドロックを発生させるのが次のコード例です。この例では2つのスレッドを生成し、それぞれ別のタスクを与えています。

例 デッドロックを発生させるBankSampleクラス

```
1. import java.util.concurrent.ExecutorService;
2. import java.util.concurrent.Executors;
3.
4. public class BankSample {
5.     public static void main(String[] args) throws Exception {
6.         BankAccount a = new BankAccount(100);
7.         BankAccount b = new BankAccount(100);
8.
9.         ExecutorService exec
10.            = Executors.newFixedThreadPool(2);
11.        exec.submit(() -> {
12.            synchronized (a) {
13.                a.withdrawals(20);
14.                synchronized (b) {
15.                    b.deposit(20);
16.                }
17.            }
```

```
18.         });
19.
20.         exec.submit(() -> {
21.             synchronized (b) {
22.                 b.withdrawals(30);
23.                 synchronized (a) {
24.                     a.deposit(30);
25.                 }
26.             }
27.         });
28.         // do something
29.     }
30. }
```

このとき、2つのスレッドでロックする順番が異なっていることに注意してください。1つ目のタスクでは、**synchronizedブロック**を使って変数aで参照できるインスタンスのロックを取得してから出金し、その後、変数bで参照できるインスタンスのロックを取得して入金しています。こうして、口座から出金して、別の口座に入金するという振り込み処理を実施している間、別のスレッドが割り込めないようにしているのです。

一方、2つ目のタスクでは、逆に変数bで参照できるインスタンスのロックを取得してから出金し、変数aで参照できるインスタンスのロックを取得して入金しようとしています。
1つ目のタスクがaをロックした状態で、bにアクセスしようとしていますが、2つ目のタスクがbをロックしています。さらに2つ目のタスクがaにアクセスしようとしますが、1つ目のタスクがaをロックしています。そのため、どちらのスレッドも処理を進めることができなくなります。このような状態を「デッドロック」と呼びます。

こうしたデッドロックを防ぐために、ロックする順番をスレッド間で揃えるという方法があります。たとえば、次のように2つ目のタスクがロックする順番をb、aから、逆順のa、bに変更すると、デッドロックは発生しなくなります。

例 ロックの順番を変更したBankSampleクラス

```
1. import java.util.concurrent.ExecutorService;
2. import java.util.concurrent.Executors;
3.
4. public class BankSample {
5.     public static void main(String[] args) throws Exception {
6.         BankAccount a = new BankAccount(100);
7.         BankAccount b = new BankAccount(100);
8.
9.         ExecutorService exec
10.             = Executors.newFixedThreadPool(2);
11.         exec.submit(() -> {
12.             synchronized (a) {
13.                 a.withdrawals(20);
14.                 synchronized (b) {
15.                     b.deposit(20);
16.                 }
17.             }
18.         });
19.
20.         exec.submit(() -> {
21.             synchronized (a) {
22.                 a.deposit(30);
23.                 synchronized (b) {
24.                     b.withdrawals(30);
25.                 }
26.             }
27.         });
28.         // do something
29.     }
30. }
```

以上のことから、選択肢**B**が正解です。
なお、選択肢Dの**ライブロック**はデッドロックと似たような結果になりますが、2つのスレッドがデッドロックを避け合うことで、結果としてロック状態に陥ることを指します。ライブロックは、道を行き交う2人が互いを避けようとして、一人が右に避けたら、もう一人も同じ方向に、気を取り直して反対に避けようとしたら、相手も同じことをしてしまって進めなくなるという喩えで説明される状態を示します。よって、選択肢Dは誤りです。

デッドロックを回避するには、ロックする順番をスレッド間で揃えるという方法があります。

15. C

➡ P100

原子性を担保するクラスに関する問題です。
synchronizedを使った排他制御は強力であると同時に、ほかのスレッドを待機させるため、パフォーマンスに影響を及ぼしやすい傾向があります。単純なコードであれば、synchronizedの代わりに原子性が担保された**java.util.concurrent.atomicパッケージ**のクラスを利用することができます。

【java.util.concurrent.atomicパッケージの主なクラス】

クラス	説明
AtomicBoolean	アトミック変数としてboolean型の値を扱うためのクラス
AtomicInteger	アトミック変数としてint型の値を扱うためのクラス
AtomicLong	アトミック変数としてlong型の値を扱うためのクラス
AtomicReference	アトミック変数として参照を扱うためのクラス

原子性とは、分離できない（してはいけない）性質のことで、一連の処理が完全に終わるか、まったく実行されないかどちらかになるよう結果が保証されなければいけない性質を表します。マルチスレッド・プログラミングでは、あるスレッドが変数の値を処理している間に、別のスレッドがその変数を書き換えてしまう可能性があるため、変数単位で排他制御できる仕組みが必要になります。java.util.concurrent.atomicパッケージに属するクラスは、このような仕組みを実現したクラスなのです。

まずは原子性が担保されないコードを見てみましょう。次のValueクラスは、int型のフィールドnumを持っており、値を追加するaddメソッドと取得するgetメソッドを持っています。

例 Valueクラス

```
1. public class Value {
2.     private int num = 0;
3.     public void add(int num) {
4.         this.num += num;
5.     }
6.     public int get() {
7.         return num;
8.     }
9. }
```

このValueクラスを受け取って処理するタスクが、次のTaskクラスです。

runメソッドがスレッドから呼ばれると、100をValueのインスタンスに追加します。

例 Taskクラス

```
1. public class Task implements Runnable {
2.     private Value val;
3.     public Task(Value val) {
4.         this.val = val;
5.     }
6.     @Override
7.     public void run() {
8.         this.add(100);
9.     }
10.    private void add(int num) {
11.        this.val.add(num);
12.    }
13. }
```

次のコード例が、マルチスレッドで動作させるためのSampleクラスです。このクラスでは、mainメソッドで生成した1つのValueインスタンスを2つのタスクで共有し、別々のスレッドで動作させています。

例 Sampleクラス

```
1. import java.util.concurrent.ExecutorService;
2. import java.util.concurrent.Executors;
3.
4. public class Sample {
5.     public static void main(String[] args) {
6.         Value val = new Value();
7.
8.         ExecutorService exec =
9.                 Executors.newFixedThreadPool(2);
10.        exec.submit(new Task(val));
11.        exec.submit(new Task(val));
12.
13.        try {
14.            Thread.sleep(200);
15.        } catch (InterruptedException e) {
16.            throw new RuntimeException(e);
```

```
17.         }
18.
19.         System.out.println(val.get());
20.         exec.shutdown();
21.     }
22. }
```

このコードを実行すると、多くの場合は「200」と表示されますが、Valueクラスが**synchronized**で排他制御されていないため、数回に1回程度「100」と表示されます。このように、単純に値を増減だけさせるのであれば、synchronizedでインスタンスを丸ごとロックしてしまうのではなく、原子性が担保された型を利用します。

次のAtomicValueクラスが、前述のValueクラスを継承し、int型のフィールドの代わりにjava.util.concurrent.atomic.AtomicInteger型を使ったフィールドに変更したものです。前述のコードで原子性が保たれていないのは、addメソッドで値の読み出しと変更を行い、そして書き込みをする前に別のスレッドが値を変更してしまうからでした。そこで、**AtomicIntegerクラス**の**addAndGetメソッド**を使い、読み出しから値の変更までの一連の動作中にほかのスレッドからの処理を受け付けないようにします。実際には、このメソッドは読み出しと変更を一度に実行するOSのネイティブコードを呼び出すことで、原子性を担保しています。

例 AtomicIntegerクラスの使用例

```
1. import java.util.concurrent.atomic.AtomicInteger;
2.
3. public class AtomicValue extends Value {
4.     private AtomicInteger num = new AtomicInteger(0);
5.     public void add(int num) {
6.         this.num.addAndGet(num);
7.     }
8.     public int get() {
9.         return this.num.intValue();
10.    }
11. }
```

Valueクラスを継承したサブクラスAtomicValueが定義できたら、mainメソッドを次のように修正しましょう。今度は何回実行しても「200」と表示されるようになるはずです。以上のことから、選択肢**C**が正解です。

例 原子性を担保するよう変更したSampleクラス

```
1. import java.util.concurrent.ExecutorService;
2. import java.util.concurrent.Executors;
3.
4. public class Sample {
5.     public static void main(String[] args) {
6.         Value val = new AtomicValue();
7.
8.         ExecutorService exec =
9.                 Executors.newFixedThreadPool(2);
10.        exec.submit(new Task(val));
11.        exec.submit(new Task(val));
12.
13.        try {
14.            Thread.sleep(200);
15.        } catch (InterruptedException e) {
16.            throw new RuntimeException(e);
17.        }
18.
19.        System.out.println(val.get());
20.        exec.shutdown();
21.    }
22. }
```

原子性を担保するjava.util.concurrent.atomicパッケージのクラスを覚えておきましょう。

16. D

➡ P101

マルチスレッド・アプリケーションで利用されるクラスは、スレッドセーフでなくてはいけません。**スレッドセーフ**なクラスとは、正しく排他制御され、複数のスレッドから同時に利用されても競合やデッドロック、ライブロックといった問題が起きないクラスのことです。ここでは、マルチスレッドで扱うスレッドセーフなコレクションについて学びましょう。

java.util.ArrayListクラスは、複数のインスタンスの参照をまとめて扱うことができる便利なクラスであり、シングルスレッド・アプリケーションでは最も頻繁に使うクラスの1つです。しかし、ArrayListはスレッドセーフではあり

ません。
このことを確認するために、次のようなコード例を見てみましょう。このコード例では、読み込みと書き出しをする2つのタスクを用意し、それぞれ別のスレッドで動作させます。
次のクラスは、コレクションの要素を拡張for文で繰り返し処理しながらコンソールに表示しています。

例 リストの要素の読み込みを行うReadTaskクラス

```
1. import java.util.List;
2.
3. public class ReadTask implements Runnable {
4.     private List<Integer> list;
5.
6.     public ReadTask(List<Integer> list) {
7.         this.list = list;
8.     }
9.
10.    @Override
11.    public void run() {
12.        try {
13.            for (Integer i : list) {
14.                try {
15.                    Thread.sleep(100);
16.                } catch (InterruptedException e) {
17.                    throw new RuntimeException(e);
18.                }
19.                System.out.println(i);
20.            }
21.        } catch (Exception e) {
22.            e.printStackTrace();
23.        }
24.    }
25. }
```

また、次のクラスはfor文を使ってコレクションに要素を追加しています。

例 リストに要素を追加するWriteTaskクラス

```
1. import java.util.List;
2.
3. public class WriteTask implements Runnable {
4.     private List<Integer> list;
5.
6.     public WriteTask(List<Integer> list) {
7.         this.list = list;
8.     }
9.
10.    @Override
11.    public void run() {
12.        for (int i = 10; i < 20; i++) {
13.            try {
14.                Thread.sleep(100);
15.            } catch (InterruptedException e) {
16.                throw new RuntimeException(e);
17.            }
18.            list.add(i);
19.        }
20.    }
21. }
```

これらのクラスのインスタンスは、1つのリストを共有します。なお、このリストには初期値として1、2、3という3つの数値を要素として追加しています。

例 2つのタスクを実行するSampleクラス

```
1. import java.util.ArrayList;
2. import java.util.List;
3. import java.util.concurrent.ExecutorService;
4. import java.util.concurrent.Executors;
5.
6. public class Sample {
7.     public static void main(String[] args) {
8.         List<Integer> list = new ArrayList<>();
9.         list.add(1);
10.        list.add(2);
11.        list.add(3);
12.
13.        ExecutorService exec
14.            = Executors.newFixedThreadPool(2);
15.        exec.submit(new WriteTask(list));
```

```
16.         exec.submit(new ReadTask(list));
17.
18.         try {
19.             Thread.sleep(5000);
20.         } catch (InterruptedException e) {
21.             throw new RuntimeException(e);
22.         }
23.         exec.shutdown();
24.     }
25. }
```

このコード例では、読み込みと書き出しを同時に行うことになります。ArrayListを使った拡張for文では、読み出しをしている際に追加や削除によってコレクション内の要素数を変更すると、**java.util.ConcurrentModificationException**がスローされます。そのため、このコードを実行すると、次のように例外が発生してしまいます。

例 実行結果

```
> java Sample
1
java.util.ConcurrentModificationException
    at java.base/java.util.ArrayList$Itr.checkForComodification(Arr
ayList.java:1013)
    at java.base/java.util.ArrayList$Itr.next(ArrayList.java:967)
    at exf.ReadTask.run(ReadTask.java:16)
    at java.base/java.util.concurrent.Executors$RunnableAdapter.cal
l(Executors.java:515)
    at java.base/java.util.concurrent.FutureTask.run(FutureTask.jav
a:264)
    at java.base/java.util.concurrent.ThreadPoolExecutor.runWorker
(ThreadPoolExecutor.java:1130)
    at java.base/java.util.concurrent.ThreadPoolExecutor$Worker.run
(ThreadPoolExecutor.java:630)
    at java.base/java.lang.Thread.run(Thread.java:832)
```

このような状態を避けるために、ArrayListの代わりに**java.util.concurrent.CopyOnWriteArrayListクラス**が用意されています。このクラスは、読み出しをする際にコピーを作ります。そのため、パフォーマンスは落ちますが、読み込みと書き出しを同時に行っても例外が発生することはありません。

例 CopyOnWriteArrayListクラスの使用例

```
1. import java.util.List;
2. import java.util.concurrent.CopyOnWriteArrayList;
3. import java.util.concurrent.ExecutorService;
4. import java.util.concurrent.Executors;
5.
6. public class Sample {
7.     public static void main(String[] args) {
8.         List<Integer> list
9.             = new CopyOnWriteArrayList<Integer>();
10.        list.add(1);
11.        list.add(2);
12.        list.add(3);
13.
14.        ExecutorService exec
15.            = Executors.newFixedThreadPool(2);
16.        exec.submit(new WriteTask(list));
17.        exec.submit(new ReadTask(list));
18.
19.        try {
20.            Thread.sleep(5000);
21.        } catch (InterruptedException e) {
22.            throw new RuntimeException(e);
23.        }
24.        exec.shutdown();
25.    }
26. }
```

このコードを実行すると、例外は発生せず、次のような結果がコンソールに表示されるはずです。ただし、コレクションの要素のコピーを取って読み出すため、追加された要素は表示されません。

例 実行結果

```
> java Sample
1
2
3
```

なお、「exec.shutdown();」でExecutorServiceをシャットダウンしたあとにコレ

クションの要素を読み出すと、次のように変更後の値が表示されます。

```
System.out.println("---");
list.forEach(System.out::println);
```

例 実行結果

```
> java Sample
1
2
3
---
1
2
3
10
11
12
13
14
15
16
17
18
19
```

以上のことから、選択肢**D**が正解です。
選択肢AのArrayListはスレッドセーフではありません。選択肢Bのjava.util.Vectorはすべてのメソッドがsynchronizedで修飾され、排他制御ができるためスレッドセーフなクラスではありますが、読み出しの最中に変更が加えられるとConcurrentModificationExceptionが発生します。よって、誤りです。選択肢Cのようなクラスは存在しません。

17. D

➡P101

synchronizedによる排他制御やjava.util.concurrent.atomicパッケージに属するクラスを利用した原子性の担保では、複数のメソッドにまたがるような処理の排他制御を実現できません。
たとえば、次のコード例のように3つのメソッドがあり、first、second、thirdの順にメソッドを呼び出さなければいけないとき、マルチスレッド環境では、その順番を保証できないのです。

例 Testクラス

```
1. public class Test {
2.     public void first() {
3.         System.out.println(
4.                 Thread.currentThread().getId()
5.                 + ":1");
6.     }
7.
8.     public void second() {
9.         System.out.println(
10.                Thread.currentThread().getId()
11.                + ":2");
12.    }
13.
14.    public void third() {
15.        System.out.println(
16.                Thread.currentThread().getId()
17.                + ":3");
18.    }
19. }
```

ここでは、まず順番が保証できない状況を確認してから、この問題を解決する方法について学びましょう。まず、先ほどのTestクラスを使ってfirst、second、thirdの順にメソッドを呼び出すRunnableインタフェースの実現クラスを用意します。

例 Taskクラス

```
1. public class Task implements Runnable {
2.     private Test test;
3.
4.     public Task(Test test) {
5.         super();
6.         this.test = test;
7.     }
8.
9.     @Override
10.    public void run() {
```

```
11.         test.first();
12.         test.second();
13.         test.third();
14.     }
15. }
```

スレッドで実行すべきタスクの準備ができたら、次のように3つのスレッドを作って保持するスレッドプールを用意し、タスクを登録して実行します。このとき、3つのスレッドが1つのTestインスタンスへの参照を共有している点に注意してください（9〜11行目）。

例 Sampleクラス

```
1. import java.util.concurrent.ExecutorService;
2. import java.util.concurrent.Executors;
3.
4. public class Sample {
5.     public static void main(String[] args) {
6.         Test t = new Test();
7.
8.         ExecutorService exec = Executors.newFixedThreadPool(3);
9.         exec.submit(new Task(t));
10.        exec.submit(new Task(t));
11.        exec.submit(new Task(t));
12.    }
13. }
```

このSampleクラスを実行すると、次のようにスレッドID 15がfirstを実行すると、次に14が、その次に13が、そして14がsecondを実行して…と、ランダムな順番で実行されていることがわかります。

例 実行結果

```
> java Sample
15:1
14:1
13:1
14:2
15:2
14:3
13:2
15:3
13:3
```

そこで、次のように共有されるTestクラスに、ロックを制御するためのReentrantLockオブジェクトを追加します。**java.util.concurrent.locks.ReentrantLockクラス**の**lockメソッド**は、そのメソッドを呼び出したスレッドが**unlockメソッド**を呼び出すまで、別のスレッドによるlockメソッドの呼び出しを待機させます。よって、選択肢**D**が正解です。

例 ReentrantLockクラスの使用例

```
1. import java.util.concurrent.locks.ReentrantLock;
2.
3. public class Test {
4.     private final ReentrantLock lock = new ReentrantLock();
5.
6.     public void lock() {
7.         this.lock.lock();
8.     }
9.
10.    public void unlock() {
11.        this.lock.unlock();
12.    }
13.     … 中略 …
14. }
```

共有されるTestクラスの修正をし終えたら、タスク側で順番にメソッドを呼び出す前後に、ロックの取得と解除をするよう修正を加えます。

例 Taskクラス（変更後）

```
1. public class Task implements Runnable {
2.     private Test test;
3.     … 中略 …
8.
9.     @Override
10.    public void run() {
11.        try {
12.            test.lock();
```

```
13.             test.first();
14.             test.second();
15.             test.third();
16.         } finally {
17.             test.unlock();
18.         }
19.     }
20. }
```

なお、**ロックと解除までの処理はtryブロックで囲み、解除はfinallyブロックに記述する**ようにしましょう。こうすることで、いずれかのスレッドに何らかの例外が発生してもロックは解除されるため、別のスレッドが処理を続行することができます。

以上の修正ができたら、Sampleクラスを再度実行してみましょう。今度は次の実行結果のように、スレッドごとにfirst、second、thirdの順にメソッドが実行されます。

例 実行結果

```
> java Sample
15:1
15:2
15:3
13:1
13:2
13:3
14:1
14:2
14:3
```


4

ストリームAPI

■java.util.Optionalクラス

■ストリームAPI

■ストリーム・パイプラインと中間操作、終端操作

■java.util.stream.BaseStreamとサブインタフェース

■並列ストリーム

■java.util.Collectionインタフェース

■リダクション操作

■java.util.stream.Collectorインタフェース

■java.util.stream.Collectorsクラス

1. 次のプログラムをコンパイル、実行したときの結果として、正しいものを選びなさい。(1つ選択)

```
1. import java.util.Optional;
2.
3. public class Ex1 {
4.     public static void main(String[] args) {
5.         Optional<String> sample = Optional.of(null);
6.         System.out.println(sample);
7.     }
8. }
```

A. 「null」と表示される
B. 「empty」と表示される
C. 「Optional.empty」と表示される
D. 実行時に例外がスローされる
E. コンパイルエラーが発生する

➡ P170

2. 次のプログラムをコンパイル、実行したときの結果として、正しいものを選びなさい。(1つ選択)

```
1. import java.util.Optional;
2.
3. public class OfTest {
4.     public static void main(String[] args) {
5.         Optional<String> sample = Optional.ofNullable(null);
6.         System.out.println(sample.get());
7.     }
8. }
```

A. 「null」と表示される
B. 「Optional.empty」と表示される
C. 実行時に例外がスローされる
D. コンパイルエラーが発生する

➡ P176

3. 次のプログラムをコンパイル、実行したときの結果として、正しいものを選びなさい。(1つ選択)

```
1. import java.util.Optional;
2.
3. public class Sample {
4.     public static void main(String[] args) {
5.         Optional<String> sample = Optional.of("A");
6.         System.out.println(sample.orElse("B"));
7.     }
8. }
```

A. 「A」と表示される
B. 「B」と表示される
C. 「Optional.empty」と表示される
D. コンパイルエラーが発生する
E. 実行時に例外がスローされる

➡ P178

4. Optionalクラスを利用している。Optionalのインスタンスが空であったときに任意の例外をスローするOptionalクラスのメソッドとして、正しいものを選びなさい。(1つ選択)

A. getメソッド
B. orElseGetメソッド
C. orElseThrowメソッド
D. getThrowメソッド

➡ P179

5. Optionalクラスを利用している。Optionalが何らかの結果を持っているとき、その値を取り出さずに処理するためのメソッドとして、正しいものを選びなさい。(1つ選択)

A. isPresentメソッド
B. ifPresentメソッド
C. presentメソッド
D. executeメソッド

➡ P180

6. OptionalクラスのifPresentOrElseTestメソッドに渡す引数の型の組み合わせとして、正しいものを選びなさい。（1つ選択）

A. Consumer, Runnable
B. Supplier, Runnable
C. Consumer, Function
D. Supplier, Function

➡ P181

7. 次のプログラムをコンパイル、実行したときの結果として、正しいものを選びなさい。（1つ選択）

```
1. import java.util.Optional;
2.
3. public class Ex7 {
4.     public static void main(String[] args) {
5.         Optional<String> sample = Optional.of("abcde");
6.         Optional<String> result = sample.map(
7.                                 str -> str.toUpperCase());
8.
9.         System.out.println(sample.get());
10.        System.out.println(result.get());
11.    }
12. }
```

A. 「abcde」「abcde」と表示される
B. 「abcde」「ABCDE」と表示される
C. 「ABCDE」「abcde」と表示される
D. 「ABCDE」「ABCDE」と表示される

➡ P182

8. 次のプログラムをコンパイル、実行したときの結果として、正しいものを選びなさい。（1つ選択）

```
1. import java.util.Optional;
2.
3. public class Test2 {
4.     public static void main(String[] args) {
5.         Optional<Integer> a = Optional.of(100);
6.         Optional<Integer> b = a.map(price -> calc(price, 3));
7.         System.out.println(b.get());
8.     }
9.
10.    private static Optional<Integer> calc(int price, int qty) {
11.        if (qty < 0) {
12.            return Optional.empty();
13.        }
14.        return Optional.of(price * qty);
15.    }
16. }
```

A. 「300」と表示される
B. 「empty」と表示される
C. コンパイルエラーが発生する
D. 実行時に例外がスローされる

➡ P184

9. 次のコードのうち、正しく記述されているものを選びなさい。（1つ選択）

A. `List.of(1,2,3).forEach(x -> System.out.println(x));`
B. `List.of(1,2,3).forEach(() -> System.out.println(x));`
C. `List.of(1,2,3).forEach(() -> return 100);`
D. `List.of(1,2,3).forEach((x) -> return x * 100);`

➡ P186

10. ストリームを利用する理由として、適切でないものを選びなさい。（1つ選択）

A. コレクションや配列の全要素を同じように変換するため
B. コレクションや配列の要素の合計や平均といった統計を取るため
C. コレクションや配列を扱うfor文などの繰り返し構文を置き換えるため
D. コレクションや配列の要素を何らかの条件でグルーピングするため
E. コレクションや配列の要素から条件に合ったデータを検索するため

➡ P188

11. int型要素を扱う配列型変数「array」からストリームを作るコードとして、正しいものを選びなさい。（1つ選択）

A. `Stream<Integer> stream = array.stream();`
B. `Stream<Integer> stream = Arrays.stream(array);`
C. `IntStream stream = Arrays.stream(array);`
D. `IntStream stream = array.stream();`

➡ P191

12. ストリームの処理順に関する説明として、正しいものを選びなさい。（1つ選択）

A. コレクションに追加した順で処理される
B. コレクションが管理する順で処理される
C. コレクションに追加した順とは逆順に処理される
D. 処理順は常に不定である

➡ P193

13. String型要素を扱うjava.util.List型変数「list」から並列ストリームを取得するコードとして、正しいものを選びなさい。（1つ選択）

A. `ParallelStream<String> stream = list.parallelStream();`
B. `ParallelStream<String> stream = list.getParallelStream();`
C. `Stream<String> stream = list.getParallelStream();`
D. `Stream<String> stream = list.parallelStream();`

➡ P194

14. 次のプログラムを実行して、【実行結果】に示した結果を得たい。「/* insert code here */」に入るコードとして正しいものを選びなさい。（1つ選択）

```
import java.util.Arrays;
import java.util.List;

public class Sample {
    public static void main(String[] args) {
        List<Integer> list = Arrays.asList(1,2,3,4,5,6,7,8,9,10);
        list.stream()
            ./* insert code here */
            .forEach(System.out::print);
    }
}
```

【実行結果】

```
678910
```

A. `filter(n -> n < 5)`
B. `filter(n -> n > 5)`
C. `map(n -> n > 5)`
D. `map(n -> n < 5)`

➡ P196

15. 次のプログラムをコンパイル、実行したときの結果として、正しいものを選びなさい。(1つ選択)

```
1. public class Value {
2.     private String data;
3.     public Value(String data) {
4.         this.data = data;
5.     }
6.     @Override
7.     public int hashCode() {
8.         return 100;
9.     }
10.    @Override
11.    public boolean equals(Object obj) {
12.        return true;
13.    }
14. }
```

```
1. import java.util.Arrays;
2. import java.util.List;
3.
4. public class Test {
5.     public static void main(String[] args) {
6.         List<Value> list = Arrays.asList(
7.             new Value("A"),
8.             new Value("B"),
9.             new Value("C"),
10.            new Value("A")
11.        );
12.        long size = list.stream().distinct().count();
13.        System.out.println(size);
14.    }
15. }
```

A. 「1」と表示される
B. 「3」と表示される
C. 「4」と表示される
D. コンパイルエラーが発生する
E. 実行時に例外がスローされる

➡ P198

16. AからZまでの文字から成るストリームを扱っている。このときの説明として、正しいものを選びなさい。(2つ選択)

A. シングルスレッドでストリームを扱っている場合は、findAnyメソッドはランダムな要素を1つ戻す
B. マルチスレッドでストリームを扱っている場合は、findAnyはランダムな要素を1つ戻す
C. シングルスレッドでストリームを扱っている場合は、findFirstメソッドはストリーム・パイプラインで最初に処理した要素を1つ戻す
D. マルチスレッドでストリームを扱っている場合は、findFirstメソッドはストリーム・パイプラインで最初に処理した要素を1つ戻す

➡ P200

17. 次のプログラムの「/* insert code here */」に入るコードとして、正しいものを選びなさい。(1つ選択)

```
1. import java.util.Arrays;
2. import java.util.List;
3.
4. public class Test {
5.     public static void main(String[] args) {
6.         List<Integer> list = Arrays.asList(5,4,3,2,1);
7.         list.stream()
8.             ./* insert code here */((a, b) -> {
9.                 if (a < b) return -1;
10.                if (b < a) return 1;
11.                return 0;
12.            })
13.            .forEach(System.out::println);
14.    }
15. }
```

A. sort
B. sorting
C. sorted
D. compare

➡ P202

18. 次のプログラムの「/* insert code here */」に入るコードとして、正しいものを選びなさい。（1つ選択）

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Stream;

public class Test {
    public static void main(String[] args) {
        List<Integer> list = Arrays.asList(1,2,3,4,5);
        /* insert code here */ =
            list.stream().map(n -> String.valueOf(n));
    }
}
```

A. Stream<String> result
B. Stream<Integer> result
C. List<String> result
D. List<Integer> result

➡ P204

19. 次のプログラムの「/* insert code here */」に入るコードとして、最も適切なものを選びなさい。（1つ選択）

```
List<Integer> list = Arrays.asList(1,2,3,4,5);
/* insert code here*/ result
        = list.stream().reduce(0, (a, b) -> a + b);
```

A. int
B. Optional<Integer>
C. Stream<Integer>
D. Integer

➡ P206

20. 次のプログラムをコンパイル、実行したときの結果として、正しいものを選びなさい。（1つ選択）

```
import java.util.Arrays;
import java.util.List;
import java.util.Optional;

public class Main {
    public static void main(String[] args) {
        List<String> list = Arrays.asList("B", "A", "D", "C");
        Optional<String> result =
                list.stream().max((a, b) -> a.compareTo(b));
        result.ifPresent(System.out::println);
    }
}
```

A. 「A」と表示される
B. 「C」と表示される
C. 「D」と表示される
D. コンパイルエラーが発生する
E. 実行時に例外がスローされる

➡ P209

21. ストリームを扱う際に、途中経過を保持するオブジェクトを使いたい。このような処理に適したインタフェースまたはクラスを選びなさい。（1つ選択）

A. java.util.stream.Collector
B. java.util.stream.Stream
C. java.util.stream.StreamSupport
D. java.util.stream.Stream.Builder

➡ P210

22. java.util.stream.Collectorインタフェースに規定されているメソッドのうち、実際に行いたい処理を記述するためのものを選びなさい。（1つ選択）

A. supplierメソッド
B. accumulatorメソッド
C. combinerメソッド
D. finisherメソッド
E. characteristicsメソッド

➡ P214

23. 次のプログラムのように、リストの要素を条件によって絞り込み、新しいリストを作りたい。「/* insert code here */」に入るコードとして、正しいものを選びなさい。（1つ選択）

```
 1. import java.util.List;
 2. import java.util.stream.Collectors;
 3.
 4. public class Sample {
 5.     public static void main(String[] args) {
 6.         List<Integer> list = List.of(1,2,3,4,5,6,7,8,9,10);
 7.         List<Integer> result =
 8.                 list.stream()
 9.                     .filter(n -> n % 2 == 0)
10.                     ./* insert code here */;
11.         result.forEach(System.out::println);
12.     }
13. }
```

A. toList()
B. collect(Collectors.toList())
C. reduce(Collectors.toList())
D. getList()

➡ P221

24. 次のプログラムの「/* insert code here */」に入るコードとして、正しいものを選びなさい。（1つ選択）

```
 1. public class Department {
 2.     private String name;
 3.     public Department(String name) {
 4.         this.name = name;
 5.     }
 6.     public String getName() {
 7.         return name;
 8.     }
 9.     @Override
10.     public int hashCode() {
11.         return name.hashCode();
12.     }
13.     @Override
14.     public boolean equals(Object obj) {
15.         if (obj instanceof Department) {
16.             Department other = (Department) obj;
17.             return this.name.equals(other.name);
18.         }
19.         return false;
20.     }
21. }
```

```
 1. import java.util.List;
 2. import java.util.Map;
 3. import java.util.stream.Collectors;
 4.
 5. public class Sample {
 6.     public static void main(String[] args) {
 7.         Department tokyo = new Department("Tokyo");
 8.         Department osaka = new Department("Osaka");
 9.         List<Employee> list = List.of(
10.             new Employee("Johnny", tokyo),
11.             new Employee("Bond", osaka),
12.             new Employee("Mickey", tokyo),
13.             new Employee("Sara", osaka)
14.         );
15.         /* insert code here */ result =
16.                 list.stream().collect(
17.                     Collectors.groupingBy(Employee::getDept)
18.                 );
19.         // do something
20.     }
21. }
```

A. Map<Department, Employee>
B. Map<List<Department>, List<Employee>>
C. Map<Department, List<Employee>>
D. List<Employee>
E. List<Department>

➡ P225

25. リスト内の書籍のタイトルごとの合計金額を算出するために、以下の「// insert code here」に入るコードとして正しいものを選びなさい。（1つ選択）

```
1. public class Book {
2.     private String title;
3.     private int price;
4.     public Book(String title, int price) {
5.         super();
6.         this.title = title;
7.         this.price = price;
8.     }
9.     public String getTitle() {
10.        return title;
11.    }
12.    public int getPrice() {
13.        return price;
14.    }
15. }
```

```
1. import java.util.Arrays;
2. import java.util.Map;
3. import java.util.stream.Collectors;
4.
5. public class Sample {
6.     public static void main(String[] args) {
7.         Map<String, Integer> map = Arrays.asList(
8.             new Book("A", 1000),
9.             new Book("B", 2000),
10.            new Book("A", 500)
11.        ).stream()
12.        .collect(
13.            // insert code here
14.        );
15.        map.forEach((k,v) -> {
16.            System.out.println(k + ":" + v);
17.        });
18.    }
19. }
```

A. `reduce(Book::getPrice)`
B. `reduce(b -> b.getPrice())`
C. `summingInt(b -> b.getPrice())`
D.
```
Collectors.groupingBy(
    Book::getTitle,
    Collectors.summingInt(Book::getPrice)
)
```

➡ P227

26. 次のプログラムをコンパイル、実行したときの結果として、正しいものを選びなさい。（1つ選択）

```
1. import java.util.Arrays;
2. import java.util.List;
3. import java.util.Set;
4. import java.util.stream.Collectors;
5. import java.util.stream.Stream;
6.
7. public class Sample {
8.     public static void main(String[] args) {
9.         List<String> list
10.            = Arrays.asList("banana", "apple", "orange");
11.        Stream<String> stream = list.stream();
12.        Set keys = stream.collect(
13.            Collectors.partitioningBy(str -> str.length() > 5)
14.        ).keySet();
15.        keys.forEach(System.out::println);
16.    }
17. }
```

A. 「true」「false」と表示される
B. 「TRUE」「FALSE」と表示される
C. 「banana」「orange」と表示される
D. コンパイルエラーが発生する
E. 実行時に例外がスローされる

➡ P228

27. 次のプログラムをコンパイル、実行したときの結果として、正しいものを選びなさい。(1つ選択)

```
1. import java.util.Arrays;
2. import java.util.List;
3. import java.util.stream.Stream;
4.
5. public class Sample {
6.     public static void main(String[] args) {
7.         List<String> list
8.             = Arrays.asList("banana", "apple", "orange");
9.         Stream<String> stream = list.stream();
10.        stream.filter(str -> str.length() > 5)
11.              .peek(str -> System.out.print(str + " "))
12.              .map(str -> str.toUpperCase())
13.              .peek(str -> System.out.print(str + " "))
14.              .forEach(System.out::print);
15.    }
16. }
```

A. 「banana orange BANANA ORANGE BANANA ORANGE」と表示される
B. 「banana orange BANANA APPLE ORANGE BANANA APPLE ORANGE」と表示される
C. 「banana BANANA orange ORANGE」と表示される
D. 「banana BANANA BANANAorange ORANGE ORANGE」と表示される
E. コンパイルエラーが発生する
F. 実行時に例外がスローされる

➡ P230

28. 次のプログラムをコンパイル、実行したときの結果として、正しいものを選びなさい。(1つ選択)

```
1. import java.util.Arrays;
2. import java.util.List;
3. import java.util.stream.Stream;
4.
5. public class Sample {
6.     public static void main(String[] args) {
7.         List<Integer> list = Arrays.asList(1,2,3);
8.         Stream<Integer> stream = list.stream();
9.         System.out.println(stream.count());
10.        stream.forEach(System.out::println);
11.    }
12. }
```

A. 「3」「1」「2」「3」と表示される
B. 「2」「1」「2」「3」と表示される
C. コンパイルエラーが発生する
D. 実行時に例外がスローされる

➡ P232

第4章　ストリームAPI
解　答

1. D

➡P154

Optionalクラスに関する問題です。
従来、メソッドの設計時には、処理結果に異常があれば例外をスローするようにしていました。このような異常への対処方法は、処理の途中で突然catchブロックに制御が遷移するため、コードの可読性を下げる原因となっていました。

Java SE 8で導入された**java.util.Optional**は、メソッドの処理結果を扱うためのクラスで、処理結果の正常・異常にかかわらず同じ型で扱うことができます。このクラスを利用することにより、例外処理に関するコードを減らすことができ、その結果、コードの可読性を上げることができます。

たとえば、次のようなコードがあったとしましょう。この**getFromArrayメソッド**は、配列とインデックス番号を受け取り、配列からインデックス番号に一致する要素を戻します。

例 OldTestクラス

```
 1. public class OldTest {
 2.     public static void main(String[] args) {
 3.         try {
 4.             String result =
 5.                 getFromArray(new String[]{"A","B",null}, 3);
 6.             if (result == null) {
 7.                 System.out.println("empty");
 8.                 return;
 9.             }
10.             System.out.println(result);
11.         } catch (IllegalArgumentException e) {
12.             System.out.println("exception");
13.         }
14.     }
15.     private static <T> T getFromArray(T[] array, int index) {
16.         if (array == null) {
17.             throw new IllegalArgumentException();
18.         }
19.         try {
20.             return array[index];
21.         } catch (ArrayIndexOutOfBoundsException e) {
22.             throw new IllegalArgumentException();
23.         }
24.     }
25. }
```

このメソッドを利用するときに注意すべきなのは、次の4点です。

1. 引数の配列は参照型であるため、配列への参照ではなくnullが渡される可能性がある
2. 引数で渡されるインデックスが、配列の要素数を超えてしまうことがある
3. 配列の要素がnullである可能性があるため、戻り値を受け取った側でnullの確認をしなければjava.lang.NullPointerExceptionが発生する
4. スローされてくるjava.lang.IllegalArgumentExceptionは非検査例外であるため、例外処理が無視される可能性がある

これらの問題を解決するのが、Optionalクラスです。次のコードが、Optionalクラスを使って先ほどのコードを書き換えた例です。mainメソッドから例外処理に関するコードがなくなった点に注目してください。

getFromArrayメソッド内で例外処理をせざるを得ないのは、配列の要素外アクセスをしたときに例外がスローされてしまうからです。仮に、配列の要素アクセスの結果がOptionalクラスを戻すように言語仕様が変更されることがあればこのような例外処理は必要なくなりますが、現実的にはそうした変更はあり得ないでしょう。

例 Optionalクラスの使用例

```
 1. import java.util.Optional;
 2.
 3. public class Test {
```

※次ページに続く

```
 4.     public static void main(String[] args) {
 5.         Optional<String> result =
 6.                 getFromArray(new String[]{"A","B",null}, 3);
 7.         if (result.isEmpty()) {
 8.             System.out.println("empty");
 9.             return;
10.         }
11.         System.out.println(result.get());
12.     }
13.     private static <T> Optional<T> getFromArray(T[] array, int index) {
14.         if (array == null) {
15.             return Optional.empty();
16.         }
17.         try {
18.             return Optional.ofNullable(array[index]);
19.         } catch (ArrayIndexOutOfBoundsException e) {
20.             return Optional.empty();
21.         }
22.     }
23. }
```

まず、getFromArrayメソッドから見ていきましょう。このメソッドでは、まず引数の配列がnullでないかを確認しています。もし、変数arrayの中身がnullであれば、Optionalクラスの**emptyメソッド**を使って、空のOptionalのインスタンスを生成し、その参照を戻します。

もし、変数arrayがnullでなければ、Optionalクラスの**ofNullableメソッド**を使って、indexに一致する配列の要素を持ったOptionalのインスタンスを生成し、その参照を戻します。このとき、配列の要素を超えるindexが渡されると、要素外アクセスの例外がスローされます。その場合、emptyメソッドを使って空のOptionalのインスタンスを生成し、その参照を戻します。また、indexで指定された配列の要素がnullであった場合は、ofNullableメソッドは空のOptionalのインスタンスを生成します。

こうして、何らかの異常があったときは空のOptionalのインスタンスへの参照を、正常な場合は正常な結果を持ったOptionalのインスタンスへの参照を戻すので、このメソッドを使うクラスでは例外処理を行う必要がありません。前述の課題とOptionalクラスを利用することによる解決策は、次ページのように対応付けることができます。

課題	解決策
1. 引数の配列は参照型であるため、配列への参照ではなくnullが渡される可能性がある	nullチェックを行い、空のOptionalのインスタンスへの参照を戻す
2. インデックスが、配列の要素数を超えてしまうことがある	例外が発生したときは空のOptionalのインスタンスへの参照を戻す
3. 配列の要素がnullである可能性があるため、戻り値を受け取った側でnullの確認をしなければNullPointerExceptionが発生する	ofNullableメソッドは、引数の値がnullだった場合、空のOptionalのインスタンスへの参照を戻す
4. スローされてくるIllegalArgumentExceptionは非検査例外であるため、例外処理が無視される可能性がある	例外ではなく戻り値を戻すため、例外処理そのものが不要

mainメソッドでは、getFromArrayメソッドの戻り値をOptional型の変数resultで受け取ります。その後、getFromArrayメソッドで何らかの異常があれば空のOptionalが戻ってきているはずなので、**isEmptyメソッド**で空かどうかを確かめています。もし、isEmptyメソッドがtrueを戻さなかったときは、Optionalクラスの**getメソッド**を使って正常な処理結果を受け取ってコンソールに表示します。

このコード例では、3つしか要素がない配列の4番目（添字は3）を取得しようとしているため、getFromArrayメソッド内で**java.lang.ArrayIndexOutOfBoundsException**が発生し、空のOptionalのインスタンスへの参照が戻されます。そのため、このコードを実行すると次のようにコンソールに表示されます。

例 実行結果

```
> java Test
empty
```

Optionalクラスには公開されたコンストラクタが存在せず、newを使ってインスタンスを生成することはできません。そのため、このクラスのインスタンスを生成するには、次ページの表に示したインスタンス生成のためのメソッドを使います。なお、このようなインスタンスを生成するためのメソッドのことを、一般的に「**ファクトリメソッド**」と呼びます。

※次ページに続く

【インスタンス生成のためのOptionalクラスのメソッド】

メソッド	説明
static <T> Optional<T> empty()	空のOptionalのインスタンスを生成し、参照を戻す
static <T> Optional<T> of(T value)	null以外の値を持ったOptionalのインスタンスを生成し、参照を戻す
static <T> Optional<T> ofNullable(T value)	値を持っているか、値がnullの場合は、空のOptionalのインスタンスを生成し、参照を戻す

値を保持したOptionalのインスタンスを生成するには、**ofメソッド**もしくはofNullableメソッドを使います。この2つのメソッドは、引数で渡す値がnullである可能性があるかどうかで使い分けます。

ofメソッドは、引数にnullを許容しません。そのため、次のコード例はコンパイルはできるものの、実行するとNullPointerExceptionが発生します。

例 ofメソッドの引数にnullを渡す

```
1. import java.util.Optional;
2.
3. public class OfTest {
4.     public static void main(String[] args) {
5.         Optional<String> sample = Optional.of(null);
6.         System.out.println(sample);
7.     }
8. }
```

例 実行結果

```
> java OfTest
Exception in thread "main" java.lang.NullPointerException
    at java.base/java.util.Objects.requireNonNull(Objects.java:221)
    at java.base/java.util.Optional.<init>(Optional.java:107)
    at java.base/java.util.Optional.of(Optional.java:120)
    at useOptional.OfTest.main(OfTest.java:5)
```

設問のコードではofメソッドの引数にnullを渡しているため、コンパイルはできるものの、実行時に例外が発生します。以上のことから、選択肢**D**が正解です。

nullの可能性がある値を使ってOptionalのインスタンスを生成するのであれば、ofメソッドではなく**ofNullableメソッド**を使います。このメソッドは、引数にnullが渡されれば、空のOptionalのインスタンスを生成して参照を戻します。次のコードが前述のコードを書き換えた例です。

例 ofNullableメソッドの使用例

```
1. import java.util.Optional;
2.
3. public class OfTest {
4.     public static void main(String[] args) {
5.         Optional<String> sample = Optional.ofNullable(null);
6.         System.out.println(sample);
7.     }
8. }
```

このコードを実行すると例外は発生せず、次のようにコンソールに表示されます。

例 実行結果

```
> java OfTest
Optional.empty
```

引数に渡す値がnullの可能性がある場合は、ofNullableメソッドを使いましょう。

試験対策

Optionalのインスタンスを生成する3つのファクトリメソッド(empty、of、ofNullable)の違いを押さえておきましょう。
ofメソッドはnullを許容しません。そのため、nullを渡すと実行時にNullPointerExceptionがスローされます。

2. C ➡P154

Optionalクラスのメソッドに関する問題です。
Optionalクラスのインスタンスから値を取り出すには、**getメソッド**を使います。次のコードではgetメソッドを使って値を取り出し、コンソールに表示しています。

例 getメソッドの使用例

```
1. import java.util.Optional;
2.
3. public class GetTest {
4.     public static void main(String[] args) {
5.         Optional<String> sample = Optional.of("Hello");
6.         System.out.println(sample.get());
7.     }
8. }
```

例 実行結果

```
> java GetTest
Hello
```

getメソッドは、値が入っていることを前提としているため、空のOptionalのインスタンスからは値を取り出せません。そのため、次のように**emptyメソッド**を使って空のOptionalを扱うように前述のコードを変更し、このコードを実行すると、getメソッドが実行されたタイミングで**java.util.NoSuchElementException**がスローされます。

例 getメソッドで空のOptionalから値を取り出すコード

```
1. import java.util.Optional;
2.
3. public class GetTest {
4.     public static void main(String[] args) {
5.         Optional<String> sample = Optional.empty();
6.         System.out.println(sample.get());
7.     }
8. }
```


例 実行結果

```
Exception in thread "main" java.util.NoSuchElementException:
No value present
   at java.base/java.util.Optional.get(Optional.java:148)
   at useOptional.GetTest.main(GetTest.java:6)
```

設問のコードでは、OptionalクラスのofNullableメソッドの引数にnullを渡して空のOptionalのインスタンスを作っています。そのため、getメソッドを呼び出したタイミングで例外がスローされます。以上のことから、選択肢**C**が正解です。

このような例外の発生を避けるためには、Optionalのインスタンスが空であるかを調べる必要があります。Optionalクラスには、値を保持しているかを調べるためにisPresentとisEmptyという2つのメソッドが用意されています。

isPresentメソッドは、Optionalのインスタンスが何らかの値を持っていればtrue、空であればfalseを戻します。次のコード例では、OptionalをisPresentメソッドで調べていますが、このインスタンスは空であるため、このメソッドはfalseを戻します。そのため、if文は実行されず、コンソールには何も表示されません。

例 isPresentメソッドの使用例

```
 1. import java.util.Optional;
 2.
 3. public class GetTest {
 4.     public static void main(String[] args) {
 5.         Optional<String> sample = Optional.empty();
 6.         if (sample.isPresent()) {
 7.             System.out.println(sample.get());
 8.         }
 9.     }
10. }
```

isPresentメソッドでは、Optionalが値を持っているかどうかを調べましたが、空であることを調べるには、**isEmptyメソッド**を使います。次のコード例を実行すると、コンソールに「empty」と表示されます。

※次ページに続く

例 isEmptyメソッドの使用例

```
1. import java.util.Optional;
2.
3. public class GetTest {
4.     public static void main(String[] args) {
5.         Optional<String> sample = Optional.empty();
6.         if (sample.isEmpty()) {
7.             System.out.println("empty");
8.             return;
9.         }
10.        System.out.println(sample.get());
11.    }
12. }
```

試験対策

getメソッドは、値が入っていることを前提としているため、getメソッドで空のOptionalのインスタンスから値を取り出そうとすると、NoSuchElementExceptionがスローされます。isPresentメソッドでは、Optionalに値が入っているかどうかを調べることができます。isEmptyメソッドでは、Optionalが空かどうかを調べることができます。

3. A ➡P155

Optionalクラスのメソッドに関する問題です。
isPresentメソッドやisEmptyメソッドを使えば、Optionalのインスタンスが値を持っているかどうかを調べられますが、より簡便に記述できるのが**orElseメソッド**です。このメソッドは、引数に代替値を指定することができ、値を持っていればその値を、空であれば引数で渡した代替値を戻します。そのため、値の有無を調べるためのif文を記述する必要がありません。
次のコードが、orElseメソッドの使用例です。このコードを実行すると、コンソールに「empty」と表示されます。

例 orElseメソッドの使用例

```
1. import java.util.Optional;
2.
3. public class GetTest {
4.     public static void main(String[] args) {
5.         Optional<String> sample = Optional.empty();
6.         System.out.println(sample.orElse("empty"));
7.     }
8. }
```

orElseメソッドは必ず何らかの値を戻しますが、Optionalのインスタンスが空のときに戻せるのは引数で渡した値だけです。固定的な値ではなく、何らかの処理をしてからその結果を戻したい場合には、orElseGetメソッドを使います。

orElseGetメソッドは引数としてSupplier型（何も受け取らず結果だけを戻す関数型インタフェース型）のラムダ式を受け取り、Optionalのインスタンスが空だった場合には、引数のラムダ式を実行してその結果を戻します。次のコード例ではOptionalは空なので、ラムダ式が戻す「else」という文字列がコンソールに表示されます。

例 orElseGetメソッドの使用例

```
1. import java.util.Optional;
2.
3. public class GetTest {
4.     public static void main(String[] args) {
5.         Optional<String> sample = Optional.empty();
6.         System.out.println(sample.orElseGet(() -> "else"));
7.     }
8. }
```

設問のコードでは、OptionalクラスのofメソッドをAという文字列を持ったOptionalのインスタンスを作成しています。orElseメソッドの引数は、空のOptionalのときの代わりとなるものなので、このコードはofメソッドの引数に渡した「A」という文字列がコンソールに表示されます。以上のことから、選択肢**A**が正解です。

4. C ➡P155

Optionalクラスのメソッドに関する問題です。
getメソッドで空の値を扱う問題点は、非検査例外が発生するため、例外処理が無視される（catchし忘れる）可能性があることでした。Optionalが空の場合、意図的に例外を発生させたいのであれば、getメソッドではなく**orElseThrowメソッド**を使いましょう。このメソッドであれば、getメソッドがスローする非検査例外であるNoSuchElementExceptionの代わりに、**任意の検査例外**をスローすることができます。

※次ページに続く

orElseThrowメソッドは、例外やエラーのスーパークラスであるThrowable型を戻すSupplier型のラムダ式を引数に受け取ります。次のコード例では、Exceptionのインスタンスを生成して戻しているため、クラス宣言でもthrowsを使ってExceptionがスローされることを宣言しています。もちろん、try-catchで捕捉して処理することも可能です。

例 orElseThrowメソッドの使用例

```
1. import java.util.Optional;
2.
3. public class GetTest {
4.     public static void main(String[] args) throws Exception {
5.         Optional<String> sample = Optional.empty();
6.         System.out.println(
7.             sample.orElseThrow(() -> new Exception()));
8.     }
9. }
```

以上のことから、選択肢**C**が正解です。
選択肢Aのgetメソッドも Optionalインスタンスが空の値を扱っているときには例外をスローしますが、スローされるのはjava.util.NoSuchElementExceptionであって、任意の例外をスローできるわけではありません。選択肢Bのorelseメソッドは、空だったときに代わりの値を戻すためのメソッドであって、例外をスローすることはできません。選択肢Dのようなメソッドは Optionalクラスには存在しません。よって、これらの選択肢は誤りです。

5. B

➡ P155

Optionalクラスのメソッドに関する問題です。
Optionalのインスタンスから値を取り出すにはgetメソッドやorElseメソッドを使いましたが、取り出さずに処理したい場合には**ifPresentメソッド**を使います。このメソッドはConsumer型（引数を受け取り何も戻さない）のラムダ式を受け取り、Optionalが値を持っていれば、引数に渡してラムダ式を実行します。

次のコード例では、Optionalのインスタンスが空でなければ、ラムダ式でその値をString型の引数strで受け取り、コンソールに表示します。

例 ifPresentメソッドの使用例（Optionalが値を持っている）

```
1. import java.util.Optional;
2.
3. public class Sample {
4.     public static void main(String[] args) {
5.         Optional<String> sample = Optional.of("test");
6.         sample.ifPresent((str) -> System.out.println(str));
7.     }
8. }
```

例 実行結果

```
> java Sample
test
```

ifPresentメソッドは、Optionalのインスタンスが値を持っていなければ何もしません。たとえば、次のようにこのコード例を修正すると、Optionalは空なのでラムダ式が実行されることはありません。getメソッドのように例外が発生することもありません。

例 ifPresentメソッドの使用例（Optionalが値を持っていない）

```
1. import java.util.Optional;
2.
3. public class Sample {
4.     public static void main(String[] args) {
5.         Optional<String> sample = Optional.empty();
6.         sample.ifPresent((str) -> System.out.println(str));
7.     }
8. }
```

以上のことから、選択肢**B**が正解です。

6. A

➡ P156

OptionalクラスのifPresentOrElseTestメソッドに関する問題です。
前問で説明したifPresentメソッドは、Optionalのインスタンスが値を保持していない場合は何も処理をしません。何か処理をしたい場合は、**ifPresentOrElseメソッド**を使います。このメソッドは、値がある場合は第1引数で指定したラムダ式（Consumer型）を実行し、値がない場合は第2引数

で指定したラムダ式(Runnable型)を実行します。よって、選択肢**A**が正解です。

次のコードは、ifPresentOrElseメソッドを使った例です。このコードをコンパイル、実行すると、空のOptionalのインスタンスに対するラムダ式が実行され、コンソールには「empty」と表示されます。

例 ifPresentOrElseメソッドの使用例

```
1. import java.util.Optional;
2.
3. public class IfPresentOrElseTest {
4.     public static void main(String[] args) {
5.         Optional<String> sample = Optional.empty();
6.         sample.ifPresentOrElse(
7.                 (str) -> System.out.println(str),
8.                 () -> System.out.println("empty"));
9.     }
10. }
```

例 実行結果

```
> java IfPresentOrElseTest
empty
```

試験対策

解答3〜6で解説した各メソッドの引数と戻り値について、しっかり押さえておきましょう。

7. B

➡P156

Optionalクラスのmapメソッドに関する問題です。
Optionalのインスタンスが持つ値を加工し、新たに別のOptionalを作るときには、次のようにgetメソッドでいったん値を取り出す方法があることを解答2で学びました。

例 getメソッドで値を取り出してから加工する

```
1. import java.util.Optional;
2.
3. public class MapTest {
4.     public static void main(String[] args) {
5.         Optional<String> sample = Optional.of("test");
6.         Optional<String> result = null;
7.         if (sample.isPresent()) {
8.             String str = sample.get();
9.             result = Optional.of(str.toUpperCase());
10.         }
11.         System.out.println(sample.get());
12.         System.out.println(result.get());
13.     }
14. }
```

このコードは、mapメソッドを使えばもっとシンプルに記述できます。**mapメソッド**はFunction型の引数を受け取り、処理結果を持った新しいOptionalのインスタンスを生成し、その参照を戻します。次のコードが、mapメソッドを使って前述のコードを書き換えた例です。

例 mapメソッドの使用例

```
1. import java.util.Optional;
2.
3. public class MapTest {
4.     public static void main(String[] args) {
5.         Optional<String> sample = Optional.of("test");
6.         Optional<String> result = sample.map(
7.                                 str -> str.toUpperCase());
8.         System.out.println(sample.get());
9.         System.out.println(result.get());
10.     }
11. }
```

前述のコードのように、isPresentメソッドで値があることを確かめ、getメソッドで取り出して加工し、その結果を使って新しいOptionalのインスタンスを作るという手順が、mapメソッドの引数にラムダ式を渡すだけでよくなりました。コード量が減り、簡潔になったことがわかります。

設問のコードでは、mapメソッドを使って、文字列を大文字に変換した結果を戻しています。そのため、コンソールには、「abcde」のあとに「ABCDE」と表示されます。以上のことから、選択肢**B**が正解です。

※次ページに続く

なおmapメソッドは、処理するOptionalのインスタンスが空だった場合には新たな空のOptionalのインスタンスを生成するため、例外が発生することはありません。
次のコード例では、空のOptionalのインスタンスに対して文字列を大文字に変更しようとしていますが、新たな空のOptionalのインスタンスへの参照が戻されるだけで、例外が発生することはありません。

例 mapメソッドの使用例（空のOptionalの場合）

```
 1. import java.util.Optional;
 2.
 3. public class MapTest {
 4.     public static void main(String[] args) {
 5.         Optional<String> sample = Optional.ofNullable(null);
 6.         Optional<String> result = sample.map(
 7.                                 str -> str.toUpperCase());
 8.         System.out.println(result.isEmpty());
 9.     }
10. }
```

8. C

➡ P157

処理結果を持つOptionalのインスタンスを生成するときに使うメソッドについて問う問題です。
前問で解説したように、Optionalクラスのmapメソッドを使うことで、処理結果を持った別のOptionalのインスタンスを新たに生成することができます。では、次のコード例のように、値ではなくOptionalを戻すメソッドをmapメソッドで呼び出したらどうなるでしょうか。

例 FlatMapTestクラス

```
 1. import java.util.Optional;
 2.
 3. public class FlatMapTest {
 4.     public static void main(String[] args) {
 5.         Optional<String> a = Optional.of("test");
 6.         Optional<String> b = a.map(str -> test(str));
 7.         System.out.println(b.get());
 8.     }
 9.     private static Optional<String> test(String str) {
10.         return Optional.of(str.toUpperCase());
11.     }
12. }
```


mapメソッド内で呼び出しているtestメソッドは、その戻り値を受け取るための変数bの型と一致しています。このコードは実行できるように見えますが、実際にはコンパイルエラーとなるのです。その原因は、mapメソッドは、処理結果を値として保持した新しいOptionalのインスタンスを生成する点にあります。
testメソッドは、String型を扱うOptionalのインスタンスへの参照を戻します。そのため、mapメソッドはこの参照を持った新しいOptionalのインスタンスを生成することになります。つまり、Optionalの中にOptionalへの参照が入れ子で入ることになるのです。そのため、mapメソッドは、Optional<Optional<String>>型の値を戻しますが、戻り値を受け取る変数bの型はOptional<String>であるため、互換性がないとして、コンパイルエラーとなるのです。

このような問題を解決するには、mapメソッドではなく、**flatMapメソッド**を使います。このメソッドは、戻り値を新しいOptionalのインスタンスに入れずにそのまま戻します。次のコードが、前述のコードを書き換えた例です。

例 flatMapメソッドの使用例

```
 1. import java.util.Optional;
 2.
 3. public class FlatMapTest {
 4.     public static void main(String[] args) {
 5.         Optional<String> a = Optional.of("test");
 6.         Optional<String> b = a.flatMap(str -> test(str));
 7.         System.out.println(b.get());
 8.     }
 9.     private static Optional<String> test(String str) {
10.         return Optional.of(str.toUpperCase());
11.     }
12. }
```

このようにflatMapメソッドは、testメソッドの戻り値を新しいOptionalに入れずにそのまま戻します。よって、コンパイルエラーは起きず、コンソールには「TEST」と表示されます。

設問のコードのcalcメソッドはOptionalを戻します。そのため、calcメソッ

ドの戻り値を使ってmapメソッドを実行すると、戻り値の型はOptional<Optional<Integer>>となってしまいます。そのため、Optional<Integer>型の変数で戻り値を受け取ることができず、コンパイルエラーとなります。以上のことから、選択肢Cが正解です。

試験対策

mapメソッドは、処理結果を持ったOptionalのインスタンスを新たに生成し、その参照を戻します。Optionalを戻すメソッドをmapメソッドから呼び出すと、Optionalの中にOptionalへの参照が入れ子で入ることになります。戻り値を新しいOptionalのインスタンスに入れずにそのまま戻したい場合は、flatMapメソッドを使います。

9. A ➡P157

forEachメソッドの使い方に関する問題です。
従来のプログラミング方法では、ListやSetといったコレクションを扱うときに、for文や拡張for文といった繰り返し構文を使いながら、コレクションから順に要素を取り出して処理していました。次のコードが、従来どおり、リストから拡張for文を使いながら順番に処理している例です。

例 拡張for文を使った要素の取り出し

```
 1. import java.util.List;
 2.
 3. public class Sample {
 4.     public static void main(String[] args) {
 5.         List<Integer> list = List.of(1, 2, 3, 4, 5);
 6.         for (Integer num : list) {
 7.             System.out.println(num);
 8.         }
 9.     }
10. }
```

このようなコレクションに対する逐次処理を簡単に記述できるように、Java SE 8で**java.lang.Iterableインタフェース**が拡張され、**forEachメソッド**が使えるようになりました。前述のコード例を、新しく拡張された機能を使って書き換えたものが、次のコード例です。for文の代わりにforEachメソッドを使っているため、コードが簡潔になったことがわかります。

例 forEachメソッドを使った要素の取り出し

```
1. import java.util.List;
2.
3. public class Sample {
4.     public static void main(String[] args) {
5.         List<Integer> list = List.of(1, 2, 3, 4, 5);
6.         list.forEach(x -> System.out.println(x));
7.     }
8. }
```

このコード例で利用したforEachメソッドは、java.lang.Iterableインタフェースで新しく定義されているもので、コレクションから要素を順番に取り出しながら、引数に渡されたConsumer型のラムダ式を実行します。コレクションから値を1つずつ取り出し、繰り返し処理を実行するのであれば、for文などの繰り返し構文よりも、このforEachメソッドを使用するほうがコードが簡潔になります。

なお、コンソールに表示するだけなら、次のように、ラムダ式の代わりに**メソッド参照**を使うほうが簡易に記述できます。

例 メソッド参照

```
1. import java.util.List;
2.
3. public class Sample {
4.     public static void main(String[] args) {
5.         List<Integer> list = List.of(1, 2, 3, 4, 5);
6.         list.forEach(System.out::println);
7.     }
8. }
```

java.util.function.Consumerは、引数を受け取り、結果を戻さない処理を記述するための関数型インタフェースです。そのため、引数を受け取らない選択肢BやC、戻り値を戻す選択肢Dは誤りです。以上のことから、選択肢Aが正解です。

10. C

➡ P158

ストリームAPIに関する問題です。
前問で解説したjava.lang.IterableインタフェースのforEachメソッドのように、Java SE 8ではコレクションをより扱いやすくするための拡張がなされました。**Stream API**（以後、ストリームAPI）もその1つです。**ストリームAPI**では、コレクションの各要素を処理していく「流れ」という意味をこめて、コレクションを「**ストリーム**」という概念で扱います。
コレクションを扱うときに、要素を1つずつ取り出して、何らかの処理を加え、特定の要素だけを取り出したり、配列に変換したり、コレクションを詰め替えたりすることがよくあります。ストリームAPIでは、これらの処理を**メソッドとラムダ式で記述**することで、コードをより簡潔にできます。
ストリームAPIを利用するのは、次のような場合です。

・コレクションや配列の全要素を同じように変換する場合
・コレクションや配列の要素の合計や平均といった統計を取る場合
・コレクションや配列の要素を何らかの条件でグルーピングする場合
・コレクションや配列の要素から条件に合ったデータを検索する場合

ストリームAPIでは、取り出した要素に加える処理のことを「**中間操作**」、中間操作を終えた要素の集合に対して行う最終の処理を「**終端操作**」と呼びます。これらの操作は、ストリームAPIの基本となる機能をまとめた**java.util.stream.Streamインタフェース**に定義されており、次のような種類があります。

【中間操作の種類】

処理を行うメソッド	概要
distinct	要素の重複を除いたストリームを戻す
filter	引数に指定した条件に一致する要素だけで構成されるストリームを戻す
limit	ストリームの要素を、指定された数に切り詰めた結果のストリームを戻す
map	指定された関数を要素に適用した結果のストリームを戻す
peek	新しい結果のストリームを生成し、指定された関数を適用して戻す
skip	引数で指定した最初のn個の要素を破棄し、残った要素で構成されるストリームを戻す
sorted	このストリームの要素を自然順序に従ってソートした結果から構成されるストリームを戻す

【終端操作の種類】

処理を行うメソッド	概要
allMatch	ストリーム内のすべての要素が条件に一致するかどうかを調べる
anyMatch	ストリーム内のいずれかの要素が条件に一致するかどうかを調べる
collect	ストリームの要素を持つコレクションを戻す
count	ストリーム内の要素を数える
findAny	ストリーム内に要素が残っているかどうかの結果を持つOptionalを戻す
findFirst	ストリーム内の最初の要素を持ったOptionalを戻す
forEach	ストリーム内の要素を使って繰り返し処理を実行する
max	ストリーム内の最大の要素を戻す
min	ストリーム内の最小の要素を戻す
noneMatch	ストリーム内の要素で条件に一致するものがないかどうかを調べる
reduce	ストリーム内の要素を累積的に結合していくリダクション処理を実行する
toArray	ストリームの要素を含む配列を戻す

ストリームAPIでは、0個以上の中間操作と1つの終端操作を組み合わせたものを「**ストリーム・パイプライン**」と呼びます。ストリームAPIは、このストリーム・パイプラインを扱うためのAPIなのです。
中間操作の具体的な処理内容は、関数型インタフェース型の引数として受け取ります。中間操作はいくつもつなげることができますが、実際に実行されるのは、終端操作を実行したタイミングです。

まずは、従来のコレクションの使用方法をおさらいしておきましょう。次のコード例は、数値のコレクションから偶数のものだけを抜き出し、コンソールに表示するというものです。

例 従来のコレクションの使用方法

```
1. import java.util.List;
2.
3. public class Sample {
4.     public static void main(String[] args) {
5.         List<Integer> a = List.of(1, 2, 3, 4, 5);
6.
```

```
        for (Integer num : a) {
            System.out.println(num);
        }
    }
}
```

このコードを、ストリームAPIを使用して書き換えたものが次のコード例です。

例 ストリームAPIを使用したコード

```
import java.util.List;

public class Sample {
    public static void main(String[] args) {
        List<Integer> list = List.of(1, 2, 3, 4, 5);
        list.stream()
            .forEach(System.out::println);
    }
}
```

このコードでは、まずList型変数listのstreamメソッドを呼び出して、リストからストリームを作ります。ストリームを作るための代表的なメソッドは、**java.util.Collectionインタフェースのstreamメソッド**です。このメソッドは、デフォルトメソッドとして実装されており、ListやSetといったCollectionのサブインタフェースでも利用できます。

```
List<Integer> list = List.of(1, 2, 3, 4, 5);
Stream<Integer> stream = list.stream();
```

ストリームを生成してから、そのストリームに対して処理を実行するのが終端操作です。このコード例では、forEachを使って繰り返し処理を実行しています。

```
List<Integer> list = List.of(1, 2, 3, 4, 5);
Stream<Integer> stream = list.stream();
stream.forEach(System.out::println);
```


もし、このストリームに中間操作が追加されていれば、終端操作が実行されるタイミングで同時に実行されることになります。

注意していただきたいのは、ストリームAPIは、for文などの繰り返し構文を置き換えるために用意されているわけではないという点です。たとえば、for文などによる繰り返し処理ではローカル変数の読み書きが自由に行えますが、ストリームのラムダ式からはfinalもしくは実質的にfinalなローカル変数しか扱えません。そのため、ストリーム・パイプライン内からローカル変数の値を変更することはできません。ほかにも、for文などの繰り返し処理ではreturnやbreak、continue、もしくは例外のスローによって処理の流れを制御することができますが、ストリームのラムダ式からは処理の流れを制御することはできません。以上のことから、選択肢**C**が正解です。

中間操作と終端操作は重要です。メソッドの種類と機能や、メソッドが中間操作か終端操作なのかを確実に押さえておきましょう。

中間操作の具体的な処理内容は、関数型インタフェース型の引数として受け取ります。中間操作のメソッドの戻り値型に注意しましょう。中間操作はいくつでもつなげることができますが、実際に実行されるのは、終端操作を実行したタイミングです。

11. C

➡ P158

配列からストリームを作るstreamメソッドに関する問題です。
コレクションではなく、**配列からストリームを作る**には、次のコード例のように**java.util.Arraysクラスのstreamメソッド**を使います。このメソッドは、引数に配列を渡すと、その配列が扱える型と同じ型のストリームを生成します。

```
String[] array = {"A", "B", "C"};
Stream<String> stream = Arrays.stream(array);
```

配列からストリームを作る際に気を付けなければいけないのが、intやlong、doubleといった**プリミティブ型の配列**を扱う場合です。たとえば、次のコードはコンパイルエラーとなってしまいます。

※次ページに続く

```
int[] array = {1, 2, 3, 4, 5};
Stream<Integer> stream = Arrays.stream(array);
```

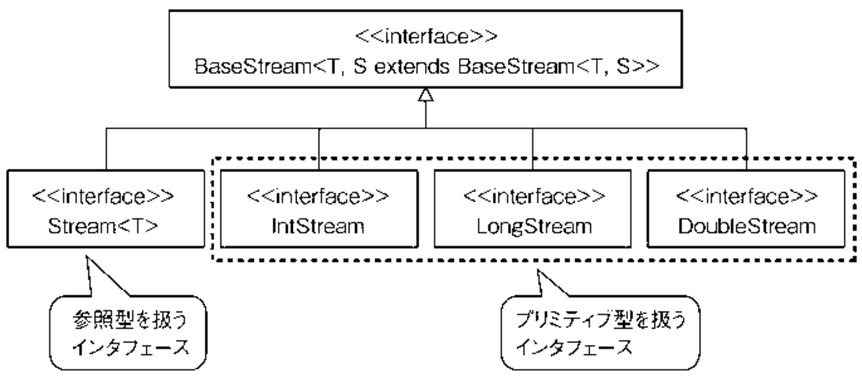
これは、プリミティブ型の配列を扱ったときのArraysクラスのstreamメソッドの戻り値がjava.util.stream.Stream型ではないためです。このメソッドはオーバーロードされており、いくつかの種類が用意されています。

【streamメソッドの主なオーバーロード】

メソッドの定義	説明
static <T> Stream<T> stream(T[] array)	配列からストリームを生成する
static IntStream stream(int[] array)	配列からIntStream型のストリームを生成する
static LongStream stream(long[] array)	配列からLongStream型のストリームを生成する
static DoubleStream stream(double[] array)	配列からDoubleStream型のストリームを生成する

注意すべき点は、intやlong、double型を扱う配列を渡すと、Stream型ではなく、それぞれIntStream、LongStream、DoubleStream型の戻り値を戻すということです。前述のコード例がコンパイルエラーとなったのは、これらのプリミティブ型に対応したストリームAPIのインタフェースと、Streamインタフェースに互換性がないためです。ここで、それぞれのインタフェースの関係性を整理すると、次のクラス図のようになります。

【BaseStreamとサブインタフェース】



ストリームの共通した機能を規定するのが、**java.util.stream.BaseStreamインタフェース**です。StreamやIntStream、LongStream、DoubleStreamは、このBaseStreamインタフェースのサブインタフェースにあたります。図からわかるとおり、Streamインタフェースは参照型のストリームを扱うためのもので、プリミティブ型のストリームを扱うインタフェースとは互換性がありません。以上のことから、選択肢**C**が正解です。

12. B

➡ P158

ストリームの処理順に関する問題です。
ストリーム内の要素を繰り返し処理する場合は、**java.util.stream.Streamインタフェースの終端操作**の1つである**forEachメソッド**を使います。たとえば次のコード例では、forEachメソッドを使って5つの文字列を持ったリストからストリームを取り出し、要素を1つずつコンソールに表示する繰り返し処理を行っています。

例 forEachメソッドの使用例

```
1. import java.util.Arrays;
2. import java.util.List;
3. import java.util.stream.Stream;
4.
5. public class ParallelTest {
6.     public static void main(String[] args) {
7.         List<String> list = Arrays.asList("A","B","C","D","E");
8.         Stream<String> stream = list.stream();
9.         stream.forEach(x -> {
10.            System.out.println(x);
11.        });
12.    }
13. }
```

このメソッドは、java.util.function.Consumer型の引数を受け取ります。Consumer型のラムダ式は、引数を受け取り、それを使って処理するものの、戻り値を戻しません。このメソッドは、解答9で解説した**Iterableインタフェースのforeachメソッド**と同じだと考えるとよいでしょう。

forEachメソッドは、ストリーム内の要素をコレクションが保持している順に処理します。**java.util.ArrayList**のように要素を追加した順番で管理しているのであれば、その順番で処理されます。**java.util.HashSet**のようにハッシュコードで並べ替えて保持しているのであれば、ハッシュコードの順ということになります。たとえば次のコードであれば、addメソッドで追加した「EDCBA」という順番ではなく、ハッシュコードで並べ替えられて「ABCDE」と表示されます。

例 ストリームの要素をハッシュコードの順で並べ替えて1つずつ表示する

```
 1. import java.util.HashSet;
 2. import java.util.Set;
 3. import java.util.stream.Stream;
 4.
 5. public class Test {
 6.     public static void main(String[] args) {
 7.         Set<String> set = new HashSet<>();
 8.         set.add("E");
 9.         set.add("D");
10.         set.add("C");
11.         set.add("B");
12.         set.add("A");
13.         Stream<String> stream = set.stream();
14.         stream.forEach(System.out::println);
15.     }
16. }
```

以上のことから、選択肢**B**が正解です。

13. D

➡ P159

並列ストリームに関する問題です。
ストリームAPIでは、コレクションの処理パフォーマンスを上げるために**マルチスレッド**をサポートしています。昨今のCPUは、複数のコアで構成されるマルチコアが主流です。マルチコアのCPUを搭載したコンピュータであれば、並列処理が可能になるため、マルチスレッドによる恩恵がより大きくなります。多くのオブジェクトの集合を扱うコレクションに対するストリーム・パイプラインでマルチスレッドによる処理が実現できると、複数のコアで一斉に処理できるため、処理パフォーマンスを劇的に上げることも可能です。なお、このような並列処理をするためのストリームのことを「**並列ストリーム**」と呼びます。

並列ストリームの扱い方はとても簡単です。コレクションからストリームを取り出すときに、streamメソッドの代わりに**java.util.CollectionインタフェースのparallelStreamメソッド**を使うだけです。たとえば次のコード例では、並列ストリームを取り出してforEachメソッドで順番に処理しています。

例 parallelStreamメソッドの使用例

```
 1. import java.util.Arrays;
 2. import java.util.List;
 3. import java.util.stream.Stream;
 4.
 5. public class ParallelTest {
 6.     public static void main(String[] args) {
 7.         List<String> list = Arrays.asList("A","B","C","D","E");
 8.         Stream<String> stream = list.parallelStream();
 9.         stream.forEach(System.out::println);
10.     }
11. }
```

このコードは、実行する度に結果が異なるはずです。これは複数のスレッドが起動し、一斉にストリームを処理しているためです。処理する順をプログラムで制御することはできません。
ストリームの処理順を維持したいのであれば、次のコード例のように、forEachメソッドの代わりに**forEachOrderedメソッド**を使います。ストリームの処理順は、コレクションが保持しているとおりとなるはずです。ただ、順番を意識しなくてはいけない分、マルチスレッドの利点であるパフォーマンス向上は見込めません。

例 forEachOrderedメソッドの使用例

```
 1. import java.util.Arrays;
 2. import java.util.List;
 3. import java.util.stream.Stream;
 4.
 5. public class ParallelTest {
 6.     public static void main(String[] args) {
 7.         List<String> list = Arrays.asList("A","B","C","D","E");
 8.         Stream<String> stream = list.parallelStream();
 9.         stream.forEachOrdered(System.out::println);
10.     }
11. }
```

以上のことから、選択肢Cは誤りで、選択肢**D**が正解です。選択肢AやBのようなParallelStreamというインタフェースやクラスは存在しません。よって、これらの選択肢は誤りです。

※次ページに続く

試験対策

マルチスレッドで一斉にストリームを処理する「並列ストリーム」では、処理する順を制御することはできません。ストリームの処理順を維持したい場合は、forEachメソッドの代わりにforEachOrderedメソッドを使います。

14. B

➡ P159

ストリーム内の要素を絞り込むfilterメソッドに関する問題です。
終端操作の1つであるforEachメソッドは、ストリーム内のすべての要素を繰り返し処理します。要素を絞り込むには、**中間操作のfilterメソッド**を使います。このメソッドは、java.util.function.Predicate型の引数を受け取り、ラムダ式で指定された条件に一致するかどうかでストリーム内の要素を絞り込みます。たとえば次のコード例では、偶数かどうかで絞り込みをしています。

例 filterメソッドの使用例

```
1. import java.util.Arrays;
2. import java.util.List;
3.
4. public class FilterSample {
5.     public static void main(String[] args) {
6.         List<Integer> list = Arrays.asList(1,2,3,4,5,6,7,8,9,10);
7.         list.stream()
8.             .filter(x -> x % 2 == 0)
9.             .forEach(System.out::println);
10.     }
11. }
```

中間操作は、実行した結果のストリームへの参照を戻します。そのため上記のコード例では、戻り値の参照を使ってさらにメソッドを呼び出す「**メソッドチェイン**」を使って、filterメソッドの戻り値の参照先にあるインスタンスのforEachメソッドを呼び出しています。メソッドチェインを使わない場合は次のような記述になり、メソッドチェインを使ったほうがコード量が少なくなることがわかります。

例 メソッドチェインを使わない記述

```
1. import java.util.Arrays;
2. import java.util.List;
3. import java.util.stream.Stream;
4.
5. public class FilterSample {
6.     public static void main(String[] args) {
7.         List<Integer> list = Arrays.asList(1,2,3,4,5,6,7,8,9,10);
8.         Stream<Integer> a = list.stream();
9.         Stream<Integer> b = a.filter(x -> x % 2 == 0);
10.        b.forEach(System.out::println);
11.    }
12. }
```

なお、filterメソッドによる絞り込み処理は、複数回行うことができます。次のコード例では、filterメソッドを2回使って、まずストリーム内の要素が5よりも大きいものに絞ってから、さらに偶数のものだけに絞っています。

例 filterメソッドを2回実行して絞り込む

```
1. import java.util.Arrays;
2. import java.util.List;
3.
4. public class FilterSample {
5.     public static void main(String[] args) {
6.         List<Integer> list = Arrays.asList(1,2,3,4,5,6,7,8,9,10);
7.         list.stream()
8.             .filter(x -> x > 5)
9.             .filter(x -> x % 2 == 0)
10.            .forEach(System.out::println);
11.    }
12. }
```

このコード例では単純な条件が2つしかないため効果がわかりにくいですが、もっと複雑な絞り込み処理でfilterメソッドをいくつも組み合わせれば、かなり簡潔なコードになるでしょう。

設問のコードでは、実行結果に示された「678910」を得るために、1〜10の数値を持ったIntegerのリストから、6以上に絞り込んで抽出しています。そのため、5よりも大きい場合にtrueを戻すラムダ式を持った選択肢**B**が正解となります。選択肢Aの条件式では、5よりも小さいものに絞り込むため、1〜4の値がコンソールに表示されてしまいます。よって、誤りです。
選択肢CやDのmapメソッドは、要素に指定された関数を適用した結果のストリームを戻すためのものです。よって、誤りです。

15. A

➡ P160

ストリーム内の要素を絞り込むメソッドに関する問題です。
前問で解説したfilterメソッドのように絞り込みのために使うのが、**distinctメソッド**です。このメソッドは、ストリーム内の要素の重複を取り除きます。たとえば次のコード例では、4つの文字列を持ったリストからストリームを取り出し、distinctメソッドで重複を取り除いてから、コンソールに表示しています。そのため、このコードを実行すると「A」「B」「C」と3つの文字列が表示されます。

例 distinctメソッドの使用例

```
1. import java.util.Arrays;
2. import java.util.List;
3.
4. public class DistinctSample {
5.     public static void main(String[] args) {
6.         List<String> list = Arrays.asList("A","B","C","A");
7.         list.stream()
8.             .distinct()
9.             .forEach(System.out::println);
10.     }
11. }
```

例 実行結果

```
> java DistinctSample
A
B
C
```

なお、重複しているかどうかは**equalsメソッドの結果がtrue**になるかどうかで判定されます。そのため、たとえインスタンスが異なってもequalsメソッドによって「同値である」と判定される場合には、重複していると見なされて取り除かれます。

設問のコードでは、ValueクラスのhashCodeメソッドとequalsメソッドがオーバーライドされ、hashCodeメソッドは常に同じ値を、equalsメソッドは常にtrueを戻します。そのため、常に「同値である」と見なされ、4つのValueクラスのインスタンスも「同値である」と判断されて1つにまとめられます。このコードを実行すると、最初のAという文字列を持ったValueクラスのインスタンスだけが残り、その他のインスタンスへの参照は破棄されます。以上のことから、選択肢**A**が正解です。

ストリーム内の要素を絞り込むメソッドには、条件を指定するfilterメソッド、重複を取り除くdistinctメソッドのほかに、数で絞り込む**limitメソッド**があります。次のコード例では、10個の整数を要素として持つリストからストリームを作り、limitメソッドの引数に5を指定して5つに絞ってから、1つずつ取り出してコンソールに表示する繰り返し処理を行っています。

例 limitメソッドの使用例

```
1. import java.util.Arrays;
2. import java.util.List;
3.
4. public class LimitSample {
5.     public static void main(String[] args) {
6.         List<Integer> list = Arrays.asList(1,2,3,4,5,6,7,8,9,10);
7.         list.stream()
8.             .limit(5)
9.             .forEach(System.out::println);
10.     }
11. }
```

例 実行結果

```
> java LimitSample
1
2
3
4
5
```

順次ストリームを扱っている場合、limitメソッドは元のコレクションが持つ順番に応じて、先頭から数えて引数の数の分だけ要素を絞り込みます。反対に、引数で指定した数以降の要素を取り出すには、次のコード例のようにskipメソッドを使います。このコードでは最初の5つをスキップして、6つ目の要素から順番にコンソールに表示します。

※次ページに続く

例 skipメソッドの使用例

```
1. import java.util.Arrays;
2. import java.util.List;
3.
4. public class SkipSample {
5.     public static void main(String[] args) {
6.         List<Integer> list = Arrays.asList(1,2,3,4,5,6,7,8,9,10);
7.         list.stream()
8.             .skip(5)
9.             .forEach(System.out::println);
10.     }
11. }
```

例 実行結果

```
> java SkipSample
6
7
8
9
10
```

なお、limitメソッドとskipメソッドは、並列ストリームを扱っていた場合には、基本的にどの要素が絞り込まれるかを制御することはできません。

試験対策

ストリーム内の要素を絞り込むメソッドには、条件を指定するfilterメソッド、重複を取り除くdistinctメソッド、数で絞り込むlimitメソッドやskipメソッドがあります。

16. A、B

➡P161

ストリーム内の要素を1つ戻すための**終端操作**が、**findAnyメソッド**と**findFirstメソッド**です。どちらも基本的には、ストリーム内の最初の要素を持ったOptionalのインスタンスへの参照を戻します。そのため、次の2つのコードは共に、「A」をコンソールに表示します。

```
String[] array = {"A", "B", "C"};
Stream<String> stream = Arrays.stream(array);
Optional<String> result = stream.findAny();
result.ifPresent(System.out::println);
```

```
String[] array = {"A", "B", "C"};
Stream<String> stream = Arrays.stream(array);
Optional<String> result = stream.findFirst();
result.ifPresent(System.out::println);
```

2つのメソッドの違いは、**並列処理時**に現れます。findAnyメソッドは、常に同じ要素を返す保証はありません。そのため、並列処理をしたときには、並列で処理されているうちのどれかが戻り値として戻されることになります。たとえば、次のコードを実行すると、1〜5の値がランダムに表示されます。なお、筆者の環境では、3が圧倒的に多く表示されましたが、これも固定ではなく、何度か実行すると変化があります。

```
List<Integer> list = List.of(1,2,3,4,5);
list.parallelStream()
    .findAny()
    .ifPresent(System.out::println);
```

一方のfindFirstメソッドでは、ストリームの最初の要素が戻されます。そのため、コンソールには常に最初の要素である「1」が表示されます。

```
List<Integer> list = List.of(1,2,3,4,5);
list.parallelStream()
    .findFirst()
    .ifPresent(System.out::println);
```

もし、**filterメソッド**などでストリームの要素が絞られたときには、その条件に一致する要素の中で、先頭にあるものが戻されます。たとえば、前述のコードを次のように修正するとストリーム内の要素は偶数に絞られるため、ストリームの要素は2と4だけになります。

```
Predicate<Integer> p = (num) -> {
    boolean result = num % 2 == 0;
    System.out.println(num + ": " + result);
```

```
    return result;
};

List<Integer> list = List.of(1,2,3,4,5);
list.parallelStream()
    .filter(p)
    .findFirst()
    .ifPresent(System.out::println);
```

このコードを実行すると、次のような結果がコンソールに表示されます。並列ストリームなので、フィルタリングの順番は毎回同じ結果にはなりませんが、それでも、findFirstメソッドが戻すのは、条件によって絞られたストリームの先頭にある2です。

例 実行結果

```
1: false
2: true
3: false
5: false
4: true
2
```

以上のことから、選択肢**A**と**B**が正解です。findFirstメソッドは、ストリームの最初の要素を戻すものであって、最初に処理した要素を戻すわけではありません。よって、選択肢CとDは誤りです。

17. C ➡P161

sortedメソッドに関する問題です。
ストリーム内の要素を並べ替えるときは、**sortedメソッド**を使います。このメソッドは、ストリーム内の要素を自然順序に従って並べ替えます。たとえば次のコード例では、4つの文字列を要素として持つリストからストリームを作り、要素を並べ替えて、その結果をコンソールに表示しています。

例 sortedメソッドの使用法

```
1. import java.util.Arrays;
2. import java.util.List;
3.
4. public class SortSample {
5.     public static void main(String[] args) {
6.         List<String> list = Arrays.asList("A","B","C","A");
7.         list.stream()
8.             .sorted()
9.             .forEach(System.out::println);
10.     }
11. }
```

例 実行結果

```
> java SortSample
A
A
B
C
```

この例のアルファベット順のような並べ替え可能なものであればよいのですが、たとえば、次のItemクラスのような自然に並び順が決まらないような場合は、どうしたらよいでしょうか。

例 自然順序で並べ替えができないクラスの例（Itemクラス）

```
1. class Item {
2.     private String name;
3.     public Item(String name) {
4.         this.name = name;
5.     }
6.     public String getName() {
7.         return this.name;
8.     }
9.     @Override
10.    public String toString() {
11.        return this.name;
12.    }
13. }
```

このような場合は、**オーバーロードされたsortedメソッド**を使います。sortedメソッドは、引数を受け取らないものと、引数としてjava.util.Comparator型を受け取るものがオーバーロードされて用意されています。

例 Itemの並べ替えを行うsortedメソッド

```
1. import java.util.Arrays;
2. import java.util.List;
3.
4. public class SortSample {
5.     public static void main(String[] args) {
6.         List<Item> list = Arrays.asList(
7.             new Item("A"),
8.             new Item("C"),
9.             new Item("B")
10.         );
11.         list.stream()
12.             .sorted((a, b) -> a.getName().compareTo(b.getName()))
13.             .forEach(System.out::println);
14.     }
15. }
```

java.util.Comparatorは関数型インタフェースですので、比較対象の2つの引数を受け取り、比較した結果を整数として戻すラムダ式を扱えます。コード例では、比較対象となる2つのItem型の引数を変数aとbで受け取り、それぞれの名前を比較するためにStringクラスが持っている**compareToメソッド**の結果を戻しています。
以上のことから、選択肢Cが正解です。

18. A

➡ P162

mapメソッドに関する問題です。
ストリームの要素に対して何らかの処理を加えたいのであれば、**mapメソッド**を使います。このメソッドは、引数としてjava.util.function.Function型のラムダ式を受け取り、ストリーム内の要素を処理します。次のコード例では、ストリームの要素である文字列の**toUpperCaseメソッド**を呼び出して、すべて大文字に変更しています。

例 mapメソッドの使用例

```
1. import java.util.Arrays;
2.
3. public class MapSample {
4.     public static void main(String[] args) {
5.         String[] fruits = {"apple", "orange", "banana"};
6.         Arrays
7.             .stream(fruits)
8.             .map(f -> f.toUpperCase())
9.             .forEach(System.out::println);
10.     }
11. }
```

例 実行結果

```
> java MapSample
APPLE
ORANGE
BANANA
```

このメソッドのポイントは、入力された要素の集合の型と、処理を施した結果を表す出力の集合の型を必ずしも一致させる必要がない点です。前述の文字列を大文字にするコード例では、入力された要素の型はString、出力された要素の型もStringでした。しかし、このメソッドが引数として受け取る関数型インタフェースFunctionは、引数と戻り値を別の型で指定することができるため、戻り値型の型パラメータを引数型とは異なるものにすれば、ストリームの要素の型を変えられるのです。

```
@FunctionalInterface
public interface Function<T,R> {
    R apply(T t);
}
```

そのため、次のコード例のようにString型の引数を受け取り、int型（正確にはボクシングされてInteger型）の戻り値を戻すことができます。このコードでは、Stringクラスの**lengthメソッド**を使って、文字数を数えて戻しています。

例 mapメソッドの使用例（引数と戻り値を別の型で指定）

```
1. import java.util.Arrays;
2.
3. public class MapSample {
4.     public static void main(String[] args) {
5.         String[] fruits = {"apple", "orange", "banana"};
6.         Arrays
```

```
 7.             .stream(fruits)
 8.             .map(f -> f.length())
 9.             .forEach(System.out::println);
10.     }
11. }
```

mapメソッドで指定された処理を実行したあとは、ストリーム内の要素はString型の集合からInteger型の集合に変換されることになります。よって、コンソールには、次のように文字数が表示されます。

例 実行結果

```
> java MapSample
5
6
6
```

このように、ストリーム内の要素への処理を加えるだけでなく、ストリーム・パイプラインで扱う型の変換を行うためにmapメソッドを使うことができます。以上のことから、選択肢**A**が正解です。これを応用すると、元のコレクションから新しいコレクションを作り出すためにmapメソッドを活用できるようになります。

19. A ➡P162

リダクション操作を行うメソッドに関する問題です。
終端操作にはforEachメソッドだけでなく、「リダクション操作」と呼ばれる種類の処理があります。**リダクション操作**とは、一連の要素を1つに「まとめる」操作のことです。たとえば、リスト内にある数値の合計を求める処理は、リダクション操作の1つです。ほかにも、リスト内の要素に何らかの処理を加え、その結果を別のリストにまとめるのもリダクション操作といえます。

reduceメソッドはリダクション操作の1つで、値を累積的に結合していく関数を実行するためのものです。たとえば次のコード例は、reduceメソッドを使って1〜5の値を持つリストからストリームを作り、リダクション操作を実行しています。

例 reduceメソッドの使用例

```
 1. import java.util.Arrays;
 2. import java.util.List;
 3. import java.util.Optional;
 4.
 5. public class ReduceSample {
 6.     public static void main(String[] args) {
 7.         List<Integer> list = Arrays.asList(1,2,3,4,5);
 8.         Optional<Integer> result =
 9.                 list.stream().reduce((a, b) -> a + b);
10.         result.ifPresent(System.out::println);
11.     }
12. }
```

reduceメソッドはBinaryOperator型の引数を受け取るため、このコード例では、2つの数値を加算して戻しています。結果が戻ってくれば、その値と次の値をラムダ式の引数に渡してさらに合算して戻します。こうして、次の値がなくなるまで加算し続けます。そのため、このコードを実行すると、次のように1＋2＋3＋4＋5の合計値がコンソールに表示されます。なお、このreduceメソッドの戻り値型はOptional<T>です。

例 実行結果

```
> java ReduceSample
15
```

計算の経過を確認するために、ラムダ式で途中の値を表示するとわかりやすいでしょう。次のコード例は、先ほどのReduceSampleクラスを修正したものです。

例 計算の経過を確認できるように修正したコード

```
 7.         List<Integer> list = Arrays.asList(1,2,3,4,5);
 8.         Optional<Integer> result =
 9.                 list.stream().reduce((a, b) -> {
10.                     int c = a + b;
11.                     System.out.print("a = " + a);
12.                     System.out.print(", b = " + b);
13.                     System.out.println(", a + b = " + c);
14.                     return c;
15.                 });
16.         result.ifPresent(System.out::println);
```

※次ページに続く

このコード例を実行すれば、次のように前の計算結果が次の計算に使われながら加算処理が行われているのがわかります。

例 実行結果

```
> java ReduceSample
a = 1, b = 2, a + b = 3
a = 3, b = 3, a + b = 6      ←前の計算結果が利用されている
a = 6, b = 4, a + b = 10     ←前の計算結果が利用されている
a = 10, b = 5, a + b = 15    ←前の計算結果が利用されている
15
```

なお、reduceメソッドはオーバーロードされており、初期値を受け取るものも用意されています。たとえば、次のコード例ではメソッドの第1引数に初期値として100という数値を渡しています。メソッドの第2引数には、これまでと同じようにBinaryOperator型のラムダ式を受け取ります。

例 初期値を受け取るreduceメソッドの使用例

```
7.          List<Integer> list = Arrays.asList(1,2,3,4,5);
8.          int result =
9.                  list.stream().reduce(100, (a, b) -> {
10.                     int c = a + b;
11.                     System.out.print("a = " + a);
12.                     System.out.print(", b = " + b);
13.                     System.out.println(", a + b = " + c);
14.                     return c;
15.                 });
16.         System.out.println(result);
```

なお、初期値を受け取らないreduceメソッドの戻り値型はOptional<T>でしたが、この初期値を受け取るタイプのreduceメソッドの戻り値型は、初期値として渡したデータと同じ型になります。このコード例では、100というint型の値を初期値として渡しているため、戻り値型もint型になります。

設問のコードでは、第1引数にint型の値、第2引数にラムダ式を渡しています。そのため、戻り値はint型となります。よって、選択肢**A**が正解です。選択肢Dもボクシングが働くためコンパイルエラーとはなりませんが、第1引数と戻り値が同じ型の選択肢Aのほうが適切です。よって、選択肢Dは誤りです。

20. C

➡ P162

リダクション操作を行うmaxメソッドに関する問題です。
前問で解説したreduceメソッドは、要素を次々と結合していくリダクション操作でした。本設問で使っているのは、リダクション操作の1つであるmaxメソッドです。このメソッドで行うのは、複数の要素から最大の要素に結果をまとめるリダクション操作です。なお、最小の要素に結果をまとめるリダクション操作のためには、minメソッドが用意されています。

ストリーム内の要素を指定されたjava.util.Comparatorで並べ替え、最大を戻すのが**maxメソッド**、最小を戻すのが**minメソッド**です。次のコード例では、maxメソッドを使って最大値を求めています。なお、Comparatorは関数型インタフェースの1つであるため、compareメソッドに相当するラムダ式をmaxメソッドの引数として渡しています。

例 maxメソッドの使用例

```
1. import java.util.Arrays;
2. import java.util.List;
3. import java.util.Optional;
4.
5. public class MaxSample {
6.     public static void main(String[] args) {
7.         List<Integer> list = Arrays.asList(98,99,96,97,95);
8.         Optional<Integer> result =
9.                 list.stream().max((a, b) -> {
10.                    if (a == b) return 0;
11.                    if (a < b) return -1;
12.                    return 1;
13.                });
14.        result.ifPresent(System.out::println);
15.    }
16. }
```

Comparatorの**compareメソッド**では、並べ替えのために2つの引数を比較します。最初の引数が2番目の引数より小さい場合は負の整数、両方が等しい場合は0、最初の引数が2番目の引数より大きい場合は正の整数を返すようにします。このコードを実行すると、ストリーム内の数値が小さい順に並び、最大値を持ったOptionalクラスのインスタンスへの参照が戻されます。このコード例では95〜99までの値のリストを扱っているため、最大値である99がコンソールに表示されます。

例 実行結果

```
> java MaxSample
99
```

最小値を求めるには、minメソッドを使います。並べ替えのルールは同じですので、ラムダ式は変更せず、maxメソッドからminメソッドに変えたものが次のコード例です（クラス名をMinSampleに変更しています）。

例 minメソッドの使用例

```
7.      List<Integer> list = Arrays.asList(98,99,96,97,95);
8.      Optional<Integer> result =
9.              list.stream().min((a, b) -> {
10.                 if (a == b) return 0;
11.                 if (a < b) return -1;
12.                 return 1;
13.             });
14.     result.ifPresent(System.out::println);
```

このコードを実行すると、最小値である95がコンソールに表示されます。

例 実行結果

```
> java MinSample
95
```

設問のコードでは、文字列を並べ替えています。並べ替えはStringクラスのcompareToメソッドを使っているため、自然順序（文字順）によって行われます。そのため、B、A、D、Cの4つの文字列はA、B、C、Dの順に並べ替えられ、その中で順番が最後のものにまとめられます。そのため、コンソールにはDが表示されます。以上のことから、選択肢**C**が正解です。

21. A ➡ P163

Collectorインタフェースに関する問題です。
次のコード例のように、forEachメソッドを使ってストリームの文字列をカンマ区切りの文字列として連結する繰り返し処理があったとします。

例 ストリームの文字列を連結する繰り返し処理の例

```
1. import java.util.Arrays;
2. import java.util.List;
3.
4. public class CollectorSample {
5.     public static void main(String[] args) {
6.         List<String> list =
7.                 Arrays.asList("A", "B", "C", "D", "E");
8.         StringBuilder builder = new StringBuilder();
9.         list.stream().forEach(str -> {
10.            if (builder.length() != 0) {
11.                builder.append(",");
12.            }
13.            builder.append(str);
14.        });
15.        System.out.println(builder);
16.    }
17. }
```

このコードでは、処理される途中の文字列を蓄積していくために**java.lang.StringBuilder**を使っています。StringBuilderが空でなければカンマ「,」を追加し、その後ろに文字列を追加することで、最初の文字列の前にカンマが付かないようにif文で制御しています。そのため、このコードを実行すると、次のようにカンマ区切りの文字列がコンソールに表示されます。

例 実行結果

```
> java CollectorSample
A,B,C,D,E
```

上記のように、このコード例は正常に動作しますが、次のような問題があります。

- ラムダ式内部から、外部のオブジェクトを操作している
- 同じように文字列のストリームからカンマ区切りの文字列を作る処理を行いたくても再利用できない

1つ目の**ラムダ式内部から、外部のオブジェクトにアクセス**する場合、その外部オブジェクトは**実質的にfinal**として扱われます。ストリーム・パイプライン内のラムダ式であっても同様です。もし、（あまり意味はありませんが）

次のように変数builderの参照先を変更しようとすると、コンパイラエラーとなります。

```
list.stream().forEach(str -> {
    builder = new StringBuilder();    ←コンパイルエラー
    if (builder.length() != 0) {
        builder.append(",");
    }
    builder.append(str);
});
```

このように変数が保持している参照を変更していないかどうかはコンパイラがチェックしますが、参照先のオブジェクトが持つ値を変更しているかどうかまではチェックしません。そのため、builderのappendメソッドで、内部の文字列を変更していてもコンパイルエラーにはなりません。

```
list.stream().forEach(str -> {
    if (builder.length() != 0) {
        builder.append(",");
    }
    builder.append(str);
});
```

これでは参照先は変わっていないものの、結果としてラムダ式が実行される前と後では異なるオブジェクトになってしまっていることになり、実質的なfinalであるとはいえません。そのため、このようなストリーム外部のオブジェクトの状態（内部的な値）を変更するような処理のことを「**副作用のある処理**」と呼びます。
ラムダ式でこのような副作用のある処理を避けるべきなのは、ラムダ式は定義したタイミングで実行されるのではなく、ラムダ式が代入された関数型インタフェースのメソッドを呼び出したときに実行される「**遅延実行**」という特性があるからです。
たとえば次のコード例のように、ラムダ式は、変数に入れてほかのメソッドなどに渡すことができるため、どのタイミングで実行されるかはラムダ式を定義したメソッド側では制御できません。

```
1. import java.util.function.Consumer;
2.
3. public class LazyExecute {
4.     public static void execute(Consumer<String> c) {
5.         c.accept("Hello");    ←このタイミングでラムダ式が実行
6.     }
7. }
```

このようにラムダ式が入った関数型オブジェクトがやり取りされる場合、次のコード例のように、ラムダ式外部のリストに要素を追加するようなコードを記述すると、ラムダ式が実行されたタイミングで結果がどうなるかは予測がつかないのです。

```
1. import java.util.ArrayList;
2. import java.util.List;
3. import java.util.function.Consumer;
4.
5. public class Test {
6.     public static void main(String[] args) {
7.         List<String> list = new ArrayList<>();
8.         Consumer<String> c = str -> {
9.             list.add(str);    ←ここで外部オブジェクトも変更
10.        };
11.        LazyExecute.execute(c);    ←ここで何が行われるかわからない
12.        System.out.println(list.size());
13.    }
14. }
```

以上のことから、このような副作用のある処理は記述すべきではありません。

2つ目の「**処理を再利用できない**」という問題は、forEachメソッドにラムダ式を直接記述しているからです。これは次のようにストリームを作る前に、処理内容を記述したConsumer型のラムダ式を用意することで解決できます。

```
Consumer<String> c =
        str -> {
            if (builder.length() != 0) {
                builder.append(",");
            }
            builder.append(str);
```

```
        };
list.stream().forEach(c);
System.out.println(builder);
```

しかし、これだけでは1つ目の「副作用のある処理」の問題が解決できません。そこで利用するのが、**java.util.stream.Collectorインタフェース**です。Collectorは、ストリームのリダクション操作のためのインタフェースで、処理の途中のオブジェクトを適切に扱うことができます。以上のことから、選択肢**A**が正解です。
選択肢BのStreamは、ストリームそのものを扱うためのインタフェースです。選択肢CのStreamSupportは、ストリーム用のライブラリ開発のためのクラスです。選択肢DのStream.Builderは、ストリームを生成するためのインタフェースです。よって、これらの選択肢は誤りです。

22. B

➡ P163

Collectorインタフェースのメソッドに関する問題です。
java.util.stream.Collector<T, A, R>インタフェースは、**3つの型パラメータ**を受け取ります。1つ目がストリーム内の要素の型、2つ目が処理途中の値を保持するためのオブジェクト、3つ目が最終的な結果の型です。

前問のカンマ区切りの文字列を作るコードを置き換えるために、Collectorインタフェースを実現したSampleCollectorクラスを定義してみましょう。このクラスで扱うのは文字列ストリームですので、1つ目の型パラメータはString型です。処理途中のカンマ区切りの文字列を蓄積するのはStringBuilderですので、2つ目の型パラメータはStringBuilderとなります。3つ目の型パラメータは、StringBuilderに蓄えた文字列を結果として戻したいのでString型となります。

例 Collectorインタフェースを実現したクラスの定義

```
1. import java.util.stream.Collector;
2.
3. public class SampleCollector
4.     implements Collector<String, StringBuilder, String> {
5. }
```

Collectorには、supplier、accumulator、combiner、finisher、characteristicsという5つの抽象メソッドがあり、これらすべてを実装しなければいけません。
supplierメソッドは、処理途中の値を保持するためのオブジェクトを生成するメソッドです。このメソッドは、引数は受け取りませんが、戻り値はjava.util.function.Supplier型のラムダ式でなければいけません。次のコードでは、ラムダ式の代わりにメソッド参照を使って、StringBuilderのインスタンスを生成して戻すSupplier型の式として戻しています。

例 supplierメソッドの実装例

```
@Override
public Supplier<StringBuilder> supplier() {
    return StringBuilder::new;
}
```

accumulatorメソッドは、具体的に実行したい処理を記述したBiConsumer型のラムダ式を戻すメソッドです。前問のコードであれば、forEachメソッドに記述していたStringBuilderに、カンマ区切りの文字列を蓄積していく処理を記述します。このメソッドが戻すBiConsumer型のラムダ式は、処理途中のカンマ区切りの文字列を蓄えるためのStringBuilder型と、ストリーム内の要素であるString型のジェネリクスの型パラメータとして受け取ります。次のコード例では、builderとstrという変数でStringBuilder型とString型の引数をそれぞれ受け取っています。

例 accumulatorメソッドの実装例

```
@Override
public BiConsumer<StringBuilder, String> accumulator() {
    return (builder, str) -> {
        if (builder.length() != 0) {
            builder.append(",");
        }
        builder.append(str);
    };
}
```

以上のことから、選択肢**B**が正解です。

combinerメソッドについては後述しますので、ここではいったんnullを戻しておきます。

例 combinerメソッドの実装例

```
@Override
public BinaryOperator<StringBuilder> combiner() {
```

```
    return null;
}
```

finisherメソッドは、処理結果を戻すラムダ式を提供するメソッドです。このコードでは、カンマ区切りの文字列を蓄えたStringBuilderを引数に受け取り、String型の戻り値を戻すjava.util.function.Function型のラムダ式を戻しています。

例 finisherメソッドの実装例

```
@Override
public Function<StringBuilder, String> finisher() {
    return builder -> builder.toString();
}
```

最後は**characteristicsメソッド**です。このメソッドは、Collectorの特徴を表すEnumのセットを戻すメソッドです。Collectorの特徴には次のような種類があり、Collectorクラスの内部Enumとして**Collector.Characteristics**が用意されています。

【Collector.Characteristicsの列挙子】

列挙子	概要
CONCURRENT	このCollectorが並行処理をすることを表す
IDENTITY_FINISH	このCollectorのfinisherメソッドが省略可能であることを表す
UNORDERED	コレクションの操作において順序の維持を保証しないことを表す

もし、これらを指定する必要がなければ、次のコードのように**java.util.EnumSetクラスのnoneOfメソッド**にCollector.Characteristicsクラスのクラスリテラル（クラスの定義情報を表すリテラル）を渡せば、Collector.Characteristics型を扱う空のSetオブジェクトが生成されます。

例 characteristicsメソッドの実装例

```
@Override
public Set<Characteristics> characteristics() {
    return EnumSet.noneOf(Characteristics.class);
}
```


これまでのコードをまとめると、次のようなCollectorの実現クラスが定義できます。

例 メソッドの実装部分を追加したSampleCollectorクラス

```
 1. import java.util.EnumSet;
 2. import java.util.Set;
 3. import java.util.function.BiConsumer;
 4. import java.util.function.BinaryOperator;
 5. import java.util.function.Function;
 6. import java.util.function.Supplier;
 7. import java.util.stream.Collector;
 8.
 9. public class SampleCollector
10.     implements Collector<String, StringBuilder, String> {
11.     @Override
12.     public Supplier<StringBuilder> supplier() {
13.         return StringBuilder::new;
14.     }
15.     @Override
16.     public BiConsumer<StringBuilder, String> accumulator() {
17.         return (builder, str) -> {
18.             if (builder.length() != 0) {
19.                 builder.append(",");
20.             }
21.             builder.append(str);
22.         };
23.     }
24.     @Override
25.     public BinaryOperator<StringBuilder> combiner() {
26.         return null;
27.     }
28.     @Override
29.     public Function<StringBuilder, String> finisher() {
30.         return builder -> builder.toString();
31.     }
32.     @Override
33.     public Set<Characteristics> characteristics() {
34.         return EnumSet.noneOf(Characteristics.class);
35.     }
36. }
```

それでは、前述のコードをCollectorを使ったものに置き換えてみましょう。ストリームでCollectorを扱うには、次の修正例のように**collectメソッド**を使います。このコードでは、collectメソッドの引数に先ほど定義したSampleCollectorのインスタンスへの参照を渡しています。

例 解答21で作成したCollectorSampleクラスの修正（collectメソッド）

```
import java.util.Arrays;
import java.util.List;
import java.util.function.Consumer;

public class CollectorSample {
    public static void main(String[] args) {
        List<String> list =
                Arrays.asList("A", "B", "C", "D", "E");

        String result =
            list.stream().collect(new SampleCollector());
        System.out.println(result);
    }
}
```

ラムダ式で行っていた処理をSampleCollectorクラスに移動したので、コード量が減り、さらに文字列ストリームからカンマ区切りの文字列を作る処理を再利用できるようになりました。また、前問の解説で問題点の1つとしていた「副作用のある処理」もありません。このコードを実行すれば、次のようにカンマ区切りの文字列がコンソールに表示されるはずです。

例 実行結果

```
> java CollectorSample
A,B,C,D,E
```

これで、SampleCollectorクラスが完成したかのように思えますが、一点やり残していることがあります。それがcombinerメソッドの実装です。まず、このメソッドが実装されないことによって起こる状態を確認しましょう。CollectorSampleクラスを次のように並列ストリームを扱うように変更して、コンパイル、実行してみます。

例 CollectorSampleクラスの修正（parallelStreamメソッド）

```
        List<String> list =
                Arrays.asList("A", "B", "C", "D", "E");

        String result =
                list.parallelStream()
                .collect(new SampleCollector());
        System.out.println(result);
```

実行すると、次のように例外がスローされてしまいます。これは、先ほど「ひとまずnullを戻す」という実装をしたcombinerメソッドが原因です。

例 実行結果

```
> java CollectorSample
Exception in thread "main" java.lang.NullPointerException:
Cannot invoke "java.util.function.BinaryOperator.apply(Object,
Object)" because "this.val$combiner" is null

… 以下省略 …
```

combinerメソッドの目的は、並列処理をしているとき、個々に作られた処理途中の値を保持するためのオブジェクトを結合することです。並列ストリームによる並列処理では、元のコレクションが分割されて、複数のストリーム・パイプラインで処理されます。そのため、今回の例であればA、B、C、D、Eの文字列はいくつかに分割されて並列処理されるため、StringBuilderも複数生成されます。これらを結合して1つのStringBuilderにしてから、最後のfinisherメソッドを実行するわけです。この結合方法を指定するのがcombinerメソッドですが、これが**nullを戻した**ために、**NullPointerException**がスローされたわけです。
そこで、次のように2つのStringBuilderを受け取り、結合する処理をcombinerメソッドに実装します。なお、combinerメソッドの戻り値型は、結合対象であるオブジェクト2つを引数として受け取るBinaryOperator型です。

例 combinerメソッドの実装例

```
@Override
public BinaryOperator<StringBuilder> combiner() {
    return (a, b) ->{
        a.append(b);
```

```
        return a;
    };
}
```

combinerメソッドの実装ができたら、コンパイルして、実行してみましょう。今回は例外がスローされませんが、カンマ区切りの文字列にはなっていません。これは、A、B、C、D、Eの5つの文字列が、5つのストリーム・パイプラインに分割されて実行されたため、各ストリーム・パイプラインのaccumulatorメソッドでif文の条件を満たさず、カンマが追加されないために起こる現象です（並列処理なので、環境によって同じ結果にはならないかもしれません）。

例 実行結果

```
> java CollectorSample
ABCDE
```

そこで、combinerメソッドでStringBuilderを結合する際にもカンマを追加するように修正してみます。

例 combinerメソッドの実装例

```
@Override
public BinaryOperator<StringBuilder> combiner() {
    return (a, b) -> {
        if (a.length() != 0) {
            a.append(",");
        }
        a.append(b);
        return a;
    };
}
```

今度は、カンマ区切りの文字列が正しくコンソールに表示されるようになりました。

例 実行結果

```
> java CollectorSample
A,B,C,D,E
```


23. B

➡ P164

Collectorsクラスに関する問題です。

この解答23から、解答24、25までの解説では、以下の列挙型ItemTypeとItemクラスを利用します。

例 列挙型ItemType

```
1. public enum ItemType {
2.     BOOK, MAGAZINE, DVD
3. }
```

例 Itemクラス

```
1. public class Item {
2.     private int id;
3.     private ItemType type;
4.     private String name;
5.     private int price;
6.
7.     public Item(int id, ItemType type, String name, int price) {
8.         super();
9.         this.id = id;
10.        this.type = type;
11.        this.name = name;
12.        this.price = price;
13.    }
14.
15.    public int getId() {
16.        return id;
17.    }
18.
19.    public ItemType getType() {
20.        return type;
21.    }
22.
23.    public String getName() {
24.        return name;
25.    }
26.
```

```
27.     public int getPrice() {
28.         return price;
29.     }
30.
31.     @Override
32.     public String toString() {
33.         return "Item [id=" + id + ", type=" + type
34.                 + ", name=" + name + ", price=" + price + "]";
35.     }
36. }
```

前問の例のように、文字列のリストからカンマ区切りの文字列を作りたいというような特別な要件がない場合、Collectorの実現クラスを毎回用意するのではなく、**java.util.stream.Collectorsクラス**を使いましょう。CollectorsはCollectorの実現クラスで、要素をコレクションに蓄積したり、さまざまな条件に従って要素を要約したりすることができる便利なクラスです。
たとえば、リストから作成したストリームをフィルタリングして得た結果のみで構成された新しいリストを作るには、次のコード例のようにCollectorsクラスの**toListメソッド**を使います。

例 toListメソッドの使用例

```
1. import java.util.Arrays;
2. import java.util.List;
3. import java.util.stream.Collectors;
4.
5. public class CollectorsSample {
6.     public static void main(String[] args) {
7.         List<Item> list = Arrays.asList(
8.             new Item(1, ItemType.BOOK, "Java", 1980),
9.             new Item(2, ItemType.BOOK, "Lambda", 2980),
10.            new Item(3, ItemType.MAGAZINE, "Software", 980),
11.            new Item(4, ItemType.MAGAZINE, "Test", 1280)
12.        );
13.
14.        List<Item> books =
15.            list.stream()
16.                .filter(item ->
17.                    item.getType() == ItemType.BOOK)
18.                .collect(Collectors.toList());
19.
20.            books.forEach(System.out::println);
21.    }
22. }
```

このメソッドを使えば、新しいリストの詰め替え処理が簡単に実装できます。このコード例では、ItemTypeをBOOKだけに絞り込んだ新しいリストを作っています。コードを実行した結果は次のとおりです。

例 実行結果

```
> java CollectorsSample
Item [id=1, type=BOOK, name=Java, price=1980]
Item [id=2, type=BOOK, name=Lambda, price=2980]
```

以上のことから、選択肢**B**が正解です。

なお、java.util.Listだけでなくjava.util.Setを扱うこともできます。その場合には、次のように**toSetメソッド**を使います。

```
Set<Item> books =
    list.stream()
        .filter(item ->
            item.getType() == ItemType.BOOK)
        .collect(Collectors.toSet());
```

ほかにも、Mapを扱うこともできます。次のコード例では、**toMapメソッド**を使って、リストから名前をキーに、インスタンス（への参照）を値にしたMapを作っています。

例 toMapメソッドの使用例

```
1. import java.util.Arrays;
2. import java.util.List;
3. import java.util.Map;
4. import java.util.stream.Collectors;
5.
6. public class CollectorsSample {
7.     public static void main(String[] args) {
```

```
 8.         List<Item> list = Arrays.asList(
 9.             new Item(1, ItemType.BOOK, "Java", 1980),
10.             new Item(2, ItemType.BOOK, "Lambda", 2980),
11.             new Item(3, ItemType.MAGAZINE, "Software", 980),
12.             new Item(4, ItemType.MAGAZINE, "Test", 1280)
13.         );
14.
15.         Map<String, Item> map =
16.                 list.stream().collect(Collectors.toMap(
17.                     Item::getName,
18.                     item -> item));
19.
20.         map.keySet().stream().forEach(System.out::println);
21.     }
22. }
```

なお、このメソッドはこれまでのtoListやtoSetとは違い、**引数を2つ**受け取ります。1つ目が**キー**を取り出すためのjava.util.function.Function型のラムダ式、2つ目が**値**を取り出すためのjava.util.function.Function型のラムダ式です。このコード例では、メソッド参照を使ってItemクラスのgetNameメソッドの戻り値（つまり名前）をキーに、引数で受け取ったItem型のitemをそのまま戻すラムダ式で、値の取り出し方を指定しています。コードを実行すれば、名前だけの一覧がコンソールに表示されるはずです。

例 実行結果

```
> java CollectorsSample
Java
Test
Software
Lambda
```

試験対策

Collectorsは、要素をコレクションに蓄積したり、さまざまな条件に従って要素を要約するクラスです。toList、toSet、toMapなどのメソッドを使うと、ストリームを操作して得た結果で構成されたコレクションを作ることができます。List、Set、Mapの各コレクションの特徴とあわせて、使い方を覚えましょう。

24. C

➡ P164

CollectorsクラスのgroupingByメソッドに関する問題です。

本設問の解説では、解答23で挙げたItemとItemTypeを利用しています。必要に応じて参照してください。
java.util.stream.Collectorsクラスの機能は、ストリームから要素を詰め替えた新しいコレクションを作るだけではありません。便利な機能にグルーピングがあります。次のコードは、ItemクラスのtypeフィールドによってグルーピングしたIt例です。

例 groupingByメソッドの使用例

```
 1. import java.util.Arrays;
 2. import java.util.List;
 3. import java.util.Map;
 4. import java.util.stream.Collectors;
 5.
 6. public class GroupingSample {
 7.     public static void main(String[] args) {
 8.         List<Item> list = Arrays.asList(
 9.             new Item(1, ItemType.BOOK, "Java", 1980),
10.             new Item(2, ItemType.BOOK, "Lambda", 2980),
11.             new Item(3, ItemType.MAGAZINE, "Software", 980),
12.             new Item(4, ItemType.MAGAZINE, "Test", 1280)
13.         );
14.
15.         Map<ItemType, List<Item>> group =
16.                 list.stream().collect(
17.                     Collectors.groupingBy(Item::getType));
18.
19.         System.out.println(group);
20.     }
21. }
```

このコード例で扱っているリストは、ItemTypeがBOOKであるものが2つ、MAGAZINEであるものが2つの、計4つのItemのコレクションです。このコレクションをBOOKとMAGAZINEに分け、2つのリストを作るのが**Collectorsクラスのgroupingbyメソッド**です。このコードを実行すると、ItemTypeをキーにしたMapが戻され、コンソールには次のように表示されます。このことから、ItemクラスのTypeフィールドの値でグルーピングされ、グループごとに別の

リストが作られたことがわかります。

例 実行結果

```
> java GroupingSample
{
    BOOK=[
        Item [id=1, type=BOOK, name=Java, price=1980],
        Item [id=2, type=BOOK, name=Lambda, price=2980]
    ],
    MAGAZINE=[
        Item [id=3, type=MAGAZINE, name=Software, price=980],
        Item [id=4, type=MAGAZINE, name=Test, price=1280]
    ]
}
```

このコード例で使用しているCollectorsクラスのgroupingByメソッドは、java.util.function.Function型のラムダ式を受け取ります。Functionは、ジェネリクスの型パラメータとして、引数の型と戻り値の型の2つの型を受け取ります。groupingByメソッドの場合は、引数として**ストリームの要素の型**、戻り値型は**ラムダ式が戻す型**になります。前述のコード例ではメソッド参照を使っていますが、引数の型はItem型、戻り値の型はItemクラスのgetTypeメソッドが戻すItemType型になります。なお、メソッド参照を使わずにラムダ式で記述すると、次のようになります。

```
Map<ItemType, List<Item>> group =
        list.stream().collect(
            Collectors.groupingBy(
                item -> item.getType()
            ));
```

CollectorsクラスのgroupingByメソッドは、java.util.Map型の戻り値を戻します。マップのキーになるのが、ラムダ式が戻す型です。コード例では、ItemクラスのgetTypeメソッドの戻り値型なので、ItemType型となります。マップのバリューになるのが、グルーピングされた要素のリストであるjava.util.List型となります。コード例では、BOOKとMAGAZINEの2つのItemTypeが使われていたので、2つのリストが作られます。

設問のコードでは、CollectorsクラスのgroupingByメソッドで、EmployeeのgetDeptメソッドの戻り値を使っていることから、Departmentごとに

Employeeのインスタンスをグルーピングしていることがわかります。そのため、結果のマップのキーはDepartment型、バリューはEmployeeのリストになります。以上のことから選択肢**C**が正解です。

25. D

➡ P166

Collectorsクラスのメソッドに関する問題です。

本設問の解説では、解答23で挙げたItemとItemTypeを利用しています。必要に応じて参照してください。
CollectorsクラスのgroupingByメソッドはいくつかのオーバーロードを持っています。たとえば、これまで利用してきたItemクラスでItemの種類ごとにグルーピングして、priceフィールドの合計を計算するということも可能です。このようにグルーピングして、グループ内の要素が持つ数値の合計値を計算するのが、**CollectorsクラスのsummingIntメソッド**です。なお、このメソッドはInteger型の戻り値を戻しますが、Long型やDouble型の戻り値のためには、**summingLong**や**summingDouble**といったメソッドが用意されています。

CollectorsクラスのsummingIntメソッドを使ったコードが、次の例です。このコード例では、CollectorsクラスのgroupingByメソッドの第1引数にはグルーピングの基準とするものを指定し、第2引数でsummingIntメソッドを使ってグループ内の要素のどのメソッドを使って合計を求めるのかを指定しています。なお、summingIntメソッドは、ストリームの要素を使って合計を計算するjava.util.stream.Collectorの実装を戻します。
このコード例では、第1引数にjava.util.function.Function型のラムダ式の代わりにItemクラスのgetTypeメソッドのメソッド参照を、第2引数のCollectorsクラスのsummingIntメソッドにはItemクラスのgetPriceメソッドのメソッド参照を渡しています。

例 summingIntメソッドの使用例

```
import java.util.Arrays;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

public class SummingIntSample {
    public static void main(String[] args) {
        List<Item> list = Arrays.asList(
            new Item(1, ItemType.BOOK, "Java", 1980),
            new Item(2, ItemType.BOOK, "Lambda", 2980),
```

```
11.             new Item(3, ItemType.MAGAZINE, "Software", 980),
12.             new Item(4, ItemType.MAGAZINE, "Test", 1280)
13.         );
14.
15.         Map<ItemType, Integer> group =
16.             list.stream().collect(
17.                 Collectors.groupingBy(
18.                     Item::getType,
19.                     Collectors.summingInt(Item::getPrice)));
20.
21.         System.out.println(group);
22.     }
23. }
```

このコードを実行すると、java.util.Map型のインスタンスが生成され、その参照が戻されます。マップのキーにはCollectorsクラスのgroupingByメソッドの戻り値型であるItemType型の値が、バリューにはCollectorsクラスのsummingIntメソッドの戻り値型であるInteger型の値が格納されます。そのため、コンソールには次のようにItemの種類ごとの合計額が表示されます。

例 実行結果

```
> java SummingIntSample
{BOOK=4960, MAGAZINE=2260}
```

設問のコードでは、書籍の合計金額を算出するためにsummingIntメソッドを使っているのは選択肢Dです。よって、選択肢**D**が正解です。

なお、Collectorsクラスには合計を求める**summingXXXメソッド**のほかに、平均を求める**averagingIntメソッド**や**averagingLongメソッド**、**averagingDoubleメソッド**なども用意されています。処理結果として合計を戻すか、平均を戻すかの違いがあるだけで、使い方はsummingXXXメソッドと同じで、引数の型なども同じです。

26. A ➡P167

CollectorsクラスのpartitioningByメソッドに関する問題です。
Collectorsクラスには、これまでに学習してきたgroupingByメソッド以外にも、グルーピングのためのメソッドが用意されています。それが、条件によってグループを分ける**partitioningByメソッド**です。groupingByメソッドは値の種類ごとにグルーピングしますが、このメソッドは、条件を指定して、その条件に合致したかどうかでグループを分けることができます。そのため、結果のMapには、条件に合致したグループと、合致しなかったグループの2種類が登録されます。

次のコードがpartitioningByメソッドの使用例です。このメソッドは、グルーピングするための条件式をjava.util.function.Predicate型のラムダ式で渡します。このコード例では、ItemのgetPriceメソッドを呼び出し、その結果が1000よりも大きいかどうかの真偽値を戻しています。

例 partitioningByメソッドの使用例

```
1. import java.util.Arrays;
2. import java.util.List;
3. import java.util.Map;
4. import java.util.stream.Collectors;
5.
6. public class PartitioningBySample {
7.     public static void main(String[] args) {
8.         List<Item> list = Arrays.asList(
9.             new Item(1, ItemType.BOOK, "Java", 1980),
10.            new Item(2, ItemType.BOOK, "Lambda", 2980),
11.            new Item(3, ItemType.MAGAZINE, "Software", 980),
12.            new Item(4, ItemType.MAGAZINE, "Test", 1280)
13.        );
14.
15.        Map<Boolean, List<Item>> map =
16.            list.stream().collect(
17.                Collectors.partitioningBy(
18.                    item -> item.getPrice() > 1000));
19.
20.        System.out.println(map);
21.    }
22. }
```

CollectorsクラスのpartitioningByメソッドは、戻り値として**マップ**を戻します。このマップのキーは、条件に合致したかどうかを表す**trueとfalseの2種類**だけです。上記のコードでは、priceが1000よりも大きいものが3つ、1000以下のものが1つあるため、実行結果は次のようになります。

※次ページに続く

例 実行結果

```
> java PartitioningBySample
{
    false=[
        Item [id=3, type=MAGAZINE, name=Software, price=980]
    ],
    true=[
        Item [id=1, type=BOOK, name=Java, price=1980],
        Item [id=2, type=BOOK, name=Lambda, price=2980],
        Item [id=4, type=MAGAZINE, name=Test, price=1280]
    ]
}
```

このようにコレクション内の要素を、条件に一致するかどうかで分けたい場合にはpartitioningByメソッドを使います。

設問のコードでは、CollectorsクラスのpartitioningByメソッドの戻り値であるMapのキーの集合だけを取り出してコンソールに表示しています。前述のとおり、キーはBoolean型の値であるため、trueとfalseしかありません。よって、選択肢**A**が正解です。

27. D ➡P168

Streamインタフェースのpeekメソッドに関する問題です。
ストリームを構築すると、主にデバッグなどの目的で、ストリーム・パイプラインの途中でどのような状態になっているかを確認したいときがあります。そのようなときに利用するのが、**中間操作**の1つである**peekメソッド**です。たとえば、次のコード例のようにmapメソッドで値を変更し、filterメソッドで絞り込みをしたあとに、コンソールに表示するストリーム・パイプラインがあったとします。このままでは、最終的にフィルタリングされた結果は表示されますが、mapメソッドによってどのような結果になったのかはわかりません。

例 ストリーム・パイプラインの最終的な結果のみを表示する

```
1. import java.util.Arrays;
2. import java.util.List;
3.
4. public class PeekSample {
5.     public static void main(String[] args) {
6.         List<Integer> list = Arrays.asList(1,2,3,4,5);
7.         list.stream()
8.             .map(x -> x * 3)
9.             .filter(x -> x % 2 == 0)
10.            .forEach(x -> {
11.                System.out.println("value:" + x);
12.            });
13.     }
14. }
```

例 実行結果

```
> java PeekSample
value:6
value:12
```

そこで、このストリーム・パイプラインにpeekメソッドを追加してみましょう。次のコード例が、前述のコード例を改良したものです。このコードでは、mapメソッドの結果を「debug:」という接頭辞を付けてコンソールに表示しています。

例 peekメソッドの使用例

```
6.         List<Integer> list = Arrays.asList(1,2,3,4,5);
7.         list.stream()
8.             .map(x -> x * 3)
9.             .peek(x -> {
10.                System.out.println("debug:" + x);
11.            })
12.            .filter(x -> x % 2 == 0)
13.            .forEach(x -> {
14.                System.out.println("value:" + x);
15.            });
```

このコードを実行すると、次のような結果がコンソールに表示されます。

例 実行結果

```
> java PeekSample
debug:3
```

```
debug:6
value:6
debug:9
debug:12
value:12
debug:15
```

peekメソッドの結果とforEachメソッドの結果が混在しているのは、ストリーム・パイプラインに要素が1つずつ渡されているからです。たとえば、コレクションの最初の要素は1ですが、その値が3倍され、「debug:」という接頭辞を付けてコンソールに表示されたあとにフィルタリングされてしまいます。次の要素は2なので、その値が3倍されて「debug:」という接頭辞を付けてコンソールに表示されたあと、今度はフィルタリングされずに「value:」という接頭辞を付けてコンソールに表示されるという流れです。
このようにストリーム・パイプラインを1つずつ要素が通りながら、指定された中間操作を実行し、最後に終端操作につなげるのがpeekメソッドです。

設問のコードでは、banana、apple、orangeという3つの文字列を持つコレクションがストリームによって処理されます。前述のとおり、ストリーム・パイプラインには、コレクションの要素が1つずつ渡されます。
まずbananaという文字列がストリーム・パイプラインに渡され、filterメソッドによるフィルタリング、peekメソッドによる中間操作、mapメソッドによる大文字への変換、そしてpeekメソッドによる中間操作が行われて、最後にforEachメソッドでコンソールに表示されます。そのため、コンソールには「banana BANANA BANANA」と表示されます。

次に、appleがストリーム・パイプラインに渡され、filterメソッドによるフィルタリングが実行されると、その文字数が5文字よりも多くはないため、その後のストリーム・パイプラインは処理されません。よって、appleの文字は表示されません。

最後に、orangeがストリーム・パイプラインに渡されるため、bananaのときと同様に処理されて、コンソールには「orange ORANGE ORANGE」と表示されます。以上のことから、選択肢**D**が正解です。

28. D ➡P169

forEachメソッドに関する問題です。
ストリーム・パイプラインの**終端操作は、1回しか呼び出せません。**ストリーム・パイプラインとは、終端操作を実行するまでの中間操作を含めた一連の処理の流れを表しているものだからです。一度、終端操作を実行すると、そのストリーム・パイプラインは終了してしまいます。そのため、次のようなコードを実行すると、例外が発生してしまいます。

```
String[] array = {"A", "B", "C"};
Stream<String> stream = Arrays.stream(array);
stream.forEach(System.out::println);
stream.forEach(System.out::println);
```

このコードでは**終端操作**の1つである**forEachメソッド**を使い、ストリーム・パイプライン内の要素を順番にコンソールに表示しています。1回目のforEachメソッドを実行した段階でこのストリーム・パイプラインの処理は終わっているため、2回目のforEachメソッドを呼び出そうとすると、異常な状態を表す**IllegalStateException**がスローされてしまいます。

例 実行結果

```
A
B
C
Exception in thread "main" java.lang.IllegalStateException:
stream has already been operated upon or closed
    at java.base/java.util.stream.AbstractPipeline.sourceStage
Spliterator(AbstractPipeline.java:279)
    at java.base/java.util.stream.ReferencePipeline$Head.forEa
ch(ReferencePipeline.java:658)
    at Sample.main(Sample.java:11)
```

設問のコードでは、1つのストリームからcountとforEachという2つの終端操作を実行しています。そのため、実行時にIllegalStateExceptionがスローされます。よって、選択肢**D**が正解です。

forEachは終端操作のメソッドであるため、1回しか呼び出すことができません。2回呼び出そうとすると、IllegalStateExceptionがスローされます。

5

入出力

- 入出力ストリーム
- java.ioパッケージのクラス
- NIO.2
- java.nio.file.Pathインタフェース
- java.nio.file.Filesクラス
- シリアライズ

1. java.io.Fileクラスに関する説明として、正しいものを選びなさい。（1つ選択）

A. パスを扱っているだけであって、ファイルやディレクトリそのものを表すわけではない
B. ファイルは扱えるが、ディレクトリは扱えない
C. ファイルやディレクトリそのものを扱うクラスである
D. ファイルを扱うインスタンスを作ると、ディスク上にファイルが生成される

➡ P242

2. java.io.Fileクラスのメソッドで、ディレクトリ内の一覧をFileとして取得するためのものを選びなさい。（1つ選択）

A. listメソッド
B. filesメソッド
C. listFilesメソッド
D. getAllメソッド

➡ P244

3. テキストデータを読み込むためのクラスとして、正しいものを選びなさい。（1つ選択）

A. java.io.InputStream
B. java.io.Reader
C. java.io.InputReader
D. java.io.StreamReader

➡ P248

4. 写真データを扱うプログラムを開発している。より効率よく写真データを読み込むために使用するクラスとして、正しいものを選びなさい。（1つ選択）

A. java.io.FileReader
B. java.io.InputStream
C. java.io.BufferedReader
D. java.io.BufferedInputStream

➡ P252

5. 次のコードの説明として、正しいものを選びなさい。（1つ選択）

```
FileWriter out = new FileWriter("output.txt", false);
```

A. 常に新しいファイルを作成する
B. 対象のファイルが存在しなければ書き込まない
C. 追記モードで書き込む
D. 上書きモードで書き込む

➡ P255

6. java.io.BufferedWriterクラスに関する説明として、正しいものを選びなさい。（1つ選択）

A. newLineメソッドは、改行コードとしてキャリッジリターンのみを出力する
B. writeメソッドを実行すると、ファイルに文字列が出力される
C. flushメソッドはバッファとファイルを同期させる
D. BufferedWriterクラスは、効率化のために直接ファイルに出力する

➡ P257

7. java.io.BufferedInputStreamクラスに関する説明として、正しいものを選びなさい。（1つ選択）

A. readByteメソッドは、1バイトずつ読み込む
B. readNBytesメソッドは、1,024バイトずつ読み込む
C. readKBytesメソッドは、1Kバイトずつ読み込む
D. readAllBytesメソッドは、すべてのバイトを一度に読み込む

➡ P258

8. java.io.ConsoleクラスのreadPasswordメソッドの戻り値の型として、正しいものを選びなさい。（1つ選択）

A. char[]
B. String
C. CharSequence
D. InputStream

➡ P260

9. 次のプログラムをコンパイル、実行したときの結果として、正しいものを選びなさい。（1つ選択）

```
1. import java.util.Scanner;
2.
3. public class Exe {
4.     public static void main(String[] args) {
5.         Scanner scan = new Scanner("A,B,C,D,E");
6.         scan.useDelimiter(",");
7.         try(scan) {
8.             scan.next();
9.             scan.next();
10.            scan.next();
11.            scan.next();
12.            System.out.println(scan.next());
13.        }
14.    }
15. }
```

A. 「E」と表示される
B. 「null」と表示される
C. コンパイルエラーが発生する
D. 実行時に例外がスローされる

➡ P262

10. java.io.Serializableインタフェースに関する説明として、正しいものを選びなさい。（1つ選択）

A. オブジェクトをシリアライズしたいクラスが実装するインタフェースである
B. シリアライズ対象のクラスは、このインタフェースに定義されているserializeメソッドでシリアライズ処理の内容を実装しなければならない
C. このインタフェースには、何もメソッドが定義されていない
D. このインタフェースを実装していなくても、シリアライズは可能である

➡ P266

11. カスタム・シリアライズを行うときに、シリアライズ時に呼び出されるメソッドとして、正しいものを選びなさい。（1つ選択）

A. readStreamメソッド
B. readObjectメソッド
C. writeStreamメソッド
D. writeObjectメソッド

➡ P273

12. java.nio.file.Pathを使ったコードとして、コンパイルエラーになるものを選びなさい。なお、Pathsはjava.nio.file.Pathsクラス、Fileはjava.io.Fileクラス、Filesはjava.nio.file.Filesクラスとする。（1つ選択）

A. `Path path = Paths.get("dir/sample.txt");`
B. `Path path = Paths.get("dir", "sample.txt");`
C. `Path path = new File("sample.txt").toPath();`
D. `Path path = Files.get("sample.txt");`

➡ P276

13. java.nio.file.Filesクラスを使ってファイルを作成するとき、誤って既存のファイルと同じパスを指定してしまった。結果として正しいものを選びなさい。（1つ選択）

A. プログラムは正しく終了し、既存のファイルは空のファイルで上書きされる
B. プログラムは正しく終了し、既存のファイルにはそのまま何も起こらない
C. 実行時に例外がスローされる
D. 結果を表すOptionalクラスが戻される

➡ P278

14. java.nio.file.Pathとjava.nio.file.Filesを使って、新しくファイルを作成しようとしている。次のコードの説明として、正しいものを選びなさい。（1つ選択）

```
Path dir = Paths.get("a", "b");
Path file = dir.resolve(Paths.get("data.txt"));

Files.createFile(file);
```

A. カレントディレクトリ内にdata.txtができる
B. a/bディレクトリ内にdata.txtができる
C. aディレクトリ内にdata.txtができる
D. bディレクトリ内にdata.txtができる
E. b/aディレクトリ内にdata.txtができる

➡ P281

15. java.nio.file.Filesクラスの説明として、正しいものを選びなさい。（2つ選択）

A. ファイルのコピーを行うには、copyメソッドを使う
B. ファイルのコピーを行うには、copyFileメソッドを使う
C. ファイルの移動を行うには、copyメソッドに移動オプションを指定する
D. ファイルの移動を行うには、moveメソッドを使う
E. ファイルの移動を行うには、moveFileメソッドを使う

➡ P283

16. java.nio.file.Filesクラスを使い、java.io.BufferedWriterを追記モードで取得したい。正しいものを選びなさい。（1つ選択）

A.
```
Path sample = Paths.get("sample.txt");
BufferedWriter out = Files.newBufferedWriter(sample);
```

B.
```
Path sample = Paths.get("sample.txt");
BufferedWriter out
    = Files.newBufferedWriter(
                sample,
                StandardOpenOption.APPEND);
```

C.
```
Path sample = Paths.get("sample.txt");
BufferedWriter out = Files.newBufferedWriter(sample, true);
```

D.
```
Path sample = Paths.get("sample.txt");
BufferedWriter out = Files.newBufferedWriter(sample, false);
```

➡ P284

17. java.nio.file.Filesクラスのメソッドで、引数で渡したディレクトリの中にさらにサブディレクトリがあったとき、その中まですべて再帰的に探索するメソッドとして、正しいものを選びなさい。（1つ選択）

A. listメソッド
B. walkメソッド
C. findメソッド
D. listFilesメソッド

➡ P286

18. java.nio.file.FilesクラスのwalkFileTreeメソッドで利用するjava.nio.file.FileVisitorインタフェースがサポートするイベントとして、誤っているものを選びなさい。（1つ選択）

A. ディレクトリに入るとき
B. ディレクトリから出るとき
C. ファイルに遭遇したとき
D. ファイルの処理に失敗したとき
E. ファイルの処理が終了したとき

➡ P290

第5章　入出力
解　答

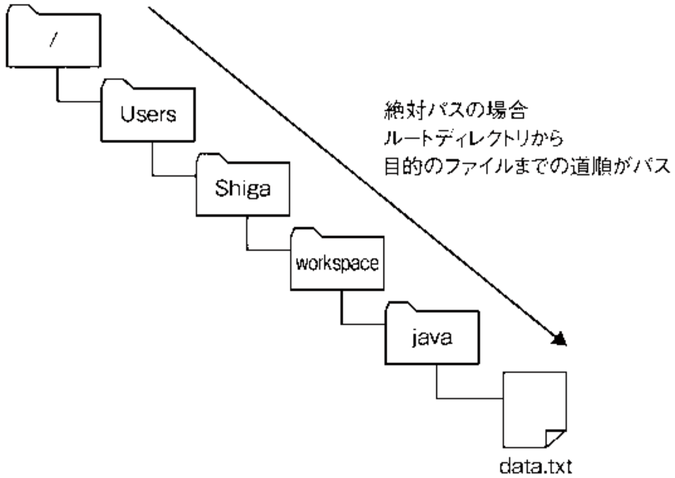
1. A ➡P236

java.io.Fileクラスに関する問題です。

java.io.Fileは、ディレクトリやファイルへのパスを表すクラスです。ディレクトリやファイルへのパスの表現には、ルートディレクトリからの絶対パスのほか、カレントのディレクトリからの相対パスを使うことができます。

注意してほしいのは、Fileクラスは「パス」を扱っているだけであって、ディレクトリやファイルそのものではない点です。たとえば、Fileのインスタンスを新しく作っても、ディスク上にファイルやディレクトリが新しく作られるわけではありません。次の図のように、パスは目的となるディレクトリやファイルまでの道順を表しているだけであって、ファイルやディレクトリそのものではないのです。

【パス】



たとえば、次のようにFileクラスのコンストラクタにファイル名やディレクトリ名を渡してインスタンスを生成すると、カレントディレクトリにあるファイル「data.txt」へのパスを扱えるようになります。

```
File file = new File("data.txt");
```


ファイル名だけでなく、絶対パスや相対パスを使ったパスをFileクラスのコンストラクタに渡せば、そのパスを扱うFileのインスタンスができます。

なお、前述のとおり、ファイルへのパスを扱えるインスタンスを作ったからといって、ファイルが作られるわけではありません。次のコード例では、Fileクラスのexistsメソッドを使って、ファイルが存在しなければ新しい空のファイルを作っています。

例 Fileクラスの使用例

```
1. import java.io.File;
2. import java.io.IOException;
3.
4. public class Sample {
5.     public static void main(String[] args) throws IOException {
6.         File file = new File("data.txt");
7.         if (file.exists() == false) {
8.             file.createNewFile();
9.         }
10.         System.out.println(file.getAbsolutePath());
11.     }
12. }
```

existsメソッドは、ファイルやディレクトリが存在するかどうかを調べるためのメソッドです。存在すればtrueを、存在しなければfalseを戻します。新しいファイルを作るには、コード例のように**createNewFileメソッド**を使います。もし、ファイルではなくディレクトリを作りたいのであれば、次のコード例のように**mkdirsメソッド**を使えば空のディレクトリが生成されます。

例 空のtestディレクトリをカレントディレクトリに作成

```
File dir = new File("test");
if (dir.exists() == false) {
    dir.mkdir();
}
```

ディレクトリの絶対パスを調べるには、コード例のように**getAbsolutePathメソッド**を使います。コード例を実行すると、次のようにルートディレクトリからの絶対パスがコンソールに表示されるはずです。

※次ページに続く

例 実行結果

```
> java Sample
/Users/shiga/workspace/java/data.txt
```

以上のことから、選択肢**A**が正解です。

2. C ➡ P236

Fileクラスの listFilesメソッドに関する問題です。
解答1では、java.io.Fileクラスはファイルだけでなく、ディレクトリも扱えることを学びました。ファイルの集合を扱うのがディレクトリの役割ですが、Fileクラスでディレクトリ内のファイルやディレクトリの一覧を取得するのが**listFilesメソッド**です。このメソッドは、ディレクトリの内容を表すFileの配列を戻り値として戻します。次のコード例は、listFilesメソッドを使ってファイルやディレクトリの一覧を表示している例です。

例 listFilesメソッドの使用例

```
1. import java.io.File;
2.
3. public class Sample {
4.     public static void main(String[] args) {
5.         File rootDir = new File(".");
6.         showTree(rootDir);
7.     }
8.
9.     private static void showTree(File dir) {
10.         File[] files = dir.listFiles();
11.         for (File file : files) {
12.             System.out.println(file.getName());
13.         }
14.     }
15. }
```

このコード例では、カレントディレクトリへのパスを表すFileのインスタンスを作って、カレントディレクトリに含まれるファイルやディレクトリの名称をコンソールに表示しています。そのため、このコードを実行すると、次のように表示されるはずです。

例 実行結果 ※実行環境によって表示結果が異なります。

```
> java Sample
bin
.classpath
.settings
.project
data.txt
src
```

このコード例では、カレントディレクトリの内容を一覧表示するだけで、内容がディレクトリであってもディレクトリ名が表示されるだけでした。これを改造して、再帰呼び出しのメソッドに変更することで、内容がディレクトリだった場合にはさらにサブディレクトリの内容を一覧表示できるようになります。

例 isDirectoryメソッドの使用例

```
1. import java.io.File;
2.
3. public class Sample {
4.     public static void main(String[] args) {
5.         File rootDir = new File(".");
6.         showTree(rootDir, "");
7.     }
8.
9.     private static void showTree(File dir, String indent) {
10.         File[] files = dir.listFiles();
11.         for (File file : files) {
12.             System.out.println(indent + file.getName());
13.             if (file.isDirectory()) {
14.                 showTree(file, indent + " ");
15.             }
16.         }
17.     }
18. }
```

この修正では、Fileの**isDirectoryメソッド**を使って、そのパスがディレクトリを表しているかどうかを調べ、もしディレクトリであれば、メソッドを再帰呼び出ししています。また、わかりやすく表示されるように、再帰呼び出しするときにはインデントを1つ追加しています。このコード例を実行する

と、次のようにディレクトリ内の内容がすべて一覧で表示されるはずです。

例 実行結果 ※実行環境によって表示結果が異なります。

```
> java Sample
bin
 ex1
  Sample.class
 ex4
  Sample.class
 ex3
  Sample.class

… 中略 …

src
 ex1
  Sample.java
 ex4
  Sample.java
 ex3
  Sample.java
 ex2
  Sample.java
 ex5
  Sample.java
```

以上のことから、選択肢**C**が正解です。なお、選択肢Aの**listメソッド**も、ディレクトリ内のファイルやディレクトリの一覧を戻しますが、このメソッドはファイル名やディレクトリ名の一覧を文字列の配列として戻します。設問は、Fileとして戻すものを選ぶ必要があるため、選択肢Aは誤りです。

listFilesメソッドはオーバーロードされており、java.io.FileFilter型の引数を受け取るlistFilesメソッドは、指定した条件に合致するファイルの一覧を戻します。**FileFilterインタフェース**は、抽象メソッドのacceptメソッドだけを持つ関数型インタフェースです。**acceptメソッド**は、File型の引数を受け取り、条件に合致するかどうかを表す真偽値を戻します。たとえば次のコードは、条件に合う内容のみに絞り込んで一覧表示するように、前述のコード例を改造したものです。

例 FileFilter型の引数を受け取るlistFilesメソッドの使用例

```
 1. import java.io.File;
 2. import java.io.FileFilter;
 3.
 4. public class Sample {
 5.     public static void main(String[] args) {
 6.         File rootDir = new File(".");
 7.
 8.         FileFilter filter = (path) -> {
 9.             if (path.isDirectory()) {
10.                 return true;
11.             }
12.             if (path.getName().endsWith(".java")) {
13.                 return true;
14.             }
15.             return false;
16.         };
17.
18.         showTree(rootDir, filter, "");
19.     }
20.
21.     private static void showTree(File dir,
22.                                  FileFilter filter,
23.                                  String indent) {
24.         File[] files = dir.listFiles(filter);
25.         for (File file : files) {
26.             System.out.println(indent + file.getName());
27.             if (file.isDirectory()) {
28.                 showTree(file, filter, indent + " ");
29.             }
30.         }
31.     }
32. }
```

このコード例では、ディレクトリもしくは拡張子が「.java」のものだけに絞り込むような条件になっています。このFileFilterをFileクラスのlistFilesメソッドの引数に渡せば、条件に合致する内容だけの配列が戻されるという仕組みです。そのため、このコード例を実行すると、次のようにディレクトリと拡張子「.java」のファイルのみの一覧が表示されるはずです。

※次ページに続く

例 実行結果 ※実行環境によって表示結果が異なります。

```
> java Sample
bin
 ex1
 ex4
 ex3
 ex2
 ex5
.settings
src
 ex1
  Sample.java
 ex4
  Sample.java
 ex3
  Sample.java
 ex2
  Sample.java
 ex5
  Sample.java
```

3. B ➡ P236

文字ストリームの入力を扱うクラスに関する問題です。
Javaでは、データの入出力を「ストリーム」という概念で扱います。**ストリーム**とは「データの流れ」という意味です。データの入出力を扱うjava.ioパッケージはJDK 1.0のころから存在しており、Java SE 8で導入された**java.util.streamパッケージで扱うストリームとは異なる概念**であることに注意してください。
ストリームを使えば、ディスク、デバイス、プログラム、メモリなど、さまざまな種類の入力元と出力先を扱うことができます。Javaでは、入出力を扱うための基本的なインタフェースやクラス群を**java.ioパッケージ**にまとめています。

ストリームには、大きく分けて入力ストリームと出力ストリームがあります。**入力ストリーム**は、入力元からデータをプログラムに読み込むためのストリームです。もう1つの**出力ストリーム**は、入力ストリームとは反対に、プログラムから出力先に書き出すためのストリームです。
ストリームには、入出力という分類のほかに、扱うデータの種類に応じた分類もあります。ストリームが扱うデータの種類には、大きく分けて文字ストリームとバイトストリームがあります。人間が読むための文字を扱うのが**文字ストリーム**、コンピュータが解釈するためのバイトコードを扱うのが**バイトストリーム**です。
入出力とデータの種類を掛け合わせると、ストリームには次の4つの種類があることがわかります。

	文字ストリーム	バイトストリーム
入力	文字入力ストリーム	バイト入力ストリーム
出力	文字出力ストリーム	バイト出力ストリーム

また、それぞれの基本的な機能を表すためのクラスがjava.ioパッケージに用意されています。これらは「**基底クラス**」とも呼ばれ、実際のプログラミングでは用途に応じたサブクラスを使用します。

	文字ストリーム	バイトストリーム
入力	java.io.Reader	java.io.InputStream
出力	java.io.Writer	java.io.OutputStream

以上のことから、選択肢**B**が正解です。

では、実際にテキストファイルを読み込み、コンソールに表示するサンプルコードを見ていきましょう。まず、カレンディレクトリなどのクラスパス上に次のような内容の「sample.txt」を用意してください。

例 sample.txt

```
hello
java
input
output
stream
```

次のコード例が、sample.txtを読み込み、コンソールに表示するためのものです。文字入力ストリームを扱うため、ファイルから文字を読み込む**java.io.FileReaderクラス**を使います。このクラスは、前述した基底クラスの1つであるjava.io.Readerクラスのサブクラスです。

※次ページに続く

例 FileReaderクラスでファイルから文字を読み込む

```
1. import java.io.FileReader;
2.
3. public class Sample {
4.     public static void main(String[] args) throws Exception {
5.         FileReader reader = null;
6.         try {
7.             reader = new FileReader("sample.txt");
8.             int i = 0;
9.
10.            while ((i = reader.read()) != -1) {
11.                char c = (char) i;
12.                System.out.print(c);
13.            }
14.        } finally {
15.            if (reader != null) {
16.                reader.close();
17.            }
18.        }
19.    }
20. }
```

FileReaderのコンストラクタには、読み込みたいファイルのパスを記述します。コード例では、読み込むテキストファイルである「sample.txt」はクラスパス上にあるため、ファイル名のみを記述しています。本来であれば、この段階でファイルが存在しなかった場合を想定して検査例外の1つであるjava.io.FileNotFoundExceptionへのcatchブロックが必要になりますが、今回はmainメソッドにthrowsを追加して省略しています。

例外が発生せず、ファイルにアクセスできたら、次はファイルの内容を読み込みます。ファイル内の文字を読み込むには、**Readerクラス**から引き継いだ**readメソッド**を使います。このメソッドは、文字を「コードポイント」と呼ばれる、文字ごとに割り振られている番号で戻すため、読み込みは基本的に**1文字単位**で行われます。ファイル内のすべての文字を読み込めたら、このメソッドはファイルの**終端**を表すために**-1**を戻します。そのため、コード例のようにint型変数iにreadメソッドの戻り値を代入し、「その値が-1でなければ繰り返し処理を継続する」という条件式のwhile文で先頭の文字から最後の文字まで読み込むことができます。
なお、readメソッドが何らかの理由でファイルを正常に読み込めなかった場合は、検査例外の1つであるjava.io.IOExceptionをスローしますが、前述のとおり、mainメソッドでthrowsを宣言しているためcatchブロックは省略しています。

コード例では、sample.txtから1文字ずつ読み込み、その結果をコンソールに次々と表示しているため、このコードを実行すると同じ内容がコンソールに表示されます。

例 実行結果

```
> java Sample
hello
java
input
output
stream
```

読み込みが終わり、不要になったストリームは閉じなければいけません。Javaのガベージコレクタは、どこからも参照されなくなったインスタンスが使っていたメモリ領域を自動的に解放しますが、入出力ストリームが扱うファイルやデータベース、ネットワーク、サーバーなどとの接続を切断することはできません。そのため、プログラマーがストリームを閉じる処理を記述し忘れると、メモリやネットワーク帯域といったリソースがいつまでも解放されず、パフォーマンスに影響を及ぼしたり、最悪の場合はコンピュータが正常に動作しなくなることがあります。このようなトラブルを起こさないよう、コード例では**finallyブロックで必ずcloseメソッドを呼び出す**ようにしています。こうすることで、例外発生の有無にかかわらず、必ずストリームを閉じることができます。

なお、次のように**try-with-resources文**を使えば、finallyブロックでストリームを明示的に閉じなくても自動的に**closeメソッド**が呼び出されます。前述のコード例でtry-with-resources文を使うと、安全性が担保されるだけでなく、コードが短くなり可読性も向上します。

例 try-with-resources文でリソースを閉じる

```
1. import java.io.FileReader;
2.
3. public class Sample {
4.     public static void main(String[] args) throws Exception {
5.         FileReader reader = new FileReader("sample.txt");
```

※次ページに続く

```
 6.         try(reader) {
 7.             int i = 0;
 8.             while ((i = reader.read()) != -1) {
 9.                 char c = (char) i;
10.                 System.out.print(c);
11.             }
12.         }
13.     }
14. }
```

リソースが枯渇するような状況を外部から意図的に作り出す攻撃のことを「リソース枯渇攻撃」と呼びます。プログラマーが無意識のうちにこのような攻撃が行えるソフトウェアを作ることのないよう、「ストリームは、必ず閉じるようにするか、try-with-resources文を使う」ことを覚えておきましょう。

4. D ➡ P236

バイトストリームの入力を扱うクラスに関する問題です。
解答3で学んだとおり、java.io.FileReaderクラスの場合は1文字単位でファイルを読み込んでいました。1文字単位で処理をすると、readメソッドの呼び出しごとに、OSが提供するネイティブAPIを使ってファイルからバイトを読み込み、char型に変換してから、読み込んだ1文字をプログラムで扱うことになります。100文字あれば100回、これを繰り返すため、扱う文字量が多くなるとパフォーマンスが大幅に低下してしまいます。そこで、Javaの入出力では、毎回ディスクアクセスを発生させずにバッファからデータを読み込む仕組みを提供しています。この仕組みを使うことで、ネイティブAPIが呼び出されるのはバッファが空になったときだけになるため、ディスクアクセスを伴う処理のパフォーマンスを劇的に改善することができるのです。

この仕組みを提供するのが、**java.io.BufferedReaderクラス**です。文字を**バッファ**に溜め込むことで、何度もネイティブAPIを呼び出さずに大量の文字を一度に扱える仕組みを提供します。なお、BufferedReaderは、その名前から推測できるようにReaderクラスのサブクラスで、**文字ストリーム**を扱います。一方、**バイトストリーム**を扱うときにバッファを使うには、**java.io.BufferedInputStreamクラス**を使います。よって、選択肢Cは誤りで、選択肢Dが正解です。
設問では、写真データを読み込むために使用するクラスであるため、文字データを扱う選択肢AのFileReaderは誤りです。選択肢BのInputStreamは、写真データのようなバイトデータを扱うクラスですが、バッファを使って処理する効率化のためのクラスではありません。よって、誤りです。

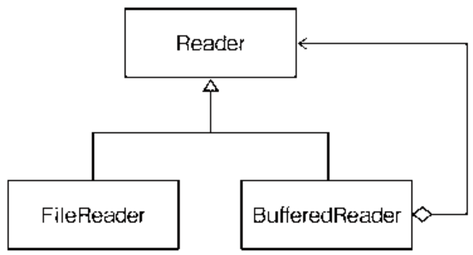
次のコードが、BufferedReaderクラスを使った例です。解答3で利用した例を、BufferedReaderに置き換えて効率化しています。

例 BufferedReaderクラスの使用例

```
 1. import java.io.BufferedReader;
 2. import java.io.FileReader;
 3.
 4. public class Sample {
 5.     public static void main(String[] args) throws Exception {
 6.         FileReader fr = new FileReader("sample.txt");
 7.         BufferedReader reader = new BufferedReader(fr);
 8.         try (reader) {
 9.             String line = null;
10.             while((line = reader.readLine()) != null) {
11.                 System.out.println(line);
12.             }
13.         }
14.     }
15. }
```

BufferedReaderは単体では動作せず、コンストラクタで受け取ったReader型の参照を通じてファイルにアクセスします。BufferedReaderは、あくまでもバッファへのアクセスにより効率的にデータを読み込むことに特化し、実際の入力元にアクセスするのはReaderの別のサブクラスに任せているのです。BufferedReaderはReader型をコンストラクタの引数として受け取るため、入力元の違いに応じて用意されているReaderのサブクラスと組み合わせることができます。

【Reader、FileReader、BufferedReaderの関係】



このように基本的な機能と、それを拡張する機能を組み合わせられるようにし、自由な機能拡張ができるようにするデザインパターンを「**Decoratorパターン**」と呼びます。もし、入力元を変えたいのならFileReaderのようなクラスを追加し、拡張機能を変えたいのならBufferedReaderのようにReader型を受け取って処理するクラスを追加することで、組み合わせのパターンを増やしてソフトウェアの柔軟な仕様変更に対応できるようにするのです。java.ioパッケージには、このように基本的な機能と拡張機能を担う数多くのクラスが用意されており、これらを自由に組み合わせることでさまざまな要求に応えられるようになっています。

FileReaderでは1文字ずつ読み込んでいましたが、BufferedReaderでは**1行ずつ**文字列を読み込みます。テキストが**終端**に達した場合には、**null**が戻されます。そのため、前述のコード例では、nullになるまで1行ずつ繰り返し処理する条件式を持つwhile文を使っています。このコードを実行すると、次のような結果がコンソールに表示されます。

例 実行結果

```
> java Sample
hello
java
input
output
stream
```

なお、第4章で学んだStream APIを使えば、BufferedReaderでの読み込みは簡潔に記述できるようになります。次のコードが、前述のコード例をStream APIを使うように修正したものです。

例 Stream APIを使った修正例

```
 1. import java.io.BufferedReader;
 2. import java.io.FileReader;
 3.
 4. public class NewSample {
 5.     public static void main(String[] args) throws Exception {
 6.         FileReader fr = new FileReader("sample.txt");
 7.         BufferedReader reader = new BufferedReader(fr);
 8.         try (reader) {
 9.             reader.lines().forEach(System.out::println);
10.         }
11.     }
12. }
```

ここでは、BufferedReaderの**linesメソッド**を呼び出してStreamオブジェクトを作り、forEachメソッドでコンソールへの出力を繰り返し処理しています。コードが短く、わかりやすくなったのが実感できると思います。

5. D

➡ P237

これまでは、文字入力ストリームを扱ってきました。ここでは文字出力ストリームについて学びます。文字出力ストリームの使い方は、使用するクラスがjava.io.Writerとそのサブクラスに変わっただけで、入力ストリームのときとあまり変わりません。まずは簡単なコード例を確認してみましょう。次のコードは、「output.txt」という名前のテキストファイルに「Hello, Java!」という文字列を書き込む例です。

例 FileWriterクラスの使用例

```
 1. import java.io.FileWriter;
 2.
 3. public class Sample {
 4.     public static void main(String[] args) throws Exception {
 5.         FileWriter out = new FileWriter("output.txt");
 6.         try (out) {
 7.             out.write("Hello, Java!");
 8.         }
 9.     }
10. }
```

テキストファイルへの書き込みには、**Writer**のサブクラスである**java.io.FileWriter**を使います。このクラスのコンストラクタには、出力先のファイル名を渡します。コンストラクタに指定した出力先のファイルが存在しなかった場合には、新たにファイルが作られます。
FileWriterのインスタンスが生成されると、出力ストリームを使って文字の書き込みを行うことができるようになります。書き込みを行うには、Writerクラスから引き継いだ**writeメソッド**を使います。このコードを実行してもコンソールには何も表示されませんが、クラスパス上にoutput.txtという名前の新しいファイルが作られ、そのファイル内に「Hello, Java!」という文字列が書き込まれます。
このコードを何回実行しても、output.txtには1行しか書き込まれません。こ

れは、ファイルの内容を毎回**上書き**しているからです。ファイルの最後に**追記**するには、次のようにFileWriterのコンストラクタの第2引数にtrueを渡します。この引数は、追記モードを有効にするかどうかを指定するもので、trueを渡すことで有効になります。デフォルトはfalseで、上書きモードです。

例 ファイルへの追記を有効にする

```
1. import java.io.FileWriter;
2.
3. public class Sample {
4.     public static void main(String[] args) throws Exception {
5.         FileWriter out = new FileWriter("output.txt", true);
6.         try (out) {
7.             out.write("Hello, Java!");
8.         }
9.     }
10. }
```

このコードの実行結果として「output.txt」を確認すると、次のように改行されずにつながって出力されていることがわかります。

例 実行結果

```
Hello, Java!Hello, Java!
```

これは、writeメソッドが自動的に**改行コード**を出力する文字列に追加してくれるわけではないからです。そこで、次のようにwriteメソッドを追加し、改行コードを書き込みます。

例 改行コードを追加

```
1. import java.io.FileWriter;
2.
3. public class Sample {
4.     public static void main(String[] args) throws Exception {
5.         FileWriter out = new FileWriter("output.txt", true);
6.         try (out) {
7.             out.write("¥n");
8.             out.write("Hello, Java!");
9.         }
10.     }
11. }
```


例 実行結果

```
Hello, Java!Hello, Java!
Hello, Java!
```

以上のことから、選択肢Dが正解です。ファイルへの出力ストリームは、ファイルが存在しなければ新しい空のファイルを作ってから、書き込みをします。よって、選択肢AやBは誤りです。

6. C

➡ P237

BufferedWriterクラスに関する問題です。
FilerReaderが1文字ずつ読み込みを行うのと同様に、FileWriterもwriteメソッドで渡された文字列を1文字ずつ書き込みます。このため、**文字出力ストリーム**にも**バッファ**を利用するクラスとして**java.io.BufferedWriter**が用意されています。次のコードは、BufferedWriterを使って効率的な書き込みができるように、前述のコード例を改造したものです。

例 BufferedWriterクラスの使用例

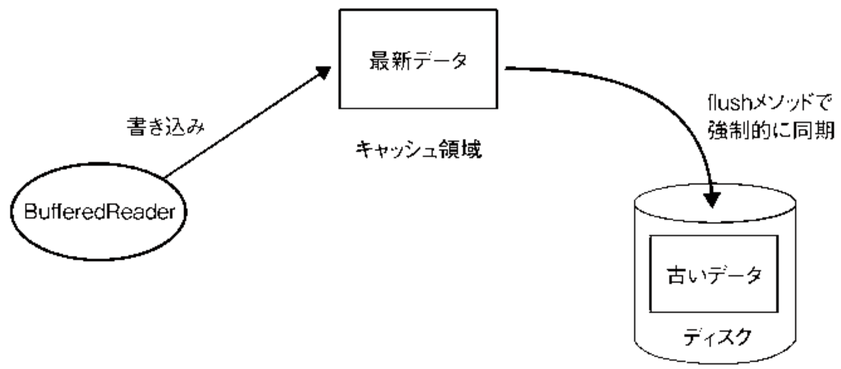
```
1. import java.io.BufferedWriter;
2. import java.io.FileWriter;
3.
4. public class Sample {
5.     public static void main(String[] args) throws Exception {
6.         FileWriter out = new FileWriter("output.txt", true);
7.         BufferedWriter writer = new BufferedWriter(out);
8.         try (writer) {
9.             writer.newLine();
10.            writer.write("buffering output");
11.            writer.flush();
12.        }
13.    }
14. }
```

BufferedWriterには、**newLineメソッド**という**改行コード**を出力するための専用のメソッドが備わっています。一般的に改行コードには、プラットフォームによってキャリッジリターン（CR）とラインフィード（LF）の2種類があります。newLineメソッドは、プログラムがどちらの改行コードを使っているかを自動的に判別し、適切なものを選択して出力します。

※次ページに続く

BufferedWriterのwriteメソッドは、ファイルに直接書き込まず、バッファに書き込みをします。そのため、多くの場合は認識できないほどですが、バッファに書き込まれた最新データと実際のファイルのデータが食い違うタイミングができてしまいます。バッファからファイルへの書き込みは自動で行われますが、強制的に書き込みをさせるのであれば、コード例のように**flushメソッド**を使います。このメソッドを使えば、バッファ内のデータがファイルへと出力され、メモリとファイルの同期が完了します。

【flushメソッドでバッファからデータを書き込むイメージ】



以上のことから、選択肢**C**が正解です。

7. D

➡ P237

BufferedInputStreamクラスに関する問題です。
これまでの設問で、文字データの入出力について学びました。ここでは、バイトストリームの入出力について学びましょう。バイトストリームは解答3で解説した基底クラスとサブクラスが異なるだけで、文字ストリームと同じように扱えます。バッファを使って効率化する点についても同じです。

次のコードが、sample.jpgという写真データをコピーする例です。このコード例では、入力ストリームと出力ストリームの両方を用意し、入力ストリームから取得したバイトデータを出力ストリームに渡すことで、JPEGファイルをコピーしています。

例 JPEGデータをコピーする

```
1. import java.io.BufferedInputStream;
2. import java.io.BufferedOutputStream;
3. import java.io.FileInputStream;
4. import java.io.FileOutputStream;
5.
6. public class Sample {
7.     public static void main(String[] args) throws Exception {
8.         FileInputStream fis
9.             = new FileInputStream("sample.jpg");
10.        BufferedInputStream bis
11.            = new BufferedInputStream(fis);
12.
13.        FileOutputStream fos
14.            = new FileOutputStream("sample_bk.jpg");
15.        BufferedOutputStream bos
16.            = new BufferedOutputStream(fos);
17.
18.        try (bis; bos) {
19.            byte[] data = null;
20.            while ((data = bis.readNBytes(1024)).length != 0) {
21.                bos.write(data);
22.            }
23.            bos.flush();
24.        }
25.    }
26. }
```

読み込む対象がJPEGファイルなので、ファイルから**バイトストリーム**を読み込む**java.io.FileInputStreamクラス**を使います。また、文字入力ストリームの場合と同様に、このクラスだけではディスクアクセスが頻繁に発生するため、バッファを使って効率化します。**java.io.BufferedInputStreamクラス**でバッファを使って効率化します。

バイト出力ストリームはファイルへの出力であるため、java.io.FileOutputStreamクラスを使います。また、入力と同様にバッファを使って効率化するためにjava.io.BufferedOutputStreamクラスも併用しています。

バイト入力ストリームでは、BufferedInputStreamが基底クラスの**InputStreamクラス**から引き継いだ**readNBytesメソッド**を使って、指定した任意のバイト数単位でデータを読み込みます。**readAllBytes**というメソッドで全バイトを一度に取り出すことも可能ですが、このコード例では大きなファイルであったときにメモリへの負担を減らすために、readNBytesメソッドを使って1,024バイト（1KByte）ずつ読み込んでいます。

※次ページに続く

読み込みがファイル終端に達し、さらに読み込むべきバイトが存在しなければ要素数0のバイト配列が戻されるため、「readNBytesメソッドの戻り値の配列のlengthが0でなければ繰り返す」という条件式でwhile文を記述しています。sample.jpgという名前のJPEGファイルを用意してから、このコード例を実行すると、同じ内容を持ったJPEGファイルがコピーされているはずです。
以上のことから、選択肢**D**が正解です。

8. A ➡ P237

コンソールからの入力を受け付けるには、次のように**java.lang.Systemクラス**のInputStream型の**inフィールド**を利用します。このコード例では、バイト入力ストリームを扱うSystem.inのInputStreamを、文字入力ストリームを扱うReaderに変換する**InputStreamReader**を経由して、バッファを利用した効率的なテキスト入力のためのBufferedReaderクラスのインスタンスを作っています。

例 System.inの使用例

```
1. import java.io.BufferedReader;
2. import java.io.InputStreamReader;
3.
4. public class UseSystemIn {
5.     public static void main(String[] args) throws Exception {
6.         try (BufferedReader br
7.                 = new BufferedReader(
8.                     new InputStreamReader(System.in))) {
9.
10.            String input = br.readLine();
11.            System.out.println(input);
12.        }
13.    }
14. }
```

このようなコンソールからの入力を受け取る方法をより簡略化するためにJava SE 6で導入されたのが、**java.io.Consoleクラス**です。次のコードが、前述のコードをConsoleクラスを利用するように書き換えた例です。少ないコード量で同じことができるのがわかります。

例 Consoleクラスの使用例

```
1. import java.io.Console;
2.
3. public class Sample {
4.     public static void main(String[] args) {
5.         Console console = System.console();
6.         String str = console.readLine();
7.         System.out.println(str);
8.     }
9. }
```

System.inではなく、Consoleクラスを使うもう1つの理由に、**パスワード入力機能**があります。System.inを利用すると、コンソールで入力されたものは、必ず表示されてしまいます。

例 実行結果

```
> java UseSystemIn
password    ⬅入力した文字列
password    ⬅出力された文字列
```

一方、Consoleクラスでは、入力内容を表示せずに入力を受け付ける**readPasswordメソッド**を利用できます。次のコードが、readPasswordメソッドを使った例です。**String型**を戻す**readLineメソッド**と違い、readPasswordメソッドは**char配列型**を戻す点に注意してください。

例 readPasswordメソッドの使用例

```
1. import java.io.Console;
2.
3. public class UsePassword {
4.     public static void main(String[] args) {
5.         Console console = System.console();
6.         char[] password = console.readPassword();
7.         System.out.println(String.valueOf(password));
8.     }
9. }
```

このコードを実行すると、入力した内容は表示されませんが、Enterキーを押すと、入力内容がプログラムに渡されて、結果が出力されます。

例 実行結果

```
> java UsePassword
                ← 「password」と入力しても表示されない
password        ← 出力された文字列
```

以上のことから、選択肢**A**が正解です。

試験対策

readPasswordメソッドは、コンソールに入力した内容を表示せずに受け付けます。ただし、入力した内容は、そのまま出力されます。

9. A

➡ P238

Scannerクラスを使ったテキストデータの処理に関する問題です。
java.util.Scannerクラスは、文字入力ストリームを簡単に扱えるようにした便利なクラスです。たとえば、ファイルの内容をコンソールに表示するだけなら、次のコード例のように記述します。

例 Scannerクラスの使用例

```
 1. import java.io.FileInputStream;
 2. import java.util.Scanner;
 3.
 4. public class Sample {
 5.     public static void main(String[] args) throws Exception {
 6.         FileInputStream fis
 7.             = new FileInputStream("sample.txt");
 8.         Scanner scanner = new Scanner(fis);
 9.         try (scanner) {
10.             while(scanner.hasNext()) {
11.                 System.out.println(scanner.next());
12.             }
13.         }
14.     }
15. }
```

このコード例では、ファイルから読み込むFileInputStreamのインスタンスを作り、それをScannerに渡し、while文の条件式で**hasNextメソッド**で次に読み込むべき行があるかを確認しながら、**nextメソッド**で**1行ずつ**読み込んでコンソールに表示しています。
このようなファイルを読み込む処理は、以前は次のように記述していました。これを置き換えたのが上記のコード例ですが、これだけではあまり便利さは感じられません。

```
String line = null;
while((line = reader.readLine()) != null) {
    System.out.println(line);
}
```

Scannerクラスが便利なのは、読み込んだ文字列を処理するメソッドを持っているためです。たとえば、文字データを読み込んで1行ずつ処理するケースとしては、カンマ区切りのテキストデータ（CSVファイル）を扱うプログラムが挙げられるでしょう。次のCSVファイルは、日本郵便が公開している東京都の郵便番号一覧を示すCSVファイルからの抜粋です。

例 postalcode.csv　※日本郵便が公開しているCSVデータの一部を加工しています。

```
13101,"102  ","1020072","ﾄｳｷｮｳﾄ","ﾁﾖﾀﾞｸ","ｲｲﾀﾞﾊﾞｼ","東京都","千代田区","飯田橋"
13101,"102  ","1020082","ﾄｳｷｮｳﾄ","ﾁﾖﾀﾞｸ","ｲﾁﾊﾞﾝﾁｮｳ","東京都","千代田区","一番町"
13101,"101  ","1010032","ﾄｳｷｮｳﾄ","ﾁﾖﾀﾞｸ","ｲﾜﾓﾄﾁｮｳ","東京都","千代田区","岩本町"
```

このCSVファイルでは、1行に1つの郵便番号に関するデータが記述されており、それぞれのデータはカンマで区切られています。このようなデータをJavaの入出力で扱うには、1行単位でデータを読み込みながら、カンマで分割して、個別のデータとして扱えるようにする必要があります。この読み込みから分割までの処理を1つにまとめたものが、Scannerクラスなのです。

それでは、Scannerクラスを使ったファイルの読み込みからテキスト処理までの一連の流れを確認してみましょう。まず、郵便番号に関連する1件分のデータを保持するためのPostalCodeクラスを用意します。このクラスでは、7桁の郵便番号、都道府県、市区町村、町域を持つための4つのフィールドを定義しています。

例 PostalCodeクラス

```
1. public class PostalCode {
2.     private String code;
3.     private String prefecture;
```

```
    private String district;
    private String town;

    public PostalCode(  String code,
                        String prefecture,
                        String district,
                        String town) {
        this.code = code;
        this.prefecture = prefecture;
        this.district = district;
        this.town = town;
    }

    @Override
    public String toString() {
        return code + " " + prefecture + district + town;
    }
}
```

次のコード例では、まず**Scannerクラス**の**useDelimiterメソッド**を使い、カンマと改行コードを**区切り文字**に変更しています。これは、Scannerのデフォルトの区切り文字が空白文字（空白、タブ、改行コード）だけになっており、カンマが含まれないためです。コード例のようにパイプ「 | 」を使えば、複数の区切り文字を列挙することができます。

例 postalcode.csvの読み込みからテキスト処理までを行う

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

public class Sample {
    public static void main(String[] args) throws Exception {
        List<PostalCode> list = createList("postalcode.csv");
        list.forEach(System.out::println);
    }

    private static List<PostalCode> createList(String file)
                                throws FileNotFoundException {

        List<PostalCode> list = new ArrayList<>();
        FileInputStream fis = new FileInputStream(file);
        Scanner scanner = new Scanner(fis);
        scanner.useDelimiter(",|¥n");

        try (scanner) {
            while (scanner.hasNext()) {
                // skip 全国地方公共団体コード
                scanner.next();
                // skip 郵便番号（5桁）
                scanner.next();
                // 郵便番号（7桁）
                String code = scanner.next().replaceAll("¥"", "");
                // skip 都道府県名 半角カタカナ
                scanner.next();
                // skip 市区町村名 半角カタカナ
                scanner.next();
                // skip 町域名 半角カタカナ
                scanner.next();
                // 都道府県名
                String pref = scanner.next().replaceAll("¥"", "");
                // 市区町村名
                String dist = scanner.next().replaceAll("¥"", "");
                // 町域名
                String town = scanner.next().replaceAll("¥"", "");

                list.add(new PostalCode(code, pref, dist, town));
            }
        }
        return list;
    }
}
```

Scannerクラスの**nextメソッド**は、指定された区切り文字で分割された文字列（トークンといいます）を1つ戻します。この例で読み込んでいるCSVファイルには、全国地方公共団体コード、郵便番号（5桁）、都道府県名のカナ、市区町村のカナ、町域名のカナなどの不要なデータが混ざっています。そこで、

nextメソッドを順番に呼び出し、必要なデータのみを変数に代入するようにしています。
繰り返し処理の最後に、PostalCodeクラスのインスタンスを作り、その参照をリストに追加して、最後にリストを戻します。mainメソッドでは、戻ってきたリストの要素の一覧をコンソールに表示しているため、このコードを実行すると次のような結果が表示されるはずです。

例 実行結果

```
> java Sample
1020072 東京都千代田区飯田橋
1020082 東京都千代田区一番町
1010032 東京都千代田区岩本町
```

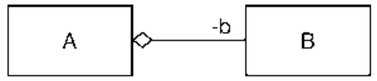
設問のコードでは、ファイルから読み込むのではなく、文字列を読み込んで処理するためのScannerを用意しています。useDelimiterメソッドでカンマを使って分割するよう指示しているため、文字列はA、B、C、D、Eの5文字に分けられます。nextメソッドを呼ぶ度にA～Eの順に文字が戻されるため、5回目のnextメソッドではEが戻され、コンソールに表示されます。以上のことから、選択肢**A**が正解です。

たとえば、設問のコードでもう1回多くnextメソッドを呼ぶと、次の要素がないためjava.util.NoSuchElementExceptionがスローされます。しかし、設問のコードでは要素が存在するため、実行時にこの例外がスローされることはありません。よって、選択肢Dは誤りです。また、取り出すべき要素がなければ例外がスローされ、nullが戻ることはありません。よって、選択肢Bも誤りです。

10. C ➡ P238

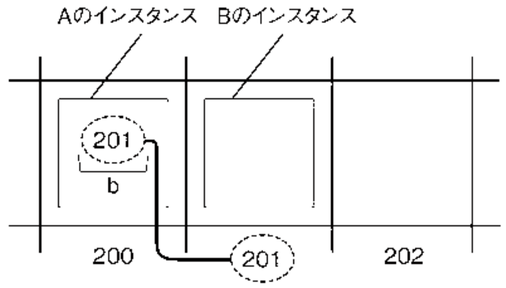
シリアライズとSerializableインタフェースに関する問題です。
Javaの入出力では、インスタンスをストリームに出力したり、ストリームから出力されたインスタンスを取り出したりすることができます。メモリ上にあるインスタンスをストリームとして出力することを「**シリアライズ**」、逆にファイルに書き出された情報をもとにインスタンスを作り直すことを「**デシリアライズ**」と呼びます。
シリアライズとは、単純にメモリ上にあるインスタンスをそのままファイルに出力すればよいというものではありません。デシリアライズ（復元）したときに正しく動作できることを保証しなくてはいけないのです。たとえば、次のようにBクラスを使うAクラスがあったとします。Aには、Bを使うためにB型のフィールドbがあります。

【AクラスとBクラスの関係】



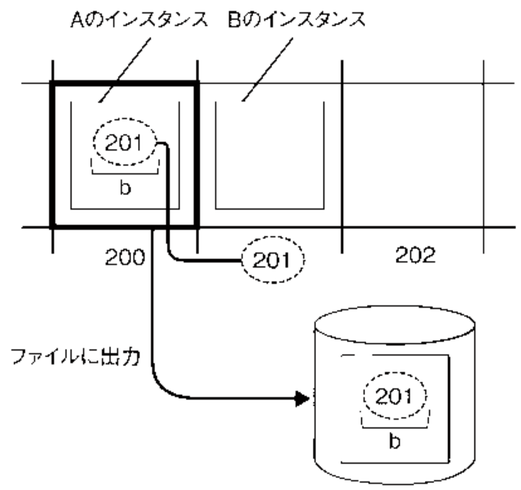
この2つのクラスを利用するとき、メモリ上では次のようなイメージで2つのインスタンスが関係することになります。Aのインスタンスが、Bのインスタンスがあるメモリ場所への参照（リンク）をフィールドbに持っているという状態です。

【メモリ上のAとBのイメージ】



このとき、次の図のようにAのインスタンスだけをファイルに出力したとしましょう。ファイルには、「Aのインスタンスは、Bのインスタンスへの参照を持っている」という情報までが書き込まれています。

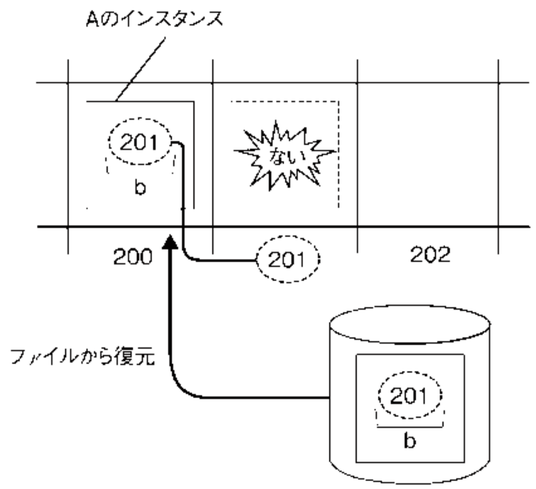
【Aをファイルに出力したときのイメージ】



このようにインスタンスをファイルに書き出して保存しておいたあと、必要なタイミングでファイルからインスタンスを復元したとします。そうすると、次の図のように、復元されたAのインスタンスが持つBのインスタンスへの参

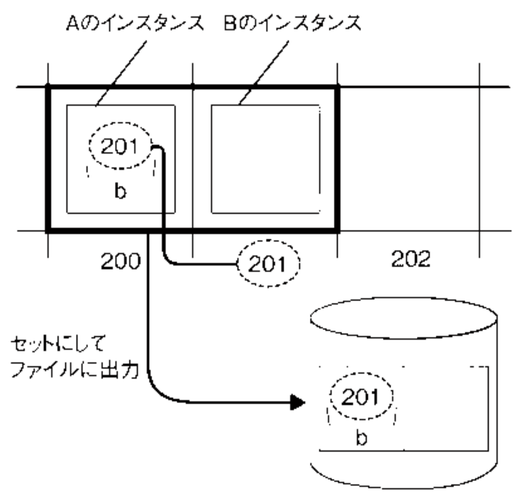
照の先にはインスタンスが存在しない可能性があります。コンピュータそのものの再起動やJVMの再起動によってメモリはクリアされる可能性がありますし、それ以外にもプログラムが進行することでガベージコレクションの対象となってしまうこともあるからです。

【Aを復元したときのイメージ】



このような事態を防ぐために、インスタンスを出力するときには、そのインスタンスが参照している先も同時に出力しなくてはいけません。このように複数のインスタンスを1つにまとめてしまうことを「シリアライズ（直列化）」と呼びます。反対にデシリアライズは、1つにまとまったものを複数のインスタンスに戻すことを意味します。

【シリアライズのイメージ】



クラスは、そのままではシリアライズできません。JVMに「このクラスはシリアライズしなければいけない」ことを示す必要があるのです。**java.io.Serializableインタフェース**は、実装クラスがシリアライズしなければいけないことをJVMに示すためのマーカーインタフェースです。

マーカーインタフェースとは、抽象メソッドを1つも持たず、クラスに何らかの意味を追加するために利用するインタフェースです。Serializableインタフェースは、クラスに「シリアライズが可能」という意味を追加するためのインタフェースなのです。そのため、Serializableインタフェースを実装していないクラスをシリアライズしようとすると、**java.io.NotSerializableException**という例外がスローされます。以上のことから、選択肢**C**が正解です。

標準クラスライブラリの多くは、Serializableインタフェースを実装しています。

それでは、オブジェクトのシリアライズの例を見ていきましょう。まず、書き出す対象のクラスとして、次のようなItemクラスを用意します。

例 Itemクラス

```
1. public class Item {
2.     private String name;
3.     private int price;
4.
5.     public Item(String name, int price) {
6.         this.name = name;
7.         this.price = price;
8.     }
9.     public String getName() {
10.        return name;
11.    }
12.    public int getPrice() {
13.        return price;
14.    }
15.    @Override
16.    public String toString() {
17.        return "Item [name=" + name + ", price=" + price + "]";
18.    }
19. }
```

書き出す対象のクラスが定義できたら、書き出すクラスを作りましょう。次のコードが、インスタンスをシリアライズして書き出すためのjava.io.ObjectOutputStreamを使った書き出しをしている例です。今回は、インスタンスをファイルに出力するため、ObjectOutputStreamのコンストラクタに、java.io.FileOutputStreamのインスタンスへの参照を渡します。このような方法は、BufferedOutputStreamなどと同じです。

例 インスタンスをファイルに出力する

```
1. import java.io.FileOutputStream;
2. import java.io.ObjectOutputStream;
3.
4. public class Sample {
5.     public static void main(String[] args) throws Exception {
6.         FileOutputStream fos
7.             = new FileOutputStream("sample.ser");
8.         ObjectOutputStream out
9.             = new ObjectOutputStream(fos);
10.        try (out) {
11.            Item item = new Item("apple", 100);
12.            out.writeObject(item);
13.        }
14.    }
15. }
```

Itemのインスタンスを作り、**ObjectOutputStreamのwriteObjectメソッド**にその参照を渡します。これで、シリアライズされてファイルに出力されます。しかし、実行すると次のような例外がスローされてしまいます。

例 実行結果

```
> java Sample
Exception in thread "main" java.io.NotSerializableException: Item
    at java.base/java.io.ObjectOutputStream.writeObject0(ObjectOu
tputStream.java:1185)
    at java.base/java.io.ObjectOutputStream.writeObject(ObjectOut
putStream.java:349)
    at Sample.main(Sample.java:12)
```

前述のとおり、シリアライズ対象のクラスは、java.io.Serializableインタフェースを実装してなくてはいけません。ItemクラスはSerializableインタフェースを実装していないため、java.io.NotSerializableExceptionがスローされてしまったのです。そこで、次のようにItemクラスでSerializableを実装するように修正します。

例 Itemクラスの修正

```
1. import java.io.Serializable;
2.
3. public class Item implements Serializable {
        … 中略 …
21. }
```

再度、Sampleクラスを実行すると何も起きずに終了し、FileOutputStreamクラスのコンストラクタで指定した「sample.ser」という名前のファイルができているはずです。

拡張子「.ser」は「Serialize」の先頭から3文字を取ったもので、慣習的に使われています。

シリアライズに続いて、デシリアライズを見ていきましょう。次のコードが、デシリアライズの実装例です。java.io.FileInputStreamでファイルから読み込み、java.io.ObjectInputStreamクラスでデシリアライズを実現します。この**ObjectInputStreamクラス**は、InputStreamからオブジェクトを作成する（デシリアライズ）するためのクラスです。

例 デシリアライズの実装例

```
1. import java.io.FileInputStream;
2. import java.io.ObjectInputStream;
3.
4. public class Sample2 {
5.     public static void main(String[] args) throws Exception {
6.         FileInputStream fis
7.             = new FileInputStream("sample.ser");
8.         ObjectInputStream in
9.             = new ObjectInputStream(fis);
10.        try (in) {
11.            Object obj = in.readObject();
12.            Item item = (Item) obj;
```

```
13.             System.out.println(item);
14.         }
15.     }
16. }
```

ObjectInputStreamでデシリアライズをするには、**readObjectメソッド**を使います。このメソッドは、デシリアライズしたインスタンスへの参照をjava.lang.Object型で戻すので、コード例のようにダウンキャストしなければいけません。このコードを実行すると、次のように出力時のデータをそのまま持ったインスタンスが復元されたことがわかります。

例 実行結果

```
> java Sample2
Item [name=apple, price=100]
```

以上のように、シリアライズとデシリアライズは、ObjectOutputStreamとObjectInputStreamを使うだけでとても簡単に実現できます。Javaの入出力機能は、非常に強力なことがわかります。

試験対策

クラスはSerializableインタフェースを実装することでシリアライズとデシリアライズが可能となります。

参考

シリアライズするオブジェクトのフィールドのうち、一時的に利用する値を保持するフィールドなどの理由からシリアライズに含まないようにするには、transientでフィールドを修飾します。この修飾子で修飾されたフィールドは、シリアライズの対象にはなりません。
また、staticで修飾されたフィールドもシリアライズの対象にはなりません。これは、staticフィールドはインスタンスとは無関係に、static領域に作られる変数だからです。シリアライズは、あくまでインスタンスを出力するためのものですので、インスタンスと無関係なものは含まれません。

11. D

➡ P239

シリアライズやデシリアライズでは、すべてのフィールドを書き出すのではなく、自由に書き出し・読み込みを制御できる**カスタム・シリアライズ**という方法があります。この方法では、java.io.Serializableインタフェースを実装したクラスで、シリアライズやデシリアライズをするときに、java.io.ObjectOutputStreamクラスやjava.io.ObjectInputStreamクラスに直接、書き込んだり、読み込んだりします。

例 カスタム・シリアライズの例（シリアライズ）

```
1. import java.io.IOException;
2. import java.io.ObjectInputStream;
3. import java.io.ObjectOutputStream;
4. import java.io.ObjectStreamField;
5. import java.io.Serializable;
6.
7. public class Employee implements Serializable {
8.     private String name;
9.     private String telephone;
10.
11.     public Employee(String name, String telephone) {
12.         super();
13.         this.name = name;
14.         this.telephone = telephone;
15.     }
16.
17.     private void writeObject(ObjectOutputStream out)
18.                                 throws IOException {
19.         System.out.println("write");
20.         out.writeObject(name);
21.         out.writeObject(telephone);
22.     }
23.
24.     private void readObject(ObjectInputStream in)
25.             throws IOException, ClassNotFoundException {
26.         System.out.println("read");
27.         this.name = (String) in.readObject();
28.         this.telephone = (String) in.readObject();
29.     }
30.
```

```
31.     public String getName() {
32.         return name;
33.     }
34.
35.     public String getTelephone() {
36.         return telephone;
37.     }
38.
39.     @Override
40.     public String toString() {
41.         return "Employee [name=" + name + ", telephone=" +
                    telephone + "]";
42.     }
43. }
```

writeObjectは、**シリアライズ時**に自動的に呼び出されるメソッドです。このメソッドでは、引数としてObjectOutputStreamオブジェクトへの参照を受け取り、writeXXXという各データ型に応じたメソッドを使って、書き出したいフィールドを出力します。前述のコード例では、nameフィールドとtelephoneフィールドの値を出力しています。
readObjectは、**デシリアライズ時**に自動的に呼び出されるメソッドです。このメソッドでは、引数としてObjectInputStreamオブジェクトへの参照を受け取り、readXXXという各データ型に応じたメソッドを使って、データを読み込みます。なお、データの読み込み順は、writeObjectメソッドで書き出した順となります。

では、実際にシリアライズしてファイルに書き出してみましょう。次のコードは、Employeeのインスタンスを1つ作り、それをemp.serというファイルに書き出す例です。

例 Sampleクラス（シリアライズ）

```
1. import java.io.FileOutputStream;
2. import java.io.ObjectOutputStream;
3.
4. public class Sample {
5.     public static void main(String[] args) throws Exception {
6.         ObjectOutputStream out
7.             = new ObjectOutputStream(
8.                     new FileOutputStream("emp.ser"));
```



```
9.
10.         try (out) {
11.             Employee emp
12.                     = new Employee("sample", "0312345678");
13.             out.writeObject(emp);
14.         }
15.     }
16. }
```

このクラスを実行すると、EmployeeクラスのwriteObjectメソッドが呼び出されるため、「write」とコンソールに表示されます。また、ファイル一覧を表示すれば、emp.serという名前でファイルが出力されていることを確認できます。

例 実行結果

```
> java Sample
write
```

次のコードが、emp.serからデシリアライズする例です。

例 Sample2クラス（デシリアライズ）

```
1. import java.io.FileInputStream;
2. import java.io.ObjectInputStream;
3.
4. public class Sample2 {
5.     public static void main(String[] args) throws Exception {
6.         ObjectInputStream in
7.             = new ObjectInputStream(
8.                     new FileInputStream("emp.ser"));
9.
10.         try (in) {
11.             Employee emp = (Employee) in.readObject();
12.             System.out.println(emp);
13.         }
14.     }
15. }
```

このクラスを実行すると、EmployeeクラスのreadObjectメソッドが呼び出されるため、「read」とコンソールに表示され、その後、Employeeクラスの

toStringメソッドを呼び出した結果がコンソールに表示されます。

例 実行結果

```
> java Sample2
read
Employee [name=sample, telephone=0312345678]
```

以上のことから、選択肢**D**が正解です。選択肢AやCのようなメソッドは定義できますが、シリアライズ時に呼び出されることはありません。選択肢BのreadObjectメソッドは、デシリアライズするときに自動的に呼び出されるメソッドです。よって、これらの選択肢は誤りです。

12. D ➡ P239

Javaでファイルを扱うには、java.io.Fileクラスを使うことは解答1で学びました。このクラスはJava 1.0から提供されており、その設計思想はとても古く、次のような課題がありました。

- UNIXでのシンボリックリンク（Windowsでいうショートカットのようなもの）を適切に扱えない
- ディレクトリツリーの探索が手軽にできない
- ファイル属性の扱いやリネーム、非同期I/Oができない
- ファイルのコピーが簡単にできない

これらを解決するためにJava SE 7で導入されたのが、「**NIO.2**（JSR 203: More New I/O APIs for the Java Platform）」です。NIO.2では入出力に関するさまざまな改善が行われましたが、その特徴の1つは、Fileクラスが担っていた、ファイルやディレクトリへのパスを表す機能とファイル操作のための機能が完全に分離されたことです。NIO.2では、ファイルやディレクトリへのパスを表すには**java.nio.file.Pathインタフェース**を使用し、ファイル操作をするには**java.nio.file.Filesクラス**などを使用します。ここでは、まずjava.nio.file.Pathインタフェースについて学びましょう。

ファイルやディレクトリへのパスを扱うのが、java.nio.file.Pathインタフェースの役割です。Pathのインスタンスへの参照は、次のようにPathを利用しやすくするヘルパークラスである、**java.nio.file.Pathsクラス**の**getメソッド**を使って取得します。

```
Path path = Paths.get("sample.txt");
```


getメソッドの引数に、目的のファイルやディレクトリへのパスを記述します。パスはルートディレクトリからの絶対パスのほか、カレントディレクトリからの相対パスのどちらでも指定可能です。たとえば、次のようにカレントディレクトリからの相対パスとして、dirディレクトリ内にあるsample.txtへのパスを表すPathクラスのインスタンスを生成して取得できます。

```
Path path = Paths.get("dir/sample.txt");
```

たとえば、Linux系のOSやmacOSなどのパス区切り文字はスラッシュ「/」ですが、Windowsでは円記号「¥」を使います。このような違いを吸収してパスを表現するために、getメソッドは可変長引数を受け取れるようになっています。このメソッドは、次のように引数として渡された複数の文字列を、システムが稼働するプラットフォームの種類に応じたパス区切り文字を挟みながら連結してくれるため、プラットフォームの違いを意識する必要がありません。

```
Path path = Paths.get("dir", "sample.txt");
System.out.println(path);     // dir/sample.txtと表示される
```

Pathsクラスを使用するほかにも、java.io.FileクラスからPathを取得することもできます。FileクラスからPathを取得するには、次のようにtoPathメソッドを使います。

```
Path path = new File("sample.txt").toPath();
```

もちろん、逆にPathからFileを取り出すことも可能です。そのため、従来のjava.ioパッケージを使った入出力との組み合わせも不自由なく実現できます。次のコードがtoFileメソッドを使ってPathからFileを取り出した例です。

```
File file = path.toFile();
```

最後に、絶対パスを取得するには、次のように**Pathの**toAbsolutePathメソッドを使います。

```
Path path = Paths.get("dir", "sample.txt");
System.out.println(path.toAbsolutePath());
```

このコードを実行すると、次のようにコンソールに絶対パスが表示されるはずです。

例 実行結果 ※実行環境によって表示結果が異なります。

```
> java Sample
/Users/shiga/workspace/5.io/dir/sample.txt
```

このように、Pathインタフェースはこれまでjava.io.Fileクラスが担っていた、ファイルやディレクトリへのパス表現の部分を分離した機能を持っていることがわかります。

以上のことから、選択肢A、B、Cは正しいコードです。選択肢**D**のようにFilesクラスからPathを取得することはできません。

試験対策

以下の基本的な機能を押さえておきましょう。
- ファイルやディレクトリへのパスを表すのは、java.nio.file.Pathインタフェース
- ファイル操作は、java.nio.file.Filesクラス
- Pathのインスタンスへの参照を取得するのは、java.nio.file.Pathsクラスのgetメソッド

13. C ➡ P239

Filesクラスのファイル操作に関する問題です。
java.nio.file.Filesクラスは、java.nio.file.Pathで表現されるファイルやディレクトリを操作するための機能を提供するクラスです。このクラスを使えば、新しいファイルやディレクトリを作ったり、既存のファイルを削除したりすることができます。java.io.Fileクラスのファイル操作部分を担うクラスだと理解しましょう。

次のコードが、Filesクラスを使用した例です。この例では、コンソールからの入力（System.in）を受け付けるScannerを用意してファイル名を受け取り、そのファイルが存在しなければ新しいファイルを作っています。その後、ファイルの最終更新日時とアクセス権を表示し、作成したファイルを削除するかを尋ねてから、コンソールからの入力が「y」だったときにはファイルを削除しています。

例 Filesクラスの使用例

```
1. import java.nio.file.Files;
2. import java.nio.file.Path;
3. import java.nio.file.Paths;
4. import java.util.Scanner;
5.
6. public class Sample {
7.     public static void main(String[] args) throws Exception {
8.         System.out.print("> ");
9.         Scanner scan = new Scanner(System.in);
10.        String name = scan.nextLine();
11.        // dirディレクトリ配下のファイル（コンソールで指定）のパスを取得
12.        Path p = Paths.get("dir", name);
13.        if (Files.exists(p) == false) {
14.            Files.createFile(p);
15.        }
16.
17.        System.out.println(Files.getLastModifiedTime(p));
18.        System.out.println(Files.getPosixFilePermissions(p));
19.
20.        System.out.print("delete y/n? >");
21.        if ("y".equals(scan.nextLine())) {
22.            Files.delete(p);
23.            System.out.println("deleted");
24.        }
25.    }
26. }
```

Pathのインスタンスが表すパスに、実際にファイルが存在するかを確かめるのが**existsメソッド**です。このメソッドの引数にPathのインスタンスへの参照（変数p）を渡すと、存在するか否かを表す真偽値が戻されます。
新しくファイルを作るのが、**Filesクラス**の**createFileメソッド**です。このメソッドは、引数で渡されたPathのファイルを新規に作ります。引数で渡されたPathと同じパスを持つファイルが既に存在する場合には、**java.nio.file.FileAlreadyExistsException**がスローされます。よって、選択肢**C**が正解です。

Filesクラスの**getLastModifiedTimeメソッド**は、ファイルの最終更新日時を調べるためのメソッドです。このメソッドを実行すると、ファイルの最終更新日時を保持したjava.nio.file.attribute.FileTime型の戻り値を戻します。このクラスは、1970年1月1日0時0分0秒0ミリ秒からの経過時間を保持しています。

Filesクラスの**getPosixFilePermissionsメソッド**は、ファイルのアクセス権を調べるためのメソッドです。**POSIX**（Portable Operating System Interface for UNIX）とは、IEEEによって定められた標準規格で、UNIX系OSとして備えるべき仕様を規定しているものです。getPosixFilePermissionsメソッドの引数にPathを渡すと、LinuxやUnixで使われる形式で、ファイルのアクセス権を調べることができます。

最後に、ファイルを削除するには、Filesクラスの**deleteメソッド**を使います。このメソッドは、引数として渡されたPathのファイルやディレクトリを削除します。

例 実行結果

```
> java Sample
> data.txt
2021-03-16T13:25:50.707185799Z
[GROUP_READ, OTHERS_READ, OWNER_READ, OWNER_WRITE]
delete y/n? >y
deleted
```

このようにFilesクラスは、Pathを引数に受け取って処理するためのstaticメソッドを持っているクラスです。

なお、前述のコードではファイルを削除する際に、existsメソッドでファイルが存在するかを確かめてからdeleteメソッドを実行していましたが、次のように**deleteIfExistsメソッド**を使えば、存在確認と削除を一度に行うことができます。

例 deleteIfExistsメソッドの使用例

```
Path fake = Paths.get("fake.txt");
try {
    Files.deleteIfExists(fake);
} catch (IOException e) {
    // ファイルは存在したが、何らかの理由で削除できなかった
    throw new RuntimeException(e);
}
```

このコードを実行すると、ファイルが存在すれば削除されますし、たとえ存在しなくても例外などがスローされることはありません。

試験対策

existsメソッドは、単にファイルが存在するかどうかを確認するメソッドです。deleteIfExistsメソッドは、ファイルの存在を確認し、存在すれば削除するという一連の処理を行うメソッドです。

14. B

➡ P240

resolveメソッドでパスを解決する方法について問う問題です。
java.nio.file.Filesクラスで、新しいファイルを作るにはcreateFileメソッドを使いましたが、ディレクトリを作るには**createDirectoriesメソッド**を使います。次のコード例では、カレントディレクトリにdir/subdirディレクトリが存在するかどうかをFilesのexistsメソッドで調べ、なければcreateDirectoriesメソッドでディレクトリを新しく作っています。

例 createDirectoriesメソッドとresolveメソッドの使用例

```
1. import java.nio.file.Files;
2. import java.nio.file.Path;
3. import java.nio.file.Paths;
4.
5. public class Sample {
6.     public static void main(String[] args) throws Exception {
7.         Path dir = Paths.get("dir", "subdir");
8.         if (Files.exists(dir) == false) {
9.             Files.createDirectories(dir);
10.        }
11.        Path a = Paths.get("data.txt");
12.        Files.createFile(dir.resolve(a));
13.    }
14. }
```

なお、新しくディレクトリを作るには、前述のcreateDirectoriesメソッドのほかにも、**createDirectoryメソッド**が用意されています。2つのメソッドの違いは、親ディレクトリが存在しなくてもサブディレクトリを作るのがcreateDirectoriesメソッドで、親ディレクトリがなければ例外をスローするのがcreateDirectoryメソッドであるということです。

たとえば、次のコード例のように、存在しないtestディレクトリ内にsampleディレクトリを作ろうとすると、**java.nio.file.NoSuchFileException**がスローされます。

```
Path dir = Paths.get("test", "sample");
Files.createDirectory(dir);
```

また、createDirectoryメソッドは、親ディレクトリが存在しないときに例外をスローするだけでなく、同じディレクトリがすでに存在しているときにも**java.nio.file.FileAlreadyExistsException**という例外を発生させます。
その点、もう1つの**createDirectoriesメソッド**は、親ディレクトリがなければ作り、同じディレクトリが存在すれば何もしません。そのため、より厳密な操作を必要としないのであれば、createDirectoriesメソッドのほうが使い勝手はよいでしょう。

既存のディレクトリ内のパスを表現する場合は、次のようにディレクトリのパスを含めたPathを作ります。

```
Path file = Paths.get("dir", "subdir", "data.txt");
```

Pathsの**getメソッド**は、2つの引数を受け取ります。1つ目は単一の文字列ですが、2つ目は文字列の可変長引数です。Pathsのgetメソッドの宣言は次のとおりです。

```
public static Path get(String first, String... more)
```

可変長引数は、いくつでも引数を渡すことができ、引数なしでもかまいません。そのため、これまで1つしか引数を渡していなくても動作していたのです。この方法は、柔軟性を確保するには有効な方法ですが、同じディレクトリ内に複数のファイルを作るときにコードが煩雑になってしまうというデメリットもあります。たとえば、「a/b/c/d」というディレクトリ内にあるa.txtとb.txtという2つのファイルを表すには、次のように記述しなければいけません。

```
Path a = Paths.get("a", "b", "c", "d", "a.txt");
Path b = Paths.get("a", "b", "c", "d", "b.txt");
```

そこで、Pathsの**resolveメソッド**を使って、ディレクトリだけを表すパスとファイルを表すパスを別々に作り、必要に応じてパスを結合します。たとえば、次のように記述することで、dir/subdir/data.txtというパスを作ることができます。


```
Path dir = Paths.get("dir", "subdir");
Path file = dir.resolve(Paths.get("data.txt"));
```

このように、ディレクトリ部分のパスを使い回しできるようにすると、多くのファイルを同じディレクトリ内に作るコードが簡潔になります。以上のことから、選択肢**B**が正解です。

15. A、D

➡ P240

Filesクラスのファイルのコピーと移動に関する問題です。
java.nio.file.Filesクラスを使うと、ファイルのコピーや移動も簡単に行えます。たとえば次のコードは、Filesのcopyメソッドを使ってファイルをコピーし、moveメソッドを使ってファイルを移動している例です。

例 copyメソッドとmoveメソッドの使用例

```
1. import java.nio.file.Files;
2. import java.nio.file.Path;
3. import java.nio.file.Paths;
4.
5. public class Sample {
6.     public static void main(String[] args) throws Exception {
7.         Path sample = Paths.get("dir", "sample.txt");
8.         if (Files.exists(sample) == false) {
9.             Files.createFile(sample);
10.        }
11.
12.        Path outFile = Paths.get("dir", "sample_bk.txt");
13.        Files.copy(sample, outFile);
14.
15.        Path dir = Paths.get("dir", "subdir");
16.        Path dest = dir.resolve(Paths.get("sample.txt"));
17.        Files.move(sample, dest);
18.    }
19. }
```

ファイルをコピーするために、Filesクラスの**copyメソッド**を使っています。このメソッドは、第1引数にコピー元へのパス、第2引数にコピー先へのパスを指定します。前述のコード例を実行すると、dir/sample.txtをコピーしたdir/sample_bk.txtというファイルが作られます。

ファイルを移動するには、Filesクラスの**moveメソッド**を使います。使い方はcopyメソッドと同じですが、copyメソッドと違うのは、ファイルのコピー後にコピー元のファイルは消されてしまうことです。dir/sample.txtはdir/subdir/sample.txtとして移動されます。

このように、ファイルのコピーや移動はjava.ioパッケージを使った場合よりも格段にコード量を減らすことができます。
以上のことから、選択肢**A**と**D**が正解です。

16. B

➡ P241

Filesクラスの便利な機能の1つに、ファイル内のデータにアクセスするストリームを作り出すためのメソッドがあります。たとえば、次のコード例は、Filesの**newBufferedReaderメソッド**を使って、バッファから読み込むためのストリームを作っています。解答4で学んだ従来の入出力の扱い方に比べて、格段にコードが減ったことがわかります。

```
Path sample = Paths.get("sample.txt");
BufferedReader br = Files.newBufferedReader(sample);
try(br) {
    br.lines().forEach(System.out::println);
}
```

読み込みと同様に、書き込みも簡単にできるようになっています。書き込みのためのバッファも、Filesクラスの**newBufferedWriterメソッド**を使うだけです。

```
Path sample = Paths.get("sample.txt");
BufferedWriter out = Files.newBufferedWriter(sample);
try (out) {
    out.write("hello");
    out.newLine();
}
```

なお、書き込みの場合はオプションを追加することができます。オプションは、**java.nio.file.OpenOptionインタフェース型**で指定します。このインタフェースは、ファイルを開く場合や作成する場合などのオプションを指定するためのものですが、実際には、このインタフェースを実装した列挙型**java.nio.file.StandardOpenOption**を使います。この列挙型には、主に次のような定数が定義されています。

【StandardOpenOptionの主な定数】

定数	意味
APPEND	追記モードでファイルを開く
CREATE	ファイルが存在しない場合は新しいファイルを作成する
CREATE_NEW	新しいファイルを作成し、ファイルがすでに存在する場合は失敗する
DELETE_ON_CLOSE	閉じるときに削除する
READ	読み込みアクセス用に開く

前述のコードを次のように修正すると、ファイルに追記することができます。

```
Path sample = Paths.get("sample.txt");
BufferedWriter out
    = Files.newBufferedWriter(
            sample,
            StandardOpenOption.APPEND);
try (out) {
    out.write("hello");
    out.newLine();
}
```

以上のことから、選択肢**B**が正解です。

次のコード例は、コンソールから入力を受け付けて、ファイルの末尾に追加したあとに、そのファイルを読み込んで全行をコンソールに表示するものです。文字入力ストリームの扱いがかなり簡潔に、少ないコード量で記述できるようになったことがわかります。

例 追記モードでの書き込み

```
1. import java.io.BufferedReader;
2. import java.io.BufferedWriter;
3. import java.nio.file.Files;
4. import java.nio.file.Path;
5. import java.nio.file.Paths;
6. import java.nio.file.StandardOpenOption;
7. import java.util.Scanner;
8.
9. public class Sample {
```

```
10.     public static void main(String[] args) throws Exception {
11.         Path sample = Paths.get("dir", "sample.txt");
12.         if (Files.exists(sample) == false) {
13.             Files.createFile(sample);
14.         }
15.
16.         Scanner scan = new Scanner(System.in);
17.         BufferedWriter out =
18.                 Files.newBufferedWriter(
19.                         sample,
20.                         StandardOpenOption.APPEND);
21.         try (scan; out) {
22.             System.out.print("> ");
23.             out.write(scan.nextLine());
24.             out.newLine();
25.         }
26.
27.         BufferedReader in = Files.newBufferedReader(sample);
28.         try (in) {
29.             in.lines().forEach(System.out::println);
30.         }
31.     }
32. }
```

17. B

➡ P241

Filesクラスのメソッドに関する問題です。
java.io.Fileクラスの機能の1つとして、ディレクトリ内のファイルやディレクトリの一覧を戻すlistFilesというメソッドがあることを解答2で学びました。このメソッドを使うと、単純にすべてのディレクトリ内の要素を戻したり、java.io.FileFilterを使って絞り込みをした結果を戻すといったことができますが、これらはjava.nio.file.Filesを使えばもっと簡単に記述できます。次のコード例は、Filesクラスのlistメソッドとwalkメソッド、そしてfindメソッドを使い、srcというディレクトリ内の要素を扱っています。

例 listメソッド、walkメソッド、findメソッドの使用例

```
1. import java.nio.file.Files;
2. import java.nio.file.Path;
3. import java.nio.file.Paths;
4. import java.nio.file.attribute.BasicFileAttributes;
5. import java.util.function.BiPredicate;
6.
7. public class Sample {
8.     public static void main(String[] args) throws Exception {
9.         Path base = Paths.get(".");
10.        Files.list(base).forEach(System.out::println);
11.
12.        System.out.println("---");
13.        Files
14.            .walk(base.resolve("src"))
15.            .forEach(System.out::println);
16.
17.        System.out.println("---");
18.        BiPredicate<Path, BasicFileAttributes> bp = (p, attr) -> {
19.            return p.toFile().getName().endsWith(".jpg");
20.        };
21.        Files
22.            .find(base, 3, bp)
23.            .forEach(System.out::println);
24.    }
25. }
```

単純にディレクトリ内のすべてのパスを取得するなら、Filesクラスの**listメソッド**を使います。このメソッドは、引数で渡されたディレクトリ内のディレクトリやファイルを表すPathオブジェクトの一覧をStreamオブジェクトとして戻します。
コード例では、StreamオブジェクトのforEachメソッドを使って繰り返しコンソールに表示しています。そのため、この部分の実行結果は次のようになります。

例 実行結果

```
./sample.ser
./test.ser
./bin
… 中略 …
./data.txt
./sample.jpg
./src
```

このように、Filesクラスのlistメソッドは、引数で渡したディレクトリ直下にあるディレクトリやファイルの一覧を戻しますが、引数で渡したディレクトリの中にさらにサブディレクトリがあっても、その中まで再帰的に探索するわけではありません。サブディレクトリまで展開したい場合には、walkメソッドを使います。コード例では、カレントディレクトリ直下にある「src」ディレクトリ内を再帰的に展開しています。そのため、このコード部分の実行結果は次のようになります。

例 実行結果

```
./src
./src/ex9
./src/ex9/Sample.java
./src/ex7
./src/ex7/Sample.java
./src/ex7/PostalCode.java

… 中略 …

./src/ex2
./src/ex2/Sample.java
./src/ex5
./src/ex5/Sample.java
```

Filesクラスの**walkメソッド**は、ディレクトリ内を**再帰的**に処理してくれますが、すべてのファイルとディレクトリを扱うことにもなってしまいます。指定したディレクトリ内のサブディレクトリまで再帰的に処理しつつ、必要なものだけに絞り込みたい場合には、Filesクラスの**findメソッド**を使いましょう。前述のコード例では、findメソッドの引数にカレントディレクトリへのパスを持つPathと、再帰的に処理するディレクトリの深さ、そして絞り込み条件を渡しています。

絞り込み条件を指定するには、java.util.function.BiPredicate型のラムダ式を使います。第1引数には処理中のパスを表すPathオブジェクト、第2引数にはファイルの属性を扱うためのjava.nio.file.attribute.BasicFileAttributesを受け取ります。
コード例では、引数として受け取ったPathを**toFileメソッド**でjava.io.File型に変換し、**getNameメソッド**で名前を取得して、戻り値のStringクラスの**endsWithメソッド**で名前が「.jpg」で終わっているかどうかの真偽値を戻しています。

これで、拡張子「.jpg」ファイルのパスだけの一覧を持つStreamオブジェクトが、findメソッドの戻り値として戻されます。そのため、この部分のコードを実行した結果は、次のようになります。

例 実行結果

```
./sample_bk.jpg
./sample.jpg
```

以上のことから、選択肢**B**が正解です。選択肢Cも再帰的に探索した結果を戻しますが、すべての結果を戻すわけではないため、誤りです。

なお、ディレクトリ内のファイルやディレクトリの一覧を扱うときに拡張for文を利用する場合は、Filesクラスの**newDirectoryStreamメソッド**を使いましょう。このメソッドは、拡張for文で繰り返し処理をするためのjava.lang.Iterableのサブインタフェースである**java.nio.file.DirectoryStream型**のオブジェクトを戻します。

例 newDirectoryStreamメソッドの使用例

```
1. import java.nio.file.DirectoryStream;
2. import java.nio.file.Files;
3. import java.nio.file.Path;
4. import java.nio.file.Paths;
5.
6. public class Sample {
7.     public static void main(String[] args) throws Exception {
8.         Path base = Paths.get(".");
9.         DirectoryStream<Path> dirs
10.             = Files.newDirectoryStream(base);
11.         for (Path p : dirs) {
12.             System.out.println(p.getFileName());
13.         }
14.     }
15. }
```

listメソッドとの使い分けは、mapメソッドやfilterメソッドなどのストリームAPIを利用するならStream型を戻すlistメソッド、その必要がなく、拡張for文で処理したいのであればnewDirectoryStreamメソッドと覚えておきましょう。

18. E

➡ P241

解答17ではjava.nio.file.Filesを使って、ディレクトリの内容を扱う基本的な方法について学びました。ここでは、ディレクトリの内容に対して、より詳細な処理をするためにFilesクラスのwalkFileTreeメソッドについて学びましょう。

次のコード例では、dir/dataディレクトリに3つの新しいファイルを作り、これらが作成されたことを確認するために一覧表示しています。その後、まずファイルを、次にディレクトリを削除しています。このとき、ディレクトリ内にファイルがあると削除できないため、先にファイルを削除しているのです。

例 walkFileTreeメソッドの使用例

```
 1. import java.io.IOException;
 2. import java.nio.file.FileVisitResult;
 3. import java.nio.file.Files;
 4. import java.nio.file.Path;
 5. import java.nio.file.Paths;
 6. import java.nio.file.SimpleFileVisitor;
 7. import java.nio.file.attribute.BasicFileAttributes;
 8.
 9. public class Sample {
10.     public static void main(String[] args) throws Exception {
11.
12.         Path dir = Paths.get("dir", "data");
13.         if (Files.exists(dir) == false) {
14.             Files.createDirectories(dir);
15.         }
16.
17.         Files.createFile(dir.resolve(Paths.get("a.txt")));
18.         Files.createFile(dir.resolve(Paths.get("b.txt")));
19.         Files.createFile(dir.resolve(Paths.get("c.txt")));
20.
21.         Files.list(dir).forEach(System.out::println);
22.
23.         Files.walkFileTree(dir, new SimpleFileVisitor<Path>() {
24.             @Override
25.             public FileVisitResult visitFile(
26.                     Path file,
27.                     BasicFileAttributes attrs) throws IOException {
28.                 Files.delete(file);
29.                 return FileVisitResult.CONTINUE;
30.             }
31.         });
32.         Files.delete(dir);
33.         System.out.println("deleted");
34.     }
35. }
```

このときに使っているのが、**FilesクラスのwalkFileTreeメソッド**です。このメソッドは、指定されたディレクトリ内を再帰的に移動しながら、次の4つのタイミングで処理を実行します。

1. ディレクトリに入るとき
2. ディレクトリから出るとき
3. ファイルに遭遇したとき
4. ファイルの処理に失敗したとき

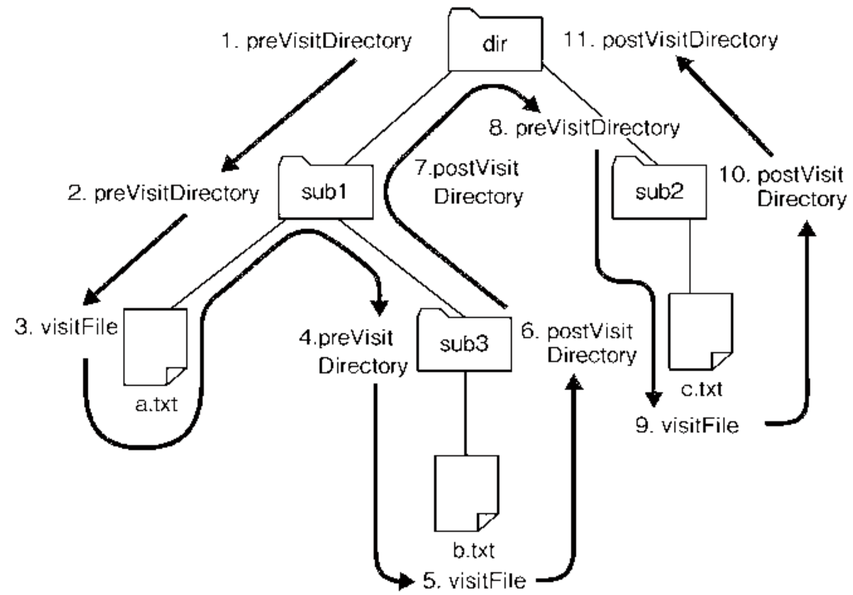
これらのタイミングに、どのような処理をすべきかを規定するのが、**java.nio.file.FileVisitorインタフェース**です。このインタフェースには、前述のタイミングに合わせて、次の**4つの抽象メソッド**が定義されています。

【FileVisitorの抽象メソッド】

タイミング	抽象メソッド
1.	preVisitDirectory(T dir, BasicFileAttributes attrs)
2.	postVisitDirectory(T dir, IOException exc)
3.	visitFile(T file, BasicFileAttributes attrs)
4.	visitFileFailed(T file, IOException exc)

たとえば、dir、sub1、sub2、sub3という4つのディレクトリと、a.txt、b.txt、c.txtという3つのファイルが次の図のように配置されていたとき、次の順番でFileVisitorのメソッドは呼び出されます。訪問者が、次々とディレクトリやファイルを尋ねて歩きながら、あらかじめ決められた処理をしていくようなイメージです。

【FileVisitorでディレクトリやファイルを処理するイメージ】



なお、それぞれのメソッドはEnumの**java.nio.file.FileVisitResult**で定義された定数を返します。ファイルの探索を続けるのであれば**CONTINUE**、終了するのであれば**TERMINATE**を返します。すべてのディレクトリやファイルを次々と処理していきたいのであれば、**CONTINUE**を戻します。

今回のようにファイルを削除したいだけであれば、**visitFileメソッド**だけが必要ですが、FileVisitorインタフェースには4つの抽象メソッドが規定されているため、すべての実装をしなければいけません。

そこで、最低限の実装だけを持つ**java.nio.file.SimpleFileVisitorクラス**をオーバーライドして使います。このクラスが持つ4つのメソッドでは、引数のnullチェックだけを行い、nullなら例外をスローし、そうでなければFileVisitResult.CONTINUEを戻して処理を続行するだけの実装がされています。前述のコード例では、SimpleFileVisitorを継承した匿名クラスを定義し、ファイルを削除するようにvisitFileメソッドだけをオーバーライドしています。

このコード例を実行すると、dir/dataディレクトリが新たに作られ、その中にa.txt、b.txt、c.txtが作られてから、これら3つのファイルが削除され、最後にディレクトリも削除されます。

例 実行結果

```
> java Sample
dir/data/c.txt
dir/data/b.txt
dir/data/a.txt
deleted
```

誤っているものを選ぶ問題ですので、選択肢**E**が正解です。

FileVisitorインタフェースに定義されている4つの抽象メソッドが、それぞれどのタイミングで処理されるのかを覚えておきましょう。

6

JDBCによる データベース連携

- JDBCの概要
- 接続文字列
- データベースの接続と切断
- データの検索・挿入・更新・削除を行うSQL文の実行
- DriverManagerクラス
- PreparedStatementインタフェース
- CallableStatementインタフェース
- ResultSetインタフェース
- ストアド・プロシージャ

1. JDBCの説明として、正しいものを選びなさい。（1つ選択）

A. データ操作のためのインタフェースだけが提供されている
B. DBMSとの連携時に発生する例外クラスが提供されている
C. DBMSと連携するための実装クラスが含まれている
D. 特定のDBMSを操作するためのインタフェースが提供されている

➡ P301

2. JDBCを利用してデータベースに接続するためのURLとして、正しいものを選びなさい。（1つ選択）

A. jdbc:postgresql:localhost:test
B. jdbc/oracle/localhost/test
C. jdbc:mysql://localhost/test
D. jdbc://db2@localhost/test

➡ P303

3. JDBCを使用してデータベースと接続するクラス、およびメソッドとして正しいものを選びなさい。（1つ選択）

A. ConnectionクラスのgetConnectionメソッド
B. ConnectionManagerクラスのgetConnectionメソッド
C. DriverクラスのgetConnectionメソッド
D. DriverManagerクラスのgetConnectionメソッド

➡ P305

4. JDBCを利用してDBMSと連携するプログラムを作っている。DBMSとの接続を解除するためのメソッドとして、正しいものを選びなさい。（1つ選択）

A. ResultSetのcloseメソッド
B. ResultSetのfinalizeメソッド
C. Connectionのcloseメソッド
D. Connectionのfinalizeメソッド

➡ P307

5. JDBCで提供されているインタフェースのうち、パラメータを受け取らないSQLを実行するためのものを選びなさい。（1つ選択）

A. java.sql.Statement
B. java.sql.PreparedStatement
C. Java.sql.CallableStatement
D. java.sql.NonParameterizedStatement

➡ P310

6. PreparedStatementオブジェクトを取得する方法として、正しいものを選びなさい。なお、変数conはConnectionオブジェクトとする。（1つ選択）

A. `PreparedStatement ps = con.prepareStatement();`
B. `PreparedStatement ps = con.prepareStatement("select * from emp");`
C. `var ps = con.getPreparedStatement("select * from emp");`
D. `var ps = con.createPreparedStatement("select * from emp");`

➡ P311

7. 次のコードをコンパイル、実行したときの結果として、正しいものを選びなさい。（1つ選択）

```
var sql = "delete from item where id = ?";
try (var ps = con.prepareStatement(sql)){
    ps.setInt(0, 1);
    ps.executeUpdate();
}
```

A. プログラムは正常に動作し、データベースから1件のデータが削除される
B. プログラムは正常に動作し、データベースからデータが削除されることはない
C. コンパイルエラーが発生する
D. 実行時に例外がスローされる

➡ P312

8. 次のコードをコンパイル、実行したときの結果として、正しいものを選びなさい。(1つ選択)

```
var sql = "select * from item where id = ?";
try (var ps = con.prepareStatement(sql)){
    ResultSet rs = ps.executeQuery();
    // do something
}
```

A. 0件の検索結果が戻される
B. 全件の検索結果が戻される
C. コンパイルエラーが発生する
D. 実行時に例外がスローされる

➡ P314

9. JDBCを使ったデータベースプログラミングをしている。UPDATE文を実行した結果、何件更新されたかを調べたい。PreparedStatementのメソッドとして、最も適切なものを選びなさい。(1つ選択)

A. executeUpdateメソッド
B. executeQueryメソッド
C. executeメソッド
D. executeBatchメソッド

➡ P314

10. 次のコードをコンパイル、実行したときの結果として、正しいものを選びなさい。(1つ選択)

```
var sql = "delete from item where id = ?";
try (var ps = con.prepareStatement(sql)){
    ps.setInt(1, 1);
    ps.executeUpdate("update item set name='test' where id = ?");
}
```

A. DELETE文が実行される
B. UPDATE文が実行される
C. コンパイルエラーが発生する
D. 実行時に例外がスローされる

➡ P316

11. 次のコードをコンパイル、実行したときの結果として、正しいものを選びなさい。(1つ選択)

```
var sql = "select * from emp";
try (PreparedStatement ps = con.prepareStatement(sql)){
    ResultSet rs = ps.executeQuery();
    System.out.println(rs.getString(2));
}
```

なお、検索する対象となるempテーブルは、以下のレコードが登録されているものとする。

ID	NAME	DEPARTMENT
1	ALLEN	R&D
2	SCOTT	SALES
3	BILL	ACCOUNTING

A. 「1」と表示される
B. 「2」と表示される
C. 「ALLEN」と表示される
D. 「SCOTT」と表示される
E. コンパイルエラーが発生する
F. 実行時に例外がスローされる

➡ P316

12. 次のコードをコンパイル、実行したときの結果として、正しいものを選びなさい。(1つ選択)

```
var sql = "select count(*) from item";
try (PreparedStatement ps = con.prepareStatement(sql)){
    System.out.println(ps.execute());
}
```

なお、検索する対象となるitemテーブルは、以下のレコードが登録されているものとする。

id	name
1	banana
2	apple
3	orange

※次ページに続く

A. 「false」と表示される
B. 「true」と表示される
C. 「2」と表示される
D. 「3」と表示される
E. コンパイルエラーが発生する
F. 実行時に例外がスローされる

➡ P319

13. JDBCを利用したプログラムで、一度に複数のSQL文を実行するためのメソッドとして、正しいものを選びなさい。(1つ選択)

A. PreparedStatementインタフェースのexecuteMultiメソッド
B. PreparedStatementインタフェースのexecuteUpdateメソッド
C. Statementインタフェースのexecute Batchメソッド
D. Statementインタフェースのexecute Multiメソッド

➡ P322

14. データベースに登録済みのストアド・プロシージャsample_procを実行するプログラムを作成している。以下の「/* insert code here */」に入るコードとして、正しいものを選びなさい。(1つ選択)

```
try (/* insert code here */) {
    cs.setString(1, "test");
    cs.setInt(2, 1);
    cs.execute();
}
```

A. var cs = con.prepareCall("sample_proc(?, ?)");
B. var cs = con.prepareCall("call sample_proc(?, ?)")
C. var cs = con.prepareStatement("sample_proc(?, ?)");
D. var cs = con.prepareStatement("call sample_proc(?, ?)");

➡ P323

第6章 JDBCによるデータベース連携
解 答

1. B

➡ P296

JDBCの概要に関する問題です。
データベースとは、データの集合を扱うソフトウェアのことです。たくさんのデータが集まったときにデータを管理し、操作するためのソフトウェアのことを**DBMS**(データベースマネジメントシステム)と呼びます。DBMSは、一般的に次のような機能を持っています。

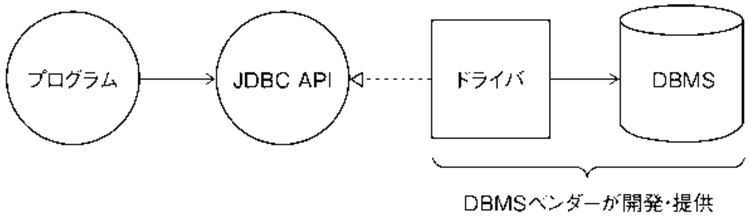
・いつでも使いたいときに使いたいデータを検索する
・新しいデータを重複なく追加する
・既存のデータをすみやかに更新、削除する
・複数のデータ間の関係性に不整合が起きないように制御する
・複数のユーザーが同時にアクセスしても、データの不整合が発生しないように制御する

また、大量のデータを扱うには、そのデータをどのように管理するかという管理方法が必要です。この管理方法のことを「**データモデル**」といいます。古くは「階層型データモデル」と「ネットワーク型データモデル」から始まり、現在主流の「リレーショナルデータモデル」など、さまざまなデータモデルが存在します。
なお、リレーショナルデータモデルを使ったデータの集合を「**RDB**(リレーショナルデータベース)」、そのデータベースを管理するソフトウェアのことを「**RDBMS**(リレーショナルデータベースマネジメントシステム)」と呼びます。

DBMS製品が登場するまで、データベースを管理するためのプログラムは、そのソフトウェアを開発するプログラマーが作らなければいけませんでした。当時、並べ替えや探索といったアルゴリズムは、データベースを処理するためのプログラム開発に欠かせない技術だったのです。しかし、複雑なアルゴリズムを作るのは多大な労力を要し、さらに品質を担保するのが難しい作業でもありました。そこに、大規模なデータベースを簡単に扱うことができるリレーショナルデータモデルを採用したRDBMS製品をさまざまなベンダーが提供し始め、瞬く間に普及していくことになります。
残る課題は、DBMS製品ごとに連携方法や操作手順などが異なる点です。そこで、JavaのプログラムからDBMSを利用するときに生じる製品ごとの違いを吸収するために作られたのが、**JDBC**(**Java DataBase Connectivity**)です。

JDBCでは、どのDBMS製品であっても同じ方法で利用できるようにするために、インタフェースと例外クラス、もしくはすべてのDBMSで共通して利用できるクラスなど、共通部分だけを提供しています（選択肢A、**B**、D）。これらは、java.sqlモジュールの**java.sqlパッケージ**や**javax.sqlパッケージ**として提供されています。なお、インタフェースを実装したクラスは、DBMS製品を作ったベンダーが「**ドライバ**」として提供しています（選択肢C）。

【JDBC】



利用者は、ベンダーからドライバをダウンロードして、クラスパス上に配置すれば、あとはインタフェースに沿った方法で利用するだけなので、DBMS製品ごとの違いを意識せずに済むのです。

以上のことから、選択肢**B**が正解となります。JDBCでは、インタフェースだけが提供されているわけではありません。よって、選択肢Aは誤りです。また、実際にDBMSと連携するクラスはDBMSベンダーが提供します。よって、選択肢Cは誤りです。JDBCでは、どのDBMS製品であっても共通して扱えるインタフェースなどを提供し、特定のDBMS製品のための機能を提供しているわけではありません。よって、選択肢Dも誤りです。

本書では、DBMSとして**Apache Derby**を利用します。このDBMSは、IBM社がJavaで開発していたDBMSをオープンソース団体のApache Software Foundationに寄贈し、Java SE 6の頃にJDKに同梱され、配布されていたものです。Javaのアプリケーションとして開発されているため、Javaが動作する環境であれば、インストール等の作業をしなくても使うことができるのが特徴です。
なお、現在はライセンスの関係でJDKに同梱されなくなりましたが、下記のURLから無償でダウンロードすることができます。

【Apache DerbyのWebサイト】
http://db.apache.org/derby/

zipファイルをダウンロードしたら、展開してください。展開先のディレクトリを本書では＜DERBY_HOME＞と記述します。展開できたら、Windowsのコマンドプロンプトやmacosのターミナルで＜DERBY_HOME＞まで移動し、以下のjavaコマンドを実行してみましょう。


```
> java -jar lib/derbyrun.jar server start
Mon Sep 14 11:30:37 JST 2020 : セキュリティ・マネージャがBasicサーバー・
セキュリティ・ポリシーを使用してインストールされました。
Mon Sep 14 11:44:03 JST 2020 : Apache Derby Network Server -
10.15.2.0 - (1873585)が起動し、ポート1527で接続の受入れ準備が完了しました
```

最初の「セキュリティ・マネージャが…」はすぐに表示されますが、そこからサーバーが起動し、待ち受け状態になったことを示す「ポート1527で接続の受入れ準備が完了しました」が表示されるまでは数十秒～数分かかる場合があります。表示されるまで、そのまま少し待ちましょう。
解答2以降では、Apache Derbyを使いながら学習を進めます。

2. C

➡ P296

接続文字列の書式について問う問題です。
解答1では、DBMSとしてApache Derbyのセットアップをしました。サーバーが無事に起動し、待ち受け状態になったことが確認できたら、次にコマンドラインクライアントを起動してみましょう。このアプリケーションは、コンソールからSQLの入力を受け付けて、結果を表示するための簡易クライアントアプリケーションです。

```
> java -jar lib/derbyrun.jar ij
ijバージョン10.15
ij>
```

コマンドラインクライアントが起動したら、次の**connectコマンド**でサーバーに接続してみましょう。

```
ij> connect 'jdbc:derby://localhost:1527/data/Sample;create=true';
```

connectコマンドの引数に渡している文字列が「**接続文字列**」と呼ばれるものです。この接続文字列は、どのマシンの、どのポート番号で、どんなデータベースを扱うサーバーに接続するかを指定したものです。今回の接続文字列では、localhostの1527番ポートで待ち受けているサーバーに、dataディレクトリ下のSampleというデータベースを読み込ませるという指定になっています。よって、選択肢**C**が正解です。なお、選択肢Cはポート番号を省略した書式です。

※次ページに続く

【接続文字列の書式（DBMSがDerbyの場合）】

jdbc:derby://<サーバーのIPアドレス>:<ポート番号>/データベースの場所

なお、前述の接続文字列では、データベースの場所の後ろに、セミコロンに続けて「create=true」という独自のパラメータを渡しています。このパラメータは、指定したデータベースの場所にデータベースがなかった場合、新規に作成することを意味します。そのため、このコマンドを最初に実行すると、<DERBY_HOME>にdataというディレクトリが作られ、その中にSampleというサブディレクトリ、さらにその中に、データベースに必要なファイル群が新規に作られているはずです。

データベースに接続ができたら、次のようにSQLでテーブルと新しいデータを挿入してみましょう。この例では、idとnameという2つの列を持ったitemテーブルを作成し、select文でデータが0件であることを確認したあと、insert文で新規にデータを1件挿入、その後、もう一度select文を実行して、データが登録されていることを確認しています。

例 テーブルの新規作成とデータの挿入、確認

```
ij> create table item (id int primary key, name varchar(20));
0行が挿入/更新/削除されました
ij> select * from item;
ID         |NAME
--------------------------------

0行が選択されました
ij> insert into item values(1, 'sample');
1行が挿入/更新/削除されました
ij> select * from item;
ID         |NAME
--------------------------------
1          |sample

1行が選択されました
```

試験対策

接続文字列の書式を覚えておきましょう。

3. D

➡ P296

JDBCのAPIに関する問題です。
前問の解説で、DBMSであるDerbyとデータベースの準備が整いました。次は、Javaのプログラムから DBMSに接続してみましょう。

JavaのプログラムからDBMSに接続し、操作するには、java.sqlパッケージやjavax.sqlパッケージに含まれるインタフェース群を使います。DBMSとの接続を管理するためのインタフェースが、**java.sql.Connection**です。このインタフェースの実装クラスのインスタンスへの参照は、**java.sql.DriverManagerクラスのgetConnectionメソッド**を使って取得します。このDriverManagerは、クライアントプログラムに代わって、クラスパス上にあるDBMSベンダーから提供されたドライバを検索し、DBMSに接続するクラスです。DriverManagerクラスのgetConnectionメソッドは、DBMSへの接続文字列を引数に受け取り、その情報に基づいて接続を確立します。以上のことから、選択肢**D**が正解となります。その他の選択肢のクラスやメソッドは存在しません。

次のコード例が、DriverManagerクラスを使ってConnection型のインスタンスへの参照を取得した例です。

例 DBMSとの接続

```
 1. import java.sql.Connection;
 2. import java.sql.DriverManager;
 3. import java.sql.SQLException;
 4.
 5. public class Sample {
 6.     public static void main(String[] args) {
 7.         try {
 8.             Connection con = DriverManager.getConnection(
 9.                 "jdbc:derby://localhost:1527/data/Sample");
10.
11.             System.out.println(con);
12.             con.close();
13.
14.         } catch (SQLException e) {
15.             throw new RuntimeException(e);
16.         }
17.     }
18. }
```

※次ページに続く

では、このSampleクラスをコンパイル、実行してみましょう。コンパイルはできても、次のように実行時に例外がスローされるはずです。この例外は、Connectionインタフェースは存在するものの、探した実装クラスがクラスパス上になかったために起きています。

例 実行結果

```
> java Sample
Exception in thread "main" java.lang.RuntimeException: java.sq
l.SQLException: No suitable driver found for jdbc:derby://loca
lhost:1527/data/Sample
    at Sample.main(Sample.java:14)
Caused by: java.sql.SQLException: No suitable driver found for
jdbc:derby://localhost:1527/data/Sample
    at java.sql/java.sql.DriverManager.getConnection(DriverMan
ager.java:702)
    at java.sql/java.sql.DriverManager.getConnection(DriverMan
ager.java:251)
    at Sample.main(Sample.java:8)
```

解答1で説明したとおり、JDBCではインタフェースや例外クラスだけが提供されており、実際に動作するインタフェースを実装したクラスは、DBMSベンダーがドライバとして提供しています。上記の例外は、Derbyが提供しているドライバをクラスパスに指定しなかったために起きているのです。

そこで、次のjavaコマンドのように、-cpオプションでDerbyのドライバであるjarファイルにクラスパスを指定してみましょう。なお、Derbyのドライバは**derbyclient.jar**ですが、このjarファイル内のクラスが依存するクラスが**derbyshared.jar**に含まれているので、両方にクラスパスを指定する必要があります。

```
> java -cp .;derbyclient.jar;derbyshared.jar Sample
org.apache.derby.client.net.NetConnection@7006c658
```

コードを実行すると、今度は例外がスローされることなく、上記のような結果が表示されるはずです。なお、表示されているorg.apache.derby.client.net.NetConnectionが、java.sql.Connectionインタフェースを実装したDerby独自のクラスです。この独自のクラスは、利用するDBMS製品ごとに異なりますが、インタフェースは変わらないので、DBMSごとの違いを意識してプログラミングする必要はありません。

Java SE 6よりも古いJavaのプログラムでは、Connectionオブジェクトを取得する前に、ドライバクラスのロードが必要でした。ドライバクラスのロードは、次のコードのようにClassクラスのforNameメソッドを使います。このメソッドの引数に、ロードしたいクラスの完全修飾クラス名を渡すと、クラスパスから該当するクラスを探し出して読み込むのです。

```
Class.forName("org.apache.derby.jdbc.ClientDriver");
```

Java SE 6(JDBC4.0)以降は、自動検索するようになったため、このようなコードは不要になりました。

4. C

➡ P296

ほとんどのDBMSは同時に接続できる数に限りがあるため、データベースとの接続は、利用が終わる度に閉じなくてはいけません。データベースとの接続を閉じるには、**Connectionインタフェース**の**closeメソッド**を使います。次のコード例では、例外が発生するかどうかにかかわらず接続を閉じられるように、**finallyブロック**でcloseメソッドを呼び出しています。

例 データベースとの接続を閉じる

```
 1. import java.sql.Connection;
 2. import java.sql.DriverManager;
 3. import java.sql.SQLException;
 4.
 5. public class Sample {
 6.     public static void main(String[] args) {
 7.         Connection con = null;
 8.         try {
 9.             con = DriverManager.getConnection(
10.                 "jdbc:derby://localhost:1527/data/Sample");
11.
12.             System.out.println(con);
13.
14.         } catch (SQLException e) {
15.             throw new RuntimeException(e);
16.         } finally {
17.             if (con != null) {
18.                 try {
```

```
19.                     con.close();
20.                 } catch (SQLException e) {
21.                     throw new RuntimeException(e);
22.                 }
23.             }
24.         }
25.     }
26. }
```

データベースとの接続を閉じるのであれば、次のように**try-with-resources**を使うほうがコードを簡潔に記述することができます。コード量が削減され、読みやすくなったことがわかります。

例 try-with-resourcesを使ってデータベースとの切断を閉じる

```
1. import java.sql.Connection;
2. import java.sql.DriverManager;
3. import java.sql.SQLException;
4.
5. public class Sample {
6.     public static void main(String[] args) {
7.         try (Connection con = DriverManager.getConnection(
8.                 "jdbc:derby://localhost:1527/data/Sample")) {
9.
10.             System.out.println(con);
11.
12.         } catch (SQLException e) {
13.             throw new RuntimeException(e);
14.         }
15.     }
16. }
```

以上のことから、選択肢**C**が正解です。なお、選択肢AのResultSetオブジェクトを閉じても、プログラムとDBMSとの接続は解除されません。
たとえば次のコードは、複数のidを繰り返しSQL文の条件に入れて実行する例です。ConnectionオブジェクトとPreparedStatementオブジェクトは、同じtry-with-resourcesで管理していますが、ResultSetオブジェクトについては入れ子になった別のtry-with-resourcesで管理しています。

例 ResultSetオブジェクトを閉じる

```
1. import java.sql.Connection;
2. import java.sql.DriverManager;
3. import java.sql.PreparedStatement;
4. import java.sql.ResultSet;
5. import java.sql.SQLException;
6.
7. public class Sample2 {
8.     public static void main(String[] args) {
9.         String url = "jdbc:derby://localhost:1527/data/Sample";
10.        String sql = "select * from item where id = ?";
11.
12.        try (Connection con = DriverManager.getConnection(url);
13.            PreparedStatement ps = con.prepareStatement(sql)) {
14.            for (String id : args) {
15.                ps.setInt(1, Integer.parseInt(id));
16.                try (ResultSet rs = ps.executeQuery()) {
17.                    while(rs.next()) {
18.                        System.out.println(
19.                                rs.getString("name"));
20.                    }
21.                }
22.            }
23.        } catch (SQLException e) {
24.            throw new RuntimeException(e);
25.        }
26.    }
27. }
```

ResultSetオブジェクトは、DBMSと連動してどこまで読み込んだのかというカーソル情報をやり取りしています。そのため、ResultSetオブジェクトをcloseして解放することで、DBMSが保持しているカーソル情報も同時に解放されます。このように、ResultSetオブジェクトの解放はカーソル情報の解放をすることが目的になるため、DBMSとの接続が解除されるわけではありません。以上のことから、選択肢Aは誤りです。

※次ページに続く

試験対策

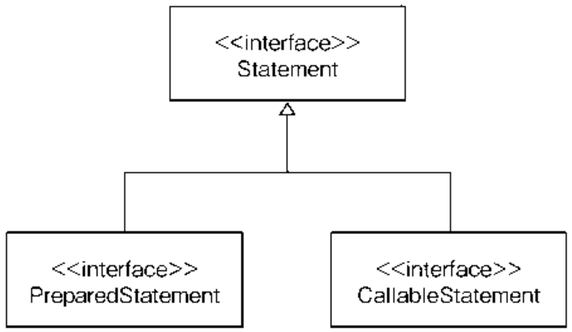
DBMSとの接続を自動的に解除するには、try-with-resourcesを使って、tryにConnectionオブジェクトを記述します。ResultSetオブジェクトを記述しても、DBMSが保持しているカーソル情報を解放するだけで、接続は解除されません。

5. A

➡ P297

JDBCのAPIに関する問題です。
解答3では、データベースとの接続を行いました。ここでは、SQL文を準備します。JDBCでは、次のクラス図のようにjava.sql.Statementインタフェースをベースとして、java.sql.PreparedStatementとjava.sql.CallableStatementの2つのサブインタフェースでSQL文を扱います。

【Statementインタフェースとサブインタフェース】



SQL文を扱うインタフェースが**PreparedStatement**、DBMSに登録されたストアド・プロシージャ（関数）を実行するのに使用されるインタフェースが**CallableStatement**と、それぞれ役割を担っています。SQL文を扱うのであれば、PreparedStatementを使うと覚えておきましょう。

StatementとPreparedStatementの違いは、パラメータを受け取るかどうかです。Statementは**パラメータなしのSQL文だけ**を実行するためのインタフェースで、PreparedStatementは**パラメータを受け取るSQL文**を実行するためのインタフェースです。通常のSQLは、検索対象のデータを絞り込むためのwhere句の条件や、新規データを挿入するときのデータ、既存データを更新するためのデータなどのようなパラメータを扱います。そのため、PreparedStatementを使うのが一般的です。
また、PreparedStatementは、DBMSが理解できるようにSQL文をあらかじめコンパイルしておきます。一方、StatementはSQL文を文字列のまま扱うため、そのSQL文を受け取ったDBMSがコンパイルしてから実行します。そのため、何度も実行するSQL文であれば、PreparedStatementのほうがパフォーマンスがよくなります。
これらの理由から、Statementを積極的に使う必要はありません。あくまでもPreparedStatementやCallableStatementをまとめるスーパーインタフェースだと理解しておきましょう。

以上のことから、選択肢**A**が正解となります。選択肢BのPreparedStatementはパラメータ付きのSQLを実行するためのもの、選択肢CのCallableStatementはストアド・プロシージャを実行するためのものです。また、選択肢Dのようなインタフェースは存在しません。よって、これらの選択肢は誤りです。

試験対策

Statementはパラメータを受け取らないSQL文を実行するためのインタフェース、PreparedStatementはパラメータを受け取るSQL文を実行するためのインタフェースです。

6. B

➡ P297

PreparedStatementに関する問題です。
Statementの2つのサブインタフェースのうち、まずは**PreparedStatementインタフェース**の利用方法を見ていきましょう。PreparedStatementの特徴は、Prepared（準備された）Statement（文）という名前からわかるように、DBMSがSQL文を理解できるように事前にコンパイルしておくことです。
SQL文をコンパイルし、PreparedStatementオブジェクトを作成するには、次のコード例のように**java.sql.Connectionインタフェース**の**prepareStatementメソッド**を使います。「prepare（準備する）」と、過去形ではないことに注意してください。

例 prepareStatementメソッドの使用例

```
1. import java.sql.Connection;
2. import java.sql.DriverManager;
3. import java.sql.SQLException;
4. import java.sql.PreparedStatement;
5.
6. public class Sample {
7.     public static void main(String[] args) {
8.         try (Connection con = DriverManager.getConnection(
9.                 "jdbc:derby://localhost:1527/data/Sample")) {
10.
11.             var sql = "select * from items";
```

```
12.             try (PreparedStatement ps = con.prepareStatement(sql)){
13.                 // do something
14.             }
15.
16.         } catch (SQLException e) {
17.             throw new RuntimeException(e);
18.         }
19.     }
20. }
```

ConnectionインタフェースのprepareStatementメソッドは、コンパイル対象のSQL文を引数として受け取ります。引数なしのオーバーロードはありませんので注意してください。
以上のことから、選択肢Aは誤りで、選択肢Bが正解となります。選択肢CやDのようなメソッドは存在しません。

7. D ➡P297

SQL文のパラメータ「?」とメソッドの引数指定に関する問題です。
PreparedStatementを利用するとき、SQL文にパラメータを渡したい場合、次のように置き換え文字としてクエスチョンマーク「**?**」を使います。

```
var sql = "insert into item values(?,?)";
```

置き換え文字の部分に値をセットするには、セットしたい列の型によってsetStringやsetIntといった「**setXXX**」という形式のメソッドを使います。次のコード例では、上記のINSERT文の置き換え文字の部分にパラメータをセットしています。

例 INSERT文にセットするパラメータをメソッドで設定する

```
1. import java.sql.Connection;
2. import java.sql.DriverManager;
3. import java.sql.PreparedStatement;
4. import java.sql.SQLException;
5.
6. public class Sample {
7.     public static void main(String[] args) {
8.         try (Connection con = DriverManager.getConnection(
9.                 "jdbc:derby://localhost:1527/data/Sample")) {
10.
11.             var sql = "insert into item values(?,?)";
12.             try (PreparedStatement ps = con.prepareStatement(sql)){
13.                 ps.setInt(1, 2);
14.                 ps.setString(2, "sample");
15.                 // do something
16.             }
17.
18.         } catch (SQLException e) {
19.             throw new RuntimeException(e);
20.         }
21.     }
22. }
```

PreparedStatementのsetXXXメソッドの第1引数には「何番目の？なのか」を、第2引数にはセットしたい値を渡します。なお、**第1引数に渡すのは1から始まる番号**です。配列の添字のように、0から始まる番号ではありませんので注意してください。
もし、第1引数に0を渡した場合、次のように実行時に例外がスローされます。

例 実行結果

```
Exception in thread "main" java.lang.RuntimeException: java.sq
l.SQLException: 列位置'0'は範囲外です。このResultSetに対する列数は'1'で
す。
    at Sample.main(Sample.java:19)
Caused by: java.sql.SQLException: 列位置'0'は範囲外です。このResult
Setに対する列数は'1'です。
… 以下省略 …
```

以上のことから、選択肢Dが正解となります。

試験対策

PreparedStatementを利用してSQL文にパラメータを渡したい場合、置き換え文字の「?」を使います。置き換え文字の部分には、「setXXX」形式（XXXはセットしたい値の型）のメソッドで値をセットします。setXXXメソッドの第1引数には「何番目の？なのか」を、第2引数にはセットしたい値を渡します。第1引数に渡すのは「1から始まる番号」です。

8. D ➡ P298

前問では、PreparedStatementのパラメータをセットするときは1から始めることを学びました。このパラメータは、すべてセットしなくてはいけません。セットし忘れても文法上の誤りではないためコンパイルエラーにはなりませんが、実行すると例外がスローされます。次のコード例では4行目をコメントアウトし、2つ目のパラメータをセットせずにSQLを実行しています。

例 2つ目のパラメータをセットしていない

```
var sql = "insert into item values(?,?)";
try (PreparedStatement ps = con.prepareStatement(sql)){
    ps.setInt(1, 2);
    // ps.setString(2, "sample");
    ps.executeUpdate();
}
```

このコードを実行すると次のような例外が発生し、スタックトレースが表示されます。

例 実行結果

```
Exception in thread "main" java.lang.RuntimeException: java.sq
l.SQLException: 現行文の少なくとも1つのパラメータが初期化されていません。
    at Sample.main(Sample.java:20)
… 以下省略 …
```

設問のSQLは、id列を検索条件に指定していますが、パラメータで値を指定していません。そのため、このコードを実行すると例外がスローされます。以上のことから、選択肢**D**が正解となります。

9. A ➡ P298

PreparedStatementインタフェースのexecuteUpdateメソッドに関する問題です。

PreparedStatementの継承元であるStatementインタフェースには、executeUpdateメソッド、executeQueryメソッド、executeメソッド、そしてexecuteBatchメソッドというSQLを実行するための4つのメソッドが用意されています。**executeUpdateメソッド**は、データの挿入、更新、削除を行うためのメソッドです。検索のためには使用できないことに注意してください。次のコードは、新規にデータを挿入するためのSQL文を実行する例です。

例 executeUpdateメソッドの使用例（データの挿入）

```
var sql = "insert into item values(?,?)";
try (PreparedStatement ps = con.prepareStatement(sql)){
    ps.setInt(1, 2);
    ps.setString(2, "sample");
    ps.executeUpdate();
}
```

executeUpdateメソッドは、実行した結果の件数をint型の値で戻します。INSERT文であれば通常は1件しか挿入しないため、次のようにUPDATE文で使用して、結果を表示してみましょう。

例 executeUpdateメソッドの使用例（データの更新）

```
var sql = "update item set name = ?";
try (PreparedStatement ps = con.prepareStatement(sql)){
    ps.setString(1, "hoge");
    int result = ps.executeUpdate();
    System.out.println(result);
}
```

解答7でデータを1件挿入してから、前述のINSERT文で新しいデータを1件挿入しているため、WHERE句で条件を指定していないUPDATE文を実行すると、2件ともデータが更新されます。よって、コンソールには「2」と表示されます。

例 実行結果

```
> java -cp .;derbyclient.jar;derbyshared.jar Sample
2
```

以上のことから、選択肢**A**が正解です。選択肢BのexecuteQueryメソッドは、検索した結果を戻すためのものです。よって、誤りです。選択肢CのexecuteメソッドはResultSetオブジェクトが戻ってきたかどうかをboolean型で戻します。よって、この選択肢も誤りです。選択肢DのexecuteBatchメソッドは複数のSQL文を一度に実行し、その結果、更新された数の配列を戻します。最も適切なメソッドを選ぶ問題なので、選択肢Dは誤りとなります。

10. D ➡ P298

設問のコードではPreparedStatementのexecuteUpdateメソッドにSQL文を渡していますが、コンパイルエラーにはなりません。これは、executeUpdateメソッドがPreparedStatementの継承元であるStatementインタフェースで定義されているメソッドだからです。
解答5で説明したとおり、StatementとPreparedStatementの違いは、事前にSQL文を（DBMSが理解できるように）コンパイルしておくかどうかです。PreparedStatementが事前にコンパイルしてからDBMSにSQL文を渡すのに対し、StatementはSQL文をそのままDBMSに渡します。そのため、**Statementのexecute Updateメソッド**は、実行するタイミングでSQL文を引数として受け取ります。PreparedStatementは、このメソッドを継承によって引き継いでいるため、**PreparedStatementのexecuteUpdateメソッド**の引数にSQL文を渡してもコンパイルエラーにはなりません。ただし、実行すると次のように例外がスローされます。

例 実行結果

```
Exception in thread "main" java.lang.RuntimeException: java.sq
l.SQLException: メソッド'executeUpdate(String)'は、プリペアド文では
許可されません。
    at Sample.main(Sample.java:19)
Caused by: java.sql.SQLException: メソッド'executeUpdate(String)'
は、プリペアド文では許可されません。
… 以下省略 …
```

以上のことから、選択肢**D**が正解となります。

11. F ➡ P299

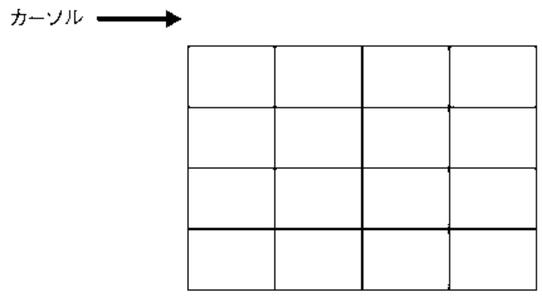
PreparedStatementインタフェースのexecuteQueryメソッドに関する問題です。データの挿入、更新、削除にはexecuteUpdateメソッドを使いますが、データを検索するには**executeQueryメソッド**を使います。更新件数を戻すexecuteUpdateメソッドに対し、executeQueryメソッドは検索結果のテーブルを戻します。次のコードは、データ検索のSQLを実行する例です。

例 executeQueryメソッドの使用例

```
1. import java.sql.Connection;
2. import java.sql.DriverManager;
3. import java.sql.PreparedStatement;
4. import java.sql.ResultSet;
5. import java.sql.SQLException;
6.
7. public class Sample {
8.     public static void main(String[] args) {
9.         try (Connection con = DriverManager.getConnection(
10.                 "jdbc:derby://localhost:1527/data/Sample")) {
11.
12.             var sql = "select * from item";
13.             try (PreparedStatement ps = con.prepareStatement(sql)){
14.                 ResultSet rs = ps.executeQuery();
15.                 while(rs.next()) {
16.                     System.out.print(rs.getInt(1) + ":");
17.                     System.out.println(rs.getString("name"));
18.                 }
19.             }
20.
21.         } catch (SQLException e) {
22.             throw new RuntimeException(e);
23.         }
24.     }
25. }
```

executeQueryメソッドが戻す検索結果の表を扱うためのインタフェースが、java.sqlモジュールの**java.sql.ResultSet**です。ResultSet型オブジェクトの概念は、次のような表とカーソルで表現できます。カーソルとは、現在、表のどの行を選択しているかを表すものです。最初、カーソルは表の外を指しています。

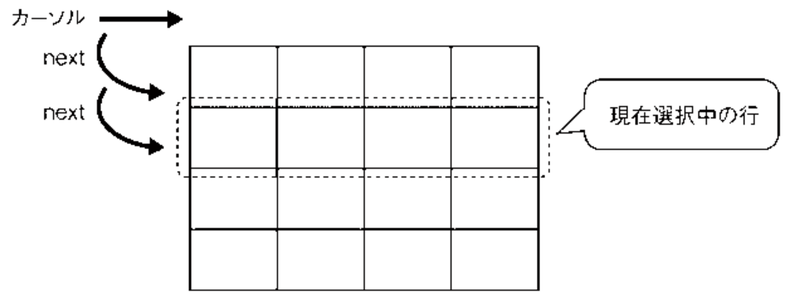
【カーソル】



このカーソルを動かして次の行を選択するのが、ResultSetの**nextメソッド**で

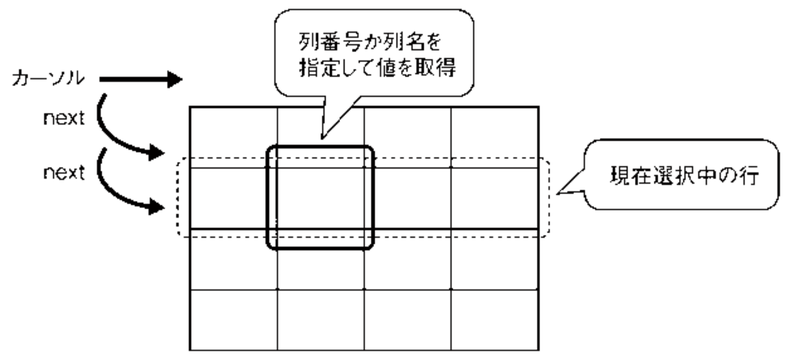
す。このメソッドを実行すると、カーソルが1行動きます。次の行が選択できればtrueを戻すため、前述のコード例のようにwhile文の条件式に利用できます。nextメソッドを繰り返し呼び出し、次に選択できる行がなくなればnextメソッドはfalseを戻し、while文から抜けます。

【nextメソッドでカーソルを1行移動させる】



nextメソッドによって行が選択できたら、次は列を指定して値を取り出します。列を指定するには、**取り出したい型に合わせたgetXXXメソッド**を使います。たとえば、String型で取り出したければ**getStringメソッド**を、int型で取り出したければ**getIntメソッド**という具合です。getXXXメソッドの引数は、列番号をint型で渡すか、列名をString型で渡します。なお、**列番号は0からではなく1から始まる**点に注意してください。

【選択している行から列を指定し、値を取り出す】



前述のコード例の抜粋を再掲します。このコードではgetIntメソッドに列番号1を渡してID列の値をint型で取り出し、getStringメソッドに列名「name」を渡してname列の値をString型で取り出してコンソールに表示しています。

```
12.             var sql = "select * from item";
13.             try (PreparedStatement ps = con.prepareStatement(sql)){
14.                 ResultSet rs = ps.executeQuery();
15.                 while(rs.next()) {
16.                     System.out.print(rs.getInt(1) + ":");
17.                     System.out.println(rs.getString("name"));
18.                 }
19.             }
```

なお、nextメソッドを一度も呼び出さなかった場合、カーソルは動かず、どの行も選択されていないことになります。行が選択されていない状態でgetXXXメソッドを呼び出すと、次のような例外がスローされます。

例 実行結果

```
Exception in thread "main" java.lang.RuntimeException: java.sql.
SQLException: 現行カーソル位置での無効な操作です。
    at Sample.main(Sample.java:21)
Caused by: java.sql.SQLException: 現行カーソル位置での無効な操作です。
… 以下省略 …
```

設問のコードにはnextメソッドが記述されていないため、実行時に例外がスローされます。よって、選択肢Fが正解です。

ResultSetのgetXXXメソッドで指定する列番号は、0からではなく1から始めなければいけないことに注意してください。

DBMSのリソースを解放するため、PreparedStatementインタフェースやResultSetインタフェースは使い終わったら明示的に閉じなければいけません。PreparedStatementのcloseメソッドを明示的に呼び出すか、try-with-resourcesにより自動的にcloseが実行されると、ResultSetオブジェクトも同時に閉じます。

12. B

➡ P299

前問までは、SQL文を実行するメソッドとしてexecuteUpdateメソッドとexecuteQueryメソッドを使って学習してきました。本設問では、挿入、更新、削除、検索のいずれのSQL文であっても実行できる**executeメソッド**について学びます。executeは、executeUpdateメソッドとexecuteQueryメソッドの両方の機能を持つメソッドだと考えるとよいでしょう。

これら3つのメソッドは、SQL文が挿入、更新、削除のいずれかであればexecuteUpdateメソッド、検索であればexecuteQueryメソッド、実行するSQLの種類が決まっていないときはexecuteメソッド、というように使い分けます。

次のコード例は、起動パラメータとしてSQL文を受け取り、実行するプログラムです。

例 起動パラメータとしてSQL文を受け取り、実行する

```
1. import java.sql.Connection;
2. import java.sql.DriverManager;
3. import java.sql.PreparedStatement;
4. import java.sql.ResultSet;
5. import java.sql.ResultSetMetaData;
6. import java.sql.SQLException;
7.
8. public class Sample {
9.     public static void main(String[] args) {
10.         try (Connection con = DriverManager.getConnection(
11.                 "jdbc:derby://localhost:1527/data/Sample")) {
12.
13.             try (PreparedStatement ps
14.                 = con.prepareStatement(args[0])){
15.
16.                 if (ps.execute() == false) {
17.                     System.out.println(ps.getUpdateCount());
18.                     return;
19.                 }
20.
21.                 ResultSet rs = ps.getResultSet();
22.                 ResultSetMetaData meta = rs.getMetaData();
23.                 int colSize = meta.getColumnCount();
24.                 while(rs.next()) {
25.                     for (int i = 1; i <= colSize; i++) {
26.                         System.out.println(rs.getString(i));
27.                     }
28.                 }
29.             }
30.         } catch (SQLException e) {
31.             throw new RuntimeException(e);
32.         }
33.     }
34. }
```


executeメソッドは、実行結果が**「検索した結果を保持するResultSet型オブジェクト」かどうかをboolean型**で戻します。そのため、executeメソッドの戻り値がfalseであった場合は、検索ではなく挿入・更新・削除のいずれかのためのSQLだとわかります。このコード例では、挿入・更新・削除のためのSQLによる結果の件数を取得する**getUpdateCountメソッド**の結果をコンソールに表示しています。以上のことから、選択肢**B**が正解となります。

たとえば、**挿入のためのSQL**を実行するには、次のようなコマンドを実行します。

```
> java -cp .;derbyclient.jar;derbyshared.jar Sample "insert
into values(3, 'test')"
1
```

もし、executeメソッドがtrueを戻した場合、実行したSQLは検索用だったことがわかります。そこで前述のコード例では、PreparedStatementのgetResultSetメソッドを使って実行結果のテーブルが入っているResultSetを取り出し、さらにResultSetから結果表のメタデータを取り出して、getColumnCountメソッドで何列あるかを調べ、全行、全列の結果をコンソールに表示しています。たとえば、**検索のためのSQL**を実行するには、次のようなコマンドを実行します。

```
> java -cp .;derbyclient.jar;derbyshared.jar Sample "select * from item"
1
hoge
2
hoge
3
test
```

このように、検索系と更新系のどちらのSQLであっても実行することができるのが、executeメソッドの特徴です。しかし、**SQLインジェクション**などの攻撃を受ける原因にもなるため、セキュリティの観点から実務での使用は推奨されません。学習用アプリケーションなどに限って利用しましょう。

試験対策

解答9～11で学んだexecuteUpdateメソッドやexecuteQueryメソッドと、executeメソッドの使い分けを覚えておきましょう。
・executeUpdateメソッド：SQLが挿入、更新、削除のいずれかの場合
・executeQueryメソッド：検索の場合
・executeメソッド：実行するSQLの種類が決まっていない場合

試験対策

executeメソッドは、実行結果が「検索した結果を保持するResultSet型オブジェクト」かどうかをboolean型で戻します。そのため、executeメソッドの戻り値がfalseであった場合は、検索ではなく挿入・更新・削除のいずれかのためのSQLだと判断できます。

13. C

➡ P300

StatementインタフェースのexecuteBatchメソッドに関する問題です。
SQL文を実行するメソッドには、これまでに学習してきたexecuteUpdateメソッド、executeQueryメソッド、そしてexecuteメソッドのほかにもう1つ、**executeBatchメソッド**があります。

このメソッドの特徴は、**一度に複数のSQL文が実行できる**ことです。これまでに学んできたメソッドは、いずれも一度に1つのSQL文しか実行できませんでした。そのため、複数のSQLを実行したいときは、次のコード例のようにメソッドを逐一呼び出す必要があったのです。

```
var sql = "insert into item values(?,?)";
try (PreparedStatement ps = con.prepareStatement(sql)){
    ps.setInt(1, 4);
    ps.setString(2, "hoge");
    ps.executeUpdate();

    ps.setInt(1, 5);
    ps.setString(2, "boo");
    ps.executeUpdate();
}
```

executeBatchメソッドを使うと、複数のSQL文を登録しておき、一度に実行することができます。実行したいSQL文を登録し、実行するには、次のコード例のように**addBatchメソッド**を使ってSQL文を登録し、executeBatchメソッドを呼び出します。

```
String[] names = {"apple", "banana", "orange"};
var sql = "insert into item values(?,?)";
try (PreparedStatement ps = con.prepareStatement(sql)){
    int i = 10;
    for (String name : names) {
        ps.setInt(1, i);
        ps.setString(2, name);
        ps.addBatch();
        i++;
    }
    int[] results = ps.executeBatch();
    System.out.println(Arrays.toString(results));
}
```

このコード例では、apple、banana、orangeという商品名を一度に挿入するSQL文を実行しています。なお、このexecuteBatchメソッドの戻り値は、各SQL文の更新件数の配列です。このコード例では新規データの挿入をしているため、更新件数「1」が3つ入った配列（[1, 1, 1]）が戻されます。
なお、executeBatchはStatementインタフェースで定義されているメソッドです。PreparedStatementはStatementインタフェースのサブインタフェースなので、executeBatchメソッドを引き継いでいます。

以上のことから、選択肢**C**が正解となります。選択肢BのexecuteUpdateメソッドは、1つのSQL文を実行するためのものです。選択肢AやDのようなメソッドは存在しません。よって、これらの選択肢は誤りです。

14. B

➡ P300

ストアド・プロシージャとCallableStatementインタフェースに関する問題です。
解答5で説明したとおり、Statementインタフェースを継承したサブインタフェースには、PreparedStatementのほかにCallableStatementもあります。このインタフェースは、SQL文を実行するためのものではなく、DBMSに登録されているストアド・プロシージャを実行するためのものです。
ストアド・プロシージャとは、SQL文や専用の言語を組み合わせて処理手順を記述し、データベースに対する連続した複数の処理を1つのプログラムにまとめ、DBMSに保存したものです。ストアド・プロシージャでは、条件分岐やループといった制御構文、変数、例外処理など、一般のコンピュータプ

ログラムと変わらない高度な機能を利用できます。

ストアド・プロシージャを記述するには、Oracle DatabaseのPL/SQLやMicrosoft SQL ServerのTransact-SQLなどのような各DBMS独自の言語を用いるのが一般的ですが、これらは便利な反面、完全な互換性があるわけではないため特定のDBMS製品に固定されてしまい、変更に弱くなるというデメリットもあります。なお、本書の解説で利用しているApache DerbyはJavaで作られていることもあり、Javaのプログラムをストアド・プロシージャとして登録することができます。

以下のコードが、Derbyに登録するストアド・プロシージャの例です。接続文字列には特別なものを使っていますが、それ以外はこれまでのJDBCを使ったプログラミングと変わらないことがわかります。

例 DBMSに登録するストアド・プロシージャ

```
1. package test;
2. import java.sql.Connection;
3. import java.sql.DriverManager;
4. import java.sql.PreparedStatement;
5.
6. public class ProcedureExample {
7.     public static void testProc(String name, int id)
                                        throws Exception {
8.         String connectionURL = "jdbc:default:connection";
9.         Connection con = DriverManager
                        .getConnection(connectionURL);
10.        String sql = "UPDATE item SET name = ? WHERE id = ?";
11.        PreparedStatement ps = con.prepareStatement(sql);
12.        ps.setString(1, name);
13.        ps.setInt(2, id);
14.        ps.executeUpdate();
15.    }
16. }
```

このクラスのtestProcメソッドはnameとidの2つの引数を受け取り、UPDATE文を作って実行しています。
このストアド・プロシージャをDerbyに登録するには、まず登録したいクラスをjarファイルとしてまとめます。jarファイルを作成するには、次のようなコマンドを実行します。作成したjarファイルをDerbyに登録するために、<Derby_HOME>に移動しておきます。


```
> jar cvf test.jar *.class
```

jarファイルが作成できたら、次のコマンドのようにDerbyのコマンドラインクライアントを起動し、jarファイルを登録し、ストアド・プロシージャを作成しましょう。

```
> cd <DERBY_HOME>
> java -jar lib/derbyrun.jar ij
ijバージョン10.15
ij>
ij> connect 'jdbc:derby://localhost:1527/data/Sample';

// jarファイルをAPPスキーマのtestという名前のオブジェクトとして登録
ij> call sqlj.install_jar('<DERBY_HOME>/test.jar','APP.test',0);

ij> call SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY('derby.database.
classpath','App.test');

// ProcedureExampleのtestProcメソッドをUPDATE_ITEMとして登録
ij> CREATE PROCEDURE UPDATE_ITEM(IN name VARCHAR(20), IN id
INTEGER) PARAMETER STYLE JAVA MODIFIES SQL DATA LANGUAGE JAVA
EXTERNAL NAME 'test.ProcedureExample.testProc';
```

これでJavaのクラスをストアド・プロシージャとして利用できるようになったので、次はストアド・プロシージャを呼び出すプログラムを作りましょう。次のコード例は、先ほど登録したストアド・プロシージャを呼び出し、実行するものです。

例 ストアド・プロシージャの呼び出し

```
1. import java.sql.CallableStatement;
2. import java.sql.Connection;
3. import java.sql.DriverManager;
4. import java.sql.SQLException;
5.
6. public class CallSample {
7.     public static void main(String[] args) {
8.         try (Connection con = DriverManager.getConnection(
9.                 "jdbc:derby://localhost:1527/data/Sample")) {
```

```
10.             String proc = "CALL UPDATE_ITEM(?, ?)";
11.             try (CallableStatement cs = con.prepareCall(proc)) {
12.                 cs.setString(1, "call procedure");
13.                 cs.setInt(2, 1);
14.                 cs.execute();
15.             }
16.
17.         } catch (SQLException e) {
18.             throw new RuntimeException(e);
19.         }
20.     }
21. }
```

ストアド・プロシージャを呼び出すには、java.sql.CallableStatementインタフェースを使います。CallableStatement型のオブジェクトは、**java.sql.Connectionインタフェース**の**prepareCallメソッド**に、呼び出すストアド・プロシージャ名を渡して取得します。CallableStatement型のオブジェクトの取得後は、PreparedStatementと同じようにパラメータをセットして、**executeメソッド**で実行します。
このコード例では、1つ目のパラメータに「call procedure」、2つ目のパラメータに「1」を渡しているため、登録したtestProcメソッドで作られるSQL文は、idが1の行のname列の値が「call procedure」に更新されるUPDATE文となります。
以上のことから、選択肢**B**が正解です。

7

汎用とコレクション

- オートボクシングとラッパークラス
- ジェネリクスの利用
- 共変性と非変性
- 非境界ワイルドカード
- 上限境界ワイルドカードと下限境界ワイルドカード
- List、Queue、Map、Setの特徴
- java.lang.Comparableインタフェース、comapreToメソッド
- java.util.Comparatorインタフェース、compareメソッド

1. プリミティブ型に対応するラッパークラスとして、誤っているものを選びなさい。(1つ選択)

A. Byte
B. Integer
C. Char
D. Boolean

➡ P335

2. 次のコードの「/* insert code here */」に宣言できるデータ型として、正しいものを選びなさい。(1つ選択)

```
1. public class Value<T> {
2.     T val;
3.     public Value(T val) {
4.         super();
5.         this.val = val;
6.     }
7.     public T getVal() {
8.         return val;
9.     }
10. }
```

```
1. public class Test {
2.     public static void main(String[] args) {
3.         Value v = new Value("Hello");
4.         /* insert code here */ a = v.getVal();
5.     }
6. }
```

A. Object
B. String
C. Value
D. T

➡ P337

3. ジェネリクスで型推論をするためのダイヤモンド演算子を使用する箇所として、誤っているものを選びなさい。(1つ選択)

A. 変数への代入
B. メソッドの戻り値
C. メソッド呼び出しの引数
D. クラス宣言

➡ P342

4. Aクラスを継承したBクラスがあるとき、ジェネリクスの型パラメータとして使用できるものを選びなさい。(1つ選択)

A. `Item<A> a = new Item<B>();`
B. `Item<A> a = new Item<A>();`
C. `Item<Object> a = new Item<A>();`
D. `Item<Object> a = new Item<B>();`

➡ P343

5. 型パラメータにおいて型変数が取り得る型をX型のサブタイプに制限する総称型のクラスFooの宣言として正しいものを選びなさい。(1つ選択)

A. `class Foo<T:X> {}`
B. `class Foo<T super X> {}`
C. `class Foo<T extends X> {}`
D. `class Foo<T limit X> {}`

➡ P345

6. 変数宣言時のジェネリクスの型パラメータで、どのような型パラメータを持つ参照であっても代入できるようにしたい。要件を満たす変数宣言を選びなさい。(1つ選択)

A. `List<*> list;`
B. `List<> list;`
C. `List<?> list;`
D. `List<#> list;`

➡ P347

7. 非境界ワイルドカードの特徴として、正しいものを選びなさい。（2つ選択）

A. メソッドの戻り値型はObject型になる
B. メソッドの戻り値型は、参照が持つ型パラメータとして指定した型になる
C. メソッドの引数にはnullリテラルしか渡せない
D. メソッドの引数にはObject型の参照しか渡せない

➡ P349

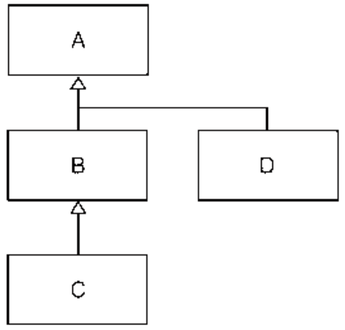
8. 次のプログラムをコンパイル、実行したときの結果として、正しいものを選びなさい。（1つ選択）

```
1. import java.util.ArrayList;
2. import java.util.List;
3.
4. public class Exercise {
5.     public static void main(String[] args) {
6.         List<? extends Number> list = new ArrayList<Integer>();
7.         list.add(10);
8.         list.add(20);
9.         list.add(30);
10.
11.        for (Number n : list) {
12.            System.out.println(n);
13.        }
14.    }
15. }
```

A. addメソッドの行でコンパイルエラーが発生する
B. for文でコンパイルエラーが発生する
C. 「10」「20」「30」と表示される
D. 実行時に例外がスローされる

➡ P351

9. 次のような関係のクラスがあった場合に使用できるコードとして、正しいものを選びなさい。（2つ選択）



A. `Test<? super B> = new Test<A>();`
B. `Test<? super B> = new Test<B>();`
C. `Test<? super B> = new Test<C>();`
D. `Test<? super B> = new Test<D>();`

➡ P354

10. インデックスによって順序付けられたデータ構造の機能を定義するコレクション・インタフェースとして、正しいものを選びなさい。（1つ選択）

A. java.util.Map
B. java.util.Collection
C. java.util.List
D. java.util.Set

➡ P358

11. java.util.Queueインタフェースが表現するデータ構造に関する説明として、正しいものを選びなさい。（1つ選択）

A. 参照されていない時間が最も長いデータを置換対象にする
B. 直近で利用されていたデータを置換対象にする
C. 最初に入れたデータが最後に取り出される
D. 最初に入れたデータが最初に取り出される

➡ P360

12. java.util.Dequeインタフェースに関する説明として、正しいものを選びなさい。(1つ選択)

A. 要素を管理するオブジェクトを数珠つなぎにして管理する
B. インデックスによる要素へのアクセスをサポートする
C. 両端から要素を挿入・削除できるデータ構造を定義する
D. 最初に格納した要素を最後に取り出すことのできる仕組みを提供する

➡ P362

13. java.util.Setインタフェースに関する説明として、正しいものを選びなさい。(1つ選択)

A. HashSetの要素は自然順序で並ぶ
B. TreeSetの要素の並び順は保証されない
C. 重複する要素を許容しない
D. キーと値のペアを要素として管理する

➡ P363

14. java.util.Mapインタフェースに関する説明として、正しいものを選びなさい。(1つ選択)

A. 実現クラスとしてVectorクラスやStackクラスを提供している
B. 一意なキーとひも付けられた値のペア要素を扱う
C. インデックスによって順序付けられた要素を扱う
D. 重複を許さない集合を扱う

➡ P365

15. Mapオブジェクトに格納されているキーと値を取り出して表示する場合に、「/* insert code here */」に入るコードとして、正しいものを選びなさい。(1つ選択)

```
import java.util.Map;

public class Test {
    public static void main(String[] args) {
        var a = Map.entry(1, "A");
        var b = Map.entry(2, "B");
        var c = Map.entry(3, "C");
        Map<Integer, String> map = Map.ofEntries(a, b, c);

        for (/* insert code here */) {
            System.out.println(entry.getKey()
                               + ":"
                               + entry.getValue());
        }
    }
}
```

A. `Map<Integer, String> entry : map.getEntry()`
B. `Set<Integer, String> entry : map.entrySet()`
C. `Map.Entry<Integer, String> entry : map.entrySet()`
D. `Set.Entry<Integer, String> entry : map.getEntry()`

➡ P366

16. java.lang.ComparableのcompareToメソッドに関する説明として、正しいものを選びなさい。(3つ選択)

A. 自分自身が比較対象よりも前に位置するように並べ替えるのであれば、正の整数を戻す
B. 自分自身が比較対象よりも前に位置するように並べ替えるのであれば、負の整数を戻す
C. 自分自身が比較対象よりも後ろに位置するように並べ替えるのであれば、正の整数を戻す
D. 自分自身が比較対象よりも後ろに位置するように並べ替えるのであれば、負の整数を戻す
E. 比較対象と同値であり、並べ替える必要がないのであれば0を戻す
F. 比較対象と同値であり、並べ替える必要がないのであれば負の整数を戻す

➡ P368

17. 次のプログラムを確認してください。

```
1. public class Product {
2.     private int id;
3.     public Product(int id) {
4.         super();
5.         this.id = id;
6.     }
7.     public int getId() {
8.         return id;
9.     }
10. }
```

このProductクラスのidフィールドで比較を行うComparatorインタフェースのメソッド実装として、正しいものを選びなさい。(1つ選択)

A. `Comparator<Product> c = p -> p.getId();`
B. `Comparator<Product> c = (p1, p2) -> p1.getId() - p2.getId();`
C. `Comparator<Product> c = p -> p.hashCode();`
D. `Comparator<Product> c = (p1, p2) -> p1.getId() > p2.getId();`

➡ P370

第7章　汎用とコレクション
解　答

1. C

➡ P328

Javaでは、プリミティブ型のデータを参照型のデータとして扱うことはできません。数値や文字、真偽値というデータは、どのメモリ領域を参照しているかという参照値とまったく互換性がないためです。プリミティブ型のデータを参照型として扱うためには、そのデータ型に応じた**ラッパークラス**を利用します。ラッパーとは、英語で「Wrapper」と記述し、「包み込むもの」という意味です。「プリミティブ型のデータを包み込むクラス」という意味で、ラッパークラスと呼んでいます。
プリミティブ型に対応したラッパークラスには、次の表に挙げる8種類のクラスが用意されています。

【プリミティブ型に対応したラッパークラス】

プリミティブ型のデータ	対応するラッパークラス
byte	java.lang.Byte
short	java.lang.Short
int	java.lang.Integer
long	java.lang.Long
double	java.lang.Double
float	java.lang.Float
char	java.lang.Character
boolean	java.lang.Boolean

Charというラッパークラスはありません。よって、選択肢**C**が正解です。

複数のプリミティブ型のデータを扱うには、配列を使わなければいけません。しかし、配列は上限が決まっていたり、使用している側のプログラムが配列の何番目を扱っているかを管理しなくてはいけないため、コレクションに比べて使いづらい面があります。そこで、JDK 1.4の頃までは、次のようにラッパークラスを使うことで、プリミティブ型のデータをコレクションで扱えるようにしていました。

```
List<Integer> list = new ArrayList<>();
list.add(new Integer(10));
```

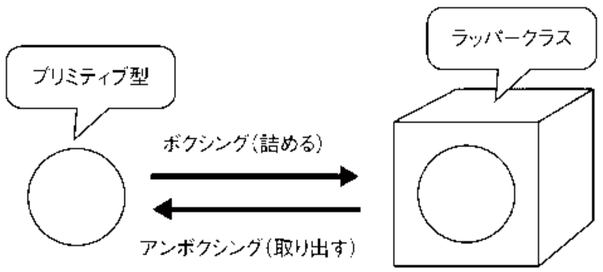
```
list.add(new Integer(20));
list.add(new Integer(30));
```

この方法では、コレクションから要素を取り出すときはラッパークラス型のままなので、そこからプリミティブ型のデータを取り出す手間がかかります。たとえば次のコード例では、java.lang.IntegerクラスのintValueメソッドでIntegerの値をint型として取り出しています。

```
for (int i = 0; i < list.size(); i++) {
    Integer obj = list.get(i);
    int num = obj.intValue();
    System.out.println(num);
}
```

このようにプリミティブ型のデータをラッパークラスで扱ったり、ラッパークラスからプリミティブ型のデータを取り出したりするコードをより簡単に記述できるようにしたのが、Java SE 5で導入された「オートボクシング」と呼ばれる、プリミティブ型と対応するラッパークラス型の自動変換機能です。**オートボクシング**とは「自動箱詰め」のことで、プリミティブ型のデータをラッパークラスという箱に詰めるようなイメージから、このように呼ばれています。また、プリミティブ型のデータを詰めた箱から取り出すことを**オートアンボクシング**といいます。

【オートボクシングとオートアンボクシングのイメージ】



オートボクシングを使えば、次のようにラッパークラス型の変数にプリミティブ型のデータを代入したり、反対にラッパークラス型への参照をプリミティブ型の変数に代入したりできます。

```
Integer a = 10; // オートボクシング
int b = a;      // オートアンボクシング
```


オートボクシングやオートアンボクシングは、コンパイラによってコードが自動的に変換されることで実現されています。そのため、プログラマーが型の互換性を意識する必要はありません。たとえば前述のコードは、コンパイルするとコンパイラによって、次のようなコードに置き換わるのです。

```
Integer a = Integer.valueOf(10);
int b = a.intValue();
```

このようにコンパイラが自動的にコードを書き換えることで、あたかもプリミティブ型とラッパークラス型の間に互換性があるかのように記述できるようにするのがオートボクシングです。

2. A

➡ P328

コレクションは、インスタンスの集合を扱うためのクラスの総称です。コレクションが扱えるクラスに制限はありません。これは、コレクションがどのようなインスタンスへの参照であっても、すべてjava.lang.Object型として扱うからです。このコレクションの性質を使うと、次のように異なる型のインスタンスの集合を作ることができます。

```
List list = new ArrayList();
list.add(10);
list.add("Java");
list.add(new Object());
```

しかし、このような型が混在したコレクションでは、要素を取り出したときにどの型でダウンキャストすればよいか迷うだけでなく、互換性のない型にダウンキャストしようとすると、実行時に例外が発生することになります。たとえば、前述のコードでは、コレクションにInteger型、String型、Object型の3つの型の参照が混在しています。このコレクションから要素を取り出し、次のようにダウンキャストをすると、実行時に**java.lang.ClassCastException**がスローされます。

```
for (int i = 0; i < list.size(); i++) {
    String str = (String) list.get(i);
    System.out.println(str);
}
```

※次ページに続く

例 実行結果

```
> java Sample
Exception in thread "main" java.lang.ClassCastException: class
java.lang.Integer cannot be cast to class java.lang.String
(java.lang.Integer and java.lang.String are in module java.
base of loader 'bootstrap') at Sample.main(Sample.java:15)
```

このように複数の型が混在している場合はコンパイルエラーを発生させるなど、コンパイラがその危険性を警告してくれればよいのですが、コンパイラはあくまでも文法上の間違いを指摘するだけであって、コレクション内にどのような型の参照が含まれているかまではチェックしてくれません。

そこで、J2SE 1.4までは、次のようにコレクションのラッパークラスを作り、特定の型だけを扱うaddメソッドやgetメソッドを定義することで、コレクションに異なる型が混在しないようなテクニックも使われていました。たとえば、次のコード例ではリストのラッパークラスを作り、Item型だけを扱うaddメソッドやgetメソッドを定義しています。こうすることで、このクラスに追加できるのはItem型とその互換クラスだけに限定することができるわけです。

例 Item型だけを扱うラッパークラス

```
 1. import java.util.ArrayList;
 2. import java.util.List;
 3.
 4. public class ItemList {
 5.     private List list = new ArrayList();
 6.     public void add(Item item) {
 7.         this.list.add(item);
 8.     }
 9.     public Item get(int index) {
10.         return (Item) this.list.get(index);
11.     }
12.     public int size() {
13.         return this.list.size();
14.     }
15. }
```

前述の「複数の型が混在している場合」の問題を解決するために導入されたのがジェネリクスです。**ジェネリクス**とは、そのクラスが扱うデータの型を実行時に決めることができる仕組みです。たとえば、次の貨幣を表すCurrencyクラスは、金額として整数だけを扱います。

例 整数だけを扱うCurrencyクラス

```
 1. public class Currency {
 2.     private Integer amount;
 3.     public Currency(Integer amount) {
 4.         super();
 5.         this.amount = amount;
 6.     }
 7.     public Integer getAmount() {
 8.         return amount;
 9.     }
10. }
```

しかし、貨幣によっては小数点以下の値も扱いたいときがあります。従来は別のクラスを用意する必要がありましたが、ジェネリクスを使えば次のように「型」の部分をあとから決められる**型パラメータ**として表現することができます。

例 型パラメータを使ったCurrencyクラス

```
 1. public class Currency<T> {
 2.     private T amount;
 3.     public Currency(T amount) {
 4.         super();
 5.         this.amount = amount;
 6.     }
 7.     public T getAmount() {
 8.         return amount;
 9.     }
10. }
```

コード中の「T」となっている部分が型パラメータです。この部分は、実行時にどの型を使うのかを指定することで、そのインスタンスが持つフィールドやメソッドの型パラメータが置き換わるのです。たとえば、次のように変数の型宣言をするときに**ダイヤモンド演算子**「<>」の中にDouble型を記述します。

※次ページに続く

例 Double型だけを扱うCurrencyのインスタンスを作る

```
1. public class Sample {
2.     public static void main(String[] args) {
3.         Currency<Double> currency = new Currency<Double>(7.25);
4.         System.out.println(currency.getAmount());
5.     }
6. }
```

このように型パラメータを与えると、作られたインスタンスは次のようなコードに変わっているとイメージするとよいでしょう。このインスタンスはDouble型だけを扱うCurrencyクラスのインスタンスということになり、フィールドの型もコンストラクタの引数の型も、そしてgetAmountメソッドの戻り値型もすべてDouble型に実行時に置き換わります。

例 Double型だけを扱うCurrencyクラス

```
1. public class Currency<Double> {
2.     private Double amount;
3.     public Currency(Double amount) {
4.         super();
5.         this.amount = amount;
6.     }
7.     public Double getAmount() {
8.         return amount;
9.     }
10. }
```

たとえば、次のように前述のコードの型パラメータだけを変えると、Integer型だけを扱うCurrencyクラスのインスタンスになります。

例 Integer型だけを扱うCurrencyのインスタンスを作る

```
1. public class Sample {
2.     public static void main(String[] args) {
3.         Currency<Integer> currency = new Currency<Integer>(100);
4.         System.out.println(currency.getAmount());
5.     }
6. }
```

インスタンスのイメージは、次のようなInteger型を扱うコードになります。

例 Integer型を扱うCurrencyクラス

```
1. public class Currency<Integer> {
2.     private Integer amount;
3.     public Currency(Integer amount) {
4.         super();
5.         this.amount = amount;
6.     }
7.     public Integer getAmount() {
8.         return amount;
9.     }
10. }
```

このように、実行時に具体的な型を指定することができるのがジェネリクスです。なお、ジェネリクスを使ったクラスのことを「**総称型**」と呼びます。Java SE 5以降のコレクションも総称型であり、前述のようなラッパークラスを今は作る必要はありません。たとえばjava.util.Listであれば、次のように型パラメータを与えることで、Itemクラスだけを扱えるリストのインスタンスを作ることができます。

```
List<Item> list = new ArrayList<Item>();
```

なお、型パラメータを受け取るクラスを扱うクラスに型パラメータを渡さなかった場合、**Object型の型パラメータ**が渡されたものとして解釈されます。そのため、次のリストは「Object型の集合を扱うインスタンス」という意味になります。

```
List list = new ArrayList();
```

以上のことから、選択肢**A**が正解です。

なお、ジェネリクスの実体は、すべてObject型で扱うクラスです。ジェネリクスは、利用するタイミングで、型パラメータで指定された型にダウンキャストしているだけです。そのため、前述のCurrencyクラスは、コンパイルすると次のように変換されます。

```
1. public class Sample {
2.     public static void main(String[] args) {
```

```
3.         Currency currency = new Currency(7.25);
4.         System.out.println((Double)currency.getAmount());
5.     }
6. }
```

ただし、慣れるまでは、コードの置き換え文字（Tとしていた部分）が型パラメータの型に置き換わったインスタンスが作られるとイメージしたほうがわかりやすいでしょう。

3. D

➡ P329

Java SE 5で導入されたジェネリクスは、変数の宣言時とコンストラクタ使用時の両方に型パラメータを指定しなければいけませんでした。Java SE 7では、変数宣言時の型パラメータを使って、コンストラクタ使用時の型パラメータを推論する「**型推論**」という機能が盛り込まれました。型推論は、**ダイヤモンド演算子**「**<>**」を記述するだけで実現できます。

```
List<String> list = new ArrayList<>();
```

ジェネリクスの型推論は、変数への代入など型が明確でなければ使えません。
型推論が使えるのは、主に次のような箇所です。

- **変数への代入**
- **メソッドの戻り値**
- **メソッド呼び出しの引数**

次のコード例が、型推論を使った例です。

例 型推論

```
1. import java.util.ArrayList;
2. import java.util.List;
3.
4. public class Sample {
5.     public static void main(String[] args) {
6.         // 変数への代入
7.         List<String> list = new ArrayList<>();
8.         // メソッド呼び出しの引数
9.         execute(new ArrayList<>());
10.     }
11.     private static List<String> test() {
12.         // メソッドの戻り値
13.         return new ArrayList<>();
14.     }
15.     private static void execute(List<String> list) {
16.         // do something
17.     }
18. }
```

以上のことから、選択肢Dが正解です。

4. B

➡ P329

Aクラスを継承したBクラスがあるとき、次のようにBのインスタンスへの参照をA型として扱うことができます。

```
A a = new B();
```

A型として扱っていたBのインスタンスへの参照を使って、B型にしかなかった差分のメソッドを使うには、A型として扱っていたBへの参照を、次のようにB型として扱うようにダウンキャストしなければいけません。

```
B b = (B) a;
```

しかし、厳密にはこの操作は安全ではありません。変数aで参照する先にあるインスタンスはスーパークラス型のAかもしれないし、Bかもしれないし、Aを継承したCやDというクラスかもしれないからです。このような危険性を回避するためにも、ここでは変性について学びます。
参照型をどの型として扱えるようにするかという性質のことを「**変性**」と呼びます。変性の種類には、次の3つがあります。

- サブクラス型の代入を許す「**共変性**（covariance）」
- サブクラス型の代入を許さない「**非変性**（invariance）」
- さらにスーパークラス型の代入を許す「**反変性**（contravariance）」

共変性とは、サブクラス型の参照をスーパークラス型の変数に代入できる性質です。ポリモーフィズムは、変数が共変性を持つことによって実現できて

いるのです。J2SE 1.4までは共変性だけをサポートしていたため、配列で次のような問題が起きました。

```
String[] strArray = {"A", "B"};
Object[] objectArray = strArray; // 代入できる
objectArray[0] = Integer.valueOf(1); // コンパイルはできる
```

このコードは、String型だけを扱う配列型変数strArrayを用意し、そこにString型だけを扱う配列のインスタンスを作り、その参照を代入しています。その後、Object型だけを扱う配列型変数objectArrayを宣言し、そこに変数strArray内の参照をコピーして代入しています。String型はObject型のサブクラスであるため共変性が働き、このコードは問題なくコンパイル、実行できます。
次の行では、配列の最初の要素に、Integer型のインスタンスへの参照を代入しようとしています。このときも共変性が働くため、Object型のサブクラスであるInteger型の参照は、この配列に代入できるとコンパイラは判断してしまいます。しかし、変数objectArrayの中身は、String型だけを扱う配列のインスタンスへの参照であるため、この配列にInteger型の参照を代入することはできません。そのため、このコードを実行すると、次のように実行時に例外がスローされることになります。

```
Exception in thread "main" java.lang.ArrayStoreException: java.
lang.Integer
    at Sample3.main(Sample.java:12)
```

このように実行時に例外がスローされるため、実行しなければ問題に気づかないかもしれません。こういった問題を解決するために、Java SE 5でジェネリクスが導入されたタイミングで、ジェネリクスについては変性を見直し非変とすることで、コンパイラが型の安全性をチェックできるようになりました。ただし、配列については過去の互換性を考慮して、共変のままとなっています。

以上のことから、次のようなジェネリクスを使ったコードは、コンパイルエラーとなります。

```
List<Object> list = new ArrayList<String>();
```

もし、このコードが許可されれば、次のようなコードが記述できることになり、実行時に例外がスローされることになるためです。


```
List<Object> list = new ArrayList<String>();
list.add(100); // ボクシングでIntegerのインスタンスへの参照となる
```

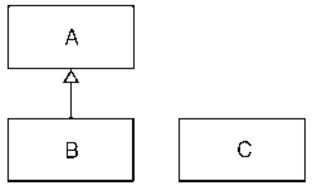
ポリモーフィズムが使える（共変である）ことがオブジェクト指向の特徴のように感じてしまいますが、ジェネリクスは非変であることを覚えておきましょう。
以上のことから、選択肢**B**が正解です。

5. C

➡ P329

ジェネリクスを使用した総称型を定義するとき、型パラメータの実際の型は使用する側が決めます。そのため、総称型側では、型パラメータとして渡されるインスタンスが持つ独自のメソッドを使うことはできません。
たとえば、次のような3つのクラスがあったとしましょう。AクラスとBクラスの間には継承関係がありますが、Cクラスとほかのクラスの間に継承関係はありません。

【Aクラス、Bクラス、Cクラスの関係】



例 Aクラス

```
1. public class A {
2.     public void hello() {
3.         System.out.println("A");
4.     }
5. }
```

例 Bクラス

```
1. public class B extends A {
2.     @Override
3.     public void hello() {
4.         System.out.println("B");
5.     }
6. }
```

例 Cクラス

```
1. public class C {
2.     public void hello() {
3.         System.out.println("C");
4.     }
5. }
```

そして、次のような総称型のTestクラスを定義し、仕様上、この型パラメータにはAクラスもしくはAクラスと互換性のあるサブクラスが渡されるものとします。TestクラスのexecuteメソッドはAクラスもしくは互換性のある型が渡されてくることを前提に、Aクラスもしくはサブクラスのhelloメソッドを呼び出したいとしましょう。

例 総称型のTestクラス

```
1. public class Test<T> {
2.     public void execute(T t) {
3.         t.hello();
4.     }
5. }
```

しかし、総称型クラスでは型パラメータTの型はまだ決まっていないため、型Tがhelloメソッドを持っていることを保証できず、このTestクラスはコンパイルエラーとなります。

そこで、制約付きの型パラメータを指定することで、このような問題を解決します。**制約付きの型パラメータ**とは、ジェネリクスを宣言するときに**extends**を使って、型パラメータとして使用できるクラスを制限する仕組みです。たとえば、次のように前述のTestクラスを修正すると、AクラスもしくはAクラスのサブクラスが型パラメータとして渡されてくるように制限します。こうすることにより、Aクラスが持っているhelloメソッドを型Tが持っていることを保証できるため、コンパイルエラーは起きません。

例 制約付きの型パラメータを指定したTestクラス

```
1. public class Test<T extends A> {
2.     public void execute(T t) {
3.         t.hello();
4.     }
5. }
```


Testクラスは型パラメータとして、Aもしくはそのサブクラス型しか受け取りません。そのため、次のように型パラメータとしてC型を渡すとコンパイルエラーとなります。

例 Sampleクラス

```
1. public class Sample {
2.     public static void main(String[] args) {
3.         Test<A> a = new Test<>();
4.         Test<B> b = new Test<>();
5.         // Test<C> c = new Test<>(); コンパイルエラー
6.
7.         a.execute(new A());
8.         a.execute(new B());
9.         b.execute(new B());
10.    }
11. }
```

以下のことから、選択肢**C**が正解となります。選択肢Bのsuperは、ワイルドカードを使ったときにしか使えません（解答9を参照）。間違えやすいので覚えておきましょう。選択肢AやDのような構文は存在しません。よって、誤りです。

ジェネリクスの宣言時に、<T extends A>のように制約付きの型パラメータを使うことで、型パラメータとして使用できるクラスをAクラスもしくはAクラスのサブクラスに制限することができ、互換性が保証されます。

6. C

➡ P329

解答4で、ジェネリクスは非変であることを学びました。そのため、ジェネリクスを使うときは、使う側と使われる側の両方で、どの型の型パラメータをやり取りするかを厳密に決めておかなければいけません。

たとえば、次のコード例のようにInteger型だけを扱うリストとString型だけを扱うリストの2つがあったとき、String型だけを扱うCollection型の引数を受け取るprintAllメソッドには、String型だけを扱うリストしか渡せません。8行目では、Integer型だけを扱うCollection型の引数をprintAllメソッドに渡しているため、コメントを外すとコンパイルエラーになります。

※次ページに続く

例 String型だけを扱うCollection型の引数を受け取るprintAllメソッド

```
1. import java.util.Collection;
2. import java.util.List;
3.
4. public class Sample {
5.     public static void main(String[] args) {
6.         List<Integer> a = List.of(1, 2, 3, 4, 5);
7.         List<String> b = List.of("A", "B", "C");
8.         // printAll(a); コンパイルエラー
9.         printAll(b);
10.     }
11.
12.     private static void printAll(Collection<String> collection) {
13.         collection.stream()
14.             .forEach(System.out::println);
15.     }
16. }
```

だからといって、次のようにCollection型の型パラメータにObject型を指定すると、ジェネリクスは非変であることから、型パラメータにObject型を持つリストしか渡すことができなくなってしまいます。

例 Object型だけを扱うCollection型の引数を受け取るprintAllメソッド

```
1. import java.util.Collection;
2. import java.util.List;
3.
4. public class Sample {
5.     public static void main(String[] args) {
6.         List<Integer> a = List.of(1, 2, 3, 4, 5);
7.         List<String> b = List.of("A", "B", "C");
8.         // printAll(a); コンパイルエラー
9.         // printAll(b); コンパイルエラー
10.     }
11.
12.     private static void printAll(Collection<Object> collection) {
13.         collection.stream()
14.             .forEach(System.out::println);
15.     }
16. }
```


コード例のprintAllメソッドは、コレクションからStreamオブジェクトを取り出し、forEachメソッドでコンソールに全要素を表示しているだけです。本来であれば、ジェネリクスで制限する必要性はありません。そこで、このような汎用的に使いたいコードでは、ジェネリクスの型パラメータは、実行時まで型が不明であることを示す**ワイルドカード**「**?**」を使います。以上のことから、選択肢**C**が正解です。

前述のコードをワイルドカードを使って修正すると、2種類のリストのどちらも扱うことができるようになります。

例 非境界ワイルドカードを使ったprintAllメソッド

```
1. import java.util.Collection;
2. import java.util.List;
3.
4. public class Sample {
5.     public static void main(String[] args) {
6.         List<Integer> a = List.of(1, 2, 3, 4, 5);
7.         List<String> b = List.of("A", "B", "C");
8.
9.         printAll(a);
10.        printAll(b);
11.    }
12.
13.    private static void printAll(Collection<?> collection) {
14.        collection.stream()
15.            .forEach(System.out::println);
16.    }
17. }
```

なお、このようなワイルドカードを「**非境界ワイルドカード**」と呼びます。ジェネリクスに制限付き型パラメータがあったように、ワイルドカードにも制限を設けることができます。次からは、上限境界ワイルドカードと下限境界ワイルドカードについて解説していきます。

7. A、C

➡ P330

前問で学習した非境界ワイルドカードには、次のような特徴があります。これらの特徴から、非境界ワイルドカードはあまり使い勝手のよい仕組みではないといえます。

※次ページに続く

1. メソッドの戻り値型はObject型になる
2. メソッドの引数にはnullリテラルしか渡せない

たとえば、次のようなコードがあったとしましょう。このTestクラスは、ジェネリクスの型パラメータで受け取った型のフィールドを持ち、コンストラクタやアクセサメソッドも型パラメータによって、扱うべき型が決定されます。

例 Testクラス

```
1. public class Test<T> {
2.     T value;
3.
4.     public Test(T value) {
5.         super();
6.         this.value = value;
7.     }
8.     public T getValue() {
9.         return value;
10.    }
11.    public void setValue(T value) {
12.        this.value = value;
13.    }
14. }
```

このクラスを使って非境界ワイルドカードの特徴を確認しているのが、次のSampleクラスです。

例 Sampleクラス

```
1. public class Sample {
2.     public static void main(String[] args) {
3.         Test<?> test = new Test<>("Hello");
4.         String msg = (String) test.getValue();
5.         // test.setValue("sample");     コンパイルエラー
6.         // test.setValue(new Object());     コンパイルエラー
7.         test.setValue(null);
8.     }
9. }
```


このコード例では、getValueメソッドを使ってフィールドの値を取得していますが、非境界ワイルドカードを使っているため、コンパイル時にはフィールドの型は「**不定である**」とコンパイラに判断されます。
非境界ワイルドカードを使うと、コンパイラは、どのような型のインスタンスへの参照が入っているかを特定できないため、**Object型で戻り値を戻す**ものと判断します。すべてのクラスはObject型のサブクラスであるため、ClassCastExceptionが発生することはないからです。
そのため、もしObject型以外の型で戻り値の参照を扱いたい場合には、コード例のようにキャスト式を記述してダウンキャストをしなければいけません。もちろん、型が異なってもコンパイラは検出できず、実行時に例外が発生する可能性はあるため、ジェネリクスが目指した安全なコードではありません。

次に、コード例では、setValueメソッドを使ってString型とObject型それぞれの参照、そしてnullリテラルの3種類の値をセットしていますが、String型とObject型についてはコンパイルエラーとなります。これは、非境界ワイルドカードによってフィールドの型が「不定である」とコンパイラによって判断されるためです。不定であるということは、Integer型やList型かもしれず、String型とObject型はこれらの型と互換性がありません。そのため、このように型が不定なフィールドに値を代入することはできません。
唯一代入できる値は**nullリテラル**のみです。nullリテラルであれば、たとえどの型なのかがわからなくても、ClassCastExceptionは発生し得ないからです。nullリテラルは、null型という特殊な型の唯一の値であり、どのような参照型の変数にでも代入できることがJavaの仕様で定められています。
以上のことから、選択肢**A**と**C**が正解となります。

8. A

➡ P330

前問で非境界ワイルドカードの特徴を学びました。ここでは、ワイルドカードに制限を付けたジェネリクスの使い方について学びます。

まず、次のような継承関係にあるA、B、Cという3つのクラスを用意します。

例 Aクラス

```
1. public class A {
2.     public void hello() {
3.         System.out.println("A");
4.     }
5. }
```

※次ページに続く

例 Bクラス

```
1. public class B extends A {
2.     @Override
3.     public void hello() {
4.         System.out.println("B");
5.     }
6. }
```

例 Cクラス

```
1. public class C extends B {
2.     @Override
3.     public void hello() {
4.         System.out.println("C");
5.     }
6. }
```

また、ジェネリクスを扱うクラスとして、次のTestクラスを用意します。このクラスは、ジェネリクスの型パラメータで受け取った型のフィールドを持ち、コンストラクタやアクセサメソッドも型パラメータによって、扱うべき型が決定されます。

例 Testクラス

```
1. public class Test<T> {
2.     T value;
3.
4.     public Test(T value) {
5.         super();
6.         this.value = value;
7.     }
8.     public T getValue() {
9.         return value;
10.    }
11.    public void setValue(T value) {
12.        this.value = value;
13.    }
14. }
```

非境界ワイルドカードを使うと、TestクラスのgetValueメソッドの戻り値は

Object型でしか扱えません。そこで、上限境界ワイルドカードという制約付きのワイルドカードを使うことで、このような問題を解決します。**上限境界ワイルドカード**とは、ジェネリクスを宣言するときに**extends**を使って、どのクラスが型パラメータとして使用できるかを制限する仕組みです。たとえば次のように変数を宣言すると、「Bもしくはそのサブクラス型の型パラメータを受け取るTest型変数のtest」という意味になります。

```
Test<? extends B> test;
```

この変数には、次のコード例のようにBかCの型パラメータを持ったTestクラスのインスタンスへの参照を代入することができます。一方で、Bクラスのスーパークラスである**A**の型パラメータを持ったものはコンパイルエラーとなります。

例 上限境界ワイルドカードを使用したSampleクラス

```
1. public class Sample {
2.     public static void main(String[] args) {
3.         Test<? extends B> test = new Test<B>(new B());
4.         B b = test.getValue();
5.         b.hello();
6.
7.         Test<? extends B> test2 = new Test<C>(new C());
8.         B b2 = test2.getValue();
9.         b2.hello();
10.
11.        // コンパイルエラー
12.        // Test<? extends B> test3 = new Test<A>(new A());
13.        // test.setValue(new B()); コンパイルエラー
14.    }
15. }
```

このようにワイルドカードに制限を設けると、Bクラスもしくはそのサブクラス型が型パラメータとして渡されてくることが保証できるため、getValueメソッドの戻り値型TをB型にしても問題が起きません。このように上限境界ワイルドカードを使うことで、非境界ワイルドカードの特徴である「メソッドの戻り値型はObject型になる」を「任意の型にする」ことができるようになります。

一方で、もう1つの特徴である「メソッドの引数にはnullリテラルしか渡せな

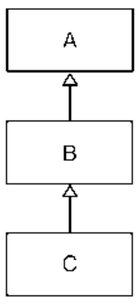
い」は、上限境界ワイルドカードでは改善されません。なぜなら、変数testやtest2で宣言している型パラメータは、<? extends B>であって<B>や<C>ではないからです。<? extends B>と型パラメータで指定されたTestのフィールドは、Bのインスタンスへの参照を持っているかもしれないし、もしくはCのインスタンスへの参照を持っているかもしれません。つまり、フィールドの型はまだ「不定」なままなのです。
型が不定なフィールドに値を代入することはできません。唯一代入できる値は**nullリテラルのみ**です。nullリテラルであれば、たとえどの型なのかがわからなくても、ClassCastExceptionは発生し得ないからです。引数に任意の型を渡すには、次の解答9で解説する、もう1つの制約付きワイルドカードである「下限境界ワイルドカード」を使います。

以上のことから、選択肢Aが正解です。上限境界ワイルドカードを使っているときは、メソッドの引数にnull以外を渡すことはできません。一方で、戻り値型はワイルドカードの上限（設問ではNumber型）になります。よって、選択肢Bのようにfor文でコンパイルエラーが発生することはありません。

9. A、B ➡P331

前問では、戻り値型に任意の型を使用できる「上限境界ワイルドカード」の使い方を学びました。ここでは、nullリテラルしか渡せなかった引数の制約を解除する「下限境界ワイルドカード」について学びましょう。
上限境界ワイルドカードは、このワイルドカードの上限はどの型で、型パラメータは必ずその型かそのサブクラス型であることを定義するものでした。一方、ここで学ぶ**下限境界ワイルドカード**は、このワイルドカードで扱える最も具体的な型は何かを定義するものです。
次のような継承関係にある3つのクラスについて見ていきます。

【継承関係にあるクラスA、B、C】



Aクラスにはhelloメソッドがあり、コンソールにAを表示します。サブクラスのBとCでは、いずれもこのメソッドをオーバーライドして、BとCをそれぞれコンソールに表示するようにしています。

例 Aクラス

```
1. public class A {
2.     public void hello() {
3.         System.out.println("A");
4.     }
5. }
```

例 Bクラス

```
1. public class B extends A {
2.     @Override
3.     public void hello() {
4.         System.out.println("B");
5.     }
6. }
```

例 Cクラス

```
1. public class C extends B {
2.     @Override
3.     public void hello() {
4.         System.out.println("C");
5.     }
6. }
```

これらのクラスを型パラメータとして受け取るTestクラスを用意します。

例 Testクラス

```
1. public class Test<T> {
2.     T value;
3.
4.     public Test(T value) {
5.         super();
6.         this.value = value;
7.     }
8.     public T getValue() {
9.         return value;
10.    }
11.    public void setValue(T value) {
```

```
12.         this.value = value;
13.     }
14. }
```

下限境界ワイルドカードは、型パラメータとして受け取れる最も具体的な型を制限するもので、指定された型、あるいはよりスーパークラス型（もしくはスーパーインタフェース型）を型パラメータとして指定します。たとえば、次のようにTest型の変数宣言をするときに**<? super B>**という型パラメータを与えると、非変であるはずのジェネリクスの制限を緩和し、Bクラスもしくはより上位のスーパークラス（もしくはインタフェース）であれば型パラメータとして使えるようになります。

```
Test<? super B> a = new Test<A>(new A());
```

このコードでは、Aクラスを型パラメータに持ったTestクラスのインスタンスを生成して、その参照を代入しています。変数aのジェネリクスは、Bもしくはより上位の型であれば型パラメータとして受け取れるため、コンパイルエラーは発生しません。
一方、次のように下限境界ワイルドカードを使った<? super B>という型パラメータを持つ変数cに、Bクラスのサブクラスである**C**を型パラメータに持つTestのインスタンスへの参照は代入できません。

```
Test<? super B> c = new Test<C>(new C());
```

ジェネリクスは非変であり、たとえBとCの間に継承関係があっても互換性はなく、かつワイルドカードはBよりも上位のものでしか有効でないためです。よって、このコードはコンパイルエラーとなります。

以上のことから、選択肢**A**と**B**が正解となります。下限境界ワイルドカードで利用できるのは、下限とされたクラスよりも上位の型です。そのため、選択肢CのようにBクラスのサブクラスであるCには互換性がありません。また、選択肢DのDクラスはAクラスのサブクラスですが、Bクラスとは無関係です。そのため、この型パラメータに互換性はありません。よって、選択肢CとDは誤りです。

上限境界ワイルドカードを使った場合、戻り値には任意の型を使うことができますが、引数にはnullリテラルしか渡せません。この下限境界ワイルドカードでは、引数の型に**任意の型**を使うことができます。

次のコード例のように、<? super B>という下限境界ワイルドカードを指定したジェネリクスを使った場合、その変数で呼び出されるメソッドの引数の型はB型になります。<? super B>は、Bもしくはより上位の型という意味であるため、Testクラスのフィールドvalueの型Tは、Bもしくはより上位の型であることが保証されています。そのため、Bもしくはそのサブクラス型であれば引数として受け取って、フィールドの値として扱うことができます。次のコード例のように、setValueメソッドの引数の型はB型となり、BやCのインスタンスへの参照は渡せますが、Aのインスタンスへの参照は渡せません。

例 Sampleクラス

```
1. public class Sample {
2.     public static void main(String[] args) {
3.         Test<? super B> a = new Test<A>(new A());
4.         Test<? super B> b = new Test<B>(new B());
5.
6.         // 型パラメータにはBもしくはより上位しか使えないのでコンパイルエラー
7.         // Test<? super B> c = new Test<C>(new C());
8.
9.         // AはBのサブクラスではないのでコンパイルエラー
10.        // a.setValue(new A());
11.        a.setValue(new B());
12.        b.setValue(new C());
13.
14.        // 取り出すときには何型が入っているかが不定なためObject型になる
15.        Object objA = a.getValue();
16.        Object objB = b.getValue();
17.
18.        // 任意の型で扱う場合にはキャスト式が必要（ただし安全でない）
19.        A aObj = (A) a.getValue();
20.        B bObj = (B) b.getValue();
21.    }
22. }
```

このように、下限境界ワイルドカードを使えば引数の型に任意の型を使うことができますが、一方で戻り値の型はObject型に限定されます。取り出すときにフィールドvalueにどの型の参照が入っているかが不定だからです。
下限境界ワイルドカードは、下限、つまり最も具体的な型を指定して、それ以上の型であれば扱えるようにするというものです。このコード例であれば、Aのインスタンスへの参照が入っていることもあれば、Bのインスタンスへの参照が入っていることもあります。もしかしたら、Objectのインスタンスへ

の参照かもしれません。そのため、戻り値には、最も安全なすべてのクラスのスーパークラスであるObject型にしてしまえばClassCastExceptionが発生することはありません。
もちろん、キャスト式を記述してダウンキャストすることは可能ですが、このようなコードはコンパイラによって安全が保証されないので、推奨される方法ではありません。

前問と本設問では、上限境界ワイルドカードと下限境界ワイルドカードを学びました。境界ワイルドカード型に関して、<? extends E>と<? super E>のどちらを使うかを決めるための原則があります。
メソッド内の目的が、戻り値を戻すことを目的とした「Producer（提供者）」であればextendsを、一方、引数を受け取って利用することを目的とした「Consumer（消費者）」であればsuperを使います。この原則はProducer-Extends and Consumer-Superを略して「PECS」とも呼ばれます。
戻り値の型はextends、引数の型はsuperと使い分ければよいのですが、残念ながら両方を兼ね備えたワイルドカードは存在しません。戻り値型と引数の型の両方を安全に使うには、ワイルドカードなしのジェネリクスで非変の状態で利用するしかないのです。

試験対策

上限境界ワイルドカードで「BもしくはBのサブクラス型」を表すのは<? extends B>です。一方、下限境界ワイルドカードで「BもしくはBのスーパークラス型」を表すのは<? super B>です。

10. C

➡ P331

インデックスによって順序付けられるデータ構造の機能を定義したものが、**java.util.Listインタフェース**です。Listでは、要素に番号を振って、順番を管理します。よって、選択肢**C**が正解です。
選択肢Aのjava.util.Mapは、キーとバリューの組み合わせで要素を管理するコレクションです。選択肢BのCollectionインタフェースは、さまざまなコレクションのスーパーインタフェースであって、特定のデータ構造の機能を表すためのものではありません。選択肢Dのjava.util.Setは、重複のない要素の集合を管理するコレクションで、順番については実現クラスに依存します。よって、これらの選択肢は誤りです。

Listインタフェースの実現クラスには、主に次のようなクラスがあります。

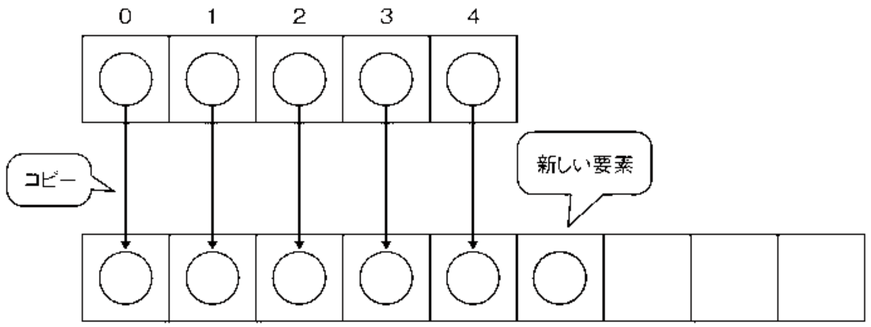
・java.util.ArrayList
・java.util.LinkedList

・java.util.Vector

ArrayListクラスは「動的配列」とも呼ばれ、内部に配列を使うことでリスト構造を実現しています。プログラマーがあらかじめ要素数を決めておかなければならない「配列」とは異なり、必要に応じて要素を自動的に増やすことができるという特徴があります。また要素は、追加した順に並びます。

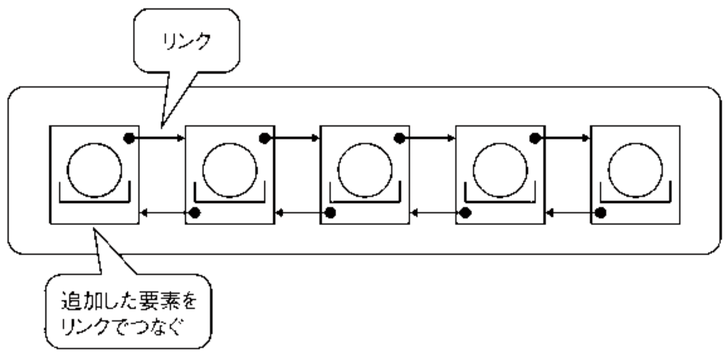
ArrayListを配列の要素数の範囲内で扱う場合の処理は高速ですが、配列の要素数を超える場合は、より大きな要素数を扱える配列を新たに作り、そこに元の要素をコピーします。このため、扱う要素数が増えるにつれてパフォーマンスが落ちるという特徴があります。

【ArrayListのイメージ】



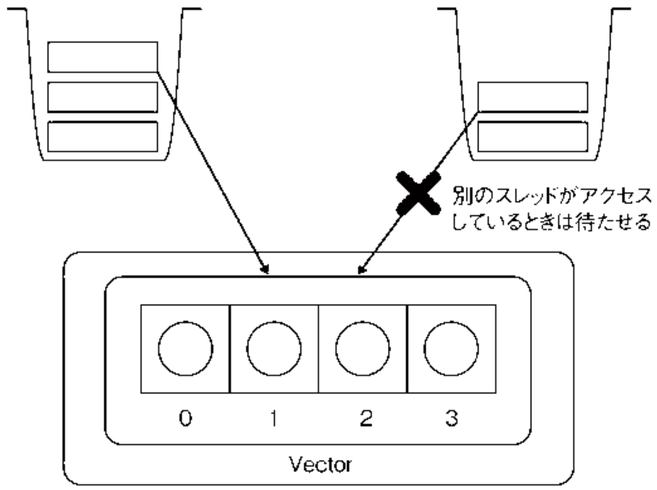
LinkedListクラスは、要素を管理するオブジェクトを数珠つなぎにした「リンク」を持つことでリスト構造を実現します。要素へは先頭もしくは末尾からリンクをたどりながらアクセスするため、ArrayListクラスに比べるとパフォーマンスは落ちます。一方で、ArrayListのように新たな配列を作り、元の配列から要素をコピーするような処理を行わずに要素を追加することができるため、要素の追加処理ではArrayListクラスよりも優れたパフォーマンスを発揮します。

【LinkedListのイメージ】



ArrayListクラスとLinkedListクラスの2つは、スレッドセーフではありません。そのため、これらのクラスをマルチスレッドで扱うと、思わぬ結果を引き起こす可能性があります。マルチスレッド環境下で順序付けられたコレクションを扱うのであれば、Vectorクラスを使いましょう。**Vector**はマルチスレッド環境下でも安全に使えるスレッドセーフなクラスですが、その分、ArrayListクラスやLinkedListクラスに比べるとパフォーマンスは落ちるという特徴があります。

【Vectorのイメージ】



試験対策

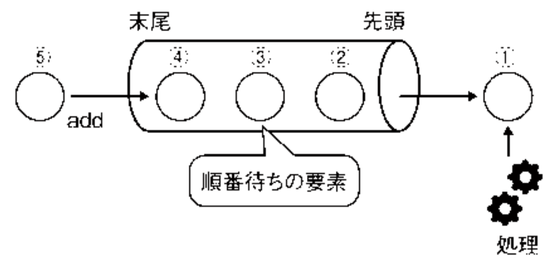
試験対策としては、Listインタフェースの実現クラスの特徴を以下のように覚えるとよいでしょう。
・ArrayListは、読み込みは速いが、書き込みは遅い
・LinkedListは、書き込みは速いが、読み込みは遅い
・Vectorは、マルチスレッドに対応しているが、パフォーマンスは悪い

11. D ➡ P331

java.util.Queueは、**FIFO**（First-In First-Out：先入れ先出し）のデータ構造の機能を定義したインタフェースです。FIFOは、最初に格納した要素を最初に取り出すことのできる仕組みを提供するデータ構造で、次ページの図のような筒をイメージしてもらうとわかりやすいと思います。

このようなデータ構造は、何らかの処理に対する順番待ちをしているイメージから「**待ち行列**」とも呼ばれています。以上のことから選択肢**D**が正解です。

【Queue（キュー）のイメージ】



次のコードが、Queueの使用例です。**addメソッド**を使って、コレクションの末尾から5つの文字列を追加し、それを順番に取り出しながらコンソールに表示しています。

例 Queueの使用例

```
1. import java.util.Queue;
2. import java.util.ArrayDeque;
3.
4. public class Sample {
5.     public static void main(String[] args) {
6.         Queue<String> queue = new ArrayDeque<>(5);
7.         queue.add("A");
8.         queue.add("B");
9.         queue.add("C");
10.        queue.add("D");
11.        queue.add("E");
12.
13.        queue.stream().forEach(System.out::println);
14.    }
15. }
```

このコードを実行すれば、次のように文字列が入れた順で表示されます。

例 実行結果

```
> java Sample
A
B
C
D
E
```

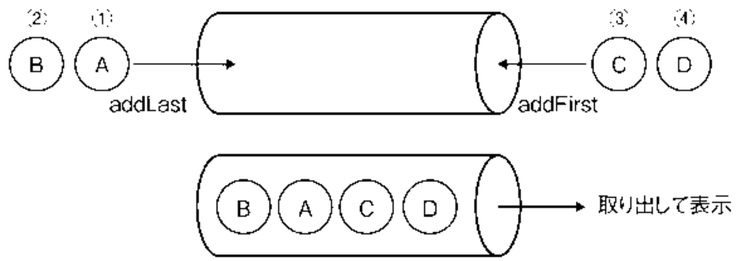
12. C

➡ P332

解答11で学習したQueueが末尾から先頭に向けて順番に入れていくコレクションだったの対し、**java.util.Deque**は末尾と先頭の両方から入れて、両方から取り出せるコレクションです。「Deque」という名前は、「double ended queue」（**両端キュー**）の省略形であり、通常は「デック」と発音されます。DequeもQueueと同様に、筒の中を出入りするイメージです。

次のコードが、Dequeを使用した例です。このコード例では**addLastメソッド**を使い、末尾からA、Bの順でキューに入れ、その後、**addFirstメソッド**を使って先頭からC、Dの順でキューに追加しています。イメージにすると、次のような図になります。

【Dequeのイメージ】



例 Dequeの使用例

```
 1. import java.util.ArrayDeque;
 2. import java.util.Deque;
 3.
 4. public class Sample {
 5.     public static void main(String[] args) {
 6.
 7.         Deque<String> deque = new ArrayDeque<>();
 8.         deque.addLast("A");
 9.         deque.addLast("B");
10.         deque.addFirst("C");
11.         deque.addFirst("D");
12.
13.         deque.stream().forEach(System.out::println);
14.     }
15. }
```

取り出すときは、先頭から順番にストリームを使っています。そのため、このコードを実行すると、次のような結果がコンソールに表示されます。

例 実行結果

```
> java Sample
D
C
A
B
```

Dequeインタフェースは、一方通行だったQueueを拡張し、先頭と末尾の両方から要素を挿入・削除できるデータ構造を定義するものです。よって、選択肢**C**が正解です。
選択肢Aのように要素を数珠つなぎにして管理するのは、Listインタフェースを実現したLinkedListクラスの特徴です。また、選択肢Bのインデックスで管理するのはListインタフェースの特徴です。選択肢Dは、Stackクラスの特徴です。よって、これらの選択肢は誤りです。

13. C

➡ P332

java.util.Setは、重複する要素を持たないことを保証するコレクションの機能を定義したインタフェースです。「重複する要素」とは、equalsメソッドによって等しいと判断される要素のことです。

このインタフェースのもう1つの特徴は、順序を規定しないことです。ListやQueueのように、入れた順序のとおりには要素が並びません。どのように並ぶかは、それぞれの実現クラスによって異なります。「Setを使ったコレクションでは、Listのように入れた順番には並ばない」ということを覚えておきましょう。

たとえば、Setインタフェースの実現クラスである**java.util.HashSetクラス**は、要素の並び順を保証しません。別の実現クラスである**java.util.TreeSetクラス**は、自然順序で並べ替えるか、引数で渡されたComparatorのアルゴリズムに従います。HashSetは並べ替えしない分、高速に処理できます。一方、TreeSetは並べ替え処理に時間がかかるため、HashSetに比べるとパフォーマンスが劣ります。もし、並べ替えの必要がなければ、HashSetを使いましょう。

次のコードが、Setを利用した例です。addメソッドを使ってB、D、A、C、Aの順に文字列を追加していますが、実現クラスとしてTreeSetクラスを利用しているため、要素は自然順序（ABCD）で並ぶことになります。また、Setは重複を許さないコレクションであるため、2回目の文字列Aの追加は実行されません。

例 Setの使用例

```
 1. import java.util.Set;
 2. import java.util.TreeSet;
 3.
 4. public class Sample {
 5.     public static void main(String[] args) {
 6.
 7.         Set<String> set = new TreeSet<>();
 8.         set.add("B");
 9.         set.add("D");
10.         set.add("A");
11.         set.add("C");
12.         set.add("A");
13.
14.         set.stream().forEach(System.out::println);
15.     }
16. }
```

このコードを実行すると、次のように自然順序で並べ替えられており、かつ重複するAは存在しないことがわかります。

例 実行結果

```
> java Sample
A
B
C
D
```

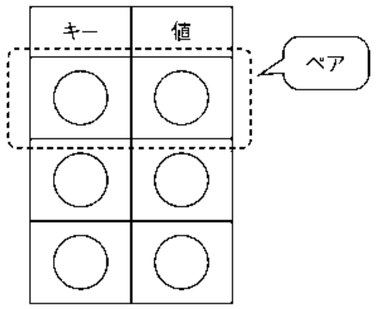
以上のことから、選択肢Cが正解です。HashSetは、要素の並べ替えをしません。並べ替えをするにはTreeSetを使います。よって、選択肢AとBは誤りです。また、キーとひも付けられた値のペアによってデータを管理する機能を定義したインタフェースはMapです。よって、選択肢Dも誤りです。

14. B

➡ P332

java.util.Mapインタフェースは、一意なキーと、そのキーにひも付けられた値のペアを要素として扱うデータ構造の機能を定義したインタフェースです。Mapは、次のような表をイメージするとよいでしょう。

【Mapのイメージ】



なお、このような**キーと値**のペアでデータを管理する方法のことを「Key Value形式」や「ディクショナリ形式」と呼びます。
次のコードが、Mapインタフェースの使用例です。Mapのもう1つの特徴は、キーと値で別々のジェネリクスの型パラメータを使うことです。Mapの**putメソッド**を使い、キーと値の2つの引数を渡してペアを作ります。
Mapから値を取り出すには、**getメソッド**の引数にキーを渡します。たとえばこのコード例であれば、キーの1つである「ONE」という文字列を引数に渡すと、1という値を持つIntegerのインスタンスへの参照が戻されます。

例 Mapの使用例

```
 1. import java.util.HashMap;
 2. import java.util.Map;
 3.
 4. public class Sample {
 5.     public static void main(String[] args) {
 6.
 7.         Map<String, Integer> map = new HashMap<>();
 8.         map.put("ONE", 1);
 9.         map.put("TWO", 2);
10.         map.put("THREE", 3);
11.
12.         map.keySet().stream().forEach(key -> {
13.             System.out.println(map.get(key));
14.         });
```

```
15.     }
16. }
```

このコード例では**keySetメソッド**を使い、キーだけの集合（java.util.Set）を取り出し、ストリームを使って繰り返し処理をしています。そのためこのコードを実行すると、次のような結果がコンソールに表示されます。

例 実行結果

```
> java Sample
1
2
3
```

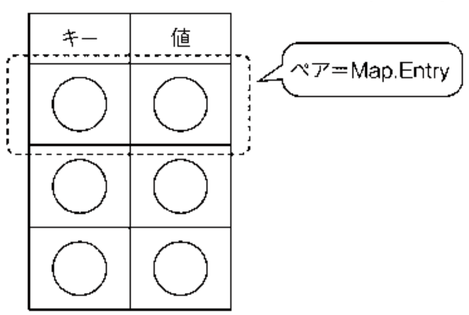
以上のことから、選択肢**B**が正解です。選択肢AはListの実現クラス、CはListの、DはSetの説明です。

15. C

➡ P333

解答14でjava.util.Mapインタフェースの特徴を学びました。ここでは、マップ内のペアを扱うためのMap.Entryインタフェースについて学びましょう。
Map.Entryは、Mapインタフェース内に定義されたインナーインタフェースです。インナーインタフェースは珍しいですが、Mapのためだけに利用することを想定し、別のインタフェースではなくインナーインタフェースとしたのでしょう。
前問では、Mapは、次のような表をイメージするとよいと述べました。このキーと値のペアを管理するのが、Map.Entryインタフェースです。

【キーと値のペアを管理するMap.Entry】



次のコードが、Map.Entryを使用した例です。Mapの**entrySetメソッド**でMap.Entryの集合を取り出し、そこからストリームで繰り返し処理をしています。forEachメソッドに渡すラムダ式の引数entryのデータ型は省略可能ですが、Map.Entryを使用していることを明示するために意図的に記述しています。

例 Map.Entryの使用例

```
1. import java.util.HashMap;
2. import java.util.Map;
3.
4. public class Sample {
5.     public static void main(String[] args) {
6.         Map<String, Integer> map = new HashMap<>();
7.         map.put("ONE", 1);
8.         map.put("TWO", 2);
9.         map.put("THREE", 3);
10.
11.         map.entrySet()
12.             .stream()
13.             .forEach((Map.Entry<String, Integer> entry) -> {
14.                 String key = entry.getKey();
15.                 Integer val = entry.getValue();
16.                 System.out.println(key + ":" + val);
17.             });
18.     }
19. }
```

ストリームから1つのMap.Entry型のインスタンスへの参照が取り出せれば、ペアのキーを取り出す**getKeyメソッド**や、値を取り出す**getValueメソッド**を使うことができます。そのため、このコードを実行すると、次のような結果がコンソールに表示されます。

例 実行結果

```
> java Sample
ONE:1
TWO:2
THREE:3
```

MapからMap.Entryを取得するには、**entrySetメソッド**を使います。よって、選択肢**C**が正解です。
なお、Mapにputすれば内部で自動的にMap.Entryのインスタンスは作られますが、単独でインスタンス化したいのであれば、設問のようにMapインタ

フェースの**entryメソッド**を使います。このメソッドはstaticですので、インスタンスを生成しなくても利用できます。ただし、このメソッドで作られたMap.Entryオブジェクトは変更不可能です。
また、作ったMap.Entryのインスタンスを使ったMapを作るには、Mapインタフェースの**ofEntriesメソッド**を使います。このメソッドもstaticであり、インスタンス化の必要はありませんが、変更不可能なMapを戻します。

16. B、C、E

➡ P334

java.lang.Comparableは、比較可能なオブジェクトであることを示すためのインタフェースです。並べ替えの対象としたいクラスで、このComparableインタフェースを実現します。たとえば次のコードは、Comparableを実現したクラスの例です。このインタフェースを実現すると、抽象メソッドの**compareTo**を実装しなければいけません。このメソッドは、比較対象となる相手を引数に受け取り、次のルールに基づいて実装します。

- **自分自身が相手よりも前**に位置するよう並べ替えるのであれば、**負の整数**を戻す
- **自分自身が相手よりも後ろ**に位置するように並べ替えるのであれば、**正の整数**を戻す
- **相手と同値**であり、並べ替える必要がないのであれば**0**を戻す

以上のことから、選択肢**B**、**C**、**E**が正解となります。

次のコード例では、価格（price）を使って、自分のほうが安ければ前、高ければ後ろにするよう実装しています。

例 Comparableを実現するItemクラス

```
1. public class Item implements Comparable<Item> {
2.     private String name;
3.     private int price;
4.
5.     public Item(String name, int price) {
6.         super();
7.         this.name = name;
8.         this.price = price;
9.     }
10.
11.    @Override
12.    public int compareTo(Item other) {
13.        if (this.price < other.price) {
14.            return -1;
15.        }
16.        if (other.price < this.price) {
17.            return 1;
18.        }
19.        return 0;
20.    }
21.
22.    public String getName() {
23.        return name;
24.    }
25.    public int getPrice() {
26.        return price;
27.    }
28.    @Override
29.    public String toString() {
30.        return "Item [name=" + name + ", price=" + price + "]";
31.    }
32. }
```

では、実際に並べ替えてみましょう。たとえば、次のように異なる価格を持った3つのItemのインスタンスを用意し、リストに追加します。その後、**java.util.Collectionsクラス**を使って並べます。この並べ替えの際にリスト内のItemは比較されますが、このときに前述のcompareToメソッドが呼び出されます。

例 並べ替えを行うSampleクラス

```
1. import java.util.Collections;
2. import java.util.List;
3. import java.util.Arrays;
4.
5. public class Sample {
6.     public static void main(String[] args) {
7.
8.         Item a = new Item("apple", 100);
9.         Item b = new Item("orange", 120);
10.        Item c = new Item("banana", 80);
11.
```

```
12.         List<Item> list = Arrays.asList(a, b, c);
13.
14.         Collections.sort(list);
15.         list.forEach(System.out::println);
16.     }
17. }
```

このコードを実行すると、次のような結果がコンソールに表示され、価格順に並べ替えられたことがわかります。

例 実行結果

```
> java Sample
Item [name=banana, price=80]
Item [name=apple, price=100]
Item [name=orange, price=120]
```

17. B

➡ P334

解答16では、java.lang.Comparableについて学びました。このインタフェースを実現したクラスは、compareToメソッドにどのように並べ替えをしたいのかというアルゴリズムを記述することができます。
Comparableの問題は、1つしかアルゴリズムを記述できない点です。常に一定の方法で並べ替えるのであれば問題ありませんが、状況に応じた並べ替えには対応できません。Comparableでは並べ替えの対象と並べ替えの方法をひとまとめにして扱っていましたが、これらを分離することでこの問題が解決します。この並べ替えの方法だけを定義するためのインタフェースが、**java.util.Comparator**です。

たとえば次のコード例では、Comparatorインタフェースを実現したItemNameComparatorクラスを定義しています。このクラスでは、Comparatorの抽象メソッド**compareメソッド**を実装し、その中で並べ替えの方法を指定しています。この並べ替えの指定についてはComparableと同じで、1つ目の引数が前なら負の整数、1つ目の引数が後ろなら正の整数、同じなら0を戻すというものです。この例では名前順に並べ替えたいので、（Comparableを実現済みの）Stringクラスが持っているcompareToメソッドをそのまま使っています。なお、並べ替えの対象であるItemクラスについては、解答16のものを流用しています。

例 Comparatorを実現するItemNameComparatorクラス

```
1. import java.util.Comparator;
2.
3. public class ItemNameComparator implements Comparator<Item> {
4.     @Override
5.     public int compare(Item a, Item b) {
6.         return a.getName().compareTo(b.getName());
7.     }
8. }
```

自分が引数で受け取った相手と比較するのがComparableインタフェースのcompareToメソッドであったのに対し、Comparatorインタフェースのcompareメソッドは2つの比較対象の引数を受け取ってそれぞれを比較します。Comparatorのcompareメソッドは引数を2つ受け取りますので、引数を1つしか受け取っていない選択肢AとCは誤りです。
また、比較した結果は前述のとおり正の整数か負の整数、もしくは0で表現します。そのため、戻り値の型がboolean型となっている選択肢Dも誤りです。
以上のことから、選択肢**B**が正解です。

では、実際に並べ替えをしてみましょう。たとえば、次のように異なる名前を持った3つのItemのインスタンスを用意し、リストに追加します。その後、Listの**sortメソッド**の引数に先ほどのItemNameComparatorのインスタンスへの参照を渡します。こうすることで、並べ替えをするときにItemNameComparatorクラスのcomapreメソッドが呼び出されることになります。

例 並べ替えを行うSampleクラス

```
1. import java.util.Arrays;
2. import java.util.List;
3.
4. public class Sample {
5.     public static void main(String[] args) {
6.         Item a = new Item("apple", 100);
7.         Item b = new Item("orange", 120);
8.         Item c = new Item("banana", 80);
9.
10.         List<Item> list = Arrays.asList(a, b, c);
11.
12.         list.sort(new ItemNameComparator());
```

```
13.         list.forEach(System.out::println);
14.     }
15. }
```

このコードを実行すると、次のような結果がコンソールに表示され、名前順に並べ替えられたことがわかります。前問のItemクラスを流用していますが、Itemクラスが持っているcompareToメソッドではなく、Comparatorを実現したItemNameComparatorが利用されたことがわかります。

例 実行結果

```
> java Sample
Item [name=apple, price=100]
Item [name=banana, price=80]
Item [name=orange, price=120]
```

なお、Comparatorインタフェースは関数型インタフェースとして使えます。そのため、ItemNameComparatorクラスを定義しなくても、リストのソートをしている部分は、次のように**ラムダ式**に置き換えることができます。

```
list.sort((o1, o2) -> {
    return o1.getName().compareTo(o2.getName());
});
```

試験対策

java.lang.Comparableインタフェースとjava.util.Comparatorインタフェースの区別を整理しておきましょう。
Comparableは、「比較可能な」オブジェクトであることを示すためのインタフェースで、抽象メソッドはcompareToメソッドです(引数は1つ)。
Comparatorは、並べ替えの方法だけを定義するためのインタフェースで、抽象メソッドはcompareメソッドです(引数は2つ)。

8

アノテーション

- マーカーインタフェース
- アノテーションの定義とアノテーションに関する情報の取得
- 注釈パラメータ
- メタ・アノテーション
- @Retentionアノテーションと保持期間
- @Overrideアノテーション
- @Deprecatedアノテーション
- @SuppressWarningsアノテーション

1. アノテーションの定義として、正しいものを選びなさい。（1つ選択）

A. `public @interface Sample {}`
B. `public interface Sample {}`
C. `public @annotation Sample {}`
D. `public annotation Sample {}`

➡ P376

2. 独自に定義するアノテーションにString型の注釈パラメータ「test」を宣言したい。正しい記述を選びなさい。（1つ選択）

A. `String test;`
B. `void test(String value);`
C. `String test();`
D. `@test`

➡ P379

3. アノテーションの情報をコンパイル後に残したくない。@Retentionの定義として、正しいものを選びなさい。（1つ選択）

A. `@Retention(RetentionPolicy.CLASS)`
B. `@Retention(RetentionPolicy.SOURCE)`
C. `@Retention(RetentionPolicy.RUNTIME)`
D. `@Retention()`

➡ P382

4. 独自のアノテーションを定義するときに、@Targetアノテーションを指定しなかった。このとき、アノテーションの対象となるものを選びなさい。（1つ選択）

A. クラス
B. メソッド
C. フィールド
D. コンストラクタ
E. 上記のすべて

➡ P387

5. スーパークラスのメソッド宣言を変更したときに、サブクラスのメソッドが一致しなくなることを検知するためのアノテーションとして、正しいものを選びなさい。（1つ選択）

A. @Override
B. @Deprecated
C. @SuppressWarnings
D. @FunctionalInterface

➡ P391

6. 非推奨のクラスに付けるアノテーションとして、正しいものを選びなさい。（1つ選択）

A. @Override
B. @Deprecated
C. @SuppressWarnings
D. @FunctionalInterface

➡ P393

7. 非推奨のメソッドの使用時に出力される警告を抑制（オフ）するために指定するアノテーションの記述として、正しいものを選びなさい。（1つ選択）

A. `@SuppressWarnings`
B. `@SuppressWarnings("unchecked")`
C. `@SuppressWarnings("removal")`
D. `@SuppressWarnings("deprecation")`

➡ P393

第8章　アノテーション
解　答

1. A

➡ P374

数多くのクラスがあったとき、あるクラスは処理し、あるクラスは処理しないというように、そのクラスを処理すべきかどうかを判断しなければいけないことがあります。
このような分岐処理に利用するのが、「マーカーインタフェース」と呼ばれるインタフェースです。**マーカーインタフェース**とは、次のTestインタフェースのように何も持たず、クラスに意味を追加することだけを目的としたインタフェースのことです。

```
public interface Test {}
```

マーカーインタフェースは、「このクラスは、このインタフェース型で扱えます」というシールを貼り付けるようなイメージと考えるとよいでしょう。

実際にコードでマーカーインタフェースの利用例を見てみましょう。次のSampleクラスはObject型だけを扱うList型のフィールドlistを持っており、addメソッドで受け取ったObject型の参照をこのリストに追加しています。ただし、何でも追加するのではなく、instanceof演算子を使って先ほどのマーカーインタフェースであるTestインタフェースを実現したものだけを選別して追加しています。

例 Testインタフェースを実現したインスタンスへの参照のみをリストに追加

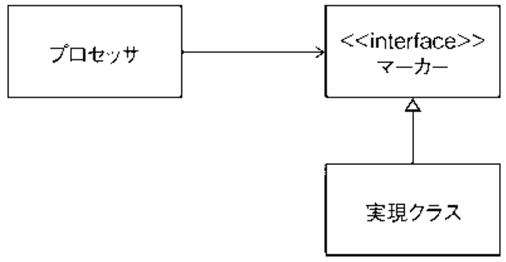
```
1. import java.util.ArrayList;
2. import java.util.List;
3.
4. public class Sample {
5.     private List<Object> list = new ArrayList<>();
6.     public void add(Object obj) {
7.         if (obj instanceof Test) {
8.             this.list.add(obj);
9.         }
10.        // do something
11.    }
12. }
```

このように、何らかの処理の対象になるか、ならないかを見分けるためだけに使うインタフェースのことを「マーカーインタフェース」と呼びます。

java.io.Serializableが、Javaの標準クラスライブラリに含まれるマーカーインタフェースの代表例です。このインタフェースは、1つも実現すべき抽象メソッドを持ちませんが、ファイルやネットワークに出力するためにインスタンスが「直列化できる」ことを表すためのインタフェースです。もし、このインタフェースを実現しないクラスのインスタンスを直列化しようとすると、直列化できないことを表す**java.io.NotSerializableException**という例外がスローされます。

なお、本書では便宜上、マーカーインタフェースによってマークされたクラス（正確にはマーカーインタフェースを実装したクラス）を処理するクラスのことを「**プロセッサ**」と呼びます。

【マーカーインタフェース、実装クラス、プロセッサ】



Java SE 5.0（J2SE 1.5）では、マーカーインタフェースを簡単に表現することを目的として、アノテーション（注釈）が追加されました。**アノテーション**は、「ある処理の対象となるものである」とプロセッサに伝えるための注釈（追加情報）を記述する機能です。これまでは、@Overrideのような事前に用意されたアノテーションを使ってきましたが、ここではアノテーションそのものを定義する方法を学びます。

アノテーションは、プロセッサに何らかの処理をしてもらうために、クラスに意味を追加するという点ではマーカーインタフェースと同じですが、以下の点で異なります。

・プロセッサに値を渡すことができる
・フィールドやメソッド、コンストラクタなど細かな単位でマークすることができる

アノテーションを独自に定義するには、次のように**@interface**というキーワードに続けて定義します。次のコードは、前述のTestインタフェースを「@interface」とすることでアノテーションに変更したものです。

```
public @interface Test {}
```

アノテーションは、コンパイルされるとインタフェースに置き換わります。次の例は、Testアノテーションをコンパイルした結果のクラスファイルをjavapコマンドで成形したものです。

例 javapコマンドの実行結果

```
Compiled from "Test.java"
public interface Test extends java.lang.annotation.Annotation {
}
```

この実行結果からわかるように、アノテーションは**java.lang.annotation.Annotationインタフェース**のサブインタフェースとして定義されています。ただし、アノテーションはマーカーインタフェースと異なり、implementsするわけではなく、**アノテーションが付与されているという情報**を残します。たとえば、次のように@Testアノテーションを付与したItemクラスを用意します。

例 Testアノテーションを付与したItemクラス

```
1. @Test
2. public class Item {}
```

このクラスをコンパイルし、javapコマンドで結果を見てみると、Testインタフェースがimplementsされているのではなく、最後のほうにアノテーションに関する情報が記述されているのがわかります。

例 javapコマンドの実行結果

```
Classfile /Users/shiga/workspace/Annotation/bin/Item.class
  Last modified 2020/02/19; size 309 bytes
  MD5 checksum 5c3319d0f3305405c131ee42ba1ebce4
  Compiled from "Item.java"
public class Item    ←インタフェースは実装されていない
```



```
… 中略 …

RuntimeInvisibleAnnotations:    ←Testアノテーションの情報
  0: #17()
    Test
```

このように実装上の違いはあるものの、基本的にアノテーションはマーカーインタフェースと同じ役割を担っています。マーカーインタフェースをより拡張した機能を提供するものだと理解するとよいでしょう。

前述のとおり、アノテーションを定義するには、@interfaceを使います。よって、選択肢**A**が正解です。選択肢Bは、インタフェースの定義です。選択肢CやDのような記述方法はありません。

2. C

➡ P374

解答1で説明したように、アノテーションはマーカーインタフェースと異なり、プロセッサに値を渡すことができます。アノテーションが付与されたクラスを定義するときに値を記述しておくことで、プロセッサはその値に応じた処理を実行することができます。
たとえば次のコードでは、名前（name）と値段（price）を表すフィールドを持つItemクラスを用意し、@Testアノテーションでそれぞれのフィールドの値を指定しています。

例 Itemクラスが持つフィールドの値を@Testアノテーションで指定

```
1. @Test(name = "test", price = 100)
2. public class Item {
3.     private String name;
4.     private int price;
5.
6.     public Item(String name, int price) {
7.         this.name = name;
8.         this.price = price;
9.     }
10.
11.    public String getName() {
12.        return name;
13.    }
14.
```

```
15.     public int getPrice() {
16.         return price;
17.     }
18. }
```

たとえば、Itemクラスをテストするためのプロセッサを用意したとき、この@Testアノテーションで指定した値を使ってそれぞれのフィールドを初期化し、その値を持つインスタンスでテストを実行するといった、分岐処理が行えます。
アノテーションで値を扱いたい場合、**その値を戻す抽象メソッド**をアノテーションに追加します。**定義するのはフィールドではない**点に注意してください。たとえば、前述のItemクラスの@Testアノテーションでは、nameとpriceの値を指定していました。この場合は、次のように抽象メソッドを追加します。

例 Testアノテーションに抽象メソッドを追加

```
1. public @interface Test {
2.     String name();
3.     int price();
4. }
```

このようなアノテーションで扱う値のことを「**注釈パラメータ**」と呼びます。注釈パラメータで値をプロセッサに渡したい場合は、次の構文のように「**注釈パラメータ名=値**」の書式で記述します。もし、注釈パラメータが複数ある場合には、カンマで区切って「注釈パラメータ名=値」の書式で列挙します。

構文

```
@アノテーション名(注釈パラメータ名=値 [ , 注釈パラメータ名=値 …] )
```

なお、アノテーションに宣言されている注釈パラメータが1つだけで、かつその注釈パラメータ名が**value**である場合、アノテーションが付与されたクラスでは、カッコ「()」の中に注釈パラメータの値だけを記述することができます。
たとえば次のコードは、String型を戻すvalueという注釈パラメータを扱えるようにするため、抽象メソッドを追加したアノテーションです。

例 注釈パラメータとしてvalueを受け取るアノテーション

```
1. public @interface SampleValue {
2.     String value();
3. }
```

このSampleValueというアノテーションを使った例が、次のコードです。SampleValueアノテーションは、valueという名前の注釈パラメータを1つだけ扱います。そのため、このコードのようにカッコの中に値だけを記述できるのです。

例 注釈パラメータの値のみを記述

```
1. @SampleValue("sample")
2. public class UseSample {
3.
4. }
```

上記のような記述方法は省略表記であり、省略せずに記述することももちろん可能です。

例 注釈パラメータと値を記述

```
1. @SampleValue(value = "sample")
2. public class UseSample {
3.
4. }
```

複数の値をアノテーションで扱いたい場合は配列を利用します。次のコードでは、int配列型の値を戻す注釈パラメータを定義しています。

例 注釈パラメータの定義

```
1. public @interface ArrayValues {
2.     int[] data();
3. }
```

複数の値を一度に渡すためには、次のように配列の初期化子を使います。

※次ページに続く

例 アノテーションに複数の値を渡す

```
1. @ArrayValues(data =  {1, 2, 3})
2. public class ArrayValueSample {
3.
4. }
```

アノテーションの注釈パラメータにデフォルト値を設定することもできます。デフォルト値を設定するには、注釈パラメータの定義で**defaultに続けてデフォルト値**を記述します。

例 注釈パラメータのデフォルト値を設定

```
1. public @interface DefaultValue {
2.     String test() default "default";
3. }
```

注釈パラメータで値を受け取るように定義されたアノテーションを使用する場合は、アノテーションが付与されたクラスでは注釈パラメータを省略できません。注釈パラメータを省略するとコンパイルエラーとなりますが、デフォルト値が指定されている場合には、次のように注釈パラメータを省略してもコンパイルエラーとはなりません。

例 注釈パラメータを省略

```
1. @DefaultValue
2. public class DefaultValueSample {
3.
4. }
```

以上のことから、選択肢**C**が正解です。

3. B

➡ P374

メタ・アノテーションについての問題です。
前問では、アノテーションを使ってプロセッサに注釈パラメータとして値を渡す方法について学びました。ここからは、プロセッサ側で注釈パラメータを受け取る方法について理解しましょう。

インスタンスの元となるクラスがどのような定義情報を持っているかを調べ、その定義情報に基づいてインスタンスを操作するための技術を「**リフレクション**」といいます。リフレクションを使えば、そのインスタンスが持っているメソッドやフィールドの種類を調べて、インスタンスを操作することができます。具体的なクラスが何かを知らなくても、インスタンスを扱えるので、さまざまなフレームワークやライブラリなどで利用されている技術です。
なお、リフレクションを利用するためのインタフェースやクラス群は、**java.lang.reflectパッケージ**※1に分類されています。

リフレクションを理解する上でポイントとなるのが**java.lang.Classクラス**です。Classクラスは、クラスがどのように定義されているのかという情報を扱うためのクラスで、この定義情報を取得するには**クラスリテラル**を使います。クラスリテラルとは、「**クラス名.class**」で表すことができるクラスの定義情報です。

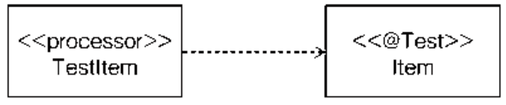
次のコードでは、Itemクラスのクラスリテラルを使って、クラスの定義情報を扱うインスタンスへの参照を取得し、それをClass型変数clazzに代入しています。

例 Itemクラスの定義情報を取得

```
1. public class TestItem {
2.     public static void main(String[] args) {
3.         Class<Item> clazz = Item.class;
4.
5.         // do something
6.     }
7. }
```

なお、この解説では、Testアノテーション、Itemクラス、TestItemクラスの3つが登場しますが、それぞれの関係は次のクラス図のとおりです。TestItemクラスと関係するItemクラスとTestアノテーションは、解答2で挙げたコード例を使います。

【Testアノテーション、Itemクラス、TestItemクラスの関係】



※1 **Java SE 11 & JDK 11 API仕様**｜ https://docs.oracle.com/javase/jp/11/docs/api/java.base/java/lang/reflect/package-summary.html

参考

UMLのクラス図では、図に登場するモデルの意味を変更するために<<～>>で囲まれたステレオタイプを使います。ステレオタイプには、<<interface>>のようにUMLの仕様であらかじめ決まっているものもありますが、自由に定義して使うことができます。この図では、アノテーションが付与されていることを<<@Test>>として表現しました。なお、TestItemには、これがプロセッサであることを表すために<<processor>>というステレオタイプを付与しています。

このように、クラスの定義情報が取得できれば、そのクラスがどのようなフィールドやメソッドを持っているかを調べられます。この定義情報から、そのクラスが持つアノテーションを取得するには、次のようにClassクラスの**getAnnotationメソッド**を使います。このメソッドの引数に、取得したい定義情報を持つアノテーションのクラスリテラルを渡します。

次のコードでは、Itemクラスの定義情報から、Itemクラスに付与されているTestアノテーションの情報を取り出しています。もし、指定したTestアノテーションが、対象となるItemクラスに存在しなければnullを戻し、returnで処理を中断しています。

例 Itemクラスの定義情報から、Testアノテーションの定義情報を取得

```
 1. public class TestItem {
 2.     public static void main(String[] args) {
 3.         Class<Item> clazz = Item.class;
 4.         Test test = clazz.getAnnotation(Test.class);
 5.
 6.         if (test == null) {
 7.             return;
 8.         }
 9.
10.         // do something
11.     }
12. }
```

Itemクラスの定義情報からTestアノテーションに関する情報を取得できれば、Testアノテーションが持つ注釈パラメータを戻す抽象メソッドを使って、注釈パラメータの値を取得することができます。次のコードでは、nameメソッドとpriceメソッドを使って名前と値段の値を取得しています。

例 注釈パラメータの値を取得

```
 1. public class TestItem {
 2.     public static void main(String[] args) {
 3.         Class<Item> clazz = Item.class;
 4.         Test test = clazz.getAnnotation(Test.class);
 5.
 6.         if (test == null) {
 7.             return;
 8.         }
 9.
10.         String name = test.name();
11.         int price = test.price();
12.
13.         Item item = new Item(name, price);
14.
15.         System.out.println(item.getName());
16.         System.out.println(item.getPrice());
17.     }
18. }
```

解答2では、ItemクラスにはTestアノテーションを付与し、名前（name）にtest、値段（price）に100という値の注釈パラメータを定義していました。そのため、上記のプロセッサを実行すると、それぞれの注釈パラメータの値が表示されるはずですが、実際には何も表示されません。アノテーションが取得できず、途中のif文でreturnされてしまったためです。しかし、ItemクラスにはTestアノテーションが付与されていたはずです。

ここで意識しなければいけないのが、**アノテーションの保持期間**です。アノテーションには3種類の保持期間があります。デフォルトは、クラスファイルには保持しているものの、実行時にアノテーションに関する情報は破棄するというものです。そのため、TestItemの実行時にTestアノテーションの情報が取得できなかったのです。

アノテーションの保持期間を表すには、**java.lang.annotationパッケージ**の**RetentionPolicy**という列挙型を使います。この列挙型には、次の3つの列挙子が用意されています。

※次ページに続く

【RetentionPolicyの列挙子】

種類	内容
CLASS	アノテーションに関する情報はクラスファイルには残っているものの、実行時に破棄される。クラスファイルに事前処理するようなプログラムを作るときに使用する
RUNTIME	実行時までアノテーションに関する情報を保持する。実行時にアノテーションを処理するために使用する
SOURCE	コンパイル時にアノテーションに関する情報は破棄される。コンパイラにアノテーションを処理させたいときに使用する

プログラマーがRetentionPolicyを使用するのは、実行時に何らかの処理をするプロセッサを作るときなので、**RUNTIME**を使うことがほとんどです。ほかの2つのRetentionPolicyは、用意されてはいるものの、一般的な用途で使うことはあまりないでしょう。

アノテーションの保持期間は、アノテーションを定義するときに**@Retention アノテーション**を使って指定します。この@Retentionは、アノテーションを定義するためにあらかじめ用意されているものです。このようにアノテーションを定義するためのアノテーションのことを「**メタ・アノテーション**」と呼びます。

次のコードは、解答2で扱っていたTestアノテーションに、@Retentionアノテーションを付与している例です。この例では、RetentionPolicyにRUNTIMEを注釈パラメータとして指定しているため、このTestアノテーションは実行時まで削除されることなくクラスファイルに残ります。

例 RetentionPolicyにRUNTIMEを指定

```
1. import java.lang.annotation.Retention;
2. import java.lang.annotation.RetentionPolicy;
3.
4. @Retention(RetentionPolicy.RUNTIME)
5. public @interface Test {
6.     String name();
7.     int price();
8. }
```

ここまでできたら、先ほどのTestItemクラスをもう一度実行してみましょう。次のようにアノテーションが実行時まで残っていることが確認できるはずです。

例 実行結果

```
> java TestItem
test
100
```

設問では、「アノテーションの情報をコンパイル後に残したくない」とあります。そのため、コンパイル時にアノテーションを削除するRetentionPolicyは、SOURCEとなります。以上のことから、選択肢**B**が正解です。

4. E

➡ P374

メタ・アノテーションに関する問題です。
マーカーインタフェースはクラスそのものをマークするために使用しますが、アノテーションを使えば、さらに細かい単位でマークを付けることができます。**@Targetアノテーション**は、そのアノテーションが何を対象としているのかを指定するためのメタ・アノテーションです。
@Targetアノテーションの注釈パラメータに渡す列挙型である**java.lang.annotation.ElementType**の列挙子には、次の表に挙げる11種類があります。このことから、アノテーションによってマークできる箇所は、マーカーインタフェースに比べて飛躍的に増え、多岐にわたることがわかります。

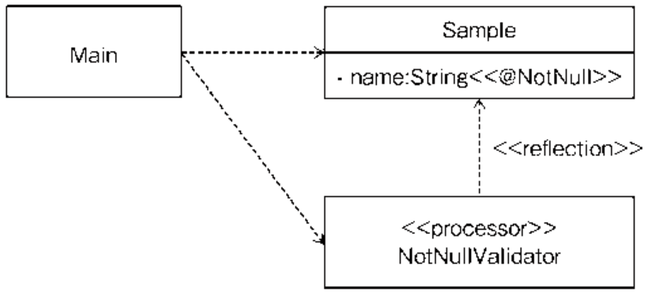
【ElementTypeの列挙子】

列挙子	対象
ANNOTATION_TYPE	アノテーション宣言
CONSTRUCTOR	コンストラクタ宣言
FIELD	フィールド宣言
LOCAL_VARIABLE	ローカル変数宣言
METHOD	メソッド宣言
MODULE	モジュール宣言
PACKAGE	パッケージ宣言
PARAMETER	メソッドの引数宣言
TYPE	クラス、インタフェース、Enum宣言
TYPE_PARAMETER	型パラメータ宣言
TYPE_USE	型の使用

なお、@Targetアノテーションで、アノテーションを付与する先を指定しなかった場合は、ElementTypeの列挙子にある箇所であれば、どこでもアノテーションの対象となります。よって、選択肢**E**が正解です。

※次ページに続く

では、実際にサンプルを見てみましょう。ここから解説するサンプルは、次のクラス図のような関係になっています。Sampleクラスはnameフィールドを持ちますが、このフィールドに、独自で定義した@NotNullアノテーションを付けています。また、リフレクションを使ってSampleクラスを処理するのが、NotNullValidatorクラスです。このクラスは、@NotNullアノテーションが付与されているフィールドが、nullでないかどうかを検証するクラスです。



まずは、アノテーションを定義するところから始めましょう。次のコードは、フィールドに付与するタイプのアノテーションを定義したものです。

例 @Targetアノテーションを付与する先をフィールドとする定義

```
1. import java.lang.annotation.ElementType;
2. import java.lang.annotation.Retention;
3. import java.lang.annotation.RetentionPolicy;
4. import java.lang.annotation.Target;
5.
6. @Target(ElementType.FIELD)
7. @Retention(RetentionPolicy.RUNTIME)
8. public @interface NotNull {
9.     String message();
10. }
```

この独自に作ったNotNullアノテーションは、フィールドの値としてnullを許容しないことを示すためのものです。プロセッサは、インスタンスがこのアノテーションでマークされたフィールドを持っていた場合、何らかのエラーを出力する処理を実行することを想定しています。たとえば、画面で入力された値を保持するオブジェクトを用意しておき、入力内容が妥当かどうかを判定する入力チェックを行うようなプロセッサだとしましょう。
NotNullアノテーションでマークするクラスの例は、次のとおりです。nameフィールドに先ほどのNotNullアノテーションを付与しています。また、**注釈パラメータとしてエラーメッセージを渡している**点にも着目してください。

例 フィールドに対してNotNullアノテーションを付与

```
1. public class Sample {
2.     @NotNull(message = "name is not allowed null")
3.     private String name;
4.     public Sample(String name) {
5.         this.name = name;
6.     }
7.     public String getName() {
8.         return this.name;
9.     }
10. }
```

次に、プロセッサの例を見てみましょう。このプロセッサのvalidateメソッドは、NotNullアノテーションが付与されているフィールドを持つオブジェクトが渡されると、そのフィールドの値がnullでないかをチェックし、nullであればエラーメッセージを戻すというものです。

例 オブジェクトのフィールドの値がnullかどうかをチェックするプロセッサ

```
1. import java.lang.reflect.Field;
2. import java.util.ArrayList;
3. import java.util.List;
4.
5. public class NotNullValidator {
6.     public static List<String> validate(Object target) {
7.         try {
8.             List<String> messages = new ArrayList<String>();
9.             Class clazz = target.getClass();
10.            Field[] fields = clazz.getDeclaredFields();
11.            for (Field field : fields) {
12.                field.setAccessible(true);
13.                NotNull annotation = field.getAnnotation(NotNull.class);
14.                if (annotation == null) {
15.                    continue;
16.                }
17.
18.                if (field.get(target) != null) {
19.                    continue;
20.                }
21.                messages.add(annotation.message());
```

```
22.             }
23.             return messages;
24.         } catch (IllegalArgumentException | IllegalAccessException e) {
25.             throw new RuntimeException(e);
26.         }
27.     }
28. }
```

このコードでは、まずクラスの定義情報を取得し（9行目）、そこから**getDeclareFiledsメソッド**を使ってフィールドの定義情報の一覧を取得します（10行目）。11行目から始まる拡張for文では、1つずつフィールドの定義情報に従って繰り返し処理をしています。この繰り返し処理では、privateフィールド（Sampleクラスのname）にアクセスするため、**java.lang.reflect.Fieldクラス**の**setAccessibleメソッド**を使ってフィールドのアクセス情報を変更してから、フィールドからアノテーション情報を取り出します。

もし、そのフィールドにアノテーションが付いていなければアノテーション情報はnullなので、continueで繰り返し処理の先頭に戻り、次のフィールドを処理します。さらにフィールドの値を取り出し、その値がnullでない場合もcontinueで繰り返し処理の先頭に戻り、次のフィールドを処理します。もし、これら2つのif文の条件に一致しなければ、アノテーションが付いていたフィールドの値はnullということになり、アノテーションからメッセージ（注釈パラメータ）を取り出し、リストmessagesに追加します。

このプロセッサを使った入力チェックのプログラムは次のとおりです。このコードでは、意図的にコンストラクタにnullを渡したSampleクラスのインスタンスを生成し、それをプロセッサに渡してチェックさせています。

例 プロセッサNotNullValidatorを使った入力チェックのプログラム

```
1. import java.util.List;
2.
3. public class Main {
4.     public static void main(String[] args) throws Exception {
5.         Sample sample = new Sample(null);
6.         List<String> messages = NotNullValidator.validate(sample);
7.         if (messages.isEmpty()) {
8.             System.out.println("valid object");
9.             return;
10.         }
11.         for (String message : messages) {
12.             System.out.println(message);
13.         }
14.     }
15. }
```

メッセージが入ったリストが空でなければ、次のようにエラーメッセージがコンソールに表示されるという仕組みになっています。

```
name is not allowed null
```

5. A

➡ P375

@Overrideアノテーションに関する問題です。
あるクラスを継承したサブクラスで、スーパークラスのメソッドを**再定義**することを「**オーバーライド**」と呼びます。たとえば、次のようにhelloというメソッドを持つAクラスがあったとします。

例 スーパークラスAのhelloメソッド

```
1. public class A {
2.     public void hello() {
3.         System.out.println("hello.");
4.     }
5. }
```

これを次のBクラスのようなサブクラスを作って、helloメソッドをオーバーライドしたとします。

例 Aクラスを継承するサブクラスBのhelloメソッド

```
1. public class B extends A {
2.     public void hello() {
3.         System.out.println("hi.");
4.     }
5. }
```

サブクラスがオーバーライドしたスーパークラスのメソッドの引数を変更すると、スーパークラスのメソッドとサブクラスのメソッドのシグニチャが異

なることになります。

例 Aクラスのhelloメソッドを変更

```
1. public class A {
2.     public void hello(String name) {
3.         System.out.println("hello." + name);
4.     }
5. }
```

スーパークラスのメソッドの引数を変更すると、オーバーライドではなく、**オーバーロード（多重定義）**しているとコンパイラに解釈されます。
設計者からすると、このような解釈は望んでいないこともあります。スーパークラスのメソッドが変更されたのなら、オーバーライドしたサブクラスのメソッドも変更するべきであると考えるからです。そうでなければ、そもそもサブクラスでオーバーライドしていた意味がありません。

また、スーパークラスの変更に伴ってサブクラスを変更する必要がある場合に、オーバーライドしたメソッドの変更をし忘れてしまう可能性もあります。そこで、オーバーライドしたメソッドを変更し忘れたことをコンパイラが検知できる仕組みとして、**@Overrideアノテーション**が提供されています。

例 @Overrideアノテーションが付与されたメソッド

```
1. public class B extends A {
2.     @Override
3.     public void hello() {
4.         System.out.println("hi.");
5.     }
6. }
```

サブクラスでオーバーライドしたメソッドに@Overrideアノテーションを付けることにより、スーパークラスのオーバーライドされた側のメソッドとシグニチャが一致するかどうかをコンパイラがチェックし、一致しなければコンパイルエラーを出力します。

以上のことから、スーパークラスの変更を検知するために使えるアノテーションは、@Overrideアノテーションです。よって、選択肢**A**が正解です。
選択肢Bの@Deprecatedは、バージョン変更などによって使用できなくなる可能性があるものに付与するためのアノテーションです。選択肢Cの@SuppressWarningsは、コンパイラが発する警告を抑制するためのアノテーションです。

選択肢Dの@FunctionalInterfaceは、そのインタフェースが関数型インタフェースであることを示すためのアノテーションです。

6. B

➡ P375

@Deprecatedアノテーションに関する問題です。
長いソフトウェアのライフサイクルの中で、モジュール、インタフェース、クラスが提供する機能は変わっていきます。そのとき、下位互換性をどのくらい担保するかが課題になってきます。下位互換性とは、ソフトウェアが変更され、新しいバージョンになっても、変更される前に使えた機能がそのまま使えるという性質を指します。
この変更前に使えた機能は、変更後の機能によって置き換え可能であるといえます。しかし、置き換えられた「変更前の機能」が使えなくなると、その部分に依存しているクラスが動作しなくなります。

そこで、変更直後は変更前の機能をそのまま使えるようにしておき、変更後の機能へ移行する一定の期間を設ける方法が採られます。この移行期間に入るタイミングで、変更前の機能を使用しないように促すために、「**非推奨**」であることを明示します。

非推奨であることを明示するには、仕様書などのドキュメントやJavadocに記載する方法などがありますが、Java SE 5.0から**@Deprecatedアノテーション**が使えるようになりました。@Deprecatedアノテーションでマークされた非推奨のクラス、メソッド、またはフィールドがプログラム中に使用されている場合、コンパイラが警告を発します。この警告は、非推奨の機能を新しい機能に置き換えるよう促すものです。

以上のことから、選択肢**B**が正解となります。選択肢Aの@Overrideは、サブクラスのメソッドがスーパークラスのメソッドをオーバーライドしているかどうかをチェックするアノテーションです。選択肢Cの@SuppressWarningsは、コンパイラが発する警告を抑制するためのアノテーションです。選択肢Dの@FunctionalInterfaceは、関数型インタフェースであることを示すためのアノテーションです。よって、これらの選択肢は誤りです。

7. D

➡ P375

@SuppressWarningsアノテーションに関する問題です。
Javaのコンパイラが発する警告は、プログラムの不具合を引き起こす可能性がある箇所を示唆します。あくまでも警告であって、コンパイルエラーではないためプログラマーは無視することもできますが、不具合の可能性があることを示唆するものが多いため、できるだけ警告が出力されないようにプロ

グラミングするほうがよいでしょう。

ただし、警告が出力されるようにプログラマーが意図的にコードを記述することもあります。こうした場合、意図していなかった箇所の警告と混同しないようにするために、**@SuppressWarningsアノテーション**を使うことで警告を抑制（オフ）できるようになっています。

たとえば次のコードでは、ジェネリクスを使わずリストに要素を追加しています。このようなジェネリクスを使っていないリストは、互換性のない型のインスタンスへの参照を混在させることができるため、ClassCastExceptionが発生する原因になります。そこでコンパイラは、このようなコードを検知して警告を発します。

例 コンパイルすると警告が発生するコード

```
1. import java.util.ArrayList;
2. import java.util.List;
3.
4. public class Sample {
5.     public void sample() {
6.         List list = new ArrayList();
7.         list.add("String");
8.     }
9. }
```

このコードをコンパイルすると、コンパイルそのものはできますが、次のような警告がコンソールに表示されます。

例 実行結果

```
> javac Sample.java
ノート:suppress/Sample.javaの操作は、未チェックまたは安全ではありません。
ノート:詳細は、-Xlint:uncheckedオプションを指定して再コンパイルしてください。
```

そこで、次の@SuppressWarningsアノテーションを使うと、コンパイル時の警告を抑制することができます。このコードをコンパイルすると、何も警告されずにコンパイルできるはずです。

例 @SuppressWarningsアノテーションで警告を抑制

```
1. import java.util.ArrayList;
2. import java.util.List;
3.
4. // コンパイラの警告を抑制することを明示
5. @SuppressWarnings("unchecked")
6. public class Sample {
7.     public void sample() {
8.         List list = new ArrayList();
9.         list.add("String");
10.     }
11. }
```

@SuppressWarningsアノテーションには、注釈パラメータとして抑制したい対象を表す文字列を渡します。Java言語仕様で規定されている文字列は、「**unchecked**」「**deprecation**」「**removal**」の3つです。

uncheckedは、前述のとおりです。deprecationとremovalは、@Deprecatedアノテーションと関係する注釈パラメータ値です。たとえば、次のように@Deprecatedアノテーションが付与されたメソッドを持つクラスがあったとします。

例 @Deprecatedアノテーションが付与された非推奨のメソッド

```
1. public class Test {
2.     @Deprecated  // testメソッドが非推奨であることを明示
3.     public void test() {}
4. }
```

このtestメソッドは、今後の改修で変更されることが決まっているため、「非推奨のメソッド」となっているものとします。次のコードのように非推奨のメソッドを使用すると、コンパイラは警告を発します。

例 非推奨のメソッドの使用

```
1. public class Sample {
2.     public void sample() {
3.         new Test().test();
4.     }
5. }
```

しかし、プログラマーが何らかの理由で、このメソッドが非推奨であることを理解した上で使っているとした場合、次のように@SuppressWarningsアノテーションにdeprecationを渡してマークすると警告を抑制できます。

例 非推奨のメソッドの使用時に警告を抑制

```
1. // 非推奨のメソッドを使用しても警告を抑制することを明示
2. @SuppressWarnings("deprecation")
3. public class Sample {
4.     public void sample() {
5.         new Test().test();
6.     }
7. }
```

@Deprecatedアノテーションは非推奨だけでなく、今後のリリースで削除が予定されているため、より強く使用を抑制したい場合にも、次のように注釈パラメータを使うことができます。

例 将来的に削除予定のあるメソッドに対するアノテーションの指定

```
1. public class Test {
2.     @Deprecated(forRemoval = true)
3.     public void test() {}
4. }
```

ただし、このように使用を強く抑制されているメソッドは、@SuppressWarningsアノテーションにdeprecationを渡してマークするだけでは抑制できません。テストなどの目的で、一時的に使用するために警告を抑制したい場合には、次のように@SuppressWarningsアノテーションにremovalという注釈パラメータ値を渡します。

例 @Deprecated(forRemoval = true)がマークされたメソッドの使用時に警告を抑制

```
1. @SuppressWarnings("removal")
2. public class Sample {
3.     public void sample() {
4.         new Test().test();
5.     }
6. }
```


以上のことから、選択肢Dが正解となります。

なお、@SuppressWarningsアノテーションと、@Deprecatedアノテーションの関係による警告のオンとオフは、次の表のとおりとなります。

【警告のオン／オフ】

非推奨のメソッドの宣言側	非推奨のメソッドの呼び出し側	警告のオン／オフ
@Deprecated	@SupressWarningsなし	オン
@Deprecated	@SupressWarnings("deprecation")	オフ
@Deprecated(forRemoval = "true")	@SupressWarnings("deprecation)	オン
@Deprecated(forRemoval = "true")	@SupressWarnings("removal")	オフ

9

例外とアサーション

- マルチキャッチ
- try-with-resources
- 抑制された例外
- アサーション

1. 次のコードのうち、正しく記述されているものを選びなさい。（1つ選択）

A.
```
try {
    // do something
} catch (AException a | BException b) {
    // do something
}
```

B.
```
try {
    // do something
} catch (AException a || BException b) {
    // do something
}
```

C.
```
try {
    // do something
} catch (AException a & BException b) {
    // do something
}
```

D.
```
try {
    // do something
} catch (AException a && BException b) {
    // do something
}
```

➡ P405

2. 複数の例外を同時にキャッチする「マルチキャッチ」に関する説明として、正しいものを選びなさい。（1つ選択）

A. マルチキャッチでは、RuntimeException を先に記述し、そのあとにExceptionを記述しなければならない
B. マルチキャッチでは、Exceptionを先に記述し、そのあとにRuntimeExceptionを記述しなければならない
C. マルチキャッチでは、例外クラスを記述する順序についての規定はない
D. ExceptionとRuntimeExceptionをマルチキャッチで同時に扱うことはできない

➡ P406

3. プログラマーが独自に作成する例外クラスに関する説明として、最も適切なものを選びなさい。（1つ選択）

A. java.lang.RuntimeExceptionクラスのサブクラスでなければいけない
B. java.lang.Exceptionクラスのサブクラスでなければいけない
C. java.lang.Throwableクラスのサブクラスでなければいけない
D. java.lang.Errorクラスのサブクラスでなければいけない

➡ P407

4. try-with-resourcesの説明として正しいものを選びなさい。（1つ選択）

A. catchブロックは必須である
B. finallyブロックは必須である
C. catchブロックかfinallyブロックのどちらかを記述しなければならない
D. try-with-resourcesの主な目的は例外処理ではない
E. 選択肢A～Eはすべて誤りである

➡ P408

5. try-with-resources文で自動的に閉じられるリソースを定義することができるのは、次のうちどれか。正しいものを選びなさい。（1つ選択）

A. java.io.Closeableインタフェースを実装したクラスのみ
B. java.lang.AutoCloseableインタフェースを実装したクラスのみ
C. java.io.Closeableインタフェースとjava.lang.AutoCloseableインタフェースの両方を実装したクラスのみ
D. java.io.Closeableインタフェースとjava.lang.AutoCloseableインタフェースのいずれかを実装したクラス

➡ P410

6. try-with-resourcesの記述として、正しいものを選びなさい。（2つ選択）

A.
```
try (A a = new A()) {
    // do something
}
```

B.
```
B b = null;
    try (b = new B()) {
        // do something
}
```

C.
```
C c = new C();
try (c) {
    // do something
}
```

D.
```
try (D d = new D()) {
    d = new D();
}
```

E.
```
E e = new E();
try (e) {
    // do something
}
e = new E();
```

➡ P412

□ **7.** 次のプログラムをコンパイル、実行した結果として、正しいものを選びなさい。(1つ選択)

```
1. public class A implements AutoCloseable {
2.     @Override
3.     public void close() throws Exception {
4.         System.out.println("A");
5.     }
6. }
```

```
1. public class B implements AutoCloseable {
2.     @Override
3.     public void close() throws Exception {
4.         System.out.println("B");
5.     }
6. }
```

```
1. public class C implements AutoCloseable {
2.     @Override
3.     public void close() throws Exception {
4.         System.out.println("C");
5.     }
6. }
```



```
1. public class TestUsing {
2.     public static void main(String[] args) throws Exception {
3.         A a = new A();
4.         try (a;
5.             B b = new B();
6.             C c = new C()) {
7.             // do something
8.         }
9.     }
10. }
```

A. 「A」「B」「C」と表示される
B. 「B」「C」「A」と表示される
C. 「C」「B」「A」と表示される
D. 「A」「C」「B」と表示される

➡ P415

□ **8.** try-with-resourcesのtryブロック内で例外が発生したとき、リソースのクローズ、catchブロック、finallyブロックはどの順番で実行されるか。正しい順番となっているものを選びなさい。(1つ選択)

A. finally, catch, close
B. close, catch, finally
C. catch, finally, close
D. catch, close, finally

➡ P415

□ **9.** 次のプログラムをコンパイル、実行したときの結果として、正しいものを選びなさい。(1つ選択)

```
1. public class TroubleResource implements AutoCloseable {
2.     @Override
3.     public void close() throws Exception {
4.         throw new RuntimeException("trouble");
5.     }
6. }
```

※次ページに続く

```
1. public class TroubleMaker {
2.     public static void main(String[] args) {
3.         try (TroubleResource a = new TroubleResource()) {
4.             throw new Exception();
5.         } catch (RuntimeException e) {
6.             System.out.println("A");
7.         } catch (Exception e) {
8.             System.out.println("B");
9.         }
10.    }
11. }
```

A. 「A」と表示される
B. 「B」と表示される
C. コンパイルエラーが発生する
D. 実行時にエラーが発生し、プログラムが強制終了する

➡ P416

□ 10. 引数paramの値がnullでないことを確認するためのアサーションを記述したい。正しい記述を選びなさい。(1つ選択)

A. `assert param == null;`
B. `assert param != null;`
C. `assert(param == null);`
D. `assert(param != null);`

➡ P418

□ 11. アサーションを有効にするためのコマンドとして、正しいものを選びなさい。(1つ選択)

A. `java -assert Sample`
B. `java -enable Sample`
C. `java -ea Sample`
D. `java -assertion Sample`

➡ P421

第9章　例外とアサーション
解　答

1. A

➡ P400

例外のマルチキャッチに関する問題です。
Javaの例外処理についての基本的な考え方は、「通常処理と例外処理を分けて記述する」です。Javaでは、通常処理はtryブロックに、例外処理はcatchブロックに分けて記述することでコードの可読性を高め、仕様変更などのコード変更時に通常処理なのか、例外処理なのかの確認を容易にしようとしています。これは重要な方法ですが、開発するソフトウェアの規模が大きくなるにつれて新たな問題が発生します。

通常処理を記述する1つのtryブロック内で、いくつもの種類の例外が発生する可能性はあります。そのとき、発生する例外が検査例外であった場合、発生する例外の種類に応じてcatchブロックを作り、例外処理を記述しなくてはいけません。たとえば、tryブロック内の処理で2種類の例外が発生するのであれば2つ、3種類なら3つのcatchブロックを作り、それぞれ例外処理を記述する必要があります。

例 3種類の例外が発生する場合の例外処理

```
try {
    // 3種類の例外が発生する処理
} catch (AException e) {
    // 例外処理
} catch (BException e) {
    // 例外処理
} catch (CException e) {
    // 例外処理
} finally {
    // finally処理
}
```

Java SE 7で導入された例外の**マルチキャッチ機能**を使うことによって、例外の種類ごとに記述していたcatchブロックの例外処理を1つにまとめることができるため、catchブロックがいくつも連なるようなコードを簡略化することができます。

※次ページに続く

次のコード例は、マルチキャッチを使って先ほどの3つのcatchブロックを1つにまとめたものです。マルチキャッチを使うことで、コード量が劇的に減り、読みやすくなりました。

例 マルチキャッチを使った例外処理

```
try {
    // 3種類の例外が発生する処理
} catch (AException | BException | CException e) {
    // 例外処理
} finally {
    // finally処理
}
```

以上のことから、選択肢**A**が正解となります。

2. D

➡ P400

例外のマルチキャッチに関する問題です。
例外のマルチキャッチでは、同じ例外処理を1つのcatchブロックにまとめて記述することができます。同じ例外処理をしないのであれば、まとめるべきではありません。たとえば、AExceptionとBExceptionが発生したときは同じ例外処理を実行し、CExceptionが発生したときに別の例外処理を実行したい場合には、次のようにcatchブロックを分けます。

例 マルチキャッチを使った例外処理

```
try {
    // 3種類の例外が発生する処理
} catch (AException | BException e) {
    // 例外処理
} catch (CException e) {
    // 例外処理
} finally {
    // finally処理
}
```

ただし、1つのcatchブロックに記述できるのは、**継承関係にない例外**だけです（選択肢**D**）。たとえば、次のようにBExceptionクラスは、AExceptionクラスを継承しているとしましょう。

例 AExceptionクラスを継承するBExceptionクラス

```
1. public class BException extends AException {
2.
3. }
```

この場合、前述のマルチキャッチを使ったコードは「その例外はすでにキャッチされている」という旨のメッセージと共にコンパイルエラーが発生します。これは、継承関係にある例外は、ポリモーフィズムを使えばスーパークラス型の例外として扱えるためです。BExceptionへの参照は、BException型だけでなくAException型としても扱えるため、そもそもマルチキャッチにする必要がないのです。そのため、前述のコードは、次のようにAExceptionをキャッチするだけでよくなります。

例 ポリモーフィズムにより、例外をスーパークラス型で扱う

```
try {
    // 3種類の例外が発生する処理
} catch (AException e) {
    // 例外処理
} catch (CException e) {
    // 例外処理
} finally {
    // finally処理
}
```

3. B

➡ P401

Javaの標準クラスライブラリには、あらかじめ多くの例外クラスが用意されています。たとえば、次のようなものがあります。

・参照先がないのに参照先のメソッドを呼び出そうとした
・アクセス権限のないファイルにアクセスしようとした
・数値を0（ゼロ）で除算しようとした
・不正な引数でメソッドが呼び出された
・データベースでSQLの実行に失敗した

このように、さまざまなプログラムの実行中のトラブルに対応する例外が用意されていますが、標準クラスライブラリの例外クラスではすべてをカバーしきれません。たとえば、業務上のルールに反していることを示すものや、ソフトウェアアーキテクチャの構造上不可欠なものなどは、ソフトウェア開

発時に独自の例外クラスとして定義する必要があります。
Javaでは、プログラムが対処できるトラブルは「**例外**」、プログラムで対処できないトラブルは「**エラー**」とされています。
例外とエラーに共通しているのが、スロー（通知）できるという性質です。この性質を表すのが、すべてのエラーと例外のスーパークラスである**java.lang.Throwableクラス**です。エラーや例外の発生時、Throwableクラスまたはそのサブクラスのインスタンスが JVMによってスローされます。Throwable直下のサブクラスには、エラーを表す**java.lang.Error**と例外を表す**java.lang.Exception**があります。独自の例外クラスを定義する際にはExceptionを継承しなければいけません。

また、Javaの例外は、例外処理（try catch文、もしくはthrows宣言）の有無をコンパイラがチェックする**検査例外**と、チェックしない**非検査例外**に分かれます。Exceptionクラスとそのサブクラスは基本的に検査例外ですが、**java.lang.RuntimeExceptionクラス**とそのサブクラスだけは非検査例外として扱われます。

独自の例外クラスがExceptionクラスのサブクラスであることは必要条件ですが、RuntimeExceptionクラスのサブクラスである必要はありません。そのため、選択肢Aが誤りで、選択肢Bが正解となります。
独自に定義した例外クラスは、Throwableクラスのサブクラスでなくてはいけませんが、Exceptionクラスのサブクラスでなくてはならないとしたほうが、より適切な説明です。よって、選択肢Cは誤りです。Errorクラスは、プログラムで制御できないトラブルを表すクラスですので、独自の例外クラスを定義するためには使用しません。よって、選択肢Dも誤りです。

4. D ➡ P401

try with resourcesに関する問題です。
try-with-resources（リソース付きtryブロック）はJava SE 7で導入された機能で、プログラムの中で扱うリソースを自動的に閉じるためのものです。
一般的にプログラムで扱うリソースとは、プログラムからアクセスするデータを指しますが、Javaではデータだけでなく、アクセスするためのインスタンスなどもリソースとして扱います。データが書き込まれているファイルだけでなく、ファイルにアクセスするFileクラスやFileReaderクラスのインスタンスも含まれると考えてください。

リソースは、使うときに開き、使い終わったら閉じなければいけません。大量のデータを効率よく管理するDBMSのように、リソースの種類によってはアクセスできる数に限りがあったり、対象となるリソースへのアクセス権を限定したりするものがあります。このようなリソースは、使用後に閉じなければ、アクセスできなくなる可能性があるのです。
問題は、「リソースを使い終わったら閉じる」のは任意であって、必須ではないため、プログラマーが閉じ忘れる場合があることです。リソースを閉じるコードを記述しなくてもコンパイルエラーにはなりませんし、リソースを閉じなくても正常に処理できてしまうので、テストでも見つかりにくいのです。このようなミスを防ぐために、自動的にリソースを閉じるための構文としてJava SE 7で導入されたのが、try-with-resources文です。

まず、try-with-resources文以前のコードを見てみましょう。次のコード例は、テキストファイルの内容をコピーするものです。**finallyブロック**で、入力のためのリソースと出力のためのリソースの**closeメソッド**を呼び出し、リソースを閉じている点に注目してください。

例 try-with-resources文以前のコード

```
 1. import java.io.BufferedReader;
 2. import java.io.BufferedWriter;
 3. import java.io.File;
 4. import java.io.FileReader;
 5. import java.io.FileWriter;
 6. import java.io.IOException;
 7.
 8. public class FileCopy {
 9.     public static void main(String[] args) {
10.         BufferedReader in = null;
11.         BufferedWriter out = null;
12.         try {
13.             in = new BufferedReader(new FileReader(new
                        File("input.txt")));
14.             out = new BufferedWriter(new FileWriter(new
                        File("result.txt")));
15.
16.             String line = null;
17.             while((line = in.readLine()) != null) {
18.                 out.append(line);
19.                 out.newLine();
20.             }
21.             out.flush();
22.         } catch (IOException e) {
23.             throw new RuntimeException(e);
24.         } finally {
```

```
25.             if (in != null) {
26.                 try {
27.                     in.close();
28.                 } catch (IOException e) {
29.                     throw new RuntimeException(e);
30.                 }
31.             }
32.             if (out != null) {
33.                 try {
34.                     out.close();
35.                 } catch (IOException e) {
36.                     throw new RuntimeException(e);
37.                 }
38.             }
39.         }
40.     }
41. }
```

このリソースを閉じるfinallyブロックを丸ごと削除してもコンパイルエラーにはなりません。リソースが解放されない可能性があっても、文法上の間違いはないからです。このプログラムはmainメソッド1つだけの簡単なものですが、数多くのクラスが連携し、データベースやネットワークを経由した別のシステムとの連携や並列処理などの実行中にリソースの閉じ忘れがあると、システム全体が動作しなくなる可能性があります。こうしたリソースの閉じ忘れによる不具合が発生しないように、try-with-resources文以前は、finallyブロックで必ずリソースを閉じるコードを記述する必要がありました。

このように、**try-with-resources文の主な目的は例外処理ではなく、リソースの閉じ忘れを防ぐこと**です。例外処理のためのtryブロックは、それに続くcatchブロックかfinallyブロックのいずれかが必須でしたが、try-with-resources文では、**catchブロックかfinallyブロック、もしくはその両方が省略可能**です。以上のことから、選択肢**D**が正解となります。

5. D ➡ P401

try-with-resources文で処理できる対象についての問題です。
try-with-resources文のカッコ「()」で扱えるのは、**java.lang.AutoCloseableインタフェース**、もしくは**java.io.Closeableインタフェース**を実装したクラスです。よって、選択肢**D**が正解です。
2つのインタフェースの違いは、導入された目的です。try-with-resourcesがJava SE 7で導入され、それに対応するようにAutoCloseableインタフェースは導入されました。一方、CloseableインタフェースはJava SE 5で導入されており、try-with-resourcesが導入される以前から存在しています。Closeableインタフェースは、java.io.BufferedReaderのようなストリームを扱うクラスがそれぞれ実装していた**closeメソッド**が共通の型として扱えるように作られたものです。
Closeableインタフェースは、Java SE 7で導入されたAutoCloseableインタフェースのサブインタフェースとして定義されています。このため、Closeable型で扱えるインスタンスへの参照は、アップキャストすればAutoCloseable型でも扱うことが可能です。
また、これら2つのインタフェースで定義されているcloseメソッドには、throwsで宣言している例外クラスの種類が違うという特徴もあります。次の2つのコード例は、それぞれのインタフェースの実装例です。

例 AutoCloseableを実装したクラス

```
1. public class SampleResource implements AutoCloseable {
2.     @Override
3.     public void close() throws Exception {
4.         System.out.println("close");
5.     }
6. }
```

このクラスは、AutoCloseableインタフェースを実装しており、その唯一の抽象メソッドであるcloseメソッドは、Exception型の例外をスローします。

例 Closeableを実装したクラス

```
1. import java.io.Closeable;
2. import java.io.IOException;
3.
4. public class SampleResource implements Closeable {
5.     @Override
6.     public void close() throws IOException {
7.         System.out.println("close");
8.     }
9. }
```

一方、Closeableインタフェースを実装したクラスでは、closeメソッドのシグニチャは同じものの、スローする例外クラスがIOException型となっています。このことからも、Closeableインタフェースはあくまでも入出力ストリームを

扱うことを前提としたインタフェースであることがわかります。
もし、try-with-resourcesで扱うリソースクラスを独自で定義するのであれば、入出力ストリームに特化したCloseableではなく、AutoCloseableを実装するのが適切でしょう。

試験対策

try-with-resourcesで扱えるのは、AutoCloseableインタフェースもしくはCloseableインタフェースを実装したクラスです。AutoCloseableのcloseメソッドがスローする例外はException、Closeableのcloseメソッドがスローする例外はIOExceptionです。

6. A、C ➡P401

try-with-resourcesでは、tryキーワードの後ろのカッコ内で宣言したリソースが、自動的に閉じる対象となります。

構文

```
try ( 自動的に閉じるリソースの宣言 ) {
    リソースを使った処理
}
```

解答4で使用したテキストファイルの内容をコピーするプログラム（FileCopyクラス）を、try with resourcesを使って記述すると次のようになります。リソースが自動的に閉じるのに伴い、finallyブロックが必要なくなったことで、コードが簡潔になりました。

例 try-with-resourcesを使ったファイルコピーのプログラム

```
 1. import java.io.BufferedReader;
 2. import java.io.BufferedWriter;
 3. import java.io.File;
 4. import java.io.FileReader;
 5. import java.io.FileWriter;
 6. import java.io.IOException;
 7.
 8. public class FileCopy {
 9.     public static void main(String[] args) {
10.         try (BufferedReader in =
11.                 new BufferedReader(new FileReader(new
                        File("input.txt")));
12.                 BufferedWriter out =
13.                     new BufferedWriter(new FileWriter(new
                            File("result.txt")));) {
14.
15.             String line = null;
16.             while((line = in.readLine()) != null) {
17.                 out.append(line);
18.                 out.newLine();
19.             }
20.             out.flush();
21.         } catch (IOException e) {
22.             throw new RuntimeException(e);
23.         }
24.     }
25. }
```

なお、**複数のリソース**を対象とする場合は、このコード例の10〜13行目のように**セミコロン**「;」で区切って列挙します。対象とするリソースが1つだけの場合は、セミコロンは省略可能です（付けることもできます）。

try-with-resourcesで自動的に閉じる対象とするリソースは、tryに続くカッコだけでなく、次のようにtryブロックの前に宣言を記述することも可能です（選択肢**C**）。

例 tryブロック前でリソースを宣言

```
1. public class SampleUsing {
2.     public static void main(String[] args) throws Exception {
3.         SampleResource resource = new SampleResource();
4.         try (resource) {
5.             // do something
6.         }
7.     }
8. }
```

このときの重要なポイントは、**「リソースはfinalもしくは実質的にfinalでなくてはならない」**ということです（選択肢D）。たとえば、次のようにtryブロック内で新しいインスタンスへの参照をリソースに代入することはできず、コンパイルエラーとなります。

※次ページに続く

例 実質的なfinalでないリソース

```
1. public class SampleUsing {
2.     public static void main(String[] args) throws Exception {
3.         SampleResource resource = new SampleResource();
4.         try (resource) {
5.             resource = new SampleResource();
6.         }
7.     }
8. }
```

同様に、たとえtryブロックの外であったとしても、リソースは変更できません。次のコードでは、tryブロックのあとでリソースを変更しようとしていますが、コンパイルエラーとなります（選択肢E）。

例 実質的なfinalでないリソース（tryブロック後で宣言したリソースの変更）

```
1. public class SampleUsing {
2.     public static void main(String[] args) throws Exception {
3.         SampleResource resource = new SampleResource();
4.         try (resource) {
5.             // do something
6.         }
7.         resource = new SampleResource();
8.     }
9. }
```

次のようなコードも、参照先がない（null）と宣言した変数をtryで変更しようとしているため、「実質的finalではない」としてコンパイルエラーとなります（選択肢B）。
なお、自動的に閉じる対象の変数がnullでも、事前にnullチェックを行ってからcloseメソッドを呼び出すため、NullPointerExceptionが発生することはありません。

例 実質的なfinalでないリソース（tryブロック前で宣言したリソースの変更）

```
1. public class SampleUsing {
2.     public static void main(String[] args) throws Exception {
3.         SampleResource resource = null;
4.         try (resource) {
5.             // do something
6.         }
7.     }
8. }
```


選択肢Aは、tryブロックの宣言と同時にリソースのインスタンスを生成しています。よって、正解です。
選択肢Bは、参照先がない（null）と宣言した変数をtryで変更しようとしているため、実質的finalではないという意味でコンパイルエラーとなります。よって、誤りです。
選択肢Cは、tryブロック前で作ったリソースを、try-with-resources文のtryブロックで扱うことを宣言しているものです。よって、正解です。
選択肢Dは、try-with-resources文のtryブロックで宣言した変数の参照を変更しようとしており、実質的finalではないという意味でコンパイルエラーとなります。よって、誤りです。
選択肢Eは、try-with-resources文で宣言した変数の参照を、tryブロック後で変更しようとしているため、実質的finalではないという意味でコンパイルエラーとなります。よって、誤りです。

7. C

➡ P402

tryで宣言されたリソースが閉じる順番についての問題です。
リソースは、tryブロックの処理が終了すると、**宣言したときと逆の順番**で自動的に閉じていきます。このとき、tryで生成したインスタンスではなく、事前に生成したインスタンスであっても、tryに記述したのと逆の順番で閉じていきます。
設問では、tryにAのインスタンスを生成しています。その後、tryでは、A、B、Cの順で記述しています。そのため、tryブロックが終わると、C、B、Aの順に閉じていきます。以上のことから、選択肢Cが正解となります。

8. B

➡ P403

try-with-resourcesを使う場合でも、catchブロックやfinallyブロックを付け、例外処理を実行することができます。例外発生の有無にかかわらず、リソースはtryブロックを抜けるタイミングで必ず実行されます。
そのため、tryブロック内で例外が発生しなければ、**リソースのクローズ、finallyブロックの順**に実行されます。例外が発生すれば、**リソースのクローズ、catchブロック、finallyブロックの順**に実行されます。

次のコードでは、tryブロック内で強制的に例外を発生させています。なお、ここで利用しているリソースクラスのSampleResourceは、解答5で使っている

もので、closeメソッドは「close」とだけコンソールに表示します。

例 try-with-resources文と例外処理の例

```
1. public class SampleUsing {
2.     public static void main(String[] args) {
3.         try (SampleResource resource = new SampleResource();) {
4.             throw new Exception();
5.         } catch (Exception e) {
6.             System.out.println("catch");
7.         } finally {
8.             System.out.println("finally");
9.         }
10.     }
11. }
```

このクラスを実行すると、次のような結果がコンソールに表示され、リソースのクローズ、catchブロック、finallyブロックの順に実行されていることがわかります。以上のことから、選択肢**B**が正解となります。

例 実行結果

```
> java SampleUsing
close
catch
finally
```

9. B ➡ P403

抑制された例外に関する問題です。
tryブロック内で例外Aが発生した場合、catchブロックを処理する前にリソースのcloseメソッドが呼ばれます。このとき、リソースのcloseメソッド内でも何らかの例外Bが発生した場合、その例外Bは、tryブロック内で発生した例外Aの内部に隠されてしまい、catchブロックでキャッチすることはできません。catchブロックでキャッチできるのは、tryブロック内で発生した例外Aです。この例外Bのように隠されてしまう例外のことを「**抑制された例外**」と呼びます。

例 抑制された例外を意図的にスローする

```
1. public class SampleResource implements AutoCloseable {
2.     @Override
3.     public void close() throws Exception {
4.         System.out.println("sample");
5.         throw new RuntimeException("sample");
6.     }
7. }
```

もし、抑制された例外を扱いたい場合には、**java.lang.Throwableクラス**の**getSuppressedメソッド**を使います。このメソッドは、抑制された例外への参照をThrowable配列型で戻します。
たとえば次のコード例では、catchブロックでキャッチするのは、4行目でスローした例外です。その例外から、getSuppressedメソッドを使って、抑制された例外の配列を取り出し、コンソールに表示しています。

例 キャッチした例外から、抑制された例外への参照を取り出す

```
1. public class SampleUsing {
2.     public static void main(String[] args) {
3.         try (SampleResource resource = new SampleResource();) {
4.             throw new Exception();
5.         } catch (Exception e) {
6.             for (Throwable t : e.getSuppressed()) {
7.                 System.out.println(t);
8.             }
9.         } finally {
10.            System.out.println("finally");
11.        }
12.    }
13. }
```

例 実行結果

```
> java SampleUsing
sample
java.lang.RuntimeException: sample
finally
```

設問のコードでは、AutoCloseableを実装したTroubleResourceをtry-with-resourcesで扱っています。tryブロック内でExceptionがスローされるため、まずTroubleResourceのcloseメソッドを呼び出して閉じようとしますが、そこでもRuntime

Exceptionがスローされます。前述のとおり、例外が発生した際にリソースを閉じようとして、さらに例外が発生した場合、閉じる際に発生した例外は「抑制された例外」として扱われることになります。そのため、catchされるのはtryブロック内で発生したExceptionとなります。以上のことから、選択肢**B**が正解となります。

10. B

➡ P404

アサーションを記述するためのassert文に関する問題です。
アサーションとは、プログラムのデバッグを行う際に使う機能です。実際のソフトウェアは、数十〜数百もしくは数千といった、たくさんのクラスが連携します。ソフトウェアの規模が大きくなると、バグが発生したときに、どこまでが正常に動作していて、どこからが正常に動作しなかったのかという「問題の切り分け」ができるかどうかが、迅速にバグを解決する上で非常に重要になります。
このとき、1つひとつのメソッド単位で確認すべきポイントは、次の3つです。

1. メソッドを呼び出す側が、呼び出す前の条件を満たしていたのか
2. メソッドを呼び出される側が、呼び出される前の条件を満たしていたのか
3. メソッドが実行された結果が、期待していたとおりに動作したのか

プログラマーとしては、3番目のポイントだけに着目しがちですが、1番目と2番目の条件はクラスを連携させる上で重要なポイントになります。

なお、これらの条件は順番に、事前条件、不変条件、事後条件といい、プログラミング言語「Eiffel」を開発したバートランド・メイヤーが提唱した「Design by Contract（契約による設計。以下、DbC）」で使われている用語です。このDbCは、「呼び出す側のモジュール」と「呼び出される側のモジュール」の間の役割分担が、より厳密に守られるようにすることでソフトウェア全体の正しさを確保するための設計指針と呼べるものです。

アサーションは、DbCでいわれている3つの条件を確かめるよう、プログラムのコード内に記述します。
事前条件を確かめるアサーションは、メソッドの最初の段階で、呼び出し元が渡した引数が、すべて呼び出される側のメソッドが想定している内容であることを検証するものです。もし、事前条件への違反があれば、それは呼び出し元のメソッドに問題があることがわかります。

次の商品に関する情報を扱うItemクラスは、名前と値段を表す2つのフィールドを持っています。

例 Itemクラス

```
1. public class Item {
2.     private String name;
3.     private int price;
4.
5.     public Item(String name, int price) {
6.         super();
7.         this.name = name;
8.         this.price = price;
9.     }
10.
11.    public String getName() {
12.        return name;
13.    }
14.    public void setName(String name) {
15.        this.name = name;
16.    }
17.    public int getPrice() {
18.        return price;
19.    }
20.    public void setPrice(int price) {
21.        this.price = price;
22.    }
23. }
```

現実世界では、名前のない商品や値段がマイナスの商品は存在しませんが、String型のフィールドにはnullやブランク文字列、int型のフィールドにはマイナスの値を代入することが可能です。たとえば次のコードでは、Itemのインスタンスを生成し、値段を表すpriceフィールドの値をマイナスに変更しています。

例 実際にはあり得ない不正な商品情報を持つSampleクラス

```
1. public class Sample {
2.     public static void main(String[] args) {
3.         Item item = new Item("apple", 100);
4.         item.setPrice(-1);
5.         System.out.println(item.getName() + ":" + item.getPrice());
6.     }
7. }
```

このプログラムを実行すると、次のように不正な値段が表示されます。

例 実行結果

```
> java Sample
apple:-1
```

このような不正な値が入れられてしまうと、正しく処理ができなくなる可能性があります。そこで、setterメソッドの2つにアサーションを導入します。以下は、アサーションの構文です。

構文

```
assert 条件式;
```

アサーションの条件式がtrueを戻せば何事もなく処理は進みますが、falseを戻せば**java.lang.AssertionError**がスローされ、処理は中断します。このエラーで処理が中断したときに表示される情報を見れば、プログラムのどこで中断し、その理由は何だったのか、ある程度見当を付けられます。
エラーが起きた理由を記すために、スローされるAssertionErrorのインスタンスに詳細なメッセージを渡すことも可能です。

構文

```
assert 条件式 : メッセージ;
```

次のコードが、Itemクラスの2つのsetterメソッドに、アサーションを組み込んだ例です。

例 アサーションの使用例

```
1. public class Item {
2.     private String name;
3.     private int price;
4.
5.     public Item(String name, int price) {
6.         super();
7.         this.name = name;
8.         this.price = price;
9.     }
10.
11.     public String getName() {
12.         return name;
13.     }
14.     public int getPrice() {
15.         return price;
16.     }
17.     public void setName(String name) {
18.         // 事前条件の確認
19.         assert name != null : "name is null";
20.         this.name = name;
21.     }
22.     public void setPrice(int price) {
23.         // 事前条件の確認
24.         assert price > 0 : "invalid price : " + price;
25.         this.price = price;
26.     }
27. }
```

名前をセットするsetNameメソッドでは、引数nameの値がnullであるかを確認し、もしnullなら「name is null」というメッセージを持ったAssertionErrorがスローされます。
同様に、値段をセットするsetPriceメソッドは、引数priceの値が0よりも小さければ不正な値段であることを示すメッセージを持ったAssertionErrorがスローされます。

assert文は、条件式がtrueを戻せばエラーを出しません。選択肢Aは、paramがnullのときにエラーが出ないことになるため、nullでないことを確認するという要件を満たしません。よって、誤りです。
選択肢Bは、paramがnullでないときはエラーは起きませんが、nullのときにエラーが発生します。よって、選択肢**B**が正解です。
選択肢CやDのような構文は存在しません。

11. C

➡ P404

アサーションを有効にするjavaコマンドの-eaオプションに関する問題です。
アサーションは、デフォルトでは無効になっています。アサーションを有効にするには、**javaコマンド**の実行時に**-eaオプション**を付けます。よって、選択肢**C**が正解です。

※次ページに続く

例 javaコマンドの-eaオプション

```
java -ea Sample
Exception in thread "main" java.lang.AssertionError: invalid price : -1
    at Item.setPrice(Item.java:24)
    at Sample.main(Sample.java:4)
```

解答10で作ったSampleクラスを利用してアサーションを実行しましたが、setterメソッドに記述していたアサーションが有効になったことで、エラーが発生しました。このときのメッセージを見ると、Sampleクラスの4行目を実行する際に、Itemクラスの24行目でエラーが発生し、そのときのメッセージは「invalid price : -1」であることから、不正な値段がItemクラスのsetterメソッドの引数に渡されたことがわかります。
Itemクラスの24行目でアサーションの条件を見れば、事前条件違反であることがわかるため、呼び出し元のSampleクラスにバグがあることを把握できます。今回は2つのクラスでの連携でしたが、もっと多くのクラスでさまざまな引数をやり取りしながら連携する場合には、アサーションによる問題の切り分けは有効な手段となり得るでしょう。

事前条件以外の不変条件と事後条件での使用例も見てみましょう。次のコードは、先ほどのItemクラスのコンストラクタに不変条件をチェックするためのアサーションを導入した例です。

例 アサーションの使用例

```
1. public class Item {
2.     private String name;
3.     private int price;
4.
5.     public Item(String name, int price) {
6.         super();
7.         this.name = name;
8.         this.price = price;
9.
10.        // 不変条件の確認
11.        assert this.name != null : "name is null";
12.        assert this.price > 0 : "invalid price : " + price;
13.    }
14.
15.    // コンストラクタ以外（setter/getterメソッド）は省略
16. }
```


不変条件は、インスタンスが使用される前（メソッドが呼び出される前）に、そのインスタンスが正しく準備ができているかどうかを確認するためのものです。また、メソッドの実行後にインスタンスが正しい状態になっているかを確認するための条件でもあります。

前述のコンストラクタのアサーションは、コンストラクタでインスタンスが正しい状態（フィールドの値が正しい状態）になっていることを確認しているものです。本来はメソッドの処理後にもフィールドをチェックすべきですが、説明が冗長になるので割愛しています。

最後に事後条件の例を挙げます。**事後条件**のアサーションは、メソッドが呼び出し元に返す結果が正しいことを検証するものです。事後条件を満たしていなければ、結果を返すメソッドにバグがあることがわかります。
この例を解説するために、先ほどまでのItemクラスに加えて、その商品を注文したときの情報を扱う注文クラス（Order）と、その注文を扱うmainメソッドを持ったMainクラスを追加します。

例 Orderクラス

```
1. public class Order {
2.     private Item item;
3.     private double qty;
4.
5.     public Order(Item item, double qty) {
6.         super();
7.         this.item = item;
8.         this.qty = qty;
9.     }
10.    public int getPrice() {
11.        int result = (int) (qty * item.getPrice());
12.        return result;
13.    }
14.    public Item getItem() {
15.        return item;
16.    }
17.    public double getQty() {
18.        return qty;
19.    }
20. }
```

Orderクラスは、コンストラクタで注文したいItemへの参照と、注文数を受け

取ります。本来であれば、ここでアサーションを使って不変条件のチェックをすべきですが、説明が冗長になるので割愛します。Orderクラスの特徴は、Itemが持っている値段（つまり、単価）と数量を掛け合わせた小計額を戻すgetPriceメソッドを持っている点です。

このgetPriceメソッドを使うのが、次のMainクラスです。このクラスはOrderクラスとItemクラスの両方をテストするためのテストプログラムだと考えてください。mainメソッドでは、Itemのインスタンスを生成し、起動パラメータで受け取った値をコンストラクタに引数として渡しています。
その後、Orderのインスタンスを生成し、Itemのインスタンスへの参照（item）と個数（qty）をコンストラクタに渡しています。これで小計の計算ができるようになります。11行目のアサーションでは、OrderのgetPriceメソッドの結果が、入力された価格と数量を掛け合わせたものと一致するかを確認しています。

例 Mainクラス

```
 1. public class Main {
 2.     public static void main(String[] args) {
 3.         String name = args[0];
 4.         int price = Integer.parseInt(args[1]);
 5.         double qty = Double.parseDouble(args[2]);
 6.
 7.         Item item = new Item(name, price);
 8.         Order order = new Order(item, qty);
 9.
10.         // 事後条件の確認
11.         assert order.getPrice() == (int)(price * qty)
12.                 : "invalid result getPrice()";
13.
14.         System.out.println("tatal price : " + order.getPrice());
15.     }
16. }
```

このように、事前、不変、事後の3つの条件を、連携しているすべてのメソッドで確認することで、どこでバグが発生しているのかという「問題の切り分け」が可能になります。

10

ローカライズ

- java.util.Localクラス
- ロケール情報の取得と構築
- java.util.Propertiesクラス
- プロパティファイル
- native2ascii
- 文字コード
- java.nio.charset.Charsetクラス
- java.util.ResourceBundleクラス
- 日付、数値のフォーマット（書式化）

1. デフォルトのロケール情報を持ったLocaleクラスのインスタンスを取得したい。正しいコードを選びなさい。（1つ選択）

A. `Locale locale = System.getDefault();`
B. `Locale locale = System.locale.getDefault();`
C. `Locale locale = Locale.getDefault();`
D. `Locale locale = Locale.getInstance();`

➡ P432

2. 日本で日本語を使うロケール情報を構築するコードとして、最も適切なものを選びなさい。（1つ選択）

A. `Locale locale = new Locale("JP");`
B. `Locale locale = new Locale("ja");`
C. `Locale locale = new Locale("JP", "ja");`
D. `Locale locale = new Locale("ja", "JP");`

➡ P433

3. アメリカのロケール情報を構築するコードとして、正しいものを選びなさい。（1つ選択）

A. `Locale locale = Locale.EN;`
B. `Locale locale = Locale.US;`
C. `Locale locale = Locale.EN_US;`
D. `Locale locale = Locale.US_EN;`

➡ P435

4. 日本で日本語のロケール情報を構築するコードとして、正しいものを選びなさい。（1つ選択）

A.
```
Locale locale = new Locale.Builder()
        .setLanguage("jp")
        .setRegion("JP")
        .setScript("Japan")
        .build();
```

B.
```
Locale locale = new Builder()
        .setLanguage("jp")
        .setRegion("JP")
        .setScript("Japan")
        .build();
```

C.
```
Locale locale = new Locale().new Builder()
        .setLanguage("jp")
        .setRegion("JP")
        .setScript("Japan")
        .build();
```

D.
```
Locale locale = new Locale()
        .setLanguage("jp")
        .setRegion("JP")
        .setScript("Japan")
        .build();
```

➡ P436

5. Localeクラスのインスタンス化の方法として、正しいものを選びなさい。（1つ選択）

A. `Locale locale = Locale.getLanguageTag("en-US-x-lvariant-POSIX");`
B. `Locale locale = Locale.forLanguageTag("en-US-x-lvariant-POSIX");`
C. `Locale locale = Locale.forTag("en-US-x-lvariant-POSIX");`
D. `Locale locale = Locale.forName("en-US-x-lvariant-POSIX");`

➡ P440

6. プロパティファイルに記述するキーと値の表記として、正しいものを選びなさい。（2つ選択）

A. `key=value`
B. `key:value`
C. `value:key`
D. `value=key`
E. `key==value`

➡ P441

7. 次のコードを確認してください。

```
Properties prop = new Properties();
prop.load(new FileReader("sample.properties"));
```

変数propを使い、値の一覧を表示するコードとして、正しいものを選びなさい。（2つ選択）

A.
```
Set keys = prop.keySet();
for (Object key : keys) {
    System.out.println(prop.getProperty(key));
}
```

B.
```
Set keys = prop.keySet();
for (Object key : keys) {
    System.out.println(prop.get(key));
}
```

C.
```
prop.forEach((k, v) -> System.out.println(v));
```

D.
```
prop.forEach((v) -> System.out.println(v));
```

➡ P442

8. プロパティファイルを読み込み処理するプログラムで、デバッグ作業を実施している。プロパティファイルのキーと値を一覧表示するための方法として、最も適切なものを選びなさい。（1つ選択）

A. Propertiesクラスのkey Setメソッドでキーの一覧を取得し、getメソッドで値を1つずつ取り出して表示する
B. Propertiesクラスのforeachメソッドに、BiConsumer型の値を取り出して表示するラムダ式を渡す
C. Propertiesクラスのforeachメソッドに、Consumer型の値を取り出して表示するラムダ式を渡す
D. Propertiesクラスのlistメソッドに、PrintWriter型のオブジェクトへの参照を渡す

➡ P444

9. 次のプログラムの「/* insert code here */」に入るコードとして、正しいものを選びなさい。（1つ選択）

```
Properties prop = new Properties();
prop.load(new FileReader("sample.properties"));
String value = /* insert code here */
System.out.println(value);
```

A. `prop.getValue("key");`
B. `prop.getProperty("key");`
C. `prop.get("key");`
D. `prop.getPropertyValue("key");`

➡ P445

10. input.propertiesというファイルを読み込み、Unicode表記に変換したoutput.propertiesというプロパティファイルを出力するためのコマンドとして、正しいものを選びなさい。（1つ選択）

A. `native2ascii output.properties input.properties`
B. `native2ascii input.properties output.properties`
C. `convert2ascii output.properties input.properties`
D. `convert2ascii input.properties output.properties`

➡ P446

11. プロパティファイルを記述するときに使用できる文字コードとして、正しいものを選びなさい。（2つ選択）

A. ISO-8859-1
B. UTF-8
C. Shift-JIS
D. EUC-JP

➡ P447

12. プロパティファイル「sample.properties」を読み込むResourceBundleオブジェクトを取得するためのコードとして正しいものを選びなさい。（1つ選択）

A.
```
ResourceBundle resource =
    new ResourceBundle("sample");
```
B.
```
ResourceBundle resource =
    new ResourceBundle("sample.properties");
```
C.
```
ResourceBundle resource =
    ResourceBundle.getBundle("sample");
```
D.
```
ResourceBundle resource =
    ResourceBundle.getBundle("sample.properties");
```

➡ P448

13. デフォルトのロケール情報が「ja_JP」の環境で、プロパティファイル「sample.properties」と「sample_ja_JP.properties」が存在する場合に、次のプログラムの実行結果として正しいものを選びなさい。（1つ選択）

```
1. import java.util.ResourceBundle;
2.
3. public class Sample {
4.     public static void main(String[] args) throws Exception {
5.         ResourceBundle resource
6.             = ResourceBundle.getBundle("sample");
7.         System.out.println(resource.getString("test"));
8.     }
9. }
```

A. sample.propertiesに記述したキー「test」の値が表示される
B. sample_ja_JP.propertiesに記述したキー「test」の値が表示される
C. コンパイルエラーが発生する
D. 実行時に例外がスローされる

➡ P450

14. ロケール情報に対応するプロパティファイルが存在しなかった場合に発生する例外として、正しいものを選びなさい。（1つ選択）

A. MissingResourceException
B. NullPointerException
C. IllegalResourceException
D. PropertyNotFoundException

➡ P451

15. デフォルトのロケールが日本という環境で、英語で内容を記述した「sample_en_US.properties」というプロパティファイルを読み込みたい。最も適切な方法を選びなさい。（1つ選択）

A. ResourceBundleクラスのsetLocaleメソッドにロケール情報を渡す
B. ResourceBundleクラスのgetBundleWithLocaleメソッドの第2引数にロケール情報を渡す
C. ResourceBundleクラスのgetBundleメソッドの第2引数にロケール情報を渡す
D. ResourceBundleクラスのgetStringメソッドの第2引数にロケール情報を渡す

➡ P452

16. 「20210718」のように年、月、日を表示するためのDateTimeFormatterクラスの定数として、正しいものを選びなさい。（1つ選択）

A. BASIC_ISO_DATE
B. ISO_LOCAL_DATE
C. ISO_LOCAL_TIME
D. ISO_ORDINAL_DATE

➡ P453

17. 通貨を表す数値のフォーマットを実行したい。この場合に使用するNumberFormatクラスのメソッドとして、正しいものを選びなさい。（1つ選択）

A. getInstanceメソッド
B. getIntegerInstanceメソッド
C. getCurrencyInstanceメソッド
D. getCurrencyメソッド

➡ P457

第10章　ローカライズ
解　答

1. C

➡ P426

デフォルトのロケール情報を取得する方法についての問題です。
「ソフトウェアの時代」と呼ばれる現代では、1つのソフトウェアがさまざまな国や地域で使われることは珍しくありません。異なる国や地域で利用されるソフトウェアを設計するときには、以下のようなさまざまなことを考慮しなければいけません。

・使われている言語
・時刻（標準時との時差）
・西暦や国・宗教ごとの暦（イスラム暦、グレゴリオ暦、ユダヤ暦、和暦など）
・日付や時刻の表記
・小数点の記号や桁区切りといった数値の表記
・通貨や度量衡（長さ、大きさ、重さ）の単位など

こうした違いに対応できるようにソフトウェアを設計することを「i18n」と呼びます。「i18n」とは、国際化を表す「internationalization」という言葉の最初の「i」から最後の「n」までの間に「nternationalizatio」と18個の文字が並ぶことを表す用語[※1]です。
i18nをするときに気を付けなければいけないのが、動作しているソフトウェアをできる限り修正せずに、それぞれの国や地域に対応できるように設計することです。たとえば、そのソフトウェアが日本で使われるのであれば日本語で、アメリカであれば英語で、ドイツであればドイツ語で自動的に表示されるようにしたいのです。
Javaは、標準でi18nに対応しており、それぞれの国や地域に柔軟に対応するためのクラスやインタフェースを提供しています。このうち、最も基盤となるクラスが**java.util.Locale**です。このクラスは、地域や言語といった「**ロケール情報**」を扱うためのクラスです。
間違えやすいのが、Localeクラスは単に国を表すものではなく、地理的・政治的・文化的な特定の地域を表すものであるという点です。言語であれば、たとえば中国語であっても、簡体字と繁体字が別のものとして扱われていたり、地域も中国（本土）と香港と台湾で別のコードが割り当てられている例がわかりやすいでしょう。次のコードは、Localeクラスの使用例です。

※1　一般的にはインターナショナライゼーションという用語がよく使われます。出題トピックの「ローカライズ」にも国際化の意味があり、両者は本質的には同義です。

例 Localeクラスの使用例（国と言語の表示）

```
 1. import java.util.Locale;
 2.
 3. public class Sample {
 4.     public static void main(String[] args) {
 5.         Locale locale = Locale.getDefault();
 6.         // 国を表示する
 7.         System.out.println(locale.getCountry());
 8.         // 言語を表示する
 9.         System.out.println(locale.getLanguage());
10.     }
11. }
```

Localeクラスの**getDefaultメソッド**は、実行しているコンピュータのデフォルトのロケール情報を戻すため、このコードを実行すると、次のように実行しているコンピュータのデフォルトの地域と言語が表示されます。

例 実行結果

```
> java Sample
JP
ja
```

たとえば、次のようにjavaコマンドのオプションとして、デフォルトの国と言語を変更すると、実行結果も変わります。このコマンドのオプションでは、システムが稼働しているコンピュータのデフォルトの言語と国（日本語と日本）から、言語を英語に、国をアメリカに変更しています。

例 実行結果

```
> java -Duser.language=en -Duser.country=US Sample
US
en
```

以上のことから、選択肢**C**が正解です。

2. D

➡ P426

ロケール情報を構築する方法についての問題です。
Localeクラスのインスタンスを生成するには、解答1で解説したgetDefaultメ

ソッドを含めて、次の5つの方法があります。

1. **getDefaultメソッド**を使う（解答1を参照）
2. **コンストラクタ**を使う
3. **ロケール定数**を使う
4. **ビルダー**を使う
5. **ファクトリメソッド**を使う

2つ目のLocaleクラスのコンストラクタを使ったインスタンス化では、コンストラクタの引数として**言語コード**と**国コード**などを受け取ります。たとえば、日本を表すロケール情報では、言語コードが「ja」で、国コードが「JP」となります。
この言語コードと国コードは、言語コードはISO639、国コードはISO3166として、国際標準化機構（ISO：International Organization for Standardization）によって標準化されています。代表的な言語コードと国コードを以下に示します。

【主な言語コード（ISO639-1）】

言語	コード
日本語	ja
英語	en
ドイツ語	de
中国語	zh
フランス語	fr

【主な国コード（ISO3166-1）】

国	コード
日本	JP
アメリカ合衆国	US
ドイツ	DE
中華人民共和国	CN
フランス	FR

たとえば、カナダのように英語とフランス語の2つの言語を公用語としている国で、フランス語を表すロケール情報を次のように指定すると、フランス語を使うカナダを表すことができます。

```
Locale locale = new Locale("fr", "CA");
```


Localeクラスのコンストラクタは、引数に言語コード（language）、国コード（country）、派生情報（variant）の3つを受け取れるよう、オーバーロードされています。

```
・Locale(String language)
・Locale(String language, String country)
・Locale(String language, String country, String variant)
```

コンストラクタの第3引数に渡す**派生情報**は、ベンダーまたはブラウザに固有のコードです。たとえば、WindowsにはWIN、MacintoshにはMAC、POSIXにはPOSIXを使います。もし、この派生情報が2つある場合には、それらを**アンダースコア**「_」で区切り、重要なものを最初に指定します。たとえば、Traditional Spanish（イスパニア語）のロケール情報を使う場合には、次のコード例のように言語、国、派生情報を指定してLocaleクラスのインスタンスを生成します。

```
Locale locale = new Locale("es", "ES", "Traditional_WIN");
```

以上のことから、選択肢**D**が正解です。
引数を1つしか渡さないコンストラクタでは、言語コードを指定します。そのため、国コードを指定している選択肢Aは誤りです。言語コードのみを指定している選択肢Bは誤りではありませんが、「日本で日本語を使う」という問題の主旨から考えると、言語コードと国コードの両方を指定している選択肢Dのほうが適切です。選択肢Cは、言語コードと国コードの指定が逆になっているため誤りです。

3. B

➡ P426

ロケール情報を構築する方法についての問題です。
前問では、コンストラクタを使ったLocaleクラスのインスタンス化について学びました。
Localeクラスのコンストラクタを使用するにあたっての注意点は、渡す言語コードや国コード、派生情報は、正しい値かどうかはチェックされないことです。たとえば、次のように無効な値をコンストラクタの引数に渡しても、コンパイルエラーになったり、実行時に例外がスローされることはありません。

```
Locale locale = new Locale("aa", "22");
```

※次ページに続く

その結果、言語コードや国コードに間違いがあっても気がつかないという問題が発生する可能性があります。そこで、このような間違いが起こらないようにするため、次のような**ロケール定数**が用意されています。

【主なロケール定数】

定数	言語	国	説明
static Locale.JAPAN	ja	JP	日本のロケール情報
static Locale.US	en	US	アメリカのロケール情報
static Locale.CANADA	en	CA	カナダのロケール情報
static Locale.CANADA_FRENCH	fr	CA	カナダのロケール情報
static Locale.UK	en	GB	イギリスのロケール情報

次のコードは、フランス語を使用するカナダのロケール情報を、ロケール定数を使って取得したものです。このようにあらかじめ用意されているロケール定数を使用することで、言語コードや国コードの間違いを防ぐことができます。

```
Locale locale = Locale.CANADA_FRENCH;
```

このようにロケール定数を使用してインスタンス化すれば間違いがありませんが、ロケール定数がすべての地域や言語の組み合わせを網羅しているわけではありません。APIドキュメントを見ればわかるとおり、それほど多くのバリエーションが用意されているわけではないのです（Java SE 11では、24種類しかありません）。もし、ロケール定数が用意されていない場合には、言語コードや国コードを間違えないように注意しながらコンストラクタを使う必要があります。

設問では、「アメリカのロケール情報を構築する」とあります。ロケール定数を使う場合、アメリカのロケール情報を構築するには選択肢**B**のように記述します。その他の選択肢のようなロケール定数は存在しません。

4. A ➡ P426

解答1～3では、デフォルトのロケール情報、コンストラクタ、ロケール定数によるLocaleクラスのインスタンス化の方法について学んできました。本設問では、4つ目のインスタンス化の方法であるビルダーについて学びます。**ビルダー**とは、インスタンスを組み立てるクラスのことです。

Localeクラスは JDK 1.1で追加されましたが、Java SE 7以降、さらに詳細な情報を扱えるようにインターネット技術の標準化団体「IETF（Internet Engineering Task Force）」が定める**IETF言語タグ**に対応するようになりました。このIETF言語タグでは、細かなロケール情報を表現するために、次のような情報（下位タグ）の組み合わせで構成されており、これらを**ハイフン**「-」で区切って1つの文字列として表します。

- language（言語）
- script（文字体系）
- region（地域）
- variant（派生）
- extension（拡張）
- privateuse（私用）

たとえば、ロケール情報をIETF言語タグで表現するには、次のコード例のようにLocaleクラスのインスタンスを生成し、そのインスタンスの**toLanguageTagメソッド**を使います。

```
Locale locale = new Locale("ja", "JP", "JP");
System.out.println(locale.toLanguageTag());
```

このコードを実行すると、次のようにハイフンで区切られたIETF言語タグの書式で日本のロケール情報がコンソールに表示されます。

例 実行結果

```
ja-JP-u-ca-japanese-x-lvariant-JP
```

前述のとおり、IETF言語タグの下位タグにはいくつもの種類があります。Localeクラスのコンストラクタで受け取る引数は、言語と国、派生情報の3つですが、これだけではほかの値を表現できません。
「引数の種類に応じてコンストラクタをオーバーロードしていく」ことも可能です。しかし、このIETF言語タグの仕様は今後拡張される可能性は十分にあり、そのたびにコンストラクタを増やすことになってしまえば、プログラマーは困惑することになるでしょう。
このような将来の懸念に対応するために、新しいLocaleクラスではコンストラクタではなく、デザインパターンの1つである「**Builderパターン**」を使っています。Builderパターンは、柔軟にインスタンスの生成をするためのパターンです。次のコードが、Builderパターンを使った例です。

※次ページに続く

例 Builderパターンの使用例

```
1. public class Item {
2.     private String name;
3.     public static class Builder {
4.         private String name;
5.         public Builder setName(String name) {
6.             this.name = name;
7.             return this;
8.         }
9.         public Item build() {
10.            if (this.name == null) {
11.                throw new IllegalStateException();
12.            }
13.            Item item = new Item();
14.            item.name = this.name;
15.            return item;
16.        }
17.    }
18.    public String getName() {
19.        return name;
20.    }
21. }
```

なお、このクラスはフィールドを1つしか持っていないため、本来ならばBuilderパターンを使う必要はありませんが、解説のためにあえて使用しています。このパターンは、次のような場合に使用を検討します。

・フィールドの数が多い
・すべてのフィールドが必須ではなく、状況に応じて要／不要が変わる

このような場合、コンストラクタの引数でフィールドの初期値を受け取ろうとすると、何番目の引数に何を渡せばよいのか、不要な引数の値はどうすればよいのか、そのクラスを使おうとするプログラマーは混乱するでしょう。そこで、前述のコード例のような、必要なフィールドの値をセットするセッターメソッドを持ったインナークラス（第1章を参照）を用意します。これでコンストラクタを使った場合に比べて、どのフィールドに何をセットするかはメソッド名から見当を付けやすくなります。なお、インナークラスを利用しているのは、たとえアクセス修飾子がprivateなフィールドであっても、インナークラスからはエンクロージングクラスに自由にアクセスできるためです（Itemクラスの14行目）。

インナークラスでは、必要な値をsetterメソッドで受け取り、最後にbuildメソッドが呼び出されることで、エンクロージングクラスのインスタンスを生成し、フィールドの値をセットして、その参照を戻します。

例 Localeのインスタンスを作るクラス

```
1. public class Main {
2.     public static void main(String[] args) {
3.         Item a = new Item.Builder()
4.                     .setName("sample")
5.                     .build();
6.         System.out.println(a.getName());
7.     }
8. }
```

このBuilderパターンを使うことで、将来、フィールドが増えてもsetterメソッドを追加するだけで対応できます。もし、コンストラクタを使っていたら、コンストラクタの引数を増やすことになるため、これまでコンストラクタを使用していたクラスに影響が出てしまいます。このような将来の柔軟性を担保するためにもBuilderパターンは有効な解決策なのです。

BuilderクラスはLocaleクラスのstaticインナークラスで、Localeのインスタンス化のためのクラスです。Builderクラスを使ったLocaleクラスのインスタンス化は、次のような手順で行います。

1. Builderクラスのインスタンスを生成する
2. 各setterメソッドを使ってロケール情報を設定する
3. buildメソッドでLocaleクラスのインスタンスを生成する

次のコードが、実際にビルダーを使用したLocaleクラスのインスタンス化をしている例です。

```
Builder builder = new Locale.Builder();
builder.setLanguage("jp");
builder.setRegion("JP");
builder.setScript("Japan");

Locale locale = builder.build();
```

なお、各setterメソッドは、戻り値としてビルダー自身への参照を戻します。

そのため、ビルダーのsetterメソッドの戻り値であるビルダー自身への参照を使って、setterメソッドや**buildメソッド**といったメソッド呼び出しを次々と数珠つなぎのようにつなぐ、いわゆる「メソッドチェイン」が利用できます。次のコードは、前述の例をメソッドチェインを使ったものに置き換えた例です。こちらのほうが、よりシンプルに記述できることがわかります。

```
Locale locale = new Locale.Builder()
                .setLanguage("jp")
                .setRegion("JP")
                .setScript("Japan")
                .build();
```

Localeクラスのインナークラスである**Builder**は、staticインナークラスです。そのため、Localeクラスのインスタンスを生成しなくても、インスタンスを生成できます。よって、選択肢**A**が正解で、選択肢Cは誤りです。
選択肢Bのようにインナークラスを直接記述してインスタンス化はできません。staticインナークラスのインスタンス化には、選択肢Aのようにエンクロージングクラス名が必要です。
選択肢Cのように、引数なしのコンストラクタはLocaleクラスには用意されていません。コンストラクタを使ったインスタンス化については、解答2を参照してください。

試験対策

Locale.Builderクラスを使ってLocaleのインスタンスを生成する方法を押さえておきましょう。

5. B

➡ P427

解答4までで、Localeクラスのインスタンス生成の方法を4通り学びました。最後に**IETF言語タグそのものを組み込んだファクトリメソッド**を使う方法について説明します。ビルダーのように1つずつsetterメソッドを使って値をセットするのではなく、言語、拡張言語、文字体系、地域、方言、拡張、私用の各要素をハイフン区切りの1つの文字列として表現した言語タグを与えてインスタンスの生成と初期化を行います。本書で解説すべき範疇を超えるため詳細は割愛しますが、言語タグを組み立てるときにインターネットの標準化団体「IANA（Internet Assigned Numbers Authority)）」のサイトに登録されているサブタグ（言語タグの1つずつの要素）を参照してください。

【IANA Language Subtag Registry】
https://www.iana.org/assignments/language-subtag-registry/language-subtag-registry

試験対策として覚えておかなければいけないことは、IETF言語タグの文字列を使ってLocaleクラスのインスタンスを生成するには、次のコード例のように**Localeクラス**の**forLanguageTagメソッド**を使用するという点です。このメソッドはstaticで、引数で渡されたIETF言語タグに準拠したLocaleクラスのインスタンスを生成して、その参照を戻します。

```
Locale locale = Locale.forLanguageTag("en-US-x-lvariant-POSIX");
System.out.println(locale);
```

以上のことから、選択肢**B**が正解です。その他の選択肢のようなメソッドは存在しません。

6. A、B

➡ P427

プロパティファイルの表記に関する問題です。
ソフトウェアの国際化に対応するとき、たとえば画面に表示する文字列やエラーメッセージなどは、ユーザーが使う言語によって切り替えなければいけません。そのため、表示したい文字列やメッセージなどをプログラム中にハードコーディング（リテラルとしてそのまま記述すること）するのではなく、言語ごとに別のファイルを用意しておき、読み込むファイルをプログラムで変更して表示を切り替えるようにします。このファイルのことを「**プロパティファイル**」と呼びます。
なお、プロパティファイルは、国際化対応専用ファイルではありません。データベースへの接続IDやパスワードなど、長期間にわたりソフトウェアが使われる中で変更される可能性がある設定情報などをハードコーディングしないためにも使用します。このようにプログラムと設定情報を分け、実行時に読み込む仕組みにすれば、プロパティファイルの中身を変更するだけで、プログラムに手を加えずにソフトウェアの動作を変えることができるのです。

プロパティファイルは、「**キー=値**」または「**キー:値**」の書式で記述したテキストファイルで、その拡張子は「**.properties**」です。たとえばデータベースへの接続情報を記述したプロパティファイルは、次のような内容になります。

例 db_settings.properties

```
server_address=192.168.0.3
```

```
name=testdb
user=testuser
password=samplepassword
```

よって、選択肢**A**と**B**が正解です。

プロパティファイルはXML形式で記述することもできますが、記述量が劇的に増えることから、現在は使われていません。

7. B、C

➡ P428

プロパティファイルの読み込みと一覧表示の方法に関する問題です。
プロパティファイルをJavaのプログラムから読み込むには、次の手順で**java.util.Propertiesクラス**を使用します。

1. プロパティファイルを読み込むjava.io.FileReaderオブジェクトやjava.io.FileInputStreamオブジェクトを生成する
2. loadメソッドの引数に、1で生成したインスタンスへの参照を渡す
3. PropertiesオブジェクトのgetメソッドやgetPropertyメソッドにキーを渡して値を取得する

前問で使用したデータベースの接続設定情報を記載したプロパティファイル（db_settings.properties）を読み込んだ例が次のコードです。このコードでは、1つずつキーを指定するのではなく、すべてのキーを取り出し、繰り返し処理を行っています。

例 プロパティファイルの読み込み

```
1. import java.io.FileReader;
2. import java.io.FileInputStream;
3. import java.io.FileNotFoundException;
4. import java.io.IOException;
5. import java.util.Properties;
6. import java.util.Set;
7.
8. public class Sample {
9.     public static void main(String[] args) throws Exception {
10.         // インスタンスを生成して、プロパティファイルを読み込み
11.         Properties prop = new Properties();
12.         prop.load(new FileReader("db_settings.properties"));
13.
14.         // キーの一覧を取り出して、キーに一致する値をコンソールに表示
15.         Set keys = prop.keySet();
16.         for (Object key : keys) {
17.             System.out.println(prop.get(key));
18.         }
19.     }
20. }
```

Propertiesオブジェクトからキーを指定して値を取り出すには、**getメソッド**や**getPropertyメソッド**を使います。2つのうち、getメソッドはPropertiesクラスのスーパークラスであるjava.util.Hashtableクラスから引き継いだメソッドです。もう一方のgetPropertyメソッドは、Propertiesクラスに定義されたメソッドです。
両者の違いは、getメソッドの引数が**Object型**であるのに対し、getPropertyメソッドは**String型**である点です。この違いは、Hashtableがさまざまなオブジェクトの集合を扱うクラスであるのに対し、Propertiesクラスは文字列のキーを指定して、文字列の値を戻すというように用途が限定されている点にあります。
前述のコード例では、ジェネリクスを指定しないSetでキーの集合を扱っているため、キーはObject型として扱われます。これをString型にダウンキャストせずに使用するために、getメソッドを使っています。よって、選択肢**B**は正解です。選択肢Aは、Object型の引数をgetPropertyメソッドに渡しているため、コンパイルエラーが発生します。よって、誤りです。

なお、このコード例のようにすべての値を利用したいのであれば、次のコード例のように**forEachメソッド**にラムダ式を渡すこともできます。

例 プロパティファイルのすべての値を一覧表示

```
1. public class Sample {
2.     public static void main(String[] args) throws Exception {
3.         Properties prop = new Properties();
4.         prop.load(new FileReader("db_settings.properties"));
5.         prop.forEach((k, v) -> System.out.println(k + "=" + v));
6.     }
7. }
```

このforEachメソッドは、java.util.function.BiConsumer型の引数を受け取ります。この関数型インタフェースは、1つしか引数を受け取らないjava.util.function.Consumerを拡張し、引数を2つ受け取るようにしたものです。コード例では、キーと値をそれぞれ変数kとvで受け取り、コンソールに表示するラムダ式をforEachメソッドで繰り返し実行しています。
選択肢Cのラムダ式は、kとvの2つの引数を受け取ります。一方、選択肢Dのラムダ式は1つしか引数を受け取りません。よって、選択肢**C**が正解で、選択肢Dは誤りです。

8. D

➡ P428

プロパティファイルの読み込みと一覧表示の方法に関する問題です。
コンソールに値の一覧を表示したいだけであれば、解答7で解説した拡張for文やforEachメソッドを使う方法よりも、**Propertiesクラス**の**listメソッド**を使うほうが簡単でしょう。このメソッドは、引数にjava.io.PrintStream型やjava.io.PrintWriter型の引数を受け取り、プロパティファイルの中身を一覧表示します。
次のコード例では、コンソールに表示するために、PrintStream型であるSystemクラスの**outフィールドへの参照**をlistメソッドに渡して一覧表示させています。

例 プロパティファイルの値を一覧表示

```
1. public class Sample {
2.     public static void main(String[] args) throws Exception {
3.         Properties prop = new Properties();
4.         prop.load(new FileReader("db_settings.properties"));
5.         prop.list(System.out);
6.     }
7. }
```

このコードを実行すると、次のようにコンソールに表示されます。

例 実行結果

```
> java Sample
-- listing properties --
password=samplepassword
name=testdb
server_address=192.168.0.3
user=testuser
```


keySetメソッドでキーを取り出して拡張for文で繰り返し処理をしたり、ラムダ式を使ってforEachメソッドを使う方法に比べると、かなり少ないコード量で記述できることがわかります。デバッグ作業などで一覧を表示したいだけであれば、このlistメソッドを使う方法が便利でしょう。

以上のことから、選択肢**D**が正解です。選択肢AやBの方法でも一覧表示することは可能ですが、デバッグ作業ではなく、もっと複雑な作業をするために利用するものです。よって、最も適切な選択肢を選ぶという本設問では、これらの選択肢は誤りとなります。また、PropertiesクラスのforEachメソッドは、BiConsumer型のラムダ式を受け取ります。Consumer型のラムダ式は受け取れません。よって、選択肢Cは誤りです。

9. B

➡ P429

解答7、8で学んだ方法は、プロパティファイルのすべての値を処理したいときに使うものでした。
ソフトウェア開発の現場では、取り出したい値のキーがあらかじめわかっていることも多く、その場合は次のように**getPropertyメソッド**を使います。こちらのほうが、一般的なPropertiesクラスの使用方法です。

例 既知のキーを指定して値を取り出す

```
1. public class Sample {
2.     public static void main(String[] args) throws Exception {
3.         Properties prop = new Properties();
4.         prop.load(new FileReader("db_settings.properties"));
5.
6.         // nameというキーの値を取り出してコンソールに表示
7.         System.out.println(prop.getProperty("name"));
8.     }
9. }
```

このコードを実行すると、次のようにdb_settings.propertiesからnameというキーの値がコンソールに表示されます。

例 実行結果

```
> java Sample
testdb
```

以上のことから、選択肢**B**が正解です。

getPropertyメソッドのほかにも、Propertiesクラスには値を取り出すためのメソッドとして、**getメソッド**もあります。このメソッドは、Propertiesクラスのスーパークラスである**Hashtable**クラスから引き継いだメソッドで、Object型の戻り値を戻します。設問では、戻り値をString型の変数で受け取っているため、このメソッドを使うとダウンキャストが必要になり、コンパイルエラーとなります。よって、選択肢Cは誤りです。
選択肢AやDのようなメソッドはありません。

10. B ➡ P429

プロパティファイルへの変換を行うnative2asciiに関する問題です。
java.util.Propertiesクラスを使うと、プロパティファイルに記述した文字列は、文字コードがISO-8859-1として読み込まれます。この文字コードは、ラテンアルファベット文字や一部の記号を合わせて計191字を規定したもので、日本語は扱えません。
プロパティファイルで日本語を扱いたい場合には、日本語からUnicode表記に変換する**native2ascii**というJavaの標準ツールを使います。たとえば、日本語の値を含む次のようなプロパティファイルの場合、いったんテキストファイル（拡張子は.txt）として保存し、このツールでUnicode表記に変換します。

例 test.txt

```
hello=こんにちは
```

このテキストファイルをプロパティファイルに変換するために、native2asciiを使います。次のコマンドは、test.txtを読み込んで、日本語の文字列をUnicode表記に変換してからtest.propertiesとしてファイルに出力する例です。

例 native2asciiコマンドでプロパティファイルに変換

```
> native2ascii test.txt test.properties
```

このコマンドを実行すると、test.propertiesという名前のファイルが出力されます。ファイルを開くと、日本語で記述していた箇所が、¥uの後ろに16進数4桁の数字のUnicode表記に変換されていることがわかります。これでは何が書いてあるのか人間には理解できませんが、プログラムから読み込めば、日本語として扱われます。

例 test.properties

```
hello=¥u3053¥u3093¥u306B¥u3061¥u306F
```


以上のことから、選択肢**B**が正解です。選択肢CやDのようなコマンドは存在しません。

11. A、B ➡ P429

プロパティファイルの文字コードとCharsetクラスに関する問題です。
Java SE 9から、ISO-8859-1のほかに、**UTF-8**で保存されたプロパティファイルも扱えるようになりました。UTF-8では、日本語をはじめとする多種多様な言語を扱うことができるので、native2asciiコマンドでUnicode形式に変換する必要がありません。たとえば、テキストエディタなどを使い、次のような日本語を含むプロパティファイルをUTF 8で保存します。

例 sample.properties（UTF-8で保存）

```
test=こんにちは
```

UTF 8で記述されたプロパティファイルをPropertiesオブジェクトで読み込むには、次のコード例のように文字コードを扱うための**java.nio.charset.Charsetクラス**を使います。

例 UTF-8のプロパティファイルをPropertyオブジェクトで読み込み（Charsetクラスを使用）

```
1. import java.io.FileReader;
2. import java.nio.charset.Charset;
3. import java.util.Properties;
4.
5. public class Sample {
6.     public static void main(String[] args) throws Exception {
7.         Properties prop = new Properties();
8.         prop.load(new FileReader(
9.                     "sample.properties",
10.                    Charset.forName("UTF-8")));
11.        System.out.println(prop.getProperty("test"));
12.    }
13. }
```

特定の文字セットを扱うCharsetのインスタンスを取得するには、コード例のように**forNameメソッド**に文字コード名を渡します。そのインスタンスへの参照を**FileReaderクラスのコンストラクタ**の第2引数に渡すと、指定した文字コードでプロパティファイルが読み込まれます。

ほかにも、次のコード例のように**java.io.InputStreamReaderクラス**のコンストラクタの第2引数で文字コードを指定する方法もあります。

例 UTF-8のプロパティファイルをPropertyオブジェクトで読み込み（InputStreamReaderクラスを使用）

```
 1. import java.io.FileInputStream;
 2. import java.io.InputStreamReader;
 3. import java.util.Properties;
 4.
 5. public class Sample {
 6.     public static void main(String[] args) throws Exception {
 7.         Properties prop = new Properties();
 8.         prop.load(new InputStreamReader(
 9.                     new FileInputStream("sample.properties"),
10.                     "utf-8"
11.                 ));
12.         System.out.println(prop.getProperty("test"));
13.     }
14. }
```

いずれのコード例も、実行すると次のように日本語が文字化けせずに表示されます。

例 実行結果

```
> java Sample
こんにちは
```

以上のことから、選択肢**A**と**B**が正解となります。
なお、java.util.ResourceBundleクラス（解答12を参照）を使えば、デフォルトでUTF-8にも対応します。

12. C

➡ P430

ResourceBundleクラスに関する問題です。
プロパティファイルを扱うには、java.util.Propertiesクラスのほかに、**java.util.ResourceBundleクラス**を使う方法もあります。これまでに学習してきたPropertiesクラスとの違いは、Localeを扱える点です。さらに、デフォルトでUTF-8で記述されたプロパティファイルを扱うこともできます。

まずは、基本的な使い方を見てみましょう。次のコード例は、前問の解説で使っていたsample.propertiesを読み込み、値を表示するものです。

例 プロパティファイルを読み込み、値を表示

```
1. import java.util.ResourceBundle;
2.
3. public class Sample {
4.     public static void main(String[] args) throws Exception {
5.         ResourceBundle resource
6.             = ResourceBundle.getBundle("sample");
7.         System.out.println(resource.getString("test"));
8.     }
9. }
```

このプログラムを実行すると次のような結果がコンソールに表示され、プロパティファイルに記述した値が表示されていることがわかります。

例 実行結果

```
> java Sample
こんにちは
```

java.util.ResourceBundleクラスは抽象クラスです。そのため、newでインスタンスを生成することはできません。よって、選択肢AとBは誤りです。インスタンスを生成するには、staticメソッドの**getBundleメソッド**を使います。プロパティファイルを読み込む場合には、ResourceBundleを継承した**java.util.PropertyResourceBundleクラス**のインスタンスが生成され、その参照が戻されます。
getBundleメソッドの引数には、プロパティファイルのファイル名から拡張子を除いたものを指定します（「sample.properties」であれば「sample」）。なお、この名前のことを「**基底名**」と呼びます。よって、選択肢**C**が正解です。
プロパティファイルからキーを指定して値を取り出すには、**getStringメソッド**を使います。このメソッドは、引数にキーとなる文字列を受け取り、値を文字列として戻します。

ResourceBundleのインスタンスは、getBundleメソッドの引数にプロパティファイルの基底名を指定して取得します。

13. B

➡ P430

ResourceBundleクラスでプロパティファイルを切り替える方法についての問題です。

ResourceBundleクラスには、クライアントのロケール情報に合わせて読み込むプロパティファイルを自動で切り替える機能が備わっています。たとえば、利用者が日本語を使ってWebサイトにアクセスしたときには、メッセージやボタンのラベルを日本語で表示するために日本語用のプロパティファイルを読み込み、英語を使う利用者なら英語用のプロパティファイルを読み込むといったことができます。

プロパティファイルを切り替えるには、同じ**基底名**の後ろに**言語コード**や**国コード**を**アンダースコア**「_」でつないだ名前のプロパティファイルをロケールごとに用意します。たとえば、日本であれば「sample_ja_JP.properties」、アメリカであれば「sample_en_US.properties」となります。

実際に、プロパティファイルを変更してみましょう。まずは、解答11で作成した「sample.properties」のファイル名を「sample_ja_JP.properties」に変更します。これで、このプロパティファイルは、日本で日本語を使う利用者用になりました。使用しているコンピュータの場所が日本で、使用言語が日本語の場合、デフォルトのロケール情報は日本ですので、前問の解説で使ったSampleクラスを実行すると、同じようにコンソールに表示されます。よって、選択肢**B**が正解です。

例 実行結果

```
> java Sample
こんにちは
```

次に、英語用のプロパティファイルを作ってみましょう。基底名は日本語用プロパティファイルと同じ「sample」で、アンダースコアでつなぐ言語コードと国コードを、それぞれenとUSに変更します。

例 sample_en_US.properties

```
test=welcome!
```

英語用のプロパティファイルが用意できたら、次のようにjavaコマンドの引数を変更してSampleクラスを実行してみましょう。今度は自動的に英語用のプロパティファイルが読み込まれ、コンソールに「welcome!」と表示されます。

例 実行結果

```
> java -Duser.language=en -Duser.country=US Sample
welcome!
```

Sampleクラスをまったく変更しませんでしたが、読み込まれるプロパティファイルが自動的に変更されたことに注目してください。このように、クライアントのロケール情報によって自動的に国際化に対応できるようにするのが、ResourceBundleクラスを使う理由です。

なお、「sample.properties」のように言語コードや国コードを付けなければ、ロケールに一致するプロパティファイルがなかったときの**デフォルトのプロパティファイル**として扱われます。世界中で利用されるソフトウェアを開発する場合は、デフォルトのプロパティファイルを英語用としておき、対応するプロパティファイルがなかったときでも世界共通語として使われている英語で表示されるようにしておくことで運用しやすくなるでしょう。

14. A

➡ P431

ロケール情報に対応するプロパティファイルが存在しない場合、**java.util.MissingResourceException**がスローされます。よって、選択肢**A**が正解です。

たとえば、解答13で作成した日本語用と英語用のプロパティファイルが存在する環境でSampleクラスを実行する場合、javaコマンドのオプションとしてデフォルトの言語をフランス語、国をカナダに変更すると、次のようにMissingResourceExceptionがスローされます。

例 実行結果

```
> java -Duser.language=fr -Duser.country=CA Sample
Exception in thread "main" java.util.MissingResourceException: Ca
n't find bundle for base name sample, locale fr_CA
    at java.base/java.util.ResourceBundle.throwMissingResourceEx
ception(ResourceBundle.java:2055)
    at java.base/java.util.ResourceBundle.getBundleImpl(ResourceB
undle.java:1689)
… 省略 …
    at Sample.main(Sample.java:7)
```

現在、存在するのは「sample_ja_JP.properties」と「sample_en_US.properties」であり、カナダのフランス語「fr_CA」に対応するプロパティファイルは存在しません。そのため、MissingResourceExceptionがスローされるのです。

このように、ロケール情報に対応するプロパティファイルがない場合、ロケー

ル情報を指定しないプロパティファイルを用意します（ここでは、「sample.properties」とします）。

例 sample.properties

```
test=hello
```

このsample.propertiesをデフォルトのプロパティファイルとして用意しておき、前述のコード例と同様にSampleクラスを実行してみましょう。コンソールには、次のように表示されるはずです。

例 実行結果

```
> java -Duser.language=fr -Duser.country=CA Sample
hello
```

15. C

➡ P431

これまでの解説では、javaコマンドのオプションを指定してデフォルトのロケール情報を切り替える方法で、ResourceBundleクラスを使っていました。この方法では、ロケール情報を切り替えるたびに**JVMを再起動**する必要があります。
任意のタイミングでロケール情報を切り替えるには、**getBundleメソッド**の第2引数にロケール情報を渡します。次のコード例では、日本用とアメリカ用のResourceBundleのインスタンスへの参照をそれぞれ取得しています。

例 getBundleメソッドの使用例

```
1. import java.util.Locale;
2. import java.util.ResourceBundle;
3.
4. public class Sample {
5.     public static void main(String[] args) throws Exception {
6.         ResourceBundle jp =
7.             ResourceBundle.getBundle("sample", Locale.JAPAN);
8.         ResourceBundle us =
9.             ResourceBundle.getBundle("sample", Locale.US);
10.        System.out.println(jp.getString("test"));
11.        System.out.println(us.getString("test"));
12.    }
13. }
```


どちらも読み込むプロパティファイルの基底名（sample）は同じで、第2引数に渡しているロケール定数が異なるだけです。このコードを実行すれば、次のようにそれぞれのロケール情報に対応するプロパティファイルが読み込まれていることがわかります。以上のことから、選択肢**C**が正解です。

例 実行結果

```
> java Sample
こんにちは
welcome!
```

16. A

➡ P431

日付のフォーマット（書式化）に関する問題です。
Java SE 8では、日付や時刻を扱うjava.time.LocalDateクラスやjava.time.LocalDateTimeクラスと共に、日付のフォーマットのためにこれまで利用してきたjava.text.DateFormatクラスに代わって、**java.time.format.DateTimeFormatterクラス**が導入されました。
このクラスのインスタンスを生成するには、次の3つの方法があります。

1. 事前に定義された定数を使用する
2. パターン文字を使用する
3. ローカライズされたスタイルを使用する

日付のフォーマットには、ISO8601という国際規格があります。この規格に則って日付・時刻のフォーマットを行うには、**DateTimeFormatterクラスにあらかじめ定義されている定数**を利用します。この定数は、DateTimeFormatter型のstaticな定数で、ISOに定義されているさまざまなフォーマットが用意されています。

【DateTimeFormatterの定数（一部抜粋）】

定数	意味	使用例
BASIC_ISO_DATE	基本的なISO日付書式	20111203
ISO_LOCAL_DATE	ローカルのISO日付書式	2011-12-03
ISO_LOCAL_TIME	ローカルのISO時刻書式	10:15:30
ISO_ORDINAL_DATE	年および年の日付の書式	2012-337

以上のことから、選択肢**A**が正解です。
これらの定数は、次のコード例のように使用できます。この例では、**ISO_DATE**という定数を使って、static領域にあるDateTimeFormatterのインスタン

スへの参照を取得して、現在日付を持ったLocalDateのインスタンスを渡してフォーマットし、コンソールに表示しています。

```
DateTimeFormatter isoFormat = DateTimeFormatter.ISO_DATE;
System.out.println(isoFormat.format(LocalDate.now()));
```

DateTimeFormatterクラスの**formatメソッド**は、LocalDateやLocalDateTimeなどのインスタンスへの参照を受け取り、書式を適用した文字列に変換します。そのため、このコードを実行すると、次のように日付がハイフン区切りで表示されます。

例 実行結果

```
2021-07-18
```

DateTimeFormatterクラスは、定数を利用するほかに、**パターン文字**を使って書式を自由に設定した上でインスタンスを生成することができます。パターン文字は、小文字のa～zと大文字のA～Zのアルファベットを使って、「日付表記の方法に関する条件（ルール）」を表したものです。たとえば、「**yyyy**」は4桁の年を、「**MM**」は2桁の月を、「**dd**」は2桁の日を表します。これらと区切り文字を組み合わせると「yyyy-MM-dd」というハイフン区切りの日付書式を表現することができます。
パターン文字を使ってDateTimeFormatterのインスタンスへの参照を取得するには、次のコード例のようにstaticメソッドである**ofPatternメソッド**を使います。

```
DateTimeFormatter isoFormat =
        DateTimeFormatter.ofPattern("yyyy年MM月dd日");
System.out.println(isoFormat.format(LocalDate.now()));
```

この例では、「yyyy年MM月dd日」という日本での日付書式を指定しているため、実行すると次のようにコンソールに表示されます。

例 実行結果

```
2021年07月18日
```

なお、どのようなパターン文字が使えるかについては、DateTimeFormatterクラスのAPIドキュメント※2を参照してください。

DateTimeFormatterクラスのインスタンスを生成する最後の方法は、**ローカライズされたスタイル**を使うことです。フォーマットの指定には、列挙型の**java.time.format.FormatStyle**を使います。この列挙型には、次の4つの列挙定数が用意されています。

【FormatStyleの列挙定数】

定数	説明
FULL	最も詳細なフルテキスト・スタイル
LONG	詳細なテキスト・スタイル
MEDIUM	やや詳細なテキスト・スタイル
SHORT	短いテキスト・スタイル

これらのスタイルを利用してDateTimeFormatのインスタンスへの参照を取得するには、DateTimeFormatterクラスのstaticメソッドである**ofLocalizedDateメソッド**（日付）や**ofLocalizedTimeメソッド**（時刻）、**ofLocalizedDateTimeメソッド**（日付と時刻）などを使います。
次のコードは、4つの書式の違いを見るための例です。

例 FormatStyleの列挙定数の使用例

```
1. import java.time.LocalDate;
2. import java.time.format.DateTimeFormatter;
3. import java.time.format.FormatStyle;
4. import java.util.Locale;
5.
6. public class FormatSample {
7.     public static void main(String[] args) {
8.
9.         Locale.setDefault(Locale.JAPAN);
10.
11.         DateTimeFormatter a =
12.                 DateTimeFormatter.ofLocalizedDate(
13.                         FormatStyle.FULL);
14.         DateTimeFormatter b =
15.                 DateTimeFormatter.ofLocalizedDate(
16.                         FormatStyle.LONG);
```

※2 https://docs.oracle.com/javase/jp/11/docs/api/java.base/java/time/format/DateTimeFormatter.html

```
17.         DateTimeFormatter c =
18.                 DateTimeFormatter.ofLocalizedDate(
19.                         FormatStyle.MEDIUM);
20.         DateTimeFormatter d =
21.                 DateTimeFormatter.ofLocalizedDate(
22.                         FormatStyle.SHORT);
24.
25.         LocalDate date = LocalDate.now();
26.         System.out.println(a.format(date));
27.         System.out.println(b.format(date));
28.         System.out.println(c.format(date));
29.         System.out.println(d.format(date));
30.
31.     }
32. }
```

このコードを実行すると、次のように表示されます。

例 実行結果

```
> java FormatSample
2021年7月18日日曜日
2021年7月18日
2021/07/18
2021/07/18
```

日本のロケールだと、このようにMEDIUMとSHORTに違いはありませんが、前述のコードの9行目で、LocaleのsetDefaultメソッドを次のようにアメリカに変更すると、異なる結果が表示されることがわかります。

```
9. Locale.setDefault(Locale.US);
```

例 実行結果

```
> java FormatSample
Sunday, July 18, 2021
July 18, 2021
Jul 18, 2021
7/18/21
```

試験対策

DateTimeFormatterクラスの定数とformatメソッドを使った日付のフォーマット、FormatStyleの列挙定数による書式の違い、パターン文字の使い方を押さえておきましょう。

17. C

➡ P431

数値のフォーマット（書式化）に関する問題です。
数値のフォーマットには、**java.text.NumberFormatクラス**を利用します。このクラスは抽象クラスであり、newを使ってインスタンス化はできません。インスタンスを生成するには、staticなファクトリメソッドを使用します。
通常の数値フォーマットであれば**getInstanceメソッド**を、整数型数値フォーマットであれば**getIntegerInstanceメソッド**、通貨フォーマットは**getCurrencyInstanceメソッド**、そしてパーセントを表記する数値フォーマットであれば**getPercentInstanceメソッド**を使います。よって、選択肢**C**が正解です。
なお、これらのメソッドはオーバーロードされており、引数なしのものとロケール情報を受け取るものの2つが用意されています。引数なしのメソッドを使えば、デフォルトのロケールが使われます。もし、デフォルトのロケールと異なるロケール情報を使いたい場合には、そのロケール情報を表すLocaleのインスタンスへの参照を引数に渡します。

次のコード例が、NumberFormatクラスの使用例です。数値フォーマットをするのが変数a、整数型数値フォーマットをするのが変数b、通貨フォーマットをするのが変数c、パーセントのフォーマットをするのが変数dです。

例 NumberFormatクラスの使用例

```
1. import java.text.NumberFormat;
2. import java.util.Locale;
3.
4. public class NumberFormatSample {
5.     public static void main(String[] args) {
6.         Locale.setDefault(Locale.JAPAN);
7.
8.         NumberFormat a = NumberFormat.getInstance();
9.         NumberFormat b = NumberFormat.getIntegerInstance();
10.        NumberFormat c = NumberFormat.getCurrencyInstance();
11.        NumberFormat d = NumberFormat.getPercentInstance();
12.
```

```
13.         System.out.println(a.format(1000.1));
14.         System.out.println(b.format(1000.1));
15.         System.out.println(c.format(1000.1));
16.         System.out.println(d.format(0.1));
17.     }
18. }
```

このコードを実行すると、次のような結果がコンソールに表示されます。数値フォーマットでは3桁区切りで表示され、整数型数値フォーマットでは3桁区切りですが、小数点以下の値は表示されません。通貨フォーマットでは3桁区切りに加えて通貨記号が表示され、パーセントのフォーマットでは0.1が10%と表示されています。

例 実行結果

```
> java NumberFormatSample
1,000.1
1,000
¥1,000
10%
```

なお、このコードはデフォルトのロケールを日本に設定していますが、次のように通貨フォーマットだけをアメリカに変更することも可能です。

```
10. NumberFormat c = NumberFormat.getCurrencyInstance(Locale.US);
```

変更した結果を実行すると、次のように通貨記号が変化し、かつ小数点以下も2桁まで表示されていることがわかります。

例 実行結果

```
> java NumberFormatSample
1,000.1
1,000
$1,000.10
10%
```


11

モジュール・システム

- モジュールの概要
- リフレクション
- モジュール定義ファイルとモジュール・ディレクティブ
- 名前付きモジュール、自動モジュール、無名モジュール
- マニフェストファイル
- ボトムアップ移行とトップダウン移行
- モジュールパス
- java.util.ServiceLoaderクラスとSPI
- 依存関係
- jdeps

1. モジュール・システムに関する説明として、誤っているものを選びなさい。（1つ選択）

A. 最適化されたモジュール構成によってパフォーマンスが向上する
B. モジュール間の依存関係を明確化できる
C. 柔軟な実行環境を実現できる
D. クラスのデータ隠蔽を強化できる
E. プラットフォーム間の整合性が向上する
F. 公開されたクラスにしかアクセスできないためセキュリティが高まる

➡ P463

2. モジュール定義ファイルに定義するモジュール・ディレクティブのうち、リフレクションに関するものを選びなさい。（1つ選択）

A. requires
B. exports
C. provides
D. uses
E. opens

➡ P465

3. モジュール・システムに関する説明として、誤っているものを選びなさい。（1つ選択）

A. 名前付きモジュールから自動モジュールにはアクセスできる
B. 無名モジュールから名前付きモジュールにアクセスするには、コンパイルや実行時に--add-modulesオプションを追加する
C. 名前付きモジュールから無名モジュールにはアクセスできる
D. 自動モジュールから無名モジュールにはアクセスできる

➡ P469

4. モジュールの移行アプローチである「ボトムアップ移行」の説明として、正しいものを選びなさい。（1つ選択）

A. 最も依存されているモジュールから順番に、自動モジュールに変換していく
B. 最も依存されているモジュールから順番に、名前付きモジュールに変換していく
C. 依存しているモジュールから順番に、自動モジュールに変換していく
D. 依存しているモジュールから順番に、名前付きモジュールに変換していく

➡ P472

5. モジュールの移行アプローチである「トップダウン移行」の説明として、正しいものを選びなさい。（2つ選択）

A. 最も依存されているモジュールから順番に、自動モジュールに変換していく
B. 最も依存されているモジュールから順番に、名前付きモジュールに変換していく
C. 依存しているモジュールから順番に、自動モジュールに変換していく
D. 依存しているモジュールから順番に、名前付きモジュールに変換していく
E. 依存されているモジュールは、いったん自動モジュールに変換しておく
F. 依存されているモジュールは、いったん無名モジュールに変換しておく

➡ P474

6. モジュールを利用したコマンドとして、正しいものを選びなさい。なお、AppモジュールがLibモジュールを利用し、モジュール定義ファイルは正しく定義されているものとする。（1つ選択）

A. `javac --module modules src/Main.java src/module-info.java`
B. `javac --module modules/lib.jar src/Main.java src/module-info.java`
C. `java --module modules App/Main`
D. `java --module-path modules -module App/Main`

➡ P476

7. **java.util.ServiceLoaderに関する説明として、誤っているものを選びなさい。（1つ選択）**

A. SPIを実現するためのクラスである
B. META-INF/services内のファイルを検索する
C. クラスファイルをロードするためのクラスである
D. loadメソッドにはclassリテラルを渡す

➡ P483

8. **SPIをモジュールで実現するために、公開するインタフェースをモジュール定義ファイルで宣言するのに使用するディレクティブとして、正しいものを選びなさい。（1つ選択）**

A. uses
B. provides
C. opens
D. services

➡ P488

9. **jdepsコマンドのオプションとして、JDKの内部APIでクラスレベルの依存関係を検索するためのものを選びなさい。（1つ選択）**

A. -jdkinternals
B. -verbose:class
C. -apionly
D. -profile

➡ P491

第11章　モジュール・システム
解　答

1. D

➡ P460

Java SE 9から導入された**モジュールは、パッケージを分類する仕組み**です。モジュールを利用することで、次のようなメリットがあります。

1. 公開されたクラスにしかアクセスできないためセキュリティが高まる
2. モジュール間の依存関係を明確化できる
3. プラットフォーム間の整合性が向上する
4. 柔軟な実行環境を実現できる
5. 最適化されたモジュール構成によってパフォーマンスが向上する

複数のパッケージをまとめたモジュールは、公開設定したパッケージ以外を非公開のパッケージとして扱います。クラス単位ではアクセス修飾子を変更することで、異なるパッケージからのアクセスを制御できましたが、モジュール・システムが導入されたことで、**パッケージ単位のアクセス制御**が可能になりました。このモジュール・システムとクラスへのアクセス修飾子を併用することで、厳密に公開できるクラスを限定することができます。その結果、クラスへの想定外のアクセスによってもたらされる意図しない結果を防ぐことができ、攻撃者がアクセスできる範囲を限定できるためセキュリティが向上します。

また、「**リフレクション**」と呼ばれる、クラスの定義情報を扱う技術もモジュール・システムによって制限されます。この技術を使えば、具体的なクラスのAPIを理解していなくても、動的にそのクラスがどのようなメソッドやフィールドを持っているかを調べて実行することができます。この機能は非常に強力な反面、攻撃者にとってはアプリケーション内にどのようなクラスがあるかを知らなくても利用できるため、セキュリティ上の問題がありました。モジュール・システムの導入によって、このような問題も解消されることになります。以上のことから、選択肢Fは正しい説明です。

モジュール・システムが導入される以前のクラス間の関係は、コードを見なければどのパッケージがどのパッケージを利用しているかはわかりませんでした。モジュールは、公開するパッケージや利用するモジュールに関する設定をモジュール設定ファイル「**module-info.java**」に定義するため、コードを見なくてもこの設定ファイルでパッケージ間の関係を理解することができます。このような**モジュール間の依存関係**が明確になることで、変更の影響

がどこまで及ぶのかがわかりやすくなり、変更に強い設計を実現することができます。以上のことから、選択肢Bは正しい説明です。

標準クラスライブラリのすべてのクラスが、プログラマーに利用されることを前提としているわけではありません。たとえば、sun.miscパッケージのように、標準クラスライブラリ内のクラスからの利用を前提として用意されているクラスも存在するからです。モジュール・システムが導入される以前であれば、これらのクラスもプログラマーが作るアプリケーションから利用できました。事実、標準クラスライブラリに含まれるsun.misc.Unsafeクラスなどは、パフォーマンス向上を求めて最大限のチューニングをするプログラマーによって利用される事例が多くありました。たとえば、このクラスを利用することで、次のようなことができるのです。

・CPUなどのハードウェア機能に直接アクセスする
・コンストラクタを実行せずにオブジェクトを作成する
・通常の検証を経ずに匿名クラスを作成する
・メモリを手動で管理する

このクラスは非常に強力ですが、このようなプラットフォームに強く依存した機能が使われると、異なるハードウェアに移植しづらくなったり、少しの修正でメモリ管理が破綻したりする問題が起きます。そこにJava SE 9からモジュール・システムが導入されたことで、本来、利用してほしくない**パッケージを隠蔽**することができるようになり、プラットフォーム間の違い（整合性）を意識する必要性が減りました。よって、選択肢Eは正しい説明です。

Javaのプログラムを実行するには、JRE（Java Runtime Environment）が必要です。JREには標準クラスライブラリが組み込まれていますが、従来であれば必要かどうかにかかわらず、すべてのライブラリを配布しなければいけませんでした。その結果、アプリケーション全体のサイズが大きくなってしまい、貧弱なネットワーク環境などの理由で配布が難しいという問題も起きました。モジュール・システムが導入されたことで、**JREに組み込むモジュールを変更する**ことができるようになりました。たとえば、GUI（Graphical User Interface）を使わないアプリケーションであれば、GUIに関するライブラリを除いた専用のJREを作り、それをアプリケーションと一緒に配布することができます。よって、選択肢Cは正しい説明です。

最後に挙げるモジュール・システムの特徴は**パフォーマンスの向上**です。前述のとおり、不要なモジュールが削られ、最適化されたJREは、アプリケーションのダウンロードやJVMの起動、アプリケーション機能の実行も軽量化します。よって、選択肢Aは正しい説明です。

モジュール・システムが強化するのは、アプリケーションがどのようなパッケージやクラスで成り立っているのかを隠蔽する「情報隠蔽」です。クラスのフィールドを隠蔽することでカプセル化を維持する「データ隠蔽」は関係しません。よって、選択肢Dの説明は誤りです。誤った説明を選択する問題なので、選択肢**D**が正解です。

2. E

➡ P460

モジュール定義ファイルに関する問題です。
モジュールの定義情報を記述するのが、**モジュール定義ファイル**です。**module-info.java**という名前のファイルを作り、module宣言を行います。

構文

```
module モジュール名 {
    モジュール・ディレクティブ
}
```

モジュールの定義情報には、**モジュール・ディレクティブ**を使います。モジュール・ディレクティブには、requiresやexports、provides、with、uses、opensなどがあります。

公開したいパッケージを指定するのが、**exportsディレクティブ**です。モジュール内には複数のパッケージが含まれますが、このディレクティブで指定されなかったパッケージについては、ほかのモジュールからはアクセスできない非公開のパッケージとして扱われます。たとえば、次のモジュール定義ファイルはcom.sample.Testモジュールの定義をしており、com.sample.serviceパッケージのみを公開しているという意味になります。

例 module-info.java（exportsの使用例）

```
1. module com.sample.Test {
2.     exports com.sample.service;
3. }
```

利用する別のモジュールを指定するのが、**requiresディレクティブ**です。このディレクティブで指定することで、指定したモジュールのモジュール定義ファイルでexportsされたパッケージのpublicなクラスを利用することができます。たとえrequiresで指定したモジュールであっても、publicでないクラスやexportsされていないパッケージにはアクセスできません。次のモジュール定義ファイルでは、com.sample.Utilモジュールを利用するための定義がされています。

例 module-info.java（requiresの使用例）

```
1. module com.sample.Test {
2.     exports com.sample.service;
3.     requires com.sample.Util;
4. }
```

なお、java.langなど、Javaでプログラミングするにあたっての基本的なパッケージをまとめた**java.baseモジュールは自動的にrequiresされる**ため、記述を省略することができます。

providesや**with**、**use**といったモジュール・ディレクティブは、SPI（Service Provider Interface）のために用意されているディレクティブです。**SPI**とは、あるプログラムやライブラリに対して、第三者が実装を提供する仕組みです。たとえば、第6章で扱ったJDBCは、インタフェースだけを定めて、その実装は各データベースベンダーが提供していました。この仕組みがSPIです。

SPIを使えば、インタフェースだけをモジュールに定義しておき、そのインタフェースを実装したクラスは別のモジュールで提供することができます。この仕組みを利用することで、変更に強いシステム構造を実現できるようになります。なお、SPIの仕組みの詳細については、java.util.ServiceLoaderについての知識が必要になります。詳細は、解答7で解説します。

リフレクションとは、クラスの定義情報を調べることで、インスタンスが具体的に何型かを知らなくても動作させることができる技術です。この技術を応用すれば、数百〜数千もある既存のクラスに何も手を入れずに機能を追加したり、変更したりすることも可能です。実際、多くのフレームワークやライブラリでも活用されています。本書では、リフレクションの概要だけにとどめ、モジュール・システムとの関係性についてのみ解説します。

まず、対象となるクラスを用意します。このItemクラスは、2つのフィールドを持ち、フィールドにアクセスするgetterメソッドを持っています。

例 Itemクラス

```
1. public class Item {
2.     private String name;
3.     private int price;
4.     public Item(String name, int price) {
5.         super();
6.         this.name = name;
7.         this.price = price;
8.     }
9.     public String getName() {
10.        return name;
11.    }
12.    public int getPrice() {
13.        return price;
14.    }
15. }
```

次に、何型のインスタンスが動いているかを隠すためのFactoryクラスを用意します。このクラスでは、Itemのインスタンスを生成し、それをjava.lang.Object型で戻します。そのため、このクラスを使ってインスタンスを取得すると、実際に動作しているインスタンスの型はわからない状態になります。このクラスでは、Itemクラスのコンストラクタを使って、appleと100という値でItemのフィールドを初期化しています。

例 Factoryクラス

```
1. public class Factory {
2.     public static Object create() {
3.         return new Item("apple", 100);
4.     }
5. }
```

次のSampleクラスが、リフレクションを使ったコード例です。このコードでは、Objectクラスが持っている**getClassメソッド**を使って、クラスの定義情報を取り出しています。クラスの定義情報を扱うためのクラスが**java.lang.Classクラス**です。

例 リフレクションの使用例（Sampleクラス）

```
1. import java.lang.reflect.Method;
2.
3. public class Sample {
4.     public static void main(String[] args) throws Exception {
5.         Object obj = Factory.create();
6.         // インスタンスから元になったクラスの定義情報を取り出す
7.         Class clazz = obj.getClass();
```

```
        // クラスの定義情報からすべてのメソッドの定義情報を取り出す
        Method[] methods = clazz.getDeclaredMethods();
        // メソッドの定義情報をもとにインスタンスのメソッドを呼び出す
        for (Method m : methods) {
            System.out.println(m.invoke(obj));
        }
    }
}
```

クラスの定義情報を取り出せれば、次はその定義情報からそのクラスはどのようなメソッドを持っているかを調べます。Classクラスの**getDeclaredMethodsメソッド**を使えば、そのクラスに宣言されているメソッドについての情報を扱うMethodクラスのインスタンスの一覧が取り出せます。最後に、そのメソッドの定義情報を使って、**invokeメソッド**でメソッドを起動しています。invokeメソッドは、どのインスタンスのメソッドを起動するかを指定するために、起動対象のインスタンスへの参照を引数として受け取ります。
注目してほしい点は、このSampleクラスにはItemクラスに関する情報が一切表れていないことです。このように、利用するクラスと利用されるクラスを完全に分離することができるのがリフレクションの特徴です。このクラスを実行すると、次のようにgetterメソッドの結果が表示されるはずです。

例 実行結果

```
> java Sample
apple
100
```

このリフレクションは大変強力な機能ですが、攻撃者が悪用できる懸念もあるため、安易に利用すべきではないでしょう。モジュール・システム導入により、このリフレクションも制限を受けることになります。リフレクションが使えるのは、あくまでもexportsされたパッケージのうち、公開クラスが持つ、公開されているメンバ（フィールドやメソッド）に限定されるからです。しかし、このリフレクションという便利な機能をまったく利用できないというのも問題があります。だからといって、リフレクションでアクセスしてほしいクラスは、必ずしもexportsによって公開したいわけではないのです。そこで、公開はしないけれどリフレクションではアクセスできるようにするのが、**opensディレクティブ**です。
次のようにopensディレクティブで指定すると、そのパッケージは外部のモジュールからリフレクションができるようになります。

例 module-info.java（opensの使用例）

```
module com.sample.Value {
    opens com.sample.value.core;
}
```

この方法では、どの外部モジュールからでもリフレクションができてしまうため、セキュリティ上の懸念が残ります。そこで、リフレクションでアクセスできるモジュールを限定することができます。次のようにopensディレクティブで指定したパッケージの後ろに、toに続けてリフレクションを許可するモジュール名を記述すると、アクセスできるモジュールを限定できます。なお、リフレクションを許可するモジュールが複数ある場合は、toに続けてモジュール名をカンマ区切りで列挙します。

例 module-info.java（opens〜toの使用例）

```
module com.sample.Value {
    opens com.sample.value.core to com.sample.Test;
}
```

以上のことから、選択肢Eが正解です。なお、パッケージ単位でリフレクションを制御するのではなく、その**モジュール全体**でリフレクションを許可したい場合には、**open module**として宣言します。

例 module-info.java（open moduleの使用例）

```
open module com.sample.Value {
    … 省略 …
}
```

モジュール定義ファイルの記述方法とディレクティブの用途を押さえておきましょう。

3. C

➡ P460

モジュールの種類に関する問題です。
モジュール・システムが導入される以前は、コンパイラやJVMが使うクラスなどのリソースを検索する仕組みとしてクラスパスが利用されてきました。クラスパスを利用すれば、任意の場所にあるクラスファイルなどを使うこと

ができます。
モジュール・システムでは、モジュールを配置するための「**モジュールパス**」が追加されました。コンパイラやJVMは、モジュールパスで指定された場所を検索することで、必要なモジュールを利用することができます。任意の場所をモジュールパスに追加したい場合には、**javacコマンド**や**javaコマンド**の**--module-pathオプション**を使います。
モジュール・システムが導入されたJava SE 9以降であっても、従来どおりクラスパスを利用できますが、これはあくまでも導入以前に開発された既存のアプリケーションやライブラリとの互換性を保つことを目的としており、内部的にモジュール・システムを利用して動作しています。そのため、新規開発するアプリケーションでは、クラスパスではなく、モジュールパスを使うことが推奨されています。

モジュール・システムでは、モジュールを「名前付きモジュール」「自動モジュール」「無名モジュール」の3種類に分けています。
名前付きモジュールとは、モジュール定義ファイル（module-info.java）でモジュール名が定義されているモジュールのことです。基本的には、このモジュールが標準となります。自動モジュールと無名モジュールは、モジュール・システムが導入される以前のアプリケーションやライブラリとの互換性を保つ目的で利用します。

以前、開発したライブラリを利用するときのように、モジュール定義ファイルを持たないjarファイルがモジュールパス上に配置された場合、そのjarファイルは自動的にモジュールとして扱われます。これを「**自動モジュール**」と呼びます。
自動モジュールはモジュール定義ファイルを持たないため、パッケージ単位でのアクセス制御はできず、すべてのパッケージが公開された（exportsされた）ものとして扱われます。また、利用しているモジュールがある場合には、それらのモジュールはすべてrequiresで宣言されているものとして扱われます。そうしないとモジュール・システムで動作するアプリケーションと混在させて動作させることができないためです。
自動モジュールはモジュール定義ファイルを持ちませんが、モジュール名を**マニフェストファイル**に定義することはできます。マニフェストファイルは、jarファイルを展開したときにルートディレクトリ直下に**META-INF**というディレクトリを作り、**MANIFEST.MF**という名前で作ります。プログラマーがjarファイルを作るときにマニフェストファイルを作らなければ、バージョン番号だけが記述されたマニフェストファイルが自動的に作られます。
もし、マニフェストファイルに自動モジュールの名前を定義するのであれば、次のように**Automatic-Module-Name属性**で指定します。

例 マニフェストファイル

```
Automatic-Module-Name: Lib
```

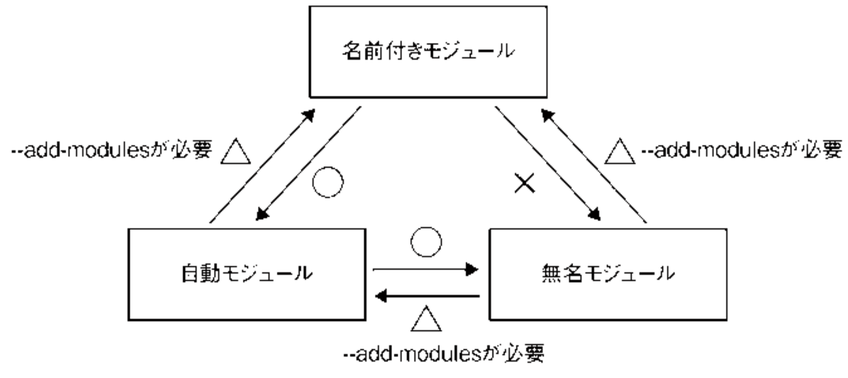
マニフェストファイルにモジュール名を指定しない場合は、jarファイルのファイル名がモジュール名として扱われます。そのとき、次のようなルールに従って名前が決定されます。

1. 拡張子「.jar」は削除される
2. ファイル名にハイフン「-」が含まれて、そのハイフン以降が数字もしくはドット「.」の場合（正規表現で「-(\d+(\.|$))」に一致する場合）、最初のハイフンよりも前の部分が利用される
3. 英数字でない文字（ハイフンなどの記号）は、すべてドットに置き換えられる

たとえば、sample-1.0.0.jarというjarファイルだった場合、拡張子「.jar」が削除され、バージョン番号「1.0.0」も削除されるため、「sample」というモジュール名として扱われます。

モジュールパスではなく、従来どおりクラスパス上に配置されたjarファイルやクラスは、無名モジュールに属しているものとして扱われます。これは前述のとおり、Java SE 9以降は、内部的にすべてモジュール・システム上で動作させるためです。モジュールパスにはなく、クラスパス上にあるすべてのクラスやパッケージが、無名モジュールに属しているものとして扱われます。このようにモジュール・システムでは、モジュール定義がないクラスやパッケージを混在させて利用できるようになっています。ただし、自動モジュールや無名モジュールはrequiresやexportsといった明示的な宣言を持たないため、これらを混在させて使う場合には、次の図のようにモジュール間の参照が制限されています。

【名前付き、自動、無名のモジュール間参照の制限】



無名モジュールはクラスパスにあるため、自動モジュールからは参照できますが、名前付きモジュールからは参照できません。これは、無名モジュールはモジュール名を持たないためで、名前付きモジュールのモジュール定義ファイルにrequiresで宣言できないからです。
名前付きモジュールから自動モジュールへは、通常の名前付きモジュール同士のようにrequiresで宣言していれば参照できます。
自動モジュールや無名モジュールから名前付きモジュールを参照する場合には、コンパイルや実行時に**--add-modulesオプション**でモジュール名を指定すれば、参照することができます。無名モジュールから自動モジュールを参照するときも同様です。

以上のことから、選択肢Cの説明は誤りです。誤った説明を選択する問題なので、選択肢**C**が正解です。

試験対策

名前付きモジュール、自動モジュール、無名モジュールの3つのモジュールが混在する場合、モジュール間の参照は制限されるものがあります。

4. B

➡ P461

ボトムアップ移行に関する問題です。
解答3では、モジュールには3つの種類があることを学びました。モジュール・システムは、名前付きモジュールだけであれば複雑ではありませんが、既存のアプリケーションやライブラリと混在させ、自動モジュールや無名モジュールを使うことになると、システム構造が複雑になってしまいます。
しかし、すべて新規に開発するのではなく、既存のライブラリを流用しながらシステム開発を行う場合、クラスパスで扱う無名モジュールやモジュールパスで扱う名前付きモジュールと自動モジュールが混在することになります。もし適切に各モジュールの依存関係が解決されなければ、新しく作ったモジュールから既存のライブラリ（無名モジュール）にアクセスできないという問題が発生してしまいます。このため、モジュール・システムへの移行をためらう開発プロジェクトも多くあります。

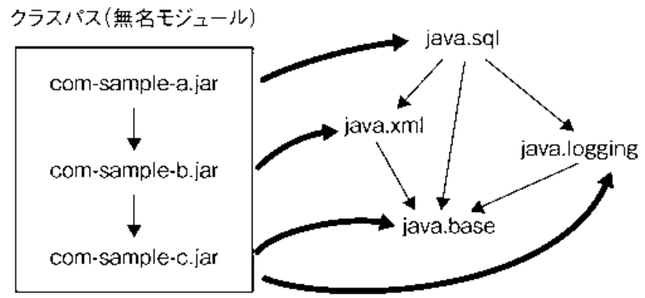
このような問題を解決するために、既存のライブラリを最新のモジュール・システムに移行させる方法として、ボトムアップ移行とトップダウン移行があります。

まず、既存のライブラリやアプリケーションを、モジュール・システムに適合させていくボトムアップ移行について学びましょう。**ボトムアップ移行**とは、複数のライブラリがあったとき、**依存関係の末端から順に、名前付きモジュールに変換していく方法**です。

たとえば、モジュール化されていない既存のライブラリ「com-sample-a.jar」「com-sample-b.jar」「com-sample-c.jar」があったとします。この3つのライブラリはクラスパス上にあり、com-sample-a.jarがcom-sample-b.jarを、com-sample b.jarがcom-sample-c.jarを利用している関係にあります。このような依存関係にあるアプリケーションをJava SE 9以降で利用すると、クラスパス上にある3つのライブラリはすべて無名モジュールとして扱われます。

無名モジュールの特徴は、すべてのモジュールに対して依存関係を持つことです。反対に、モジュールパス上からは、これらの無名モジュールにはアクセスできません。
たとえば、java.baseモジュールのほか、java.sql、java.xml、java.loggingという標準クラスライブラリに含まれるモジュールを利用している場合、次のようなモジュール・グラフを描くことができます。

【モジュール・グラフで表した依存関係】

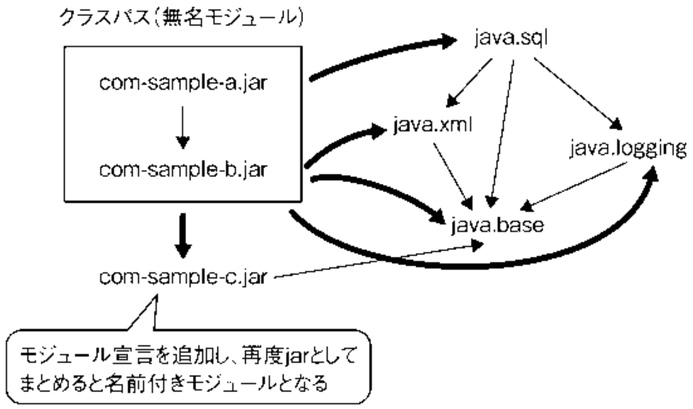


参考

本来のモジュール・グラフは、モジュール同士の関係を表すためのもので、クラスパス内の依存関係は表しません。

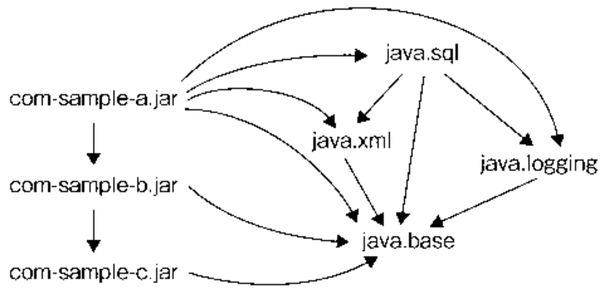
ボトムアップ移行では、クラスパス上にある無名モジュールのうち、最も依存されている側（末端側）から、モジュールに変換していきます。この例であれば、com-sample-c.jarから移行作業を始めます。移行作業は、com-sample-c.jar内のどのパッケージが利用されているのかを調べ、そのパッケージのみを公開するようにモジュール宣言を追加します。クラスパスだけを利用していた従来の方法では、すべてのパッケージが公開されてしまいましたが、ボトムアップ移行の方法でモジュール化することで、必要のないパッケージは非公開になり、最低限の機能のみを公開する安全なモジュールとなります。最後に、モジュール宣言を含む必要なファイルをjarファイルにまとめ直して、名前付きモジュールにします。

【com.sample.cモジュールに変換したあとの依存関係】



あとは、同じようにクラスパス上にある無名モジュールを順に、名前付きモジュールにしていきます。このとき、jdepsなどのツールを使いながら、最低限どのパッケージを公開するのか、どのモジュールを利用するのかを確認しながら移行作業を慎重に進めていきます。すべての無名モジュールを名前付きモジュールに移行できれば、移行作業は完了です。

【名前付きモジュールに移行完了したあとの依存関係】



このように、利用する側ではなく、利用される側の無名モジュールから名前付きモジュールに変換し、モジュール・システムを利用したシステム構成に移行することを「ボトムアップ移行」と呼びます。以上のことから、選択肢Bが正解です。

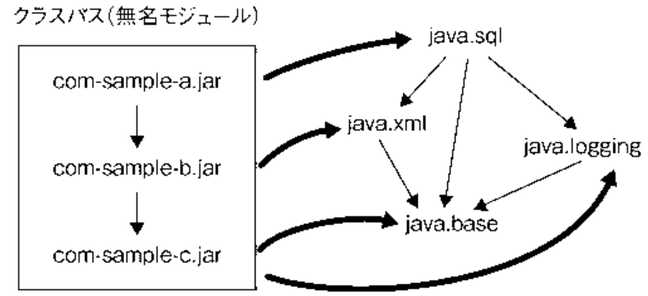
5. D、E

➡ P461

解答4では、2つあるモジュール・システムへの移行方法のうち、ボトムアップ移行について学びました。ここでは、トップダウン移行について学びましょう。ボトムアップ移行は安全で確実な方法ですが、クラスパスの末端にあるライブラリがメンテナンス中であるなどの理由でモジュール化できないときは、**トップダウン移行**を採用します。

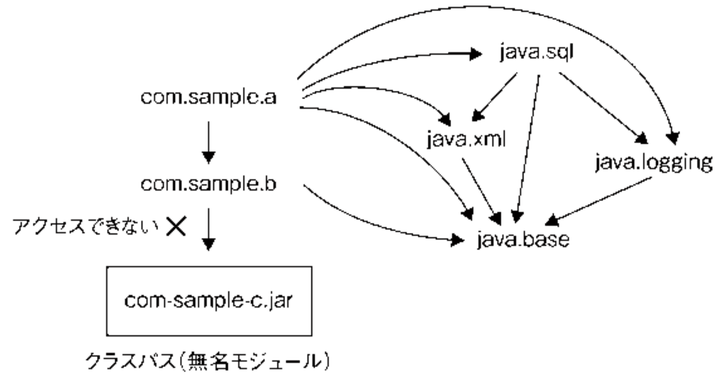
まずは、解答4で使った例を確認しましょう。この例では、図のようにモジュール化されていない既存のライブラリ「com-sample-a.jar」「com-sample-b.jar」「com-sample-c.jar」がクラスパス上にあり、com-sample-a.jarがcom-sample-b.jarを、com-sample-b.jarがcom-sample-c.jarを利用している関係にあります。

【モジュール・グラフで表した依存関係】



また、クラスパス上にあるライブラリからは、java.baseモジュールのほか、java.sql、java.xml、java.loggingという標準クラスライブラリに含まれるモジュールが利用されています。
このような依存関係があるとき、末端にあるcom-sample-c.jarが無名モジュールなまま、利用している側のcom-sample-a.jarとcom-sample-b.jarの2つを先にモジュール化してしまうと、これらの2つのモジュールからcom-sample-c.jarに含まれるクラスにアクセスできないという問題が発生してしまいます。名前付きモジュールから無名モジュールにはアクセスできないというルールがあるためです。

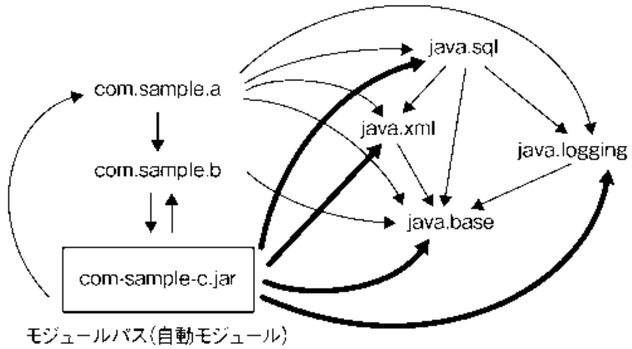
【利用している側を先にモジュール化したあとの依存関係】



このような問題を解決するには、クラスパス上にあって**無名モジュール**として扱われているjarファイルをモジュールパス上に移動し、**自動モジュール**にしてしまうのです。そうすることで、名前付きモジュールからアクセスできるようになります。

自動モジュールはモジュール宣言を持たないため、自動モジュールが依存するほかのモジュールを知る方法はありません。そのため、自動モジュールはほかのすべての名前付きモジュールを読み取り（requires）するように作成されます。よって、モジュールグラフからわかるとおり、次の図のように複雑な依存関係を持つことになります。

【自動モジュールに変換したあとの依存関係】

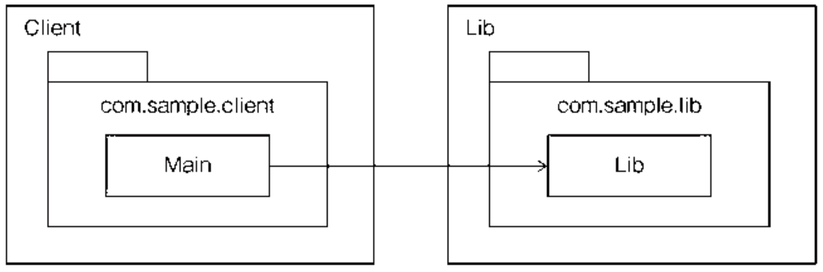


最後に、自動モジュールにしていたメンテナンス中のモジュールに適切なモジュール宣言を追加し、jarファイルにまとめれば、移行が完了します。このようにトップダウン移行では、依存されている側のライブラリをいったん自動モジュールにすることで、システム全体をモジュール・システムに適合させていきます。
以上のことから、選択肢**D**と**E**が正解です。

6. D ➡ P461

本設問では、モジュールを作り、利用するまでの流れを見ていきましょう。ここでは、次の図のようにClientモジュールがLibモジュールを利用することを想定します。Clientモジュールには、com.sample.clientパッケージがあり、その中にMainクラスがあります。一方、Libモジュールには、com.sample.libパッケージがあり、その中にLibクラスがあるという構成です。

【ClientモジュールとLibモジュール】



実際のコードは、次のとおりです。Libクラスにはtestメソッドがあるだけで、呼び出されるとコンソールに「test」と表示します。

例 LibモジュールのLibクラス

```
1. package com.sample.lib;
2.
3. public class Lib {
4.     public void test() {
5.       System.out.println("test");
6.     }
7. }
```

モジュールを作るにあたって、どのパッケージを公開するのかを指定しなければいけません。モジュール定義ファイルでは、次のようにexportsを使って、com.sample.libパッケージを公開しています。これで利用される側のモジュールは準備ができました。

例 Libモジュールのmodule-info.java

```
1. module Lib {
2.     exports com.sample.lib;
3. }
```

次に、利用する側のモジュールを準備しましょう。ClientモジュールのMainクラスはmainメソッドだけが定義されており、LibモジュールのLibクラスのインスタンスを作って、testメソッドを呼び出すだけです。

例 ClientモジュールのMainクラス

```
1. package com.sample.client;
2. import com.sample.lib.Lib;
3.
4. public class Main {
5.     public static void main(String[] args) {
6.         Lib lib = new Lib();
7.         lib.test();
8.     }
9. }
```

利用する側のモジュールのモジュール定義ファイルでは、requiresを使って

Libモジュールを利用することを宣言します。

例 Clientモジュールのmodule-info.java

```
1. module Client {
2.     requires Lib;
3. }
```

ここまでの状態で、ディレクトリ構成は以下のようになっているとします。srcディレクトリにソースファイルとモジュール定義ファイルを置き、コンパイルしたファイルはclassesディレクトリに、そしてモジュールとしてまとめたjarファイルをmodulesディレクトリに置くことにします。

```
.
├── classes
├── modules
└── src
    ├── client
    │   ├── com
    │   │   └── sample
    │   │       └── client
    │   │           └── Main.java
    │   └── module-info.java
    └── lib
        ├── com
        │   └── sample
        │       └── lib
        │           └── Lib.java
        └── module-info.java
```

ソースファイルが用意できたので、次はコンパイルしましょう。コンパイルは、利用される側のモジュールから行います。これは、利用する側のモジュールをコンパイルするときに、利用される側のモジュールが必要になるためです。次のようにjavacコマンドを使って、classes/libディレクトリ内に出力するよう**-dオプション**を指定し、モジュール定義ファイルとLibクラスの両方をコンパイルします。

例 Libクラスとモジュール定義ファイルのコンパイル

```
> javac -d classes/lib/ src/lib/module-info.java src/lib/com/
sample/lib/Lib.java
```


コンパイルが無事終わると、classes/libディレクトリにクラスファイルが出力されているはずです。クラスだけでなく、モジュール定義ファイルもコンパイルされていることがわかります。

```
.
├── classes
│   └── lib
│       ├── com
│       │   └── sample
│       │       └── lib
│       │           └── Lib.class
│       └── module-info.class
├── modules
└── src
    ├── client
    │   ├── com
    │   │   └── sample
    │   │       └── client
    │   │           └── Main.java
    │   └── module-info.java
    └── lib
        ├── com
        │   └── sample
        │       └── lib
        │           └── Lib.java
        └── module-info.java
```

コンパイル後、jarコマンドを使ってclassファイルやコンパイル済みのモジュール定義ファイルをjarファイルにまとめてしまいましょう。このコマンドでは、modulesディレクトリにlib.jarというファイルを出力しています。

例 lib.jarファイルの作成

```
> jar cvf modules/lib.jar -C classes/lib .
マニフェストが追加されました
module-infoが追加されました: module-info.class
com/を追加中です(入=0)(出=0)(0%格納されました)
com/sample/を追加中です(入=0)(出=0)(0%格納されました)
com/sample/lib/を追加中です(入=0)(出=0)(0%格納されました)
com/sample/lib/Lib.classを追加中です(入=387)(出=275)(28%収縮されました)
```

コマンドの実行後、次のようにjarファイルが出力されていれば成功です。

```
.
├── classes
│   lib
│   ├── com
│   │   └── sample
│   │       lib
│   │           Lib.class
│   └── module-info.class
├── modules
│   ── lib.jar
└── src
    … 以下省略 …
```

これで、利用される側のモジュールの準備が整いました。次は、利用する側のモジュールを作っていきましょう。ソースファイルはすでに用意しているので、コンパイルします。

別のモジュールを利用している場合には、その利用しているモジュールを**--module-pathオプション**で検索できるようにしなければいけません。このオプションには、モジュールを検索する場所を指定します。

たとえば、次のコマンドのように--module-pathオプションにmodulesという値を渡せば、カレントディレクトリにあるmodulesディレクトリ内にモジュールがあるかどうかを検索します。このディレクトリにはlib.jarファイルがあるため、lib.jarファイル内のモジュール定義ファイルを調べ、exportsされているパッケージかどうかがチェックされた上で、Mainクラスがコンパイルされます。

例 --module-pathを指定し、Mainクラスをコンパイル

```
> javac --module-path modules -d classes/client/ src/client/mo
dule-info.java src/client/com/sample/client/Main.java
```

なお、このjavacコマンドでは、-dオプションを使ってclasses/clientディレクトリにコンパイル結果を出力しています。これで、ディレクトリ構成は次のようになるはずです。


```
.
  classes
  ├── Client
  │   ├── com
  │   │   └── sample
  │   │       └── client
  │   │           ── Main.class
  │   │   module-info.class
  │   Lib
  │   ── com
  │       ── sample
  │           └── lib
  │               └── Lib.class
  │   ── module-info.class
  modules
  └── lib.jar
── src
  … 以下省略 …
```

コンパイル後は、Clientモジュールもjarファイルとしてまとめてしまいましょう。jarコマンドはファイルをまとめるだけで、モジュールの依存関係を調べないため、特別なオプションは不要です。

例 client.jarファイルの作成

```
> jar cvf modules/client.jar -C classes/client .
マニフェストが追加されました
module-infoが追加されました: module-info.class
com/を追加中です(入=0)(出=0)(0%格納されました)
com/sample/を追加中です(入=0)(出=0)(0%格納されました)
com/sample/client/を追加中です(入=0)(出=0)(0%格納されました)
com/sample/client/Main.classを追加中です(入=337)(出=253)(24%収縮されました)
```

これで、modulesディレクトリに、2つのjarファイルができ、2つのモジュールが用意できたことになります。

※次ページに続く

```
.
├── classes
│   ├── Client
│   │   ├── com
│   │   │   └── sample
│   │   │       └── client
│   │   │           │   Main.class
│   │   └── module-info.class
│   └── Lib
│       ├   com
│       │   └── sample
│       │       └── lib
│       │           └── Lib.class
│       └── module-info.class
├── modules
│   │   client.jar
│   │   lib.jar
└── src
    … 以下省略 …
```

最後にモジュールを実行してみましょう。コンパイル時と同様に、javaコマンドの--module-pathオプションでモジュールを探しに行く場所を指定します。モジュール内の特定のクラスを実行したい場合は、**--moduleオプション**に「**モジュール名/完全修飾クラス名**」の書式で、実行したいクラスを指定します。そのため、次のコマンドを実行すると、「test」とMainクラスが呼び出したLibクラスの結果がコンソールに表示されるはずです。

例 実行結果

```
> java --module-path modules --module Client/com.sample.client.Main
test
```

設問の選択肢AとBは、コンパイルするときに依存するモジュールを検索する--module-pathオプションを使っていません。--moduleは、javaコマンドの実行時に指定するオプションです。よって、誤りです。
選択肢Cは、実行するときに依存するモジュールを検索する--module-pathオプションを使っていません。よって、誤りです。
以上のことから、選択肢**D**が正解です。

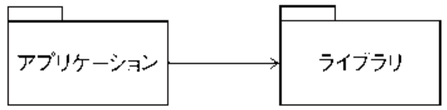
7. C

➡ P462

これまで学んできたモジュール・システムは、セキュリティや保守時の劣化を防止することを目的としてきましたが、このほかにも、変更に強く、柔軟なシステムを構築するためにも利用できます。

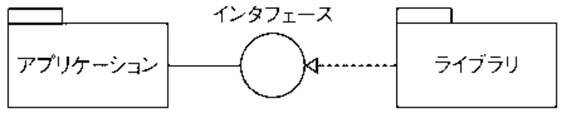
まずモジュール・システムでの実装の前に、柔軟なシステムを実現するための設計原則「**依存関係逆転の原則**（DIP：Dependency Inversion Principle）」について学びましょう。この原則では、2つ以上のモジュールの依存関係があるとき、その依存関係を逆転させます。たとえば次のように、あるアプリケーションがライブラリを使っている関係があったとき、アプリケーションはライブラリに依存している状況だといえます。

【アプリケーションがライブラリに依存している状況】



依存しているということは、ライブラリがなければアプリケーションは成り立たず、ライブラリに変更が生じたときには、その影響を受けることを意味します。これでは、ライブラリが「主」で、アプリケーションが「従」の主従関係になってしまいます。極論すれば、利用しているライブラリの都合にアプリケーションが振り回されるのです。
アプリケーション側からすると、このように振り回されるような関係は望ましくありません。そこで、アプリケーションが必要とする機能をまとめたインタフェースを作り、ライブラリはそのインタフェースを実装するようにします。

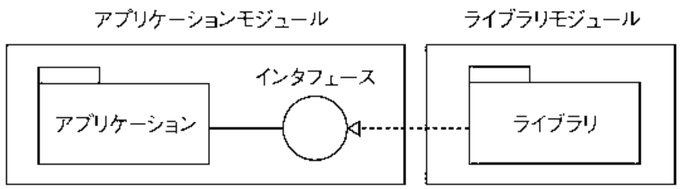
【インタフェースを介在させた状況】



さらに、アプリケーションとインタフェースをモジュールにまとめることで、モジュール間の依存関係は、アプリケーションからライブラリに向いていたのが、ライブラリからアプリケーションの向きに逆転します。

※次ページに続く

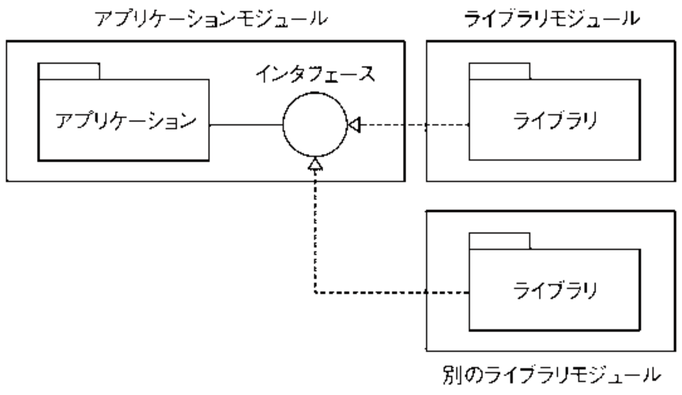
【依存関係が逆転した状況】



こうすることで、アプリケーションが依存するインタフェースをアプリケーション側が管理することができるため、インタフェースに従っているライブラリが自由に変更を加えることができなくなります。このように、依存するインタフェースを誰が管理するのかというポイントは、変更に強い全体設計を考える上でとても重要になります。

この依存関係逆転の原則は、主従関係を逆転させることで、本来は「従」の立場であるものに振り回されない設計を実現するだけでなく、柔軟な変更を実現する効果もあります。たとえば次の図のように、同じインタフェースを実装した別のライブラリを用意すれば、アプリケーションを変更せずに、変化に対応することができます。

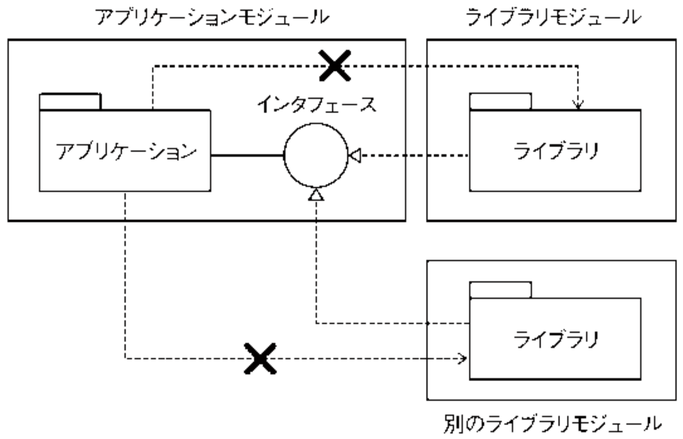
【別のライブラリを追加】



この仕組みを採用しているのが、第6章で学んだJDBCです。JDBCは、APIだけを定め、APIを実装したドライバは各データベースベンダーが提供します。APIは同じであるため、利用するドライバさえ切り替えれば、さまざまなデータベースと同じ方法でやり取りができます。この仕組みにより、アプリケーションはデータベースの違いを意識する必要がなくなります。なお、このようにAPIに対して第三者がその実装を提供する仕組みを、「**SPI（Service Provider Inteface）**」と呼びます。

SPIを実現するには、ライブラリのインスタンスを生成する機能を提供するものが必要です。もし、アプリケーションが直接、ライブラリ内のクラスのインスタンスを生成しているのでは、ライブラリが切り替わったときに、アプリケーション側のコードも変更しなければいけません。

【アプリケーションとライブラリが直接結合した状況】



このような直接的な結合を避けるために、アプリケーションやライブラリとは別のクラスが必要なのです。そこで利用するのが、**java.util.ServiceLoaderクラス**です（選択肢A）。それでは、ServiceLoaderクラスを使ったSPIの実装を見てみましょう。なお、ここではモジュール・システムは利用せず、従来のクラスパスを使って実現します。

まず、アプリケーション側のインタフェースを定めます。このHelloインタフェースは、文字列を戻すhelloというメソッドを1つだけ持っています。

例 Helloインタフェース

```
1. package app;
2.
3. public interface Hello {
4.     public String hello();
5. }
```

このHelloインタフェースの実装はSPIによって提供されるため、このインタフェースを利用するアプリケーションを先に作りましょう。次のSampleクラスではServiceLoaderクラスを利用し、クラスパス上にあるHelloインタフェースの実装を検索し、そのインスタンスを取得しています。

※次ページに続く

例 Sampleクラス（ServiceLoaderクラスの使用例）

```
 1. package app;
 2.
 3. import java.util.Iterator;
 4. import java.util.ServiceLoader;
 5.
 6. public class Sample {
 7.     public static void main(String[] args) {
 8.         Iterator<Hello> hellos =
 9.                 ServiceLoader.load(Hello.class).iterator();
10.         while(hellos.hasNext()) {
11.             Hello h = hellos.next();
12.             System.out.println(h.hello());
13.         }
14.     }
15. }
```

loadメソッドにclassリテラル（クラスの定義情報を表すデータ）を渡すことで、そのインタフェースを実装したクラスを検索し、そのインスタンスの集合を扱うServiceLoaderを戻します。このクラスはjava.lang.Iterableを実装しているため、iteratorメソッドを使えば、インタフェースを実装したクラスのインスタンスの集合を扱うためのイテレータを戻します。

以上で、アプリケーション側の準備は整いました。アプリケーション側のディレクトリ構造は次のとおりです。インタフェースの実装に関するものが含まれていないことを確認してください。

```
application/
├── bin
│   └── app
│       ├── Hello.class
│       └── Sample.class
└── src
    └── app
        ├── Hello.java
        └── Sample.java
```

次はライブラリ側を作りましょう。ライブラリ側は、Helloインタフェースを実装したクラスとSPIのための設定情報を提供します。次のコードが、Helloインタフェースの実装クラスです。

例 HelloImplクラス

```
1. package lib;
2. import app.Hello;
3.
4. public class HelloImpl implements Hello {
5.     @Override
6.     public String hello() {
7.         return "Hello, SPI!";
8.     }
9. }
```

実装クラスが用意できたら、SPIのための設定情報を定義しましょう。**META-INF**というディレクトリ内にservicesというディレクトリを作り、その中に**インタフェースのFQCN**（完全修飾クラス名）と同じ名前のファイルを作ります（選択肢B）。

```
library/
├── bin
│   ├── META-INF
│   │   └── services
│   │       └── app.Hello
│   └── lib
│       └── HelloImpl.class
└── src
    └── lib
        └── HelloImpl.java
```

ファイルの中には、次のように実装クラスのFQCNを記述しておきます。ServiceLoaderクラスは、loadメソッドで渡されたclassリテラルに基づき、クラスパス上にあるMETA-INF/services内のインタフェース名と同じファイルを探し、その中の実装クラスのFQCNを使ってインスタンスを生成するという仕組みです（選択肢D）。

例 app.Helloファイル

```
lib.HelloImpl
```

これで準備が完了したので、ライブラリをjarファイルにまとめます。

※次ページに続く

例 impl.jarファイルの作成

```
> jar cvf impl.jar bin/META-INF/services/app.Hello bin/lib/HelloIm
pl.class
マニフェストが追加されました
META-INF/services/app.Helloを追加中です(入=13)(出=15)(-15%収縮されました)
lib/HelloImpl.classを追加中です(入=384)(出=260)(32%収縮されました)
```

jarファイルができたら、カレントディレクトリをアプリケーション側のbinディレクトリに移動して実行してみましょう。次のように表示されれば成功です。

例 実行結果

```
> java -cp .:../../library/impl.jar app.Sample
Hello, SPI!
```

これでアプリケーションとライブラリの連携はできました。このとき、アプリケーション側を先に作り、そのあとにライブラリ側を作りましたが、アプリケーション側で一切変更を加えていないにもかかわらず、ライブラリと連携ができたことに注目してください。この方法であれば、インタフェースを実装した別のライブラリを用意して切り替えるのも簡単にできるのです。こうして、変化に柔軟に対応できるシステム構造を実現できました。
誤っている説明を選択する問題なので、選択肢**C**が正解です。クラスをロードするのは、ClassLoaderの役割です。

試験対策

SPIは、アプリケーションとライブラリが直接結合するのを避け、変更に対して柔軟に対応できるシステム構造を実現するための仕組みです。ServiceLoaderクラスの使い方を理解しましょう。

8. A

➡ P462

解答7では、java.util.ServiceLoaderを使ったSPIの実装について学びました。本設問では、モジュール・システムを利用したSPIの実装について学びましょう。まず、解答7で利用したアプリケーションにモジュール定義ファイルを追加します。モジュール定義ファイルでは**usesディレクティブ**を使って、SPIとして利用するインタフェースを指定します。

例 module-info.java（appモジュール）

```
1. module app {
2.     exports app;
3.     uses app.Hello;  // app.Helloインタフェースを SPIとして使う
4. }
```

モジュール定義ファイルができたら、jarコマンドを使ってファイルをまとめましょう。これでアプリケーション側の準備は完了です。

例 app.jarファイルの作成

```
> jar cvf app.jar module-info.class app/*.class
マニフェストが追加されました
module-info.classを追加中です(入=174)(出=135)(22%収縮されました)
app/Hello.classを追加中です(入=134)(出=113)(15%収縮されました)
app/Sample.classを追加中です(入=1032)(出=594)(42%収縮されました)
```

次は、SPIを実装するライブラリ側の準備です。ライブラリ側にも、次のようなモジュール定義ファイルを追加します。ここでは、**provides〜withディレクティブ**を使って、app.Helloインタフェースを実装しているのが、lib.HelloImplクラスであることを定義しています。

例 module-info.java（libモジュール）

```
1. module lib {
2.     exports lib;
3.     requires app;
4.     provides app.Hello with lib.HelloImpl;
5. }
```

モジュール定義ファイルの準備ができたら、jarコマンドでまとめます。

例 lib.jarファイルの作成

```
> jar cvf ../lib.jar module-info.class app/*.class
マニフェストが追加されました
module-infoが追加されました: module-info.class
app/Hello.classを追加中です(入=134)(出=113)(15%収縮されました)
app/Sample.classを追加中です(入=1032)(出=594)(42%収縮されました)
```

※次ページに続く

以上で、すべての準備が完了しました。現在のディレクトリ構造とファイル配置は次のような状態で、./appをカレントディレクトリとしています。

```
.
├── app
│   ├── app.jar
│   ├── bin
│   │   ├── app
│   │   │   ├── Hello.class
│   │   │   └── Sample.class
│   │   └── module-info.class
│   └── src
│       ├── app
│       │   ├── Hello.java
│       │   └── Sample.java
│       └── module-info.java
└── library
    ├── bin
    │   ├── META-INF
    │   │   └── services
    │   │       └── app.Hello
    │   ├── lib
    │   │   └── HelloImpl.class
    │   └── module-info.class
    ├── lib.jar
    └── src
        ├── META-INF
        │   └── services
        │       └── app.Hello
        ├── lib
        │   └── HelloImpl.java
        └── module-info.java
```

ここまでできたら、次のコマンドで実行してみましょう。次のように表示されれば成功です。

例 実行結果

```
> java --module-path app.jar:../library/lib.jar -m app/app.Sample
Hello, SPI!
```

なお、META-INF/services/app.Helloは本来不要なため、残しておく必要はありません。もしくは、モジュール・システムに対応していないバージョンとの互換性を担保するために残す方法もあります。

以上のことから、選択肢**A**が正解です。選択肢Bのprovidesは、SPIを実装したクラスを宣言するためのものです。選択肢Cのopensは、リフレクションを可能にするためのものです。選択肢Dのようなディレクティブは存在しません。

9. A

➡ P462

jdepsコマンドに関する問題です。

jdepsはJDKに含まれるツールで、クラスやモジュールの依存関係を調べるために利用します。たとえば、jarファイルを引数に指定すると、jarファイルが依存しているモジュールやパッケージを確認することができます。

例 jdepsコマンドの使用例

```
> jdeps app/app.jar
app
 [file:///Users/shiga/java/app/app.jar]
   requires mandated java.base
app -> java.base
   app    -> java.io      java.base
   app    -> java.lang    java.base
   app    -> java.util    java.base
```

これらの情報を利用すれば、モジュール化されていなかったライブラリをボトムアップ移行やトップダウン移行するときに、モジュール定義ファイルをどのように記述すべきかがわかります。

jdepsには、多くのオプションが用意されています。選択肢Bの**-verbose:class オプション**は、クラスレベルの依存関係を検索するためのものです。たとえば、次のようにオプションを指定してコマンドを実行すると、tools.jarが依存するクラスの一覧が表示されます。

例 -verbose:classオプションの使用例

```
> jdeps -verbose:class -cp lib/tools.jar com.sun.tools.jdeps.Main

lib/tools.jar -> /usr/java/jre/lib/rt.jar
   com.sun.tools.jdeps.Main (tools.jar)
      -> java.io.PrintWriter
      -> java.lang.Exception
      -> java.lang.Object
   … 以下省略 …
```

この情報は、JDKの内部APIを含みません。よって、選択肢Bは誤りです。

選択肢Aの**-jdkinternals**は、JDKの内部APIでクラスレベルの依存関係を検索するためのものです。内部APIとは、Javaプログラマーが利用するためのライブラリではなく、標準クラスライブラリ内で利用されることを目的としたライブラリのことです。デフォルトでは、-includeオプションが指定されていなければ、-classpathオプションおよび入力ファイルで指定されたクラスがすべて分析されます。以上のことから、選択肢**A**が正解です。

選択肢Dの**-profile**は、依存するプロファイルを検索するためのものです。**プロファイル**とは、標準クラスライブラリのサブセットのことで、すべてのライブラリを提供するのがプロファイル「Full JRE」です。一方、compact1〜3までのプロファイルは、Full JREよりも制限されたクラスライブラリしか提供しません。これらのプロファイルは、それより番号の低いプロファイルのAPIを含んでいるため、compact1が最も少ないライブラリのみを提供しているプロファイルとなります。以上のことから、選択肢Dも誤りです。
選択肢Cの**-apionly**は、分析対象をpublicなクラスのpublicもしくはprotectedなメンバに制限するものです。よって、誤りです。

例 -profileオプションの使用例

```
> jdeps -profile demo/jfc/Notepad/Notepad.jar
demo/jfc/Notepad/Notepad.jar -> /usr/java/jre/lib/rt.jar (Full JRE)
  <unnamed> (Notepad.jar)
   -> java.awt                    Full JRE
   -> java.awt.event              Full JRE
   -> java.beans                  Full JRE
   -> java.io                     compact1
   -> java.lang                   compact1
   -> java.net                    compact1
   -> java.util                   compact1
   -> java.util.logging           compact1
   -> javax.swing                 Full JRE
   -> javax.swing.border          Full JRE
  … 以下省略 …
```

試験対策

jdepsコマンドのオプションの意味を覚えておきましょう。

12

Java SEアプリケーションにおけるセキュアコーディング

- セキュアコーディングの重要な10項目
- 入力検証とUnicode正規化
- 整数オーバーフロー攻撃と対策
- 機密情報の扱い
- DoS攻撃と対策
- セキュリティマネージャによるセキュリティ対策
- セキュリティポリシーの設定
- 不変オブジェクトによるセキュリティ対策
- SQLインジェクション対策

1. アメリカのCERTが定めるセキュアコーディングプラクティスのうち、最も重要なものとされている原則を選びなさい。（1つ選択）

A. 入力を検証する
B. セキュリティポリシーを実装するよう設計する
C. 多重防御を実践する
D. セキュアコーディング標準を採用する

➡ P498

2. Unicode正規化を実現するためのメソッドとして、正しいものを選びなさい。（1つ選択）

A. java.text.Normalizerクラスのnormalizeメソッド
B. java.text.Normalizerクラスのtranslateメソッド
C. java.util.Normalizationクラスのnormalizeメソッド
D. java.util.Normalizationクラスのtranslateメソッド

➡ P500

3. 整数オーバーフロー攻撃に対応できるようメソッドの処理を記述している。指針として適切でないものを選びなさい。（1つ選択）

A. 事前条件テストを行う
B. アップキャストを行う
C. ダウンキャストを行う
D. BigIntegerクラスを使う

➡ P502

4. 機密情報の扱いに関する説明として、誤っているものを選びなさい。（1つ選択）

A. 例外が発生したとき、どのような例外が発生したのかをユーザーに見せてはいけない
B. デバッグに使用したログは、監査のときに必要になるので保存しておかなければならない
C. 機密情報を扱ったインスタンスは、ガベージコレクションを待たずに、速やかに削除しなければならない
D. コアダンプは、できるだけ早期に削除すべきである

➡ P504

5. Javaアプリケーションの開発において、DoS攻撃対策として適切でないものを選びなさい。（1つ選択）

A. ファイルを読み込む際には、ファイルのサイズを調べてから読み込むようにすべきである
B. ファイルへの接続は、try-with-resources文を使ったリソースの自動解放機能を利用するべきである
C. データベースへの接続は、ガベージコレクションによるメモリの自動解放を利用するべきである
D. メモリに過剰なデータを読み込まないよう、プログラムで予防的な制限をかけるべきである

➡ P506

6. セキュアなアプリケーションを開発しようとしている。次のうち、正しい説明を選びなさい。（1つ選択）

A. Javaはデフォルトでセキュリティマネージャが有効になっているため、権限のないファイルやディレクトリに対するアクセスを防ぐことができる
B. セキュリティマネージャを有効にするには、javaコマンドのオプションに「-Djava.security.manager」を追加する
C. 権限のないファイルやディレクトリに対するアクセスを防ぐには、不正なアクセスを監視するプログラムを独自に開発する
D. Javaのアプリケーションが行う処理の権限を監視するのが、java.security.Policyクラスである

➡ P507

7. 特定のjarファイルだけに、セキュリティ上の特権を与えたい。セキュリティポリシーファイルの記述として、正しいものを選びなさい。（1つ選択）

A.
```
grant {
    permission java.security.AllPermission;
};
```

B.
```
grant base "file:/Users/shiga/workspace/chap12/lib.jar" {
    permission java.io.FilePermission "/", "read";
};
```

C.
```
grant jar "file:/Users/shiga/workspace/chap12/lib.jar" {
    permission java.io.FilePermission "/", "read";
};
```

D.
```
grant codeBase "file:/Users/shiga/workspace/chap12/lib.jar" {
    permission java.io.FilePermission "/", "read";
};
```

➡ P510

8. 不変オブジェクトを定義する際に、セキュリティ面で適切でない方法を選びなさい。（1つ選択）

A. フィールドの値を変更するようなsetterメソッドは提供しない
B. フィールドの値を変更するのであれば、ディープコピーしたインスタンスを生成して、その参照を戻す
C. フィールドはfinalで宣言し、コンストラクタで初期値を設定したら変更できないようにする
D. 継承ができないように、クラス宣言をfinalで修飾する
E. 引数なしのpublicなコンストラクタを用意する
F. インスタンスを生成するための専用のビルダーを用意する

➡ P514

9. 可変オブジェクトを安全に扱うための指針として、誤っているものを選びなさい。（1つ選択）

A. フィールドをprivateで修飾し、アクセサメソッドを提供する
B. getterメソッドでは、フィールドのオブジェクトをコピーを作って戻す
C. コピーコンストラクタを用意する
D. 可変オブジェクトを受け取るコンストラクタでは、受け取った参照をそのまま利用しない

➡ P517

10. セキュアコーディングを実践するための方法として、誤っているものを選びなさい。（1つ選択）

A. パッケージ内で公開するインタフェースを限定する
B. 原則的に、クラスはprotectedで宣言する
C. モジュールを使って公開パッケージを限定する
D. クラスを宣言するときにはfinalで修飾する

➡ P520

11. SQLインジェクションを防ぐために使用するインタフェースとして、正しいものを選びなさい。（1つ選択）

A. java.sql.Statement
B. java.sql.PreparedStatement
C. java.sql.CallableStatement
D. java.sql.SecureStatement

➡ P521

12. 機密情報を扱うインスタンスをシリアライズするときに実施すべき対策として、正しいものを選びなさい。（1つ選択）

A. フィールドをtransientで修飾する
B. フィールドをstrictfpで修飾する
C. クラスをvolatileで修飾する
D. クラスをfinalで修飾する

➡ P523

13. シリアライズ・プロキシ・パターンの説明として、正しいものを選びなさい。（1つ選択）

A. シリアライズするときにはwriteReplaceメソッドが、デシリアライズするときにはreadObjectメソッドが利用される
B. シリアライズするときにはwriteReplaceメソッドが、デシリアライズするときにはreadResolveメソッドが利用される
C. シリアライズするときにはreadObjectメソッドが、デシリアライズするときにはwriteReplaceメソッドが利用される
D. シリアライズするときにはreadResolveメソッドが、デシリアライズするときにはwriteReplaceメソッドが利用される

➡ P524

第 12 章　Java SEアプリケーションにおけるセキュアコーディング
解　答

1.　A

➡ P494

Javaに限ったことではありませんが、システムを運用する上で、気を付けなければいけないのがセキュリティ対策です。特に、Javaが使われているWebアプリケーションなどでは、膨大な数の悪意を持ったユーザーからの攻撃に日々晒されることを前提とすべきです。
カーネギーメロン大学のSoftware Engineering Institute（SEI）では、セキュリティに配慮したコーディング標準「SEI CERT Oracle Cording Standard for Java」※1を公開しています。その中でも**セキュアコーディング**（セキュアプログラミング／防御的プログラミング）で最も重要なトップ10を「**Top 10 Secure Coding Practices**」としてまとめています。以下の1〜10が、その概要です。

1. 入力を検証する

すべての信頼できないデータソースからの入力を検証することを意味します。ソフトウェアの脆弱性は外部からの入力によって発生することが多いため、正しい値が入力されていることを確認する入力検証が適切に行われるだけで、ソフトウェアの脆弱性のかなり多くの部分を排除できるためです。そのため、この原則が最も重要なセキュアコーディングの原則として定められています。よって、選択肢**A**が正解です。

2.コンパイラからの警告に注意を払う

コンパイラが発生させる警告を無視せず、コードを修正することで警告が発生しないようにコーディングすることです。コンパイラのほかに、セキュリティ問題を発見し、排除するための分析ツールを併用するのもよいでしょう。これらのツールやコンパイラによっては、警告レベルを制御できるものもありますが、できるだけ高い警告レベルを採用することで、セキュリティのレベルが担保されます。また、分析ツールを利用することで、プログラマーの学習効果も働き、セキュリティに対する意識が向上するメリットもあります。

3. セキュリティポリシーを実装するよう設計する

設計者に対して、セキュリティポリシーに従った設計をするよう強制することです。たとえば、異なる権限を異なる場合に必要とするシステムであれば、それぞれが適切な権限で動作するよう、システムを独立したサブシステムに分離し、相互通信させることを検討します。

4. シンプルにする

設計時にできるだけ簡素でかつ小さな単位に分割するように心がけることです。複雑な設計により、エラー発生の可能性が劇的に増大し、セキュリティ対策もより複雑化してしまうからです。

5. 拒否をデフォルトにする

ブラックリスト方式ではなくホワイトリスト方式を採用することです。ブラックリスト方式とは、拒否すべき対象だけを挙げておき、それ以外をすべて許可する方式です。一方のホワイトリスト方式は、許可すべき対象だけを挙げておき、それ以外を拒否する方式です。ホワイトリスト方式のほうがブラックリスト方式に比べて、許可される範囲が最小限におさまるため、より安全性が確保されるのです。

6. 最小権限の原則に従う

アプリケーションを実行するのに必要な（アプリケーションそのものを含む）すべてのプロセス（実行中のプログラム）は、最小限の権限で実行することです。たとえばデータベースを利用するなら、アプリケーションを実行するのに必要な最小限の権限を持つ専用のユーザーを作成すべきです。不要に強い権限を与えてしまうと、悪意のあるユーザーに攻撃されたときの影響を最小化できないためです。

7. ほかのシステムに送信するデータを無害化する

アプリケーションと連携するサブシステム、ライブラリ、データベースなどに渡すデータをすべて無害化することです。攻撃者はこれらのコンポーネントに対して、SQLコマンドやその他のインジェクション攻撃ができてしまうからです。入力検証でチェックをすべきですが、入力検証だけで連携するすべてのコンポーネントを網羅することはできません。そのため、各コンポーネントにデータを送る際には、そのデータを無害化する「出力に対する考慮」が重要なのです。

8. 多重防御を実践する

リスクは複数の防御的戦略によって管理することです。これはたとえ最初の防御施策が不適切だったとしても、次の防御施策が問題の発生を防いだり、結果を緩和したりすることができるからです。たとえば、セキュアプログラミングのテクニックとセキュアな実行環境を組み合わせることにより、開発したコードの脆弱性を突いた攻撃を実際の運用時に受けても、問題が深刻化する可能性が低減します。

※次ページに続く

※1　https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88487865
日本語訳「Javaセキュアコーディングスタンダード CERT/Oracle版」 https://www.jpcert.or.jp/securecoding_standard_java.html

9. 効果的な品質保証テクニックを利用する

アプリケーションに問題を引き起こしそうなデータを大量に送り込むことで、その応答や挙動を監視して脆弱性を検出するファジングテストや、攻撃者に成りすました侵入テスト、コードレビューなどの品質保証テクニックを活用し、脆弱性を識別・排除することです。ソフトウェア開発する組織は、いかにして品質保証を担保するのかという組織的な取り組みを常日頃から整備すべきでしょう。

10. セキュアコーディング標準を採用する

使用するプログラミング言語とプラットフォームに適したセキュアコーディング標準を定め、運用することです。コーディングスタイルを定めたコーディング規約だけでなく、自社独自やプロジェクト独自のセキュアコーディング標準を定めることで、セキュアコーディングに関する意識や取り組み、教育を実施することができるからです。

2. A ➡ P494

解答1で学習したように、セキュアプログラミングの基本は入力検証です。入力検証の前に、文字列の正規化について学びましょう。まずは入力された文字列を処理しやすい形に変形したり整理したりする**正規化**（**normalization**）について学びましょう。

Javaは、文字を**Unicode**で扱います。Unicodeは、世界中の文字を1つのコード体系で表現することを目指して作られた文字コードです。そのため、人間にとっては同じ意味の文字が複数登録されていることがあります。たとえば、半角カナの「ｱ」と全角カナの「ア」や、数字の「①」と「1（全角）」と「1（半角）」、はかにもわかりづらいところでは「﹣（U＋FE63）」と「－（U＋FF0D）」などがあります。このような文字を統一して利用できる仕組みがなければ、すべての文字に対応する必要が生じてしまいます。そこで、「**Unicodeの正規化**」を行うことで、文字を統一して扱えるようにするのです。**入力検証は、この正規化を行ったあと**に実施しなければいけません。

世界中のさまざまな文字を表現するために、Unicodeでは複数の文字を結合して1つの文字を表現できるようになっています。このような結合された文字のことを「**結合文字**」と呼びます。たとえば、「で（U+3067）」という文字はあらかじめ合成されている文字（**合成済み文字**）なので1文字ですが、Unicodeでは「て（U+3066）」の後ろに「濁点（U−3099）」を配置することによって、「で」を表現することもできるのです。
結合文字は複雑な仕組みではありますが、世界中の文字を扱うためにはしかたのない方法です。Unicode正規化は、このような結合文字を合成済みに統一したり、合成済み文字を結合文字にしたりする処理です。

Unicodeを標準化しているUnicodeコンソーシアムでは、Unicode正規化について、次の4つの方法を定めています。

【Unicode正規化の方法】

方法	概要
NFD （Normalization Form Canonical Decomposition）	正準等価性に基づく分解
NFC （Normalization Form Canonical Composition）	正準等価性に基づく分解後、正準等価性に基づいて合成
NFKD （Normalization Form Compatibility Decomposition）	互換等価性に基づく分解
NFKC （Normalization Form Compatibility Composition）	互換等価性に基づく分解後、正準等価性に基づいて合成

分解とは、合成済み文字を結合文字列に変換するような処理のことです。**合成**は分解の逆で、結合文字列を合成済み文字に変換するような処理のことです。また、分解・合成の判断材料に用いるのが**等価性**です。Unicodeの正規化においては「正準等価」と「互換等価」の2種類が規定されています。**正準等価**とは、同じ文字の見た目と意味を持つ文字であれば、合成済み文字と結合文字列は等価であると考える方法です。**互換等価**は、正準等価をより緩くし、見た目が異なっても同じ意味であれば等価であると考える方法です。
互換等価は正準等価を含んでいますが、その逆は成り立ちません。前述の「で」と「て＋濁音」は正準等価であり、かつ互換等価です。一方で、1文字で書く「㈱」と3文字の「（株）」は互換等価ですが、正準等価ではありません。

では、実際にJavaでUnicode正規化を行うコードを見ていきましょう。
Unicode正規化を行うには、**java.text.Normalizerクラス**を使います。このクラスの**normalizeメソッド**の引数に、正規化したい文字列と、前述した4つの正規化のいずれかを指定するEnumの列挙子を渡します。
次のコードでは、「て＋濁音」を表すUnicodeエスケープシーケンス（¥uから始まる番号）を表示します。まず、このコードでは文字結合が行われ、表示上は1文字に見えるものの、lengthメソッドが戻す文字数は2となります。その後、正規化を行うと、lengthメソッドは文字数1を戻し、1文字となることが確認できます。

例 Unicode正規化

```
1. import java.text.Normalizer;
2. import java.text.Normalizer.Form;
3.
```

```
 4. public class Sample {
 5.     public static void main(String[] args) {
 6.         String de = "¥u3066¥u3099";
 7.         System.out.println(de);
 8.         System.out.println(de.length());
 9.
10.         String result = Normalizer.normalize(de, Form.NFKC);
11.         System.out.println(result);
12.         System.out.println(result.length());
13.     }
14. }
```

このコードを実行すると、最初は2文字だった「で」が1文字に正規化されているのがわかります。以上のことから、選択肢**A**が正解です。

例 実行結果

```
> java Sample
で    ←1文字に見えるが
2     ←実際は2文字
で
1     ←正規化されて1文字になる
```

3. C ➡ P494

入力検証に関する問題です。
入力値を使った攻撃手段に、**整数オーバーフロー攻撃**というものがあります。これは、整数の演算結果が想定外の値を作り出してしまう「整数オーバーフロー」を悪用し、意図的にこうした処理を行うコードをシステムに侵入させ、実行する攻撃です。このような攻撃への対策として、プログラムは処理の実行前に、あらかじめ整数がオーバーフローしないようチェックしなければいけません。

たとえば、byte型は8ビット分のデータ、つまり2進数で8桁の数値を扱います。8つの0と1の組み合わせで−128〜127の256通りの数字が表現できます。このとき、−128を2進数にすると「10000000」というビット列になります。一方で、127を2進数にすると「01111111」です。
そこで、10進数で127、つまり2進数の01111111に1を加算すると、一番右のビットから順に繰り上がって、2進数の並びは10000000となります。127に1を足すと−128という結果になるのです。

実際には、byte型の値で何か問題となることは考えにくいのですが、このような仕組みを応用して想定外の演算結果を作り出せば、システムのさまざまなところでエラーが発生する可能性が出てきます。こうして攻撃者は得られたエラー情報から、さまざまな情報を推測することができるわけです。

この攻撃への対策には、次の3つの方法が考えられます。

1. 事前条件テスト
2. アップキャスト
3. BigIntegerクラスを使う

事前条件テストとは、演算子に渡す値が整数オーバーフローを発生させない範囲に収まっていることを事前にチェックすることです。演算を行った場合に整数オーバーフローが発生することを検出したら、例外をスローするというものです。ただし、この方法は演算子ごとに検証方法を変えなくてはいけないため実装が難しく、かつ実装の正確性に問題が起きやすいという特徴があります。
そこで、可能であればアップキャストを採用します。**アップキャスト**とは、入力された値を1つ大きなプリミティブ整数型に型変換し、その型で演算を行う方法です。演算途中の値が元の整数型で表現できる範囲に収まっているかを検査し、範囲外の値であれば例外をスローすればよいのです。ただし、この方法は**long型**には使えません。long型よりも大きな値を扱える整数型が存在しないためです。

例 整数オーバーフローが発生しないかチェックする

```
public long check(long value) throws ArithmeticException {
    if ((value < Integer.MIN_VALUE)
        || (value > Integer.MAX_VALUE)) {
            throw new ArithmeticException("Integer overflow");
    }
    return value;
}
```

このように、プリミティブ型では扱えないような大きな数値を扱う場合は、**java.math.BigIntegerクラス**を使いましょう。long型の最大値は9223372036854775807ですが、BigIntegerクラスにはこの最大値よりも大きな値を格納することができます。また、値を格納するためのメモリは必要に応じて動的に割り当てられるため、理論上、BigIntegerクラスに格納できる値に上限はありません。

※次ページに続く

例 BigIntegerクラスに数値を格納する

```
private static final BigInteger bigMaxInt
    = BigInteger.valueOf(Integer.MAX_VALUE);

private static final BigInteger bigMinInt
    = BigInteger.valueOf(Integer.MIN_VALUE);

public BigInteger check(BigInteger val)
                    throws ArithmeticException {
    if (val.compareTo(bigMaxInt) == 1 ||
        val.compareTo(bigMinInt) == -1) {
        throw new ArithmeticException("Integer overflow");
    }
    return val;
}
```

なお、BigIntegerによる処理は、元のプリミティブ型の演算よりも大幅に遅くなる可能性があります。int型までの範囲であればアップキャストでの演算を考慮してもよいでしょう。
以上のことから、選択肢**C**が正解です。

4. B

➡ P494

機密情報の扱いに関する問題です。
攻撃者の目的は、**DoS攻撃**のようにサーバーやJVMをダウンさせることだけではありません。アプリケーションが扱う**機密情報**を盗み出すことを目的としている攻撃者もいます。このような攻撃を避けるためには、まず攻撃者に対して攻撃を行う手がかりとなる情報を漏らさないことです。その1つが、**例外に関する情報**です。

システムで処理をしている最中に例外が発生した場合、スタックトレースやエラー画面をそのままユーザーに見せてしまうと、攻撃者はその情報を手掛かりにして攻撃方法を変えることができます。例外が発生したときに出力される**スタックトレース**も、メモリ上のプロセスの構成や状態に関する情報を攻撃者に漏洩するものになり得ます。そのため、catchし忘れた非検査例外が原因で、アプリケーションが終了するようなことがあってはいけません。

たとえば、例外が発生したことが攻撃者に伝わることで、次ページの表に挙げたさまざまな**情報漏洩**や**脅威**が起こり得ます。

【例外と起こり得る情報漏洩・脅威】

例外	起こり得る情報漏洩や脅威
java.io.FileNotFoundException	ファイルシステムの構成、ユーザー名の一覧
java.lang.OutOfMemoryError	DoS攻撃
java.lang.StackOverflowError	DoS攻撃
java.net.BindException	信頼できないクライアントがサーバーのポートを選択できる場合、開いているポートの一覧
java.security.acl.NotOwnerException	所有者の一覧
java.sql.SQLException	データベースの構成、ユーザー名の一覧
java.util.ConcurrentModificationException	スレッドセーフでないコードに関する情報
java.util.jar.JarException	ファイルシステムの構成
java.util.MissingResourceException	利用しているリソースの一覧
javax.naming.InsufficientResourcesException	不十分なサーバーリソース、DoS攻撃

例外情報のほかに、デバッグで利用される**コアダンプ**やデバッグ時の**ログ**などの機密性の高い情報を、たとえ管理者であったとしても、必要以上に長く保存してはいけません。これらが漏洩するだけで、攻撃者は攻撃にとって有用な情報を得ることになるからです。

また、機密情報を扱ったインスタンスはできるだけ速やかにメモリから削除するよう設計しましょう。ガベージコレクションがあるからといって、放置してはいけません。**ガベージコレクション**が実行されるまでの間に、攻撃者によって機密情報が漏洩する可能性があるからです。利用し終わった機密情報を持つインスタンスは、明示的にその状態をクリアする機能を持たせることで、フィールドの値をクリアし終わってから、ガベージコレクションにインスタンスの処理がされるよう設計しましょう。

以上のことから、選択肢Bの説明が誤りです。誤った説明を選択する問題なので、選択肢**B**が正解です。

5. C ➡ P495

DoS攻撃への対策に関する問題です。
システムの運用にあたっては、**DoS攻撃**に備えなければいけません。この攻撃は、システムが稼働しているサーバーなどに過剰な負荷をかけることでメモリやCPUなどのリソースを使い切らせ、サーバーをダウンさせるものです。Javaのアプリケーションであれば、**JVM**に負荷をかけることでダウンさせる攻撃もDoS攻撃となります。

情報漏洩を防ぐことだけが、DoS攻撃への対処方法ではありません。たとえば、**ファイルの読み込み**を悪用してDoS攻撃を行うことは可能です。第5章で学んだように、ファイルからテキストを読み込むときは、java.io.BufferedReaderを使って"1行ずつ"データを読み込むのが一般的です。この"1行ずつ"とは、文字列の中で改行（¥n）か復帰（¥r）、または復帰とそれに続く改行（¥r¥n）が表れるまでの文字列を指します。そのため、これらの改行コードがない、非常に長い行を持ったファイルを読み込ませることで、メモリを使い果たしてしまうように仕向けることができます。

このような攻撃を防ぐには、ファイルを読み込む際に**ファイルサイズを制限する**方法があります。次のコード例は、**java.nio.file.Filesクラス**の**sizeメソッド**を使ってファイルサイズを調べ、それがあらかじめ規定しておいた上限である1MB（1,048,576バイト）を超える場合には例外をスローします。

例 ファイルサイズを制限して読み込む

```
public static final int FILE_SIZE_LIMIT = 1048576;

public ReadNames(String filename) throws IOException {
    long size = Files.size(Paths.get(filename));
    if (FILE_SIZE_LIMIT < size) {
      throw new IOException("File too large");
    } else if (size == 0L) {
      throw new IOException("File size cannot be determined");
    }
    // do something
}
```

Javaアプリケーションに対するDoS攻撃を受ける可能性がある脆弱性の1つに、**リソースの解放し忘れ**があります。Javaはガベージコレクタによるメモリの自動解放機能を備えているものの、ファイルへの接続やデータベースへの接続といった、メモリ以外のリソースを自動的に検知、解放する機能は備えていません。そのため、プログラマーがリソースを解放し忘れるとDoS攻撃の対象となる可能性があります。
DoS攻撃を避けるためにも、ファイルやデータベースなどを利用する場合は、**try-with-resources**を使ってリソースを自動的に解放することを徹底しましょう。次のコード例は、JDBCを利用したデータベース接続を行うものですが、try-with-resourcesで接続を自動的に切断しています。

例 try-with-resourcesでデータベースから自動的に切断する

```
try (Connection conn = getConnection();
    PreparedStatement stmt = conn.prepareStatement(sqlQuery);
    ResultSet rs = stmt.executeQuery()) {
    // do something
} catch (SQLException e) {
    // 例外処理
}
```

以上のことから、選択肢**C**が正解です。

6. B ➡ P495

Javaのセキュリティマネージャに関する問題です。
Javaアプリケーションにおけるセキュリティ対策の基本は、**標準クラスライブラリが持つセキュリティ機構**をしっかりと活用することです。実行時にセキュリティポリシーを設定しておけば、許可した動作しかできなくなるのです。実際に試してみましょう。
たとえば次のクラスを実行すると、ルートディレクトリ内のファイルとディレクトリの一覧が表示されます。なお、パスを変更すればどこでも一覧を見ることができます。つまり、どこに何があるかを調べることができるわけです。

例 Sampleクラス（ファイルとディレクトリの一覧を表示）

```
1. import java.nio.file.Files;
2. import java.nio.file.Paths;
3.
4. public class Sample {
5.     public static void main(String[] args) throws Exception {
6.         Files.list(Paths.get("/")).forEach((p) -> {
7.             System.out.println(p.getFileName());
8.         });
9.     }
10. }
```

例 実行結果 ※実行環境によって表示結果が異なります。

```
> java Sample
home
usr
bin
sbin
… 以下省略 …
```

このようなコードは、セキュリティの観点からは望ましくなく、本来は許可されたディレクトリだけにアクセスできるようにするべきでしょう。では、実際にJavaの標準のセキュリティ機構を有効にしてみましょう。先ほどのSampleクラスに、**java.lang.SecurityManager**のインスタンスを生成して、JVMに登録するコードを追加してみましょう。

例 Sampleクラス（SecurityManagerのインスタンス生成）

```
1. public class Sample {
2.     public static void main(String[] args) throws Exception {
3.         System.setSecurityManager(new SecurityManager());
4.         Files.list(Paths.get("/")).forEach((p) -> {
5.             System.out.println(p.getFileName());
6.         });
7.     }
8. }
```

このようにSecurityManagerを追加することで、**セキュリティポリシー**に則ったセキュリティ機構が有効になります。そのため、このコードを実行すると、先ほどとは違い、アクセス権がない旨を表す**java.security.AccessControlException**が発生します。これで任意の場所を調べることができなくなりました。

例 実行結果

```
> java Sample
Exception in thread "main" java.security.AccessControlException: acc
ess denied ("java.io.FilePermission " "/" "read")
    at java.base/java.security.AccessControlContext.checkPermissi
on(AccessControlContext.java:472)
… 以下省略 …
```


なお、セキュリティ機構を有効にするだけなら、SecurityManagerを使わなくても、次のように**-Dオプション**を指定してjavaコマンドを実行するだけでもできます。よって、選択肢**B**が正解です。

例 実行結果

```
> java -Djava.security.manager Sample
```

このままではすべての場所にアクセスできなくなりますので、セキュリティポリシーを設定して、アクセスを許可してみましょう。セキュリティポリシーを記述するには、ファイルに記述する方法とJavaのコードで記述する方法があります。ここでは、より細かな制御ができる、Javaのコードで記述する方法についてのみ解説します。
セキュリティポリシーを設定するには、**java.security.Policyクラス**を継承したサブクラスを作り、次のSamplePolicyクラスのように権限のコレクションを持ったクラス（java.security.PermissionCollection）をフィールドとして用意します。

例 SamplePolicyクラス（セキュリティポリシーの設定）

```
1. import java.security.Permission;
2. import java.security.PermissionCollection;
3. import java.security.Policy;
4. import java.security.ProtectionDomain;
5.
6. public class SamplePolicy extends Policy {
7.     private final PermissionCollection permissions;
8.
9.     public SamplePolicy(PermissionCollection permissions) {
10.         this.permissions = permissions;
11.     }
12.
13.     @Override
14.     public boolean implies( ProtectionDomain domain,
15.                             Permission permission) {
16.         return permissions.implies(permission);
17.     }
18. }
```

準備ができたら、先ほどのSampleクラスを次のように修正しましょう。この修正では、アクセス許可の集合を扱う**java.security.Permissions**のインス

タンスを作り、**ファイルへのアクセス権**を指定する**java.io.FilePermission**をPermissionsクラスの**addメソッド**で追加します。このFilePermissionクラスのコンストラクタには、ルートディレクトリに対する読み込み権限を許可する旨を指定します。

例 Sampleクラス（アクセス権の設定）

```
1. import java.io.FilePermission;
2. import java.nio.file.Files;
3. import java.nio.file.Paths;
4. import java.security.Permissions;
5. import java.security.Policy;
6.
7. public class Sample {
8.     public static void main(String[] args) throws Exception {
9.         Permissions permissions = new Permissions();
10.        permissions.add(new FilePermission("/", "read"));
11.        Policy.setPolicy(new SamplePolicy(permissions));
12.        System.setSecurityManager(new SecurityManager());
13.
14.        Files.list(Paths.get("/")).forEach((p) -> {
15.            System.out.println(p.getFileName());
16.        });
17.    }
18. }
```

なお、権限の種類にはファイルに対する権限のほか、リフレクションの許可、システム・プロパティへのアクセス、環境変数PATHの読み込み許可など、さまざまな権限に関するものが用意されていますが、本書では割愛します。詳細は、公式ドキュメント[※2]を参照してください。
許可する権限の設定が終われば、再度実行してみましょう。例外は発生せず、最初に実行したときと同様に、ルートディレクトリ内の一覧が表示されるようになったはずです。

7. D

➡ P495

解答6では、Javaのコードでセキュリティポリシーを設定する方法を学びました。ここでは、ファイルで記述する方法について学びましょう。

※2 **セキュリティ開発者ガイド** https://docs.oracle.com/javase/jp/11/security/general-security1.html

まず、前問と同じように、ルートディレクトリ内の一覧を表示するクラスを作りましょう。

例 Libクラス

```
1. package com.lib;
2.
3. import java.io.IOException;
4. import java.nio.file.Files;
5. import java.nio.file.Paths;
6.
7. public class Lib {
8.     public void test() throws Exception {
9.         Files.list(Paths.get("/")).forEach((p) -> {
10.            System.out.println(p.getFileName());
11.        });
12.    }
13. }
```

クラスが準備できたら、jarコマンドでライブラリとしてまとめてしまいます。

例 lib.jarの作成

```
> jar cvf lib.jar com/lib/Lib.class
マニフェストが追加されました
com/lib/Lib.classを追加中です(入=384)(出=260)(32%収縮されました)
```

これで利用される側のライブラリができたので、**セキュリティポリシーファイル**をクラスパス上に作ります。次のsample.policyファイルは、「すべての許可を与える」という設定になっています。どのような処理も許可されます。

例 sample.policyファイル（セキュリティポリシーファイル）

```
1. grant {
2.     permission java.security.AllPermission;
3. };
```

最後に利用する側のクラスを作りましょう。このクラスでは、先ほどのライブラリ（Libクラス）を読み込んで、testメソッドを実行しているだけです。

※次ページに続く

例 ライブラリを利用するSampleクラス

```
1. import com.lib.Lib;
2.
3. public class Sample {
4.     public static void main(String[] args) throws Exception {
5.         Lib lib = new Lib();
6.         lib.test();
7.     }
8. }
```

これですべての準備が整ったので、実行してみましょう。実行時には、**-Dオプション**を付けて**java.security.manager**を有効にし、さらに先ほど用意したセキュリティポリシーファイル（sample.policy）を読み込むように指定しています。実行すれば、セキュリティポリシーですべてを許可しているため、問題なくファイルの一覧が表示されるはずです。

例 実行結果

```
> java Sample -cp .:lib.jar -Djava.security.manager -Djava.sec
urity.policy=sample.policy
home
usr
bin
sbin
… 以下省略 …
```

では、次にライブラリだけに権限を付与するようにセキュリティポリシーを変更してみます。特定のコードやjarファイルだけにセキュリティポリシーを適用するには、**codeBase**で指定します。次の例では、lib.jarだけにルートディレクトリへの読み込み許可を与えています。

例 sample.policyファイル（セキュリティポリシーファイル）

```
1. grant codeBase "file:/Users/shiga/workspace/chap12/lib.jar" {
2.     permission java.io.FilePermission "/", "read";
3. };
```

以上のことから、選択肢**D**が正解です。

これでライブラリはルートディレクトリにアクセスできるようになりました

が、一方でライブラリを利用しているSampleクラスには許可が与えられていないことになります。このように複数のモジュールやライブラリが混在するシステムで、それぞれに**異なる権限**が与えられた場合、セキュリティの観点から**最小限の権限**で実行されます。そのため、先ほどまでは問題なく実行できていたSampleクラスは、権限がないために例外が発生してしまいます。

例 実行結果

```
> java Sample -cp .:lib.jar -Djava.security.manager -Djava.sec
urity.policy=sample.policy
Exception in thread "main" java.security.AccessControlExceptio
n: access denied ("java.io.FilePermission" "/" "read")
    at java.base/java.security.AccessControlContext.checkPerm
ission(AccessControlContext.java:472)
… 以下省略 …
```

しかし、このような状況を改善するために、関連するすべてのモジュールやライブラリに同等の権限を持たせることは、安全性の観点からするべきではありません。そこで、特権ブロックという仕組みを使います。**特権ブロック**は、そのライブラリのコードを、全体を通した低い権限でなく、ライブラリに本来与えられている権限で実行する仕組みです。なお、この特権は、呼び出し先で自分より高い権限で実行するものではなく、**呼び出された側で（呼び出し元とは無関係に）自分自身の権限で実行する**ためのものです。

それでは、特権ブロックを使うように、ライブラリを修正してみましょう。特権ブロックを使うには、**java.security.AccessControllerクラス**の**doPrivilegedメソッド**に、**java.security.PrivilegedActionインタフェース**を実装したクラスを渡します。次のコード例では、PrivilegedActionを実装した無名クラスを使っています。

例 Libクラス（特権ブロックを使うコードの追加）

```
1. package com.lib;
2.
3. import java.io.IOException;
4. import java.nio.file.Files;
5. import java.nio.file.Paths;
6. import java.security.AccessController;
7. import java.security.PrivilegedAction;
8.
9. public class Lib {
```

```
10.     public void test() throws Exception {
11.         AccessController.doPrivileged(new PrivilegedAction<Object>() {
12.             @Override
13.             public Object run() {
14.                 try {
15.                     Files.list(Paths.get("/")).forEach((p) -> {
16.                         System.out.println(p.getFileName());
17.                     });
18.                 } catch (IOException e) {
19.                     throw new RuntimeException(e);
20.                 }
21.                 return null;
22.             }
23.         });
24.     }
25. }
```

PrivilegedActionクラスの**runメソッド**に特権で実行したい処理を記述すれば、セキュリティポリシーに定義された権限で実行することができます。準備ができたら、もう一度jarコマンドでまとめましょう。

例 lib.jarファイルの作成

```
> jar cvf lib.jar com/lib/Lib.class
マニフェストが追加されました
com/lib/Lib.classを追加中です(入=384)(出=260)(32%収縮されました)
```

最後に Dオプションを指定して、実行してみましょう。今度はファイル一覧が表示されるはずです。

試験対策

セキュリティポリシーファイルの設定をはじめとする、セキュリティマネージャの使い方を理解しましょう。

8. E

➡ P496

オブジェクトの「**可変性**」とは、インスタンスの状態（フィールドの値の組み合わせ）が変わることがあるかどうかという性質をいいます。この可変性が、さまざまなセキュリティトの問題の原因となってしまう可能性があります。

フィールドの値が変わることがあるクラスのことを「**可変オブジェクト**（Mutable-Object）」、変わらないクラスのことを「**不変オブジェクト**（Immutable-Object）」と呼びます。フィールドをprivateにして、アクセサメソッドを作って…と、特に何も意識せずにクラスを作ると可変オブジェクトになります。

可変オブジェクトの問題は、オブジェクトを利用したいときにフィールドの値が想定していた値から変わっている可能性があることです。特にマルチスレッド環境では、複数のスレッドで同じインスタンスを共有することがあります。たとえば、あるスレッドAがインスタンスを使って処理を進めている最中に、別のスレッドBがそのインスタンスのフィールドの値を変更してしまったために、スレッドAが想定していなかった結果に陥ることがあります。その結果、当初想定していなかった動作によりcatchされない例外が発生し、システムがダウンするという最悪の事態もあり得るのです。

このような事態にならないために、クラスを設計・定義するときには不変オブジェクトにするという方法があります。不変オブジェクトであれば、フィールドの値が変更できないため、前述のような事態を避けることができるからです。不変オブジェクトを作るには、次のような方法があります。

- フィールドの値を変更するようなsetterメソッドは提供しない
- フィールドの値を変更するのであれば、ディープコピーしたインスタンスを生成して、その参照を戻す
- フィールドはfinalで宣言し、コンストラクタで初期値を設定したら変更できないようにする
- 継承ができないように、クラス宣言をfinalで修飾する
- コンストラクタをアクセス修飾子なし、もしくはprivateで修飾する
- インスタンスを生成するための専用のビルダーを用意する

では、実際に不変オブジェクトを定義してみましょう。次のImmutableTestクラスは、前述のルールに従って不変オブジェクトになっています。不変オブジェクトの特徴は、値の変更をするときに新しい値を持った新しいインスタンスを生成して戻すことです。

例 ImmutableTestクラス（不変オブジェクト）

```
1. package com.sample;
2.
3. // finalで修飾して継承を禁止
```

```
4. public final class ImmutableTest {
5.     // フィールドはfinalで宣言し、変更させない
6.     private final int num;
7.     private final String str;
8.
9.     // 外部のパッケージからはインスタンス化させない
10.    ImmutableTest(int num, String str) {
11.        this.num = num;
12.        this.str = str;
13.    }
14.
15.    // getterは、参照をそのまま戻さない
16.    // 新しいインスタンスを生成してその参照を戻す
17.    public String getStr() {
18.        return new String(str);
19.    }
20.
21.    // setterは、元の値を変更せず
22.    // 新しい値を持った新しいインスタンスを生成して戻す
23.    public ImmutableTest setStr(String str) {
24.        return new ImmutableTest(this.num, str);
25.    }
26. }
```

このクラスのコンストラクタは、アクセス修飾子がありません。そのため、同じパッケージ内のクラスからしかアクセスができず、直接インスタンスを生成できないよう制限をかけています。もしインスタンスを生成したいのであれば、次の**ビルダー**を使います。

例 TestBuilderクラス（ビルダー）

```
1. package com.sample;
2.
3. public class TestBuilder {
4.     private int num;
5.     private String str;
6.
7.     public void setNum(int num) {
8.         this.num = num;
9.     }
10.    public void setStr(String str) {
11.        this.str = str;
12.    }
13.
14.    public ImmutableTest build() {
15.        // 値がセットされていなければデフォルト値をセット
16.        if (this.str == null) {
17.            this.str = "none";
18.        }
19.        return new ImmutableTest(num, str);
20.    }
21. }
```

コンストラクタを使わせず、ビルダーを経由してインスタンスを生成するのは、すべてのフィールドに初期値をセットすることを保証したいからです。コンストラクタで初期化をすると、必要なフィールドの値がnullのままだったり、データ型ごとの初期値のままであったりするので、すべてのフィールドが任意の状態になっていることを保証するためにビルダーを使います。このコード例のビルダーでは、String型のフィールドstrに初期値が設定されていなければ「none」という文字列を設定するようにしています。
以上のことから、選択肢**E**が正解です。

9. A

➡ P496

前問で不変オブジェクトについて学びましたが、すべてのクラスを不変オブジェクトとして設計することは現実的ではありません。そこで、ここでは可変オブジェクトを扱うときの注意点について学びましょう。
まず、**可変オブジェクト**を設計、実装しているとき、getterメソッドで可変オブジェクトへの参照を戻さないようにしましょう。**getterメソッドで参照を戻す**という行為は、**フィールドを公開（public）**にしているのと何ら変わりがないのです。可変オブジェクトへの参照をgetterメソッドで戻すのであれば、**コピーを作って戻す**ようにすると安全です。次のコードは、その一例です。

例 getterメソッドでインスタンスのコピー（クローン）を作って戻す

```
1. import java.util.Date;
2.
3. public class TestCalendar {
4.     private Date date;
```

```
 5.
 6.     public TestCalendar(Date date) {
 7.         this.date = date;
 8.     }
 9.
10.     public Date getDate() {
11.         return (Date) date.clone();
12.     }
13. }
```

このテクニックは、参照を戻すときだけでなく、可変オブジェクトへの参照を受け取るときにも使えます。引数で受け取った可変オブジェクトへの参照をそのままフィールドに代入するのではなく、コピーしてから代入するのです。

例 引数で受け取ったオブジェクトをそのまま代入しない

```
 1. import java.util.Date;
 2.
 3. public class TestCalendar {
 4.     private Date date;
 5.
 6.     public TestCalendar(Date date) {
 7.         this.date = new Date(date.getTime());
 8.     }
 9.
10.     public Date getDate() {
11.         return (Date) date.clone();
12.     }
13. }
```

このとき、「java.util.Dateクラスもcloneメソッドを持っているのだから使えばよいのではないか？」と考える人もいるでしょう。このクラスはfinalなクラスではなく、継承可能なクラスであり、cloneメソッドがオーバーライド可能です。そのため、悪意のあるユーザーが記述した実装で上書きされたとしても、ポリモーフィズムを使っていると、上書きされたことに気がつかないのです。したがって、引数を生成するのが信頼できるソースからのものであるときのみ、cloneメソッドを使うべきでしょう。
もし、可変オブジェクトを「使う」のではなく、自らが可変オブジェクトを「設計」しているのであれば、そのインスタンスの安全なコピーを作成する手段を用意しておきましょう。
たとえば、次のコード例では、今あるインスタンスをコピーするためのコンストラクタを用意しています。このようなコピーすることを目的としたコンストラクタを、「**コピーコンストラクタ**」と呼びます。

例 コピーコンストラクタ

```
 1. import java.util.Date;
 2.
 3. public class TestCalendar {
 4.     private Date date;
 5.
 6.     public TestCalendar(Date date) {
 7.         this.date = new Date(date.getTime());
 8.     }
 9.
10.     public TestCalendar(TestCalendar input) {
11.         this.date = new Date(input.date.getTime());
12.     }
13.
14.     public Date getDate() {
15.         return (Date) date.clone();
16.     }
17.
18.     @Override
19.     public boolean equals(Object obj) {
20.         if (this == obj)
21.             return true;
22.         if (obj == null)
23.             return false;
24.         if (getClass() != obj.getClass())
25.             return false;
26.         TestCalendar other = (TestCalendar) obj;
27.         if (date == null) {
28.             if (other.date != null)
29.                 return false;
30.         } else if (!date.equals(other.date))
31.             return false;
32.         return true;
33.     }
34. }
```

なお、上記のコード例では、コピーコンストラクタを使ったときの比較をするためにequalsメソッドのオーバーライドも追加しています。実際の使用例は次のとおりです。

例 TestCalendarを使うSampleクラス

```
 1. import java.util.Date;
 2.
 3. public class Sample {
 4.     public static void main(String[] args) {
 5.         TestCalendar a = new TestCalendar(new Date());
 6.         TestCalendar b = new TestCalendar(a);
 7.
 8.         System.out.println(a == b);
 9.         System.out.println(a.equals(b));
10.     }
11. }
```

2つのTestCalendarクラスのインスタンスを生成していますが、1つはDate型の引数を受け取るコンストラクタを、もう1つはコピーコンストラクタを利用しています。このコードを実行すると、変数aとbは異なる参照を持っていますが、フィールドの値は等しいことがわかります。
以上のことから、選択肢**A**が正解です。

10. B

➡ P496

アクセシビリティと拡張性に関する問題です。
セキュリティ対策を施すにあたって、コードの攻撃対象を減らすことは大変重要です。攻撃対象が減れば、その分、対策も的確にしやすくなるからです。まず、標準で備わっているアクセス制御の仕組みを適切に使いましょう。「パッケージ内で公開されるインタフェースを限定すること」、「クラスは原則的にアクセス修飾子をなし（デフォルト）にしてパッケージ外には非公開にすること」、「モジュールを使って公開するパッケージを限定すること」です。こうすることで、アクセスできる範囲が狭まり、攻撃者が持ち得る攻撃手段も減らすことができるのです。

また、たとえ可変オブジェクトであったとしても、**finalでクラス宣言を修飾**して継承を制限したほうがよいでしょう。finalでないクラスは、攻撃者によって悪意を持ってオーバーライドされる可能性があるためです。継承ではなく、委譲を使うようにすべきです。
以上のことから、選択肢**B**が誤りです。

IDE（統合開発環境）が広く使われる昨今、意識しなければすべてのクラスがpublicで宣言され、公開されてしまっています。クラスのアクセス修飾子「なし」が本来Javaのデフォルト（標準）であり、publicとして公開されるのは、限定されたごく一部のクラスだけにすべきです。

11. B

➡ P497

SQLインジェクション対策に関する問題です。
よく知られている攻撃手段の1つに**SQLインジェクション**があります。この攻撃手段は、SQLを文字列として扱い、入力された値とパラメータを使ってSQL文をプログラムで組み立てる仕組みを悪用したものです。次の例を見てみましょう。

```
String sql = "select * from test where name = '" + param + "'";
Statement stmt = connection.createStatement();
ResultSet rs = stmt.executeQuery(sql);
```

このとき、変数paramに「allen」という文字列が入っていれば、次のようなSQL文になります。このSQL文には何ら問題はなく、正常に動作するでしょう。

```
String sql = "select * from test where name = 'allen'";
Statement stmt = connection.createStatement();
ResultSet rs = stmt.executeQuery(sql);
```

しかし、攻撃者が「'allen' or a' = 'a」を入力し、それをそのまま変数paramに渡したとしたら、次のようなSQL文ができあがります。

```
String sql = "select * from test where name = 'allen' or 'a' = 'a'";
Statement stmt = connection.createStatement();
ResultSet rs = stmt.executeQuery(sql);
```

このSQL文のwhere句の条件は、name列がallen、もしくは文字「a」が「a」であるときにtrueを戻します。文字「a」が「a」であることは常に正しいので、name列の値にかかわらず、このSQLは全件分のレコードを戻してしまいます。このような攻撃を避けるためには、**java.sql.PreparedStatement**を使用します。第6章で解説したとおり、そもそもjava.sql.Statementはパラメータなしの

SQL文だけを実行するためのインタフェースで、PreparedStatementはパラメータを受け取るSQL文を実行するためのインタフェースです。パラメータを受け取ってSQL文を実行するのであれば、次のようにPreparedStatementを使いましょう。

```
String sql = "SELECT * FROM test WHERE name = ?";
PreparedStatement ps = con.prepareStatement(sql);
ps.setString(1, name);
ResultSet rs = ps.executeQuery();
```

PreparedStatementのパラメータをセットする**setXXX**というメソッドは、必要に応じて自動的に**引用符**を付けてくれます。また、パラメータとして渡されるのは値のみであり、SQL構文などを渡すと**java.sql.SQLException**がスローされます。そのため、前述のような方法を使って不正なSQL文を実行することはできません。

以上のことから、選択肢**B**が正解です。選択肢AのStatementは、パラメータを受け取るSQL文を実行するにはセキュリティ上の懸念があります。

なお、どうしてもPreparedStatementではなく、Statementを使わなければいけないような場合で、かつパラメータを受け取るSQL文を扱うなら、Statementインタフェースの**enquoteIdentifierメソッド**を使いましょう。

```
String sql = "select * from test where name = "
                    + Statement.enquoteIdentifier(param);
Statement stmt = connection.createStatement();
ResultSet rs = stmt.executeQuery(sql);
```

このメソッドは、パラメータの型に応じて自動的に引用符を付けてくれます。

選択肢Cのjava.sql.CallableStatementは、JDBCでストアドプロシージャを実行するためのものです。選択肢Dのようなインタフェースは存在しません。よって、これらは誤りです。

試験対策

SQLインジェクションを防ぐために、PreparedStatementを使うようにしましょう。Statementを使わなければいけない場合は、enquoteIdentifierメソッドを使って不正なSQL文が実行されないようにしましょう。

12. A

➡ P497

メモリ上にあるインスタンスをシリアライズすることで、ファイルなどへの書き出しができることを第5章で学びました。これによってメモリの空きが少なくなったときの一時退避としてファイルに書き出したり、JVMが終了したときにメモリを元の状態に戻したりすることができます。**シリアライズ**はこのように非常に強力な機能ですが、セキュリティの観点からは気を付けるべき点があります。

インスタンスがシリアライズされるとアクセス制御ができなくなり、攻撃者はシリアライズされたバイトデータを解析することで、インスタンスのprivateなフィールドにアクセスすることができます。そのため、機密情報を扱うインスタンスをシリアライズしてはいけません。

もし、機密情報を扱うインスタンスをシリアライズするのであれば、機密情報を保持するフィールドを**transient**で修飾しましょう。この修飾子が付いているフィールドは、シリアライズの対象から除外されます。

例 transient修飾子の使用例

```
1. public class Employee implements Serializable {
2.     private String name;
3.     private transient String telephone; // シリアライズから除外
4.
5. … 省略 …
6.
7. }
```

ただし、transientを指定することによって、フィールドの値は出力されなくなります。逆にいえば、シリアライズした結果をネットワークなどで伝送する場合、その情報は「伝送先に渡さなくても支障がないもの」ということになります。シリアライズされたバイトストリームで機密情報や重要な情報を伝送する場合は、ネットワークあるいはバイトストリームそのものを**暗号化**すべきでしょう。以上のことから、選択肢**A**が正解です。

なお、前述のとおりフィールドにtransientを付けることで出力させないフィールドを指定することができますが、**serialPersistentFieldsフィールド**を定義し、出力してもよいフィールドを指定するという方法もあります。たとえば、前述のEmployeeクラスに、次のようなObjectStreamFieldを扱う配列型変数serialPersistentFieldsを定義し、空の初期化子を使って初期化します。

※次ページに続く

```
private static final ObjectStreamField[] serialPersistentFields = {

};
```

重要なことは、たとえフィールドにtransientが付いていたとしても、serialPersistentFieldsで指定したフィールドはシリアライズ対象となる点です。そのため、このコードをEmployeeクラスに追加すると、transientで修飾されたtelephoneフィールドの値も出力されます。

選択肢Bのstrictfpは、浮動小数点数をIEEE 754という規格で決められたルールで厳密に処理するための修飾子です。選択肢Cのvolatileは、マルチスレッド環境下で、フィールドのキャッシュを抑制してパフォーマンスを向上させるための修飾子です。なお、この修飾子はクラス宣言ではなく、フィールドを修飾するために使用します。選択肢Dのfinalは、継承したサブクラスを作らせないために使用する修飾子です。よって、これらは誤りです。

staticなフィールドもシリアライズ対象にはなりません。これは、staticなフィールドはインスタンスごとに保有するフィールドではないためです。

13. B

➡ P497

デシリアライズをする際、シリアライズされたデータを読み込むInputStreamを悪意のある内容に差し替えられると、本来はあり得ない値を保持したインスタンスを作ることができてしまいます。インスタンスの生成時にどのような値を持つべきかはコンストラクタで制御することができますが、デシリアライズすると、コンストラクタを使わずにインスタンスが生成され、各フィールドが初期化されてしまうためです。

このようなリスクを避けるために利用されるのが、「**シリアライズ・プロキシ**」と呼ばれるデザインパターンです。このパターンは、シリアライズしたいデータを保持する代わりのクラスを用意しておき、対象の代わりにシリアライズし、デシリアライズするときに、本来のシリアライズ対象のインスタンスを生成し、データを戻すことで、インスタンスを復元します。この方法が、通常のデシリアライズと異なるのは、インスタンスを復元するときに、コンストラクタを経由することができる点です。

では、まずシリアライズ・プロキシを利用しない例を見てみましょう。次のコードは、コンストラクタで引数のチェックをしており、インスタンスを生成する際に不正な値がフィールドに入ることを防いでいます。

例 シリアライズ・プロキシを利用しない例

```
1. import java.io.Serializable;
2.
3. public class Item implements Serializable {
4.     private String name;
5.     private int price;
6.
7.     public Item(String name, int price) {
8.         if (name == null || price < 0) {
9.             throw new IllegalArgumentException();
10.        }
11.        this.name = name;
12.        this.price = price;
13.    }
14.
15.    public String getName() {
16.        return name;
17.    }
18.
19.    public int getPrice() {
20.        return price;
21.    }
22.
23.    @Override
24.    public String toString() {
25.        return "Item [name=" + name
26.                + ", price=" + price + "]";
27.    }
28. }
```

しかし、前述のとおりデシリアライズされるときには、コンストラクタは使われません。そのため、このチェックを経由しない悪意のある値をフィールドに入れることが可能です。そこで、シリアライズ・プロキシ・パターンを使って、このようなリスクを低減していきましょう。このパターンでは、本来のシリアライズ対象のクラスの代わりにシリアライズされるstaticインナークラスを使います。
前述のItemクラスに、次のようにstaticインナークラスを追加してみましょう。

このクラスは、エンクロージングクラスであるItemクラスと同じフィールドを持ち、コンストラクタでそれぞれを受け取って初期化します。そのため、このクラスのインスタンスは、Itemクラスのインスタンスと同じ状態（複数のフィールドとその値のセットのこと）を保持することができます。

例 Itemクラスにstaticインナークラスを追加する

```
1. import java.io.InvalidObjectException;
2. import java.io.ObjectInputStream;
3. import java.io.Serializable;
4.
5. public class Item implements Serializable {
6.      … 省略 …
7.     private Object writeReplace() {
8.         System.out.println("writeReplace");
9.         return new ItemSerializationProxy(name, price);
10.     }
11.
12.     private static class ItemSerializationProxy
13.                         implements Serializable {
14.
15.         private final String n;
16.         private final int p;
17.
18.         ItemSerializationProxy(String name, int price) {
19.             this.n = name;
20.             this.p = price;
21.         }
22.
23.         private Object readResolve() {
24.             System.out.println("readResolve");
25.             return new Item(n, p);
26.         }
27.     }
28. }
```

注目してほしいのは、エンクロージングクラスのwriteReplaceメソッドと、staticインナークラスのreadResolveメソッドです。この2つのメソッドは、シリアライズとデシリアライズをするときに自動的に呼び出されるメソッドです。

エンクロージングクラスの**writeReplaceメソッド**は、シリアライズのときに自動的に呼び出され、同じフィールドの値を持ったstaticインナークラスのインスタンスを生成して戻します。JVMは、シリアライズ対象のクラスがこのメソッドを持っていれば、自動的にこのメソッドを呼び出し、その戻り値のインスタンスをシリアライズします。その結果、エンクロージングクラスのインスタンスはシリアライズされず、staticインナークラスのインスタンスがシリアライズされることになります。

staticインナークラスの**readResolveメソッド**は、デシリアライズするときに自動的に呼び出され、復元されたstaticインナークラスのインスタンスと同じフィールドの値を持ったエンクロージングクラスのインスタンスを返します。このとき、エンクロージングクラスのインスタンスはコンストラクタを使って初期化されているため、不正な値がフィールドにセットされることはありません。以上のことから、選択肢Bが正解です。デシリアライズするときに自動的に呼び出されるのは、readResolveメソッドであって、writeReplaceメソッドではありません。よって、選択肢CとDは誤りです。

最後に、次の**readObjectメソッド**をエンクロージングクラスに追加しておきましょう。第5章で学んだように、このメソッドはデシリアライズするときに自動的に呼び出されるメソッドです。

```
private void readObject(ObjectInputStream in)
                throws InvalidObjectException {
    System.out.println("read");
    throw new InvalidObjectException("proxy required");
}
```

シリアライズ・プロキシ・パターンを使っていれば、本来はエンクロージングクラスのインスタンスがシリアライズされることはありません。当然、デシリアライズされることもないはずです。しかし、このシリアライズ・プロキシ・パターンを利用する前にシリアライズしたものが存在したり、攻撃者自身がシリアライズしたものを持ち込んだりする可能性もあり得ます。そのような方法でエンクロージングクラスのインスタンスがシリアライズされ、デシリアライズされることを想定し、readObjectメソッドで例外をスローするようにしています。こうしておけば、不正なデシリアライズを防ぐことができるのです。以上のことから、選択肢Aは誤りです。

13

総仕上げ問題

■試験番号：1Z0-816

■問題数：80問

■試験時間：180分

1. 次のプログラムを確認してください。

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class Sample {
    public static void main(String[] args) {
        try (BufferedReader br
                = new BufferedReader(
                        new InputStreamReader(System.in))) {
            System.out.print(" in : ");
            String input = br.readLine();
            System.out.println(" out: " + input);
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}
```

このプログラムを次のコマンドで実行したときの結果として、正しいものを選びなさい。(1つ選択)

```
> java Sample hello
```

A. プログラム実行中に入力待ちになる
B. in : hello
   out: hello
C. in :
   out:
D. in :
   out: hello
E. 実行時に例外がスローされる

➡ P582

2. 次のコードのうち、ファイルの読み込み後にStreamオブジェクトを取得できるものを選びなさい。(1つ選択)

A. `var a = Files.lines(Paths.get(fileName));`
B. `var b = Files.readAllLines(Paths.get(fileName));`
C. `Stream<String> c = Files.lines(fileName);`
D. `String<String> d = readAllLines(Paths.get(fileName));`

➡ P582

3. 次のTestインタフェースを実装したサービスをロードするために、ServiceLoader<Test>型のサービスローダーloaderを作成した場合、Testのexecute()メソッドを呼び出すにはどのコードを実行すればよいか。正しいものを選びなさい。(1つ選択)

```
public interface Test {
    public void execute();
}
```

A.
```
for (Test test : loader) {
    test.execute();
}
```
B. `loader.execute();`
C.
```
for (Test test : loader.services()) {
    test.execute();
}
```
D.
```
Test test = loader.getTest();
test.execute();
```

➡ P582

4. ストリームに対して、peekメソッドではなく、forEachメソッドを使用するのはどのような場合か。正しいものを選びなさい。(1つ選択)

A. ストリームの先頭から要素を削除する
B. 現在の要素を処理して何も戻さない
C. ストリームの末尾から要素を削除する
D. 現在の要素を処理してストリームを戻す

➡ P583

5. appleの価格の平均を求めたい。次のプログラムの「// insert code here」に入るコードとして、正しいものを選びなさい。(1つ選択)

```
 1. import java.util.List;
 2. import java.util.stream.Collectors;
 3.
 4. public class Sample {
 5.     public static void main(String[] args) {
 6.         List<Item> list = List.of(
 7.                 new Item("apple", 100),
 8.                 new Item("banana", 80),
 9.                 new Item("apple", 200)
10.         );
11.         // insert code here
12.         System.out.println(result);
13.     }
14. }
```

A.
```
double result = list.stream()
    .filter(e -> e.getName().equals("apple"))
    .mapToInt(Item::getPrice)
    .average()
    .getAsDouble();
```

B.
```
double result = list.stream()
    .filter(e -> e.getName().equals("apple"))
    .average()
    .getAsDouble();
```

C.
```
double result = list.stream()
    .filter(e -> e.getName().equals("apple"))
    .filter(s -> s.getPrice())
    .average()
    .getAsDouble();
```

D.
```
double result = list.stream()
    .collect(Collectors.groupingBy(Item::getName,
        Collectors.averagingDouble(Item::getPrice)));
```

➡ P583

6. Comparatorの実装として、正しいものを選びなさい。(1つ選択)

A.
```
public class A implements Comparator {
    @Override
    public boolean compare(Object o1, Object o2) {
        return o1.equals(o2);
    }
}
```

B.
```
public class B implements Comparator {
    public int compare(String str1, String str2) {
        return str1.length() - str2.length();
    }
}
```

C.
```
new Comparator<String> () {
    public int compare(String o1, String o2) {
        return o1.compareTo(o2);
    }
};
```

D.
```
new Comparator<String>() {
    public int compareTo(String o1, String o2) {
        return o1.compareTo(o2);
    }
};
```

➡ P584

7. 次のプログラムを確認し、コンパイルエラーとなるものを選びなさい。(1つ選択)

```
List<Integer> list = List.of(0, 1, 2, 3, 4);
```

A. `Double a = list.get(0);`
B. `double b = list.get(0);`
C. `Integer c = Integer.valueOf(list.get(0));`
D. `Double d = Double.valueOf(list.get(0));`
E. `Integer e = list.get(0);`
F. `int f = list.get(0);`

➡ P584

□ **8.** 次のプログラムをコンパイル、実行したときの結果として、正しいものを選びなさい。(1つ選択)

```
1. public interface Test {
2.     public default void execute(String str) {
3.         System.out.println("A");
4.     }
5. }
```

```
1. public abstract class AbstractTest {
2.     private void execute(String str) {
3.         System.out.println("B");
4.     }
5. }
```

```
1. public class Sample extends AbstractTest implements Test {
2.     public static void main(String[] args) {
3.         new Sample().execute("hello");
4.     }
5. }
```

A. 「A」と表示される
B. 「B」と表示される
C. コンパイルエラーが発生する
D. 実行時に例外がスローされる

➡ P585

□ **9.** 次のインタフェースのうち、戻り値を戻さないメソッドを持つものを選びなさい。(1つ選択)

A. java.util.function.Predicate
B. java.util.function.Function
C. java.util.function.Supplier
D. java.util.function.Consumer

➡ P585

□ **10.** 次のプログラムをコンパイル、実行したときの結果として、正しいものを選びなさい。(1つ選択)

```
1. public class Test implements AutoCloseable {
2.     @Override
3.     public void close() throws Exception {
4.         throw new Exception("A");
5.     }
6. }
```

```
1. public class Sample {
2.     public static void main(String[] args) {
3.         try (Test t = new Test()) {
4.             throw new RuntimeException("B");
5.         } catch (Exception e) {
6.             System.out.println(e.getMessage());
7.         }
8.     }
9. }
```

A. 「A」と表示される
B. 「B」と表示される
C. 何も表示されない
D. Testクラスでコンパイルエラーとなる
E. Sampleクラスでコンパイルエラーとなる

➡ P585

□ **11.** 機密情報の改ざんを防止する方法として、適切なものを選びなさい。(2つ選択)

A. 機密情報を扱うインスタンスから情報が漏洩する可能性を考え、信頼できるオブジェクトのみがアクセスできるようにする
B. 機密情報は、ユーザーが独自に作成したライブラリで処理する
C. セキュリティマネージャを使用する
D. 機密情報は、変更不可能なオブジェクトまたは不変オブジェクトに格納する
E. 機密情報が含まれているインスタンスは、不要になったらできるだけ速やかにメモリから削除する

➡ P586

12. 次のプログラムの変数numをスレッドセーフにするための修正方法として、正しいものを選びなさい。(2つ選択)

```
1. public class Sample {
2.     private int num = 0;
3.     public static void main(String[] args) {
4.         Sample s = new Sample();
5.         ExecutorService service
6.                 = Executors.newFixedThreadPool(10);
7.         for (int i = 0; i < 10; i++) {
8.             service.submit(() -> {
9.                 for (int j = 0; j < 10000; j++) {
10.                    s.num++;                    // line
11.                }
12.            });
13.        }
14.        service.shutdown();
15.    }
16. }
```

A. mainメソッドのメソッド宣言にsynchronizedを追加する

B. lineの位置のコードを、次のコードに置き換える

```
synchronized (s) {
    s.num++;
}
```

C. lineの位置のコードを、次のコードに置き換える

```
synchronized (s.num) {
    s.num++;
}
```

D. numフィールドのフィールド宣言にvolatileを追加する

E. numフィールドのデータ型をAtomicIntegerに置き換え、lineの位置のコードを、次のコードに置き換える

```
s.num.incrementAndGet();
```

➡ P587

13. 次のプログラムを実行したときに、コンソールに出力される結果として、正しいものを選びなさい。(1つ選択)

```
1. import java.time.LocalDate;
2. import java.time.format.DateTimeFormatter;
3. import java.time.format.FormatStyle;
4. import java.util.Locale;
5.
6. public class Sample {
7.     public static void main(String[] args) {
8.         Locale l = new Locale("en", "US");
9.         LocalDate today = LocalDate.of(2021, 4, 1);
10.        String mToday = today.format(
11.                DateTimeFormatter.ofLocalizedDate(
12.                        FormatStyle.MEDIUM).withLocale(l));
13.        String aToday = today.format(
14.                DateTimeFormatter.ofLocalizedDate(
15.                        FormatStyle.SHORT).withLocale(l));
16.        System.out.println(mToday);
17.        System.out.println(aToday);
18.    }
19. }
```

A. 4/1/21
Apr 1, 2021

B. Apr 1, 2021
4/1/21

C. Thursday, April 1, 2021
April 1, 2021

D. April 1, 2021
4/1/21

➡ P588

14. 以下に示すプログラムの「/* insert code here */」に記述するコードとして、適切なのはAとBのどちらか。その理由と共に正しい記述を選択肢より選びなさい。（1つ選択）

```
A："SELECT * FROM TABLE WHERE NAME =" + name
B："SELECT * FROM TABLE WHERE NAME ="
                + stmt.enquoteIdentifier(name, true);
```

```
 1. import java.sql.Connection;
 2. import java.sql.SQLException;
 3. import java.sql.Statement;
 4.
 5. public class Sample {
 6.     public void execute(Connection con, String name)
 7.                                 throws SQLException {
 8.         Statement stmt = con.createStatement();
 9.         String sql = /* insert code here */;
10.         stmt.execute(sql);
11.     }
12. }
```

A. B：呼び出し側のコードが提供する値はすべて引用符で閉じる必要がある
B. A：呼び出し側のコードが提供するnameの値が正確に送られる
C. B：呼び出し側のコードが提供する値を引用符で閉じるとSQLインジェクションが防止される
D. AとBは機能的に等価である
E. 識別子を引用符で閉じる必要はない

➡ P588

15. try-with-resources文で扱うクラスに関する説明として、正しいものを選びなさい。（1つ選択）

A. AutoCloseableを実装し、autoCloseメソッドをオーバーライドする
B. AutoCloseableを継承し、closeメソッドをオーバーライドする
C. AutoCloseableを継承し、autoCloseメソッドをオーバーライドする
D. AutoCloseableを実装し、closeメソッドをオーバーライドする

➡ P588

16. 次のプログラムをコンパイル、実行したときの結果として、正しいものを選びなさい。（1つ選択）

```
 1. import java.util.Arrays;
 2. public class Sample {
 3.     private String name;
 4.     private int num;
 5.
 6.     public Sample(String name, int num) {
 7.         this.name = name;
 8.         this.num = num;
 9.     }
10.
11.     public int getNum() {
12.         return this.num;
13.     }
14.
15.     public static void main(String[] args) {
16.         var samples = Arrays.asList(
17.                     new Sample("A", 8),
18.                     new Sample("B", 21),
19.                     new Sample("C", 23),
20.                     new Sample("D", 12)
21.                 );
21.
22.         int result = samples.stream()
24.                 .mapToInt(Sample::getNum)
25.                 .filter(s -> s < 20)
26.                 .reduce(0, (a, b) -> a + b);
27.
28.         System.out.println(result);
29.     }
30. }
```

A. 「44」と表示される
B. 「20」と表示される
C. 「35」と表示される
D. 「29」と表示される
E. コンパイルエラーが発生する

➡ P589

17. 次のプログラムはコンパイルエラーとなる。コンパイルエラーの解消方法として、正しいものを選びなさい。(2つ選択)

```
 1. import java.util.List;
 2. import java.util.Optional;
 3.
 4. public class Sample {
 5.     public static void main(String[] args) {
 6.         List<Item> items = List.of(
 7.                     new Item("A", 100),
 8.                     new Item("B", 200),
 9.                     new Item("C", 300)
10.                 );
11.
12.         Stream itemStream = items.stream();
13.         itemStream.filter(item -> item.getPrice() > 200)
14.                     .forEach(System.out::println);
15.     }
16. }
```

A. itemStreamのfilterメソッドに渡す引数を、「item -> ((Item) item).getPrice() > 200」に置き換える

B. itemStreamのfilterメソッドに渡す引数を、「(Item item) -> item.getPrice() > 200」に置き換える

C. 変数itemStreamのデータ型をStreamからStream<Item>に置き換える

D. List<Item> itemsを初期化するために、「List.stream();」の結果を代入する

➡ P589

18. 次のようなコンソールで動作するアプリケーションを開発している。

```
 1. import java.io.Console;
 2.
 3. public class Sample {
 4.     public static void main(String[] args) {
 5.         Console console = System.console();
 6.         var name = console.readLine("name > ");
 7.         var pass = console.readPassword("pass > ");
 8.         System.out.println("name : " + name + ", pass : "
 9.                                     + String.valueOf(pass));
10.     }
11. }
```

ユーザーとのやり取り、および結果の表示として、正しいものを選びなさい。なお、nameにはsample、passにはpasswordと入力しているものとする。(1つ選択)

A.
```
name > sample
pass > password
name : sample, pass : password
```

B.
```
name > sample
pass > password
name : sample, pass : [C@4f6abcdef
```

C.
```
name > sample
pass >
name : sample, pass : password
```

D.
```
name > sample
pass >
name : sample, pass :
```

➡ P589

19. 次のプログラムをコンパイル、実行したときの結果として、正しいものを選びなさい。(1つ選択)

```
1. import java.util.HashSet;
2. import java.util.LinkedList;
3. import java.util.List;
4. import java.util.Set;
5.
6. public class Sample {
7.     public static void main(String[] args) {
8.         List<String> list = new LinkedList<>();
9.         Set<String> set = new HashSet<>();
10.        String[] array = {"a", "b", "c", "d", "a"};
11.        for(String str : array) {
12.            list.add(str);
13.            set.add(str);
14.        }
15.        System.out.print(set.size() + "," + list.size() + ",");
16.        Set set2 = new HashSet<>(list);
17.        List list2 = new LinkedList<>(set);
18.        System.out.print(set2.size() + "," + list2.size());
19.    }
20. }
```

A. 「5, 5, 4, 4」と表示される
B. 「4, 5, 4, 4」と表示される
C. 「4, 5, 4, 5」と表示される
D. 「4, 4, 4, 4」と表示される

➡ P590

20. 次のコードのうち、正しく記述されているものを選びなさい。(1つ選択)

A. `Runnable a = {System.out.println("hello.")};`
B. `Runnable b = "hello" -> System.out.println();`
C. `Runnable c = () -> {System.out.println("hello");};`
D. `Runnable d = () -> System.out::print;`
E. `Runnable e = -> System.out.println("hello.");`

➡ P590

21. 次のプログラムをコンパイル、実行したときの結果として、正しいものを選びなさい。(1つ選択)

```
1. import java.util.List;
2. import java.util.Optional;
3.
4. public class Sample {
5.     public static void main(String[] args) {
6.         var list = List.of("apple", "banana", "orange", "melon");
7.         Optional<String> result = list.stream()
8.                                     .filter(x -> x.contains("z"))
9.                                     .reduce((i, j) -> i + "," + j);
10.        result.ifPresent(System.out::print);
11.    }
12. }
```

A. 常にコンソールへの表示が実行される
B. 結果が存在しない場合、実行時にNoSuchElementExceptionが発生する
C. 結果が存在しない場合、実行時にNullPointerExceptionが発生する
D. 結果が存在しない場合、コンソールへの表示は実行されない

➡ P591

22. 次のコマンドの実行結果に関する説明として、正しいものを選びなさい。(1つ選択)

```
> jdeps -jdkinternals C:¥workspace¥test¥jar¥lib.jar
```

A. -jdkinternalsオプションの結果をコンソールに表示するには、-summaryオプションか-verboseオプションが必要であるというエラーを表示する
B. モジュールの依存関係と、参照されるJDKの内部APIすべてのパッケージ名のリストを表示する
C. -jdkinternalsオプションにより、JDKの内部APIのクラス同士の依存関係が、jar内のすべてのクラスについて分析される
D. -jdkinternalsオプションにより、jar内のすべてのクラスが分析され、クラス同士の依存関係がすべて出力される

➡ P591

23. 次のプログラムをコンパイル、実行したときの結果として、正しいものを選びなさい。(1つ選択)

```
1. import java.util.stream.IntStream;
2. import java.util.stream.Stream;
3.
4. public class Sample {
5.     public static void main(String[] args) {
6.         int array[][] = {{1, 2}, {3, 4}, {5, 6}};
7.         long count = Stream.of(array)
8.                 .flatMapToInt(IntStream::of)
9.                 .map(n -> n * 2)
10.                .peek(System.out::print)
11.                .filter(n -> (n % 3 == 0))
12.                .count();
13.        System.out.println(" " + count);
14.    }
15. }
```

A. 「2」と表示される
B. 「24681012 2」と表示される
C. 「2610 2」と表示される
D. 「4812 2」と表示される
E. コンパイルエラーが発生する

➡ P591

24. Java SEセキュアコーディングガイドラインに沿って、機密情報を保護する内容として、正しいものを選びなさい。(3つ選択)

A. 機密情報をカプセル化する
B. 障害時の解析を効率化するため、情報を透過的に処理する
C. 機密情報を明確に識別する
D. ユーザーからの入力を通常の情報として扱う
E. 機密情報を格納する前に入力を検証する
F. 機密情報を保持するオブジェクトへのアクセスを制限する
G. 機密情報をほかの情報と同じように管理する

➡ P592

25. 次のプログラムはコンパイルエラーとなる。コンパイルエラーを解消する方法として、正しいものを選びなさい。(1つ選択)

```
1. import java.io.Serializable;
2.
3. enum Type implements Serializable {
4.     A(1), B(2), C(3);
5.     int val;
6.     public Type(int val) {
7.         this.val = val;
8.     }
9. }
```

A. valフィールドをprivate finalで修飾する
B. コンストラクタのアクセス修飾子をpublicからprivateに変更する
C. valフィールドをprivateで修飾する
D. Serializableインタフェースの実装をやめる
E. Enum宣言のアクセス修飾子を「なし」からpublicに変更する

➡ P592

26. インタフェースと抽象クラスの違いに関する説明として、正しいものを選びなさい。(2つ選択)

A. インタフェースにはインスタンスフィールドを定義できないが、抽象クラスには定義できる
B. インタフェースにはstaticメソッドを定義できないが、抽象クラスには定義できる
C. インタフェースにはprotectedなメソッドを定義できないが、抽象クラスには定義できる
D. finalメソッドは、インタフェースにも抽象クラスにも定義できる
E. 実装があるメソッドは、インタフェースには定義できないが、抽象クラスには定義できる

➡ P592

27. 次のプログラムの「here」の行を実行したときの状態に関する説明として、正しいものを選びなさい。(1つ選択)

```
 1. import java.io.FileReader;
 2. import java.io.IOException;
 3. import java.io.Reader;
 4.
 5. public class Sample {
 6.     public static void main(String[] args) {
 7.         try (   Reader a = new FileReader("sample1.txt");
 8.                 Reader b = new FileReader("sample2.txt");
 9.                 Reader c = new FileReader("sample3.txt")) {
10.         } catch (IOException e) {
11.             // do something
12.         }
13.
14.         // here
15.     }
16. }
```

A. コンパイルエラーが発生する
B. aのみが閉じられる
C. 3つのReaderオブジェクトがすべて閉じられる
D. 3つのReaderオブジェクトはすべて開いたままである
E. 実行時に例外がスローされる

➡ P593

28. 次のプログラムをコンパイル、実行したときの結果として、正しいものを選びなさい。(1つ選択)

```
Path p = Paths.get("a/b/c/../../d");
System.out.println(p.normalize());
```

A. 「a/d」と表示される
B. 「a/b/c/././d」と表示される
C. 「a/b/c/../../d」と表示される
D. 「a/b/c/d」と表示される

➡ P593

29. 次のプログラムをコンパイル、実行したときの結果として、正しいものを選びなさい。(1つ選択)

```
1. public class SampleException extends Exception {
2.     public SampleException(String message) {
3.         super(message);
4.     }
5. }
```

```
1. public class TestException extends SampleException {
2.     public TestException(String message) {
3.         super(message);
4.     }
5. }
```

```
1. public class Sample {
2.     private static void test() throws TestException {
3.         try {
4.             // do something
5.             throw new TestException("A");
6.         } catch (SampleException | RuntimeException e) {
7.             throw new RuntimeException("B");
8.         }
9.     }
10.
11.    public static void main(String[] args) {
12.        try {
13.            test();
14.        } catch (Exception e) {
15.            System.out.println(e.getMessage());
16.        }
17.    }
18. }
```

A. 「A」と表示される
B. 「B」と表示される
C. 何も表示されない
D. コンパイルエラーが発生する

➡ P594

30. sample.txtが存在しない場合、次のプログラムをコンパイル、実行したときの結果として正しいものを選びなさい。(1つ選択)

```
 1. import java.io.IOException;
 2. import java.nio.file.Files;
 3. import java.nio.file.Path;
 4. import java.nio.file.Paths;
 5.
 6. public class Sample {
 7.     public static void main(String[] args) {
 8.         try {
 9.             Path path = Paths.get("sample.txt");
10.             boolean result = Files.deleteIfExists(path);
11.             if (result) {
12.                 System.out.println("A");
13.             } else {
14.                 System.out.println("B");
15.             }
16.         } catch (IOException e) {
17.             System.out.println("C");
18.         }
19.     }
20. }
```

A. 「A」と表示される
B. 「B」と表示される
C. 「C」と表示される
D. コンパイルエラーが発生する
E. 実行時に例外がスローされる

➡ P594

31. モジュールを定義している。次のモジュール定義ファイルに関する説明として、正しいものを選びなさい。(2つ選択)

```
module Sample {
    exports test;
    uses test.Hello;
}
```

A. コンパイルエラーを発生させないためには、test.Helloを実装するモジュールが1つ以上モジュールパスに含まれている必要がある
B. エラーを発生させずに実行するには、test.Helloを実装するモジュールが、1つ以上モジュールパスに含まれている必要がある
C. test.Helloの実装をSampleモジュールに追加できる
D. test.Helloを実装するモジュールは、モジュール定義ファイルに「requires Sample;」というディレクティブを含める必要がある
E. test.Helloを実装するモジュールは、Sampleモジュールを再コンパイルせずにコンパイルおよび追加できる

➡ P594

32. 次のプログラムに関する説明として、正しいものを選びなさい。(1つ選択)

```
Stream<Integer> a = Stream.of(1,2,3,4,5);
IntStream b = a.mapToInt(n -> n);
DoubleStream c = b.mapToDouble(n -> n);
Stream<Integer> d = c.mapToInt(n -> n);
```

A. 1行目でコンパイルエラーとなる
B. 2行目でコンパイルエラーとなる
C. 3行目でコンパイルエラーとなる
D. 4行目でコンパイルエラーとなる

➡ P595

33. 次のプログラムに関する説明として、正しいものを選びなさい。（1つ選択）

```
1. public class Test {
2.     public synchronized void execute(Sample... samples) {
3.         for (int i = 0; i < 10; i++) {
4.             while (!samples[0].hello(this)) {
5.                 // do something
6.             }
7.             while (!samples[1].hello(this)) {
8.                 // do something
9.             }
10.            samples[1].bye();
11.            samples[0].bye();
12.        }
13.    }
14. }
```

```
1. public class Sample {
2.     public Test test;
3.     public synchronized boolean hello(Test test) {
4.         if (this.test == null) {
5.             this.test = test;
6.             return true;
7.         }
8.         return false;
9.     }
10.    public synchronized void bye() {
11.        test = null;
12.    }
13. }
```

```
1. public class Main {
2.     public static void main(String[] args) {
3.         Test t1 = new Test();
4.         Test t2 = new Test();
5.
6.         Sample s1 = new Sample();
7.         Sample s2 = new Sample();
8.
9.         new Thread(() -> {
10.            t1.execute(s1, s2);
11.        }).start();
12.
13.        new Thread(() -> {
14.            t2.execute(s1, s2);
15.        }).start();
16.    }
17. }
```

A. デッドロックの可能性がある
B. ライブロックの可能性がある
C. IllegalMonitorStateExceptionがスローされる
D. コンパイルエラーが発生する

➡ P595

34. 次のプログラムをコンパイル、実行したときの結果として、正しいものを選びなさい。（1つ選択）

```
1. import java.util.ArrayList;
2. public class Sample {
3.     public static void main(String[] args) {
4.         var list = new ArrayList<>();
5.         list.add("A");
6.         list.add(100);
7.         list.add("B");
8.         list.set(1, 200);
9.         list.remove(2);
10.        list.set(3, 300);
11.        System.out.println(list);
12.    }
13. }
```

A. [B, 300]が表示される
B. [A, 200, null, 300]が表示される
C. [A, 100, B]が表示される
D. 実行時に例外がスローされる

➡ P596

35. 次のプログラムをDoS攻撃から保護するための方法として、正しいものを選びなさい。(1つ選択)

```
 1. import java.io.BufferedReader;
 2. import java.io.File;
 3. import java.io.FileReader;
 4. import java.io.IOException;
 5. import java.io.Reader;
 6. import java.util.ArrayList;
 7. import java.util.List;
 8.
 9. public class Sample {
10.     public static void main(String[] args) {
11.         List<Reader> list = new ArrayList<>();
12.         File file = new File("sample.txt");
13.
14.         try (BufferedReader br =
                    new BufferedReader(new FileReader(file))) {
15.             for (String fileName = br.readLine();
16.                 fileName != null; fileName = br.readLine()) {
17.                 BufferedReader br2
18.                     = new BufferedReader(new
                            FileReader(fileName));
19.                 list.add(br2);
20.                 process(br2);
21.             }
22.         } catch (IOException e) {
23.         } finally {
24.             for (Reader r : list) {
25.                 try {
26.                     r.close();
27.                 } catch (IOException ex) {
28.                     // do something
29.                 }
30.             }
31.         }
32.     }
33.
34.     private static void process(BufferedReader br2) {
35.         // do something
36.     }
37. }
```

A. br2を生成するタイミングで、try-with-resourcesを使う
B. processメソッドの実行後に、br2.closeメソッドを呼び出す
C. processメソッドの呼び出し時に、try-with-resourcesを使う
D. br2を生成する前に、brのサイズをチェックして、容量をオーバーしないかを確認する
E. finallyブロックに、「br.close();」を記述する

➡ P596

36. 次のプログラムをコンパイル、実行したときの結果として、正しいものを選びなさい。(1つ選択)

```
 1. import java.util.ArrayList;
 2. import java.util.List;
 3.
 4. public class Sample {
 5.     public static void main(String[] args) {
 6.         List<Integer> a = new ArrayList<>();
 7.         a.add(1);
 8.         a.add(2);
 9.         List b = List.copyOf(a);
10.         b.add(3);
11.         List<List<Integer>> c = List.of(a, b);
12.         System.out.println(c);
13.     }
14. }
```

A. [[1, 2, 3], [1, 2, 3]]が表示される
B. [[1, 2], [1, 2]]が表示される
C. [[1, 2], [1, 2, 3]]が表示される
D. コンパイルエラーが発生する
E. 実行時に例外がスローされる

➡ P597

37. 次のプログラムを実行し、コンソールに「2」と表示したい。「// insert code here」に記述するコードとして、正しいものを選びなさい。（1つ選択）

```
1. public enum Type {
2.     A(1), B(2), C(3);
3.     int num;
4.     Type(int num) {
5.         this.num = num;
6.     }
7.     // insert code here
8. }
```

```
1. public class Sample {
2.     public static void main(String[] args) {
3.         System.out.println(Type.values()[1]);
4.     }
5. }
```

A.
```
public String toString() {
    return String.valueOf(Type.values()[1]);
}
```

B.
```
String toString() {
    return "2";
}
```

C.
```
public String toString() {
    return String.valueOf(A.num);
}
```

D.
```
public String toString() {
    return String.valueOf(num);
}
```

➡ P597

38. 次のコードのうち、正しく記述されているものを選びなさい。（2つ選択）

A. `List<? super Number> a = new ArrayList<Integer>();`
B. `List<? super Float> b = new ArrayList<Double>();`
C. `List<? super Short> c = new ArrayList<Number>();`
D. `List<? extends Number> d = new ArrayList<Byte>();`
E. `List<? extends Number> e = new ArrayList<Object>();`

➡ P598

39. 次のクラスをスレッドセーフにするためには、何をするべきか。正しいものを選びなさい。（1つ選択）

```
1. import java.util.concurrent.locks.ReentrantLock;
2.
3. public class Sample {
4.     private final ReentrantLock lock = new ReentrantLock();
5.     public void test() throws Exception {
6.         try {
7.             lock.lock();
8.             // do something
9.         } finally {
10.            lock.unlock();
11.        }
12.    }
13. }
```

A. lockフィールドの宣言をtestメソッド内に移動する
B. ReentrantLockのコンストラクタにtrueを渡す
C. lockフィールドの宣言をstaticにする
D. 変更は必要ない

➡ P598

40. 次のプログラムをコンパイル、実行したときの結果として、正しいものを選びなさい。なお、変数dsはjavax.sql.DataSource型のオブジェクトであり、TEST表が正しく定義されているものとする。(1つ選択)

```
 1. import java.sql.Connection;
 2. import java.sql.PreparedStatement;
 3. import javax.sql.DataSource;
 4.
 5. public class Sample {
 6.     public static void main(String[] args) throws Exception {
 7.
 8.         // do something
 9.
10.         try (Connection con = ds.getConnection();
11.                 PreparedStatement ps =
12.                     con.prepareStatement
13.                         ("INSERT INTO TEST VALUES (?,?)")) {
14.             ps.setInt(1, 100);
15.             ps.setString(2, "SAMPLE");
16.             ps.executeUpdate();
17.             ps.setInt(1, 101);
18.             ps.executeUpdate();
19.         }
20.     }
21. }
```

A. 「100, "SAMPLE"」および「101, NULL」という2つの行が挿入される
B. 「100, "SAMPLE"」という1つの行が挿入される
C. 「100, "SAMPLE"」および「101, "SAMPLE"」という2つの行が挿入される
D. 実行時に例外がスローされる
E. コンパイルエラーが発生する

➡ P599

41. JDBCを利用してデータベースに接続するためのURLとして、正しいものを選びなさい。(1つ選択)

A. jdbc:postgresql://localhost:5432/test
B. http://localhost.postgresql.com::5432/test
C. http://localhost.postgresql.jdbc:5432/test
D. jdbc.postgresql.com://localhost:5432/test

➡ P600

42. 次のうち、オートボクシングが成功するコードを選びなさい。(2つ選択)

A. `Float a = 1.0;`
B. `Long b = 10L;`
C. `Integer c = 10;`
D. `Double d = 10;`

➡ P600

43. 商品リストから、商品の種類がいくつあるかを出力したい。プログラムに追加するコードとして、正しいものを選びなさい。(1つ選択)

A.
```
long count = list.stream()
            .filter(Item::getType)
            .distinct()
            .count();
```

B.
```
long count = list.stream()
            .map(e -> e.getType())
            .collect(Collectors.toSet())
            .count();
```

C.
```
long count = list.stream()
            .map(e -> e.getType())
            .count();
```

D.
```
long count = list.stream()
            .map(Item::getType)
            .distinct()
            .count();
```

➡ P601

44. 以下の「/* insert code here */」に入るコードとして、正しいものを選びなさい。（1つ選択）

```
List<Integer> list = Arrays.asList(1, 2, 3, 4, 5);
UnaryOperator<Integer> op = /* insert code here */;
list.replaceAll(op);
list.stream().forEach(System.out::println);
```

A. `a, b -> a * b`
B. `(int a) -> a * 2`
C. `var a -> a * 2`
D. `a -> a`

➡ P601

45. 10個の乱数を出力するコードとして、正しいものを選びなさい。（1つ選択）

A.
```
IntStream.generate(Random.nextInt)
        .limit(10)
        .forEach(System.out::println);
```

B.
```
var r = new Random();
IntStream.generate(r::nextInt)
        .limit(10)
        .forEach(System.out::println);
```

C.
```
IntStream.generate(Random::nextInt)
        .limit(10)
        .forEach(System.out::println);
```

D.
```
var r = new Random();
new IntStream.generate(r::nextInt)
        .limit(10)
        .forEach(System.out::println);
```

➡ P602

46. 数値のリストの平均を求めるコードとして、正しいものを選びなさい。（2つ選択）

A.
```
double a = list.stream()
        .collect(Collectors.averagingDouble(n -> n));
```

B.
```
double b = list.parallelStream()
        .mapToInt(m -> m).average().getAsDouble();
```

C.
```
double c = list.stream()
        .average().getAsDouble();
```

D.
```
double d = list.stream()
        .parallel().averagingDouble(n -> n);
```

E.
```
double e = list.stream()
        .mapToInt(n -> n).average().parallel();
```

➡ P602

47. 次のリストがあるとき、このリスト内から「G」が含まれる最初の要素を調べるコードとして、正しいものを選びなさい。（1つ選択）

```
var alphabet = List.of("ABC", "DEF", "GHI", "JKL");
```

A.
```
Optional<String> result
        = alphabet.stream().anyMatch(a -> a.contains("G"));
```

B.
```
alphabet.stream()
        .filter(a -> a.contains("G"))
        .forEachOrdered(System.out::println);
```

C.
```
Optional<String> result
        = alphabet.stream().filter(a -> a.contains("G"));
```

D.
```
String result = alphabet
        .stream()
        .filter(a -> a.contains("G")).findAny();
```

➡ P603

48. 次のプログラムをコンパイル、実行したときの結果として、正しいものを選びなさい。(1つ選択)

```
1. public class Item {
2.     private String name;
3.     private int price;
4.     public Item(String name, int price) {
5.         super();
6.         this.name = name;
7.         this.price = price;
8.     }
9.     public String getName() {
10.        return name;
11.    }
12.    public int getPrice() {
13.        return price;
14.    }
15. }
```

```
1. import java.util.ArrayList;
2. import java.util.Comparator;
3. import java.util.List;
4.
5. public class Sample {
6.     public static void main(String[] args) {
7.         List<Item> list =
8.                 new ArrayList<>(List.of(
9.                         new Item("banana", 100),
10.                        new Item("apple", 80),
11.                        new Item("orange", 80)));
12.
13.        list.sort(
14.                Comparator.comparing(Item::getPrice)
15.                .thenComparing(Item::getName)
16.                .reversed());
17.
18.        list.forEach(item ->
19.            System.out.print(" " + item.getName()));
20.    }
21. }
```

A. 「banana orange apple」と表示される
B. 「apple orange banana」と表示される
C. 「apple banana orange」と表示される
D. 「banana apple orange」と表示される

➡ P603

49. 次のプログラムを確認してください。

```
var exe = Executors.newFixedThreadPool(5);
Future f = exe.submit(() -> 10);
```

submitメソッドの引数に渡すラムダ式は、どの関数型インタフェースで扱うことができるか。正しいものを選びなさい。(1つ選択)

A. java.util.concurrent.Callable
B. java.lang.Runnable
C. java.util.function.Function
D. java.util.function.Predicate

➡ P604

50. 次のプログラムをコンパイル、実行したときの結果として、正しいものを選びなさい。(1つ選択)

```
IntStream stream = IntStream.iterate(0, a -> a + 1);
stream.limit(5)
    .mapToObj(x -> x + ",")
    .forEach(System.out::print);
```

A. 「1,2,3,4,5」と表示される
B. 「0,1,2,3,4」と表示される
C. 「0,1,2,3,4,」と表示される
D. 「1,2,3,4,5,」と表示される
E. コンパイルエラーが発生する

➡ P604

51. 以下のパッケージが含まれるjarファイルを使用するJava 8ベースのアプリケーションがある。

- com.sample.test
- com.sample.hello
- com.sample.boo

このjarファイルをモジュール化して、test、hello、booという3つのモジュールにするためには、どのようにモジュール定義ファイルを記述すればよいか。正しいものを選びなさい。（1つ選択）

A.
```
module com.sample.test {
    provides com.sample.test;
}
module com.sample.hello {
    provides com.sample.hello;
}
module com.sample.boo {
    provides com.sample.boo;
}
```

B.
```
module com.sample.test {
    opens com.sample.test;
}
module com.sample.hello {
    opens com.sample.hello;
}
module com.sample.boo {
    opens com.sample.boo;
}
```

C.
```
module com.sample.test {
    requires com.sample.test;
}
module com.sample.hello {
    requires com.sample.hello;
}
module com.sample.boo {
    requires com.sample.boo;
}
```

D.
```
module com.sample.test {
    exports com.sample.test;
}
module com.sample.hello {
    exports com.sample.hello;
}
module com.sample.boo {
    exports com.sample.boo;
}
```

➡ P604

52. 次のプログラムはコンパイルエラーとなる。コンパイルエラーを解消するための方法として、正しいものを選びなさい。（2つ選択）

```
1. public class Sample {
2.     public static void main(String[] args) {
3.         var subject = new Subject("A");
4.     }
5.     class Subject {
6.         String name;
7.         Subject(String name) {
8.             this.name = name;
9.         }
10.    }
11. }
```

A. Subjectクラスの宣言をstaticで修飾する
B. Subjectクラスの宣言自体を、インナークラスではなく別のクラスとして定義する
C. mainメソッド内の「var subject」を「Subject subject」に変更する
D. Subjectクラスのコンストラクタをpublicにする
E. Subjectクラスをpublicで宣言する

➡ P604

53. 次のプログラムを確認してください。

```
1. public class Order {
2.     private String item;
3.     private int price;
4.     public Order(String item, int price) {
5.         super();
6.         this.item = item;
7.         this.price = price;
8.     }
9.     public String getItem() {
10.        return item;
11.    }
12.    public int getPrice() {
13.        return price;
14.    }
15. }
```

```
List<Order> list = List.of(
        new Order("apple", 100),
        new Order("banana", 80));
BinaryOperator<Integer> op = (a, b) -> a + b;
int total = list.stream()
            .map(o -> o.getPrice() * 10)
            .reduce(0, op);
```

totalを導き出すコードと同じ結果になるものを選びなさい。(1つ選択)

A.
```
int result = list.stream()
                .mapToInt(o -> o.getPrice() * 10)
                .sum();
```

B.
```
int result = list.stream()
                .map(Order::getPrice * 10)
                .reduce(op)
                .orElse(0);
```

C.
```
int result = list.stream()
                .map(o -> o.getPrice() * 10)
                .reduce(op)
                .ifPresent(p -> p.intValue());
```

D.
```
int result = list.stream()
                .mapToInt(o -> o.getPrice() * 10)
                .reduce(0, op);
```

➡ P605

54. 次のコードのうち、正しく記述されているものを選びなさい。(1つ選択)

A. `Consumer a = msg -> System.out::println;`
B. `Consumer b = (String msg) -> System.out.print(msg);`
C. `Consumer c = System.out::print;`
D. `Consumer d = var msg -> {System.out.println(msg);};`

➡ P606

55. 次のプログラムの実行結果として「2021年 4月 1日　木曜日」と表示したい。以下の「/* insert code here */」に入るコードとして、正しいものを選びなさい。(1つ選択)

```
LocalDate date = LocalDate.of(2021, 4, 1);
DateTimeFormatter formatter =
    DateTimeFormatter.ofPattern(/* insert code here */);
System.out.println(formatter.format(date));
```

A. `"yyyy"+年 + "MMM dd+"日"+eeee"`
B. `"yyyy"+年 + "MMMM d+"日"+eeee"`
C. `"yyyy'年' MMMM d'日' eeee"`
D. `"yyyy'年' MMM dd'日' eeee"`

➡ P606

56. 次のプログラムを確認してください。

```
1. public enum Type {
2.     A(1), B(2), C(3);
3.     int val;
4.     private Type(int val) {
5.         this.val = val;
6.     }
7. }
```

以下の「/* insert code here */」に入るコードとして、正しいものを選びなさい。(1つ選択)

```
1. public class Sample {
2.     public static void main(String[] args) {
3.         Type type = Type.A;
4.
5.         switch(type) {
6.             case /* insert code here */:
7.                 System.out.println("A");
8.                 break;
9.             default:
10.                System.out.println("B");
11.                break;
12.        }
13.    }
14. }
```

A. Type.A.valueOf()
B. A
C. Type.A
D. A.toString()

➡ P606

57. デフォルトのロケールを設定するコードとして、正しいものを選びなさい。(2つ選択)

A. Locale.setDefault(Locale.Category.FORMAT, Locale.JAPANESE);
B. Locale.setDefault("es", Locale.US);
C. Locale.setDefault(Locale.JAPANESE);
D. Locale.setDefault(Locale.Category.FORMAT, "ja_JP");
E. Locale.setDefault("ja_JP");

➡ P607

58. Systemクラスに関する説明として、正しいものを選びなさい。(1つ選択)

A. System.inは、標準入力ストリームである。このストリームは常に開いている
B. System.inにほかのストリームを置き換えることはできない
C. デフォルトでは、System.outはjava.io.OutputStreamのインスタンスである
D. System.outは標準出力ストリームである。このストリームは、System.outを呼び出した場合のみ開く

➡ P607

59. 次のコードの実行結果に関する説明として、正しいものを選びなさい。(2つ選択)

```
List<Integer> list = List.of(5,4,3,2,1);
list.parallelStream().forEach(n -> System.out.print(n + " "));
```

A. forEachメソッドをforEachOrderedメソッドに置き換えると「1 2 3 4 5」が出力される
B. 出力される数字の順番は毎回変わる
C. forEachメソッドをforEachOrderedメソッドに置き換えると「5 4 3 2 1」のように出力されるが、順番は毎回変わる
D. forEachメソッドをforEachOrderedメソッドに置き換えると「5 4 3 2 1」が出力される
E. 常に「5 4 3 2 1」が出力される

➡ P607

60. 次のプログラムは実行すると例外が発生する。例外の発生を解消する方法として、正しいものを選びなさい。(1つ選択)

```
import java.util.Arrays;

public class Sample {
    static class Item {
        private String name;
        public Item(String name) {
            super();
            this.name = name;
        }
    }

    public static void main(String[] args) {
        Item[] items = {
                new Item("apple"),
                new Item("orange"),
                new Item("banana")
        };
        Arrays.sort(items);
        for (Item item : items) {
            System.out.println(item.name);
        }
    }
}
```

A. ItemクラスがComparator<Item>を実現するようにし、次のコードを実装する

```
public int compare(Item a, Item b) {
    return a.name.compareTo(b.name);
}
```

B. ItemクラスがComparator<Item>を実現するようにし、次のコードを実装する

```
public int compareTo(Item o) {
    return name.compareTo(o.name);
}
```

C. ItemクラスがComparable<Item>を実現するようにし、次のコードを実装する

```
public int compare(Item a, Item b) {
    return a.name.compareTo(b.name);
}
```

D. ItemクラスがComparable<Item>を実現するようにし、次のコードを実装する

```
public int compareTo(Item o) {
    return name.compareTo(o.name);
}
```

➡ P607

61. 次のプログラムをコンパイル、実行したときの結果として、正しいものを選びなさい。なお、srcとdestは存在するものとする。(1つ選択)

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

public class Sample {
    public static void main(String[] args) throws IOException {
        Path src = Paths.get("a/sample.txt");
        Path dest = Paths.get("dest");
        Files.move(src, dest);
    }
}
```

A. a/sample.txtのコピーがdestディレクトリに作られ、a/sample.txtが削除される
B. sample.txtの名前がdestに変更される
C. java.nio.file.FileAlreadyExistsExceptionがスローされる
D. java.nio.file.NoSuchFileExceptionがスローされる
E. コンパイルエラーが発生する

➡ P608

62. 次の3つのモジュール定義ファイルに関する説明として、正しいものを選びなさい。（2つ選択）

```
module api {
    exports com.sample.api;
}

module provider {
    requires api;
    provides com.sample.api with com.test.impl;
}

module client {
    requires api;
    uses com.sample.api;
}
```

A. providerモジュールは、実行時にcom.sample.apiのAPIを提供できる唯一のモジュールである
B. clientモジュールでproviderモジュールを要求する必要がある
C. providerモジュールでcom.test.implパッケージをエクスポートする必要がある
D. apiという独立したモジュールに配置すると、com.sample.apiのAPIを使用する複数のモジュールのインストールが容易になる

➡ P608

63. 次のうち、プリミティブ型のデータを戻すことができるインタフェースを選びなさい。（1つ選択）

A. java.util.function.ToDoubleFunction
B. java.util.function.Supplier
C. java.util.function.LongConsumer
D. java.util.function.BiFunction

➡ P609

64. 次のプログラムに関する説明として、正しいものを選びなさい。（1つ選択）

```
public class A {

}

public final class B extends A {

}
```

A. Aクラスで引数なしのコンストラクタが定義されていないので、コンパイルエラーになる
B. AクラスかBクラスのどちらかで、toStringメソッドをオーバーライドしていないので、コンパイルエラーになる
C. コンパイルに成功する
D. finalクラスは、ほかのクラスを拡張できないので、コンパイルエラーになる

➡ P609

65. 関数型インタフェースに関する説明として、正しいものを選びなさい。（1つ選択）

A. @FunctionalInterfaceを付けなければならない
B. 抽象メソッドを1つだけ持たなければならない
C. デフォルトメソッドを1つだけ持たなければならない
D. privateなメソッドとstaticなメソッドは定義できない

➡ P610

66. 次のコードのうち、正しく記述されているものを選びなさい。（1つ選択）

A. `Function func = String::toUpperCase;`
B. `UnaryOperator<String> func = String::toUpperCase;`
C. `Function<String> func = s -> s.toUpperCase();`
D. `UnaryOperator func = s -> s.toUpperCase();`

➡ P610

67. 次のプログラムをコンパイル、実行したときの結果として、正しいものを選びなさい。(1つ選択)

```
1. public class Test<T> {
2.     public int count(T[] array, T val) {
3.         int count = 0;
4.         for (T v : array) {
5.             if (val.equals(v)) {
6.                 count++;
7.             }
8.         }
9.         return count;
10.    }
11. }
```

```
1. public class Sample {
2.     public static void main(String[] args) {
3.         String[] array = {"A", "B", "C", "D"};
4.         Integer[] array2 = {1,2,3,4,5};
5.
6.         Test test = new Test();
7.         Test<String> test2 = new Test<>();
8.
9.         System.out.println(test2.count(array, "A"));
10.        System.out.println(test.count(array, "A"));
11.        System.out.println(test.count(array2, 1));
12.    }
13. }
```

A. 「1」「null」「null」の順に表示される
B. 「1」「1」「1」と表示される
C. コンパイルエラーが発生する
D. 実行時にNullPointerExceptionがスローされる
E. 実行時にClassCastExceptionがスローされる

➡ P610

68. 次のコードのうち、正しく記述されているものを選びなさい。(1つ選択)

A.
```
Comparator<Integer> a = new Comparator<>() {
    public int compare(Integer a, Integer b) {
        return a.compareTo(b);
    }
};
```

B.
```
Comparator b = new Comparator<>() {
    public int compare(Integer a, Integer b) {
        return a.compareTo(b);
    }
};
```

C.
```
var c = new Comparator<>() {
    public int compare(Integer a, Integer b) {
        return a.compareTo(b);
    }
};
```

D.
```
Comparator<> d = new Comparator<Integer>() {
    public int compare(Integer a, Integer b) {
        return a.compareTo(b);
    }
};
```

➡ P611

69. ユーザーからの入力を安全に検証するために必要なことを選びなさい。(2つ選択)

A. 信頼できるドメイン固有のライブラリを使用して、入力を検証する
B. 検証を回避するため、必要に応じて入力された値を修正する
C. 有効な文字および入力値のみを受け入れる
D. 値の数値範囲チェック処理をデータベースに委譲する
E. 入力はすでに検証されていると仮定する

➡ P611

70. Java SEのセキュアコーディングガイドラインに従うために、次のクラスにどのような変更をすればよいか。変更内容として正しいものを選びなさい。(3つ選択)

```
1. public class Sample {
2.     String[] array;
3.     public Sample(String[] array) {
4.         this.array = array;
5.     }
6.     public String[] getArray() {
7.         return array;
8.     }
9. }
```

A. getArrayメソッドの戻り値を戻すコードを、「return array.clone();」に変更する
B. フィールドarrayをprivate finalで修飾する
C. getArrayメソッドをsynchronizedで修飾する
D. コンストラクタをprivateにする
E. フィールドをvolatileで修飾する
F. getArrayメソッドの名前をget$Arrayに変更する
G. コンストラクタでフィールドを初期化するとき、「array.clone();」の結果を代入するよう変更する

➡ P612

71. 重要な機密情報を扱うオブジェクトを安全にシリアライズするためには何を行うべきか。正しい説明を選びなさい。(2つ選択)

A. serialPersistentFieldsフィールドを定義して、シリアライズされるべきフィールドを指定する
B. 抽象クラスにする
C. フィールドをtransientで修飾して宣言する
D. インスタンスをシリアライズ・プロキシに置き換えて、readResolveのみを実装し、writeReplaceは実装しない
E. インスタンスをシリアライズ・プロキシに置き換えて、writeReplaceのみを実装し、readResolveは実装しない

➡ P612

72. 次のプログラムを使った処理として、誤っているものを選びなさい。(1つ選択)

```
List<String> data = List.of("apple", "banana", "orange");
List<String> backup = new ArrayList<>();
```

A.
```
data.parallelStream()
    .filter(str -> str.indexOf('n') != -1)
    .forEach(str -> backup.add(str));
```

B.
```
data.stream()
    .filter(str -> str.indexOf('n') != -1)
    .parallel()
    .collect(backup);
```

C.
```
data.stream()
    .filter(str -> str.indexOf('n') != -1)
    .parallel()
    .forEach(str -> backup.add(str));
```

D.
```
backup = data.stream()
            .filter(str -> str.indexOf('n') != -1)
            .parallel()
            .collect(Collectors.toList());
```

➡ P613

73. 次のコードのうち、正しく記述されているものを選びなさい。(1つ選択)

A. `Function<int> a = n -> n * 2;`
B. `Function<int, int> c = n -> n * 2;`
C. `Function<Integer, Integer> b = n -> n * 2;`
D. `Function<Integer> d = n -> n * 2;`
E. `Function e = n -> n * 2;`

➡ P613

74. 次のコードを確認してください。

```
1. public class Sample {
2.     public static void main(String[] args) {
3.         new Sample().test(new int[] {4,3,2,1});
4.     }
5.
6.     void test(int[] array) {
7.         int length = array.length;
8.         for (int i = 1; i < length; i++) {
9.             int tmp = array[i];
10.            int n = i - 1;
11.            // code 1
12.            while( (-1 < n) && (array[i] < array[n])) {
13.                // code 2
14.                array[n + 1] = array[n];
15.                n--;
16.                // code 3
17.            }
18.            array[n + 1] = tmp;
19.            // code 4
20.        }
21.    }
22. }
```

配列arrayの要素がソートされたことを確認するには、code 1〜4のどこに次のコードを挿入すればよいか。（1つ選択）

```
assert n < 0 || array[n] <= array[n+1];
```

A. code 1
B. code 2
C. code 3
D. code 4

➡ P613

75. 次のプログラムをコンパイル、実行したときの結果として、正しいものを選びなさい。（1つ選択）

```
1. import java.util.List;
2. import java.util.stream.IntStream;
3.
4. public class Sample {
5.     public static void main(String[] args) {
6.         final List<String> list
7.             = List.of("A", "B", "C", "D");
8.         final List<String> list2
9.             = List.of("D", "E", "F", "G");
10.
11.        IntStream
12.            .range(0, Math.min(list.size(), list2.size()))
13.            .mapToObj((i) -> list.get(i) + " " + list2.get(i))
14.            .forEach(System.out::println);
15.    }
16. }
```

A. 何も表示されない
B. 「A D」「B E」「C F」「D G」と表示される
C. 「A D」と表示される
D. コンパイルエラーが発生する

➡ P613

76. readObjectメソッドが呼び出されるタイミングとして、正しいものを選びなさい。(1つ選択)

```
 1. import java.io.ObjectInputStream;
 2. import java.io.Serializable;
 3.
 4. public class Sample implements Serializable {
 5.     private String value;
 6.
 7.     public Sample(String value) {
 8.         this.value = value;
 9.     }
10.
11.     private void readObject(ObjectInputStream in) {
12.         try {
13.             in.defaultReadObject();
14.         } catch (Exception e) {
15.             throw new RuntimeException();
16.         }
17.     }
18. }
```

A. このオブジェクトのデシリアライズ後
B. このオブジェクトのシリアライズ後
C. このオブジェクトのデシリアライズ前
D. このオブジェクトのシリアライズ前
E. 呼び出されることはない

➡ P614

77. 次のプロパティファイルがある。このとき、次のプログラムを実行すると、どのプロパティファイルの内容が表示されるか。正しいものを選びなさい。(1つ選択)

- sample.properties
- sample_en.properties
- sample_US.properties
- sample_en_US.properties

```
 1. import java.util.Locale;
 2. import java.util.ResourceBundle;
 3.
 4. public class Sample {
 5.     public static void main(String[] args) {
 6.         Locale current =
 7.                 new Locale.Builder()
 8.                 .setLanguage("en").build();
 9.
10.         ResourceBundle resource =
11.                 ResourceBundle.getBundle("sample", current);
12.         System.out.println(resource.getString("hello"));
13.     }
14. }
```

A. sample.properties
B. sample_en.properties
C. sample_US.properties
D. sample_en_US.properties

➡ P614

78. 次のプログラムをコンパイル、実行したときの結果として、正しいものを選びなさい。(1つ選択)

```
 1. import java.util.List;
 2. import java.util.function.Consumer;
 3.
 4. public class Sample {
 5.     public static void main(String[] args) {
 6.         List<String> list = List.of("A", "B", "C");
 7.         Consumer<String> a = System.out::print;
 8.         Consumer<String> b =
 9.             a.andThen(
10.                 str ->
11.                 System.out.println(":" + str.toLowerCase())
12.             );
13.         list.forEach(b);
14.     }
15. }
```

A. 「:a:b:c」「ABC」と表示される
B. 「a:A」「b:B」「c:C」と表示される
C. 「ABC」「:a:b:c」と表示される
D. 「:a:b:c」と表示される
E. 「A:a」「B:b」「C:c」と表示される

➡ P614

79. 次のプログラムの「// insert code here」に記述するHelloクラスのインスタンスを生成するコードとして、正しいものを選びなさい。(1つ選択)

```
1. public class Sample {
2.     class Hello {
3.         // some code
4.     }
5.     public static void main(String[] args) {
6.         // insert code here
7.     }
8. }
```

A.
```
Sample a = new Sample();
Hello h = new Hello();
```

B.
```
Sample b = new Sample();
Sample.Hello h = b.new Hello();
```

C.
```
Sample c = new Sample();
Sample.Hello h = c.new Sample.Hello();
```

D.
```
Sample.Hello d = new Sample.Hello();
```

➡ P615

80. 次のようなセキュリティポリシーファイルで運用されているライブラリがある。このセキュリティポリシーに関する説明として、正しいものを選びなさい。(1つ選択)

```
grant codebase "sample/lib.jar" {
    permission java.security.AllPermission;
};
```

A. 過剰な権限が与えられているので、付与を削除する
B. 必要な権限のみをリストすることは問題ないので、何もしない
C. 内部ツールであり、外部に公開されないので、何もしない
D. 過剰な権限付与を参照するツールに対するセキュリティリスクを、責任者に正式に報告する
E. 調査しているバグとは関係ないので、何もしない

➡ P615

第13章　総仕上げ問題
解　答

1. A

➡ P530

java.lang.Systemクラスの**inフィールド**は、キーボードから入力されるデータを扱うためのInputStream型オブジェクトです。設問のコードは、バイトストリームを扱うInputStreamを、文字ストリームを扱うReaderに変換するInputStreamReaderクラスを使い、さらに読み込みの効率化のためにBufferedReaderを併用しています。
BufferedReaderの**readLineメソッド**を実行すると、キーボードからの入力をバイトストリームとして受け取り、それを文字ストリームに変換して文字列として戻します。もし、キーボードからの入力がなければ、入力されるまでの間、プログラムは処理を停止して、入力待ちになります。

設問では、javaコマンドのオプションとして、プログラムにデータを渡す「起動パラメータ」を使っています。起動パラメータは、プログラムの実行時にデータを入力し、mainメソッドの引数として渡されるもので、プログラム実行中にキーボード入力として取り出せるものではありません。
そのため、設問のコードではキーボードからの入力を待つだけで、プログラムの進行は停止します。以上のことから、選択肢**A**が正解です。

2. A

➡ P531

java.nio.file.Filesクラスの**linesメソッド**は、ファイル内のすべての行を読み込み、**Streamとして戻す**メソッドです。このメソッドは、読み取る対象のPathオブジェクトを引数に受け取ります。よって、選択肢**A**が正解です。選択肢Cは、String型だけを扱うSteamオブジェクトを戻そうとしているため誤りです。
Filesクラスの**readAllLinesメソッド**は、ファイル内のすべての行を読み込み、**Listとして戻す**メソッドです。よって、選択肢BとDは誤りです。

3. A

➡ P531

java.util.ServiceLoaderクラスの**loadメソッド**は、ServiceLoaderのインスタンスを生成し、その参照を戻します。設問のようにTest型のサービスを扱うServiceLoaderのインスタンスを生成する場合は、次のように記述します。

```
ServiceLoader<Test> loader = ServiceLoader.load(Test.class);
```


ServiceLoaderクラスは、java.lang.Iterableインタフェースを実装しているため、拡張for文で繰り返し、サービスの実装を取り出して、処理することができます。よって、選択肢**A**が正解です。選択肢Cのように、java.lang.Iterableインタフェースの実装を取得するためにservicesメソッドを呼び出す必要はありません。
ServiceLoaderクラスは、設問のコードであれば、Testインタフェースのような実際のサービスのインタフェースを実装しているわけではありません。クラスパス上にあるサービスのインタフェースを実装したクラスを探し出し、そのコレクションを扱うのが、ServiceLoaderクラスの役割です。そのため、選択肢BのようにTestインタフェースのexecuteメソッドを呼び出すことはできません。よって、誤りです。
ServiceLoaderクラスに、getTestのような特定のインタフェースの実装を取得するようなメソッドは存在しません。よって、選択肢Dも誤りです。

4. B

➡ P531

peekメソッドは、ストリーム・パイプラインの途中がどのような状態になっているかを確認する**中間操作**の1つです。そのため、このメソッドはストリーム・パイプラインへの処理が終わればそのままストリームを戻し、終端操作につなげられるようにします。
一方で、**forEachメソッド**は、ストリーム・パイプラインの**終端操作**であり、パイプラインを通ってきた要素を最終的に繰り返し処理するためのものです。そのため、このメソッドは何も戻しません。以上のことから、選択肢**B**が正解です。
どちらのメソッドも、ストリーム・パイプライン内の要素を削除するためのものではありません。よって、選択肢AとCは誤りです。

5. A

➡ P532

リストの中にはappleとbananaが混在しているため、appleの平均価格を求めるには、まず要素をappleだけに絞らなければいけません。特定の要素に絞り込むためには、**filterメソッド**を使って、要素のgetNameメソッドの戻り値がappleと等しいものだけをストリーム・パイプラインに通るようにします。
要素をappleに絞れれば、**mapToIntメソッド**を使って、Itemを扱うストリームを、価格だけを扱うストリームに変換します。map系のメソッドは、ストリーム・パイプラインで扱う要素を変更するために利用することが多いので覚えておくとよいでしょう。
最後に**averageメソッド**で平均を求め、**getAsDoubleメソッド**で結果をdouble型で戻せば、平均価格を求めることができます。以上のことから、選択肢**A**が正解です。

6. C

➡ P533

java.util.Comparatorは、2つのオブジェクトを受け取って比較するためのインタフェースです。このインタフェースには**compareメソッド**が定義されており、引数で受け取った2つのオブジェクトをどのように比較するかを実装します。選択肢Dは、compareメソッドではなく、compareToメソッドを実装しているため誤りです。

なお、compareメソッドの引数の型は、ジェネリクスの型パラメータとして受け取った型です。そのため、選択肢Cのように型パラメータを渡せばその型に、渡さなかった場合には、選択肢Aのようにjava.lang.Object型を受け取るメソッドとして解釈されます。選択肢Bのように、型パラメータを渡さないにもかかわらず、String型の引数を受け取ることはありません。よって、選択肢Bは誤りです。

compareメソッドは、比較した結果をint型の値で戻します。このメソッドは、最初の引数が2番目の引数より小さい場合は負の整数（-1）、両方が等しい場合は0、最初の引数が2番目の引数より大きい場合は正の整数（1）を戻します。よって、boolean型を戻すとした選択肢Aは誤りです。
以上のことから、選択肢**C**が正解です。

7. A

➡ P533

ボクシングに関する問題です。
設問のコードで、リストの要素はInteger型です。そのため、**オートアンボクシング**できるのは、int型もしくは暗黙の型変換ができるdouble型です。よって、getメソッドの戻り値をdouble型で受け取っている選択肢Bと、int型で受け取っている選択肢Fは正しいコードです。
選択肢Eは、Integer型の要素をそのまま取り出して、Integer型の変数で受け取っているため問題ありません。

Integerクラスのインスタンスを生成するvalueOfメソッドには、int型もしくはString型の引数を受け取るものが用意されていますが、Integer型を受け取るものは存在しません。しかし、アンボクシングすればInteger型をint型として扱えるため、選択肢CのようにvalueOfメソッドの引数にInteger型を渡すことは可能です。
同様に、DoubleクラスのvalueOfメソッドに、Integer型を受け取るものは存在しません。しかし、アンボクシングすればInteger型をint型として扱えることができ、また、int型は**暗黙的な型変換**によりdouble型として扱うことができるため、選択肢DのようにvalueOfメソッドの引数にInteger型を渡すことは可能です。

ボクシングやアンボクシングができるのは、**ラッパークラス型とプリミティブ型の間だけ**です。ラッパークラス同士では、自動的な型変換は行われません。そのため、選択肢AのようにInteger型で取り出した参照を、互換性のないDouble型の変数に代入することはできません。以上のことから、選択肢**A**が正解です。

8. D

➡ P534

クラスが定義するメソッドと、インタフェースのデフォルトメソッドが重複した場合、**クラスに定義されたメソッドが優先**されます。インタフェースのデフォルトメソッドは、あくまでも実装がなかったときの標準的な処理を記述するためのものだからです。

設問のコードでは、スーパークラスのAbstractTestがprivateなexecuteメソッドを、インタフェースのTestがpublicなデフォルトメソッドを定義しています。前述の「クラスに定義されているメソッドが優先される」というルールは、コンパイル時ではなく実行時に適用されます。そのため、コンパイラは、インタフェースに定義されているpublicなexecuteメソッドを実行するものと解釈し、コンパイルは成功します。しかし、実行時には前述のルールが適用されるため、不正なアクセスをしたことを示す例外IllegalAccessErrorがスローされます。
以上のことから、選択肢**D**が正解です。

9. D

➡ P534

java.util.functionパッケージのインタフェースで、戻り値を戻さないメソッドを持つのは**Consumerインタフェース**です。よって、選択肢**D**が正解です。
選択肢Aのjava.util.function.Predicateは、条件に一致するかどうかを判定した結果のboolean型の戻り値を戻します。選択肢Bのjava.util.function.Functionは、ジェネリクスで引数と戻り値の型を受け取ります。よって、この関数型インタフェースの抽象メソッドは、戻り値を戻します。選択肢Cのjava.util.function.Supplierは、何かを提供するための関数型インタフェースです。よって、このインタフェースの抽象メソッドも、戻り値を戻します。以上のことから、これらの選択肢は誤りです。

10. B

➡ P535

try-with-resourcesは、tryブロックを抜ける際にリソースの**closeメソッド**を自動的に呼び出します。もし、tryブロック内で例外が発生し、catchブロックに遷移したときには、closeメソッドが実行されてから、catchブロック内の処理が実行されます。

ただし、設問のコードのようにcloseメソッドでも例外が発生した場合には、その例外は**抑制**されて、tryブロック内で発生した例外だけが処理されます。以上のことから、選択肢Bが正解です。

11. C、D

➡ P535

機密情報を扱うインスタンスからの情報漏洩を防ぐため、インスタンスが保持するフィールドの値を明示的にクリアする機能を持たせ、フィールドの値をクリアし終わってから、ガベージコレクションによってインスタンスが削除されるように設計するのがよいでしょう。インスタンスを削除するのはガベージコレクタの役割で、ガベージコレクションが実行されるまでの間に、機密情報が漏洩するかもしれないからです。よって、選択肢Eは、機密情報を保護することにはなりません。

機密情報は、自らが作ったライブラリで処理することが望ましいように思えますが、そのライブラリが悪意のあるコードに置き換えられている可能性を考慮すべきでしょう。機密情報を持ったインスタンスへの参照を悪意のあるコードに渡してしまえば、その機密情報を書き換えられてしまう危険性があるのです。同じことは、信頼できるオブジェクトについてもいえることです。よって、選択肢AとBは誤りです。

また、悪意のあるコードを実行させないことも重要です。クラスローダーを書き換えて、悪意のあるコードが含まれるクラスを実行させるような攻撃を防ぐには、標準クラスライブラリの**セキュリティマネージャ**を利用してさまざまな権限を制御するのが有効でしょう。セキュリティマネージャが扱える権限には、現在のクラスローダーの取得や新しいクラスローダーの生成など、クラスローダーを制御する権限がいくつかあります。よって、選択肢Cは正解です。

インスタンスが持つ機密情報の改ざんを防ぐには、インスタンスそのものが**不変オブジェクト**として設計されている必要があります。不変オブジェクトでは、フィールドの値を変更することはできませんが、変更後の新しい値を持った新しいインスタンスを生成して、その参照を戻すという方法で対処できます。よって、選択肢Dは正解です。

12. B、E

➡ P536

変数numを**スレッドセーフ**にするには、**synchronized**を使って排他制御を行うか、**AtomicInteger**などのマルチスレッドに対応可能なプリミティブ型のラッパークラスを利用します。

synchronizedによる排他制御の対象は、Sampleのインスタンスが持つnumフィールドです。staticに動作するmainメソッドを排他制御にしても、Sampleのインスタンスが持つnumフィールドはスレッドセーフになりません。よって、選択肢Aは誤りです。

排他制御を実現するには、メソッド宣言をsynchronizedで修飾する以外に、選択肢BやCのようにsynchronizedブロックを利用する方法もあります。このブロックでは、どのインスタンスをロックするかを指定しなければいけません。よって、選択肢Bが正解です。選択肢Cは、フィールドの値を参照しているだけであり、値そのものをロックしてもインスタンスはロックされません。よって、誤りです。

マルチスレッドによる並列処理では、各スレッドがフィールドの値のキャッシュを持ちます。スレッドが、ヒープ上にあるインスタンスを参照せずに処理を進められるため、パフォーマンスが向上するのが理由です。このようなメリットがある一方で、スレッドが持っている値と、インスタンスのフィールドの値にズレが生じてしまう可能性があります。

そこで、このような不整合が発生しないようするために、選択肢Dの**volatile**でフィールドを修飾すると、キャッシュされずに安全に扱えるようになります。2つのスレッドが1つのフィールドを共有する場合に、一方がフィールドの値を書き込みしている最中に、もう一方がキャッシュではなく、フィールドの値を正しく読み込むことで、不整合が発生しないようにするためのものです。両方のスレッドが、フィールドの値に書き込みをすることを制御するためのものではありません。よって、誤りです。

synchronizedは、インスタンスをロックしてほかのスレッドからのアクセスを排除するため、パフォーマンスに影響を及ぼすこともあります。これに対して、選択肢EのAtomicIntegerはパフォーマンスへの影響を抑えながら、排他制御を実現します。スレッドセーフなインスタンスのフィールドを守るには最適なクラスです。よって、選択肢Eは正解です。

13. B

➡ P537

設問のコードでは、アメリカで英語を使う**ロケール情報**のインスタンスを生成し（8〜9行目）、日付の書式設定を行っています（10〜15行目）。日付の書式を指定する際に、列挙型である**java.time.format.FormatStyleの列挙子**「MEDIUM」と「SHORT」を使用しています。この列挙型には、このほかにFULLとLONGを含む4つの列挙子が定義されています。列挙子によって、日付の書式は次のようになります。

【FormatStyleの列挙子による日付の書式（2021年4月1日の例）】

列挙子	書式
FULL	Thursday, April 1, 2021
LONG	April 1, 2021
MEDIUM	Apr 1, 2021
SHORT	4/1/21

設問のコードで使用しているのは、MEDIUMとSHORTです。よって、この書式で出力している選択肢**B**が正解です。

14. C

➡ P538

AのSQL文では、変数nameの値に「'a' or 'a = 'a'」を与えると、nameが「a」もしくは「aがaである」ことが条件となり、常に条件が一致することになります。このようなSQL文は、SQLインジェクションの原因になり得るため推奨されません。よって、選択肢BやDは誤りです。

java.sql.Statementインタフェースの**enquoteIdentifierメソッド**を使うことで、SQLに与える値を引用符で囲むことができます。そうすることで、SQLインジェクションを防ぐ一定の効果があります。SQL文のリテラルは、文字列や日付は引用符で囲む必要がありますが、数値を囲む必要はありません。よって、選択肢Aのようにすべて引用符で閉じる必要はありません。以上のことから、選択肢**C**が正解です。

15. D

➡ P538

java.lang.AutoCloseableインタフェースには、**closeメソッド**だけが定義されており、**try-with-resources**で自動的にリソースを閉じるときの処理を実装します。よって、選択肢**D**が正解です。
AutoCloseableインタフェースにautoCloseというメソッドは存在しません。また、AutoCloseableはインタフェースであり、try-with-resources文で扱うには、継承ではなく実現したクラスを用意しなければいけません。以上のことから、選択肢A、B、Cは誤りです。

16. B

➡ P539

設問のコードでは、**mapToIntメソッド**を使って、要素がSample型のストリームを、getNumメソッドの戻り値の集合から成るint型のストリームに変換しています。
その上で、**filterメソッド**で20よりも小さいものに絞り込んでいるため、ストリームを通過するのは8と12だけとなります。最後に、**reduceメソッド**で2つの値を合計しているため、コンソールには「20」と表示されます。以上のことから、選択肢**B**が正解です。

17. A、C

➡ P540

ジェネリクスに関する問題です。
設問のコードでは、変数itemStreamを型パラメータなしでStream型として宣言しています。Stream型は、ストリームで扱う要素の型を型パラメータとして受け取りますが、設問のように**型パラメータを指定しなければ、Object型のストリームとして扱われる**ことになります。そのため、filterメソッド内で使っているgetPriceメソッドのようなItemクラスの独自メソッドを使用することはできません。よって、Item型の引数を受け取るとした選択肢Bは誤りです。
もし、Itemクラスに定義されている独自のメソッドを使用するには、変数itemをダウンキャストするか、もしくは変数itemStreamを宣言するときにItem型の型パラメータを渡すしかありません。よって、選択肢**A**と**C**が正解です。
Listインタフェースには、Item型だけを扱うList型を戻すstreamメソッドは存在しません。よって、選択肢Dは誤りです。

18. C

➡ P541

java.io.Consoleは、キーボードからの入力を受け取ることができるクラスです。入力を受け取るときに**readLineメソッド**を使えば、コンソールに入力した内容をそのまま表示しながら受け取ることができます。一方、**readPasswordメソッド**を使うと、入力した内容は機密情報であるパスワードとして扱われ、コンソールに表示せずに受け取ることができます。
ただし、機密情報として受け取っても、その後は単なる文字列として扱われるため、System.out.printlnを使えば、そのまま表示されます。よって、選択肢**C**が正解です。

選択肢AとBは、readPasswordメソッドを使っているにもかかわらず、入力内

容がコンソールに表示されています。選択肢Dは、readPasswordメソッドの入力内容はコンソールに表示されていませんが、System.out.printlnで単なる文字列になっている部分も表示されていません。よって、これらの選択肢は誤りです。

19. B

➡ P542

リストとセットの違いに関する問題です。
リストとセットの違いは、**要素の重複**を許す／許さないという点です。リストは重複した要素を持つことができますが、セットは重複した要素を持つことができません。「重複する要素」とは、**equalsメソッドによって等しいと判断される要素**のことです。

設問のコードでは、5つの文字列を持った配列を作り、リストとセットにそれぞれ追加しています。「a」「b」「c」「d」「a」の5つのうち2つ（a）は、equalsメソッドがtrueを戻す「重複した要素」です。そのため、リストには5つとも追加されますが、セットには4つしか追加されません。よって、コンソールには4と5が表示されます（15行目）。

次に、リストをコンストラクタに渡してHashSetのインスタンスを生成し（16行目）、セットを渡してLinkedListのインスタンスを生成しています（17行目）。リストは5つの要素を持っていましたが、HashSetのインスタンスを生成する段階で重複が取り除かれるため、要素数は4となります。セットは重複が排除されたあとの4つしか要素を持っていないため、LinkedListのインスタンスも4つしか要素を持ちません。そのため、コンソールには4と4が表示されることになります。

以上のことから、コンソールには「4, 5, 4, 4」と表示されるため、選択肢**B**が正解となります。

20. C

➡ P542

ラムダ式の定義に関する問題です。
選択肢Aは、引数を扱うカッコ「()」と**アロー演算子**「->」がありません。
選択肢Bは、引数を受け取る変数ではなく、値そのものを記述しています。
選択肢Eは、引数を受け取る変数もカッコも記述されていません。よって、これらは誤りです。

java.lang.Runnableインタフェースに定義されている抽象メソッドrunは、引数を受け取らず、戻り値も戻しません。一方、メソッド参照「System.out::print」は、戻り値は戻しませんが、コンソールに表示する対象である引数を必要とします。よって、選択肢Dも誤りです。
以上のことから、選択肢**C**が正解です。

21. D

➡ P543

java.util.OptionalクラスのifPresentメソッドに関する問題です。
ifPresentメソッドは、結果が存在すれば、渡されたConsumer型の処理を実行しますが、結果が存在しなければ何もしません。

設問のコードでは、文字列を持ったリストからストリームを作り、filterメソッドで「z」を文字列に含むものに絞り込んでいますが、リストには該当する要素はありません。そのため、**reduceメソッド**は何もせずに、何も結果を戻しません。よって、コンソールには何も表示されません。以上のことから選択肢**D**が正解です。

22. C

➡ P543

jdepsコマンドの**-jdkinternalsオプション**は、JDKの内部APIでクラスレベルの依存関係を検索するためのものです。内部APIとは、Javaプログラマーが利用するためのライブラリではなく、標準クラスライブラリ内で利用されることを目的としたライブラリのことです。デフォルトでは、**-includeオプション**が指定されていなければ、-classpathオプションおよび入力ファイルで指定されたクラスがすべて分析されます。以上のことから、選択肢**C**が正解です。

23. B

➡ P544

設問のコードでは、**java.util.stream.Streamインタフェース**の**ofメソッド**を使って、int型を扱う二次元配列をもとにストリームを作っています。このタイミングでは、ストリーム内には、{1, 2}、{3, 4}、{5, 6}という2つずつの要素を持つ一次元配列への参照が3つ入っています。
これを、**flatMapToIntメソッド**を使ってIntStream型のストリームに変換しています。これで配列内の要素だけから成るストリームができあがります。このタイミングで、ストリームには、1、2、3、4、5、6の6つの整数が入っています。

次に、**mapメソッド**で全要素の値を2倍にしているため、ストリーム内の要素は、2、4、6、8、10、12になります。これを**peekメソッド**でコンソールに表示したあと、filterメソッドで3で割り切れるものに絞り込みます。そのため、ストリーム内には6と12だけが残ります。最後に、**countメソッド**でストリーム内の要素を数えるため、コンソールには2が表示されます。
以上のことから、コンソールには「246810 2」と表示されるため、選択肢**B**が正解となります。

24. C、E、F

➡ P544

機密情報を扱うには、まず「何が機密情報であるのか」を明確化しなくてはいけません。機密情報として識別することが、情報を適切に保護する第一歩となります（選択肢**C**）。識別された機密情報は、ほかの情報とは切り分けて厳重に扱う必要があります（選択肢G）。
セキュアコーディングガイドラインに挙げられている、特に重要な10項目のうちの1つである「入力検証」に沿って、ユーザーから入力された値が不正である可能性を見据え、外部から入力されたデータはすべて検証した上で扱うべきです（選択肢D、**E**）。また、ホワイトリストや最小権限の原則を取り入れて、オブジェクトへのアクセスを制限することも重要な施策です（選択肢**F**）。
障害時の解析などで使われる例外情報やコアダンプなどは、情報漏洩を目的とした攻撃の手掛かりとなり得るため、透過的に処理すべきではありません（選択肢B）。

カプセル化はあくまでも、責務の明確なモジュール（部品）を定義するために行うもので、セキュアコーディングと直接的な関係ありません（選択肢A）。

25. B

➡ P545

列挙型のコンストラクタのアクセス修飾子は、**private**でなくてはいけません。publicにしておくと、プログラマーが自由にインスタンスを生成するコードを記述できるようになり、規定の列挙子以外を使えるようになってしまうためです。列挙型は、あくまでも定数の集合を作るためのものであることを忘れないようにしましょう。よって、選択肢**B**が正解です。

なお、enum型のコンストラクタは暗黙的にprivateとして扱われます。アクセス修飾子を記述しなかったとしても、デフォルトのアクセス修飾子ではなく、privateとして扱われる点に注意してください。

26. A、C

➡ P545

インタフェースはインスタンスを生成できないため、定数しか定義できません。インスタンスフィールドの目的は、インスタンスごとに異なる値を持たせることです。そのため、インタフェースには定義できません。一方、抽象クラスは、継承した具象クラスのインスタンスが利用されることが前提であるため、インスタンスフィールドを定義できます。よって、選択肢**A**は正解です。
インタフェースには、抽象メソッドのほか、インスタンスを生成する必要がないstaticメソッドを定義できます。よって、選択肢Bは誤りです。

インタフェースに定義できるのは、public、もしくはprivateなメソッドだけです。protectedなメソッドは定義できません。よって、選択肢**C**は正解です。
finalで修飾されたメソッドは、オーバーライドできません。インタフェースには、オーバーライドできないメソッドは定義できません。よって、選択肢Dは誤りです。
インタフェースには、抽象メソッドのほか、privateなメソッドやstaticなメソッド、デフォルトメソッドを定義できます。よって、選択肢Eは誤りです。

27. C

➡ P546

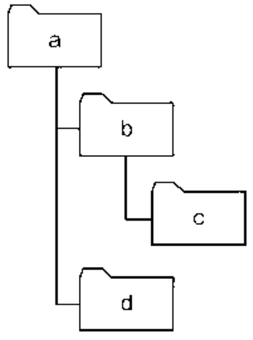
try-with-resourcesは、tryブロックを抜ける際に**closeメソッド**を自動的に呼び出し、リソースを閉じます。設問のコードでは、3つのFileReaderクラスのインスタンスがtry-with-resourcesに与えられているため、tryブロックを抜ける際に、これらすべてのcloseメソッドが呼ばれます。以上のことから、選択肢**C**が正解です。

28. A

➡ P546

設問のコードで指定されているPathは、「aの中のb、bの中のc、さらに2つ上に戻って、aの中のd」という意味になります。これは、次のようなディレクトリ構造を表しています。

【設問のコードが表すディレクトリ構造】



このようなディレクトリ構造あれば、「aの中のd」と記述したほうが無駄がなくわかりやすくなるでしょう。java.nio.file.Pathインタフェースの**normalizeメソッド**は、このような無駄なパス表現を排除し、「a/d」として扱います。以上のことから、選択肢**A**が正解です。

29. B

➡ P547

設問のコードでは、**継承関係にある2つの例外クラス**が用意され、サブクラスであるTestExceptionがスローされています。スローされたTestExceptionは、スーパークラスであるSampleExceptionをキャッチするcatchブロックで捕捉され、RuntimeExceptionがスローされ直されます。そのため、コンソールには「B」と表示されます。よって、選択肢**B**が正解です。

最終的にはRuntimeExceptionがスローされ、TestExceptionがこのメソッドからスローされることはありません。そのため、throws句でTestExceptionがスローされる可能性を示唆するのは間違いであるかのように思えますが、throws句では実際にその例外がスローされるかどうかまではチェックされません。あくまでもtestメソッドを利用している側にtry-catchなどの例外処理を促すのがthrows句の目的だからです。よって、コンパイルエラーは起きません。

30. B

➡ P548

java.nio.file.Filesクラスの**deleteIfExistsメソッド**は、パスで表現されるファイルやディレクトリが存在すれば削除し、削除したかどうかをboolean型の結果で戻します。設問では、sample.txtが存在しなかったときの挙動を問うています。10行目でdeleteIfExistsメソッドがfalseを戻すため、11〜15行目のif-else文ではelseブロックが実行され、コンソールには「B」が表示されます。以上のことから、選択肢**B**が正解です。

31. D、E

➡ P549

モジュール定義ファイルの**usesディレクティブ**は、どのインタフェースを**SPI**（**Service Provider Interface**）として利用するかを宣言するものです。SPIの実装は、あとから提供されるものであり、インタフェースを利用している側は実装に依存する必要はありません。そのため、コンパイル時に実装が提供されている必要はありません。よって、選択肢Aは誤りです。

SPIを実装したモジュールがなかった場合、ServiceLoaderの**loadメソッド**は空のServiceLoaderのインスタンスを戻すだけで、エラーにはなりません。while文や拡張for文などが一度も繰り返し処理を実行しないだけです。よって、選択肢Bは誤りです。

SPIとして拡張可能なインタフェースを提供しているモジュールが、自身のインタフェースの実装を提供してしまうと、SPIである意味がなくなります。よって、選択肢Cは誤りです。

SPIを実装するモジュールは、SPIの拡張ポイントとしてのインタフェースを参照する必要があります。そのため、実装モジュールはSampleモジュールを利用できるように、そのモジュール定義ファイルに「requires Sample;」というディレクティブを含める必要があります。また、SPIの拡張ポイントとなるインタフェースが変わらない限り、SPIを利用する側のモジュールを再コンパイルする必要はなく、独立してコンパイルすることができます。よって、選択肢**D**と**E**が正解です。

32. D

➡ P549

java.util.stream.Streamインタフェースの**mapToXXXメソッド**は、ストリームをそれぞれプリミティブ型のデータを扱うストリームに変換します。変換できるのは、int型のストリームを扱う**IntStream**、double型のストリームを扱う**DoubleStream**、long型を扱う**LongStream**です。

設問のコードでは、Streamインタフェースのofメソッドを使ってInteger型の要素を扱うストリームを作っています。2行目では**mapToIntメソッド**を使ってIntStream型のストリームに変換し、3行目では**mapToDoubleメソッド**を使ってDoubleStream型のストリームに変換しています。4行目では、mapToIntメソッドを使ってIntStreamに変換していますが、その参照をInteger型を扱うStream型変数に代入しようとしています。
StreamとIntStreamは、共に**BaseStreamインタフェース**を継承するサブインタフェースであって、互換性はありません。そのため、このコードは4行目でコンパイルエラーとなります。以上のことから、選択肢**D**が正解です。

33. A

➡ P550

設問のコードでは、2つのインスタンス（TestとSample）を共有する2つのスレッドによる**並列処理**が実行されます。互いのスレッドがアクセスするメソッドは**synchronized**で修飾されており、一方のスレッドがインスタンスを使っているときには、もう一方のスレッドは待ち状態に入ります。また、一方のスレッドが1つ目のインスタンスを使っているとき、もう一方のスレッドが2つ目のインスタンスを使っており、それぞれがもう1つのインスタンスを使おうとしたときに、synchronizedによって**デッドロック**が発生してしまう可能性があります。よって、選択肢**A**が正解です。

選択肢Bの**ライブロック**は、デッドロックを回避するコード同士が絡み合って起きる事象ですが、設問のコードでそのような記述はありません。よって、誤りです。
選択肢CのIllegalMonitorStateExceptionは、スレッドをObjectクラスのwaitメソッドを使って停止したり、notifyメソッドやnotifyAllメソッドで停止したス

レッドを再開するときに発生する例外です。設問のコードでは、これらのメソッドを使っていません。よって、選択肢Cも誤りです。

34. D ➡ P551

リストの操作に関する問題です。
リストには、要素を追加するためのメソッドが2つ用意されています。要素を末尾に追加するのが**addメソッド**、任意の場所の要素を置き換えるのが**setメソッド**です。addメソッドには、追加する要素を引数として渡します。setメソッドは引数を2つ受け取り、第1引数には「どの位置の要素」を、第2引数には「どの要素に置き換えるのか」を指定します。

設問のコードでは、ArrayListのインスタンスを生成し、addメソッドを3回呼び出して「A」「100」「B」という要素をそれぞれ追加しています。その後、setメソッドを呼び出して、追加した要素の値を100から200に置き換えています（8行目）。

さらに、9行目では、追加した要素をremoveメソッドで削除しているため、リスト内の要素は「A」と「200」という2つだけが残ることになります。その上で、10行目のsetメソッドでは4番目の要素を300に置き換えようとしています。しかし、存在しない要素を置き換えることはできないため、このコードを実行すると要素外アクセスを表す**java.lang.IndexOutOfBoundsException**がスローされます。以上のことから、選択肢**D**が正解です。

35. A ➡ P552

利用したリソースは、**DoS攻撃**を防ぐためにも、利用終了と同時に閉じなければいけません。設問のコードでは、変数brで扱うインスタンスはtry-with-resourcesにより自動的に閉じますが、変数br2で扱うインスタンスは明示的に閉じていません。そのため、変数br2を宣言し、BufferedReaderのインスタンスを生成するときに**try-with-resources**を使って、**自動的にリソースが閉じる**ようにするべきです。よって、選択肢**A**が正解です。

選択肢Bのように、processメソッドの実行後にcloseメソッドを呼び出すと、例外が発生したときにリソースが閉じられません。また、try-with-resourcesでは、自動的に閉じたいリソースの**インスタンスを生成**するタイミングで、**tryブロック**に指定します。選択肢Cのようにprocessメソッドの呼び出し時ではありません。

BufferedReaderの**readLineメソッド**で1行分のデータを読み込む際に、改行がない巨大なファイルを読み込ませ、メモリを圧迫することでサービス停止に追い込むDoS攻撃を防ぐには、選択肢Dの方法は有効です。しかし、設問のコードであれば、変数br2で読み込むサイズを確認する前に、変数brで読み込むサイズを確認するべきでしょう。よって、選択肢Dでは不十分な対策といわざるを得ません。

選択肢Eは、変数brで扱うリソースは解放できますが、変数br2で扱うリソースは解放されません。よって、誤りです。

36. E ➡ P553

java.util.Listインタフェースに定義されている、List型のインスタンスへの参照を戻すstaticメソッドは、すべて「**変更不可能なリスト**」を生成して戻します。**変更可能なリスト**を生成するには、**java.util.ArrayListクラス**などのインスタンスを生成する必要があります。

設問のコードでは、ArrayListのインスタンスを生成し、2つの要素を追加したあと、ListインタフェースのstaticメソッドであるcopyOfメソッドを使って新しいリストを生成しています。前述のとおり、このリストは「変更不可能なリスト」であるため、次の行でaddメソッドを使って要素を追加しようとすると、**java.lang.UnsupportedOperationException**がスローされます。以上のことから、選択肢**E**が正解です。

37. D ➡ P554

列挙型のvaluesメソッドは、列挙型が持っている列挙子（定数）の配列を戻します。また、**java.lang.Stringクラスのvalue0fメソッド**は、受け取った参照が持つtoStringメソッドを呼び出します。

選択肢Aは、2番目の列挙子のインスタンスへの参照を渡して、Stringクラスのvalue0fメソッドを呼び出しています。前述のとおり、このメソッドは、参照が持つtoStringメソッドを呼び出すため、toStringメソッドの中でさらにtoStringメソッドを呼び出す無限の再帰呼び出しになり、メモリのスタック領域がオーバーフローしてしまいます。よって、選択肢Aは誤りです。

toStringメソッドは、java.lang.Objectクラスに定義されているメソッドで、サブクラスでオーバーライドすることが前提とされています。このメソッドはpublicで定義されており、オーバーライドするサブクラスではpublicよりも厳しいアクセス修飾子に変更することはできません。選択肢Bは、アクセス修飾子「なし」でメソッドを定義しており、publicよりも厳しいアクセス修飾子を使っていることになるため、コンパイルエラーとなります。よって、選択肢Bも誤りです。

選択肢Cは、Typeに定義されている列挙子の1つであるAのnumフィールドの値を文字列に変換して戻しています。Aのnumフィールドの値は1であるため、設問の要件のように2が表示されることはありません。よって、選択肢Cも誤りです。
以上のことから、選択肢**D**が正解となります。

38. C、D

➡ P555

上限境界ワイルドカードと下限境界ワイルドカードを使ったジェネリクスに関する問題です。
上限境界は、**extends**を使ってそのクラスを上限としたワイルドカードを利用するものです。選択肢Dでは、Number型を上限にしたワイルドカードであるため、代入できる参照のジェネリクスの型パラメータは、Numberクラスかそのサブクラスでなくてはいけません。選択肢DのByte型はNumber型のサブクラスですが、選択肢EのObject型はNumber型のスーパークラスです。よって、選択肢**D**が正解で、選択肢Eは誤りです。

下限境界は、**super**を使ってそのサブクラスを下限としたワイルドカードを利用するものです。代入できる参照のジェネリクスの型パラメータは、下限境界の型のスーパークラスである必要があります。
選択肢Aは、Number型を下限とするワイルドカードに、Number型のサブクラスであるInteger型を型パラメータに持つインスタンスへの参照を代入しようとしています。
選択肢Bは、Float型を下限とするワイルドカードに、Float型とは互換性がないDouble型を型パラメータに持つインスタンスへの参照を代入しようとしています。
選択肢Cは、Short型を下限とするワイルドカードに、そのスーパークラスであるNumber型を型パラメータに持つインスタンスへの参照を代入しようとしています。よって、選択肢**C**が正解で、選択肢AとBは誤りです。

39. D

➡ P555

synchronizedブロックによる排他処理は、メソッド単位で行われるため、複数のメソッドをまたぐような排他処理はできません。このような場合には、**java.util.concurrent.locks.ReentrantLockクラス**を利用します。ReentrantLockクラスを利用することで、**lockメソッド**で明示的にロックを獲得し、**unlockメソッド**で解除するまでの間、排他処理を実現します。設問のコードは、この方法によってスレッドセーフな状態を実現しています。

ReentrantLockクラスのコンストラクタはオーバーロードされ、boolean型の引数を受け取るものが用意されています。この引数によって、**ロック取得順序の公平・不公平のポリシー**を選択することができます。
公平であれば、ロック待ちのスレッドから順番にロックを取得していきます。一方の不公平であれば、ロックの取得については順番が決まっておらず、そのときの状態によってロックの取得順が異なります。

不公平であれば、パフォーマンスは優れるものの、一部のスレッドがロックを取得できない可能性が生じてしまいます。これを「飢餓状態」と呼びます。一方、公平であれば、飢餓状態は防げるものの、不公平ロックに比べるとパフォーマンスが劣るという特徴があります。
公平にするのであればtrueを、不公平にするのであればfalseを渡します。デフォルトは不公平なポリシーで、引数なしのコンストラクタを呼び出すと不公平なポリシーに設定されます。
公平な場合は、lockメソッドを実行した順にロックを取得することが保証されます。一方、不公平な場合は、ロックを取得できる順序が必ずしもlockメソッドを呼び出した順になるとは限りません。なお、順序の保証をする処理で待機状態が発生するため、公平にした場合は不公平にした場合に比べてパフォーマンスが劣る傾向があります。

設問のコードでは、ReentrantLockのインスタンスへの参照がフィールドで保持されているため、複数のメソッドにまたがった排他制御が可能です。選択肢Aのようにメソッドのブロック内に移動すると、そのメソッドでしか排他制御ができなくなってしまうため、ReentrantLockを使う理由がなくなります。よって、選択肢Aは誤りです。
前述のようにReentrantLockのコンストラクタでboolean型の引数を受け取るのは、ロックの取得順に関する公平性を決めるためです。スレッドセーフに関する制御をするためではありません。よって、選択肢Bは誤りです。
また、選択肢Cのようにlockメソッドの呼び出しをstaticにしても動作は変わりません。よって、選択肢Cは誤りです。
以上のことから、選択肢**D**が正解です。

40. C

➡ P556

PreparedStatementにセットしたパラメータは、SQLの実行後も残ります。一度セットしたパラメータを変更したい場合には、**java.sql.PreparedStatementクラス**の**clearParametersメソッド**を使ってパラメータをすべてクリアするか、**setXXXメソッド**を使って任意の場所のパラメータを上書きするしかありません。
設問のコードでは、**INSERT文**を実行するためのPreparedStatementを用意し、14行目と15行目で2つのパラメータをセットした上で、16行目のexecuteUpdateメソッドでSQL文を実行しています。その後、17行目で2つあるうちの1つだけのパラメータを上書きし、再度実行しているため、2つ目のパ

ラメータは残ったままのものを利用していることになります。そのため、データベースには「100, "SAMPLE'」および「101, 'SAMPLE"」という2つの行が挿入されます。以上のことから、選択肢**C**が正解です。

41. A

➡ P557

JDBCの**接続文字列**の書式は次のとおりです。

【接続文字列の書式】
jdbc:DBMS名://<サーバーのIPアドレス>:<ポート番号>/データベースの場所

よって、選択肢**A**が正解です。
JDBCの接続文字列は、選択肢BやCのようなURLではありません。また、選択肢Dも接続文字列の書式と異なります。以上のことから、これらの選択肢は誤りです。

42. B、C

➡ P557

オートボクシングや**オートアンボクシング**は、プリミティブ型のデータをラッパークラス型変数で扱ったり、その逆を実現する機能です。

選択肢Cは、int型の値をそのラッパークラスであるInteger型の変数に代入しています。オートボクシングによって、このように本来は互換性がないint型の値を参照型で扱うことができます。選択肢Bも同様に、long型の値をそのラッパークラスであるLong型の変数に代入しています。整数リテラルのデフォルトの型はint型ですが、「L」という接尾辞を付けるとlong型として扱われます。よって、選択肢**B**と**C**は正解です。

選択肢Aは、1.0という浮動小数点数リテラルをFloat型の変数に代入しています。float型の値であれば、オートボクシングによってFloat型で扱えますが、浮動小数点数リテラルのデフォルトの型はdouble型です。double型の値をFloat型では扱えません。リテラルがfloat型であることを明示するには、数値の後ろに「1.0F」のように接尾辞を付けます。よって、選択肢Aは誤りです。
選択肢Dは、int型の整数リテラルをDouble型の変数に代入しようとしているため誤りです。

43. D

➡ P557

ストリームから商品の種類が何件あるかを数えるには、要素が商品のストリームから、商品の種類だけのストリームにして、**重複**をなくした上で、ストリーム内の要素数を数えます。

ストリーム内の要素を変更するには、**mapメソッド**を使います。**filterメソッド**では、要素を条件で絞り込むことはできますが、ストリームの要素の種類を変更することはできません。また、選択肢Aは、filterメソッドにboolean型を戻す条件式ではなく、商品の種類を戻すメソッド参照を指定しています。そのため、選択肢Aはコンパイルエラーとなります。

選択肢Bは、mapメソッドで商品の種類だけのストリームを構築し、重複を許さないSetオブジェクトに格納しています。しかし、**collectメソッド**はストリーム・パイプラインの終端操作であることから、Setオブジェクトを戻してストリーム・パイプラインの処理は終了してしまいます。そのため、**countメソッド**でストリーム内の要素を数えることはできません。

選択肢Cは、商品の種類だけのストリームを構築してはいますが、重複したままとなります。そのため、商品の数は数えられますが、種類の数を数えることはできません。選択肢Dでは、**distinctメソッド**を使って重複を排除しています。以上のことから、選択肢**D**が正解です。

44. D

➡ P558

java.util.Listインタフェースの**replaceAllメソッド**は、デフォルトメソッドの1つで、リスト内の各要素に対し、引数として受け取ったUnaryOperator型のラムダ式を適用するメソッドです。
UnaryOperatorは、単項演算子の処理を定義するための関数型インタフェースです。単項演算子の処理なので、扱える値は1つだけです。2つ扱おうとしている選択肢Aは、**BinaryOperator**で利用できるラムダ式です。よって、選択肢Aは誤りです。
設問のコードのUnaryOperatorは、ジェネリクスの型パラメータにInteger型を指定しています。**ジェネリクス**を利用するときに**ボクシング**は使えません。そのため、選択肢Bのように引数の型宣言にint型を指定することはできません。よって、選択肢Bも誤りです。
ラムダ式で変数のデータ型を宣言するには、データ型と変数名をカッコ「()」で囲まなければいけません。これは、**varを使った型推論**の場合でも同じです。よって、選択肢Cは誤りです。
以上のことから、処理する値が1つだけで、ラムダ式を正しく記述している選択肢**D**が正解です。

45. B

➡ P558

java.util.stream.IntStreamインタフェースの**generateメソッド**は、int型の値を戻すIntSupplier型の引数を受け取って、引数のラムダ式が戻す値を含むランダムな整数のストリームを生成するメソッドです。
選択肢Aは、generateメソッドに渡しているのがメソッド参照になっていません。よって、コンパイルエラーとなります。
乱数を生成する**java.util.Randomクラス**の**nextIntメソッド**は、staticメソッドではなく、インスタンスメソッドです。そのため、「クラス名::メソッド名」でメソッド参照することはできません。よって、選択肢Cはコンパイルエラーとなります。
整数ストリームを作り出す**IntStream**は、インタフェースです。そのため、newを使ってインスタンス化できません。よって、選択肢Dもコンパイルエラーとなります。
以上のことから、選択肢**B**が正解です。

46. A、B

➡ P559

java.util.stream.Collectorsクラスの**averagingDoubleメソッド**は、ストリームの要素をdouble型に変換し、平均を求めることができます。そのため、数値のリストの平均を求めるのに最も適しているでしょう。よって、選択肢**A**は正解です。

java.util.stream.IntStreamインタフェースのaverageメソッドは、ストリーム内の数値の平均を求め、OptionalDouble型の結果を戻します。**java.util.OptionalDoubleクラス**の**getAsDoubleメソッド**は、結果の値をdouble型で戻します。選択肢Bは、**mapToIntメソッド**で、ストリーム内の数値をそのままIntStream型のストリームに変更しています。これは、コレクションのstreamメソッドやparallelStreamメソッド、もしくはストリームのparallelメソッドで得られるStream型には、平均を求めるaverageメソッドがないためです。よって、選択肢**B**は正しく平均を求めることができます。一方、存在しないaverageメソッドを呼び出している選択肢CやDは、コンパイルエラーとなります。

前述のとおり、IntStreamインタフェースのaverageメソッドは、OptionalDouble型の結果を戻します。OptionalDoubleは結果を保持するクラスであり、並列ストリームを取得するためのparallelメソッドは持っていません。よって、選択肢Eは誤りです。

47. B

➡ P559

java.util.stream.Streamインタフェースの**anyMatchメソッド**は、このストリーム内のいずれかの要素が引数に一致するかどうかをboolean型で戻します。選択肢Aのように、Optional型の戻り値は戻しません。よって、選択肢Aは誤りです。

Streamインタフェースの**filterメソッド**は、ストリーム・パイプラインの中間操作のためのメソッドであり、Stream型の戻り値を戻します。Optional型の結果は戻しません。よって、選択肢Cは誤りです。

Streamインタフェースの**findAnyメソッド**は、ストリーム内の要素を1つ持つOptional型の結果を戻します。String型の結果は戻しません。よって、選択肢Dは誤りです。

以上のことから、選択肢**B**が正解です。このコードでは、filterメソッドでGという文字を持った文字列に絞り込み、順番にコンソールに表示しているだけです。このストリームは、**並列ストリーム**ではないため**forEachメソッド**が使えますが、**forEachOrderedメソッド**を使っても問題はありません。

48. A

➡ P560

java.util.Comparatorインタフェースの**comparingメソッド**は、並べ替えで利用するキーを受け取り、そのキーを使って並べ替えるComparator型オブジェクトを戻すメソッドです。設問のコードでは、Itemクラスのprice（価格）を並べ替えに使うComparator型オブジェクトを生成しています。価格を使って並べ替えると、apple、orange、bananaの順になります。
Comparatorの**thenComparingメソッド**は、直前のcomparingメソッドが比較対象の2つが等しいと判断した場合、追加の条件で並べ替えるComaprator型のオブジェクトを戻します。設問のコードでは、Itemクラスのpriceが等しい場合には、name（名前）を使って並べ替えるComparator型オブジェクトを生成します。appleとorangeは価格が等しいため、thenComparingメソッドで与えられた追加条件を使ってこの2つの並べ替えが実行されますが、名前順であればappleはorangeよりも前に位置するので、順番は変わらず、apple、orange、bananaとなります。
設問のコードでは、さらに**reversedメソッド**を使って、比較条件に従って並べ替えたものを反転しています。そのため、結果は、banana、orange、appleの順になります。以上のことから、選択肢**A**が正解です。

49. A

➡ P561

java.util.concurrent.ExecutorServiceインタフェースの**submitメソッド**はオーバーロードされており、java.util.concurrent.Callable型もしくはjava.lang.Runnable型のラムダ式を引数として受け取ります。よって、選択肢CとDは誤りです。

CallableとRunnableの違いは、戻り値の有無です。Runnableは戻り値を戻せません。戻り値を戻せるのは、Callableです。設問のコードのようにFuture型の戻り値を受け取るには、Callable型の引数を受け取らなくてはいけません。よって、選択肢**A**が正解です。

50. C

➡ P561

java.util.streamパッケージの**Streamインタフェース**や**IntStreamインタフェース**などの**iterateメソッド**は、第1引数に初期値、第2引数に初期値に対して何らかの演算を行うUnaryOperator型のラムダ式を渡すことで、ラムダ式の処理を実行しながら、初期値から始まる無限に続くストリームを構築することができます。設問のコードであれば、初期値0から始まり、1つずつ増える無限の数列を作り出すことができます。

設問のコードでは、**limitメソッド**を使ってストリーム内の要素を5つに限定しているため、0〜4までの数値がこのストリームの要素になります。その後、**mapToObjメソッド**で数値の後ろにカンマを追加した文字列のストリームに変換し、それをコンソールに表示しています。そのため、コンソールには「0,1,2,3,4,」と表示されます。よって、選択肢**C**が正解です。

51. D

➡ P562

モジュールでパッケージを利用できるようにするには、**モジュール定義ファイル**に**exportsディレクティブ**でパッケージ名を記述します。選択肢Dのモジュール定義ファイルでは、各モジュールが公開するパッケージを指定しています。よって、選択肢**D**が正解です。

52. A、B

➡ P563

インナークラスのインスタンス化に関する問題です。**インナークラスのインスタンス**を作るには、次のコードのようにインナークラスを持つ**エンクロージングクラスのインスタンス**を生成し、そのインスタンスを使います。

```
var subject = new Sample().new Subject("A");
```

もしくは、インナークラスを**static**で修飾することで、エンクロージングクラスのインスタンスを生成しなくても、直接インスタンス化できるようになります。よって、選択肢**A**は正解です。
なお、インナークラスではなく別のクラスとして定義すれば、通常のクラスとして扱えます。よって、選択肢**B**も正解です。

53. A

➡ P564

ストリーム内の要素を使って数値の合計を算出するには、**reduceメソッド**を使って累積値を求めるか、**sumメソッド**を使って合計値を求めるという二通りの方法があります。設問のコードは、前者の方法でappleとbananaの価格の合計を算出しています。

選択肢Aは、getPriceメソッドの戻り値である価格だけのIntStream型ストリームを構築し、sumメソッドを使って合計値を求めています。よって、設問のコードと同じ結果となります。以上のことから、選択肢**A**が正解です。

ストリーム内の要素に対して一括処理をする**mapメソッド**は、Function型のラムダ式を受け取ります。getPriceメソッドのメソッド参照は、Function型で扱うことはできません。そのため、選択肢Bはコンパイルエラーとなります。

reduceメソッドはオーバーロードされており、BinaryOperator型の引数だけを受け取るものはOptional型の戻り値を戻します。任意の型の戻り値を戻したい場合には、第1引数に戻り値型として利用したい型のデータを渡します。設問のコードであれば、第1引数に0を渡しているため、戻り値型はint型になります。選択肢Cは、引数にBinaryOperatorしか渡していないため、戻り値型はOptional型になります。OptionalクラスのifPresentメソッドは戻り値を戻しません（void）。よって、選択肢Cはコンパイルエラーとなります。

BinaryOperatorを受け取り、累積値を求めるreduceメソッドは、java.lang.stream.Streamインタフェースのメソッドで、IntStreamインタフェースには存在しません。選択肢Dのコードは、**mapToIntメソッド**を使って、数値だけのIntStream型ストリームを構築しているため、reduceメソッドが使えずコンパイルエラーとなります。

以上のことから、選択肢B、C、Dは誤りです。

54. C

➡ P565

メソッド参照は、すでにあるメソッドの定義をラムダ式の代わりに関数型インタフェース型の変数に入れて利用するものです。一方、**ラムダ式**は、一時的に使うために行う処理を関数型インタフェース型の変数に入れて利用します。ラムダ式の**アロー演算子**「->」の先には、処理ブロックを記述します。選択肢Aのように、メソッド参照を記述することはできません。また、ラムダ式で受け取る引数の型宣言をするには、カッコ「()」で囲まなければいけません。よって、選択肢Dは誤りです。

Consumer型はジェネリクスの型パラメータを受け取ります。選択肢Bのように型パラメータを渡さなかった場合には、受け取る引数はObject型として解釈されるため、String型の引数を受け取ることはできません。よって、選択肢Bは誤りです。

以上のことから、選択肢**C**が正解です。

55. C

➡ P565

java.time.format.DateTimeFormatterクラスのofPatternメソッドは、日付の書式を表すパターンを文字列として受け取ります。選択肢AとBは、「年」や「日」という文字列リテラルをダブルクォーテーション「」で囲んでいないためコンパイルエラーとなります。よって、これらの選択肢は誤りです。

パターン文字の間に入る「年」や「月」「日」などがパターン文字ではないことを示すためには、選択肢CやDのように**シングルクォーテーション**「」で囲みます。
「'月'd'日'」の「日」を表すには、「d」を使います。**dの数は、表示する最小桁数**を表します。「d」だけの場合、最小桁数は1桁となるため、たとえば「'日'」のように1桁の日は「1」、「10日」のように2桁の日は「10」と表示されます。「dd」の場合は、最小桁数が2桁になります。そのため、「1日」のように1桁の日は「01」、「10日」のように2桁の日は「10」と表示されます。
設問では、「4月1日」と日を1桁で表示しなければいけません。そのため、「d」と1つだけで記述する必要があります。以上のことから、選択肢**C**が正解で、選択肢Dは誤りです。

56. B

➡ P566

switch文の条件に**列挙型**を指定した場合、case式には**列挙子**を記述します。よって、選択肢**B**が正解です。「Type.A」のように、完全名で記述することはできません。選択肢Dのように文字列に変換してしまうと、条件式と型が一致せず、case式の比較ができなくなるためコンパイルエラーとなります。よって、選択肢Dも誤りです。なお、選択肢Aの**valueOf**は、文字列からEnum型のインスタンスを生成するためのメソッドですが、引数が必要です。よって、選択肢Aは誤りです。

57. A、C

➡ P567

java.util.LocaleクラスのsetDefaultメソッドには、オーバーロードされたメソッドが2つあります。1つは、Localeの参照を受け取るもの、もう1つは、**Localeのカテゴリ**を指定して、Localeの参照を受け取るものです。よって、選択肢**A**と**C**が正解です。
選択肢Bは、文字列とLocaleの参照を受け取っています。選択肢Dは、Localeのカテゴリは指定していますが、Localeへの参照ではなく文字列を渡しています。選択肢Eは、文字列を引数に渡しています。選択肢B、D、Eのようなオーバーロードは存在しません。

58. A

➡ P567

java.lang.Systemクラスのinフィールドは、標準入力を扱うInputStream型のstaticなフィールドです。このストリームは、JVMが開始した時点で開かれます。よって、選択肢**A**が正解です。
Systemクラスの**outフィールド**は、PrintStream型のstaticなフィールドです。また、このフィールドもinフィールドと同じように、JVMが開始した時点で開かれます。よって、選択肢CとDは誤りです。
Systemクラスの**setInメソッド**を使えば、ストリームを置き換えることができます。よって、選択肢Bは誤りです。

59. B、D

➡ P567

設問のコードでは、**parallelStreamメソッド**を使っていることから、**並列ストリーム**を作っていることがわかります。並列ストリームの処理順は、プログラムで制御することはできません。そのため、**forEachメソッド**による処理を実行すると、順番は毎回変わります。順番を維持したい場合は、forEachではなく、**forEachOrderedメソッド**を使います。以上のことから、選択肢**B**と**D**が正解です。

60. D

➡ P568

ソートなどで比較されるオブジェクトに、比較のための機能を持たせるには、**比較可能であることを示すjava.lang.Comparableインタフェース**を実現するようにします。もし、Comparableインタフェースを実現しないクラス

を比較しようとすると、Comparableに型変換しようとして失敗し、java.lang.ClassCastExceptionが発生します。
選択肢AとBの**java.util.Comparatorインタフェースは、比較する機能を持たない2つのオブジェクトを比較するためのもの**です。
Comparableインタフェースには**compareToメソッド**が定義されており、実現クラスはこのメソッドをオーバーライドしなければいけません。Comparableは、自分自身と対象を比較するためのインタフェースですので、選択肢Cのように2つの引数を受け取って比較することはありません。これはComparatorを実現しているクラスがやるべきことです。以上のことから、選択肢**D**が正解です。

61. C

➡ P569

設問のコードは、**moveメソッド**でsample.txtをdestディレクトリ内に移動しているコードに見えますが、実際にはsample.txtをdestという名前で上書きしようとしています。しかし、すでにdestディレクトリが存在するため、**java.nio.file.FileAlreadyExistsException**がスローされます。よって、選択肢**C**が正解です。
もし、destディレクトリ内に移動したいのであれば、5行目を「dest/sample.txt」のように変更し、パスを作る必要があります。

62. B、C

➡ P570

SPI（Service Provider Interface）に関する問題です。
設問のモジュール定義から、clientモジュールがapiモジュールを使っているものの、apiモジュールではインタフェースのみが提供されており、インタフェースの実現クラスは、providerモジュールが提供しているという関係であることがわかります。
SPIとしてインタフェースを実現するモジュールの定義ファイルでは、**requiresディレクティブ**でモジュールを要求し、**provides〜withディレクティブ**で、どのモジュールのインタフェースを、どのクラスで実現するかを指定します。provides〜withディレクティブで唯一の実装を宣言することはできません。よって、選択肢Aは誤りです。

SPIとして公開するインタフェースは、**usesディレクティブ**で指定します。clientモジュールでは、SPIとして公開するインタフェースを定義していますが、実現するモジュールへの利用を定義するrequiresが抜けています。そのため、providerモジュールをrequiresする必要があります。よって、選択肢**B**は正解です。

SPIを実現したモジュールでは、利用してもらうパッケージを**exports**する必要があります。設問のモジュール構造であれば、providerモジュールでcom.test.implパッケージをエクスポートする必要があります。よって、選択肢**C**は正解です。

apiという独立したモジュールに配置しても、利用するclientモジュールがどのモジュールをrequiresするかによって決まるため、com.sample.apiのAPIを使用する複数のモジュールのインストールが容易になることはありません。よって、選択肢Dは誤りです。

63. A

➡ P570

java.util.functionパッケージの**ToDoubleFunction**や**LongConsumer**は、**プリミティブ型**のデータを扱うための関数型インタフェースで、FunctionやConsumerをプリミティブ型のデータを扱うように特化したものです。FunctionとConsumerのうち、**戻り値を戻すのはFunction**であり、ToDoubleFunctionは、double型の戻り値を戻す関数型インタフェースです。
Consumerは、「消費者」という意味のとおり、**引数を受け取って処理**するための関数型インタフェースです。LongConsumerは、long型の引数を受け取って処理するための関数型インタフェースです。
以上のことから、選択肢**A**が正解となります。
なお、**Supplier**や**BiFunction**は、**参照型**を扱うための関数型インタフェースです。プリミティブ型のデータを扱うためのものではありません。よって、選択肢BとDは誤りです。

64. C

➡ P571

クラス宣言を**final**で修飾すると、そのクラスを継承したサブクラスを定義できなくなります。設問のコードでは、Bクラスがfinalで修飾されているため、このクラスを継承したサブクラスは定義できません。

コンストラクタは、明示的に定義されていなければデフォルトコンストラクタが追加されるため、コンパイルエラーとなることはありません。よって、選択肢Aは誤りです。
java.lang.Objectクラスに定義されている**toStringメソッド**は、オーバーライドされることを前提として使うメソッドですが、抽象メソッドではなく、具象メソッドであるため、オーバーライドしなくてもコンパイルエラーになることはありません。よって、選択肢Bは誤りです。
finalクラスは、そのクラスを継承したサブクラスを定義できないだけであって、finalクラスそのものが何かを継承することを制限するものではありません。よって、選択肢Dは誤りです。
以上のことから、選択肢**C**が正解です。

65. B ➡ P571

関数型インタフェースは、ラムダ式を扱うためのインタフェースで、**抽象メソッドを1つだけ**持っていなければいけません。抽象メソッドが複数あると、ラムダ式がどのメソッドの実装を意味するのかが判断できなくなるため、関数型インタフェースとしては使えなくなります。
抽象メソッドは1つである必要がありますが、privateなメソッドやstaticなメソッド、デフォルトメソッドなどの具象メソッドは自由に定義できます。
以上のことから、選択肢**B**が正解です。

66. B ➡ P571

選択肢Aは、Function型の変数を宣言するときに**ジェネリクスの型パラメータ**を渡していません。そのため、Object型を扱うFunction型変数であると解釈されます。java.lang.StringクラスのtoUpperCaseメソッドは、引数にString型のデータを受け取り、String型の戻り値を戻します。Object型の引数や戻り値を戻すことはできません。そのため、「String::toUpperCase」というメソッド参照を、Object型を扱うFunction型変数funcに代入することはできません。よって、選択肢Aは誤りです。

java.util.function.Functionインタフェースは、ジェネリクスの型パラメータを2つ受け取り、引数の型と戻り値の型として利用します。選択肢Cは、型パラメータを1つしか受け取っていません。よって、この選択肢は誤りです。

選択肢DのUnaryOperatorについても、選択肢Aと同様にジェネリクスの型パラメータを渡していません。そのため、ラムダ式の引数sはObject型として解釈されますが、ObjectクラスにはtoUpperCaseメソッドは存在しません。よって、この選択肢も誤りです。
以上のことから、「String::toUpperCase」というメソッド参照を、String型を扱うUnaryOperator型変数funcに代入している選択肢**B**が正解です。

67. B ➡ P572

ジェネリクスに関する問題です。
設問のSampleクラスでは、**ジェネリクス**を利用するTestクラスのインスタンスを2つ生成しています。1つは型パラメータを受け取らないインスタンス、もう1つはString型の型パラメータを受け取るインスタンスです。**型パラメータ**を受け取らなかった場合には、**Object型を扱う**ものとして解釈されます。
Sampleクラスでは、ジェネリクスの型パラメータを受け取り、String型だけを扱うTest型のtest2のcountメソッドに、String配列型のarrayと文字列の「A」が渡され、文字列に一致する数をコンソールに表示します。配列には1つだけAがあるため、コンソールには1が表示されます。
次に、Object型を扱うTest型のtestのcountメソッドにString配列型のarrayと文字列「A」が渡され、文字列に一致する数をコンソールに表示します。配列には1つだけAがあるため、コンソールには1が表示されます。最後も、Object型を扱うTest型のtestのcountメソッドにInteger配列型のarray2と数値1が渡され、一致する数をコンソールに表示します。配列には1つだけ「1」があるため、コンソールには1が表示されます。
以上のことから、選択肢**B**が正解です。

68. A ➡ P573

ジェネリクスとダイヤモンド演算子に関する問題です。
ダイヤモンド演算子「<>」は、変数の型パラメータを利用して、**インスタンス生成時の型パラメータの型を推論**する機能です。宣言した変数の**ジェネリクス**が**型パラメータ**を持たなかった場合は参照する情報がないため、**java.lang.Object型**を型パラメータに使うものと見なされます。よって、選択肢BとCのComparatorの型パラメータには、Object型が使われることになります。このため、これらのcompareメソッドの引数は、Integer型ではなくObject型でなくてはいけません。よって、選択肢BとCは誤りです。
ダイヤモンド演算子は、インスタンス生成時の型パラメータを省略するものです。よって、選択肢Dのように変数宣言で使うことはできません。
以上のことから、文法上の誤りがない選択肢**A**が正解です。

69. A、C ➡ P573

セキュアプログラミングの基本は**入力検証**です。信頼できない入力は、できる限り処理しないようにすべきです。信頼できるドメイン固有のライブラリを使用して入力を検証したり、有効な文字および入力値のみを受け入れるといったことを実施します。

選択肢Bは、改ざんにあたります。
入力値の検証は、できるだけ画面などのユーザーに近いプログラムが担当すべきです。不正な値をデータベースまで送ることがあってはいけません。よって、選択肢Dは誤りです。
セキュリティの基本は多重防御です。選択肢Eのようにすでに検証済みだと仮定して、何も行わないことはセキュリティホールになり得ます。よって、選択肢Eも誤りです。
以上のことから、選択肢**A**と**C**が正解です。

70. A、B、G

➡ P574

セキュアなオブジェクトを設計するにあたって、オブジェクトの不変性は重要な設計上のポイントです。**可変オブジェクト**は、フィールドの値を自由に変えることができてしまうため、ある時点では安全でも、別のタイミングでは想定外の変更を加えられてしまっている可能性があります。
不変オブジェクトを定義するには、フィールドを保護するためにアクセス修飾子を**private**にし、設定した値が変更されないことを保証するために**final**で修飾します。フィールドの値を変更するメソッドは、変更後の新しい値を持った新しいインスタンスを生成して戻すようにします。フィールドの値をコピーして戻すgetterメソッドなどは、**フィールドの値のコピーを作り、その参照を戻す**ようにします。
また、**不変オブジェクトの初期化を行うコンストラクタ**では、受け取った引数をそのまま利用しないほうがよいでしょう。受け取った引数が参照である場合、その参照は外部のクラスが共有していることになります。共有している参照を使えば、自由にフィールドの値を変更できることと実質的に等しい状態になるからです。
以上のことから、選択肢**A**、**B**、**G**が正解です。

選択肢CやEは、マルチスレッド環境下で利用されるオブジェクトをスレッドセーフにするための方法です。選択肢Dのようにコンストラクタをprivateにしてしまうと、インスタンスを生成できなくなります。選択肢Fのようにメソッド名に「$」を使ってもセキュアなオブジェクトにはなりません。よって、これらの選択肢は誤りです。

71. A、C

➡ P574

機密情報を持つインスタンスをシリアライズするときは、内部の機密情報を保持しないように処理しなければいけません。一度、ファイルなどにシリアライズされてしまえば、デシリアライズするだけで機密情報が漏れてしまうからです。
インスタンスのどのフィールドをシリアライズに含めるかを制御するには、機密情報を扱うフィールドに**transient**を付けます。そのほかに、**serialPersistentFields**フィールドを定義して、シリアライズされるべきフィールドを指定し、機密情報を含むフィールドを除外する方法もあります。以上のことから、選択肢**A**と**C**が正解です。
シリアライズ・プロキシ・パターンを利用した場合、**writeReplaceメソッドとreadResolveメソッドはセットで定義**しなければいけません。そのため、どちらか一方だけを実装するとした選択肢DとEは誤りです。

72. B

➡ P575

java.util.stream.Streamインタフェースの**collectメソッド**は、ストリーム内の要素の集合を戻す**可変リダクション処理**を実行するためのものです。引数には、可変リダクションを実現するjava.util.stream.Collector型の引数か、Collectorを作るために必要なSupplierやBiConsumerなどの引数を受け取るものの2つがオーバーロードされています。コレクションを受け取るオーバーロードは存在しません。よって、選択肢Bはコンパイルエラーとなります。以上のことから、選択肢**B**が正解です。

73. C

➡ P575

関数型インタフェースの1つである**java.util.function.Function**は、引数と戻り値の型を**ジェネリクスの型パラメータ**として受け取ります。よって、型パラメータを2つ受け取る選択肢**C**が正解です。選択肢Bも型パラメータを2つ受け取っていますが、型パラメータに使えるのは参照型のみです。

74. D

➡ P576

設問のassert文では、nの値が0より小さい、もしくはarrayのn番目の要素の値が、次の要素の値以下でなければいけません。
設問のコードは、配列内の要素を変数tmpに取っておき（9行目）、配列のi番目の要素とその前の要素（n番目）を比較して（12行目）、前の要素のほうが大きければ要素の値を入れ替えます（14行目）。その後、取っておいた変数tmpの値を入れ替えた値に戻すことで、並べ替えを実現しています（18行目）。そのため、正しく並べ替えができたかどうかを確認するのは、code 4のタイミングとなります。以上のことから、選択肢**D**が正解です。

75. B

➡ P577

java.util.stream.IntStreamインタフェースの**rangeメソッド**は、整数のストリームを構築するメソッドです。第1引数に「どの値から開始するか」という初期値、第2引数に上限となる値を渡すと、上限の値を含まない範囲で、初期値から1つずつ増やした整数のストリームを構築します。設問のコードでは、0から始まり、変数listかlist2のサイズの小さいほうを上限としています。変数listとlist2で参照できるリストは、どちらも要素数が4つですので、0、1、2、3の数値から成るストリームが構築されることになります。
その後、ストリームに対して、**mapToObjメソッド**で2つのリストからそれぞれ0、1、2、3番目の要素を取り出しながら文字列連結を行い、文字列だけのストリームを作り直して、**forEachメソッド**でコンソールに表示しています。
そのため、コンソールには、選択肢Bのように表示されます。

76. C

➡ P578

設問のコードでは、Sampleクラスが**Serializableインタフェース**を実装していることから、このクラスがシリアライズやデシリアライズされる対象であることがわかります。
すべてのフィールドの書き出しや復元をしないのであれば、シリアライズは**writeObjectメソッド**で、デシリアライズは**readObjectメソッド**で、引数として受け取ったObjectOutputStreamやObjectInputStreamに対して、書き込みや読み込みを行います。
これらのメソッドは、シリアライズやデシリアライズを行う内容を記述するものです。そのため、これらのメソッドが呼び出されるのは、シリアライズやデシリアライズの前のタイミングとなります。よって、シリアライズ後やデシリアライズ後とした選択肢AやBは誤りです。

設問のコードでは、readObjectメソッドを定義しています。このメソッドは、デシリアライズするタイミングで、読み込んだデータから復元すべきフィールドを制御するためのものです。よって、選択肢**C**が正解で、選択肢Dは誤りです。

77. B

➡ P579

設問のコードでは、**java.util.Locale.Builderクラス**のインスタンスを生成し、setterメソッドで言語を設定しています。ロケールは、言語と地域などの情報の組み合わせによって表現されます。設問のコードのように言語を「en」と指定した場合は、**リソースバンドル**で利用されるプロパティファイルは、**getBundleメソッド**で指定した基底名「sample」と組み合わせた「sample_en.properties」になります。よって、選択肢**B**が正解です。

78. E

➡ P580

java.util.function.Consumerインタフェースの**andThenメソッド**は、Consumerオブジェクトを**合成**し、連続した処理を行います。設問のコードでは、「A」「B」「C」の文字列を持つストリームを作り、ストリーム内の大文字の文字列をコンソールに出力したあとに、toLowerCaseメソッドで小文字に変換した文字列をコロン「:」でつなげて出力するという連続処理を実行しています。よって、選択肢**E**が正解です。

79. B

➡ P581

インナークラスのインスタンス化に関する問題です。
インナークラスをインスタンス化するには、まずインナークラスを囲むエンクロージングクラスのインスタンス化が必要です。インナークラスは、あくまでもインナークラスのメンバの1つであり、エンクロージングクラスのインスタンスを生成しなければ利用できないからです。
インナークラスのインスタンスを生成するには、「**エンクロージングクラスのインスタンスへの参照.new インナークラスのコンストラクタ**」と記述します。よって、選択肢**B**が正解です。

80. A

➡ P581

設問のセキュリティポリシーファイルでは、**AllPermission**が設定されています。これは、「すべての権限を許可する」過剰な権限であり、セキュリティ上、制限すべきでしょう。セキュリティの考え方は、すべて拒否して、許可するものだけを設定する「ホワイトリスト形式」が基本です。つまり、最小限の権限のみを許可することで、不要なセキュリティインシデントを防ぐことができます。以上のことから、選択肢**A**が正解です。