総仕上げ問題（ダウンロード版）
解答

1. C

➡ Q2

設問のコードでは、java.util.stream.Streamインタフェースの**mapToIntメソッド**によってストリーム内の文字列要素がそれぞれ「自身の文字数（length）」に置き換えられます（Blueは4、Redは3、Yellowは6、Greenは5）。次に、**filterメソッド**によって4文字以上（`len > 3`）の要素のみ抽出されます（4, 6, 5）。次の**limitメソッド**は、取り出す要素を、引数に指定された個数だけに限定するためのメソッドです。設問のコードでは要素を3個に限定しているため、抽出されたすべての要素「4, 6, 5」が、次のforEachメソッドの処理で画面に表示されます。したがって、選択肢**C**が正解です。

2. E

➡ Q2

設問のコードでは、リスト内のそれぞれの要素に対して**mapメソッド**を適用して、現在の値に1を加えた値に要素を置き換え、その要素を**peekメソッド**で表示しています（`[1, 2, 3]` → `[2, 3, 4]`）。したがって、選択肢**E**が正解です。

「// line n3」の行のcountメソッドは終端操作であり、ストリーム内の要素数をlong型として返します。しかし、このコードでは返された要素数「3」を表示する処理を記述していないため、要素数「3」が表示されることはありません（選択肢F）。

3. A、E

➡ Q3

インタフェースは主に**型**および**振る舞い**の仕様を規定し、互いに関連のないクラスがその仕様を実装することを想定して設計されます（選択肢**A**）。対する抽象クラスはインタフェースの役割を備えると同時に実装を提供することが可能であり、主に継承階層において関連するクラス間の共通の実装コードを1つにまとめる目的で使用されます（選択肢B）。

インタフェースで宣言するメソッドは、暗黙的にpublicとなります（選択肢C）。また、同様にフィールドは暗黙的にpublic static finalとなります（選択肢D）。

Javaではクラスの**多重継承**ができないため、複数のクラスが提供する実装コードを継承によって再利用することはできません。しかし、複数のインタフェースであればimplements可能であり、かつJava SE 8ではインタフェースがメソッドの実装を提供できるようになったため、抽象クラスの代わりにインタフェースを使用することで、多重継承のメリットを享受することができます（選択肢**E**）。

4. B

➡ Q3

設問のコードでは、待機させるスレッド数を3に設定した**CyclicBarrier**を生成しています。CyclicBarrierの第2引数には**Runnable**オブジェクトを渡すことができ、変数r1が参照する「ラムダ式で実装されたRunnableオブジェクト」を渡しています。この「// line n1」と「// line n2」の行はいずれも問題のないコードであり、コンパイルエラーにはなりません（選択肢C、D）。

次に、「`new Thread(r2).start()`」によって3つのスレッドが順次起動され、変数r2が参照するRunnableオブジェクトを実行します。各スレッドは最初に「Awaiting...」を表示し、「`barrier.await()`」を実行したスレッドは、順にその他のスレッドを待機することになります。このバリアは待機スレッド数が3になった時点で解除され、各スレッドは実行を再開します。したがって、「Awaiting...」が3回表示された後にバリアが解除され、「Let's go」が表示されます（選択肢**B**）。

「Let's go」を表示するRunnableオブジェクトは、バリアが解除される際に実行されるため、選択肢Aのような実行順序になることはありません。

5. A

➡ Q4

java.time.LocalDateクラスの**withYearメソッド**は、引数に指定された「年」だけを変更した新しいLocalDateオブジェクトを生成して返します。また、**withDayOfMonthメソッド**は引数に指定された「日」だけを変更した新しいLocalDateオブジェクトを生成して返します。

【メソッド定義】

```
public LocalDate withYear(int year)
public LocalDate withDayOfMonth(int dayOfMonth)
```

LocalDateクラスは**不変クラス**であり、これらのメソッドは元のオブジェクトの値を変更しません。したがって、設問のコードの変数ldが参照するLocalDateオブジェクトの値は変更されません（選択肢**A**）。選択肢Bを正解とするためには、以下のように値が変更された新しいオブジェクトを参照する変数を使用するコードを記述する必要があります。

例 値が変更された新しいオブジェクトを参照する変数を使用

```
11. LocalDate ld = LocalDate.of(2016, 9, 15);
12. LocalDate ld2 = ld.withYear(2017);
13. LocalDate ld3 = ld2.withDayOfMonth(25);
14. System.out.println(ld3);
```

あるいは、以下のように記述することもできます。

例 値が変更された新しいオブジェクトを参照する変数を使用

```
11. LocalDate ld = LocalDate.of(2016, 9, 15);
12. System.out.println(ld.withYear(2017).withDayOfMonth(25));
```

6. D、E

➡ Q4

設問のZクラスはYクラスを継承しているため、Zクラスのコンストラクタの先頭には、Yクラスの引数を取らないコンストラクタ・メソッド呼び出し「super();」がコンパイラによって挿入されます。しかし、Yクラスには引数を取らないコンストラクタが存在しないため、「super();」はコンパイルエラーとなります。この問題を解決するには、スーパークラスに存在するコンストラクタを呼び出すコードを明示的に記述する必要があります（選択肢**D**）。

また、ZクラスではdoItメソッドをオーバーライドしていますが、オーバーライドしたメソッドでは**可視性**を広げることはできても、狭くすることはできません。オーバーライド元となるYクラスのdoItメソッドはpublicであるため、Zクラスでオーバーライドする場合には、同じpublicにする必要があります（選択肢**E**）。

抽象クラスでは抽象メソッドを宣言することが可能ですが、抽象メソッドが1つも存在しない場合でも抽象クラスとして宣言することは問題ありません。また、抽象クラスからインスタンスを生成することはできませんが、コンストラクタを宣言することは可能です。したがって、クラスXは問題のないコードであり、選択肢Aのようにabstractキーワードを取り除く必要はありません。

すべてのコンストラクタの先頭には、明示的なコンストラクタ呼び出しを記述していない限り、スーパークラスの引数を取らないコンストラクタ呼び出し（super();）がコンパイラによって挿入されます。選択肢Bは明示的にそのコードを記述しているだけであり、問題解決にはなりません。

YクラスではXクラスのdoItメソッドをオーバーライドする際に可視性をprotectedからpublicに広げていますが、先述したように可視性を広げることは問題ありません（選択肢C）。

7. D

➡ Q5

設問のコードで使用している**flatMapToIntメソッド**の戻り値はIntStream型であり、引数に記述されているのはStream<String>型オブジェクトを返すラムダ式であるため、コンパイルエラーとなります（選択肢**D**）。

flatMapToIntメソッドは、ストリームの要素をint型の値に置き換えるために使用するメソッドです。flatMapToIntメソッドではなく、**flatMapメソッド**の呼び出しに修正すれば、選択肢Aが正解となります。設問のコードのような入れ子構造のストリームを展開して通常の平坦なストリームのように処理するには、flatMapメソッドを使用します。

なお、選択肢BやCのように表示するためには、filterメソッドを使用するなどの追加の処理が必要です。

8. A

➡ Q6

日付ベースの時間量を扱うクラスは**java.time.Periodクラス**であり、**betweenメソッド**によって、引数に渡した2つの日付（LocalDateオブジェクト）の間隔を求めることができます。betweenメソッドは時間量を表すPeriodオブジェクトを返すだけなので、日数や年数など具体的な値を取得するためには**getXxxメソッド**を使用する必要があります。設問の要件のように、年数を求めたい場合にはgetYearsメソッドを使用します。getYearsメソッドの戻り値はint型です（選択肢**A**、選択肢C）。

選択肢BやDで使用している**java.time.Durationクラス**は、日付ベースではなく時間ベースの時間量を表すクラスです。また、DurationクラスでofYearsというメソッドは提供されていません。**java.time.Instantクラス**は時間量ではなく単一時点を表すクラスであり、コンストラクタは提供されていません（選択肢E）。

9. C

➡ Q6

BiConsumer<T, U>は、T型とU型の2つの引数を受け取り、戻り値は返さない抽象メソッドを宣言する関数型インタフェースです。したがって、設問のコードのように3つの型引数を指定するとコンパイルエラーとなります（選択肢**C**）。また、BiConsumerで宣言されている抽象メソッド（SAM）は**acceptメソッド**であり、applyメソッドではありません。

BiConsumerではなくBiFunctionを使用し、ラムダ式を「`(i, d) -> (int)(i * d)`」のように記述すれば、「2」と表示されます（選択肢A）。また、BiFunctionの型引数を「`<Integer, Double, Double>`」にして、変数resultの型をdouble型にすれば、設問のコードのラムダ式のままで「2.4」と表示されます（選択肢B）。

10. B ➡ Q7

設問のコードは、問題なくコンパイルすることができ、実行すると「10」が表示されます（選択肢**B**）。内部クラスのインスタンスをstaticメソッドから利用する場合には、外側のクラスのインスタンス参照が必要です。「// line n2」の行ではOuterクラスのインスタンスを生成し、その参照を用いて内部クラスのインスタンス生成を行っています。また、内部クラスのprivateフィールドに対しては、その内部クラスが宣言されているクラスのメンバー・メソッドから直接アクセスすることが可能です。

11. B ➡ Q8

列挙型において宣言される列挙定数は、実際にはその列挙型のインスタンスを参照する定数です。また、列挙型は暗黙的にEnumクラスのサブクラスとしてコンパイルされるため、設問のコードにおけるinstanceof演算子の結果はすべて「true」となります。したがって、選択肢**B**が正解です。列挙型は何らかのクラスを継承することはできませんが、インタフェースをimplementsすることは可能です（選択肢D）。

12. C ➡ Q8

java.util.Mapインタフェースではstreamメソッドを提供していないため、設問のコードはコンパイルエラーとなります（選択肢**C**）。Mapオブジェクトをストリームapiで扱う場合には、**entrySetメソッド**を使用してSetオブジェクトを取得してからstreamメソッドを使用する必要があります。entrySetメソッドは**Set<Map.Entry<K, V>>オブジェクト**を返すため、このオブジェクトから**Map.Entryオブジェクト**を取り出してgetKeyメソッドとgetValueメソッドを使用してキーと値を処理することができます。また、**java.util.stream.Collectorsクラス**の**joiningメソッド**はストリーム内の要素を結合するメソッドであり、設問のコードのストリームを取得する行が以下のように記述されていれば選択肢Aが正解となります。

例 entrySetメソッドを使用してMap.Entryオブジェクトのストリームを生成

```
11. String result = itemMap.entrySet().stream()
```

13. A、C ➡ Q9

設問の選択肢のうち、正しい説明は選択肢**A**と**C**です。その他の選択肢は、以下の理由により誤りです。

B. IntSupplierインタフェースはSupplierインタフェースのプリミティブ・バー

ジョン（int型に特化）であり、宣言されている抽象メソッドはgetAsIntです。Supplierインタフェースであれば抽象メソッドはgetメソッドとなります。

D. UnaryOperatorインタフェースはFunctionインタフェースのサブインタフェースです。

E. BiFunctionインタフェースはFunctionインタフェースの特殊化バージョン（引数を2つ取る）であり、Functionインタフェースのサブインタフェースではありません。

F. Predicateインタフェースの型パラメータで宣言されている型変数は1つです。

14. C

➡ Q9

設問のコードのdoItメソッドは、Y型を上限境界とするワイルドカード型変数「?」を宣言しています。したがって、このワイルドカードに指定できる型引数はX型もしくはY型であり、Z型を指定することはできないため、「// line n3」の行でコンパイルエラーとなります（選択肢**C**）。なお、「// line n4」の行は型引数を指定せずにAクラスのインスタンスを生成していますが、このようなコードは警告が表示されてもコンパイルエラーにはなりません。

15. B

➡ Q10

設問のコードでは、try-with-resources文を使用してResourceAクラスとResourceBクラスのインスタンスを生成しています。両クラスは**AutoCloseableインタフェース**を正しく実装できているため、コードの実行がtry-with-resources文を抜ける際に、自動的に**closeメソッド**が呼び出されます。設問のようにtry-with-resources文の宣言で複数のインスタンスを生成（オープン）している場合、closeメソッドは**インスタンス生成の逆順**で呼び出されます。したがって、最初に「M」が表示された後、ResourceBのcloseメソッドが呼び出されて「Bc」が表示され、次にResourceAのcloseメソッドが呼び出されて「Ac」が表示されます。結果として「MBcAc」が表示されるため、選択肢**B**が正解です。

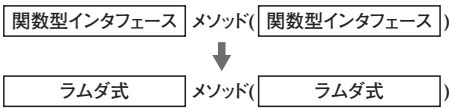
AutoCloseableインタフェースで宣言されている抽象メソッドはcloseメソッドのみです。try-with-resources文で自動的に呼び出されるのはcloseメソッドだけで、openメソッドは自動的に呼び出されることはありません（選択肢C、D）。

16. B

➡ Q11

設問の関数型インタフェース「Foo」では、Function<Integer, String>型オブジェクトを引数に取って、Function<String, String>型オブジェクトを戻り値として返す抽象メソッド「doIt」が宣言されています。このように、関数型インタフェース型オブジェクトを引数に受け取り、関数型インタフェース型オブジェクトを戻り値として返すメソッドは、「ラムダ式を受け取り、ラムダ式

を返すメソッド」として考えることができます。



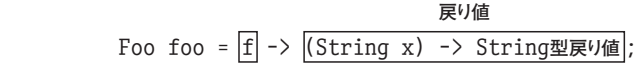
また、関数型インタフェースのSAMである抽象メソッドの実装はラムダ式で記述できるため、設問のFooインタフェース（doItメソッド）の実装は次のコードのように考えることができます。

さらに、ラムダ式における仮引数宣言の型とカッコ「()」は省略できるため、上記コードは以下のように記述することができます。

```
Foo foo = f -> ラムダ式
```

したがって、選択肢AやCのように仮引数宣言で「(f, x)」のように2つの引数を受け取る宣言は誤りであることがわかります。また、この2つの選択肢のコードは、ラムダ式を返すコードになっていない点にも注意しましょう。

次に、Fooインタフェースの実装における戻り値型は「Function<String, String>型」となっています。すなわち、String型引数を受け取り、String型戻り値を返すFunctionインタフェースの実装を以下に示すようなラムダ式で記述することができます。



設問の選択肢のどのコードも、受け取ったFunctionオブジェクトのメソッドを呼び出していますが、FunctionインタフェースのSAMとなる抽象メソッドはapplyメソッドであり、選択肢Cや選択肢Dのようなacceptメソッドではありません（acceptはConsumerインタフェースのSAMです）。したがって、選択肢CとDも正解から除外されます。

最後に、doItメソッドを実行している部分も見ておきましょう。このdoItメソッドの引数にはFunction<Integer, String>型オブジェクトを渡す必要があり

ます。これは、Integer型引数を受け取り、String型の戻り値を返すラムダ式で記述することができます。そして、このdoItメソッドの実行によって返されるFunction<String, String>オブジェクトのSAMであるapplyメソッドを呼び出すことによって、設問の要件である「#3$」が表示されることになります。この部分に関しては選択肢AとBはいずれも正しいコードですが、選択肢Aは先述したように誤りであるため、結果的に選択肢**B**が正解となります。

17. C ➡ Q11

現在の協定世界時（UTC）は、**java.time.ZonedDateTimeクラス**の**nowメソッド**にUTCを表すZoneIdオブジェクトを渡すことで取得できます。ZoneIdオブジェクトの生成にはいくつかの方法がありますが、ZoneIdのサブクラスであるZoneOffsetクラスで定義されている定数「UTC」を使用するのが簡単です。

例 UTCの取得

```
11. ZonedDateTime utc = ZonedDateTime.now(ZoneOffset.UTC);
```

また、Java SE 8で導入された新しいDate and Time APIのZonedDateTimeオブジェクトから旧来のjava.util.Dateオブジェクトを取得するには、ZonedDateTimeのtoInstantメソッド（ChronoZonedDateTimeインタフェースのdefaultメソッドを継承）を使用してエポック・ミリ秒を取得し、Dateクラスの**fromメソッド**を使用してDateオブジェクトを生成します。したがって、選択肢**C**が正解です。

ZoneIdクラスでは定数「UTC」は宣言されていません（選択肢A、B）。また、現在日時の取得には、ofメソッドではなくnowメソッドを使用します（選択肢B、D）。

18. C ➡ Q12

設問のMyListResourceBundleクラスが継承している**ListResourceBundleクラス**は、リソース・バンドルを作成するために利用できる抽象クラスです。ListResourceBundleクラスは、protected abstractとして宣言されている**getContentsメソッド**の実装を提供する必要があります。getContentsメソッドは、Object[][]としてリソース・バンドルにおけるプロパティリストを返す必要があります。したがって、選択肢**C**が正解となります。その他の選択肢は、いずれもObject[][]を返す実装となっていないため誤りです。

19. C

➡ Q13

設問のコードは、IntStream内の要素「1, 2, 3」に対してmapメソッドを使用して要素「1, 1, 1」に置き換え、その後sumメソッドですべての値を足し合わせています。この処理の結果は3となりますが、IntStream内の要素がどのようなint値であっても、結果は「要素の個数」となります。このような要素の個数を取得する場合には、**countメソッド**を使用することができます。したがって、選択肢**C**が正解です。countメソッドはint値ではなくlong値を返す点にも注意しましょう。

選択肢Bのdistinctメソッドは要素の重複を除くためのメソッドであり、選択肢Dのlimitメソッドは引数に指定された値の個数に要素を切り詰めるためのメソッドです。また、選択肢Aのようなtotalというメソッドはは IntStreamインタフェースでは宣言されていません。

20. C

➡ Q13

設問のコードではHashMapオブジェクトから**entrySetメソッド**を使用してMap.Entryオブジェクトを取得し、さらにそのentrySetを使用してArrayListオブジェクトを生成しています。その後、**java.util.Collectionsクラス**の**sortメソッド**を使用してArrayListオブジェクトの要素（**Map.Entryオブジェクト**）を並べ替えています。このsortメソッドの第2引数に指定しているラムダ式では、Map.Entryオブジェクトから取得した値の降順で並べ替えを行っています。したがって、選択肢**C**が正解となります。

21. D

➡ Q14

関数型インタフェースとラムダ式およびSAMとなる抽象メソッドに関する問題です。各選択肢に関する説明は以下のとおりです。

A. Function<T, R>インタフェースをパラメータ化する場合には、抽象メソッドapplyに対する「引数の型（T）」と「戻り値の型（R）」の2つを指定する必要があります。選択肢のコードでは1つしか型引数を指定していないため誤りです。「`Function<Integer, Double>`」と宣言していれば正解となります。

B. DoubleFunction<R>インタフェースは、double型の値を返すFunctionのプリミティブ特殊化型であり、パラメータ化する際には「戻り値の型（R）」を指定する必要があります。選択肢のコードではInteger型を指定していますが、ラムダ式の戻り値はdouble型となるため誤りです。「`DoubleFunction<Double>`」と宣言していれば正解となります。

C. 戻り値をDouble型としてDoubleFunction<R>インタフェースをパラメータ化しています。このステートメントに問題はありませんが、DoubleFunctionの抽象メソッドはapplyAsDoubleではなくapplyであるため誤りです。applyメソッドを使用していれば正解となります。

D. IntToDoubleFunctionインタフェースは、int値を受け取ってdouble値を返すFunctionのプリミティブ特殊化型であり、applyAsDoubleメソッドを宣言しています（非ジェネリック・インタフェース）。選択肢のコードに誤りはなく、実行すると「1.5」が表示ます。したがって、正解です。

Function<T, R>インタフェースの型パラメータ「T」と「R」は、それぞれ抽象メソッドapplyの「引数」と「戻り値」の型です。このFunctionのプリミティブ特殊化型として、以下のようなインタフェースが用意されています。

●抽象メソッドapplyの「戻り値の型（R）」のみを指定する特殊化型インタフェース

- IntFunction<R>
- LongFunction<R>
- DoubleFunction<R>

●抽象メソッドapplyAsXxxの「引数の型（T）」のみを指定する特殊化型インタフェース

- ToIntFunction<T>
- ToLongFunction<T>
- ToDoubleFunction<T>

※ applyAsXxxの「Xxx」の部分は戻り値の型（結果をXxx型として関数を適用する）を表す

●抽象メソッドapplyAsXxxの引数と戻り値、両方の型が決められているFunctionのプリミティブ特殊化型インタフェース（非ジェネリック）

- IntToDoubleFunction
- IntToLongFunction
- DoubleToIntFunction
- DoubleToLongFunction
- LongToIntFunction
- LongToDoubleFunction

※ applyAsXxxの「Xxx」の部分は戻り値の型（結果をXxx型として関数を適用する）を表す

Functionだけでなく、SupplierやConsumer、Predicateの基本となる4つの関数型インタフェースと、それらのプリミティブ特殊化となるインタフェース、そしてその実装として記述できるラムダ式、さらにそれぞれのインタフェースで宣言されている抽象メソッドをしっかりと覚えておきましょう。

22. B、D、E

➡ Q14

LocalDateTimeクラス（あるいはLocalDateクラスやLocalTimeクラス）は、そのオブジェクトが表す日時を構成するさまざまなフィールド値を個別に取得するためのゲッター・メソッドを提供しています。getYearメソッドはLocalDateTimeオブジェクトが表す日時における「年」をint値として返します（選択肢**B**）。getMonthメソッドは、同様に「月」をMonth型オブジェクトとして返します（選択肢**D**）。getDayOfMonthメソッドは、その月における「日」をint値として返します（選択肢**E**）。選択肢A、C、Fは、いずれも存在しないメソッドです。

年や月は絶対的な値ですが、「日」に関しては年や月、週における相対的な値として「getDayOfXxx（Xxxの部分はYear、Month、Week）」という名前のメソッドとして提供されている点に注意しましょう。

23. C

➡ Q15

設問のコードでは、関数型インタフェースBarのSAMであるdoItメソッドに対してFooクラスのコンストラクタ参照を代入しています。このdoItメソッドはint型の引数を取り、Foo型オブジェクトを返す定義となっているため、ラムダ式で記述した場合には「`i -> new Foo(i)`」となります。また、このラムダ式をコンストラクタ参照「`Foo::new`」で置き換えた場合には、暗黙的にint型の引数を受け取って、そのint値を取るオーバーロード・コンストラクタを呼び出す式となります。したがって、SAMであるdoItメソッドを呼び出した場合には「`10 -> new Foo(10)`」の式が実行され、結果としてint値を取るFooクラスのコンストラクタが呼び出されて「Foo(10)」が表示されます。以上のことから、選択肢**C**が正解です。

メソッド参照やコンストラクタ参照では、SAMである抽象メソッドの定義に一致した適切なオーバーロード・メソッドが自動的に選択されることを覚えておきましょう。

24. A

➡ Q15

設問のコードでは、ToIntFunctionの実装として「`str::indexOf`」というメソッド参照を記述しています。このメソッド参照は「`s -> str.indexOf(s)`」というラムダ式と等価であり、変数strが参照する文字列「Java8」において、引数sに指定された文字列の位置インデックスを返す処理となっています。また、ToIntFunction<String>はString型の引数を受け取って、int型の値を返す関数を表しており、「// line n1」の行は正しいメソッド参照の代入式となっています。ToIntFunctionインタフェースで宣言されている抽象メソッドはapplyAsIntであり、「// line n2」の行では文字列型の引数8を渡して「`str::indexOf`」（"8"

-> "Java8".indexOf("8"))を実行していることになり、結果としてインデックス「4」が表示されます。したがって、選択肢**A**が正解です。

25. C、E

➡ Q16

JDBCを使用したデータベース接続では、「jdbc:**データベース製品名**://**ホスト名**:**ポート番号**/**データベース名**」という形式の**接続URL**を使用する必要があります。この接続URLの中で先頭からホスト名（選択肢**C**）までの情報とデータベース名（選択肢**E**）は必須であり、省略することはできません。ポート番号（選択肢A）を省略した場合には、そのデータベース製品のデフォルト・ポート番号（原則としてIANAの登録済みポート番号）が指定されたものとみなされます。ユーザー名（選択肢D）やパスワード（選択肢B）はオプションの接続パラメータとして指定することも可能ですが、個別に指定することもでき、必須ではありません。

26. B

➡ Q16

java.util.function.Predicateインタフェースのdefaultメソッドとして提供されている**andメソッド**は、2つのPredicateを論理積（AND）としてストリームに適用します。また、同様にdefaultメソッドとして提供されている**negateメソッド**は論理否定（NOT）の機能を持つメソッドです。したがって、設問のfilterメソッドで指定している「p1.and(p2)」は、「偶数のみ（p1）」と「3よりも大きい（p2）」という2つのPredicateオブジェクトの論理積（AND）となり、結果のストリーム要素は4となります。さらに、その結果に対して論理否定（NOT）のnegateメソッドを適用しているため、最終的な結果は「1235」となります。したがって、選択肢**B**が正解です。

Predicateメソッドにはそれ以外に論理和（OR）の機能を提供するためのorメソッドもdefaultメソッドとして用意されていることも覚えておきましょう。

27. B

➡ Q16

設問のコードのListオブジェクトはMapオブジェクトを格納する宣言となっていますが、さらにこのMapオブジェクトはList<Integer>型のキーとList<String>型の値からなる要素を格納する「Map<List<Integer>, List<String>>」として宣言されています。

「// line n1」の行では、Listオブジェクトにnull要素を追加しています。ArrayListクラスは要素にnullを許容するため、この行は問題のないコードです。次の「// line n2」の行ではListオブジェクトにMapオブジェクトを追加していますが、このMapオブジェクトの型はキーと値にそれぞれIntegerとStringを指定したMap<Integer, String>であり、「Map<List<Integer>, List<String>>」

との互換性はありません。したがって、「// line n2」の行はコンパイルエラーとなります（選択肢**B**）。

28. B、C、D

➡ Q17

各選択肢に関する説明は以下のとおりです。

A. トランザクションの分離レベルに関する設定は、java.sql.DriverManagerクラスではなくjava.sql.Connectionインタフェースで宣言されている**setTransactionIsolationメソッド**を使用します。説明は誤りです。

B. Statementオブジェクトがcloseされるか、もしくは別のResultSetオブジェクトを取得した場合、ResultSetオブジェクトも自動的にcloseされます。正しい説明です。

C. データベースそのものに関連する情報は、Connectionオブジェクトの**getMetaDataメソッド**によって返されるDatabaseMetaDataオブジェクトから取得することができます。正しい説明です。

D. **JDBC 4.0**以降に準拠したドライバでは、プログラムで明示的にClass.forNameメソッドを使用したドライバのロードを行う必要はありませんが、ドライバの完全修飾クラス名を記述した「java.sql.Driver」ファイルをJARファイルの「META-INF/services」ディレクトリ内に配置しておく必要があります。通常、このファイルはデフォルトで配置されています。正しい説明です。

E. クエリの結果セットのレコードに対するカーソルの操作はResultSetMetaDataインタフェースではなくResultSetインタフェースで宣言されているメソッドを使用します。説明は誤りです。

したがって、選択肢**B**、**C**、**D**が正解です。

29. C

➡ Q17

Mapインタフェースの**putIfAbsentメソッド**は、第1引数に指定されたキーの要素がMap内に存在しない場合にのみ、そのキーと第2引数に指定された値からなる要素をMapに追加します。設問のコードでは、putIfAbsentメソッドを呼び出している時点で「Allan」というキーを持つ要素はすでに存在するため、値が「50」に置き換えられることはありません。

また、2つの引数を取る**removeメソッド**は、第1引数に指定されたキーと第2引数に指定された値からなる要素がMap内に存在する場合にのみ、その要素

を削除します。設問のコードではremoveメソッドで指定されているキー「Bob」とその値「30」からなる要素がMap内に存在するため、この要素は削除されます。この時点でMap内の要素は「`{Chris=40, Allan=20}`」となります（HashMapのため順序は保証されません）。

次に呼び出しているMapインタフェースの**replaceAllメソッド**はBiFunctionオブジェクトを引数に取り、その実装に従ってすべての要素の値を置き換えます。ここでBiFunctionの実装として記述されているラムダ式「`(k, v) -> k.length() + v`」では、キーの長さ（文字列長）と値を足し合わせた値を返しており、Map内のすべての要素の値は「`{Chris=45, Allan=25}`」と置換されます。したがって、選択肢**C**が正解です。

30. A ➡ Q18

ForkJoinPoolを使用してForkJoinTask（RecursiveActionあるいはRecursiveTask）を実行するためのメソッドは、invoke、submit、executeの3つが提供されています。これらのメソッドのうち、非同期で実行したタスクの終了を待機するのは**invokeメソッド**だけです。したがって、選択肢**A**が正解です。また、ForkJoinPoolで使用されるスレッド・プールはメインスレッドが終了すると自動的に終了します。ForkJoinPoolクラスではcomputeメソッドは提供されていません（選択肢D）。

31. C ➡ Q18

Streamインタフェースにおいて、ストリーム内の最大要素を返すメソッドは**maxメソッド**です。引数にはComparator<? super T>型オブジェクトを受け取り、戻り値はOptional<T>型として返します。したがって、選択肢**C**が正解です。IntStreamなどのプリミティブ・バージョンのストリームでは、最大値はデフォルトで自然順序によって判定されますが、Streamインタフェースの場合には判定基準となるComparatorを指定する必要があることを忘れないようにしましょう。

32. A ➡ Q19

設問のコードでは、MapオブジェクトからMap.Entryオブジェクトのストリームを生成して**sortedメソッド**でソートを行っています。sortedメソッドの引数にはMap.Entryオブジェクトの**comparingByValueメソッド**によるComparatorオブジェクトを渡しており、Mapオブジェクトの値によるソートが行われます（値が同じ場合にはキーによるソート）。値が文字列の場合、このメソッドの引数に何も指定しなければ大文字と小文字は識別されません。そのため、デフォルトでは選択肢Bのように大文字が先となりますが、設問のコードのようにStringクラスのCASE_INSENSITIVE_ORDERを渡した場合には

大文字と小文字の違いが無視されるため、値が同じ場合にキーでのソートが行われます。したがって、選択肢**A**が正解です。

33. C ➡ Q19

設問のコードではストリーム内の文字列要素の長さをlengthメソッドで取得し、mapメソッドを使用してストリーム「335」を生成しています。次に、reduceメソッドでそれらをすべて足し合わせているため、「11」が表示されます。したがって、選択肢**C**が正解です。

34. C ➡ Q20

設問のコードでは、**java.io.Closeableインタフェース**と**java.lang.AutoCloseableインタフェース**をそれぞれ実装したクラスを、try-with-resources文を使用してインスタンス生成を行っています。try-with-resources文を抜ける際には、インスタンス生成とは逆順にこれらのcloseメソッドが自動的に呼び出されます。また、finally句は常に実行されます。したがって、選択肢**C**が正解です。

35. B ➡ Q21

関数型プログラミングでは、複数の関数を組み合わせた関数を「合成関数」と呼びます。Functionインタフェースでは合成関数を作成するのと同じような機能を提供するdefaultメソッドとして、**andThen**メソッドと**compose**メソッドを提供しています。andThenメソッドは引数に指定したFunctionオブジェクトを後から適用し、composeメソッドは引数に指定したFunctionオブジェクトを先に適用します。したがって、設問のコードでは変数f2が参照する「`i -> i + 2`」が先に適用された「123」のストリームに対して、次に変数f1が参照する「`i -> i * 2`」が適用されるため、最終的なストリームは「246」となります。したがって、選択肢**B**が正解です。composeメソッドではなくandThenメソッドを使用した場合には選択肢Aが正解となります。

36. A ➡ Q21

Predicateインタフェースでは、equalsメソッドによる2つのオブジェクトの等価性を判定するPredicateオブジェクトを返すためのstaticメソッドとして**isEqualメソッド**が提供されています。設問のコードのラムダ式「`s -> s.equals("B")`」は、「`Predicate.isEqual("B")`」に置き換えることができます。したがって、選択肢**A**が正解です。

選択肢CのequalsメソッドはPredicateインタフェースのstaticメソッドとしては提供されていません。また、選択肢BとDはメソッド参照の構文仕様上、誤りです。

37. A

➡ Q22

java.time.temporalインタフェースで提供されている列挙型の**ChronoUnit**は、日付期間の単位を表す列挙定数を宣言しています。設問のコードで使用されているYEARSは、1年の期間単位を表す列挙定数です。そして、このChronoUnit列挙定数が表す期間をDurationオブジェクトとして取得するには、**getDurationメソッド**を使用することができます。DurationクラスのmultipliedByメソッドは、指定された引数でその単位を乗算するメソッドです。設問のコードでは引数に10を指定しているため、変数d1は10年の期間を表すDurationオブジェクトを参照することになります。

また、ChronoUnit列挙型では1年を表すYEARSのほかにも、10年を1つの単位として表すDECADESも提供しており、変数d2も10年の期間を表すDurationオブジェクトを参照することになります。したがって、この2つのDurationオブジェクトをequalsメソッドで比較した結果はtrueとなるため、選択肢**A**が正解です。

38. A

➡ Q22

java.time.MonthDayクラスは、日付に含まれる「月と日」の値を扱います。MonthDayオブジェクトの生成にはいくつかのファクトリ・メソッドが用意されていますが、設問のコードのように**ofメソッド**や**fromメソッド**を使用することができます。また、MonthDayオブジェクトの比較にはequalsメソッドを使用することができ、月と日の値が同じ場合にtrueが返ります。したがって、設問のコードでは変数mdは月の値が「10」、日の値が「1」であるMonthDayオブジェクトを参照しており、現在日付が「2016年10月1日」の場合に「`LocalDate.now()`」で得られる値を使用したMonthDayオブジェクトとの比較はtrueとなるため、選択肢**A**が正解となります。

39. A

➡ Q22

設問のコード12行目では、Streamインタフェースのreduceメソッドは引数を1つだけ渡しています。単一のBinaryOperatorオブジェクトを引数に取るreduceメソッドは、Optionalオブジェクトを返します。11行目ではString型変数に代入しようとしているため、コンパイルエラーとなります。したがって、選択肢**A**が正解です。String型ではなくOptional<String>型で変数を宣言し、「// line n2」の行でgetメソッドを使用して値を取り出すコードに変更した場合には選択肢Cが正解となります。

40. C

➡ Q23

java.io.BufferedReaderクラスの**readメソッド**はint値を返すため、String型変数を使用して値を代入しようとしているコードが原因でコンパイルエラーとなります。したがって、選択肢**C**が正解です。readメソッドではなく**readLineメソッド**を使用すればコンパイル・実行が可能となり、選択肢Bが正解となります。readLineメソッドは1行のテキストを読み込んでString値として返しますが、その内容にテキスト終端の改行コードは含まれません。また、ファイルへの書き込みに使用しているwriteメソッドは改行コードを書き込まないため、選択肢Aのように自動的に改行されることはありません。改行が必要であれば、明示的に改行コードを書き込むか、newLineメソッドを使用する必要があります（プラットフォームに応じた改行コードを自動的に判別して書き込むnewLineメソッドの使用が推奨されます）。

また、入力用のファイルは事前に存在する必要がありますが、出力用のファイルが存在しない場合には自動的に作成されるため、そのことが原因で例外（FileNotFoundException）がスローされることはない点も覚えておきましょう。ファイルがすでに存在する場合には上書きされます。

41. A、C

➡ Q23

従来の日付関連のAPIでは、月や曜日を数値として表していたため、ソースコードの可読性低下と潜在的バグを埋め込みやすいという問題がありました。Java SE 8の新しいDate and Time APIでは、月や曜日を**列挙定数**として表すことで従来のAPIの問題を改善しています。具体的には「月」を表す列挙定数は**Month**列挙型（選択肢**C**）で、「曜日」を表す列挙定数は**DayOfWeek**列挙型（選択肢**A**）で宣言しています。MonthOfYear（選択肢B）やWeek（選択肢D）という列挙型は提供されていません。

42. C

➡ Q24

インタフェースの**sumメソッド**は、ストリーム内の要素であるint値をすべて足し合わせた値を返します。この処理は**reduceメソッド**を使用して記述することもでき、その場合には、第1引数に指定した初期値をベースに、第2引数に指定したラムダ式によってストリーム内の要素を順に足し合わせるコードを記述します（選択肢**C**）。IntStreamインタフェースのreduceメソッドに指定可能なラムダ式はIntBinaryOperator型であり、このインタフェースで宣言されているSAMのapplyAsIntメソッドはint型の数を2つ取ります。したがって、引数を1つしか取らないラムダ式の記述は誤りです（選択肢A、D）。また、初期値を指定せずにラムダ式だけを引数に取るreduceメソッドの戻り値はOptionalInt型となるため、変数宣言の型と戻り値の型が適合しません（選択肢B）。

43. B、D

➡ Q24

Streamインタフェースの**collectメソッド**は、ストリーム要素に対する可変リダクション操作を適用するためのメソッドです。collectメソッドには可変リダクションを実装するための3つの引数（Supplier、BiConsumer、BiConsumer）を個別に指定することも可能ですが、一般的によく使用される可変リダクション操作の場合には、ユーティリティ・クラスであるCollectorsクラスのstaticメソッドによって生成されるCollectorオブジェクトを渡す便利な方法を使用することができます。このCollectorsクラスではさまざまな可変リダクション操作用のメソッドが用意されていますが、要素をグループ化するgroupingByメソッド（選択肢**B**）や、文字列要素を結合するjoiningメソッド（選択肢**D**）、ストリーム要素からListオブジェクトを作成するtoListメソッドなどがよく使用されます。選択肢Aや選択肢Cのようなメソッドは存在しないため誤りです。

44. A、B

➡ Q24

ForkJoinPoolクラスはExecutorServiceインタフェースの実装の1つですが、他の実装クラスとは異なり、Work-stealingアルゴリズムを使用してタスクを実行します。デフォルトの並列性レベルは、システム上で使用可能なプロセッサと同じです（選択肢**A**、**B**）。ちなみに、使用可能なプロセッサの数はRuntime.getRuntimeメソッドを使用して、availableProcessorメソッドで取得することができます。また、ForkJoinPoolオブジェクトの並列性レベルはgetPallalelismメソッドで取得できます。

ForkJoinPoolクラスでは他のExecutorServiceの実装と同様にexecute、invoke、submitという3つのタスク実行用メソッドが提供されています（選択肢C）。

ForkJoinPoolクラスが採用しているWork-stealingアルゴリズムは、大きなタスクを小さなタスクに分割して実行し、最終的にそれらの結果をすべて統合する戦略（Divide and Conquer Algorithm：分割統治法）であるため、タスク間の協調や調整が必要となる処理には適していません（選択肢D）。

45. C

➡ Q25

LocalDateオブジェクトが表す「月」だけを変更した新しいLocalDateオブジェクトを生成するには、**withメソッド**もしくは**withMonthメソッド**のいずれかを使用することができます。withメソッドの引数にはjava.time.temporal.ChronoField列挙定数と値を指定し、withMonthメソッドの引数には単純に値のみを指定します。したがって、withメソッドを使用して「月」を表す列挙定数「ChronoField.MONTH_OF_YEAR」と値「9」を指定している選択肢**C**が正解です。

選択肢Aのようなatメソッドや選択肢Dのようなofメソッドは提供されていません。また、選択肢BのようなChronoUnit列挙定数を引数に取るofメソッドは提供されていません。

46. C

➡ Q25

リソース・バンドルとなるファイルの検索では、基底名とロケール（**言語コード_国コード**）が一致するファイルを最初に検索し、存在しなければ言語コードのみが一致するファイルを検索します。それらのファイルが存在しない場合に基底名のみが一致するファイルを検索します。したがって、選択肢**C**が正解です。また、クラスファイルとプロパティファイルの両方が存在する場合には、常にクラスファイルが優先されることも覚えておきましょう。

47. D

➡ Q26

設問のコードでは、HashSetオブジェクトに「A」と「B」という文字列要素を追加した後、そのHashSetオブジェクトからIteratorオブジェクトを取得しています。その後、HashSetオブジェクトに要素「C」を追加してからIteratorオブジェクトを反復処理していますが、通常のコレクションからIteratorを取得した後に、元のコレクションが変更されていた場合には例外**java.util.ConcurrentModificationException**がスローされます。したがって、選択肢**D**が正解です。

通常のコレクションではなく並行処理ユーティリティに含まれるコレクションを使用した場合には、IteratorがConcurrentModificationExceptionをスローすることはありません。設問のコードで、HashSetの代わりにCopyOnWriteArraySetを使用した場合には選択肢Aが正解となり、ConcurrentSkipListSetを使用した場合には選択肢Bが正解となります。

48. E

➡ Q26

設問のコードでは、最初のpeekメソッドによる「123」と、mapメソッド適用後の2番目のpeekメソッドによる「149」が表示されますが、ストリームの要素ごとに2回のpeekメソッドの呼び出しが適用されるため、結果として「112439」と表示されます。したがって、選択肢**E**が正解です。

49. C

➡ Q27

ラムダ式の外部で宣言されている変数をラムダ式の中で参照する場合、その変数は「Effectively Final（実質的にfinal）」と呼ばれ、暗黙的にfinal宣言されているのと同じ制約が課されます。設問のコードでは、ラムダ式の外部で宣言されている変数nameをラムダ式で参照しているため、nameは実質的に

finalとみなされ、nameに値を再代入することはできません。したがって、コードはコンパイルエラーとなります（選択肢**C**）。

50. B

➡ Q27

入力された引数をそのまま返すFunctionインタフェースのメソッドは**identity**です。したがって、選択肢**B**が正解です。その他の選択肢のメソッドは、いずれもFunctionインタフェースでは宣言されていません。

51. C

➡ Q28

設問のコードにおいて、配列に格納されているBookオブジェクトのpriceの合計値を算出するために、**Streamインタフェース**の**collectメソッド**を使用することができます。collectメソッドの引数に渡すCollectorオブジェクトの生成には、**Collectorsクラス**の**summingDoubleメソッド**を使用することができます。summingDoubleメソッドは、合計する個々の値を取得するためのToDoubleFunctionオブジェクトを引数に取るため、BookクラスのgetPriceをメソッド参照で渡すことができます。したがって、選択肢**C**が正解です。

配列からストリームを生成するには、Arraysクラスのstreamメソッドを使用して、引数に配列オブジェクトの参照を渡す必要があります。配列オブジェクトそのものからstreamメソッドを呼び出すことはできないため、選択肢AとBは誤りです。Collectorsクラスでsummingというメソッドは提供されていないため選択肢Dは誤りです。

52. A

➡ Q29

BinaryOperator<T>インタフェースは、2つの引数と戻り値のすべてが同じT型となるBiFunction<T, T, T>インタフェースのサブインタフェースであり、BiFunction型変数にBinaryOperator型オブジェクトを代入することが可能です。したがって、設問のコードは正常にコンパイルと実行ができ、ラムダ式「`(i1, i2) -> i1 * i2`」の結果である「6」が表示されます（選択肢**A**）。

53. D

➡ Q29

Collectorsクラスの**partitioningByメソッド**は、引数に渡されたPredicateオブジェクトの判定に従ってtrueとfalseのグループに要素を分割したMap<Boolean, List<T>>オブジェクトを返します。設問のコードでは、ストリームの要素の中から文字列「ar」を含むものをtrueのグループに、そうでないものをfalseのグループに分割しています。結果はtrueまたはfalseをキーに、グループ化された要素をListオブジェクトとしたMapオブジェクトが返されるため、選択肢**D**のように表示されます。

54. D

➡ Q30

AtomicIntegerから値を取得するたびに値を1ずつインクリメントする場合はgetAndIncrementメソッドやincrementAndGetメソッドなどを使用しますが、増分値をカスタマイズしたい場合にはgetAndAccumulateメソッドやaccumulateAndGetメソッドを使用することができます。これらのメソッドは第1引数にint型の「更新値X」を、第2引数にカスタマイズ処理を実装するIntBinaryOperatorオブジェクトを取るため、選択肢**D**が正解となります。

【メソッド定義】

```
int getAndAccumulate(int x, IntBinaryOperator accumulatorFunction)
int accumulateAndGet(int x, IntBinaryOperator accumulatorFunction)
```

第2引数に指定するIntBinaryOperatorのapplyAsIntメソッドは、第1引数に現在の値を、第2引数に「更新値X」を受け取ります。したがって、設問のコードではforループで3回繰り返し処理が行われる際にラムダ式の引数xとyはそれぞれ以下に示す値となり、「024」が表示されます。

- 1回目…… x → 0, y → 2
- 2回目…… x → 2, y → 2
- 3回目…… x → 4, y → 2

なお、このように増分値を単純に「更新値X」分だけ増やしていくのであれば、簡易的に使用できるgetAndAddメソッドやaddAndGetメソッドを使用することができます。したがって、設問の正解のコードは以下のように記述することもできます。

例 getAndAddメソッドによる値の増分

```
11.  System.out.println(ai.getAndAdd(2));
```

AtomicIntegerクラスでは、選択肢AとBのように引数を取るIncrementAndGetメソッドは提供されていません。また、getAndAccumulateメソッドは引数を2つ取るため、選択肢Cも誤りです。

55. A

➡ Q30

IntStreamインタフェースではPredicateオブジェクトを引数に取るanyMatchメソッドは宣言されていないため、「// line n1」の行がコンパイルエラーとなります。したがって、選択肢**A**が正解です。「`Predicate<Integer>`」ではなく、「`IntPredicate`」とすれば選択肢Bが正解となります。anyMatchメソッドは、引数に指定されたPredicateもしくはIntPredicateオブジェクトの評価のいずれ

かに一致する要素があればtrueを返します。

オブジェクトを扱う一般的なStreamや関数型インタフェースとプリミティブ・バージョンは、型の互換性がないことに注意しましょう。

56. D

➡ Q31

設問のコードではstaticクラスIncとDecのrunメソッドを、それぞれ個別のスレッドで実行しています。incrementAndGetメソッドはAtomicIntegerオブジェクトが保持する値を1増やして返すメソッドであり、反対にincrementAndGetメソッドは値を1減らして返すメソッドです。このコードでは、forループの中でIncとDecそれぞれのインスタンスを3回生成しているため、全部で6つの個別のスレッドが生成されます。これらのスレッドの実行順序は保証されていないため、実行されるごとに-3から3までの6つの異なる値が表示されることになります（選択肢**D**）。ただし、通常の実行ではIncもしくはDecを実行するスレッドが3回連続して実行される可能性はタイミング的に極めて低いため、3もしくは-3が表示されることはないでしょう。

57. D

➡ Q32

Supplierインタフェースでは、引数を受け取らず、戻り値を返す抽象メソッド**get**が宣言されています。Supplier型の変数に代入できる適切なラムダ式あるいはメソッド参照（もしくはコンストラクタ参照）は、このgetメソッドの定義に一致している必要があります。各選択肢に関する説明は以下のとおりです。

A. メソッド参照の構文仕様に則っていないため誤りです。メソッド呼び出しの「()」は不要です。
B. ラムダ式の構文仕様に則っていないため誤りです。メソッド呼び出しの「()」が必要です。
C. コンストラクタ参照の構文仕様としては正しいですが、LocalDateクラスでpublicなコンストラクタは宣言されていないため誤りです。
D. 引数を取らずに新しいLocalDateオブジェクトを返すコンストラクタ参照の記述です。

したがって、選択肢**D**が正解です。

58. A、B

➡ Q32

列挙型で宣言されている列挙定数は、列挙型インスタンスへの参照となります。したがって、選択肢**A**のようにSystem.out.printlnメソッドの引数に渡した場合には、内部的にtoStringメソッドが呼び出されて結果が表示されます。すべての列挙型が暗黙的に継承するEnumクラスでのtoStringメソッドの実装は

列挙定数と同じ文字列となります。また、列挙定数が宣言されている順番（ゼロ・スタート）は選択肢**B**のように**ordinalメソッド**で取得することができます。

選択肢CのindexAtメソッドや選択肢DのvalueメソッドはEnumクラスでは宣言されていないため誤りです。

59. C

➡ Q32

設問のコードでは、**java.util.Arrays**クラスの**asListメソッド**によって返されるListオブジェクトを使用して新たなArrayListオブジェクトを生成し（// line n1）、さらにそのArrayListオブジェクトを使用して新たなArrayListオブジェクトを生成しています（// line n2）。このArrayListオブジェクトは要素が空ですが、この2行のコードは問題なくコンパイルと実行が可能です。

次の行（// line n3）では、ArrayListオブジェクトからgetメソッドを使用して最初の要素を取得しようとしていますが、このArrayListはワイルドカード「?」でパラメータ化されているため、getメソッドの戻り値の型はObject型として扱われることになります（「?」は「`? extends Object`」と等価です）。したがって、この行ではInteger型で変数を宣言しているため、コンパイルエラーとなります（選択肢**C**）。

getメソッドの戻り値をInteger型に明示的にダウンキャストするか、もしくは変数をObject型で宣言していればコンパイルは成功します。その場合には、空のArrayListオブジェクトからgetメソッドを使用して要素を取り出そうとしているため、実行時にIndexOutOfBoundsExceptionがスローされます（選択肢E）。もし、ArrayListオブジェクトが空でなければ、最初の要素が取り出されて表示されます。

60. C

➡ Q33

BufferedReaderクラスの**markメソッド**は、読み込み位置にマークを付けるためのメソッドです。マークを付けた後に読み込むことのできる文字数の上限をint値で指定する必要があり、設問のコードのように引数を取らないmarkメソッドは提供されていないため、13行目でコンパイルエラーとなります（選択肢**C**）。適切なint値が指定されている場合は「XYZ」の行の先頭にマークが設定され、resetメソッドによって読み込み位置がそこに戻されるため、選択肢Aが正解となります。

61. C

➡ Q34

関数型インタフェース「Foo」の抽象メソッド「doIt」は、int型引数を取り、T型オブジェクトを返す定義となっています。したがって、この定義に一致する**コンストラクタ参照**を記述している選択肢**C**が正解となります。

選択肢Cのメソッド参照をラムダ式で記述した場合は「`i -> new String[i]`」となり、次の行の「`doIt(3)`」の呼び出しによって、3つの要素を持つ配列オブジェクトが生成されます。このように、配列オブジェクトを生成するコンストラクタ参照は「**データ型[]::new**」という構文で記述します。選択肢BとDは構文仕様に則っていないため誤りです。選択肢Aはコンストラクタ参照の構文としては正しいですが、「`String::new`」は配列オブジェクトの生成ではなく、String型オブジェクトの生成式となるため誤りです。

62. A

➡ Q35

設問のコードで使用している**BlockingQueue**はQueueインタフェースのサブインタフェースであり、要素の格納や取得においてブロッキング機能を備えるキューを定義します。具体的には、要素を格納しようとした際にキューが満杯の場合には、キューに空きができるまで待機し、要素を取得しようとした際にキューが空の場合には、キューに要素が格納されるまで待機します。BlockingQueueインタフェースの実装としては、ArrayBlockingQueueクラスやLinkedBlockingQueueクラス、PriorityBlockingQueueクラス、DelayQueueクラスなどがあります。

設問のコードでは、ProducerオブジェクトがArrayBlockingQueueに対して要素を追加するのと並行的に、ConsumerオブジェクトがそのArrayBlockingQueueから要素を取り出しています。両者はランダム秒待機した後にこの操作を行っていますが、Producerのほうが待機時間が短いため、キューはいずれ満杯になります。キューが満杯になった場合、Producerのputメソッドはブロックされて待機し、Consumerが要素を取り出すとブロックが解除されて要素が追加されます。仮にConsumerのほうが早く要素を取り出してキューが空になった場合には、takeメソッドはブロックされて、Producerがキューに要素を格納するまで待機することになります。したがって、設問のプログラムは明示的に終了するまで正常に実行され続けます（選択肢**A**）。

63. D

➡ Q37

Connectionインタフェースでは、引数を1つだけ取る**createStatementメソッド**は提供されていないため、設問のコードはコンパイルエラーとなります。createStatementは、引数を取らないか、もしくは2つの引数を取るメソッドのみが提供されています。スクロール可能なResultSetを使用したい場合には、

第1引数に「**ResultSet.TYPE_SCROLL_INSENSITIVE**」もしくは「**ResultSet.TYPE_SCROLL_SENSITIVE**」のいずれかを指定し、第2引数には「**ResultSet.CONCUR_READ_ONLY**」もしくは「**ResultSet.CONCUR_UPDATABLE**」のいずれかを指定します。

スクロール可能なResultSetを使用するための引数が正しく指定されている場合には、afterLastメソッドによってカーソルは最終行の次に移動し、previousメソッドによってカーソルが1つ前に戻るため、選択肢Cが正解となります。

64. B

➡ Q38

InnerのpublicフィールドmessageはOuterクラス内のstaticメンバー・クラスです。messageはstaticフィールドではないため、このフィールドにアクセスするためにはInnerクラスのインスタンス生成が必要です。ただし、Innerクラス自体はOuterクラスのstaticメンバーであるため、Outerクラスのインスタンス生成は必要ありません。したがって、Outer.Innerクラスのインスタンスを生成している選択肢**B**が正解です。

65. A

➡ Q38

abstractキーワードとstaticキーワードを同時に使用することはできないため、「// line n1」の行がコンパイルエラーとなります。したがって、選択肢**A**が正解です。その他の行は、インタフェース内の宣言として正しいコードです。

66. C

➡ Q39

設問のコードのmainメソッドではExampleクラスのdoItメソッドを呼び出していますが、このdoItメソッドはString型の引数を1つ渡し、戻り値をExample<String, String>型の変数に代入しています。したがって、このdoItメソッドの正しい実装は「T型の値を受け取って、2つのT型でパラメータ化した自身のインスタンスを返す」ものとなります（選択肢**C**）。選択肢AとBは、戻り値の定義が1つのT型でパラメータ化したExampleインスタンスを返すものとなっているため誤りです。また、選択肢Dは型変数Tの宣言が重複しているため誤りです。

67. C

➡ Q40

設問のコードのComparator型変数sortByFirstNameは、firstNameの昇順でソートするラムダ式が代入されています。もう1つのComparator型変数sortByLastNameは、lastNameの昇順でソートするラムダ式が代入されています。設問の表示結果はfirstNameの降順、lastNameの降順でソートされているため、sortedメソッドにsortByFirstNameを最初に適用し、次にthenComparingメソッドを使用して

sortByLastNameを適用した結果（この時点でfirstNameの昇順、lastNameの昇順となっています）をreverseメソッドで逆順にしています。したがって、選択肢**C**が正解です。その他の選択肢に関する説明は以下のとおりです。

A. firstNameの昇順、lastNameの昇順でソートされます。
B. firstNameの降順、lastNameの昇順でソートされます。
D. firstNameの昇順、lastNameの降順でソートされます。

68. A

➡ Q42

設問のコードでは、最初にXクラスのdoItメソッドを呼び出しており、「X.doIt()」が表示されます。このdoItメソッドでは例外Exceptionをスローしているため、その例外はmainメソッドのcatchブロックで捕捉されて「exception」が表示されますが、その前にtry-with-resources文を使用して、オブジェクトを生成しているXクラスとYクラスのcloseメソッドが呼び出されます。closeメソッドが呼び出される順番はtry-with-resources文における宣言の逆順となるため、「closing Y」が表示され、次に「closing X」が表示されます。したがって、選択肢**A**が正解です。

69. D

➡ Q43

Functionインタフェースは T型の引数を受け取ってR型の戻り値を返す抽象メソッド「apply」を宣言しています。したがって、型の宣言は「Function<T, R>」であり、抽象メソッドの定義は「R apply(T t)」となります（選択肢**D**）。

70. C

➡ Q44

設問のコードでは、**java.nio.file.Filesクラス**の**linesメソッド**を使用してsample.txtファイル中に記述されているテキストを1行ずつ読み取り、その1行に含まれる単語をスペースで分割し、flatMapメソッドを使用して「Red, Blue, Red, Green, Green, Red」という要素のストリームを生成しています。

次に、**collectメソッド**を使用してリダクション処理を実行してMapオブジェクトを生成しています。この処理では**Collectorsクラス**の**groupingBy メソッド**を使用し、キー出現回数を値として単語のグループ化を行っており、Mapオブジェクトは「{Red=3, Blue=1, Green=2}」となります。その後、**entrySetメソッド**を使用してこのMapオブジェクトから**Map.Entryオブジェクト**のストリームを取得し、**sortedメソッド**で並べ替えを行っています。sortedメソッドの引数にはMap.Entryの**comparingByValueメソッド**を使用しており、値での昇順ソートが行われます。この時点で要素は「{Blue=1, Green=2, Red=3}」となり、最後にforEachメソッドでキーだけを表示しているため、「BlueGreenRed」が表示されます。したがって、選択肢**C**が正解です。

71. C

➡ Q44

Streamインタフェースのmaxメソッドは、ストリーム内の要素の最大値を求めるメソッドです。この両メソッドは、引数にComparatorオブジェクトを受け取り、戻り値としてOptional型オブジェクトを返します。したがって、選択肢**C**が正解です。最小値を求める場合はminメソッドを使用します。

Streamインタフェースはあらゆるオブジェクトのストリームを扱うため、最大・最小の判断基準をComparatorオブジェクトで指定する必要があります。引数を取らないmaxメソッドおよびminメソッドはStreamインタフェースでは宣言されていません（選択肢A、B）。IntStreamやLongStream、DoubleStreamなどの数値を扱うストリーム・インタフェースであれば、引数を取らないmaxメソッドおよびminメソッドが提供されています。

72. B

➡ Q45

Optionalクラスの**ifPresentメソッド**は、値が存在する場合は引数で指定されたConsumerオブジェクトの実装を呼び出します。値が存在しない場合には何も実行しません。いずれの場合も戻り値はvoidであるため、このメソッドに続けて他のメソッドを呼び出すことはできません。したがって、「// line n2」の行で呼び出している「`orElse("Empty")`」が原因でコンパイルエラーになります（選択肢**B**）。

73. A

➡ Q45

flatMapメソッドは、引数に指定されたFunctionオブジェクトの実装に従って、入れ子構造のストリームを展開した平坦なストリームを生成します。設問のコードでは、Stringクラスのsplitメソッドは引数に指定された文字列を区切り記号として、その文字列を分割するため、flatMapメソッドが適用された時点でストリーム内の要素は「abcxyz」となります。このときに、「c」と「x」の間にはストリーム要素として「空文字」が含まれている点に注意してください。したがって、表示される文字列には「空文字:」も追加されるため、選択肢**A**が正解です。

74. A

➡ Q46

Streamインタフェースの**flatMapメソッド**は、ストリーム内の入れ子構造になっている要素を展開し、新たなストリームを生成します。設問のコードでは、3重構造のListオブジェクト（`List<List<List<String>>>`）から生成されたストリーム「`[[A, B] [C, D, E] [F]] [[e, a, d] [c, f]]`」に対してflatMapメソッド（およびstreamメソッド）を適用し、一番外側のListが外された「`[A, B] [C, D, E] [F] [e, a, d] [c, f]`」という新たなストリームを生成しています。

次に、filterメソッドによってListの要素数が2よりも大きいものだけを抽出しているため、この時点でストリーム内の要素は「[C, D, E] [e, a, d]」となります。そして、mapメソッドに渡しているメソッド参照「String::toLowerCase」は、ラムダ式「s -> s.toLowerCase()」と等価であり、渡される要素がString型であることを前提としています。しかし、渡される要素はString型ではなくList<String>型であるため、toLowerCaseメソッドの呼び出しはコンパイルエラーとなります。したがって、選択肢**A**が正解です。

filterメソッドの呼び出しの直後に、「flatMap(Collection::stream)」メソッドの記述がもう1つあればストリームの要素は「C D E e a d」となるため、toLowerCaseメソッドですべて小文字に変換された後、distinctメソッドで重複要素が排除されます（選択肢C）。distinctメソッドの記述がなければ選択肢Dが正解となります。

75. C

➡ Q47

Collectionインタフェースでdefaultメソッドとして宣言されている**removeIfメソッド**は、引数に指定したPredicateオブジェクトの条件判定に一致する要素をリストから削除します。したがって、選択肢**C**が正解です。選択肢Aのメソッドや選択肢Dのremoveメソッドは、Predicateオブジェクトを引数に取らないため誤りです。また、選択肢Bのようなメソッドは存在しません。

76. B、D

➡ Q47

Java SE 8では従来の日付・時刻関連のAPIを刷新して、新しい「Date and Time API（JSR 310）」を導入しました。このAPIは、コンピュータにおける日付と時刻の形式に関する国際標準規格である「ISO 8601」をベースに設計されています。日時を表現するクラスはImmutableであり、スレッド・セーフとなっています。したがって、選択肢**B**と**D**が正解です。

77. C、D

➡ Q48

設問のコードで使用しているshutdownメソッドは、Executorインタフェースではなく**ExecutorService**インタフェースで宣言されているため、「// line n3」を選択肢**C**のように修正する必要があります。また、「// line n4」で使用しているcallメソッドはExecutorServiceインタフェースでは宣言されていません。Callableタスクを実行してFutureオブジェクトを取得するには選択肢**D**のようにsubmitメソッドを使用します。

「// line n1」および「// line n2」のコードはいずれも修正する必要はないため、選択肢AとBは誤りです。

78. A、C、D

➡ Q49

選択肢**A**の「レース・コンディション（Race Condition＝競合条件）」とはマルチスレッドにおける最も一般的な問題であり、システムの振る舞いが複数スレッドの実行順序に依存してしまう状況を指します。特に共有データに対するアクセスの競合によってレース・コンディションが発生し、システムが予測不能な挙動を起こさないように十分に注意する必要があります。

選択肢**C**の「デッドロック（Dead Lock）」とは、複数のスレッドが互いに他方のロックの解放を待ち続けることにより、スレッドの実行が永久に停止してしまう状態です。また、複数スレッドがロックの取得と解放を行ってはいるものの、うまくタイミングが合わずに処理が進まない状態を「ライブロック」といいます。デッドロックを回避するにはロックを適切に設計することが重要ですが、デッドロックを検出してロックを強制的に解放させるような仕組みや、一定時間ロック待ちになった場合にはロックを解放するタイムアウトの設定などを検討することもできます。

選択肢**D**の「スターベーション（Starvation）」とは、あるスレッドが、必要とする共有リソースに対して半永久的にアクセスできないような状況です。スターベーションに陥ったスレッドは、必要とする共有リソースを獲得できずに処理の進行が止まってしまいます。スターベーションが発生する主な原因としては、スレッドに対してリソースを平等に割り当てるスケジューリングがうまく機能していないことが挙げられます。スターベーションを防ぐには、優先度の高い特定のスレッドばかりがリソースを占有してしまわないようにロックの粒度を小さくしたり、ロックを奪い合う頻度を少なくしたりするなどの工夫が必要となります。

その他の選択肢はマルチスレッド・アプリケーションの一般的な問題ではありません。選択肢Bのリクエスト・フォージェリや選択肢Eのインジェクションは、Webアプリケーションにおけるセキュリティの問題です。

79. B

➡ Q49

Collectionsクラスでstaticメソッドとして提供されている**reverseOrderメソッド**は、逆順のソートを行うComparatorを返すためのメソッドです。reverseOrderメソッドの引数を指定しなかった場合は、対象オブジェクトが実装しているComparableインタフェースに基づく逆順ソートが行われます。Mapオブジェクトの entrySetメソッドによって返されるMap.Entryオブジェクトは、Comparableインタフェースを実装していないため、引数を指定しないreverseOrderメソッドを使用した場合には実行時に例外がスローされます（選択肢D）。設問のコードのように引数にComparatorオブジェクトを指定した場合には、そのComparatorに基づく逆順ソートが行われるため、キーによる逆順ソートとな

る選択肢**B**が正解となります。

80. D

➡ Q50

try-with-resources文で宣言可能なクラスはAutoCloseableインタフェースもしくはそのサブインタフェースであるCloseableインタフェースをimplementsし、抽象メソッド「public void close()」の実装を提供する必要があります。したがって、選択肢**D**が正解です。

選択肢Aと選択肢Cはそれぞれ可視性が「protected」と「パッケージスコープ」となっていますが、抽象メソッドの実装での可視性を狭くすることはできないため誤りです。選択肢Bはcloseメソッドを実装できていないため誤りです。

81. A、B

➡ Q51

関数型インタフェースとは、抽象メソッドを1つだけ宣言しているインタフェースです（選択肢**A**）。選択肢Dのようにメソッドが1つも宣言されていないインタフェースは、関数型インタフェースの要件を満たしていません。

選択肢**B**では2つの抽象メソッドが宣言されていますが、すべてのインタフェースの実装クラスは暗黙的にObjectクラスのサブクラスとなるため、Objectクラスで宣言されているメソッドと同じシグニチャの抽象メソッド「`boolean equals(Object obj)`」は関数型インタフェースの抽象メソッドとみなされません。したがって、選択肢**B**は1つの抽象メソッド「`void doIt()`」だけが宣言されていることになります。

同じ理由から、選択肢Cは抽象メソッドが1つも宣言されていないインタフェースとみなされるため、関数型インタフェースとしては誤りです。

82. B

➡ Q51

collectメソッドを使用してストリーム内の複数の文字列を1つの文字列に連結するリダクション処理では、**Collectorsクラス**の**joiningメソッド**を使用することができます。引数を取らないjoiningメソッドは単純にストリーム内の文字列をつなぎ合わせるだけですが、デリミタ（区切り文字）、プレフィクス、サフィクスの3つの文字列を引数に取るjoiningメソッドを使用することで、設問の要件を満たすことができます。したがって、選択肢**B**が正解です。その他の選択肢は、いずれもCollectorsクラスには存在しないメソッドです。

83. C

➡ Q52

設問のコードでは、文字列「Write_Once_Run_Anywhere」をsplitメソッドによって分割した配列に対して、parallelメソッドを使用した並列ストリームを適用しており、forEachメソッドではStringConcatenatorクラスのconcatenateメソッドはマルチスレッドで実行されることになります（内部的にはForkJoinPoolが使用されます）。したがって、StringConcatenatorクラスのpublicフィールドであるresultに文字列を結合する順序は保証されず、実行するたびに異なる結果が表示されます（選択肢**C**）。このように並列ストリームを使用する場合には、共有データへのアクセスと、結果に及ぼす影響を理解しておく必要があります。なお、並列ストリームではなく、順次ストリームであれば選択肢Bが正解となります。

84. C

➡ Q53

データベースに接続するには**DriverManagerクラス**のstaticな**getConnectionメソッド**を使用します。したがって、選択肢**C**が正解です。

JDBCのライブラリでは、選択肢AのDatabaseManagerや選択肢DのDatabaseDriverというクラスは提供されていません。また、選択肢BのConnectionインタフェースはgetConnectionというメソッドを提供していません。

85. A

➡ Q54

Filesクラスの**listメソッド**は、Pathオブジェクトとして指定されたディレクトリ内のコンテンツ一覧をStream<Path>オブジェクトとして返します。listメソッドは、指定されたディレクトリ自体は要素に含めません。また、サブディレクトリを再帰的に処理しないため、設問のコードではdirディレクトリ内のa.txtファイル、b.txtファイル、subディレクトリの3つのコンテンツのみを要素として持つストリームが生成されます。Streamオブジェクトのcountメソッドはストリームに含まれる要素数を返すため、「3」が表示されます。したがって、選択肢**A**が正解です。

指定されたPathオブジェクトがディレクトリを表していない場合には、例外**NotDirectoryException**がスローされることも覚えておきましょう。

また、listメソッドではなく**walkメソッド**を使用した場合には、指定したディレクトリを含めたサブディレクリすべてをたどる再帰処理が行われるため、選択肢Dが正解となります。