48. Which of the following are not required parameters for the NIO.2 `Files.find()` method? (Choose two.)

A. `BiPredicate`

B. `FileVisitOption...`

C. `int`

D. `long`

E. `Path`

49. Which statements are correct? (Choose two.)

A. A `Comparable` implementation is often implemented by a lambda.

B. A `Comparable` object has a `compare()` method.

C. The `compare()` and `compareTo()` methods have the same contract for the return value.

D. There can be multiple `Comparator` implementations for the same class.

E. Two objects that return `true` for `equals()` will always return `0` when passed to `compareTo()`.

50. What is the output of the following code snippet, assuming none of the files referenced exist within the file system?

```
Path t1 = Paths.get("/sky/.././stars.exe");
Path t2 = Paths.get("/stars.exe");
Path t3 = t1.resolve(t2);

boolean b1 = t1.equals(t2);
boolean b2 = t1.normalize().equals(t2);
boolean b3 = Files.isSameFile(t1.normalize(),t2);
boolean b4 = Files.isSameFile(t2,t3);

System.out.print(b1+","+b2+","+b3+","+b4);
```

A. `false,false,true,true`

B. `false,true,true,false`

C. `false,true,true,true`

D. `true,false,true,false`

E. The code does not compile.

F. The code compiles but throws an exception at runtime.

51. Let's say we have a `Reader` instance that will produce the characters with the numeric values `{1,2,3,4,5,6,7}`. Which of the following are possible outcomes of executing the `checkLottoNumbers()` method with this `Reader` instance? (Choose two.)

```
23: public String checkLottoNumbers(Reader r) throws IOException {
24:    r.read();r.skip(1);
25:    r.mark(5);
```

```
26:    r.skip(1);
27:    r.reset();
28:    return r.read()+"-"+r.read(new char[5]);
29: }
```

A. An `IOException` on line 25

B. An `IOException` on line 27

C. `'c'-4` is returned.

D. `'d'-3` is returned.

E. `3-4` is returned.

F. `4-3` is returned.

52. Fill in the blanks: The name of the `abstract` method in the `Function` interface is _____, while the name of the `abstract` method in the `Consumer` interface is _____.

A. `accept()`, `apply()`

B. `accept()`, `get()`

C. `apply()`, `accept()`

D. `apply()`, `apply()`

E. `apply()`, `test()`

53. Assuming the following program is executed with assertions enabled, which is the first line to throw an exception at runtime?

```
1:  package school;
2:  public class Teacher {
3:     public int checkClasswork(int choices) {
4:        assert choices++==10 : 1;
5:        assert true!=false : new StringBuilder("Answer2");
6:        assert(null==null) : new Object();
7:        assert ++choices==11 : "Answer4";
8:        assert 2==3 : "";
9:        return choices;
10:    }
11:    public final static void main(String... students) {
12:       try {
13:          new Teacher().checkClasswork(10);
14:       } catch (Error e) {
15:          System.out.print("Bad idea");
16:          throw e;
17:       }
18:    }
19: }
```

A. Line 4
B. Line 5
C. Line 6
D. Line 7
E. Line 8
F. None of the above since the class does not compile

54. Which of the following are valid functional interfaces in the `java.util.function` package? (Choose three.)

A. `BooleanSupplier`
B. `CharSupplier`
C. `DoubleUnaryOperator`
D. `ObjectIntConsumer`
E. `ToLongBiFunction`
F. `TriPredicate`

55. Which statements about the following class are correct? (Choose two.)

```
package knowledge;
class InformationException extends Exception {}
public class LackOfInformationException extends InformationException {
   public LackOfInformationException() {  // t1
      super("");
   }
   public LackOfInformationException(String s) {  // t2
      this(new Exception(s));
   }
   public LackOfInformationException(Exception c) {  // t3
      super();
   }
   @Override public String getMessage() {
      return "lackOf";
   }
}
```

A. `LackOfInformationException` compiles without issue.
B. The constructor declared at line `t1` does not compile.
C. The constructor declared at line `t2` does not compile.
D. The constructor declared at line `t3` does not compile.
E. The `getMessage()` method does not compile because of the `@Override` annotation.
F. `LackOfInformationException` is a checked exception.

56. How many changes do you need to make in order for this code to compile?

```
public class Ready {
   private static double getNumber() {
      return .007;
   }
   public static void math() {
      Supplier<double> s = Ready:getNumber;
      double d = s.get();
      System.out.println(d);
   }
}
```

A. None
B. One
C. Two
D. Three
E. Four

57. Which statement about the following class is correct?

```
package robot;
import java.util.concurrent.*;
public class PassButter extends RecursiveTask<String> {  // j1
   final int remainder;
   public PassButter(int remainder) {  // j2
      this.remainder = remainder;
   }
   @Override
   protected String compute() {
      if (remainder <= 1)
         return "1";
      else {
         PassButter otherTask = new PassButter(remainder - 1);
         String otherValue = otherTask.fork().join();  // j3
         return otherValue
            + new PassButter(remainder - 2).compute();
      }
   }
   public static void main(String[] purpose) {
      ForkJoinPool pool = new ForkJoinPool();
      ForkJoinTask<?> task = new PassButter(10);
```

```
            System.out.print(pool.invoke(task));
            pool.shutdown();
        }
    }
```

A. The code does not compile due to line `j1`.

B. The code does not compile due to line `j2`.

C. The code does not compile due to line `j3`.

D. The code compiles and properly implements the fork/join framework in a multi-threaded manner.

E. The code compiles but does not implement the fork/join framework in a proper multi-threaded manner.

F. The class compiles and prints an exception at runtime.

58. Which can fill in the blank so this code outputs `true`?

```
import java.util.function.*;
import java.util.stream.*;

public class HideAndSeek {
   public static void main(String[] args) {
      Stream<Boolean> hide = Stream.of(true, false, true);
      Predicate<Boolean> pred = b -> b;
      boolean found = hide.filter(pred)._________(pred);
      System.out.println(found);
   }
}
```

A. Only `anyMatch()`

B. Only `allMatch()`

C. Both `anyMatch()` and `allMatch()`

D. Only `noneMatch()`

E. The code does not compile with any of these options.

59. Given the following code, Java will try to find a matching resource bundle. Which order will Java search to find a match?

```
Locale.setDefault(new Locale("en"));
ResourceBundle.getBundle("AB", new Locale("fr"));
```

A. `AB.class`, `AB.properties`, `AB_en.properties`, `AB_fr.properties`

B. `AB.properties`, `AB.class`, `AB_en.properties`, `AB_fr.properties`

C. `AB_en.properties`, `AB_fr.properties`, `AB.class`, `AB.properties`

D. `AB_fr.properties`, `AB.class`, `AB.properties`, `AB_en.properties`

E. `AB_fr.properties`, `AB_en.properties`, `AB.class`, `AB.properties`

F. `AB_fr.properties`, `AB_en.properties`, `AB.properties`, `AB.class`

60. What is the result of the following?

```
Set<Integer> dice = new TreeSet<>();
dice.add(6);
dice.add(6);
dice.add(4);

dice.stream().filter(n -> n != 4).forEach(System.out::println).count();
```

A. It prints just one line.

B. It prints one line and then the number 3.

C. There is no output.

D. The code does not compile.

E. The code compiles but throws an exception at runtime.

61. Given the following two property files in the pod package, what does the following class output?

```
pod.container.properties
name=generic
number=2

pod.container_en.properties
name=Docker
type=container

package pod;
import java.util.*;
public class WhatKind {
   public static void main(String[] args) {
      Locale.setDefault(new Locale("ja"));
      ResourceBundle rb = ResourceBundle.getBundle("pod.container");
      String name = rb.getString("name");    // r1
      String type = rb.getString("type");    // r2
      System.out.println(name + " " + type);   }
}
```

A. `Docker container`

B. `generic container`

C. `generic null`

D. The code does not compile.

E. Line `r1` throws an exception.

F. Line `r2` throws an exception.

62. What is the result of the following?

```
import java.util.stream.*;

public class StreamOfStreams {

   public static void main(String[] args) {
      Integer result =
         Stream.of(getNums(9, 8), getNums(22, 33))  // c1
         .filter(x -> !x.isEmpty())                 // c2
         .flatMap(x -> x)                           // c3
         .max((a, b) -> a - b)                      // c4
         .get();
      System.out.println(result);
   }
   private static Stream<Integer> getNums(int num1, int num2) {
      return Stream.of(num1, num2);
   }
}
```

A. The code compiles and outputs 8.

B. The code compiles and outputs 33.

C. The code does not compile due to line c1.

D. The code does not compile due to line c2.

E. The code does not compile due to line c3.

F. The code does not compile due to line c4.

63. Which of the following shows a valid `Locale` format? (Choose two.)

A. `de`

B. `DE`

C. `de_DE`

D. `DE_de`

64. What is true of the following if the `music` database exists and contains a `songs` table with one row when run using a JDBC 4.0 driver? (Choose two.)

```
import java.sql.*;
public class Music {
   public static void main(String[] args) throws Exception {
      String url = "jdbc:derby:music";
      Connection conn = DriverManager.getConnection(url);
      Statement stmt = conn.createStatement();
      stmt.execute("update songs set name = 'The New Song'");
   }
}
```

A. The code does not compile.

B. The code does not update the database because it calls `execute()` rather than `executeUpdate()`.

C. The code does not update the database because the `Statement` is never closed.

D. The code runs without error.

E. The `execute()` method returns a `boolean`.

F. The `execute()` method returns an `int`.

65. How many of the following pairs of values can fill in the blanks to comply with the contract of the `hashCode()` and `equals()` methods?

```
class Sticker {
   @Override
   public int hashCode() {
      return _____________;
   }
   @Override
   public boolean equals(Sticker o) {
      return _____________;
   }
}
```

I. `5, false`

II. `5, true`

III. `new Random().nextInt(), false`

IV. `new Random().nextInt(), true`

A. None

B. One

C. Two

D. Three

E. Four

F. None of the above. The code does not compile with any of the options.

66. What is the output of the following application?

```
package winter;

abstract class TShirt {
   abstract int insulate();
   public TShirt() {
      System.out.print("Starting...");
   }
}
```

```
public class Wardrobe {
   abstract class Sweater extends TShirt {
      int insulate() {return 5;}
   }
   private static void dress() {
      class Jacket extends Sweater {  // v1
         int insulate() {return 10;}
      };
      final TShirt outfit = new Jacket() {  // v2
         int insulate() {return 20;}
      };
      System.out.println("Insulation:"+outfit.insulate());
   }

   public static void main(String... snow) {
      new Wardrobe().dress();
   }
}
```

A. `Starting...Insulation:20`

B. `Starting...Insulation:40`

C. The code does not compile because of line `v1`.

D. The code does not compile because of line `v2`.

E. The code does not compile for a different reason.

67. Which statements about the following application are true?

```
1:  package armory;
2:  import java.util.function.*;
3:  class Shield {}
4:  public class Sword {
5:     public class Armor {
6:       int count;
7:       public final Function<Shield,Sword,Armor> dress = (h,w) -> new Armor();
8:       public final IntSupplier<Integer> addDragon = () -> count++;
9:     }
10:    public static void main(String[] knight) {
11:      final Armor a = new Armor();
12:      a.dress.apply(new Shield(), new Sword());
13:      a.addDragon.getAsInt();
14:    }
15: }
```

I. The lambda expression for `dress` on line 7 compiles without issue.

II. The lambda expression for `addDragon` on line 8 compiles without issue.

III. Not counting the lambda expressions on lines 7 and 8, the code does not contain any compilation errors.

A. I only

B. I and II only

C. I, II, and III

D. II and III only

E. None of the above

68. Which two conditions best describe a thread that appears to be active but is perpetually stuck and never able to finish its task? (Choose two.)

A. Deadlock

B. Livelock

C. Loss of precision

D. Out of memory error

E. Race condition

F. Starvation

69. Which statements are true about the following date/times? (Choose two.)

```
2017-04-01T17:00+03:00[Africa/Nairobi]
2017-04-01T10:00-05:00[America/Panama]
```

A. The first date/time is earlier.

B. The second date/time is earlier.

C. Both represent the same date/time.

D. The two date/times are zero hours apart.

E. The two date/times are one hour apart.

F. The two date/times are two hours apart.

70. What is true about the following?

```
import java.util.*;
public class Yellow {
   public static void main(String[] args) {
      List list = Arrays.asList("Sunny");
      method(list);     // c1
   }
   private static void method(Collection<?> x) {    //c2
      x.forEach(a -> {});   // c3
  }
}
```

A. The code doesn't compile due to line `c1`.
B. The code doesn't compile due to line `c2`.
C. The code doesn't compile due to line `c3`.
D. The code compiles and runs without output.
E. The code compiles but throws an exception at runtime.

71. What is true about the following code? (Choose two.)

```
public static void main(String[] args) throws Exception {
   String url = "jdbc:derby:hats;create=true";
   Connection conn = null;
   Statement stmt = null;

   try {
     conn = DriverManager.getConnection(url);
     stmt = conn.createStatement();
     stmt.executeUpdate(
       "CREATE TABLE caps (name varchar(255), size varchar(1))");
   } finally {
       conn.close();
       stmt.close();
   }
}
```

A. If using a JDBC 3.0 driver, this code throws an exception.
B. If using a JDBC 4.0 driver, this code throws an exception.
C. The resources are closed in the wrong order.
D. The resources are closed in the right order.
E. The `Connection` is created incorrectly.
F. The `Statement` is created incorrectly.

72. How many lines of the following application contain a compilation error?

```
package puzzle;
final interface Finder {
   default long find() {return 20;}
}
abstract class Wanda {
   abstract long find();
}
final class Waldo extends Wanda implements Finder {
   long find() {return 40;}
```

```
   public static final void main(String[] pictures) {
      final Finder f = new Waldo();
      System.out.print(f.find());
   }
}
```

A. One
B. Two
C. Three
D. None. The code compiles and prints `20` at runtime.
E. None. The code compiles and prints `40` at runtime.

73. What is the output of the following?

```
1:  package reader;
2:  import java.util.stream.*;
3:
4:  public class Books {
5:     public static void main(String[] args) {
6:        IntStream pages = IntStream.of(200, 300);
7:        long total = pages.sum();
8:        long count = pages.count();
9:        System.out.println(total + "-" + count);
10:    }
11: }
```

A. `2-2`
B. `200-1`
C. `500-0`
D. `500-2`
E. The code does not compile.
F. The code compiles but throws an exception at runtime.

74. What is the output of executing the following code snippet?

```
30: ExecutorService e = Executors.newSingleThreadExecutor();
31: Runnable r1 = () -> Stream.of(1,2,3).parallel();
32: Callable r2 = () -> Stream.of(4,5,6).parallel();
33:
34: Future<Stream> f1 = e.submit(r1);
35: Future<Stream> f2 = e.submit(r2);
36:
```

```
37: Stream<Integer> s = Stream.of(f1.get(),f2.get())
38:       .flatMap(p -> p)
39:       .parallelStream();
40:
41: ConcurrentMap<Boolean,List<Integer>> r =
42:       s.collect(Collectors.groupingByConcurrent(i -> i%2==0));
43: System.out.println(r.get(false).size()+" "+r.get(true).size());
```

A. `3 3`

B. `2 4`

C. The code does not compile due to one error.

D. The code does not compile due to two errors.

E. The code does not compile due to three errors.

F. The code compiles but a `NullPointerException` is thrown at runtime.

75. Fill in the blanks: If your application is _____, it must first have been _____ with respect to supporting multiple languages.

A. extracted, internationalized

B. extracted, localized

C. internationalized, extracted

D. internationalized, localized

E. localized, extracted

F. localized, internationalized

76. Which statement about the following class is true? Assume the file system is available and able to be modified.

```
package forest;
import java.io.File;
public class CreateTree {
   public boolean createTree(String tree) {
      if(new File(tree).exists()) {
         return true;
      } else {
         return new File(tree).mkdir();
      }
   }
   public static void main(String[] seeds) {
      final CreateTree creator = new CreateTree();
      System.out.print(creator.createTree("/woods/forest"));
   }
}
```

A. The class compiles and always prints `true` at runtime.

B. The class compiles and always prints `false` at runtime.

C. The class compiles but the output cannot be determined until runtime.

D. The class compiles but may throw an exception at runtime.

E. The class does not compile.

77. What does the following print?

```
1:    class SmartWatch extends Watch {
2:      private String getType() { return "smart watch"; }
3:      public String getName() {
4:         return getType() + ",";
5:      }
6:   }
7:   public class Watch {
8:      private String getType() { return "watch"; }
9:      public String getName(String suffix) {
10:        return getType() + suffix;
11:     }
12:     public static void main(String[] args) {
13:        Watch watch = new Watch();
14:        Watch smartWatch = new SmartWatch();
15:        System.out.print(watch.getName(","));
16:        System.out.print(smartWatch.getName(""));
17:     }
18:  }
```

A. `smart watch,smart watch`

B. `smart watch,watch`

C. `watch,smart watch`

D. `watch,watch`

E. None of the above

78. In most of the United States, daylight savings time ends on November 5, 2017 at 02:00 a.m., and we repeat that hour. What is the output of the following?

```
import java.time.*;

public class FallBack {

   public static void main(String[] args) {
      LocalDate localDate = LocalDate.of(2017, Month.NOVEMBER, 5);
      LocalTime localTime = LocalTime.of(1, 0);
      ZoneId zone = ZoneId.of("America/New_York");
      ZonedDateTime z = ZonedDateTime.of(localDate, localTime, zone);
```

```
      for (int i = 0; i < 6; i++)
         z = z.plusHours(1);

      System.out.println(z.getHour());
   }
}
```

A. 5
B. 6
C. 7
D. The code does not compile.
E. The code compiles but throws an exception at runtime.

79. Which statements about the following application are true?

```
package party;
import java.util.concurrent.*;
public class Plan {
   private ExecutorService service = Executors.newCachedThreadPool();
   public void planEvents() {
      service.scheduleWithFixedDelay(
            () -> System.out.print("Check food stock"),
            1, TimeUnit.HOURS);
      service.scheduleAtFixedRate(
            () -> System.out.print("Check drink stock"),
            1, 1000, TimeUnit.SECONDS);
      service.execute(() -> System.out.print("Take out trash"));
   }
}
```

I. The `scheduleWithFixedDelay()` method call compiles.
II. The `scheduleAtFixedRate()` method call compiles.
III. The `execute()` method call compiles.

A. I only
B. II only
C. III only
D. I and II
E. I, II, and III
F. None of the above

80. Which of the following classes are checked exception? (Choose three.)

A. `java.io.NotSerializableException`

B. `java.lang.AssertionError`

C. `java.lang.IllegalArgumentException`

D. `java.sql.SQLException`

E. `java.text.ParseException`

F. `java.util.MissingResourceException`

81. Which of the following are valid functional interfaces? (Choose two.)

A.
```
interface CanClimb {default void climb() {}
   static void climb(int x) {}}
```

B. `interface CanDance {int dance() { return 5;}}`

C. `interface CanFly {abstract void fly();}`

D.
```
interface CanRun {void run();
   static double runFaster() {return 2.0;}}
```

E.
```
interface CanSwim {abstract Long swim();
   boolean test();}
```

82. How many of the following could be valid JDBC URL formats for an imaginary driver named `magic` and a database named `box`?

I. `jdbc;box;magic`

II. `jdbc;magic;@127.0.0.1:1234`

III. `jdbc;magic;//@127.0.0.1:1234`

IV. `jdbc;magic;127.0.0.1:1234/box`

V. `magic;jdbc;127.0.0.1:1234/box`

A. None

B. One

C. Two

D. Three

E. Four

F. Five

83. What is the output of the following?

```
Stream<String> s = Stream.of("speak", "bark", "meow", "growl");
Map<Integer, String> map = s.collect(toMap(String::length, k -> k));
System.out.println(map.size() + " " + map.get(4));
```

A. `2 bark`
B. `2 meow`
C. `4 bark`
D. `4 meow`
E. The output is not guaranteed.
F. The code compiles but throws an exception at runtime.

84. What is the output of the following application?

```
package music;
interface DoubleBass {
   void strum();
   default int getVolume() {return 5;}
}
interface BassGuitar {
   void strum();
   default int getVolume() {return 10;}
}
class ElectricBass implements DoubleBass, BassGuitar {
   @Override public void strum() {System.out.print("A");}
}
public class RockBand {
   public static void main(String[] strings) {
      final class MyElectricBass extends ElectricBass {
         public void strum() {System.out.print("E");}
      }
   }
}
```

A. `A`
B. `E`
C. The code compiles and runs without issue but does not print anything.
D. One line of code does not compile.
E. Two lines of code do not compile.
F. Three lines of code do not compile.

85. Which NIO.2 `Files` methods return a `Stream`? (Choose three.)
A. `find()`
B. `lines()`
C. `list()`
D. `listFiles()`
E. `readAllLines()`
F. `walkFileTree()`

Appendix

Answers to Review Questions

Chapter 1: Java Basics

1. D. An entry point in a Java application consists of a `main()` method with a single `String[]` argument, return type of `void`, and modifiers `public` and `static`. The name of the variable in the input argument does not matter. Option A is missing the `static` modifier, Option B is missing the `String[]` argument, and Option C has the wrong access modifier and method name. Only Option D fulfills these requirements. Note that the modifier `final` is optional and may be added to an entry point method.

2. A. The diagram is an example of object-oriented design in Java, making Option B a true statement. Options C and D are also true, as they follow from the inheritance model in the diagram. Option A is the correct answer, since platform independence has nothing to do with the diagram.

3. C. The proper extension for a Java compiled bytecode file is `.class`, making Option C the correct answer.

4. B. The fact that the `Date` class exists in both packages does not impact the ability to import both packages, so lines 1 and 2 compile without issue, and Option A is incorrect. Line 4 will not compile because the `Date` class used is ambiguous, making Option B correct and Option D incorrect. Finally, Option C is incorrect because line 5 does compile, as the fully qualified name of the class is used.

5. A. Options B, C, and D are each attributes of traditional object-oriented programming. Option A is incorrect as an object-oriented project tends to group data and the actions related to that data into a single object.

6. D. Only local variables have such a small scope, making Option D the correct answer.

7. B. The package `java.lang` is imported into every Java class, so Option B is correct. The other options must be explicitly imported. Option A exists but must be explicitly imported. Options C and D do not exist in the standard Java runtime.

8. C. Java accepts Options A, B, and D as valid comments. Note that the `/* */` syntax can have additional (and uneven) star (`*`) characters as shown in B and D. Option C is incorrect as hashtag (`#`) is not a valid comment character in Java.

9. D. A valid `.java` file may define any number of classes or interfaces but have at most one `public` class. It can also not define any `public` classes. For these reasons, Option A, B, and C are incorrect, leaving Option D as the only correct answer.

10. B. Notice in this question that `main()` is not a `static` method, therefore it can access both class and instance variables. Since there are two class variables and two instance variables defined, Option B is the correct answer.

11. B. A class will compile if it has unused or redundant `import` statements, making Option A and C incorrect. Option D is also incorrect as the compiler must be able to locate the class of the `import` statement. The correct answer is Option B. Removing unused `import` statements does not cause a class to become unable to be compiled.

12. A. The code does not compile because of line 5, making Option A the correct answer. For this question, it helps to understand variable scope. The `main()` method is `static` and does not have access to any class instance variables. The `birds` variable is not `static` and requires a class instance variable to access. Therefore, the code does not compile when the `static` method attempts to access a non-`static` variable without an instance of the class.

13. D. The `java` command can only execute compiled `.class` files, so I is false. Java is most certainly object oriented, one of the key design principles, so II is also false. The `javac` command compiles into bytecode, which must be run in a Java virtual machine (JVM), and is not native machine code, so III is false as well. Since none of the statements are true, Option D is the correct answer.

14. D. A class can start with a comment, an optional `package` statement, or an `import` statement if there is no `package` statement. It cannot start with a variable definition, making Option D the correct answer.

15. C. Classes may be defined without a package declaration and are placed in the default package, so Option A is incorrect. Option B is a completely false statement as no such file is required in Java. Option C is correct as it is one of the primary reasons for organizing your application into packages. Option D is incorrect as package-private allows access to methods and variables to be limited to those classes within the same package.

16. B. The compilation command requires the full or relative name of the file, including the `.java` extension, making Options A and C incorrect. The execution command requires the class name without a filename extension, making Option D incorrect. Option B is the only correct set of compilation and execution commands.

17. D. Encapsulation is the technique of removing access to a class's instance variables from processes outside the class, making Option D the correct answer.

18. D. The `height` variable is declared within the if-then statement block. Therefore, it cannot be referenced outside the if-then statement and the code does not compile.

19. A. A Java bytecode file is a binary encoded set of instructions designed to be run on any computer with a compatible JVM, making Option A the only correct answer. By compatible JVM, we mean one capable of running the class file. For example, a Java 6 JVM may have trouble executing a Java 8 compiled file. Option B is incorrect, and is more a facet of machine language compiled programming classes. Option C is also incorrect as binary data is not particularly human readable. Finally, Option D is incorrect as the compiled file can be distributed without its `.java` source file and execute without issue.

20. D. Unlike with some other programming languages, the proper way to terminate a line of code is with a semicolon (`;`), making D the only correct answer.

21. C. The code compiles and runs without issue, so Options A and B are incorrect. The question relies on your ability to understand variable scope. The variable `today` has local scope to the method in which it is executed. The variable `tomorrow` is re-declared in the method, but the reference used on line 7 is to the instance variable with a value of `10`. Finally, the variable `tomorrow` is `static`. While using an instance reference to access a `static` variable is not recommended, it does not prevent the variable from being read. The result is line 7 evaluates and prints (20 + 10 + 1) = 31, making C the correct answer.

22. C. Line 1 is missing the `class` keyword. Line 2 contains two types for the same variable. Line 3 is a valid definition for a method, making C the correct answer. Finally, line 4 contains an access modifier, `private`, after the return type, which is not allowed. In addition, `void` is an invalid type for variables.

23. D. Platform independence is the property of Java that allows it to be run on a variety of different devices.

24. A. Options B, C, and D are each correct statements about JVMs. Option A is incorrect. Not only is it not a statement about JVMs, it is actually false as Java bytecode can often be easily decoded/decompiled.

25. B. There is no such thing as package variables, so Option A is incorrect. Option C is incorrect as the variable is only in scope within a specific instance of the class. Option D is also incorrect as the variable is only in scope for a single method that it is defined in. Option B is the only correct answer as class variables are in scope within the program.

26. C. Option A is incorrect as the sub-package `recurring` is not included by the `import` statements. Option B is also incorrect as it uses the plural `directors` instead of the singular `director` used in the `import` statements. Option D is incorrect as the wildcard is applied to the sub-package `movie.director`, not the package `movie`. Finally, Option C is correct as it is a valid class accessible from the wildcard import.

27. D. Java classes are defined in this order: `package` statement, `import` statements, class declaration, making Option D the only correct answer. Note that not all of these statements are required. For example, a class may not have a `package` statement, but if it does, it must come first in the file.

28. D. The `import` statements for `stars.*` and `stars.Blackhole` are redundant `import` statements, since only the class `Blackhole` is used, and therefore one of them can be safely removed. The `import` statements `java.lang.*` and `java.lang.Object` are both not required as `java.lang` is automatically imported in every Java class. Therefore, three of the four `import` statements can be safely removed, making the correct answer Option D.

29. C. The application prints the third argument of the input methods. Note that double quotes `""` group input arguments. Therefore, the third argument of Option A is `White-tailed deer`. The third argument of Option B is `3`. The third argument of Option C is `White-tailed`, making it the correct answer. Finally, Option D only has two input arguments, leading to an `ArrayIndexOutOfBoundsException` trying to read the third argument at runtime.

30. B. The `javac` command compiles a `.java` file into a `.class` bytecode file, making Option B the correct answer.

31. B. Java is object oriented, not procedural, so Option A is a false statement. Java allows method overloading in subclasses, so Option B is correct. Operator overloading is permitted in languages like C++, not Java, so Option C is also untrue. Finally, Option D is not a true statement as the JVM manages the location of objects in memory that can change and is transparent to the Java application.

32. D. Option A is incorrect as the return type of the method cannot be `null`. Option B is also incorrect as the return type cannot be `void` if the method uses a `return` statement. Option C is incorrect too as the `class` keyword is replaced with `int`. Option D is correct because it's the only answer that allows the code to compile without issue. Note that other values are possible for this question. For example, either `int` or `long` can be entered in the last blank. The key here is that only one of the available answer choices allows the code to compile.

33. A. The code compiles so Option D is incorrect. The input to the constructor is ignored, making the assignment of `end` to be `4`. Since `start` is `2`, the subtraction of `4` by `2` results in the application printing `2`, followed by `5`, making Option A the correct answer.

34. D. Option A is a false statement, while Options B and C are actually arguments against using inheritance. Option D is one of the most important reasons Java supports inheritance, to allow increased code reuse among classes.

35. A. The double slash (`//`) syntax can have any number of slashes as a comment, so long as it starts with two of them, making Option A the correct answer. The (`#`) is not a comment character in Java, regardless of whether it is followed by a (`!`), so Option B is incorrect. Option C is incorrect as a single slash (`/`) is not a valid comment in Java. Finally, Option D is incorrect as Option A is a valid comment.

36. B. An entry point in a Java application consists of a `main()` method with a single `String[]` argument, return type of `void`, and modifiers `public` and `static`. Option D is the typical syntax for this method, although Options A and C are also valid forms of this method. Note that the modifier `final` is optional and may be added to the method signature. Furthermore, the `main()` method may take a vararg or array. Option B is the only invalid declaration as it does not take an array as an argument.

37. B. The line of code cannot be inserted at `a1` because no variables are allowed outside of the class declaration in this file, making Options A and D incorrect. The line of code can also not be inserted at `a3` as local variables defined within methods cannot have access modifiers such as `public`, making Option C incorrect. The code can be inserted independently at `a2` and `a4` as instance variables can be defined anywhere in the class outside a method. Therefore, Option B is the correct choice.

38. A. Option A is the only correct answer as a `class` definition is the only required component in a Java class file. Note that we said a Java class file here; Java also allows interfaces and enums to be defined in a file. A `package` statement and `import` statements are optional for declaring a class, making Options B and C incorrect. A class may also be defined with package-level access in a file, making Option D an incorrect answer.

39. D. The proper extension for a Java compiled bytecode file is `.java`, making Option D the correct answer.

40. C. Remember that `java.lang` is automatically imported in all Java classes, therefore both `java.lang.Math` and `pocket.complex.Math` are both imported into this class. Importing both sets of packages does not cause any compilation issues, making Option A incorrect. Line 3 is unnecessary import but including it does not prevent the class from compiling, making Option B incorrect. While both versions of `Math` may be imported into the class, the usage of the `Math` class requires a package name. Because of this, line 6 does not compile as the class reference is ambiguous, making Option C the correct answer and Option D incorrect.

41. A. Options B and C are accessible within the class as they are covered by the `import` statements. Option D is also fine as `java.lang.Object` is available without an explicit import. The only class not automatically accessible within the class without the full package name is `dog.puppy.female.KC` as the `import` statements do not include sub-packages; therefore, Option A is the correct answer.

42. B. Object-oriented programming is the technique of structuring data into objects, which may contain data and a set of actions that operate on the data, making Option B the correct answer.

43. A. All of the `import` statements in this class are required. Removing any of them would cause the class to not compile, making Option A the correct answer.

44. C. The `numLock` variable is not accessible in the `static main()` method without an instance of the `Keyboard` class; therefore, the code does not compile, and Option C is the correct answer.

45. D. The code compiles and runs without issue, so Option A is incorrect. The question involves understanding the value and scope of each variable at the `print()` statement. The variables `feet` and `tracks` are locally scoped and set to `4` and `15`, respectively, ignoring the value of `tracks` of `5` in the instance of the class. Finally, the `static` variable `s.wheels` has a value of `1`. The result is the combined value is `20`, making Option D the correct answer.

46. B. First off, the `color` variable defined in the instance and set to `red` is ignored in the method `printColor()` as local scope overrides instance scope, so Option A is incorrect. The value of `color` passed to the `printColor()` method is `blue`, but that is lost by the assignment to `purple`, making Option B the correct answer and Option C incorrect. Option D is incorrect as the code compiles and runs without issue.

47. C. The `javac` command takes a text-based `.java` file and returns a binary bytecode `.class` file, making II a true statement. The `java` command uses a period (`.`) to separate packages, not a slash (`/`), making I a true statement and III a false statement. For these reasons, Option C is the correct answer.

48. D. The application compiles without issue, so Option C is incorrect. The application does not execute though, as the `main()` method does not have the correct method signature. It is missing the required input argument, an array of `String`. Trying to execute the application without a proper entry point produces an error, making Option D the correct answer.

49. C. Option A does not compile because it is missing the closing bracket for the class. Option D does also not compile as `void` is not a valid type for a variable. Regardless, Options A and D are incorrect as they are missing the `getRating()` method. Note that Option A also uses an abbreviation for `numberOfPages`. Option B is incorrect as it is missing the `numberOfPages` attribute. Option C is the correct answer as it properly defines the attribute `numberOfPages` and method `getRating()`.

50. C. Garbage collection can happen at any time while an application is running, especially if the available memory suddenly becomes low, making Option A incorrect. Option B is also incorrect, since it is trivial to create a Java application with an infinite loop that never terminates. Option D is incorrect because the computer must be able to run the JVM in order to execute a Java class. Option C is the only correct answer, as the JVM does require an entry point method to begin executing the application.

Chapter 2: Working with Java Data Types

1. A. Option A does not compile because Java does not allow declaring different types as part of the same declaration. The other three options show various legal combinations of combining multiple variables in the same declarations with optional default values.

2. D. The `table` variable is initialized to `"metal"`. However, `chair` is not initialized. In Java, initialization is per variable and not for all the variables in a single declaration. Therefore, the second line tries to reference an uninitialized local variable and does not compile, which makes Option D correct.

3. B. Instance variables have a default value based on the type. For any non-primitive, including `String`, that type is a reference to `null`. Therefore Option B is correct. If the variable was a local variable, Option C would be correct.

4. B. An identifier name must begin with a letter, `$`, or `_`. Numbers are only permitted for subsequent characters. Therefore, Option B is not a valid variable name.

5. B. In Java, class names begin with an uppercase letter by convention. Then they use lowercase with the exception of new words. Option B follows this convention and is correct. Option A follows the convention for variable names. Option C follows the convention for constants. Option D doesn't follow any Java conventions.

6. C. Objects have instance methods while primitives do not. Since `int` is a primitive, you cannot call instance methods on it. `Integer` and `String` are both objects and have instance methods. Therefore, Option C is correct.

7. C. Underscores are allowed between any two digits in a numeric literal. Underscores are not allowed at the beginning or end of the literal, making Option C the correct answer.

8. C. Option A is incorrect because `int` is a primitive. Option B is incorrect because it is not the name of a class in Java. While Option D is a class in Java, it is not a wrapper class because it does not map to a primitive. Therefore, Option C is correct.

9. C. There is no class named integer. There is a primitive `int` and a class `Integer`. Therefore, the code does not compile, and Option C is correct. If the type was changed to `Integer`, Option B would be correct.

10. C. The `new` keyword is used to call the constructor for a class and instantiate an instance of the class. A primitive cannot be created using the `new` keyword. Dealing with references happens after the object created by `new` is returned.

11. D. Java uses the suffix `f` to indicate a number is a `float`. Java automatically widens a type, allowing a float to be assigned to either a `float` or a `double`. This makes both lines `p1` and `p3` compile. Line `p2` does compile without a suffix. Line `p4` does not compile without a suffix and therefore is the answer.

12. A. A `byte` is smaller than a `char`, making Option C incorrect. `bigint` is not a primitive, making Option D incorrect. A `double` uses twice as much memory as a `float` variable, therefore Option A is correct.

13. D. The instance variables, constructor, and method names can appear in any order within a class declaration.

14. B. Java does not allow multiple Java data types to be declared in the same declaration, making Option B the correct answer. If `double` was removed, both `hot` and `cold` would be the same type. Then the compiler error would be on `x3` because of a reference to an uninitialized variable.

15. C. Lines 2 and 7 illustrate instance initializers. Line 6 is a static initializer. Lines 3–5 are a constructor.

16. A. Since `defaultValue` is a local variable, it is not automatically initialized. That means the code will not compile with any type. Therefore, Option A is correct. If this was an instance variable, Option C would be correct as `int` and `short` would be initialized to `0` while `double` would be initialized to `0.0`.

17. A. The `finalize()` method may not be called, such as if your program crashes. However, it is guaranteed to be called no more than once.

18. D. `String` is a class, but it is not a wrapper class. In order to be a wrapper class, the class must have a one-to-one mapping with a primitive.

19. C. Lines 15–17 create the three objects. Lines 18–19 change the references so `link2` and `link3` point to each other. The lines 20–21 wipe out two of the original references. This means the object with `name` as x is inaccessible.

20. C. Options A and D are incorrect because `byte` and `short` do not store values with decimal points. Option B is tempting. However, `3.14` is automatically a `double`. It requires casting to `float` or writing `3.14f` in order to be assigned to a `float`. Therefore, Option C is correct.

21. B. `Integer` is the name of a class in Java. While it is bad practice to use the name of a class as your local variable name, this is legal. Therefore, `k1` does compile. It is not legal to use a reserved word as a variable name. All of the primitives including `int` are reserved words. Therefore, `k2` does not compile, and Option B is the answer. Line `k4` doesn't compile either, but the question asks about the first line to not compile.

22. B. Dot notation is used for both reading and writing instance variables, assuming they are in scope. It cannot be used for referencing local variables, making Option B the correct answer.

23. C. Class names follow the same requirements as other identifiers. Underscores and dollar signs are allowed. Numbers are allowed, but not as the first character of an identifier. Therefore, Option C is correct. Note that class names begin with an uppercase letter by convention, but this is not a requirement.

24. D. This question is tricky as it appears to be about primitive vs. wrapper classes. Looking closely, there is an underscore right before the decimal point. This is illegal as the underscore in a numeric literal can only appear between two digits.

25. C. Local variables do not have a default initialization value. If they are referenced before being set to a value, the code does not compile. Therefore, Option C is correct. If the variable was an instance variable, Option B would be correct. Option D is tricky. A local variable will compile without an initialization if it isn't referenced anywhere or it is assigned a value before it is referenced.

26. C. Since `defaultValue` is an instance variable, it is automatically initialized to the corresponding value for that type. For `double`, that value is `0.0`. By contrast, it is `0` for `int`, `long`, and `short`. Therefore Option C is correct.

27. B. Option B is an example of autoboxing. Java will automatically convert from primitive to wrapper class types and vice versa. Option A is incorrect because you can only call methods on an object. Option C is incorrect because this method is used for converting to a wrapper class from a `String`. Option D is incorrect because autoboxing will convert the primitive to an object before adding it to the `ArrayList`.

28. C. Java does not allow calling a method on a primitive. While autoboxing does allow the assignment of an `Integer` to an `int`, it does not allow calling an instance method on a primitive. Therefore, the last line does not compile.

29. D. In order to call a constructor, you must use the `new` keyword. It cannot be called as if it was a normal method. This rules out Options A and B. Further, Option C is incorrect because the parentheses are required.

30. A. Option A (I) correctly assigns the value to both variables. II does not compile as `dog` does not have a type. Notice the semicolon in that line, which starts a new statement. III compiles but only assigns the value to `dog` since a declaration only assigns to one variable rather than everything in the declaration. IV does not compile because the type should only be specified once per declaration.

31. C. The wrapper class for `int` is `Integer` and the wrapper class for `char` is `Character`. All other primitives have the same name. For example, the wrapper class for `boolean` is `Boolean`.

32. A. Assuming the variables are not primitives, they allow a `null` assignment. The other statements are false.

33. A. An example of a primitive type is `int`. All the primitive types are lowercase, making Option A correct. Unlike object reference variables, primitives cannot reference `null`. `String` is not a primitive as evidenced by the uppercase letter in the name and the fact that we can call methods on it. You can create your own classes, but not primitives.

34. D. While you can suggest to the JVM that it might want to run a garbage collection cycle, the JVM is free to ignore your suggestion. Option B is how to make this suggestion. Since garbage collection is not guaranteed to run, Option D is correct.

35. C. All three references point to the `String` apple. This makes the other two `String` objects eligible for garbage collection and Option C correct.

36. B. A constructor can only be called with a class name rather than a primitive, making Options A and C incorrect. The newly constructed `Double` object can be assigned to either a `double` or `Double` thanks to autoboxing. Therefore, Option B is correct.

37. B. First line 2 runs and sets the variable using the declaration. Then the instance initializer on line 6 runs. Finally, the constructor runs. Since the constructor is the last to run of the three, that is the value that is set when we print the result, so Option B is correct.

38. C. Objects are allowed to have a `null` reference while primitives cannot. `int` is a primitive, so assigning `null` to it does not compile. `Integer` and `String` are both objects and can therefore be assigned a `null` reference. Therefore, Option C is correct.

39. C. An instance variable can only be referenced from instance methods in the class. A `static` variable can be referenced from any method. Therefore, Option C is correct.

40. B. Underscores are allowed between any two digits in a numeric literal. Underscores are not allowed adjacent to a decimal point, making Option B the correct answer.

41. A. These four types represent nondecimal values. While you don't need to know the exact sizes, you do need to be able to order them from largest to smallest. A `byte` is smallest. A `short` comes next, followed by `int` and then `long`. Therefore, Option A is correct.

42. A. Java uses dot notation to reference instance variables in a class, making Option A correct.

43. B. If there was a `finalize()` method, this would be a different story. However, the method here is `finalizer`. Tricky! That's just a normal method that doesn't get called automatically. Therefore `clean` is never output.

44. A. Options B and C do not compile. In Java, braces are for arrays rather than instance variables. Option A is the correct answer. It uses dot notation to access the instance variable. It also shows that a private variable is accessible in the same class and that a narrower type is allowed to be assigned to a wider type.

45. B. The `parseInt()` methods return a primitive. The `valueOf()` methods return a wrapper class object. In real code, autoboxing would let you assign the return value to either a primitive or wrapper class. In terms of what gets returned directly, Option B is correct.

46. B. On line 9, all three objects have references. The `elena` and `zoe` objects have a direct reference. The `diana` object is referenced through the `elena` object. On line 10, the reference to the `diana` object is replaced by a reference to the `zoe` object. Therefore, the `diana` object is eligible to be garbage collected, and Option B is correct.

47. C. Options A and B are `static` methods rather than constructors. Option D is a method that happens to have the same name as the class. It is not a constructor because constructors don't have return types.

48. A. Remember that garbage collection is not guaranteed to run on demand. If it doesn't run at all, Option B would be output. If it runs at the requested point, Option C would be output. If it runs right at the end of the `main()` method, Option D would be output. Option A is the correct answer because `play` is definitely called twice. Note that you are unlikely to see all these scenarios if you run this code because we have not used enough memory for garbage collection to be worth running. However, you still need to be able to answer what could happen regardless of it being unlikely.

49. B. Each wrapper class has a constructor that takes the primitive equivalent. The methods mentioned in Options A, C, and D do not exist.

50. C. The `main()` method calls the constructor which outputs `a`. Then the main method calls the `run()` method. The `run()` method calls the constructor again, which outputs `a` again. Then the `run()` method calls the `Sand()` method, which happens to have the same name as the constructor. This outputs `b`. Therefore, Option C is correct.

Chapter 3: Using Operators and Decision Constructs

1. B. A `switch` statement supports the primitive types `byte`, `short`, `char`, and `int` and the classes `String`, `Character`, `Byte`, `Short`, and `Integer`. It also supports enumerated types. Floating-point types like `float` and `double` are not supported, therefore Option B is the correct answer.

2. A. Remember that in ternary expressions, only one of the two right-most expressions are evaluated. Since `meal>6` is `false`, `--tip` is evaluated and `++tip` is skipped. The result is that `tip` is changed from `2` to `1`, making Option A the correct answer. The value of `total` is `6`, since the pre-increment operator was used on `tip`, although you did not need to know this to solve the question.

3. C. The first assignment creates a new `String "john"` object. The second line explicitly uses the `new` keyword, meaning a new `String` object is created. Since these objects are not the same, the `==` test on them evaluates to `false`. The `equals()` test on them returns `true` because the values they refer to are equivalent. Therefore, the correct answer is C.

4. D. This code does not compile because it has two `else` statements as part of a single if-then statement. Notice that the second `if` statement is not connected to the last `else` statement. For this reason, Option D, none of the above, is the correct answer.

5. C. A `default` statement inside a `switch` statement is optional and can be placed in any order within the `switch`'s `case` statements, making Options A and B incorrect. Option D is an incorrect statement as a `switch` statement can be composed of a single `default` statement and no `case` statements. Option C is correct because a `default` statement does not take a value, unlike a `case` statement.

6. B. The initial assignment of `thatNumber` follows the first branch of the ternary expression. Since `5 >= 5` evaluates to `true`, a value of `3` is assigned to `thatNumber`. In the next line, the pre-increment operator increments the value of `thatNumber` to `4` and returns a value of `4` to the expression. Since `4 < 4` evaluates to `false`, the if-then block is skipped. This leaves the value of `thatNumber` as 4, making Option B the correct answer.

7. B. The `break` statement exits a `switch` statement, skipping all remaining branches, making Option B the correct answer. In Option A, `exit` is not a statement in Java. In Option C, `goto` is a reserved word but unused in Java. Finally, in Option D, `continue` is a statement but only used for loops.

8. C. Option A is incorrect as only one of the two right-hand expressions is evaluated at runtime. Parentheses are often helpful for reading ternary expressions but are not required, making Option B incorrect. Option C is a correct statement about ternary operators as they are commonly used to replace short if-then-else statements. Finally, Option D is incorrect as only `boolean` expressions are permitted in the left-most operand of a ternary expression.

9. C. On line 4, `candidateA` and `candidateB` are numbers, but the `&&` operation can only be applied to `boolean` expressions. Therefore, the code does not compile because of line 4, making C the correct answer. All of the other lines are correct. Note that if line 4 is fixed, line 3 does not produce a `NullPointerException` at runtime. The conditional `||` and the preceding `null` check allows the code to only call `intValue()` if `candidateA` is not `null`.

10. A. The first step is to determine whether or not the if-then statement's expression is executed. The expression `6 % 3` evaluates to `0`, since there is no remainder, and since `0 >= 1` is `false`, the expression `triceratops++` is not called. Notice there are no brackets `{}` in the if-then statement. Despite the `triceratops--` line being indented, it is not part of the if-then statement. Recall that Java does not use indentation to determine the beginning or end of a statement. Therefore, `triceratops--` is always executed, resulting in a value of `2` for `triceratops` and making Option A the correct answer.

11. D. Option A is incorrect because `else` statements are entirely optional. Option B is also incorrect. The target of an if-then statement is not evaluated if the `boolean` test is `false`. Option C is incorrect. While an if-then statement is often used to test whether an object is of a particular type in order to cast it, it is not required to cast an object. Option D is correct as an if-then statement may execute a single statement or a block of code `{}`.

12. D. For this question, it helps to notice that the second if-then statement is not connected to the first if-then statement, as there is no `else` joining them. When this code executes, the first if-then statement outputs `Not enough` since `flair` is `>= 15` and `< 37`. The second if-then statement is then evaluated. Since `flair` is not `37`, the expression `Too many` is outputted. Since two statements are outputted, Option D, none of the above, is the correct answer.

13. B. A `case` value must be a constant expression, such as a literal or `final` variable, so Options A and C are true statements about `case` values. A `case` statement may be terminated by a `break` statement, but it is not required, making Option B the false statement and correct answer. Option D is also a true statement about `case` values.

14. D. The question is about `boolean` operators. Since Options A and B are numeric operators, they can be instantly disregarded. The question then simplifies to which `boolean` expression, `&&` or `||`, corresponds to the truth table that only evaluates to `true` if both operands are `true`. Only the conjunctive logical `&&` operator represents this relationship, making Option D the correct answer.

15. C. The value of `jumps` and `hops` is unimportant because this code does not compile, making Option C the correct answer. Unlike some other programming languages, Java does not automatically convert integers to `boolean` values for use in if-then statements. The statement `if(jumps)` evaluates to `if(0)`, and since `0` is not a `boolean` value, the code does not compile. Note that the value of the `jumps` variable is irrelevant in this example; no integer evaluates to a `boolean` value in Java.

16. B. Prefix operators modify the variable and evaluate to the new value, while postfix operators modify the variable but return the original value. Therefore, Option B is the correct answer.

17. B. For this problem, it helps to recognize that parentheses take precedence over the operations outside the parentheses. Once we replace the variables with values, the expression becomes: `3+2*(2+3)`. We then calculate the value inside the parentheses to get `3+2*5`. Since the multiplication operator has higher precedence than addition, we evaluate it first, resulting in `3+10 = 13`, making Option B the correct answer.

18. B. Any value that can be implicitly promoted to `int` will work for the `case` statement with an `int` input. Since `switch` statements do not support `long` values, and `long` cannot be converted to `int` without a possible loss of data, Option B is the correct answer.

19. D. While parentheses are recommended for ternary operations, especially embedded ones, they are not required, so Option C is incorrect. The code does not compile because `day` is an `int`, not a `boolean` expression, in the second ternary operation, making Option D the correct answer. Remember that in Java, numeric values are not accepted in place of `boolean` expressions in if-then statements or ternary operations.

20. C. While the code involves numerous operations, none of that matters for solving this problem. The key to solving it is to notice that the line that assigns the `leaders` variable has an uneven number of parentheses. Without balanced parentheses, the code will not compile, making Option C the correct answer.

21. B. Remember that Java evaluates `+` from left to right. The first two values are both numbers, so the `+` is evaluated as numeric addition, resulting in a reduction to `11 + "7" + 8 + 9`. The next two terms, `11 + "7"`, are handled as string concatenation since one of the terms is a `String`. This allows us to reduce the expression to `"117" + 8 + 9`. Likewise, the final two terms are each evaluated one at a time with the `String` on the left. Therefore, the final value is `11789`, making Option B the correct answer.

22. B. The subtraction `-` operator is used to find the difference between two numbers, while the modulus `%` operator is used to find the remainder when one number is divided by another, making Option B the correct answer. The other options use operators that do not match this description.

23. B. The code compiles without issue, making Option D incorrect. The focus of this question is showing how the division and modulus of two numbers can be used to reconstitute one of the original operands. In this example, `partA` is the integer division of the two numbers. Since `3` does not divide `11` evenly, it is rounded down to `3`. The variable `partB` is the remainder from the first expression, which is `2`. The `newDog` variable is an expression that reconstitutes the original value for `dog` using the division value and the remainder. Note that due to operator precedence, the multiplication `*` operation is evaluated before the addition `+` operation. The result is the original value of `11` for `dog` is outputted by this program.

24. B. The code compiles without issue, so Option D is incorrect. In this question's `switch` statement, there are no `break` statements. Once the matching `case` statement, `30`, is reached, all remaining `case` statements will be executed. The variable `eaten` is increased by `1`, then `2`, then reduced by `1`, resulting in a final value of `2`, making Option B the correct answer.

25. C. Ternary operations require both right-hand expressions to be of compatible data types. In this example, the first right-hand expression of the outer ternary operation is of type `String`, while the second right-hand expression is of type `int`. Since these data types are incompatible, the code does not compile, and Option C is the correct answer.

26. A. For this question, remember that if two `String` objects evaluate to `true` using `==`, then they are the same object. If they are the same `String` object, `equals()` will trivially return `true`. Option A correctly reflects this principle. Option B is incorrect as two `String` objects that are not the same may still be equivalent in terms of `equals()`. For example, `apples == new String(apples)` evaluates to `false`, but `equals()` will evaluate to `true` on these `String` objects. Likewise, Options C and D are also incorrect because two `String` objects that are equivalent in terms of `equals()` may be different objects.

27. B. The statement compiles and runs without issue, making Options C and D incorrect. Since we are given that `myTestVariable` is not `null`, the statement will always evaluate to `false`, making Option B the correct answer. Note that if `myTestVariable` was `null`, then the code would still compile but throw a `NullPointerException` calling `equals()` at runtime.

28. D. The code does not compile, making Option D the correct answer. The reason the code does not compile is due to the test in the second if-then statement. The expression `(streets && intersections > 1000)` is invalid because `streets` is not a `boolean` expression and cannot be used as the left-hand side of the conjunctive logical `&&` operator. The line of code is designed to resemble the corrected expression `(streets > 1000 && intersections > 1000`. Notice the fixed expression requires two relational `>` operators. If the second if-then statement was corrected, then the application would compile and produce two `1`'s, making Option C the correct answer.

29. B. The `&` and `&&` (AND) operators are not interchangeable, as the conjunctive `&` operator always evaluates both sides of the expression, while the conditional conjunctive `&&` operator only evaluates the right-hand side of the expression if the left side is determined to be `true`. This is why conditional operators are often referred to as short-circuit operators, skipping the right-hand side expression at runtime. For these reasons, Option B is the correct answer. Note that Option C is an incorrect statement as well, since it describes disjunctive (OR) operators.

30. C. The code compiles, so Option A is incorrect. Since `w` starts out `true`, the third line takes the first right-hand side of the ternary expression returning and assigning `5` to `x` (post-increment operator) while incrementing `y` to `6`. Note that the second right-hand side of the ternary expression `y--` is not evaluated since ternary operators only evaluate one right-hand expression at runtime. On the fourth line, the value of `w` is set to `!z`. Since `z` is `false`, the value of `w` remains `true`. The final line outputs the value of `(5+6)` and `(true ? 5 : 10)`, which is `11 5`, making Option C the correct answer.

31. A. The first assignment actually uses two `String` objects, the literal `"bob"` and the `String` created with the `new` keyword. Regardless, only the second object is assigned to the variable `bob`. The second variable, `notBob`, is assigned a reference to the value of the `bob` variable. This means that not only does the `equals()` test pass, but they are actually the same object, so the `==` test is `true` as well. Therefore, the correct answer is Option A.

32. B. The question is about operator precedence and order of operation. The multiplication `*` and modulus `%` operators have the highest precedence, although what is inside the parentheses needs to be evaluated first. We can reduce the expression to the following: `12 + 6 * 3 % 2`. Since multiplication `*` and modulus `%` have the same operator precedence, we evaluate them from left to right as follows: `12 + 6 * 3 % 2` → `12 + 18 % 2` → `12 + 0` → `12`. We see that despite all of the operators on the right-hand side of the expression, the result is zero, leaving us a value of `12`, making Option B the correct answer.

33. D. The XOR `^` operator evaluates to `true` if `p` and `q` differ and `false` if they are the same. Therefore, the missing values are `true` and `false`, making Option D the correct answer.

34. C. The key to understanding this question is to remember that the conditional conjunction `&&` operator only executes the right-hand side of the expression if the left-hand side of the expression is `true`. If `data` is an empty array, then the expression ends early and nothing is output. The second part of the expression will return `true` if `data`'s first element is `sound` or `logic`. Since we know from the first part of the statement that `data` is of length at least one, no exception will be thrown. The final part of the expression with `data.length<2` doesn't change the output when `data` is an array of size one. Therefore, `sound` and `logic` are both possible outputs. For these reasons, Option C is the only result that is unexpected at runtime.

35. C. In Option A, the division operator `/` incorrectly comes after the decrement `--` operator. In Option B, the subtraction operator `-` incorrectly comes after the modulus `%` operator. In Option D, the division operator `/` incorrectly comes after the subtraction `-` operator. The correct answer is Option C, where all three operators have the same order of precedence.

36. D. The exclusive or (XOR) `^` operator requires evaluating both operands to determine the result. For this reason, Options A and B are incorrect. For Option B, you can't have a short-circuit operation if both operands are always read, therefore `^^` does not exist. Option C is an incorrect statement as the `^` operator only returns `true` if exactly one operand is `true`. Finally, Option D is correct as the `^` is only applied to `boolean` values in Java.

37. C. The diagram represents the overlap of `x` and `y`, corresponding to when one of them is true. Therefore, `x || y`, Option C, most closely matches this relationship. Note that `z` is unused in the diagram and therefore is not required in any expression.

38. D. The value of a `case` statement must be constant, a literal value, or `final` variable. Since `red` is missing the `final` attribute, no variable type allows the code to compile, making Option D the correct answer.

39. C. The question is asking which operator represents greater than or equal to and which operator is strictly less than. The `>=` and `<` correspond to these operators, respectively. Therefore, Option C is the correct answer. Note that the question does not specify which order the operators needed to appear in, only to select the two operators that match the question description.

40. B. The code compiles and runs without issue, making Options C and D incorrect. The key here is understanding operator precedence and applying the parentheses to override precedence correctly. The first expression is evaluated as follows: `10 * (2 + (3 + 2) / 5)` → `10 * (2 + 5 / 5)` → `10 * (2 + 1)` → `10 * 3`, with a final value of `30` for `turtle`. Since `turtle` is not less than `5`, a value of `25` is assigned to `hare`. Since `turtle` is not less than `hare`, the last expression evaluates to `Turtle wins!`, which is outputted to the console, making Option B the correct answer.

41. A. All of the terms of `getResult()` in this question evaluate to `0`, since they are all less than or equal to `5`. The expression can therefore be reduced to `0+0+0+0+""`. Since Java evaluates the `+` operator from left to right, the four operands on the left are applied using numeric addition, resulting in the expression `0+""`. This expression just converts the value to a `String`, resulting in an output of `0`, making Option A the correct answer.

42. A. The code compiles without issue, so Option D is incorrect. The key here is that the if-then statement in the `runTest()` method uses the assignment operator (`=`) instead of the (`==`) operator. The result is that `spinner` is assigned a value of `true`, and the statement `(spinner = roller)` returns the newly assigned value. The method then returns `up`, making Option A the correct answer. If the (`==`) operator had been used in the if-then statement, then the process would have branched to the `else` statement, with `down` being returned by the method.

43. D. The conditional disjunction (OR) `||` operator is `true` if either of the operands are `true`, while the logical complement (`!`) operator reverses or flips a `boolean` value, making Option D the correct answer. The other options use operators that do not match this description. In particular, Options A and C include operators that can only be applied to numerical values, not `boolean` ones.

44. A. While parentheses are recommended for ternary operations, especially embedded ones, they are not required, so Option C is incorrect.. The first ternary operation evaluates `characters <= 4` as `false`, so the second ternary operation is executed. Since `story > 1` is `true`, the final value of `movieRating` is `2.0`, making Option A the correct answer.

45. B. Barring any JVM limitations, a `switch` statement can have any number of `case` statements (including none) but at most one `default` statement, with Option B correctly identifying this relationship.

46. A. The application uses the conditional conjunction `&&` operator to test if `weather[0]` is `null`, but unfortunately this test does not work on zero-length arrays. Therefore, it is possible this code will throw an `ArrayIndexOutOfBoundsException` at runtime. The second

part of the expression evaluates to `true` if the first input of `weather` matches `sunny`. The final part of the expression, `&& !false`, is a tautology in that it is always `true` and has no impact on the expression. Either an exception will be thrown or text will be output, based on the value of `weather`, therefore Option A is the correct answer.

47. D. The question looks a lot more difficult than it is. In fact, to solve it you don't have to compute anything! You just have to notice that the logical complement operator (`!`), which can only be applied to `boolean` values, is being applied to a numeric value. Therefore, the answer is that the expression wouldn't compile or run, making Option D the correct answer.

48. C. The disjunctive logical `||` operator evaluates to `true` if either operand is `true`. Another way to look at it is that it only evaluates to `false` if both operands are `false`. Therefore, the missing values are both `true`, making Option C the correct answer.

49. A. In Option B, the subtraction operator `-` incorrectly comes after the decrement `--` operator. In Option C, the division operator `/` incorrectly comes after the increment `++` operator. In Option D, the modulus operator `%` incorrectly comes after the increment `++` operator. The correct answer is Option A, where the subtraction `-` and addition `+` operators are followed by the division `/` and multiplication `*` operators.

50. C. The key to solving this problem is remembering that the type of the value returned by a ternary operation is determined by the expressions on the right-hand side. On line `p1`, the expressions are of type `int`, but the assignment is to the variable `game`, of type `String`. Since the assignment is invalid, the code does not compile, and Option C is correct.

Chapter 4: Creating and Using Arrays

1. B. Three dots (`...`) are the syntax for a method parameter of type varargs. It is treated like an array.

2. B. Array indexes are zero based in Java. A varargs parameter is simply another way of passing in data to a method. From within the method, it is treated just like you had written `Frisbee[] f` as the method parameter. Therefore, the first element uses the 0th index, and Option B is correct.

3. D. Trick question! While `int` is a primitive, all arrays are objects. One way to tell is that an array has a public instance variable called `length`. Another way is that you can assign it a variable of type `Object`. Therefore, Option D is correct.

4. C. The array braces are allowed to appear before or after the variable name, making the `tiger` and `bear` declarations correct. The braces are not allowed to appear before the type making the `lion` declaration incorrect. Therefore, Option C is correct.

5. C. From within a method, an array or varargs parameter is treated the same. However, there is a difference from the caller's point of view. A varargs parameter can receive either an array or individual values, making Options A and B compile. However, an array parameter can only take an array, which prevents Option C from compiling.

6. A. Arrays use the `length` variable to determine the number of elements, making Option A correct. For an `ArrayList`, Option D would have been the answer.

7. C. A two-dimensional array is declared by listing both sizes in separate pairs of braces. Option C correctly shows this syntax.

8. B. There is nothing wrong with this code. It correctly creates a seven-element array. The loop starts with index 0 and ends with index 6. Each line is correctly output. Therefore, Option B is correct.

9. B. Sorry. This is just something you have to memorize. The `sort()` and `binarySearch()` methods do sorting and searching, respectively.

10. B. The elements of the array are of type `String` rather than `int`. Therefore, we use alphabetical order when sorting. The character 1 sorts before the character 9, alphabetically making Option A incorrect. Shorter strings sort before longer strings when all the other characters are the same, making Option B the answer.

11. B. Array indices start with 0, making Options C and D incorrect. The `length` attribute refers to the number of elements in an array. It is one past the last valid array index. Therefore, Option B is correct.

12. C. When using an array initializer, you are not allowed to specify the size separately. The size is inferred from the number of elements listed. Therefore, `tiger` and `ohMy` are incorrect. When you're not using an array initializer, the size is required. An empty array initializer is allowed. Option C is correct because `lion` and `bear` are legal.

13. B. Since no elements are being provided when creating the arrays, a size is required. Therefore, `lion` and `bear` are incorrect. The braces containing the size are required to be after the type, making `ohMy` incorrect. The only one that is correct is `tiger`, making the correct answer Option B.

14. C. The `binarySearch()` method requires a sorted array in order to return a correct result. If the array is not sorted, the results of a binary search are undefined.

15. A. An `ArrayList` expands automatically when it is full. An array does not, making Option A the answer. The other three statements are true of both an array and an `ArrayList`.

16. C. This code creates a two-dimensional array of size 1×2. Lines `m1` and `m2` assign values to both elements in the outer array. Line `m3` attempts to reference the second element of the outer array. Since there is no such position, it throws an exception, and Option C is correct.

17. B. The code sorts before calling `binarySearch()`, so it meets the precondition for that method. The target string of `"Mac"` is the second element in the sorted array. Since array indices begin with zero, the second position is index 1, and Option B is correct.

18. A. A multi-dimensional array is created with multiple sets of size parameters. The first line should be `char[] ticTacToe = new char[3][3];`. Therefore, Option A is the answer.

19. B. The first line creates one object; the array itself. While there are four references to `null` in that array, none of those are objects. The second line creates one object and points one

of the array references to it. So far there are two objects: the array itself and one object it is referencing. The third line does the same, bringing up the object count to three. Therefore, Option B is correct.

20. B. As with a one-dimensional array, the braces must be after the type, making `alpha` and `beta` illegal declarations. For a multi-dimensional array, the braces are allowed to be before and/or after the variable name. They do not need to be in the same place. Therefore, the remaining three are correct, and Option B is correct.

21. B. Options A, C and D represent 3x3 2D arrays. Option B best represents the array in the code. It shows there are three different arrays of different lengths.

22. D. `names.length` is the number of elements in the array. The last valid index in the array is one less than `names.length`. In Java, arrays do not resize automatically. Therefore, the code throws an `ArrayIndexOutOfBoundsException`.

23. C. The code `days.size()` would be correct if this was an `ArrayList`. Since it is an array, `days.length` is the correct code. Therefore, the code does not compile, and Option C is the answer.

24. C. Since the braces in the declaration are before the variable names, the variable type `boolean[][][]` applies to both variables. Therefore, both `bools` and `moreBools` can reference a 3D array.

25. C. Calling `toString()` on an array doesn't output the contents of the array, making Option C correct. If you wanted Option A to be the answer, you'd have to call `Arrays.toString(strings)`.

26. B. Arrays begin with an index of 0. This array is a 3×3 array. Therefore. only indexes 0, 1. and 2 are valid. Line `r2` throws an `ArrayIndexOutOfBoundsException`. Therefore. Option B is correct.

27. D. Three dots in a row is a varargs parameter. While varargs is used like an array from within the method, it can only be used as a method parameter. This syntax is not allowed for a variable, making Option D the answer.

28. D. Line 6 assigns an `int` to a cell in a 2D array. This is fine. Line 7 casts to a general `Object[]`. This is dangerous, but legal. Why is it dangerous, you ask? That brings us to line 8. The compiler can't protect us from assigning a `String` to the `int[]` because the reference is more generic. Therefore, line 8 throws an `ArrayStoreException` because the type is incorrect, and Option D is correct. You couldn't have assigned an `int` on line 8 either because `obj[3]` is really an `int[]` behind the scenes and not an `int`.

29. C. The code sorts before calling `binarySearch`, so it meets the precondition for that method. The target string of `"RedHat"` is not found in the sorted array. If it was found, it would be between the second and third element. The rule is to take the negative index of where it would be inserted and subtract 1. It would need to be inserted as the third element. Since indexes are zero based, this is index 2. We take the negative, which is -2, and subtract 1, giving `-3`. Therefore, Option C is correct.

30. B. Array indexes begin with zero. `FirstName` is the name of the class, not an argument. Therefore, the first argument is `Wolfie`, and Option B is correct.

31. C. The name of the program is `Count` and there are two arguments. Therefore, the program outputs 2, and Option C is correct.

32. B. This class is called with two arguments. The first one (`seed`) is stored in the variable `one`. Then the array is sorted, meeting the precondition for binary search. Binary search returns `1` because `seed` is the second element in the sorted array, and Java uses zero-based indexes. Option B is correct.

33. D. Options A and B show the braces can be before or after the variable name and produce the same array. Option C specifies the same array the long way with two arrays of length 1. Option D is the answer because it is different than the others. It instead specifies an array of length 1 where that element is of length 2.

34. C. Arrays are indexed using numbers, not strings, making Options A and B incorrect. Since array indexes are zero based, Option C is the answer.

35. D. In Java, arrays are indexed starting with 0. While it is unusual for the loop to start with 1, this does not cause an error. What does cause an error is the loop ending at `data.length`, because the `<=` operator is used instead of the `<` operator. The last loop index is 6, not 7. On the last iteration of the loop, the code throws an `ArrayIndexOutOfBoundsException`. Therefore, Option D is correct.

36. C. Array indexes begin with zero. FirstName is the name of the class, not an argument. The first and only argument is Wolfie. There is not a second argument, so Option C is correct.

37. D. This code is correct. Line `r1` correctly creates a 2D array. The next three lines correctly assign a value to an array element. Line `r3` correctly outputs `3 in a row!`

38. D. Arrays expose a `length` variable. They do not have a `length()` method. Therefore, the code does not compile, and Option D is correct.

39. B. This one is tricky since the array braces are split up. This means that `bools` is a 3D array reference. The braces both before and after the variable name count. For `moreBools`, it is only a 2D array reference because there are only two pairs of braces next to the type. In other words, `boolean[][]` applies to both variables. Then `bools` gets another dimension from the braces right after the variable name. However, `moreBools` stays at 2D, making Option B correct.

40. B. Since no arguments are passed from the command line, this creates an empty array. Sorting an empty array is valid and results in an empty array. Therefore, Option B is correct.

41. D. Java requires having a sorted array before calling `binarySearch`. Since the array is not sorted, the result is undefined, and Option D is correct. It may happen that you get `1` as the result, but this behavior is not guaranteed. You need to know for the exam that this is undefined even if you happen to get the "right" answer.

42. B. Line 8 attempts to store a `String` in an array meant for an `int`. Line 8 does not compile, and Option B is correct.

43. A. This array has two elements, making `listing.length` output 2. While each array element does not have the same size, this does not matter because we are only looking at the first element. The first element has one. This makes the answer Option A.

44. C. `FirstName` is the name of the class, not an argument. There are no other arguments, so `names` is an empty array. Therefore, Option C is correct.

45. A. In Java, arrays are indexed starting with 0. While it is unusual for the loop to start with 1, this does not cause an error. It does cause the code to output six lines instead of seven since the loop doesn't cover the first array element. Therefore, Option A is correct.

46. B. The name of the program is `Count`, and there is only one argument because double quotes are used around the value. That argument is a `String` with three characters: `1`, a space, and `2`. Therefore, the program outputs `1`, and Option B is correct.

47. A. Java requires having a sorted array before calling `binarySearch()`. You do not have to call `Arrays.sort` to perform the sort though. This array happens to already be sorted, so it meets the precondition. The target string of `"Linux"` is the first element in the array. Since Java uses zero-based indexing, the answer is Option A.

48. A. From within a method, an array parameter and a varargs parameter are treated the same. From the caller, an array parameter is more restrictive. Both types can receive an array. However, only a varargs parameter is allowed to automatically turn individual parameters into an array. Therefore, statement I is correct and the answer is Option A.

49. B. All of the variables except `nums2b` point to a 4D array. Don't create a 4D array; it's confusing. The options show the braces can be before or after the variable in any combination. Option B is the answer because `nums2b` points to a 3D array. It only has three pairs of braces before the variable and none after. By comparison, `nums2a` has three pairs of braces before the variable and the fourth pair of braces after.

50. C. Binary search returns an `int` representing the index of a match or where a match would be. An `int` cannot be stored in a `String` variable. Therefore, the code does not compile and the answer is Option C.

Chapter 5: Using Loop Constructs

1. D. A `while` loop has a condition that returns a `boolean` that controls the loop. It appears at the beginning and is checked before entering the loop. Therefore, Option D is correct. A traditional `for` loop also has a `boolean` condition that is checked before entering the loop. However, it is best known for having a counter variable, making Option B incorrect. Option A is incorrect because the `boolean` condition on a `do-while` loop is at the end of the loop. Option C is incorrect because there is no condition as part of the loop construct.

2. B. A traditional `for` loop is best known for having a loop variable counting up or down as the loop progresses. Therefore, Option B is correct. Options A and D are incorrect because `do-while` and `while` loops are known for their `boolean` conditions. Option C is incorrect because the `for`-each loop iterates through without an index.

3. A. A `do-while` loop checks the loop condition after execution of the loop body. This ensures it always executes at least once, and Option A is correct. Option B is incorrect because there are loops you can write that do not ever enter the loop body, such as `for (int i=0;i<1;i++)`. Similarly, Option D is incorrect because a `while` loop can be written where the initial loop condition is `false`. Option C is incorrect because a `for`-each loop does not enter the loop body when iterating over an empty list.

4. C. While a traditional `for` loop often loops through an array, it uses an index to do so, making Option B incorrect. The `for`-each loop goes through each element, storing it in a variable. Option C is correct.

5. B. The `continue` keyword is used to end the loop iteration immediately and resume execution at the next iteration. Therefore, Option B is correct. Option A is incorrect because the `break` statement causes execution to proceed after the loop body. Options C and D are incorrect because these are not keywords in Java.

6. A. The `break` keyword is used to end the loop iteration immediately, skip any remaining executions of the loop, and resume execution immediately after the loop. Therefore, Option A is correct. Option B is incorrect because execution proceeds at the next execution of the current loop for `continue`. Options C and D are incorrect because these are not keywords in Java.

7. B. A traditional `for` loop is best known for having an initialization statement, condition statement, and update statement. Option B is correct.

8. C. With a traditional `for` loop, you control the order in which indexes are visited in code. This means you can loop through an array in ascending or descending order, and Option C is correct.

9. A. With a `for`-each loop, the loop order is determined for you. With an `array`, this means starting with index 0, and Option A is correct. A traditional `for` loop allows you to control the order and iterate in either order.

10. A. A `do-while` loop has a condition that returns a `boolean` at the end of the loop. Therefore, Option A is correct. Option D is incorrect because a `while` loop has this condition at the beginning of the loop. A traditional `for` loop is best known for having a loop variable, making Option B incorrect. Option C is incorrect because there is no condition as part of the loop construct.

11. B. A `while` loop requires a `boolean` condition. While `singer` is a variable, it is not a `boolean`. Therefore, the code does not compile, and Option B is correct.

12. B. This is a correct loop to go through an `ArrayList` or `List` starting from the end. It starts with the last index in the list and goes to the first index in the list. Option B is correct.

13. A. The first time through the loop, the index is 0 and `glass,` is output. The `break` statement then skips all remaining executions on the loop and the `main()` method ends. If there was no `break` keyword, this would be an infinite loop because there's no incrementor.

14. A. Immediately after `letters` is initialized, the loop condition is checked. The variable `letters` is of length 0, which is not equal to 2 so the loop is entered. In the loop body, `letters` becomes length 1 with contents `"a"`. The loop index is checked again and now 1 is not equal to 2. The loop is entered and `letters` becomes length 2 and contains `"aa"`. Then the loop index is checked again. Since the length is now 2, the loop is completed and `aa` is output. Option A is correct.

15. D. There are three arguments passed to the program. This means that `i` is 3 on the first iteration of the loop. The program prints `args`. Then `i` is incremented to 4. Which is also greater than or equal to 0. Since `i` never gets smaller, this code produces an infinite loop and the answer is Option D.

16. B. Since `count` is a class variable that isn't specifically initialized, it defaults to 0. On the first iteration of the loop, `"Washington"`, is 11 characters and `count` is set to 1. The `if` statement's body is not run. The loop then proceeds to the next iteration. This time, the post-increment operator uses index 1 before setting `count` to 2. `"Monroe"` is checked, which is only 6 characters. The `break` statement sends the execution to after the loop and 2 is output. Option B is correct.

17. C. At first this code appears to be an infinite loop. However, the `count` variable is declared inside the loop. It is not in scope after the loop where it is referenced by the `println()`. Therefore, the code does not compile, and Option C is correct.

18. D. A `for` loop is allowed to have all three segments left blank. In fact, `for(;;) {}` is an infinite loop.

19. C. It is not possible to create an infinite loop using a `for`-each because it simply loops through an array or `ArrayList`. The other types allow infinite loops, such as, for example, `do { } while(true)`, `for(;;)` and `while(true)`. Therefore, Option C is correct. And yes, we know it is possible to create an infinite loop with `for`-each by creating your own custom `Iterable`. This isn't on the OCA or OCP exam though. If you think the answer is Option D, this is a great reminder of what not to read into on the real exam!

20. A. This is a correct loop to go through an `ArrayList` or `List` starting from the beginning. It starts with index 0 and goes to the last index in the list. Option A is correct.

21. D. Braces are optional around loops if there is only one statement. Parentheses are not allowed to surround a loop body though, so the code does not compile, and Option D is correct.

22. B. The `for`-each loop uses a variable and colon as the syntax, making Option B correct.

23. C. In this figure, we want to end the inner loop and resume execution at the `letters` label. This means we only want to break out of the inner loop. A `break` statement does just that. It ends the current loop and resumes execution immediately after the loop, making `break;` a correct answer. The `break numbers;` statement explicitly says which loop to end, which does the same thing, making it correct as well. By contrast, `break letters;` ends the outer loop, causing the code only to run the `println()` once. Therefore, two statements correctly match the diagram, and Option C is correct.

24. B. In this figure, we want to end the inner loop and resume execution at the `letters` label. The `continue letters;` statement does that. The other two statements resume execution at the inner loop. Therefore, only the second statement correctly matches the diagram, and Option B is correct.

25. C. A `while` loop checks the `boolean` condition before entering the loop. In this code, that condition is false, so the loop body is never run. No output is produced, and Option C is correct.

26. C. A `for`-each loop is allowed to be used with arrays and `ArrayList` objects. `StringBuilder` is not an allowed type for this loop, so Option C is the answer.

27. B. This is a correct do-`while` loop. On the first iteration of the loop, the `if` statement executes and prints `inflate-`. Then the loop condition is checked. The variable `balloonInflated` is `true`, so the loop condition is `false` and the loop completes.

28. D. Immediately after `letters` is initialized, the loop condition is checked. The variable `letters` is of length 0, which is not equal to 3, so the loop is entered. In the loop body, `letters` becomes length 2 and contains `"ab"`. The loop index is checked again and now 2 is not equal to 3. The loop is entered and `letters` becomes length 4 with contents `"abab"`. Then the loop index is checked again. Since the length 4 is not equal to 3, the loop body is entered again. This repeats for 6, 8, 10, etc. The loop never ends, and Option D is correct.

29. B. In a `for` loop, the segments are an initialization expression, a `boolean` conditional, and an update statement in that order. Therefore, Option B is correct.

30. B. On the first iteration through the outer loop, `chars` becomes 1 element. The inner loop is run once and `count` becomes 9. On the second iteration through the outer loop, `chars` becomes 2 elements. The inner loop runs twice so `count` becomes `7`. On the third iteration through the outer loop, `chars` becomes 3 elements. The inner loop runs three times so `count` becomes `4`. On the fourth iteration through the outer loop, `chars` becomes 4 elements. The inner loop runs four times so `count` becomes `0`. Then both loops end. Therefore, Option B is correct.

31. A. On the first iteration of the outer loop, `i` starts out at `10`. The inner loop sees that `10 > 3` and subtracts 3, making the `7` the new value of `i`. Since `7 > 3`, we subtract 3 again, making `i` set to `4`. Yet again `4 > 3`, so `i` becomes `1`. Then `k` is finally incremented to `1`. The outer loop decrements `i i`, making it `0`. The `boolean` condition sees that `0` is not greater than `0`. The outer loop ends and `1` is printed out. Therefore, Option A is correct.

32. D. Options A and C do not compile as they do not use the correct syntax for a `for`-each loop. The `for`-each loop is only able to go through an array in ascending order. It is not able to control the order, making Option C incorrect. Therefore, Option D is the answer.

33. C. Since there are no brackets around the `for` statement, the loop body is only one line. The `break` statement is not in the loop. Since `break` cannot be used at the top level of a method, the code does not compile, and Option C is correct.

34. C. Multiple update expressions are separated with a comma rather than a semicolon. Tricky, we know. But it is an important distinction. This makes Option C correct.

35. D. There are three arguments passed to the program. This means that `i` is 3 on the first iteration of the loop. The program attempts to print `args[3]`. Since indexes are zero based in Java, it throws an `ArrayIndexOutOfBoundsException`.

36. B. The first time the loop condition is checked, the variable `tie` is `null`. The loop body executes, setting `tie`. Despite the indention, there are no brackets surrounding the loop body so the `print` does not run yet. Then the loop condition is checked and `tie` is not `null`. The `print` runs after the loop, printing out `shoelace` once, making Option B correct.

37. C. The code compiles as is. However, we aren't asked about whether the code compiles as is. Line 27 refers to a loop label. While the label is still present, it no longer points to a loop. This causes the code to not compile, and Option C is correct.

38. C. The `continue` statement is useless here since there is no code later in the loop to skip. The `continue` statement merely resumes execution at the next iteration of the loop, which is what would happen if the if-then statement was empty. Therefore, `count` increments for each element of the array. The code outputs `4`, and Option C is correct.

39. C. A `do-while` loop requires a `boolean` condition. The `builder` variable is a `StringBuilder` and not a `boolean`. The code does not compile, and Option C is correct.

40. A. At first this code appears to be an infinite loop. However, there is a `break` statement. On line 6, `count` is set to `0`. On line 9, it is changed to `1`. Then the condition on line 10 runs. `count` is less than 2 so the inner loop continues. Then `count` is set to 2 on the next iteration of the inner loop. The loop condition on line 10 runs again and this time is false. The inner loop is completed. Then line 11 of the outer loop runs and sends execution to after the loop on line 13. At this point `count` is still 2, so Option A is correct.

41. C. Option A breaks out of the inner loop, but the outer loop is still infinite. Option B has the same problem. Option C is correct because it breaks out of both loops.

42. B. This code is correct. It initializes two variables and uses both variables in the condition check and the update statements. Since it checks the size of both arrays correctly, it prints the first two sets of elements, and Option B is correct.

43. B. Looping through the same list multiple times is allowed. The outer loop executes twice. The inner loop executes twice for each of those iterations of the outer loop. Therefore, the inner loop executes four times, and Option B is correct.

44. B. The initializer, which is `alpha`, runs first. Then Java checks the condition, which is `beta`, to see if loop execution should start. Since `beta` returns `false`, the loop is never entered, and Option B is correct.

45. B. The initializer, which is `alpha`, runs first. Then Java checks the condition, which is `beta`, to see if loop execution should start. Then the loop body, which is `delta`, runs. After the loop execution, the updater, which is `gamma`, runs. Then the loop condition, which is `beta`, is checked again. Therefore, Option B is correct.

46. C. Option A goes through five indexes on the iterations: 0, 1, 2, 3 and 4. Option B also goes through five indexes: 1, 2, 3, 4 and 5. Option D goes through five iterations as well, from 0 to 4. However, Option C goes through six iterations since the loop condition is at the end of the loop. Therefore it is not like the others, and Option C is the answer.

47. D. The first time the loop condition is checked, the variable `tie` is `null`. However, the loop body is empty due to the semicolon right after the condition. This means the loop condition keeps running with no opportunity for `tie` to be set. Therefore, this is an infinite loop, and Option D is correct.

48. C. Remember to look for basic errors before wasting time tracking the flow. In this case, the label of the loop is trying to use the keyword `for`. This is not allowed, so the code does not compile. If the label was valid, Option A would be correct.

49. D. On the first iteration of the loop, the `if` statement executes printing `inflate-`. Then the loop condition is checked. The variable `baloonInflated` is `true`, so the loop condition is `true` and the loop continues. The `if` statement no longer runs, but the variable never changes state again, so the loop doesn't end.

50. B. In a `for` loop, the type is only allowed to be specified once. A comma separates multiple variables since they are part of the same statement. Therefore, Option B is correct.

Chapter 6: Working with Methods and Encapsulation

1. C. The `protected` modifier allows access by subclasses and members within the same package, while the package-private modifier allows access only to members in the same package. Therefore, the `protected` access modifier allows access to everything the package-private access modifier, plus subclasses, making Option C the correct answer. Options A, B, and D are incorrect because the first term is a more restrictive access modifier than the second term.

2. B. The `super()` statement is used to call a constructor in a parent class, while the `this()` statement is used to call a constructor in the same class, making Option B correct and Option A incorrect. Options C and D are incorrect because they are not constructors.

3. D. The `sell()` method does not compile because it does not return a value if both of the if-then statements' conditional expressions evaluate to `false`. While logically, it is true that `price` is either less than `10` or greater than or equal to `10`, the compiler does not know that. It just knows that if both if-then statements evaluate to `false`, then it does not have a return value, therefore it does not compile.

4. D. The three overloaded versions of `nested()` compile without issue, since each method takes a different set of input arguments, making Options B and C incorrect. The code does not compile, though, due to the first line of the `main()` method, making Option A incorrect. The no-argument version of the `nested()` method does not return a value, and trying to output a `void` return type in the `print()` method throws an exception at runtime.

5. B. Java uses pass-by-value to copy primitives and references of objects into a method. That means changes to the primitive value or reference in the method are not carried to the calling method. That said, the data within an object can change, just not the original reference itself. Therefore, Option B is the correct answer, and Options C and D are incorrect. Option A is not a real term.

6. C. Option A is incorrect because the getter should return a value. Option B is incorrect because the setter should take a value. Option D is incorrect because the setter should start with `set` and should not return a value. Option C is a correct setter declaration because it takes a value, uses the `void` return type, and uses the correct naming convention.

7. B. Options A, C, and D are true statements about calling `this()` inside a constructor. Option B is incorrect because a constructor can only call `this()` or `super()` on the first line of the constructor, but never both in the same constructor. If both constructors were allowed to be called, there would be two separate calls to `super()`, leading to duplicate initialization of parent constructors, since the other constructor referenced by `this()` would also call `super()` (or be chained to one that eventually calls `super()`).

8. B. Option A is incorrect because the `public` access modifier starts with a lowercase letter. Options C and D are incorrect because the return types, `void` and `String`, are incompatible with the method body that returns an integer value of `10`. Option B is correct and has package-private access. It also uses a return type of `Long` that the integer value of `10` can be easily assigned to without an explicit cast.

9. C. The only variables always available to all instances of the class are those declared `static`; therefore, Option C is the correct answer. Option A may seem correct, but `public` variables are only available if a reference to the object is maintained among all instances. Option B is incorrect because there is no `local` keyword in Java. Option D is also incorrect because a `private` instance variable is only accessible within the instance that created it.

10. A. First off, all of the lines compile but they produce various different results. Remember that the default initialization of a `boolean` instance variable is `false`, making `outside false` at line `p1`. Therefore, `this(4)` will cause `rope` to be set to `5`, while `this(5)` will cause `rope` to be set to `6`. Since `5` is the number we are looking for, Option A is correct, and Option C is incorrect. Option B is incorrect. While the statement does create a new instance of `Jump`, with `rope` having a value of `5`, that instance is nested and the value of `rope` does not affect the surrounding `instance` of `Jump` that the constructor was called in. Option D is also incorrect. The value assigned to `rope` is `4`, not the target `5`.

11. B. Options A, C, and D are true statements. In particular, Option C allows us to write the `equals()` methods between two objects that compare `private` attributes of the class. Option D is true because `protected` access also provides package-private access. Option B is false. Package-private attributes are only visible if the two classes are in the same package, regardless of whether one extends the other.

12. D. The class data, `stuff`, is declared `public`, allowing any class to modify the `stuff` variable and making the implementation inherently unsafe for encapsulation. Therefore, there are no values that can be placed in the two blanks to ensure the class properly encapsulates its data, making Option D correct. Note that if `stuff` was declared `private`, Options A, B, and C would all be correct. Encapsulation does not require JavaBean syntax, just that the internal attributes are protected from outside access, which all of these sets of values do achieve.

13. C. Option A is incorrect because Java only inserts a no-argument constructor if there are no other constructors in the class. Option B is incorrect because the parent can have a default no-argument constructor, which is inserted by the compiler and accessible in the child class. Finally, Option D is incorrect. A class that contains two no-argument constructors will not compile because they would have the same signature. Finally, Option C is correct. If a class extends a parent class that does not include a no-argument constructor, the default no-argument constructor cannot be automatically inserted into the child class by the compiler. Instead, the developer must explicitly declare at least one constructor and explicitly define how the call to the parent constructor is made.

14. A. A method may contain at most one varargs parameter, and it must appear as the last argument in the list. For this reason, Option A is correct, and Options B, C, and D are incorrect.

15. C. To solve this problem, it helps to remember that Java is a pass-by-value language in which copies of primitives and object references are sent to methods. This also means that an object's data can be modified within a method and shared with the caller, but not the reference to the object. Any changes to the object's reference within the method are not carried over to the caller. In the `slalom()` method, the `Ski` object is updated with an `age` value of `18`. Although, the last line of the `slalom()` method changes the variable value to `null`, it does not affect the `mySkier` object or reference in the `main()` method. Therefore, the `mySkier` object is not `null` and the `age` variable is set to `18`, making Options A and D incorrect. Next, the `name` variable is reassigned to the `Wendy` object, but this does not change the reference in the `main()` method, so `myName` remains `Rosie`. Finally, the `speed` array is assigned a new object and updated. Since the array is updated after the reference is reassigned, it does not affect the `mySpeed` array in the `main()` method. The result is that `mySpeed` continues to have a single element with the default `int` value of `0`. For these reasons, Option B is incorrect, and Option C is correct.

16. B. Options A and D would not allow the class to compile because two methods in the class cannot have the same name and arguments, but a different return value. Option C would allow the class to compile, but it is not a valid overloaded form of our `findAverage()` method since it uses a different method name. Option B is a valid overloaded version of the `findAverage()` method, since the name is the same but the argument list differs.

17. D. Implementing encapsulation prevents internal attributes of a class from being modified directly, so Option C is a true statement. By preventing access to internal attributes, we can also maintain class data integrity between elements, making Option B a true statement. Option A is also a true statement about encapsulation, since well-encapsulated classes are often easier to use. Option D is an incorrect statement. Encapsulation makes no guarantees about performance and concurrency.

18. A. Option B is incorrect because `String` values are immutable and cannot be modified. Options C and D are also incorrect since variables are passed by value, not reference, in Java. Option A is the correct answer. The contents of an array can be modified when passed to a method, since a copy of the reference to the object is passed. For example, the method can change the first element of a non-empty array.

19. B. Option A is not a valid syntax in Java. Option C would be correct if there was a `static` import, but the question specifically says there are not any. Option D is almost correct, since it is a way to call the method, but the question asks for the best way to call the method. In that regard, Option B is the best way to call the method, since we are given that two classes are in the same package, therefore the package name would not be required.

20. D. Options A and B are incorrect because a method with a non-`void` return type requires that the method return a value using the `return` statement. Option C is also incorrect since a method with a `void` return type can still call the `return` command with no values and exit the method. Therefore, Option D is the correct answer.

21. C. The `finish()` method modifies two variables that are marked `final`, `score` and `result`. The `score` variable is modified by the post-increment `++` operator, while the `result` variable is modified by the compound addition `+=` operator. Removing both `final` modifiers allows the code to compile. For this reason, Option C is the correct answer.

22. D. The `super()` statement is used to call a constructor in the parent class, while `super` is used to reference a member of the parent class. The `this()` statement is used to call a constructor in the current class, while `this` is used to reference a member of the current class. For these reasons, Option D is the correct answer.

23. B. The method signature has package-private, or default, access; therefore, it is accessible to classes in the same package, making Option B the correct answer.

24. A. The access modifier of `strength` is `protected`, meaning subclasses and classes within the same package can modify it. Changing the value to `private` would improve encapsulation by making the `Protect` class the only one capable of directly modifying it. For these reasons, the first statement is correct. Alternatively, the second and third statements do not improve the encapsulation of the class. While having getters and setters for `private` variables is helpful, they are not required. Encapsulation is about protecting the data elements. With this in mind, it is clear the `material` variable is already protected. Therefore, Option A is the correct answer.

25. A. Option A is correct since method names may include the underscore _ character as well as the dollar $ symbol. Note that there is no rule that requires a method start with a lowercase character; it is just a practice adopted by the community. Option B is incorrect because the hyphen - character may not be part of a method name. Option C is incorrect since `new` is a reserved word in Java. Finally, Option D is incorrect. A method name must start with a letter, the dollar $ symbol, or an underscore _ character.

26. D. The code does not compile, regardless of what is inserted into the line because the method signature is invalid. The return type, `int`, should go before the method name and after any access, `final`, or `static` modifiers. Therefore, Option D is the correct answer. If the method was fixed, by swapping the order of `int` and `static` in the method declaration, then Option C would be the correct answer. Options A and B are still incorrect, though, since each uses a return type that cannot be implicitly converted to `int`.

27. B. Java uses pass-by-value, so changes made to primitive values and object references passed to a method are not reflected in the calling method. For this reason, Options A and C are incorrect statements. Option D is also an invalid statement because it is a special case of Option A. Finally, Option B is the correct answer. Changes to the data within an object are visible to the calling method since the object that the copied reference points to is the same.

28. C. The code contains a compilation problem in regard to the `contents` instance variable. The `contents` instance variable is marked `final`, but there is a `setContents()` instance method that can change the value of the variable. Since these two are incompatible, the code does not compile, and Option C is correct. If the `final` modifier was removed from the `contents` variable declaration, then the expected output would be of the form shown in Option A.

29. A. JavaBean methods use the prefixes `get`, `set`, and `is` for `boolean` values, making Option A the correct choice.

30. C. Option A is incorrect because the keywords `static` and `import` are reversed. The `Closet` class uses the method `getClothes()` without a reference to the class name `Store`, therefore a `static` import is required. For this reason, Option B is incorrect since it is missing the `static` keyword. Option D is also incorrect since `static` imports are used with members of the class, not a class name. Finally, Option C is the correct answer since it properly imports the method into the class using a `static` import.

31. D. In Java, the lack of an access modifier indicates that the member is package-private, therefore Option D is correct. Note that the `default` keyword is used for interfaces and `switch` statements, and is not an access modifier.

32. B. The code does not compile, so Option A is incorrect. The class contains two constructors and one method. The first method, `Stars()`, looks a lot like a no-argument constructor, but since it has a return value of `void`, it is a method, not a constructor. Since only constructors can call `super()`, the code does not compile due to this line. The only constructor in this class, which takes an `int` value as input, performs a pointless assignment, assigning a variable to itself. While this assignment has no effect, it does not prevent the code from compiling. Finally, the `main()` method compiles without issue since we just inserted the full package name into the class constructor call. This is how a class that does not use an `import` statement could call the constructor. Since the method is in the same class, and therefore the same package, it is redundant to include the package name but not disallowed. Because only one line causes the class to fail to compile, Option B is correct.

33. A. An instance method or constructor has access to all `static` variables, making Option A correct. On the other hand, `static` methods and `static` initializers cannot reference instance variables since they are defined across all instances, making Options B and C incorrect. Note that they can access instance variables if they are passed a reference to a specific instance, but not in the general case. Finally, Option D is incorrect because `static final` variables must be set when they are declared or in a `static` initialization block.

34. B. The method `calculateDistance()` requires a return type that can be easily converted to a `short` value. Options A, C, and D are incorrect because they each use a larger data type that requires an explicit cast. Option D also does not compile because the `Short` constructor requires an explicit cast to convert the value of `4`, which is assumed to be an `int`, to a `short`, as shown in `new Short((short)4)`. Option B is the correct answer since a `byte` value can be easily promoted to `short` and returned by the method.

35. C. Overloaded methods have the same name but a different list of parameters, making the first and third statements true. The second statement is false, since overloaded methods can have the same or different return types. Therefore, Option C is the correct answer.

36. C. The declaration of `monday` does not compile, because the value of a `static final` variable must be set when it is declared or in a `static` initialization block. The declaration of `tuesday` is fine and compiles without issue. The declaration of `wednesday` does not compile because there is no data type for the variable. Finally, the declaration of `thursday` does not compile because the `final` modifier cannot appear before the access modifier. For these reasons, Option C is the correct answer.

37. D. The `Puppy` class does not declare a constructor, so the default no-argument constructor is automatically inserted by the compiler. What looks like a constructor in the class is actually a method that has a return type of `void`. Therefore, the line in the `main()` method to create the `new Puppy(2)` object does not compile, since there is no constructor capable of taking an `int` value, making Option D the correct answer.

38. A. The `public` modifier allows access to members in the same class, package, subclass, or even classes in other packages, while the `private` modifier allows access only to members in the same class. Therefore, the `public` access modifier allows access to everything the `private` access modifier does, and more, making Option A the correct answer. Options B, C, and D are incorrect because the first term is a more restrictive access modifier than the second term.

39. A. The code compiles without issue, so Option D is incorrect. The key here is that Java uses pass by value to send object references to methods. Since the `Phone` reference `p` was reassigned in the first line of the `sendHome()` method, any changes to the `p` reference were made to a new object. In other words, no changes in the `sendHome()` method affected the object that was passed in. Therefore, the value of `size` was the same before and after the method call, making the output `3` and Option A the correct answer.

40. B. Options A and D are equivalent and would allow the code to compile. They both are proper ways to access a `static` method from within an instance method. Option B is the correct answer. The class would not compile because `this.Drink` has no meaning to the compiler. Finally, Option C would still allow the code to compile, even though it is considered a poor coding practice. While `static` members should be accessed in a `static` way, it is not required.

41. C. The method signature requires one `int` value, followed by exactly one `String`, followed by `String` varargs, which can be an array of `String` values or zero or more individual `String` values. Only Option C conforms to these requirements, making it the correct answer.

42. D. Option A is a statement about `final static` variables, not all `static` variables. Option B only applies to `static` variables marked `private`, not `final`. Option C is false because `static` imports can be used to reference both variables and methods. Option D is the correct answer because a `static` variable is accessible to all instances of the class.

43. A. Option A is the correct answer because the first line of a constructor could be `this()` or `super()`, making it an untrue statement. Option B is a true statement because the compiler will insert the default no-argument constructor if one is not defined. Option C is also a true statement, since zero or more arguments may be passed to the parent constructor, if the parent class defines such constructors. Option D is also true. The value of a `final` instance variable should be set when it is declared, in an initialization block, or in a constructor.

44. D. The last `static` initialization block accesses `height`, which is an instance variable, not a `static` variable. Therefore, the code will not compile no matter how many `final` modifiers are removed, making Option D the correct answer. Note that if the line `height = 4;` was removed, then no `final` modifiers would need to be removed to make the class compile.

45. D. Since a constructor call is not the first line of the `RainForest()` constructor, the compiler inserts the no-argument `super()` call. Since the parent class, `Forest`, does not define a no-argument `super()` constructor, the `RainForest()` constructor does not compile, and Option D is correct.

46. A. The code compiles without issue, so Option D is incorrect. In the `main()` method, the value 2 is first cast to a `byte`. It is then increased by one using the addition `+` operator. The addition `+` operator automatically promotes all `byte` and `short` values to `int`. Therefore, the value passed to the `choose()` in the `main()` method is an `int`. The `choose(int)` method is called, returning `5` and making Option A the correct answer. Note that without the addition operation in the `main()` method, `byte` would have been used as the parameter to the `choose()` method, causing the `choose(short)` to be selected as the next closest type and outputting 2, making Option B the correct answer.

47. C. The variable `startTime` can be automatically converted to `Integer` by the compiler, but `Integer` is not a subclass of `Long`. Therefore, the code does not compile due the wrong variable type being passed to the `getScore()` method on line `m2`, and Option C is correct.

48. A. Java methods must start with a letter, the dollar `$` symbol, or underscore `_` character. For these reasons, Options B and D are incorrect, and Option A is correct. Option C is incorrect. The hashtag (`#`) symbol cannot be included in a method name.

49. B. The `protected` modifier allows access by any subclass or class that is in the same package, therefore Option B is the correct answer.

50. D. A `static` import is used to import `static` members of another class. In this case, the `withdrawal()` and `deposit()` methods in the `Bank` class are not marked `static`. They require an instance of `Bank` to be used and cannot be imported as `static` methods. Therefore, Option D is correct. If the two methods in the `Bank` class were marked `static`, then Option A would be the correct answer since wildcards can be used with `static` imports to import more than one method. Option B reverses the keywords `static` and `import`, while Option C incorrectly imports a class, which cannot be imported via a `static` import.

Chapter 7: Working with Inheritance

1. C. The code does not compile, so Option A is incorrect. This code does not compile for two reasons. First, the `name` variable is marked `private` in the `Cinema` class, which means it cannot be accessed directly in the `Movie` class. Next, the `Movie` class defines a constructor that is missing an explicit `super()` statement. Since `Cinema` does not include a no-argument constructor, the no-argument `super()` cannot be inserted automatically by the compiler without a compilation error. For these two reasons, the code does not compile, and Option C is the correct answer.

2. D. All abstract interface methods are implicitly `public`, making Option D the correct answer. Option A is incorrect because `protected` conflicts with the implicit `public` modifier. Since `static` methods must have a body and `abstract` methods cannot have a body, Option B is incorrect. Finally, Option C is incorrect. A method, whether it be in an interface or a class, cannot be declared both `final` and `abstract`, as doing so would prevent it from ever being implemented.

3. C. A class cannot contain two methods with the same method signature, even if one is `static` and the other is not. Therefore, the code does not compile because the two declarations of `playMusic()` conflict with one another, making Option C the correct answer.

4. A. Inheritance is often about improving code reusability, by allowing subclasses to inherit commonly used attributes and methods from parent classes, making Option A the correct answer. Option B is incorrect. Inheritance can lead to either simpler or more complex code, depending on how well it is structured. Option C is also incorrect. While all objects inherit methods from `java.lang.Object`, this does not apply to primitives. Finally, Option D is incorrect because methods that reference themselves are not a facet of inheritance.

5. A. Recall that `this` refers to an instance of the current class. Therefore, any superclass of `Canine` can be used as a return type of the method, including `Canine` itself, making Option C an incorrect answer. Option B is also incorrect because `Canine` implements the `Pet` interface. An instance of a class can be assigned to any interface reference that it inherits. Option D is incorrect because `Object` is the superclass of instances in Java. Finally, Option A is the correct answer. `Canine` cannot be returned as an instance of `Class` because it does not inherit `Class`.

6. B. The key here is understanding which of these features of Java allow one developer to build their application around another developer's code, even if that code is not ready yet. For this problem, an interface is the best choice. If the two teams agree on a common interface, one developer can write code that uses the interface, while another developer writes code that implements the interface. Assuming neither team changes the interface, the code can be easily integrated once both teams are done. For these reasons, Option B is the correct answer.

7. B. The `drive()` method in the `Car` class does not override the version in the `Automobile` class since the method is not visible to the `Car` class. Therefore, the `final` attribute in the `Automobile` class does not prevent the `Car` class from implementing a method with the same signature. The `drive()` method in the `ElectricCar` class is a valid override of the method in the `Car` class, with the access modifier expanding in the subclass. For these reasons, the code compiles, and Option D is incorrect. In the `main()` method, the object created is an `ElectricCar`, even if it is assigned to a `Car` reference. Due to polymorphism, the method from the `ElectricCar` will be invoked, making Option B the correct answer.

8. D. While Java does not allow a class to extend more than one class, it does allow a class to implement any number of interfaces. Multiple inheritance is, therefore, only allowed via interfaces, making Option D the correct answer.

9. C. There are three problems with this method override. First, the `watch()` method is marked `final` in the `Television` class. The `final` modifier would have to be removed from the method definition in the `Television` class in order for the method to compile in the `LCD` class. Second, the return types `void` and `Object` are not covariant. One of them would have to be changed for the override to be compatible. Finally, the access modifier in the child class must be the same or broader than in the parent class. Since package-private is narrower than `protected`, the code will not compile. For these reasons, Option C is the correct answer.

10. C. First off, the return types of an overridden method must be covariant. Next, it is true that the access modifier must be the same or broader in the child method. Using a narrower access modifier in the child class would not allow the code to compile. Overridden methods must not throw any new or broader checked exceptions than the method in the superclass. For these reasons, Options A, B, and D are true statements. Option C is the false statement. An overridden method is not required to throw a checked exception defined in the parent class.

11. C. The `process()` method is declared `final` in the `Computer` class. The `Laptop` class then attempts to override this method, resulting in a compilation error, making Option C the correct answer.

12. A. The code compiles without issue, so Option D is incorrect. The rule for overriding a method with exceptions is that the subclass cannot throw any new or broader checked exceptions. Since `FileNotFoundException` is a subclass of `IOException`, it is considered a narrower exception, and therefore the overridden method is allowed. Due to polymorphism, the overridden version of the method in `HighSchool` is used, regardless of the reference type, and `2` is printed, making Option A the correct answer. Note that the version of the method that takes the varargs is not used in this application.

13. B. Interface methods are implicitly `public`, making Option A and C incorrect. Interface methods can also not be declared `final`, whether they are `static`, `default`, or `abstract` methods, making Option D incorrect. Option B is the correct answer because an interface method can be declared `static`.

14. C. Having one class implement two interfaces that both define the same `default` method signature leads to a compiler error, unless the class overrides the `default` method. In this case, the `Sprint` class does override the `walk()` method correctly, therefore the code compiles without issue, and Option C is correct.

15. B. Interfaces can extend other interfaces, making Option A incorrect. On the other hand, an interface cannot implement another interface, making Option B the correct answer. A class can implement any number of interfaces, making Option C incorrect. Finally, a class can extend another class, making Option D incorrect.

16. D. The code does not compile because `super.height` is not visible in the `Rocket` class, making Option D the correct answer. Even though the `Rocket` class defines a height value, the `super` keyword looks for an inherited version. Since there are none, the code does not compile. Note that `super.getWeight()` returns `3` from the variable in the parent class, as polymorphism and overriding does not apply to instance variables.

17. D. An abstract class can contain both abstract and concrete methods, while an interface can only contain abstract methods. With Java 8, interfaces can now have `static` and `default` methods, but the question specifically excludes them, making Option D the correct answer. Note that concrete classes cannot contain any abstract methods.

18. C. The code does not compile, so Option D is incorrect. The `IsoscelesRightTriangle` class is abstract; therefore, it cannot be instantiated on line `g3`. Only concrete classes can be instantiated, so the code does not compile, and Option C is the correct answer. The rest of the lines of code compile without issue. A concrete class can extend an abstract class, and an abstract class can extend a concrete class. Also, note that the override of `getDescription()` has a widening access modifier, which is fine per the rules of overriding methods.

19. D. The `play()` method is overridden in `Saxophone` for both `Horn` and `Woodwind`, so the return type must be covariant with both. Unfortunately, the inherited methods must also be compatible with each other. Since `Integer` is not a subclass of `Short`, and vice versa, there is no subclass that can be used to fill in the blank that would allow the code to compile. In other words, the `Saxophone` class cannot compile regardless of its implementation of `play()`, making Option D the correct answer.

20. C. A class can implement an interface, not extend it. Alternatively, a class extends an `abstract` class. Therefore, Option C is the correct answer.

21. A. The code compiles and runs without issue, making Options C and D incorrect. Although `super.material` and `this.material` are poor choices in accessing `static` variables, they are permitted. Since `super` is used to access the variable in `getMaterial()`, the value `papyrus` is returned, making Option A the correct answer. Also, note that the constructor `Book(String)` is not used in the `Encyclopedia` class.

22. B. Options A and C are both true statements. Either the `unknownBunny` reference variable is the same as the object type or it can be explicitly cast to the `Bunny` object type, therefore giving it access to all its members. This is the key distinction between reference types and object types. Assigning a new reference does not change the underlying object. Option D is also a true statement since any superclass that `Bunny` extends or interface it implements could be used as the data type for `unknownBunny`. Option B is the false statement and the correct answer. An object can be assigned to a reference variable type that it inherits, such as `Object unknownBunny = new Bunny()`.

23. D. An abstract method cannot include the `final` or `private` method. If a class contained either of these modifiers, then no concrete subclass would ever be able to override them with an implementation. For these reasons, Options A and B are incorrect. Option C is also incorrect because the `default` keyword applies to concrete interface methods, not abstract methods. Finally, Option D is correct. The `protected`, package-private, and `public` access modifiers can each be applied to abstract methods.

24. D. The declaration of `Sphere` compiles without issue, so Option C is incorrect. The `Mars` class declaration is invalid because `Mars` cannot extend `Sphere`, an interface, nor can `Mars` implement `Planet`, a class. In other words, they are reversed. Since the code does not compile, Option D is the correct answer. Note that if `Sphere` and `Planet` were swapped in the `Mars` class definition, the code would compile and the output would be `Mars`, making Option A the correct answer.

25. B. A reference to a class can be implicitly assigned to a superclass reference without an explicit class, making Option B the correct answer. Assigning a reference to a subclass, though, requires an explicit cast, making Option A incorrect. Option C is also incorrect because an interface does not inherit from a class. A reference to an interface requires an explicit cast to be assigned to a reference of any class, even one that implements the interface. An interface reference requires an explicit cast to be assigned to a class reference. Finally, Option D is incorrect. An explicit cast is not required to assign a reference to a class that implements an interface to a reference of the interface.

26. B. Interface variables are implicitly `public`, `static`, and `final`. Variables cannot be declared as `abstract` in interfaces, nor in classes.

27. C. The class is loaded first, with the `static` initialization block called and `1` is outputted first. When the `BlueCar` is created in the `main()` method, the superclass initialization happens first. The instance initialization blocks are executed before the constructor, so `32` is outputted next. Finally, the class is loaded with the instance initialization blocks again being called before the constructor, outputting `45`. The result is that `13245` is printed, making Option C the correct answer.

28. C. Overloaded methods share the same name but a different list of parameters and an optionally different return type, while overridden methods share the exact same name, list of parameters, and return type. For both of these, the one commonality is that they share the same method name, making Option C the correct answer.

29. A. Although the casting is a bit much, the object in question is a `SoccerBall`. Since `SoccerBall` extends `Ball` and implements `Equipment`, it can be explicitly cast to any of those types, so no compilation error occurs. At runtime, the object is passed around and, due to polymorphism, can be read using any of those references since the underlying object is a `SoccerBall`. In other words, casting it to a different reference variable does not modify the object or cause it to lose its underlying `SoccerBall` information. Therefore, the code compiles without issue, and Option A is correct.

30. C. Both of these descriptions refer to variable and `static` method hiding, respectively, making Option C correct. Only instance methods can be overridden, making Options A and B incorrect. Option D is also incorrect because *replacing* is not a real term in this context.

31. B. The code does not compile, so Option D is incorrect. The issue here is that the override of `getEqualSides()` in `Square` is invalid. A `static` method cannot override a non-`static` method and vice versa. For this reason, Option B is the correct answer.

32. C. The application does not compile, but not for any reason having to do with the cast in the `main()` method. The `Rotorcraft` class includes an abstract method, but the class itself is not marked `abstract`. Only interfaces and abstract classes can include abstract methods. Since the code does not compile, Option C is the correct answer.

33. B. A class can trivially be assigned to a superclass reference variable but requires an explicit cast to be assigned to a subclass reference variable. For these reasons, Option B is correct.

34. C. A concrete class is the first non-abstract subclass that extends an abstract class and implements any inherited interfaces. It is required to implement all inherited abstract methods, making Option C the correct answer.

35. D. First of all, interfaces can only contain `abstract`, `final`, and `default` methods. The method `fly()` defined in `CanFly` is not marked `static` or `default` and defines an implementation, an empty `{}`, meaning it cannot be assumed to be `abstract`; therefore, the code does not compile. Next, the implementation of `fly(int speed)` in the `Bird` class also does not compile, but not because of the signature. The method body fails to return an `int` value. Since it is an overloaded method, if it returned a value it would compile without issue. Finally, the `Eagle` class does not compile because it extends the `Bird` class, which is marked `final` and therefore, cannot be extended. For these three reasons, Option D is the correct answer.

36. B. Abstract classes and interfaces can both contain `static` and `abstract` methods as well as `static` variables, but only an interface can contain `default` methods. Therefore, Option B is correct.

37. C. Java does not allow multiple inheritance, so having one class extend two interfaces that both define the same `default` method signature leads to a compiler error, unless the class overrides the method. In this case, though, the `talk(String...)` method defined in the `Performance` class is not an overridden version of method defined in the interfaces because the signatures do not match. Therefore, the `Performance` class does not compile since the class inherits two `default` methods with the same signature and no overridden version, making Option C the correct answer.

38. A. In Java, only non-`static`, non-`final`, and non-`private` methods are considered virtual and capable of being overridden in a subclass. For this reason, Option A is the correct answer.

39. B. An interface can only extend another interface, while a class can only extend another class. A class can also implement an interface, although that comparison is not part of the question text. Therefore, Option B is the correct answer.

40. A. The code compiles without issue, so Option D is incorrect. Java allows methods to be overridden, but not variables. Therefore, marking them `final` does not prevent them from being reimplemented in a subclass. Furthermore, polymorphism does not apply in the same way it would to methods as it does to variables. In particular, the reference type determines the version of the `secret` variable that is selected, making the output `2` and Option A the correct answer.

41. D. Options A and C are incorrect because an overridden method cannot reduce the visibility of the inherited method. Option B is incorrect because an overridden method cannot declare a broader checked exception than the inherited method. Finally, Option D is the correct answer. The removal of the checked exception, the application of a broader access modifier, and the addition of the `final` attribute are allowed for overridden methods.

42. C. The `setAnimal()` method requires an object that is `Dog` or a subclass of `Dog`. Since `Husky` extends `Dog`, Options A and B both allow the code to compile. Option D is also valid because a `null` value does not have a type and can be assigned to any reference variable. Option C is the only value that prevents the code from compiling because `Wolf` is not a subclass of `Dog`. Even though `Wolf` can be assigned to the instance `Canine` variable, the setter requires a compatible parameter.

43. A. An interface method can be abstract and not have a body, or it can be `default` or `static` and have a body. An interface method cannot be `final` though, making Option A the correct answer.

44. A. It looks like `getSpace()` in the `Room` class is an invalid override of the version in the `House` class since package-private is a more restrictive access modifier than `protected`, but the parameter list changes; therefore, this is an overloaded method, not an overridden one. Furthermore, the `Ballroom` class is abstract so no object is instantiated, but there is no requirement that an abstract class cannot contain a runnable `main()` method. For these reasons, the code compiles and runs without issue, making Option A correct.

45. D. Trick question! Option A seems like the correct answer, but the second part of the sentence is false, regardless of whether you insert *overloaded* or *overridden*. Overridden methods must have covariant return types, which may not be exactly the same as the type in the parent class. Therefore, Option D is the correct answer.

46. B. If a parent class does not include a no-argument constructor, a child class can still explicitly declare one; it just has to call an appropriate parent constructor with `super()`, making Option A incorrect. If a parent class does not include a no-argument constructor, the child class must explicitly declare a constructor, since the compiler will not be able to insert the default no-argument constructor, making Option B correct. Option C is incorrect because a parent class can have a no-argument constructor, while its subclasses do not. If Option C was true, then all classes would be required to have no-argument constructors since they all extend `java.lang.Object`, which has a no-argument constructor. Option D is also incorrect. The default no-argument constructor can be inserted into any class that directly extends a class that has a no-argument constructor. Therefore, no constructors in the subclass are required.

47. D. The object type relates to the attributes of the object that exist in memory, while the reference type dictates how the object is able to be used by the caller. For these reasons, Option D is correct.

48. A. The `play()` method is overridden in `Violin` for both `MusicCreator` and `StringInstrument`, so the return type must be covariant with both. `Long` is a subclass of `Number`, and therefore, it is covariant with the version in `MusicCreator`. Since it matches the type in `StringInstrument`, it can be inserted into the blank and the code would compile. While `Integer` is a subclass of `Number`, meaning the override for `MusicCreator` is valid, it is not a subclass of `Long` used in `StringInstrument`. Therefore, using `Integer` would cause the code to not compile. Finally, `Number` is compatible with the version of the method in `MusicCreator` but not with the version in `StringInstrument`, because `Number` is a superclass of `Long`, not a subclass. For these reasons, `Long` is the only class that allows the code to compile, making Option A the correct answer.

49. B. The primary motivation for adding `default` interface methods to Java was for backward compatibility. These methods allow developers to update older classes with a newer version of an interface without breaking functionality in the existing classes, making Option B the correct answer. Option is A is nonsensical and not the correct answer. Options C and D sound plausible, but both could be accomplished with `static` interface methods alone.

50. C. The rule for overriding a method with exceptions is that the subclass cannot throw any new or broader checked exceptions. Since `IOException` is a superclass of `EOFException`, from the question description, we see that this is a broader exception and therefore not compatible. For this reason, the code does not compile, and Option C is the correct answer.

Chapter 8: Handling Exceptions

1. D. A `try` block must include either a `catch` or `finally` block, or both. The `think()` method declares a `try` block but neither additional block. For this reason, the code does not compile, and Option D is the correct answer. The rest of the lines compile without issue, including `k1`.

2. B. The correct order of blocks is `try`, `catch`, and `finally`, making Option B the correct answer.

3. D. Option D is the correct model. The class `RuntimeException` extends `Exception`, and both `Exception` and `Error` extend `Throwable`. Finally, like all Java classes, they all inherit from `Object`. Notice that `Error` does not extend `Exception`, even though we often refer to these generally as exceptions.

4. A. While `Exception` and `RuntimeException` are commonly caught in Java applications, it is recommended `Error` not be caught. An `Error` often indicates a failure of the JVM which cannot be recovered from. For this reason, Option A is correct, and Options C and D are incorrect. Option B is not a class defined in the Java API; therefore, it is also incorrect.

5. D. The application does not compile because `score` is defined only within the `try` block. The other three places it is referenced, in the `catch` block, in the `finally` block, and outside the try-catch-finally block at the end, are not in scope for this variable and each does not compile. Therefore, the correct answer is Option D.

6. B. `ClassCastException`, `ArrayIndexOutOfBoundsException`, and `IllegalArgumentException` are unchecked exceptions and can be thrown at any time. `IOException` is a checked exception that must be handled or declared when used, making Option B the correct answer.

7. A. The `throws` keyword is used in method declarations, while the `throw` keyword is used to throw an exception to the surrounding process, making Option A the correct answer. The `catch` keyword is used to handle exceptions, not to create them or in the declaration of a method.

8. B. `IOException` is a subclass of `Exception`, so it must appear first in any related `catch` blocks. If `Exception` was to appear before `IOException`, then the `IOException` block would be considered unreachable code because any thrown `IOException` is already handled by the `Exception` `catch` block. For this reason, Option B is correct.

9. C. The application first enters the `try` block and outputs `A`. It then throws a `RuntimeException`, but the exception is not caught by the `catch` block since `RuntimeException` is not a subclass of `ArrayIndexOutOfBoundsException` (it is a superclass). Next, the `finally` block is called and `C` is output. Finally, the `RuntimeException` is thrown by the `main()` method and a stack trace is printed. For these reasons, Option C is correct.

10. C. The application does not compile, so Option D is incorrect. The `openDrawbridge()` method compiles without issue, so Options A and B are incorrect. The issue here is how the `openDrawbridge()` method is called from within the `main()` method on line `p3`. The `openDrawbridge()` method declares the checked exception, `Exception`, but the `main()` method from which it is called does not handle or declare the exception. In order for this code to compile, the `main()` method would have to have a try-catch statement around line `p3` that properly handles the checked exception, or the `main()` would have to be updated to declare a compatible checked exception. For these reasons, line `p3` does not compile, and Option C is the correct answer.

11. B. `NullPointerException` and `ArithmeticException` both extend `RuntimeException`, which are unchecked exceptions and not required to be handled or declared in the method in which they are thrown. On the other hand, `Exception` is a checked exception and must be handled or declared by the method in which it is thrown. Therefore, Option B is the correct answer.

12. A. The code compiles and runs without issues, so Options C and D are incorrect. The `try` block throws a `ClassCastException`. Since `ClassCastException` is not a subclass of `ArrayIndexOutOfBoundsException`, the first `catch` block is skipped. For the second `catch` block, `ClassCastException` is a subclass of `Throwable`, so that block is executed. Afterward, the `finally` block is executed and then control returns to the `main()` method with no exception being thrown. The result is that `1345` is printed, making Option A the correct answer.

13. C. A `finally` block can throw an exception, in which case not every line of the `finally` block would be executed. For this reason, Options A and D are incorrect. Option B is also incorrect The `finally` block is called regardless of whether or not the related `catch` block is executed. Option C is the correct answer. Unlike an if-then statement, which can take a single statement, a `finally` statement requires brackets `{}`.

14. C. The code does not compile because the `catch` blocks are used in the wrong order. Since `IOException` is a superclass of `FileNotFoundException`, the `FileNotFoundException` is considered unreachable code. For this reason, the code does not compile, and Option C is correct.

15. C. A `try` statement requires a `catch` or a `finally` block. Without one of them, the code will not compile; therefore, Option D is incorrect. A `try` statement can also be used with both a `catch` and `finally` block, making Option C the correct answer. Note that `finalize` is not a keyword, but a method inherited from `java.lang.Object`.

16. B. Option A is a true statement about exceptions and when they are often applied. Option B is the false statement and the correct answer. An application that throws an exception can choose to handle the exception and avoid termination. Option C is also a true statement. For example, a `NullPointerException` can be avoided on a `null` object by testing whether or not the object is `null` before attempting to use it. Option D is also a correct statement. Attempting to recover from unexpected problems is an important aspect of proper exception handling.

17. D. The code does not compile because the `catch` block uses parentheses `()` instead of brackets `{}`, making Option D the correct answer. Note that `Boat` does not extend `Transport`, so while the override on line `j1` appears to be invalid since `Exception` is a broader checked exception than `CapsizedException`, that code compiles without issue. If the `catch` block was fixed, the code would output `4`, making Option A the correct answer.

18. B. Overridden methods cannot throw new or broader checked exceptions than the one they inherit. Since `Exception` is a broader checked exception than `PrintException`, Option B is not allowed and is the correct choice. Alternatively, declaring narrower or the same checked exceptions or removing them entirely is allowed, making Options A and C incorrect. Since Option B is correct, Option D is incorrect.

19. D. All three of those classes belong to the `java.lang` package, so Option C seems like the correct answer. The Java compiler, though, includes `java.lang` by default, so no `import` statement is actually required to use those three classes, making Option D the correct answer.

20. C. The code does not compile because the `catch` block is missing a variable type and name, such as `catch (Exception e)`. Therefore, Option C is the correct answer. Both implementations of `getSymbol()` compile without issue, including the overridden method. A subclass can swallow a checked exception for a method declared in a parent class; it just cannot declare any new or broader checked exceptions.

21. B. Checked exceptions must be handled or declared or the program will not compile, while unchecked exceptions can be optionally handled. On the other hand, `java.lang.Error` should never be handled by the application because it often indicates an unrecoverable state in the JVM, such as running out of memory. For these reasons, Option B is the correct answer.

22. B. The application does not compile, so Option D is incorrect. The checked `KnightAttackingException` thrown in the `try` block is handled by the associated `catch` block. The `ClassCastException` is an unchecked exception, so it is not required to be handled or declared and line `q1` compiles without issue. The `finally` block throws a checked `CastleUnderSiegeException`, which is required to be handled or declared by the method, but is not. There is no try-catch around line `q2`, and the method does not declare a compatible checked exception, only an unchecked exception. For this reason, line `q2` does not compile, and Option B is the correct answer. Lastly, line `q3` compiles without issue because the unchecked `RuntimeException` is not required to be handled or declared by the call in the `main()` method.

23. A. If an exception matches multiple `catch` blocks, the first one that it encounters will be the only one executed, making Option A correct, and Options B and C incorrect. Option D is also incorrect. It is possible to write two consecutive `catch` blocks that can catch the same exception, with the first type being a subclass of the second. In this scenario, an exception thrown of the first type would match both `catch` blocks, but only the first `catch` block would be executed, since it is the more specific match.

24. C. The code does not compile due to the call to `compute()` in the `main()` method. Even though the `compute()` method only throws an unchecked exception, its method declaration includes the `Exception` class, which is a checked exception. For this reason, the checked exception must be handled or declared in the `main()` method in which it is called. While there is a try-catch block in the `main()` method, it is only for the unchecked `NullPointerException`. Since `Exception` is not a subclass of `NullPointerException`, the checked `Exception` is not properly handled or declared and the code does not compile, making Option C the correct answer.

25. D. A `NullPointerException` can be thrown if the value of `list` is `null`. Likewise, an `ArrayIndexOutOfBoundsException` can be thrown if the value of `list` is an array with fewer than 10 elements. Finally, a `ClassCastException` can be thrown if `list` is assigned an object that is not of type `Boolean[]`. For example, the assignment `list = (Boolean[]) new Object()` will compile without issue but throws a `ClassCastException` at runtime. Therefore, the first three options are possible, making Option D the correct answer.

26. B. A `StackOverflowError` occurs when a program recurses too deeply into an infinite loop. It is considered an error because the JVM often runs out of memory and cannot recover. A `NullPointerException` occurs when an instance method or variable on a `null` reference is used. For these reasons, Option B is correct. A `NoClassDefFoundError` occurs when code available at compile time is not available at runtime. A `ClassCastException` occurs when an object is cast to an incompatible reference type. Finally, an `IllegalArgumentException` occurs when invalid parameters are sent to a method.

27. C. Checked exceptions are commonly used to force a caller to deal with an expected type of problem, such as the inability to write a file to the file system. Without dealing with all checked exceptions thrown by the method, the calling code does not compile, so Option A is a true statement. Option B is also a true statement. Declaring various different exceptions informs the caller of the potential types of problems the method can encounter. Option C is the correct answer. There may be no recourse in handling an exception other than to terminate the application. Finally, Option D is also a true statement because it gives the caller a chance to recover from an exception, such as writing file data to a backup location.

28. D. This code does not compile because the `catch` and `finally` blocks are in the wrong order, making Option D the correct answer. If the order was flipped, the output would be `Finished!Joyce Hopper`, making Option B correct.

29. A. A `try` statement is not required to have a `finally` block, but if it does, there can be at most one. Furthermore, a `try` statement can have any number of `catch` blocks or none at all. For these reasons, Option A is the correct answer.

30. D. The code compiles without issue, so Option C is incorrect. The key here is noticing that `count`, an instance variable, is initialized with a value of `0`. The `getDuckies()` method ends up computing `5/0`, which leads to an unchecked `ArithmeticException` at runtime, making Option D the correct answer.

31. B. If both the `catch` and `finally` blocks throw an exception, the one from the `finally` block is propagated to the caller, with the one from the `catch` block being dropped, making Option B the correct answer. Note that Option C is incorrect due to the fact that only one exception can be thrown to the caller.

32. A. The application does not compile because the `roar()` method in the `BigCat` class uses `throw` instead of `throws`, making Option A the correct answer. Note that if the correct keyword was used, the code would compile without issues, and Option D would be correct. Also the override of `roar()` in the `Lion` class is valid, since the overridden method has a broader access modifier and does not declare any new or broader checked exceptions.

33. A. Although this code uses the `RuntimeException` and `Exception` classes, the question is about casting. `Exception` is not a subclass of `RuntimeException`, so the assignment on the second line throws a `ClassCastException` at runtime, making Option A correct.

34. C. All exceptions in Java inherit from `Throwable`, making Option C the correct answer. Note that `Error` and `Exception` extend `Throwable`, and `RuntimeException` extends `Exception`.

35. B. If both values are valid non-`null String` objects, then no exception will be thrown, with the statement in the `finally` block being executed first, before returning control to the `main()` method; therefore, the second statement is a possible output. If either value is `null`, then the `toString()` method will cause a `NullPointerException` to be thrown. In both cases, the `finally` block will execute first, printing `Posted:`, even if there is an exception. For this reason, the first statement is not a possible output, and Option B is correct.

36. A. `ClassCastException` is a subclass of `RuntimeException`, so it must appear first in any related `catch` blocks. If `RuntimeException` was to appear before `ClassCastException`, then the `ClassCastException` block would be considered unreachable code, since any thrown `ClassCastException` is already handled by the `RuntimeException catch` block. For this reason, Option A is correct.

37. C. Option A is incorrect. You should probably seek help if the computer is on fire! Option B is incorrect because code that does not compile cannot run and therefore cannot throw any exceptions. Option C is the best answer, since an `IllegalArgumentException` can be used to alert a caller of missing or invalid data. Option D is incorrect; finishing sooner is rarely considered a problem.

38. C. The code does not compile due to an invalid override of the `operate()` method. An overridden method must not throw any new or broader checked exceptions than the method it inherits. Even though `RuntimeException` is a subclass of `Exception`, `Exception` is considered a new checked exception, since `RuntimeException` is an unchecked exception. Therefore, the code does not compile, and Option C is correct.

39. D. A `NullPointerException` is an unchecked exception. While it can be handled by the surrounding method, either through a try-catch block or included in the method declaration, these are optional. For this reason, Option D is correct.

40. D. In this application, the `throw RuntimeException(String)` statement in the `zipper()` method does not include the `new` keyword. The `new` keyword is required to create the object being thrown, since `RuntimeException(String)` is a constructor. For this reason, the code does not compile, and Option D is correct. If the keyword `new` was inserted properly, then the `try` block would throw a `CastClassException`, which would be replaced with a `RuntimeException` to the calling method by the `catch` block. The `catch` block in the `main()` method would then be activated, and no output would be printed, making Option C correct.

41. C. For this question, notice that all the exceptions thrown or caught are unchecked exceptions. First, the `ClassCastException` is thrown in the `try` block and caught by the second `catch` block since it inherits from `RuntimeException`, not `IllegalArgumentException`. Next, a `NullPointerException` is thrown, but before it can be returned the `finally` block is executed and a `RuntimeException` replaces it. The application exits and the caller sees the `RuntimeException` in the stack trace, making Option C the correct answer. If the `finally` block did not throw any exceptions, then Option B would be the correct answer.

42. D. Trick question! Options A, B, and C are each invalid overrides of the method because the return type must be covariant with `void`. For this reason, Option D is the correct answer. If the return types were changed to be `void`, then Option A would be a valid override. Options B and C would still be incorrect, since overridden methods cannot throw broader checked exceptions than the inherited method.

43. D. The code does not compile because the `catch` block is missing a variable name, such as `catch (Error e)`. Therefore, Option D is the correct answer. If a variable name was added, the application would produce a stack trace at runtime and Option C would be the correct answer. Because `IllegalArgumentException` does not inherit from `Error`, the `catch` block would be skipped and the exception sent to the `main()` method at runtime. Note that the declaration of `RuntimeException` by both methods is unnecessary since it is unchecked, although allowed by the compiler.

44. D. The `openDrawbridge()` is capable of throwing a variety of exceptions, including checked `Exception` and `DragonException` as well as an unchecked `RuntimeException`. All of these are handled by the fact that the method declares the checked `Exception` class in the method signature, which all the exceptions within the class inherit. For this reason, the `openDrawbridge()` method compiles without issue. The call to `openDrawbridge()` in the `main()` method also compiles without issue because the `main()` method declares `Exception` in its signature. For these reasons, the code compiles but a stack trace is printed at runtime, making Option D the correct answer. In case you are wondering, the caller would see `RuntimeException: Or maybe this one` in the stack trace at runtime, since the exception in the `finally` block replaces the one from the `try` block. Note that the exception in the `catch` block is never reached because the `RuntimeException` type declared in the `catch` block does not handle `Exception`.

45. C. Both `IllegalArgumentException` and `ClassCastException` inherit `RuntimeException`, but neither is a subclass of the other. For this reason, they can be listed in either order, making Option C the correct statement.

46. D. The class `RuntimeException` is not an interface and it cannot be implemented. For this reason, the `Problem` class does not compile, and Option D is the correct answer. Note that this is the only compilation problem in the application. If `implements` was changed to `extends`, the code would compile and `Problem?Fixed!` would be printed, making Option A the correct answer.

47. D. The question is designed to see how closely you pay attention to `throw` and `throws`! The `try` block uses the incorrect keyword, `throws`, to create an exception. For this reason, the code does not compile, and Option D is correct. If `throws` was changed to `throw`, then the code would compile without issue, and Option B would be correct.

48. D. A Java application tends to only throw an `Error` when the application has entered a final, unrecoverable state. Options A and C are incorrect. These types of errors are common and expected in most software applications, and should not cause the application to terminate. Option B uses the word *temporarily*, meaning the network connection will come back. In this case, a regular exception could be used to try to recover from this state. Option D is the correct answer because running out of memory is usually unrecoverable in Java.

49. C. While a `catch` block is permitted to include an embedded try-catch block, the issue here is that the variable name `e` is already used by the first `catch` block. In the second `catch` block, it is equivalent to declaring a variable `e` twice. For this reason, line `z1` does not compile, and Option C is the correct answer. If a different variable name was used for either `catch` block, then the code would compile without issue, and Option A would be the correct answer.

50. B. The `finally` block of the `snore()` method throws a new checked exception on line `x1`, but there is no try-catch block around it to handle it, nor does the `snore()` method declare any checked exceptions. For these reasons, line `x1` does not compile, and Option B is the correct answer. The rest of the lines of code compile without issue, even line `x3` where a `static` method is being accessed using an instance reference. Note that the code inside the `try` block, if it ran, would produce an `ArrayIndexOutOfBoundsException`, which would be caught by the `RuntimeException catch` block, printing `Awake!`. What happens next would depend on how the `finally` block was corrected.

Chapter 9: Working with Selected Classes from the Java API

1. C. Option A is incorrect because `StringBuilder` does not support multiple threads. In fact, threads aren't even covered on the OCA, which should be your clue that this answer is wrong! You don't need to know this for the exam, but `StringBuffer` supports multiple threads. Option B is incorrect because `==` compares references, not values. Option D is incorrect because both `String` and `StringBuilder` support languages and encodings. Option C is correct and the primary reason to use `StringBuilder`. `String` often creates a new object each time you call certain methods on the object like `concat()`. `StringBuilder` optimizes operations like `append()` because it is mutable.

2. D. A `String` can be created using a literal rather than calling a constructor directly, making Option A incorrect. A string pool exists for `String` reuse, making Option B incorrect. A `String` is final and immutable, making Option C incorrect and Option D correct.

3. D. This question is testing whether you understand how method chaining works. Option A creates an empty `StringBuilder` and then adds the five characters in `clown` to it. Option B simply creates the `clown` when calling the constructor. Finally, Option C creates the same value, just in two parts. Therefore, Option D is correct.

4. B. Since `StringBuilder` is mutable, each call to `append` adds to the value. When calling `print`, `toString()` is automatically called and `333 806 1601` is output. Therefore, Option B is correct.

5. B. `List` is an interface and not a class. It cannot be instantiated. While `Object` is a concrete class, it does not implement the `List` interface so it cannot be assigned to `frisbees`. Note that if you were to add an explicit cast, it would compile and throw an exception at runtime. Of the three options, only `ArrayList` can fill in the blank, so Option B is correct.

6. C. An `ArrayList` does not automatically sort the elements. It simply remembers them in order. Since Java uses zero-based indexes, Option C is correct.

7. C. Calling the constructor and then `insert()` is an example of method chaining. However, the `sb.length()` call is a problem. The `sb` reference doesn't exist until after the chained calls complete. Just because it happens to be on a separate line doesn't change when the reference is created. Since the code does not compile, Option C is correct.

8. A. While the `ArrayList` is declared with an initial capacity of one element, it is free to expand as more elements are added. Each of the three calls to the `add()` method adds an element to the end of the `ArrayList`. The `remove()` method call deletes the element at index 2, which is `Art`. Therefore, Option A is correct.

9. C. On line 12, the value of the `StringBuilder` is `12`. On line 13, it becomes `123`. Since `StringBuilder` is mutable, storing the result in the same reference is redundant. Then on line 14, the value is reversed, giving us `321` and making Option C correct.

10. D. Option A is incorrect as it describes autoboxing. Options B and C are not possible in Java. Option D is correct as it describes lambdas. Lambdas use deferred execution and can be run elsewhere in the codebase.

11. D. A `StringBuilder` is mutable, so the length is two after line 6 completes. The `StringBuilder` methods return a reference to the same object so you can chain method calls. Therefore, `line` and `anotherLine` refer to the same object. This means that line 7 prints `true`. Then on line 9, both references point to the same object of length 2, and Option D is correct.

12. D. The `add()` and `get()` methods are available on `ArrayList`. However, `ArrayList` uses `size` rather than `length` to get the number of elements. Therefore, Option D is correct. If `length` was changed to `size`, Option B would compile if put in the blank. Option A still wouldn't compile in the blank because a cast would be needed to store the value in `str`.

13. D. Option A is tricky, but incorrect. While a lambda can have zero parameters, a `Predicate` cannot. A `Predicate` is defined as a type mapping to a `boolean`. Option B is clearly incorrect as `->` separates the parts of a lambda. Options C and D are similar. Option C is incorrect because `return` is only allowed when the brackets are present. Option D is correct.

14. A. Lines 20–22 create an `ArrayList` with two elements. Line 23 replaces the second one with a new value. Now `chars` is `[a, c]`. Then line 24 removes the first element, making it just `[c]`. Option A is correct because there is only one element, but it is not the value `b`.

15. D. Trick question. There is no reverse method on the `String` class. There is one on the `StringBuilder` class. Therefore, the code does not compile, and Option D is correct.

16. A. When creating a lambda with only one parameter, there are a few variants. The `pred1` approach shows the shortest way, where the type is omitted and the parentheses are omitted. The `pred2` approach is similar except it includes the parentheses. Both are legal. The `pred4` approach is the long way with both the parentheses and type specified. The only one that doesn't compile is `pred3`. The parentheses are required if including the type.

17. A. This is a correct example of code that uses a lambda. The interface has a single abstract method. The lambda correctly takes one `double` parameter and returns a `boolean`. This matches the interface. The lambda syntax is correct. Since 45 is greater than 5, Option A is correct.

18. A. Since `String` is immutable, each call to `concat()` returns a new object with the new value. However, that return value is ignored and the `teams` variable never changes in value. Therefore it stays as `694`, and Option A is correct.

19. A. The `ArrayList` class is in the `java.util` package, making I correct. The `LocalDate` class is in the `java.time` package, making II incorrect. The `String` class is in the `java.lang` package, which means you can use it without typing an import, making III incorrect. Therefore, Option A is correct.

20. C. Option A is straightforward and outputs `radical robots`. Option B does the same in a convoluted manner. First Option B removes all the characters after the first one. It doesn't matter that there aren't actually 100 characters to delete. Then it appends `obots` to the end, making the builder contain `robots`. Finally, it inserts the remainder of the string immediately after the first index. Try drawing the flow if this is hard to envision. Option D also creates the same value by inserting `robots` immediately after the end of the `StringBuilder`. Option C is close, but it has an off-by-one error. It inserts `robots` after the letter `l` rather than after the space. This results in the value `radicalrobots` followed by a space. Option C is different than the others and the correct answer.

21. A. Since we are creating the list from an array, it is a fixed size. We are allowed to change elements. At the end of this code, `museums` is `[Art, Science]`. Therefore, it contains `Art`, and Option A is correct.

22. D. Options A and B are not true if the `String` is `"deabc"`. Option C is not true if the `String` is `"abcde"`. Option D is true in all cases.

23. D. Line 25 does not compile. On an `ArrayList`, the method to get the number of elements is `size`. The `length()` method is used for a `String` or `StringBuilder`.

24. B. The `toString()` method call doesn't help in narrowing things down as all Java objects have that method available. The other two methods are more helpful. `String` is the only type of these three to have a `startsWith()` method, making Option B correct. `String` also has the `replace()` method declared here. If you memorized the whole API, you might know that `StringBuilder` also has a `replace()` method, but it requires three parameters instead of two. Please don't memorize the API in that level of detail. We included what you need to know in our study guide. If you do have this outside knowledge, be careful not to read into the questions!

25. B. The <> is known as the diamond operator. Here, it works as a shortcut to avoid repeating the generic type twice for the same declaration. On the right side of the expression, this is a handy shortcut. Java still needs the type on the left side so there is something to infer. In the figure, position P is the left side and position Q is the right side. Therefore, Option B is correct.

26. D. The type in the lambda must match the generic declared on the `Predicate`. In this case, that is `String`. Therefore, Options A and B are incorrect. While Option C is of the correct type, it uses the variable `s`, which is already in use from the `main()` method parameter. Therefore, none of these are correct, and Option D is the answer.

27. A. A `String` is immutable so a different object is returned on line 6. The object `anotherLine` points to is of length 2 after line 6 completes. However, the original `line` reference still points to an object of length 1. Therefore, Option A is correct.

28. C. While it is common for a `Predicate` to have a generic type, it is not required. However, it is treated like a `Predicate` of type `Object` if the generic type is missing. Since `startsWith()` does not exist on `Object`, the first line does not compile, and Option C is correct.

29. B. `LocalDate` only includes the date portion and not the time portion. There is no class named `LocalTimeStamp`. The other two, `LocalDateTime` and `LocalTime`, both include the time elements, making Option B correct.

30. D. Line 4 creates a `String` of length 5. Since `String` is immutable, line 5 creates a new `String` with the value `1` and assigns it to `builder`. Remember that indexes in Java begin with 0, so the `substring()` method is taking the values from the fifth element through the end. Since the first element is the last element, there's only one character in there. Then line 6 tries to retrieve the second indexed element. Since there is only one element, this gives a `StringIndexOutOfBoundsException`, and Option D is correct.

31. D. When you're using brackets, both the `return` keyword and semicolon are needed for the lambda to compile, making Option D correct.

32. B. Java 8 date and time classes are immutable. The `plusDays` method returns a `LocalDate` object presenting Christmas Eve (December 24th). However, this return value is ignored. The `xmas` variable still represents the original value, so Option B is correct.

33. A. Line 3 creates an empty `StringBuilder`. Line 4 adds three characters to it. Line 5 removes the first character, resulting in `ed`. Line 6 deletes the characters starting at position 1 and ending right before position 2, which removes the character at index 1, which is `d`. The only character left is `e`, so Option A is correct.

34. B. While it is common for a `Predicate` to have a generic type, it is not required. When the generic is omitted, it is treated like a `Predicate` of type `Object`. Since the `equals()` method exists on `Object`, this is fine. Option B is correct because the `Predicate` tests as `false`.

35. C. In Java, most things use zero-based indexes, including arrays and a `String`. Months are an exception to this convention starting Java 8. This makes the answer either Option C or D. However, `LocalTime` does not contain date fields, so it has to be Option C.

36. C. `Predicate` is an interface with one method. The method signature is `boolean test(T t)`. Option C is the answer because the method accepts one parameter rather than two.

37. B. Be careful here. The `Period` class uses a `static` helper method to return the period. It does not chain method calls, so `period1` only represents three days. Since three days is less than 10 days, `period2` is larger, and Option B is correct.

38. B. The code starts by correctly creating a date representing January 1, 2017, and a period representing one day. It then explicitly defines the format as month followed by day followed by year. Finally, the code subtracts a day, giving us the formatted version of December 31, 2016.

39. C. The `trim()` method returns a `String` with all leading and trailing white space removed. In this question, that's the seven-character `String`: `":) - (:"`. Options A and B are incorrect because they do not remove the first blank space in `happy`. Option D is incorrect because it does not remove the last character in `happy`. Therefore, Option C is correct.

40. C. The `Period` class creates immutable objects and is usually used to add/subtract from a `LocalDate` or `LocalDateTime` object. It allows creating date, week, month, or year periods. Since it cannot be used for time, Option C is the answer.

41. D. Line 4 creates a `StringBuilder` of length 5. Pay attention to the `substring()` method `StringBuilder`. It returns a `String` with the value `321`. It does not change the `StringBuilder` itself. Then line 6 is retrieving the second indexed element from that unchanged value, which is `4`. Therefore, Option D is correct.

42. B. This one is tricky. There are two `remove()` methods available on `ArrayList`. One removes an element by index and takes an `int` parameter. The other removes an element by value. Due to the generics, it takes an `Integer` parameter in this example. Since the `int` primitive is a better match, the element with index 2 is removed, which is the value of `1`. Therefore, Option B is correct.

43. C. `ArrayList` has a `size()` method rather than a `length()` method, making Option A incorrect. The `charAt()` and `length()` methods are declared on both `String` and `StringBuilder`. However, the `insert()` method is only declared on a `StringBuilder` and not a `String`. Therefore, Option C is correct.

44. C. The `minusNanos` and `plusNanos` are the smallest units available, making Option C correct. Option D is incorrect because `LocalTime` is not that granular. Note that while you can add milliseconds by adding many nanoseconds, there isn't a method for it. A millisecond is also larger than a nanosecond. Finally, don't be tricked by the fact that `LocalTime` is immutable. You can still add time; it just gets returned as a different object.

45. D. When creating a formatter object, remember that `MM` represents month while `mm` represents minute. Since there are not minutes defined on a `LocalDate` object, the code throws an `UnsupportedTemporalTypeException`. You don't need to know the name of the exception, but you do need to know that an exception is thrown.

46. D. There are two signatures for the `replace()` method. One takes two `char` parameters. The other signature takes a `CharSequence`. Both `String` and `StringBuilder` implement this interface. This makes all three alternatives correct, and Option D is correct.

47. C. Pay attention to the data types. The `print()` method is looping through a list of `String` objects. However, the `Predicate` expects an `Integer`. Since these don't match, the `if` statement does not compile.

48. D. Line 12 creates an empty `ArrayList`. While it isn't recommended to use generics on only the left side of the assignment operator, this is allowed. It just gives a warning. Lines 13 and 14 add two elements. Line 15 resets to an empty `ArrayList`. Line 16 adds an element, so now we have an `ArrayList` of size 1. Line 17 attempts to remove the element at index 1. Since Java uses zero-based indexes, there isn't an element there and the code throws an `IndexOutOfBoundsException`.

49. C. The declaration of `witch` is incorrect. It tries to store a `char` into a `String` variable reference. This does not compile, making Option C correct. If this was fixed, the answer would be Option B.

50. C. The Java 8 date and time classes are immutable. This means they do not contain setter methods and the code does not compile.

Chapter 10: OCA Practice Exam

1. E. The first time through the loop, we are calling `indexOf` on an empty `StringBuilder`. This returns `-1`. Since we cannot insert at index -1, the code throws a `StringIndexOutOfBoundsException`.

2. C, E. In Option A, the assignment operator `=` incorrectly comes after the addition `+` operator. In Option B, the addition operator `+` incorrectly comes after the division `/` operator. In Option D, the subtraction operator `-` incorrectly comes after the multiplication `*` operator. This leaves Options C and E as the correct answers. For these answers, it may help to remember that the modulus operator `%`, multiplication operator `*`, and division operator `/` have the same operator precedence.

3. B, C, F. Option A is incorrect because a getter should not take a value. Option D is incorrect because the prefix `is` should only be with `boolean` values. Option E is incorrect

because `gimme` is not a valid JavaBean prefix. Options B, C, and F are each proper JavaBean method signatures.

4. A, E. Line 24 does not compile because arrays use `length`. It is `ArrayList` that uses `size()`. All of the other lines compile, making Option A correct. It is allowed to split up the braces in the 2D array declaration on line 20. The code is also allowed to use `crossword.length` as the loop condition on line 22, although this is not a good idea. The array starts out with all 200 of the cells initialized to the default value for an `int` of 0. Both loops iterate starting at 0 and stopping before 10, which causes only half of the array to be set to `'x'`. The other half still has the initial default value of 0, making Option E correct.

5. B, D. Options A and E are incorrect because they indicate states that the application can possibly recover from. An `Error` generally indicates an unrecoverable problem. While it is possible to catch an `Error`, it is strongly recommended that an application never do so, making Options B and D correct. Finally, Option C is incorrect because `Error` extends from `Throwable`, not `Exception`, and is unchecked.

6. A, C, D. The first `import` statement allows only the class `forest.Bird` to be available, making Option A correct and Options E and F incorrect. Option B is incorrect since the third `import` statement only allows access to classes within the `savana` package, not any sub-packages. Option C is correct because the second `import` statement allows any class in the `jungle.tree` package to be accessible. Finally, Option D is correct because `java.lang.*` is implicitly included in all Java classes.

7. C. *Mutable* means the object can change state. *Immutable* means the object cannot change state. An `ArrayList` stores a collection of objects. It mutates as the elements change. A `StringBuilder` is also mutable as it improves performance by not creating a new object each time it changes. A `String` is immutable. Methods that look like they change the value simply return a different `String` object. The date/time objects added in Java 8, such as `LocalDateTime`, are also immutable. Therefore, Option C is correct with `String` and `LocalDateTime` as the immutable object types.

8. C. On the first iteration through the loop, the first five characters are removed and `builder` becomes `s growing`. Since there are more than five characters left, the loop iterates again. This time, five more characters are removed and `builder` becomes `wing`. This matches Option C.

9. D. The code compiles without issue, so Option E is incorrect. The key here is that none of the variables are assigned the same object due to the use of the `new` keyword. Comparing any two variables with `==` will always result in an evaluation of `false`, making the first two values of the `print` statement be `false` and `false`. On the other hand, they all have an underlying `String` value equivalent to `up`, so calling `equals()` on any two variables will return `true`. Option D is the correct answer that matches what the application will print.

10. C. Lines 4 and 5 both print `false` since a `String` should be compared with a method rather than `==`, especially when not comparing two values from the string pool. Line 6 also prints `false` because one value is uppercase and the other is lowercase. Line 7 prints `true` because both values are uppercase. Lines 8 and 9 print `true` because they don't look at the case. This makes Option C the answer.

11. A, B, C. Let's look at each one in turn. Option A is correct because the labels are not referenced. Option B is correct because the outer `while` is broader than the inner `while`. Since there is no other code in the loop, it is not needed. Option C is also correct because a label is not used. Option D is incorrect because the inner loop is more specific than the outer loop. While the code still compiles, it prints one less chapter. Options E and F are incorrect because you cannot remove one half of a loop construct and have it compile.

12. B, C. A `long` cannot contain a number with decimal points, preventing Options B and C from compiling. Options D and E show you can force a number to be a `long` by ending it with an upper- or lowercase `L`. This does not work if the number has a decimal point. Option F shows how to use underscores to break up a number.

13. A. A `while` loop checks the condition before executing. Since the hour is not less than one, the loop never enters, and Option A is correct. This is good, because we'd have an infinite loop if the loop was entered since the result of `plusHours` is ignored.

14. D. This question appears to ask you about involved array logic. Instead, it is checking to see if you remember that instance and class variables are initialized to `null`. Line 6 throws a `NullPointerException`. If the array was declared, the answer would be E because the code would throw an `ArrayStoreException` on line 8.

15. C, E. The diamond operator is only allowed to be used when instantiating rather than declaring. In other words, it can't go on the left side of the equal sign. Therefore, Options B, D, and F are incorrect. The remaining three options compile. However, Option A produces a warning because generics are not used on the right side of the assignment operator. Therefore, Options C and E are correct. Option C is better than Option E since it uses the diamond operator rather than specifying a redundant type.

16. B, D. At the end of the method, `shoe1` and `shoe3` both point to `"flip flop"`. `shoe2` points to `"croc"`. Since there are no references to `"sandal"`, it is eligible for garbage collection, making Option B correct. However, garage collection is not guaranteed to run, so Option D is also correct.

17. C. The code does not compile, so Options A and B are incorrect. The `getFish()` method is declared properly in the `Fish` class and successfully overridden in the `Clownfish` class. An overridden method must not declare any new or broader checked exceptions, but it is allowed to declare narrower exceptions or drop checked exceptions. The overridden method also uses a covariant return type. The use of `final` on the method and class declarations has no meaningful impact, since the methods and classes are not extended in this application. So where does the compilation error occur? In the `main()` method! Even though the `Clownfish` version of `getFish()` does not declare a checked exception, the call `f.getFish()` uses a `Fish` reference variable. Since the `Fish` reference variable is used and that version of the method declares a checked `Exception`, the compiler enforces that the checked exception must be handled by the `main()` method. Since this checked exception is not handled with a try-catch block nor by the `main()` method declaration, the code does not compile, and Option C is the correct answer.

18. A. This is a correct example of using lambdas. The code creates an `ArrayList` with three elements. The `print()` method loops through and checks for negative numbers. Option A is correct.

19. F. A `try` statement requires a `catch` or a `finally` block. It can also have both a `catch` and a `finally` block. Since no option matches these rules, Option F is the correct answer. Note that `finalize` is not a keyword but a method inherited from `java.lang.Object`. Lastly, the `throws` keyword can be applied to method declarations and is not used as part of a `try` statement.

20. A. On line 12, `result` is first set to `8`. On line 13, the `boolean` condition is `true` because `8 > 7`. On line 13, `result` is incremented to `9`. Then the inner loop runs, decrementing `result` until it is no longer greater than `5`. On line 18, loop execution is completed because `result` is equal to `5`. The `break` statement says to skip to after the labeled loop, which is line `20`. Then `result` is printed as `5`, making Option A correct.

21. C. The code compiles and runs without exception, making Options E and F incorrect. The question is testing your knowledge of variable scope. The `teeth` variable is `static` in the `Alligator` class, meaning the same value is accessible from all instances of the class, including the `static main()` method. The `static` variable `teeth` is incremented each time the constructor is called. Since `teeth` is a local variable within the `snap()` method, the argument value is used, but changes to the local variable do not affect the `static` variable `teeth`. Since the local variable `teeth` is not used after it is decremented, the decrement operation has no meaningful effect on the program flow or the `static` variable `teeth`. Since the constructor is called twice, with `snap()` executed after each constructor call, the output printed is `1 2`, making Option C the correct answer.

22. A. A `String` is immutable. Since the result of the `concat()` method call is ignored, the value of `witch` never changes. It stays as a single letter, and Option A is correct.

23. A, C, F. An interface method is exactly one of three types: `default`, `static`, or `abstract`. For this reason, Options A, C, and F are correct. An interface method cannot be `protected` nor `private` because the access modifier is always `public`, even when not specified, making Options B and D incorrect. Option E is also incorrect because `final` cannot be applied to `static` methods, since they cannot be overridden. It can also not be applied to `default` and `abstract` methods because they are always able to be overridden.

24. D. Look at the loop condition carefully. It tries to assign `null` to a `String` variable. This is not an expression that returns a `boolean`. Therefore, the code does not compile, and Option D is correct. If this was fixed by making the loop condition `tie == null`, then Option B would be correct.

25. B, F. A class may be defined without a `package` statement, making the class part of the default package. For this reason, Options A and D are incorrect. Every Java class implicitly imports exactly one package, `java.lang.*`, making Option B correct and Option C incorrect. Option E is incorrect because an `import` statement is not required. Finally, Option F is correct; any class that does not extend another class implicitly extends `java.lang.Object`.

26. D. A class cannot inherit two interfaces that declare the same `default` method, unless the class overrides them. In this case, the version of `grow()` in the `Tree` class is an overloaded method, not an overridden one. Therefore, the code does not compile due to the declaration of `Tree` on line `m1`, and Option D is the correct answer.

27. D. Variables are allowed to start with an underscore and are allowed to contain a `$`. Therefore, all the variable declarations compile, making Options A, B, and C incorrect. However, the `println()` refers to the uninitialized local `boolean`. Since local variables are not automatically initialized, the code does not compile, and Option D is correct.

28. A. Prefix operators, such as `--x` and `++x`, modify the variable and evaluate to the new value, while postfix operators, such as `x--` and `x++`, modify the variable but return the original value. Therefore, Option A is the correct answer.

29. B, C, E. The constructors declared by Options A, D, and F compile without issue. Option B does not compile. Since there is no call to a parent constructor or constructor in the same class, the compiler inserts a no-argument `super()` call as the first line of the constructor. Because `Big` does not have a no-argument constructor, the no-argument constructor `Trouble()`does not compile. Option C also does not compile because `super()` and `this()` cannot be called in the same constructor. Note that if the `super()` statement was removed, it would still not compile since this would be a recursive constructor call. Finally, Option E does not compile. There is no matching constructor that can take a `String` followed by a `long` value. If the input argument `deep` was an `int` in this constructor, then it would match the constructor used in Option D and compile without issue.

30. E, F. A `static` method is not allowed to access instance variables without an instance of the class, making Options E and F correct. Notice that only `max` is initialized to `100` in Option E. Since `min` doesn't have a value specified, it gets the default value, which is `0`.

31. B, E. The ternary `? :` operator only evaluates one of the two right-hand expressions at runtime, so Option A is incorrect. A `switch` statement may contain at most one optional `default` statement, making Option B correct. A single if-then statement can have at most one `else` statement, so Option C is incorrect. Note that you can join if-then-else statements together, but each `else` requires an additional if-then statement. The disjunctive `|` operator will always evaluate both operands, while the disjunctive short-circuit `||` operator will only evaluate the right-hand side of the expression if the left-hand side evaluates to `false`. Therefore, they are not interchangeable, especially if the right-hand side of the expression modifies a variable. For this reason, Option D is incorrect. Finally, Option E is correct. The logical complement `!` operator may only be applied to `boolean` expressions, not numeric ones.

32. C. Line 3 creates an empty `StringBuilder`. Line 4 adds three characters to it. Line 5 removes the first character resulting in `ed`. Line 6 deletes the characters starting at position 1 and ending right before position 1. Since there are no indexes that meet that description, the line has no effect. Therefore, Option C is correct.

33. A, D. Java methods must start with a letter, the dollar `$` symbol, or the underscore `_` character. For this reason, Option B is incorrect, and Options A and D are correct. Despite how Option A looks, it is a valid method signature in Java. Options C, E, and F do not compile because the symbols `-`, `\`, and `#` are not allowed in method names, respectively.

34. B, C. First off, Option A is incorrect, since whether or not `static` or inherited methods are chosen is a matter of design and individual preference. Options B and C are true statements about inheritance and two of the most important reasons Java supports inheritance. Option D is incorrect because all Java classes extend `java.lang.Object`. Option E is incorrect. Whether or not inheritance simplifies or complicates a design is based on the skills of the developer creating the application.

35. E. All of the statements are true statements about Java, making Option E the correct answer. Java was built with object-oriented programming and polymorphism in mind. Also, Java supports functional programming using lambda expressions.

36. C. This array has three elements, making `listing.length` output `3`. It so happens that each element references an array of the same size. But the code checks the first element and sees it is an array of size two, making the answer Option C.

37. A, B, E. A `switch` statement supports the primitive types `byte`, `short`, `char`, and `int` and their associated wrapper classes `Character`, `Byte`, `Short`, and `Integer`. It also supports the `String` class and enumerated types. Floating-point types like `float` and `double` are not supported, nor is the `Object` class. For these reasons, Options A, B, and E are correct.

38. D. The lambda syntax is incorrect. It should be `->`, not `=>`. Therefore, Option D is correct. If this was fixed, Option A would be correct.

39. B, C, E. The `/* */` syntax can have additional (and uneven) `*` characters in Java, making Options B and E correct. Option C is the standard way to comment a single line with two slashes `//`. Option A contains a `*/` in the middle of the expected comment, making the part after the comment `Insert **/` invalid. Option D is incorrect because a single slash `/` is not valid comment in Java. Finally, the `#` is not a comment character in Java, so Option F is incorrect.

40. A, F. A `static` import is used to import `static` members of another class. Option A is correct because the method `getGrass` and variable `seeds` are imported. Option F is also correct because a wildcard on the `Grass` class for all visible `static` members is allowed. Option B is incorrect because the wildcard must be on a class, not a package. Options C and E are incorrect since the keywords `import` and `static` are reversed. Option D is incorrect because the `static` keyword is missing.

41. D. When converting an array to a `List`, Java uses a fixed-sized backed list. This means that the list uses an array in the implementation. While changing elements to new values is allowed, adding and removing elements is not.

42. A, D. Variable names can begin with an underscore, making Option A correct. To use an underscore in a numeric literal, it must be between two digits, making Option D correct.

43. B. While no arguments are passed from the command line, this doesn't matter because the `main()` method redefines the `args` array. Remember that `String` values sort alphabetically rather than by number. Therefore, `01` sorts before `1`, and Option B is correct.

44. D. The `public` modifier allows access members in the same class, package, subclass, or even classes in other packages, while the `static` modifier allows access without an instance of the class. For these reasons, Option D is the correct answer. Option A is incorrect because `final` is not related to access, and package-private prevents access from classes outside the package. Option B is incorrect because `class` is not a modifier; it is a keyword. Option C is incorrect because `instance` is not a Java keyword or modifier, and `protected` prevents classes that are not subclasses and are outside the package from accessing the variable. Finally, Option E is incorrect. The `default` keyword is for interface methods and `switch` statements, not class variables.

45. A. Looping through the same list multiple times is allowed. Notice how there are not braces around the loops. This means that only the `print` statement is inside the loop. It executes four times. However, the `println()` only executes once at the end, making Option A the answer.

46. C, D. The `javac` command compiles a `.java` file into a `.class` bytecode file, making Option C a correct answer, while also making Options B, E, and F incorrect. The `javac` command compiles to a set of java instructions, or bytecode, not machine instructions, making Option A incorrect and Option D correct.

47. C. The `parseInt()` method returns an `int` primitive. Thanks to autoboxing, we can also assign it to an `Integer` wrapper class object reference. The `char` and `short` types are smaller than `int` so they cannot store the result. Therefore, lines 3 and 4 compile, and Option C is correct.

48. B, D, F. The compiler will broaden the data type on a numeric value until it finds a compatible signature. There are two versions of the `drive()` methods that return a value of `3`, one that takes a `short` and one that takes a `double`. Option A is incorrect because `boolean` cannot be converted to either of these types and trying to do so triggers a compiler error. Option B is correct because the data type `short` matches our message signature. Options C and E are incorrect. Remember that `int` and `long` are larger than `short` and will trigger different overloaded versions of `drive()` to be called, one that returns `5` and one that returns `2`, respectively. Option D is correct. The `byte` value can be implicitly converted to `short`, and there are no other matching method signatures that take a `byte` value. Finally, Option F is correct because `float` can be implicitly converted to `double`, and there is no other version of `drive()` that takes a `float` value.

49. A. Trick question. This appears to be about equality, but it is really about you recognizing that the `main()` method is missing the `static` keyword. Running this problem gives a runtime exception because the `main()` method is not properly declared. Therefore, Option A is the answer. If this was fixed, the answer would be Option C because the `int` and `String` comparisons return `true`.

50. D. The code compiles without issue, so Options E and F are incorrect. Note that line `p2` accesses a `static` method using an instance reference, which is discouraged but permitted in Java. First, a varargs `int` array of `[0,0]` is passed to the `swing()` method. The `try` block throws `ArrayIndexOutOfBoundsException`, since the third element is requested and the size of the array is two. For this reason, the `print()` statement in the `try` block is not executed. Next, since `ArrayIndexOutOfBoundsException` is a subclass of `RuntimeException`, the `RuntimeException catch` block is executed and `2` is printed. The rest of the `catch` blocks are skipped, since the first one was selected. The `finally` block then executes and prints `4`. Lastly, control is returned to the `main()` method without an exception being thrown, and `5` is printed. Since `245` is printed, Option D is the correct answer.

51. E. In the first iteration through the loop, `container` is 2 and `cup` is printed. Notice how the loop body subtracts 1 to account for indexes being zero based in Java. Then the update statement runs, setting `container` to 3. The condition is run and sees that 3 is in fact greater than 0. The loop body subtracts 1 and tries to get the element at index 2. There isn't one and the code throws an exception. This makes Option E correct. You might be tempted to think this is an infinite loop. If the body did not throw an exception, it would be!

52. A, E, F. An entry point in a Java application consists of a `main()` method with a single `String[]` or vararg `String...` argument, return type of `void`, and modifiers `public` and

`static`. Note that the name of the variable in the input argument does not matter and the `final` modifier is optional. Options A, E, and F match this description and are correct. Option B is incorrect because the argument is a single `String`. Option C is incorrect, since the access modifier is incorrectly marked `protected`. Finally, Option D is incorrect because it has two return types, `int` and `void`.

53. C, D, E. For this question, it helps to remember that the value of a `case` statement must be a literal expression or a `final` constant variable, and have a compatible data type. For these reasons, Lines 10 and 12 do not compile, making Options C and E correct answers. Line 10 uses a constant value, but `long` is not compatible with `switch` statements, while Line 12 uses a variable that is not marked `final`. Next, a `switch` statement may only have one `default` block. Therefore, Line 11 or 14 must be removed. Since Line 14 is not in the list of options, Option D becomes the last correct answer. The rest of the lines are fine since removing Lines 10, 11, and 12 allows the code to compile.

54. A, B, C. All of the compilation issues with this code involve access modifiers. First, all interface methods are implicitly `public`, and explicitly setting an interface method to `protected` causes a compilation error on line `h1`, making Option A correct. Next, lines `h2` and `h3` both override the interface method with the package-private access modifier. Since this reduces the implied visibility of `public`, the overrides are invalid and neither line compiles. Therefore, Options B and C are also correct. Note that the `RuntimeException` is allowed in an overridden method even though it is not in the parent method signature because only new checked exceptions in overridden methods cause compilation errors. Line `h4` is valid. An object can be implicitly cast to a superclass or inherited interface. Finally, lines `h5` and `h6` will compile without issue but independently throw a `ClassCastException` and a `NullPointerException` at runtime, respectively. Since the question only asks about compilation problems, neither of these are correct answers.

55. B, E, F. Unchecked exceptions inherit the `RuntimeException` class and are not required to be caught in the methods where they are declared. Since `ArithmeticException` and `IllegalArgumentException` extend `RuntimeException`, they are included as unchecked exceptions, making Options B, E, and F correct. The rest are checked exceptions, which inherit `Exception` but not `RuntimeException`.

56. F. The code compiles without issue, making Options D and E incorrect. Applying the ternary `? :` operator, the variable `ship` is assigned a value of `10.0`. The expression in the first if-then statement evaluates to `true`, so `Goodbye` is printed. Note that there is no `else` statement between the first and second if-then statements, therefore the second if-then statement is also executed. The expression in the second if-then statement evaluates to `false`, so the `else` statement is called and `See you again` is also printed. Therefore, Option F is the correct answer, with two statements being printed.

57. B, C, D. The `clock` variable is accessed by a class in the same package; therefore, it requires package-private or less restrictive access (`protected` and `public`). The `getTime()` method is accessed by a subclass in a different package; therefore, it requires `protected` or less restrictive access (`public`). Options B, C, and D conform to these rules, making them the correct answer. Options A and F cause the `Snooze` class to fail to compile because the `getTime()` method is not accessible outside the package, even though `Snooze` is a subclass of `Alarm`. Option E causes the `Coffee` class to fail to compile because the `clock` variable is only visible within the `Alarm` class.

58. B. This problem appears to be to be about overriding a method, but in fact, it is much simpler. The class `CarbonStructure` is not declared `abstract`, yet it includes an `abstract` method. To fix it, the definition of `CarbonStructure` would have to be changed to be an `abstract` class, or the `abstract` modifier would need to be removed from `getCount()` in `CarbonStructure` and a method body added. Since the only answer choice available is to change the `getCount()` method on line `q1`, Option B is the correct answer. Note that the rest of the application, including the override on line `q2`, is correct and compiles without issue. The return types `Long` and `Number` are covariant since `Number` is a superclass of `Long`. Likewise, the exception thrown in the subclass method is narrower, so no compilation error occurs on `q2`.

59. C. Line 5 does not declare a `main()` method that can be the entry point to the program. It does correctly declare a regular instance method and does compile. Line 6 does not compile because `LocalDate` needs to use a static method rather than a constructor. Line 7 is incorrect because `Period` methods should not be chained. However, it does compile, returning a period of 1 day. Line 8 does not compile because the correct class name is `DateTimeFormatter`. Line 9 is correct. Option C is correct because lines 6 and 8 do not compile.

60. A, E. A `try` block can have zero or more `catch` blocks, and zero or one `finally` blocks, but must be accompanied by at least one of these blocks. For these reasons, Options B, D, and F are incorrect, and Option E is correct. A `finally` block must appear after the last `catch` block, if there are any, making Option C incorrect, and Option A correct.

61. B. The code compiles without issue, so Option E is incorrect. For this problem, it helps to remember that `+` and `*` have a higher precedence than the ternary `? :` operator. In the first expression, `1 + 2 * 5` is evaluated first, resulting in a reduction to `11>=2 ? 4 : 2`, and then `fish` being assigned a value of `4`. In the second expression, the first ternary expression evaluates to false resulting in a reduction to the second right-hand expression `5>=5 ? 9 : 7`, which then assigns a value of `9` to `mammals`. In the `print()` statement, the first `+` operator is an addition operator, since the operands are numbers, resulting in the value of `4 + 9`, `13`. The second `+` operator is a concatenation since one of the two operands is a `String`. The result `13` is printed, making Option B the correct answer.

62. A, C, E. An object can be cast to a superclass or inherited interface type without an explicit cast. Furthermore, casting an object to a reference variable does not modify the object in any way; it just may change what methods and variables are immediately accessible. For these reasons, Options A, C, and E are correct. Option B is incorrect; since the compiler can try to block or warn about invalid casts, it cannot prevent them. For example, any object can be implicitly cast to `java.lang.Object`, then explicitly cast to any other object, leading to a `ClassCastException` at runtime. Option D is also incorrect because assigning an object to a subclass reference variable requires an explicit cast. Finally, Option F is incorrect. An object can always be cast to one of its inherited types, superclass or interface, without a `ClassCastException` being thrown.

63. F. The array is not sorted. It does not meet the pre-condition for a binary search. Therefore, the output is not guaranteed and the answer is Option F.

64. B. While `shoe3` goes out of scope after the `shopping()` method, the `croc` object is referenced by `shoe1` and therefore cannot be garbage collected. Similarly, the `sandal` object is now referenced by `shoe2`. No variables reference the `flip flop` object, so it is eligible to be garbage collected, and Option B is correct.

65. E. The `throws` keyword is used in method declarations, while the `throw` keyword is used to throw an exception to the surrounding process, and the `finally` keyword is used to add a statement that is guaranteed to execute even if an exception is thrown. For these reasons, Option E is the correct answer.

66. B, E. The first two iterations through the loop complete successfully, making Option B correct. However, the two arrays are not the same size and the `for` loop only checks the size of the first one. The third iteration throws an `ArrayIndexOutOfBoundsException`, making Option E correct.

67. E. For this question, it helps to try all answers out. Most of them do not make any sense. For example, overloading a method is not a facet of inheritance. Likewise, concrete and abstract methods can both be overridden, not just one. The only answer that is valid is Option E. Without virtual methods, overriding a method would not be possible, and Java would not truly support polymorphism.

68. E. The code does compile. Line `s1` is a bit tricky because `length` is used for an array and `length()` is used for a `String`. Line `s1` stores the length of the `Fall` in a variable, which is 4. Line `s2` throws an `ArrayIndexOutOfBoundsException` because 4 is not a valid index for an array with four elements. Remember that indices start counting with zero. Therefore, Option E is correct.

69. D. The code definitely does not compile, so Option A is incorrect. The first problem with this code is that the `Drum` class is missing a constructor causing the class declaration on line 8 to fail to compile. The default no-argument constructor cannot be inserted if the superclass, `Instrument`, does not define a no-argument constructor. The second problem with the code is that line 11 does not compile, since it calls `super.play(5)`, but the version of `play()` in the parent class does not take any arguments. Finally, line 15 does not compile. While `mn` may be a reference variable that points to a `Drum()` object, the `concert()` method cannot be called unless it is explicitly cast back to a `Drum` reference. For these three reasons, the code does not compile, and Option D is the correct answer.

70. B. The application compiles and runs without issue, so Options E and F are incorrect. Java uses pass-by-value, so even though the change to `length` in the first line of the `adjustPropellers()` method does not change the value in the `main()` method, the value is later returned by the method and used to reassign the `length` value. The result is that `length` is assigned a value of `6`, due to it being returned by the method. For the second parameter, while the `String[]` reference cannot be modified to impact the reference in the calling method, the data in it can be. Therefore, the value of the first element is set to `LONG`, resulting in an output of `6,LONG`, making Option B the correct answer.

71. D. The first compilation problem with the code is that the second `catch` block in `openDrawbridge()` is unreachable since `CableSnapException` is a subclass of `OpenDoorException`. The `catch` blocks should be ordered with the more narrow exception classes before the broader ones. Next, the variable `ex` is declared twice within the same scope since it appears in the second `catch` block as well as the embedded try-catch block. Finally, the `openDrawbridge()` method declares the checked `Exception` class, but it is not handled in the `main()` method with a try-catch block, nor in the `main()` method declaration. For these three reasons, Option D is correct.

72. D. Object orientation is the property of structuring an object with its related data and methods. Encapsulation is the property of removing direct access to the underlying data from processes outside the class. The two go hand and hand to improve class design, making Option D the correct choice.

73. E. In Java, `String` is a class and not a primitive. This means it needs to begin with an uppercase letter in the declaration. The code does not compile, making Option E correct. If this was fixed, the answer would be Option B.

74. A. This class is called with three command-line arguments. First the array is sorted, which meets the pre-condition for binary search. At this point, the array contains `[flower, plant, seed]`. The key is to notice the value of `args[0]` is now `flower` rather than `seed`. Calling binary search to find the position of `flower` returns `0`, which is the index matching that value. Therefore, the answer is Option A.

75. B, C, D. A `for`-each loop is a specialized loop that just iterates through an array or list. It can be rewritten using explicit indexing code in any of the other three loop types. Therefore, Options B, C, and D are correct. Option A is incorrect because a `do-while` loop is guaranteed to execute at least once. Option E is incorrect because the traditional `for` loop can loop backwards or by skipping indexes. Option F is incorrect because non-index-related `boolean` conditions are allowed to be used in a `while` loop.

76. E. The `LocalDate` class is only for day/month/year values. It does not support time, so `getHour()` and `plusHours()` do not compile, making Option E the answer.

77. C. All arrays are objects regardless of whether they point to primitives or classes. That means both `balls` and `scores` are objects. Both are set to `null` so they are eligible for garbage collection. The `balls` array is initialized to all `null` references. There are no objects inside. The `scores` array is initialized to all 0 values. Therefore, only two objects exist to be eligible for garbage collection, and Option C is correct.

78. B. Since there are not brackets around the `while` loop, only line 17 is in the loop body. Line 18 gets executed once after the loop completes. This means that `count` will be 1 assuming the loop completes. Subtracting a month from `JANUARY` results in `DECEMBER`. Since the loop completes E is incorrect and Option B is the answer. Note that if the brackets were added as the indentation suggests, Option D would be the answer since we are counting months backwards.

79. D. Line 10 does not compile because the override reduces the visibility of an inherited method, with the package-private modifier being more restrictive than the `protected` modifier. Line 11 does also not compile, since the left-hand side of a compound assignment operator must be used with a variable, not a method. Finally, Line 12 does not compile because `super.grunt()` is inherited as an `abstract` method in the `PolarBear` class, meaning the parent class has no implementation. For these three reasons, Option D is the correct answer.

80. B, E. Package-private, or default, access is denoted by the absence of an access modifier, making Option A incorrect. Option B is correct, since a `switch` statement can contain a `default` execution path. Options C and F are incorrect because keywords in Java cannot be used as method or variable names. Finally, interfaces can contain `default` interface methods but they must be concrete with a method body. For this reason, Option E is correct and Option D is incorrect.

Chapter 11: Java Class Design

1. D. The `toString()` method is declared in the `Object` class. Therefore it is available to be called in any Java class and is overridden in some. Java automatically calls the `toString()` method when you print an object, making Option D correct. Option C is incorrect because `toString()` is a method, not a variable.

2. B. This code is not a singleton because it has a `public` constructor. Remember that a `public` no-argument constructor is provided automatically if no constructor is coded. This code is well encapsulated because the instance variable is `private`. It is not immutable since there is a setter method. Therefore, Option B is correct.

3. C. The singleton pattern ensures there will be no more than one instance of the object. Depending on how it is implemented, it is possible for there to be zero instances. But it is not possible to have more than one, making Option C correct. Option D means the variable is shared across instances or even without an instance being created but does not limit the number of the instances of the class itself.

4. C. Both objects are instances of the class `Laptop`. This means the `startup()` method in the `Laptop` class gets called both times thanks to polymorphism.

5. D. We know that the variable `o` that `equals()` is called on isn't `null`, since we can't call instance methods on a `null` reference. However, a `null` reference could be passed as a method parameter. If a `null` is passed in, the method should return `false` since an object and a `null` are not equal. Options A and B are incorrect because the first line of those methods should return `false` rather than `true`. Option C is incorrect because the cast is missing. The `Object` class does not have a `text` variable available. Option D shows a properly implemented `equals()` method and is correct.

6. A. Option A is correct because mutability means the state can change and immutability means it cannot. In Option C, static means the state isn't tied to an instance. In Option B, rigidity is not a common programming term.

7. B. The `Hammer` class is a subclass of the `Tool` class. Since the `use()` method in `Hammer` is intended to override the one in `Tool`, there are certain rules. One is that the access modifier must not be more specific. Therefore, trying to make it `private` is a problem. Option B is correct and `r2` is the only line with a compiler error in this code.

8. D. The singleton pattern requires that only one instance of the class exist. Neither of these classes meets that requirement since they have the default no-argument constructor available. There should have been a `private` constructor in each class. Therefore, Option D is correct. Remember that the exam doesn't always include import statements to simplify the code you need to read.

9. B. While using `null` with `instanceof` compiles, it always returns `false`. The other two `instanceof` calls show that `instanceof` can be used with both classes and interfaces. They both return `true`, making Option B correct.

10. D. The `static` keyword is used to create a class-level variable, making Option D correct. Note that a singleton is where you limit a class so only one instance can be created. This means there are not multiple instances to share a variable across.

11. A. Option A is a requirement of a singleton class rather than an immutable one. The other three options are requirements of an immutable class.

12. C. If the variables are `public`, the class is not encapsulated because callers have direct access to them. This rules out Options A and B. Having `private` methods doesn't allow the callers to use the data, making Option D an undesirable answer. Option C is correct and the classic definition of encapsulation where the data is not exposed directly.

13. A. While both objects are instances of `Laptop`, we are not calling methods in this example. Virtual method invocation only works for methods, not instance variables. For instance variables, Java actually looks at the type of the reference and calls the appropriate variable. This makes each reference call a different class's instance variable in this example, and Option A is correct.

14. B. An immutable class must not allow the state to change. In the `Flower` class, the caller has a reference to the `List` being passed in and can change the size or elements in it. Similarly, any class with a reference to the object can get the `List` by calling `get()` and make these changes. The `Flower` class is not immutable. The `Plant` class shows how to fix these problems and is immutable. Option B is correct.

15. C. An instance method can access both instance variables and `static` variables. Both methods compile and Option C is correct.

16. B. A `static` method can access `static` variables, but not instance variables. The `getNumRakes()` method does not compile, so Option B is correct.

17. A. You are allowed to use `null` with `instanceof`; it just prints `false`. The `bus` variable is both a `Vehicle` and a `Bus`, so lines 18 and 19 print `true`. Then it gets interesting. We know that `bus` is not an `ArrayList` or `Collection`. However, the compiler only knows that `bus` is not an `ArrayList` because `ArrayList` is a concrete class. Line 20 does not compile. The compiler can't definitively state that `bus` is not a `Collection`. Some future program could create a subclass of `Bus` that does implement `Collection`, so this line compiles. Therefore, only line 20 fails to compile, and Option A is correct.

18. B. `Building` and `House` are both properly declared inner classes. Any `House` object can be stored in a `Building` reference, making the declarations for `p` and `r` compile. The declaration for `s` is also correct. It so happens that `bh` is a `House` object, so the cast works. The declaration of `q` is a problem though. While the cast itself is fine, a `Building` cannot be stored in a `House` reference, which means the assignment fails to compile. Option B is correct and is the only line with a compiler error in this code. Note that if the declaration of `q` was removed, the declaration of `p` would produce a `ClassCastException` at runtime.

19. D. If two instances of a class have the same hash code, they might or might not be equal. The reverse is not true. If two objects are equal, they must have the same hash code in order to comply with the contracts of these methods. However, in this case, the answer is none of the above because the method can't simply return `true` or `false`. Based on the rules of `equals()`, if `null` is passed in, the result must be `false`. If an object identity is passed in, the result must be `true` due to reflexivity. As a result, Option D is correct.

20. D. This class is a good example of encapsulation. It has a `private` instance variable and is accessed by a `public` method. No changes are needed to encapsulate it, and Option D is correct.

21. B. The singleton pattern requires that only one instance of the class exist. The `ExamAnswers` class is close. However, `getExamAnswers()` is not `static`, so you can't retrieve the instance. Option B is the answer because `TestAnswers` is a correct implementation. It has a `static` variable representing the one instance and a `static` method to retrieve it.

22. C. The `static` initializer is only run once. The `static` method is run twice since it is called twice. Therefore, three lines are printed, and Option C is correct.

23. C. Option A is allowed because the `turnOn()` method is `public` and can be called from anywhere. Options B and D are allowed since the method is in the same class, which is always allowed! Option C is not allowed because `wash()` is a package-private method in another package. Option C is the correct answer.

24. B. The `display()` method has `protected` access. This means it can be accessed by instance methods in the same package and any subclasses. There are no subclasses in this example, so we only need to count the classes in the same package. Option B is correct because `Flashlight` and `Phone` are in the package.

25. B. Line 15 calls the method on line 9 since it is a `Watch` object. That returns `watch`, making Option A incorrect. Line 16 calls the method on line 3 since it is a `SmartWatch` object and the method is properly overridden. That returns `smart watch`, so Option B is the answer, and Option C is incorrect.

26. A. Clearly a `Bus` is a `Vehicle` since the `Bus` class implements `Vehicle`. The `Van` class is also a `Vehicle` since it extends `Bus`. This question also confirms you know that arrays can be tested with `instanceof`, which they can. Therefore, Option A is correct.

27. C. There is no `instanceOf` keyword, making Options B and D incorrect. There is an `instanceof` keyword. If an object is the wrong type, the `equals()` method should return `false`, making Option C the answer.

28. D. The `Hammer` class is a subclass of the `Tool` class. Luckily, the `use()` method has a different signature so it is not an override. This means it is fine that the access modifier is stricter, and Option D is correct. Line `r3` is a valid method unrelated to the superclass.

29. B. Lazy instantiation is part of a possible implementation for the singleton pattern. It defers creating the object until the first caller requests it. While this does save memory, it only does so if the object is never requested. This does not save memory when actually creating the object. Option B is correct.

30. D. Notice how the code begins at line 30. This means you have to infer the surrounding code. Here it is reasonable to assume the classes are inner classes. `Building` and `House` are defined correctly. Any `House` or `Building` reference can potentially be a `House`. The compiler does not know which ones work and which don't. This means all three casts compile.

31. C. Encapsulation doesn't allow callers access to the instance variables, which makes it easier to change the code. The instance variables can be any type, which means they can be mutable or immutable. There are not constraints on the implementation of methods. The purpose of encapsulation is to lessen how tightly tied or coupled the classes are. Option C is the opposite of this, making it the answer.

32. A. An immutable class must not allow the state to change. The `Flower` class does this correctly. While the class isn't `final`, the getters are, so subclasses can't change the value returned. The `Plant` class lacks this protection, which makes it mutable. Option A is correct.

33. D. A `static` initializer is not allowed inside of a method. It should go on the class level rather than the method level. Therefore, the code does not compile, and Option D is correct.

34. A. An object is required to have the same value for repeated calls to `hashCode()` if the value has not changed. This makes III and IV incorrect. If two objects are equal, they are required to have the same hash code. Since equality must be reflexive, it cannot return `false` if the same object is passed, and I is incorrect. Since `equals()` must return `false` when `null` is passed in, it cannot be `true` and II is incorrect. Therefore, Option A is the answer.

35. D. By definition, you cannot change the value of an instance variable in an immutable class. There are no setter methods, making Option A incorrect. While Option B would allow you to set the value, the class would no longer be immutable. Option D is correct. If you are an advanced developer, you might know that you can use reflection to change the value. Don't read into questions like this on the exam. Reflection isn't on the exam, so you can pretend it doesn't exist.

36. B. Option A is incorrect because the "is-a" principle is about inheritance. For example, a `String` is an `Object`. Option C is incorrect because singletons require a `static` variable to ensure there is only one instance. While it is common to have instance variables as well, this is not required to implement the pattern. Option B is correct. For an object to be composed of other objects, instance variables are required.

37. B. The `static` initializer only runs once since statics are shared by all instances. The instance initializer runs twice because we call the constructor twice. Therefore, Option B is correct.

38. A. While there is a `default` keyword in Java, it is only allowed in interfaces or in `switch` statements. It is not a visibility modifier. The author of this code probably intended for the method to be package-private, which doesn't use a visibility modifier. The line with `default` doesn't compile, so Option A is correct. If `default` was removed, the code would all compile.

39. A. The reference `b` points to a `Building` object, which cannot be stored in a `House` reference. This means the assignment to `p` compiles but fails at runtime. The other two casts would run without issue if the code got that far.

40. C. The `hashCode()` method in the `Object` class does not have a parameter. Therefore, the `Sticker` class provides an overloaded method rather than an overridden one. Since it is not an overridden method, the contract for the `Object` class' `hashCode()` method does not apply, and any `int` value can be returned. Therefore, Option C is correct.

Chapter 12: Advanced Java Class Design

1. B. The lambda expression `s -> true` is valid, making Options A, C, and D incorrect. Parentheses `()` are not required on the left-hand side if there is only one variable. Brackets `{}` are not required if the right-hand side is a single expression. Parameter data types are only required if the data type for at least one parameter is specified, otherwise none are

required. The remaining choice, the arrow operator `->`, is required for all lambda expressions, making Option B the correct answer.

2. D. The application contains a compilation error. The `case` statements incorrectly use the enum name as well as the value, such as `DaysOff.ValentinesDay`. Since the type of the enum is determined by the value of the variable in the `switch` statement, the enum name is not allowed and throws a compilation error when used. For this reason, Option D is correct. If the enum name `DaysOff` was removed, the application would output `12`, since the lack of any `break` statements causes multiple blocks to be reached, and Option C would have been the correct answer.

3. C. A functional interface must include exactly one `abstract` method, either by inheritance or declared directly. It may also have any number, including zero, of `default` or `static` methods. For this reason, both parts of Option D are incorrect. The first part of Option A is incorrect because more than one `abstract` method disqualifies it as a functional interface. The first part of Option B is incorrect because the method must be `abstract`; that is to say, any method will not suffice. Finally, Option C is the correct answer. The first part of the sentence defines what it means to be a functional interface. The second part refers to the optional `@FunctionalInterface` annotation. It is considered a good practice to add this annotation to any functional interfaces you define because the compiler will report a problem if you define an invalid interface that does not have exactly one `abstract` method.

4. C. While an anonymous inner class can extend another class or implement an interface, it cannot be declared `final` or `abstract` since it has no class definition. For this reason, Option C is correct. The other classes may be declared `final` or `abstract` since they have a class definition.

5. B. Option A is incorrect because the lambda expression is missing a semicolon (`;`) at the end of the `return` statement. Option C is incorrect because the local variable `test` is used without being initialized. Option D is also incorrect. The parentheses are required on the left-hand side of the lambda expression when there is more than one value or a data type is specified. Option B is the correct answer and the only valid lambda expression.

6. B. An enum cannot be marked `abstract`, nor can any of its values, but its methods can be marked `abstract`, making Option B the correct answer. Note that if an enum contains an `abstract` method, then every enum value must include an override of this `abstract` method.

7. B. The code compiles without issue, so Option D is incorrect. The first `print()` statement refers to `value` in the `Deeper` class, so `5` is printed. The second and third `print()` statements actually refer to the same value in the `Deep` class, so `2` is printed twice. The prefix `Matrix.` is unnecessary in the first of the two `print()` statements and does not change the result. For these reasons, Option B is the correct answer.

8. D. A local inner class can access `final` or effectively final local variables, making Option D the correct answer. The second statement is invalid because access modifiers like `private` cannot be applied to local variables.

9. C. The type of the variable in the `switch` statement is the enum `Currency`, but the `case` statements use `int` values. While the `enum` class hierarchy does support an `ordinal()` method, which returns an `int` value, the enum values cannot be compared directly with `int` values. For this reason, the code does not compile, since the `case` statement values are not compatible with the variable type in the `switch` statement, making Option C the correct answer.

10. C. A local variable is effectively final when it's primitive value or object reference does not change after it is initialized, making Option C the correct answer. Option D is incorrect. Any change to the variable after it is initialized disqualifies it for being considered effectively final.

11. D. Both the `Drive` and `Hover` interfaces define a `default` method `getSpeed()` with the same signature. In fact, both `getSpeed()` methods return the same value of `5`. The class `Car` implements both interfaces, which means it inherits both `default` methods. Since the compiler does not know which one to choose, the code does not compile, and the answer is Option D. Note that if the `Car` class had overridden the `getSpeed()` method, then the code would have compiled without issue and printed `10` at runtime. In particular, the local class `Racecar` defined in the `main()` method compiles without issue, making Option C incorrect.

12. B. An interface can be extended by another interface and a class can be extended by another class, making the second part of Options A, C, and D incorrect. Option B is correct because an enum cannot be extended. Note that Option C is also incorrect for this reason.

13. B. If the program is called with a single input `WEST`, then `WEST` would be printed at runtime. If the program is called with no input, then the `compass` array would be of size zero, and an `ArrayIndexOutOfBoundsException` would be thrown at runtime. Finally, if the program is called with a string that does not match one of the values in `Direction`, then an `IllegalArgumentException` would be thrown at runtime. The only result not possible is `south`, since the enum value is in uppercase, making Option B the correct answer.

14. B. Enumerated types support creating a set of reusable values whose values are fixed and consistent across the entire application. For these reason, Options A, C, and D are incorrect. Option B is the false statement because enum values are defined at compile time and cannot be changed or added at runtime.

15. D. The program contains three compilation problems. First off, the enum `Color` extends the enum `Light`, but enums cannot extend other enums so the definition is invalid. Second, the enum value list must end with a semicolon (`;`) if the enum definition contains anything other than the enum values. Since it includes a constructor, a semicolon (`;`) is required after `GREEN`. Finally, enum constructors must be `private`, meaning the `protected` constructor for `Color` does not compile. For these three reasons, Option D is the correct answer.

16. D. Both abstract classes and interfaces can include `static` methods, so Options A and C are incorrect. A static nested class can include `static` methods, but it is the only type of inner class in which this is allowed. Local inner classes, anonymous inner classes, and member inner classes do not support `static` methods. For these reasons, Option D is correct, and Option B is incorrect.

17. B. A functional interface must contain exactly one `abstract` method. The `Bend` interface contains two `abstract` methods, `pump()` and `bend()`, since it extends `Pump` and inherits `pump()`. For this reason, the `Bend` method is not a valid functional interface and therefore cannot be used as a lambda expression, making Option B the correct answer. The rest of the code compiles without issue. Note that the usage of an instance variable to call a `static` method, `r.apply()` in the `main()` method, is permitted but discouraged.

18. C. Applying the `@Override` annotation is optional and not required to override a method or implement an interface method, making Options A and B incorrect. While partially helpful as a form of documentation, it is not the best reason to apply the annotation, making Option D

incorrect. The best reason is that the compiler will actually fail to compile if the method that the `@Override` annotation is being applied to is not actually overriding an inherited method. This behavior helps correct typos or changes in superclasses or interfaces that could break the class or lead to unexpected behavior. For this reason, Option C is the best choice.

19. C. The `Bottle` class includes a static nested class `Ship` that must be instantiated in a static manner. Line `w2` uses an instance of `Bottle` to instantiate the `Ship`. While this would be allowed if `Ship` was a member inner class, since it is a static nested class, line `w2` does not compile, and Option C is the correct answer. Note that if `Ship` was changed to be a member inner class, the code would still not compile since a member inner class cannot include static members and enums are inherently static. Therefore, the correct change would be to fix the declaration on line `w2`.

20. A. Option A is the invalid lambda expression because the type is specified for the variable `j`, but not the variable `k`. The rest of the options are valid lambda expressions. To be a valid lambda expression, the type must be specified for all of the variables, as in Option C, or none of them, as in Options B and D.

21. D. This application declares an anonymous inner class that implements the `Edible` interface. Interface methods are `public`, whereas the override in the anonymous inner class uses package-private access. Since this reduces the visibility of the method, the declaration of `eat()` on line 8 does not compile. Next, the declaration of the `apple` object must end with a semicolon (`;`) on line 11, and it does not. For these two reasons, the code does not compile, and Option D is the correct answer. Note that if these two issues were corrected, with the `public` modifier and missing semi-colon (`;`), then the correct answer would be Option A because the code does not actually call the `eat()` method; it just declares it.

22. A. The code compiles without issue and prints `15`, making Option A correct and Option D incorrect. The `main()` method defines a local class `Oak` that correctly extends `Tree`, a static nested class, making Option B incorrect. Finally, the method `getWater()` is permitted to read the variable `water`, defined in the `main()` method, since it is effectively final, having a value of `15` when it is defined. For this reason, Option C is also incorrect.

23. C. Interfaces allow Java to support multiple inheritance because a class may implement any number of interfaces. On the other hand, an anonymous inner class may implement at most one interface, since it does not have a class definition to implement any others. For these reasons, Option C is the correct answer.

24. A. The code does not compile because the declaration of `isDanger()` in the class `SeriousDanger` is an invalid method override. An overridden method may not throw a broader checked exception than it inherits. Since `Exception` is a superclass of `Problem`, thrown by the inherited method in the `Danger` class, the override of this checked exception is invalid. For this reason, line `m1` does not compile, and Option A is the correct answer. The rest of the lines of code compile without issue.

25. B. Options A, C, and D are true statements about interfaces and abstract classes. Option B is the correct answer because neither abstract classes nor interfaces can be marked `final`. For Option D, methods and variables can both be marked `private` in abstract classes in some cases. The "some cases" refers to the fact that the `private` modifier cannot be applied to `abstract` methods, since a method cannot be marked both `final` and `abstract`. Since abstract classes can contain concrete methods, which can take the `private` access modifier, the statement is true.

26. A. The code compiles without issue, so Option C is incorrect. Enum ordinal values are indexed starting with zero, so `0` is printed first. The second line compiles and runs without issue, with `flurry` being converted to `FLURRY`, using the `toUpperCase()` method. Since there is a matching enum named `FLURRY`, that value is printed next. For these reasons, Option A is the correct answer.

27. D. Java was updated to include `default` interface methods in order to support backward compatibility of interfaces. By adding a `default` method to an existing interface, we can create a new version of the interface, which can be used without breaking the functionality of existing classes that implement an older version of the interface. For this reason, Option D is the correct answer. Options A and C are not applicable to `default` interface methods, whereas Option B could be achieved by using `static` interface methods.

28. C. The `Penguin` class includes a member inner class `Chick`. Member inner classes cannot include `static` methods or variables. Since the variable `volume` is marked `static`, the member inner class `Chick` does not compile, making Option C the correct answer. Note that the variable `volume` referenced in the `chick()` method is one defined in the `Penguin` outer class. If the `static` modifier was removed from the `volume` variable in the `Chick` class, then the rest of the code would compile and run without issue, printing `Honk(1)!` at runtime.

29. D. Member inner classes require an instance of the surrounding class to be instantiated. Option A is incorrect since we are told that the instantiation request is from a `static` method. Note that this call would be valid from a non-`static` method in `Dinosaur`. Option B is incorrect because it lacks the `new` keyword. Option C is incorrect. `Pterodactyl` is a member inner class, not a static nested class. Option D is correct and uses the instance `dino` to create a new `Pterodactyl` object.

30. C. First off, both `CanBurrow` and `HasHardShell` are functional interfaces since they contain exactly one `abstract` method, although only the latter uses the optional `@FunctionalInterface` annotation. The declarations of these two interfaces, along with the `abstract` class `Tortoise`, compile without issue, making Options A and B incorrect. The code does not compile, though, so Option D is incorrect. The class `DesertTortoise` inherits two `abstract` methods, one from the interface `CanBurrow` and the other from the abstract parent class `Tortoise`. Since the class only implements one of them and the class is concrete, the class declaration of `DesertTortoise` fails to compile on line `k3`, making Option C the correct answer.

31. B. First off, the two interface definitions contain identical methods, with the `public` modifiers assumed in all interfaces methods. For the first statement, the `write()` method is marked `default` in both interfaces, which means a class can only implement both interfaces if the class overrides the `default` method with its own implementation of the method. Since the `Twins` method does override `write()`, the method compiles without issue, making the first statement incorrect. Next, the `publish()` method is marked `static` in both interfaces and the `Twins` class. While having a `static` method in all three is allowed, marking a `static` method with the `@Override` annotation is not because only member methods may be overridden. For this reason, the second statement is correct. Finally, the `think()` method is assumed to be `abstract` in both interfaces since it doesn't have a `static` or `default` modifier and does not define a body. The `think()` method is then correctly overridden with a concrete implementation in the

`Twins` class, making the third statement incorrect. Since only the second statement was true, Option B is the correct answer.

32. D. An enum and static inner class can define `static` methods, making Option D the correct answer. Options A, B, and C are incorrect because the other types of inner classes cannot define `static` methods. Note that interfaces and abstract classes can define `static` methods.

33. C. First off, Option A does not compile since the variables `p` and `q` are reversed, making the return type of the method and usage of operators invalid. The first argument `p` is a `String` and `q` is an `int`, but the lambda expression reverses them, and the code does not compile. Option B also does not compile. The variable `d` is declared twice, first in the lambda argument list and then in the body of the lambda expression. The second declaration in the body of the lambda expression causes the compiler to generate a duplicate local variable message. Note that other than it being used twice, the expression is valid; the ternary operator is functionally equivalent to the `learn()` method in the `BiologyMaterial` class. Option C is the correct answer since it compiles and handles the input in the same way as the `learn()` method in the `BiologyMaterial` class.

34. C. The code does not compile since it contains two compilation errors, making Option A incorrect. First, the enum list is not terminated with a semicolon (`;`). A semicolon (`;`) is required anytime an enum includes anything beyond just the list of values, such as a constructor or method. Second, the access modifier of `TRUE`'s implementation of `getNickName()` is package-private, but the `abstract` method signature has a `protected` modifier. Since package-private is a more restrictive access than `protected`, the override is invalid and the code does not compile. For these two reasons, Option C is the correct answer. Note that the `@Override` annotation is optional in the method signature, therefore `FALSE`'s version of `getNickName()` compiles without issue. Also, note that the `Proposition` constructor does not include a `private` access modifier, but the constructor compiles without issue. Enum constructors are assumed to be `private` if no access modifier is specified, unlike regular classes where package-private is assumed if no access modifier is specified.

35. A. The code compiles and runs without issue, printing `8` at runtime, making Option A correct and Option D incorrect. The `AddNumbers` interface is a valid functional interface. While it includes both `static` and `default` methods, it only includes one `abstract` method, the precise requirement for it to be considered a functional interface, making Option B incorrect. Finally, Option C is incorrect because the lambda expression is valid and used correctly.

36. A. While this code included a large number of `final` modifiers, none of them prevent the code from compiling when a valid expression is placed in the blank, making Option D incorrect. Option B is incorrect since it returns the `size` variable defined in the `Insert` member inner class, not the `Bottle` class, printing `25` at runtime. Option C is incorrect because the expression is invalid and does not compile when inserted into the blank. Finally, Option A is the correct answer because it compiles, properly references the variable `size` in the `Bottle` class, and prints `14` at runtime.

37. C. The `main()` method attempts to define an anonymous inner class instance but fails to provide the class or interface name, or use the `new` keyword. The right-hand side of the assignment to the `seaTurtle` variable should start with `new CanSwim()`. For this reason, Option C is the correct answer. If the code was corrected with the proper declaration, it would output `7`, and Option B would be the correct answer.

38. D. The code does not compile, so Options A and B are incorrect. The declarations of the local inner classes `Robot` and `Transformer` compile without issue. The anonymous inner class that extends `Transformer` compiles without issue, since the `public` variable `name` is inherited, making Option C incorrect. The only compilation problem in this class is the last line of the `main()` method. The variable `name` is defined inside the local inner class and not accessible outside class declaration without a reference to the local inner class. Due to scope, this last line of the `main()` method does not compile, making Option D the correct answer. Note that the first part of the `print()` statement in the `main()` method, if the code compiled, would print `GiantRobot`.

39. B. The `Dancer` class compiles without issue, making Option A incorrect. The `SwingDancer` class, though, does not compile because `getPartner()` is an invalid method override. In particular, `Leader` and `Follower` are not covariant since `Follower` is not a subclass of `Leader`. Therefore, line `u2` does not compile, making Option B correct and Option D incorrect. Note that the `abstract` method `getPartner(int)` is not implemented in `SwingDancer`, but this is valid because `SwingDancer` is an `abstract` class and is not required to implement all of the inherited abstract methods.

40. C. The code does not compile, so Options A and B are incorrect. The problem here is that the `DEFAULT_VALUE` is an instance variable, not a `static` variable; therefore, the static nested class `GetSet` cannot access it without a reference to the class. For this reason, the declaration of the `static` nested class `GetSet` does not compile, and Option C is the correct answer. The rest of the code compiles without issue. Note that if the `DEFAULT_VALUE` variable was modified to be `static`, then the code would compile without issue, and Option B would be the correct answer.

Chapter 13: Generics and Collections

1. C. When declaring a class that uses generics, you must specify a name for the formal type parameter. Java uses the standard rules for naming a variable or class. A question mark is not allowed in a variable name, making I incorrect. While it is common practice to use a single uppercase letter for the type parameter, this is not required. It certainly isn't a good idea to use existing class names like the `News` class being declared here or the `Object` class built into java. However, this is allowed, and Option C is correct.

2. B. Option A is incorrect because the `filter()` method is available on `Stream`, but not `List`. Option C is incorrect because the `replace()` method is available on `List`, but not `Stream`. Option D is tricky because there is a `sort()` method on `List` and a `sorted()` method on `Stream`. These are different method names though, so Option D is incorrect. Option B is the answer because both interfaces have a `forEach()` method.

3. A. Notice how there is unnecessary information in this description. The fact that patrons select books by name is irrelevant. The checkout line is a perfect example of a double-ended queue. We need easy access to one end of the queue for patrons to add themselves to the queue. We also need easy access to the other end of the queue for patrons to get off the queue when it is their turn. The book lookup by ISBN is a lookup by key. We need

a map for this. A `HashMap` is probably better here, but it isn't a choice. So the answer is Option A, which does include both a double-ended queue and a map.

4. B. Java talks about the collections framework, but the `Map` interface does not actually implement the `Collection` interface. `TreeMap` has different methods than `ArrayDeque` and `TreeSet`. It cannot fill in the blank, so Option B is correct.

5. B. Options C and D are incorrect because the method signature is incorrect. Unlike the `equals()` method, the method in `Comparator` takes the type being compared as the parameters when using generics. Option A is a valid `Comparator`. However, it sorts in ascending order by length. Option B is correct. If `s1` is three characters and `s2` is one character, it returns `-2`. The negative value says that `s1` should sort first, which is correct, because we want the longest `String` first.

6. D. `TreeMap` and `TreeSet` keep track of sort order when you insert elements. `TreeMap` sorts the keys and `TreeSet` sorts the objects in the set. This makes Option D correct. Note that you have the option of having `JellyBean` implement `Comparable`, or you can pass a `Comparator` to the constructor of `TreeMap` or `TreeSet`.

7. C. Option A is incorrect because a pipeline still runs if the source doesn't generate any items and the rest of the pipeline is correct. Granted some of the operations have nothing to do, but control still passes to the terminal operation. Option B is incorrect because intermediate operations are optional. Option C is the answer. The terminal operation triggers the pipeline to run.

8. B. The `Iterator` interface uses the `hasNext()` and `next()` methods to iterate. Since there is not a `hasMore()` method, it should be changed to `hasNext()`, making Option B the answer. With respect to Option A, the missing generic type gives a warning, but the code still runs. For Option C, iterators can run as many times as you want, as can the `forEach()` method on `list`.

9. A. First the code creates an `ArrayList` of three elements. Then the list is transformed into a `TreeSet`. Since sets are not allowed to have duplicates, the set only has two elements. Remember that a `TreeSet` is sorted, which means that the first element in the `TreeSet` is `3`. Therefore, Option A is correct.

10. C. The word *reduction* is used with streams for a terminal operation, so Options A and B are incorrect. Option D describes a valid terminal operation like `anyMatch()`, but is not a reduction. Option C is correct because a reduction has to look at each element in the stream in order to determine the result.

11. A. The `offer()` method adds an element to the back of the queue. After line 7 completes, the queue contains `18` and `5` in that order. The `push()` method adds an element to the front of the queue. How rude! The element `13` pushes past everyone on the line. After line 8 completes, the queue now contains `13`, `18`, and `5`, in that order. Then we get the first two elements from the front, which are `13` and `18`, making Option A correct.

12. D. The `Magazine` class doesn't implement `Comparable<Magazine>`. It happens to implement the `compareTo()` method properly, but it is missing actually writing `implements Comparable`. Since `TreeSet` doesn't look to see if the object can be compared until runtime, this code throws a `ClassCastException` when `TreeSet` calls `add()`, so Option D is correct.

13. C. Line 8 does not compile. `String::new` is a constructor reference. A constructor or method reference is equivalent to a lambda. It participates in deferred execution. When it is executed later, it returns a `String`. It does not return a `String` on line 8. It actually returns a `Supplier<String>`, which cannot be stored in `list`. Since the code does not compile, Option C is correct.

14. B. This code adds two elements to a list. It then gets a stream and iterates through the list, printing two lines. The last line does the same thing again. Since a fresh stream is created, we are allowed to iterate through it, and Option B is correct.

15. D. The `Comic` interface declares a formal type parameter. This means that a class implementing it needs to specify this type. The code on line 21 compiles because the lambda reference supplies the necessary context making Option A incorrect. Option B declares a generic class. While this doesn't tell us the type is `Snoopy`, it punts the problem to the caller of the class. The declaration of `c2` on line 22 compiles because it supplies the type, making Option B incorrect. The code on line 23 compiles because the `SnoopyClass` itself supplies the type making Option C incorrect. Option D has a problem. `SnoopyClass` and `SnoopyComic` appear similar. However, `SnoopyComic` refers to `C`. This type parameter exists in the interface. It isn't available in the class because the class has said it is using `Snoopy` as the type. Since the `SnoopyComic` class itself doesn't compile, the line with `c4` can't instantiate it, and Option D is the answer.

16. A. In streams, the `filter()` method filters out any values that do not match. This means the only value to make it to the terminal operator `count()` is `Chicago`, and Option A is correct.

17. C. When implementing `Comparable`, you implement the `compareTo()` method. Since this is an instance method, it already has a reference to itself and only needs the item it is comparing. Only one parameter is specified, and Option C is correct. By contrast, the `Comparator` interface uses the `compare()` method and the method takes two parameters.

18. C. The source and any intermediate operations are chained and eventually passed to the terminal operation. The terminal operation is where a non-stream result is generated, making Option C correct.

19. A. A constructor reference uses the `new` keyword where a method name would normally go in a method reference. It can implicitly take zero or one parameters just like a method reference. In this case, we have one parameter, which gets passed to the constructor. Option A is correct.

20. D. A custom sort order is specified using a `Comparator` to sort in descending order. However, this `Comparator` is not passed when searching. When a different sort order is used for searching and sorting, the result is undefined. Therefore, Option D is correct.

21. D. Java only allows you to operate on a stream once. The final line of code throws an `IllegalStateException` because the stream has already been used up. Option D is the correct answer.

22. D. The `Wash` class takes a formal type parameter named `T`. Option C shows the best way to call it. This option declares a generic reference type that specifies the type is `String`. It also

uses the diamond syntax to avoid redundantly specifying the type on the right-hand side of the assignment. Options A and B show that you can omit the generic type in the reference and still have the code compile. You do get a compiler warning scolding you for having a raw type. But compiler warnings do not prevent compilation. With the raw type, the compiler treats `T` as if it is of type `Object`. That is OK in this example because the only method we call is `toString()` implicitly when printing the value. Since `toString()` is defined on the `Object` class, we are safe, and Options A and B work. Since all three can fill in the blank, Option D is the answer.

23. D. The missing generic type gives a warning, but the code still runs, so Option A is incorrect. The `Iterator` interface uses `hasNext()` and `next()` methods to iterate, so Option B is incorrect. Option C applies to calling the same stream twice. One of our calls is to an `Iterator` anyway, so Option C is incorrect. This code is in fact correct, making the answer Option D.

24. B. This is a `static` method reference. It uses `::` to separate the class name and method name. Option B is correct.

25. B. A source and the terminal operation are required parts of a stream pipeline and must occur exactly once. The intermediate operation is optional. It can appear zero or more times. Since more than once falls within zero or more, Option B is correct.

26. B. `ArrayList` allows `null` elements, making Option B correct. `TreeSet` does not allow nulls because they need to compare the values. `ArrayDeque` uses `null` for a special meaning, so it doesn't allow it in the data structure either.

27. D. Option A is the only one of the three options to compile. However, it results in no lines being output since none of the three strings are empty. Options B and C do not even compile because a method reference cannot have an operator next to it. Therefore, Option D is correct.

28. A. Unfortunately you do have to memorize two facts about sort order. First, numbers sort before letters. Second, uppercase sorts before lowercase. Since `TreeMap` orders by key, the first key is `3` and the last is `three`, making Option A correct.

29. C. The `?` is an unbounded wildcard. It is used in variable references but is not allowed in declarations. In a `static` method, the type parameter specified inside the `<>` is used in the rest of the variable declaration. Since it needs an actual name, Options A and B are incorrect. We need to specify a type constraint so we can call the `add()` method. Regardless of whether the type is a class or interface, Java uses the `extends` keyword for generics. Therefore, Option D is incorrect, and Option C is the answer.

30. B. On a stream, the `filter()` method only keeps values matching the lambda. The `removeIf()` does the reverse on a `Collection` and keeps the elements that do not match. In this case, that is `Austin` and `Boston` so Option B is correct.

31. D. The code correctly creates an `ArrayDeque` with three elements. The stream pipeline does compile. However, there is no terminal operation, which means the stream is never evaluated and the output is something like `java.util.stream.ReferencePipeline$2@404b9385`. This is definitely not one of the listed choices, so Option D is correct.

32. C. The `forEach()` method that takes one parameter is defined on the `Collection` interface. However, a map is not a `Collection`. There is a version of `forEach()` defined on the `Map` interface, but it uses two parameters. Since two parameters can't be used with a method reference, Option C is the answer.

33. C. This code is almost correct. Calling two different streams is allowed. The code attempts to use a method reference when calling the `forEach()` method. However, it does not use the right syntax for a method reference. A double colon needs to be used. The code would need to be changed to `System.out::println` to work and print two lines for each call. Since it does not compile, Option C is correct.

34. B. This code shows a proper implementation of `Comparable`. It has the correct method signature. It compares the magazine names in alphabetical order. Remember that uppercase letters sort before lowercase letters. Since `Newsweek` is uppercase, Option B is correct.

35. C. The `filter()` method requires a `boolean` returned from the lambda or method reference. The `getColor()` method returns a `String` and is not compatible. This causes the code to not compile and Option C to be correct.

36. A. Option A is correct as the source and terminal operation are mandatory parts of a stream pipeline. Option B is incorrect because a `Stream` must return non-primitives. Specialized interfaces like `IntStream` are needed to return primitives. Option C is incorrect because `Stream` has methods such as `of()` and `iterate()` that return a `Stream`. Option D is incorrect because infinite streams are possible.

37. B. The stream pipeline is correct and filters all values out that are 10 characters or smaller. Only `San Francisco` is long enough, so `c` is `1`. The `stream()` call creates a new object, so stream operations do not affect the original list. Since the original list is still 3 elements, Option B is correct.

38. B. Options A and C are incorrect because a generic type cannot be assigned to another direct type unless you are using upper or lower bounds in that statement. Now, we just have to decide whether a lower or upper bound is correct for the `T` formal type parameter in `Wash`. The clue is that the method calls `size()`. This method is available on `Collection` and all classes that extend/implement it. Therefore, Option B is correct.

39. C. A `Comparator` takes two parameters, so Options A and B are incorrect. Option D doesn't compile. When using brackets, a `return` keyword and semicolon are required. Option C is a correct implementation.

40. B. Option D is incorrect because there is a `charAt()` instance method. While Option C is correct that the method takes in an `int` parameter, autoboxing would take care of conversion for us if there were no other problems. So Option C is not the answer. Option A is not true because there are constructor and instance method references. Option B is the answer. With method references, only one item can be supplied at runtime. Here, we need either a `String` instance with no parameters in the method or a static method with a single parameter. The `charAt()` method is an instance method with a single parameter so does not meet this requirement.

Chapter 14: Lambda Built-in Functional Interfaces

1. C. The `Supplier` functional interface does not take any inputs, while the `Consumer` functional interface does not return any data. This behavior extends to the primitive versions of the functional interfaces, making Option C the correct answer. Option A is incorrect because `IntConsumer` takes a value, while `LongSupplier` returns a value. Options B and D are incorrect because `Function` and `UnaryOperator` both take an input and produce a value.

2. A. The `LongSupplier` interface does not take any input, making Option D incorrect. It also uses the method name `getAsLong()`. The rest of the functional interfaces all take a `long` value but vary on the name of the `abstract` method they use. `LongFunction` contains `apply()` and `LongPredicate` contains `test()`, making Options B and C, respectively, incorrect. That leaves us with `LongConsumer`, which contains `accept()`, making Option A the correct answer.

3. A. The code compiles without issue, so Options C and D are incorrect. The value for distance is 2, which based on the lambda for the `Predicate` will result in a `true` expression, and `Saved` will be printed, making Option A correct.

4. C. Both are functional interfaces in the `java.util.function` package, making Option A true. The major difference between the two is that `Supplier<Double>` takes the generic type `Double`, while the other does not take any generic type and instead uses the primitive `double`. For this reason, Options B and D are true statements. For `Supplier<Double>` in Option B, remember that the returned `double` value can be implicitly cast to `Double`. Option C is the correct answer. Lambdas for `Supplier<Double>` can return a `null` value since `Double` is an object type, while lambdas for `DoubleSupplier` cannot; they can only return primitive `double` values.

5. B. The lambda `(s,p) -> s+p` takes two arguments and returns a value. For this reason, Option A is incorrect because `BiConsumer` does not return any values. Option D is also incorrect, since `Function` only takes one argument and returns a value. This leaves us with Options B and C, which both use `BiFunction`, which takes two generic arguments and returns a generic value. Option C is incorrect because the datatype of the unboxed sum `s+q` is `int` and `int` cannot be autoboxed or implicitly cast to `Double`. Option B is correct. The sum `s+p` is of type `double`, and `double` can be autoboxed to `Double`.

6. C. To begin with, `ToDoubleBiFunction<T,U>` takes two generic inputs and returns a `double` value. Option A is compatible because it takes an `Integer` and `Double` and returns a `Double` value that can be implicitly unboxed to `double`. Option B is compatible because `long` can be implicitly cast to `double`. While we don't know the data types for the input arguments, we know that some values, such as using `Integer` for both, will work. Option C cannot be assigned and is the correct answer because the variable `v` is of type `Object` and `Object` does not have a `length()` method. Finally, Option D is compatible. The variable `y` could be declared `double` in the generic argument to the functional interface, making `y/z` a `double` return value.

7. C. The `BiPredicate` interface takes two generic arguments and returns a `boolean` value. Next, `DoubleUnaryOperator` takes a `double` argument and returns a `double` value. Last, `ToLongFunction` takes a generic argument and returns a `long` value. That leaves Option C, which is the correct answer. While there is an `ObjDoubleConsumer` functional interface, which takes a generic argument and a `double` value and does not return any data, there is no such thing as `ObjectDoubleConsumer`. Remember that `Object` is abbreviated to `Obj` in all functional interfaces in `java.util.function`.

8. C. The code does not compile, so Options A and D are incorrect. The `IntUnaryOperator` functional interface is not generic, so the argument `IntUnaryOperator<Integer>` in the `takeTicket()` does not compile, making Option C the correct answer. The lambda expression compiles without issue, making Option B incorrect. If the generic argument `<Integer>` was dropped from the argument declaration, the class would compile without issue and output `51` at runtime, making Option A the correct answer.

9. A. Option A is the correct answer because `BiPredicate` takes two generic types and returns a primitive `boolean` value. Option B is incorrect, since `CharSupplier` does not exist in `java.util.function`. Option C is also incorrect, since `LongFunction` takes a primitive `long` value and returns a generic type. Remember, Java only includes primitive functional interfaces that operate on `double`, `int`, or `long`. Finally, Option D is incorrect. `UnaryOperator` takes a generic type and returns a generic value.

10. D. First off, the `forEach()` method requires a `Consumer` instance. Option C can be immediately discarded because `Supplier<Double>` does not inherit `Consumer`. For this same reason, Option B is also incorrect. `DoubleConsumer` does not inherit from `Consumer`. In this manner, primitive functional interfaces cannot be used in the `forEach()` method. Option A seems correct, since `forEach()` does take a `Consumer` instance, but it is missing a generic argument. Without the generic argument, the lambda expression does not compile because the expression `p<5` cannot be applied to an `Object`. The correct functional interface is `Consumer<Double>`, and since that is not available, Option D is the correct answer.

11. C. `BiFunction<Double,Double,Double>` and `BinaryOperator<Double>` both take two `Double` input arguments and return a `Double` value, making them equivalent to one another. On the other hand, `DoubleFunction<Double>` takes a single `double` value and returns a `Double` value. For this reason, it is different from the other two, making Option C correct and Option D incorrect.

12. B. `BinaryOperator<Long>` takes two `Long` arguments and returns a `Long` value. For this reason, Option A, which takes one argument, and Option D, which takes two `Integer` values that do not inherit from `Long`, are both incorrect. Option C is incorrect because the local variable `c` is re-declared inside the lambda expression, causing the expression to fail to compile. The correct answer is Option B because `intValue()` can be called on a `Long` object. The result can then be cast to `long`, which is autoboxed to `Long`.

13. C. The program does not compile, so Option A is incorrect. The `Supplier` functional interface normally takes a generic argument, although generic types are not strictly required since they are removed by the compiler. Therefore, line `d1` compiles while triggering a compiler warning, and Options B and D are incorrect. On the other hand, line `d2` does cause a compiler error, because the lambda expression does not return a value. Therefore, it is not compatible with `Supplier`, making Option C the correct answer.

14. A. The input type of a unary function must be compatible with the return type. By compatible, we mean identical or able to be implicitly cast. For this reason, Option A is the correct answer. Option B is incorrect since all of the `UnaryOperator` functional interfaces, generic or primitive, take exactly one value. Option C is incorrect because the primitive functional interfaces do not take a generic argument. Finally, Option D is incorrect. For example, the generic `UnaryOperator<T>` returns an `Object` that matches the generic type.

15. C. Remember that all `Supplier` interfaces take zero parameters. For this reason, the third value in the table is `0`, making Options A and B incorrect. Next, `DoubleConsumer` and `IntFunction` each take one value, `double` and `int`, respectively. On the other hand, `ObjDoubleConsumer` takes two values, a generic value and a `double`, and returns `void`. For this reason, Option C is correct, and Option D is incorrect.

16. D. All `Consumer` functional interfaces have a `void` return type. For this reason, the first and last values in the table are both `void`, making Options A and B incorrect. `IntFunction` takes an `int` and returns a generic value, while `LongSupplier` does not take any values and returns a `long` value. For this reason, Option C is incorrect, and Option D is correct.

17. B. The `removeIf()` method requires a `Predicate` since it operates on a `boolean` result, making Option A incorrect. The `forEach()` method takes a `Consumer` and does not return any data, making Option B correct, and Options C and D incorrect.

18. C. The code does not compile, so Option A is incorrect. The lambda expression compiles without issue, making Option B incorrect. The `task` variable is of type `UnaryOperator<Doll>`, with the abstract method `apply()`. There is no `accept()` method defined on that interface, therefore the code does not compile, and Option C is the correct answer. If the code was corrected to use the `apply()` method, the rest of it would compile without issue. At runtime, it would then produce an infinite loop. On each iteration of the loop, a new `Doll` instance would be created with `5`, since the post-decrement (`--`) operator returns the original value of the variable, and that would make Option D the correct answer.

19. C. To begin with, `Consumer` uses `accept()`, making Option A incorrect. Next, `Function` and `UnaryOperator` use `apply()`, making Options B and D, respectively, incorrect. Finally, `Supplier` uses `get()`, making Option C the correct answer.

20. D. First off, Options A and B are incorrect because the second functions for both return a `double` or `Double` value, respectively. Neither of these values can be sent to a `UnaryOperator<Integer>` without an explicit cast. Next, Option C is incorrect. The first functional interface `Function<Double,Integer>` takes only one input, but the diagram shows two inputs for the first functional interface. That leaves us with Option D. The first functional interface `BiFunction<Integer,Double,Integer>` takes an `int`, which can be implicitly autoboxed to `Integer`, and a `Double` and returns an `Integer`. The next functional interface, `BinaryOperator<Integer>`, takes two `Integer` values and returns an `Integer` value. Finally, this `Integer` value can be implicitly unboxed and sent to `IntUnaryOperator`, returning an `int`. Since these behaviors match our diagram, Option D is the correct answer.

21. D. Options A, B, and C are true statements about functional interfaces. A lambda may be compatible with multiple functional interfaces, but it must be assigned to a functional interface when it is declared or passed as a method argument. Also, a method can be created with the return type that matches a functional interface, allowing a lambda expression to be returned. Option D is the correct answer. Deferred execution means the lambda expression is not evaluated until runtime, but it is compiled. Compiler errors in the lambda expression will prevent the code from compiling.

22. B. Option A is incorrect because the `String "3"` is not compatible with the return type `int` required for `IntSupplier`. Option B is the correct answer. Although this will result in a divide by zero issue at runtime, the lambda is valid and compatible with `IntSupplier`. Option C is incorrect because the lambda expression is invalid. The `return` statement is only allowed inside a set of brackets `{}`. Finally, Option D is incorrect. The method reference is used for `Supplier`, not `Consumer`, since it takes a value and does not return anything.

23. C. The lambda expression is invalid because the input argument is of type `Boss`, and `Boss` does not define an `equalsIgnoreCase()` method, making Option C the correct answer. If the lambda was corrected to use `s.getName()` instead of `s`, the code would compile and run without issue, printing `[JENNY, GRACE]` at runtime and making Option A the correct answer.

24. D. First of all, `Consumer<Object>` takes a single `Object` argument and does not return any data. The classes `ArrayList` and `String` do not contain constructors that take an `Object`, so neither of the first two statements are correct. The third statement does support an `Object` variable, since the `System.out.println(Object)` method exists. For these reasons, Option D is the correct answer.

25. B. The `java.util.function` package does not include any functional interfaces that operate on the primitive `float`, making Option A incorrect. Remember, Java only includes primitive functional interfaces that operate on `double`, `int`, or `long`. Option B is correct because it is a valid functional interface. Option C is incorrect because there is no `UnaryIntOperator` functional interface. Note that there is one called `IntUnaryOperator`. Option D is incorrect. The `java.util.function` package does not include any tri- operators, although many are easy to write.

26. D. A lambda expression can match multiple functional interfaces. It matches `DoubleUnaryOperator`, which takes a `double` value and returns a `double` value. Note that the data type of `s+1` is `double` because one of the operands, in this case `s`, is `double`. It also matches `Function<String,String>` since the `(+)` operator can be used for `String` concatenation. Finally, it matches `IntToLongFunction` since the `int` value `s+1`can be implicitly cast to `long`. On the other hand, the lambda expression is not compatible with `UnaryOperator` without a generic type. When `UnaryOperator` is used without a generic argument, the type is assumed to be `Object`. Since the `(+)` operator is not defined on `Object`, the code does not compile due to the lambda expression body, making Option D the correct answer. Note that if the lambda expression did not rely on the `(+)` operator, such as `s -> s`, then `UnaryOperator` would be allowed by the compiler, even without a generic type.

27. B. The `BiFunction` interface takes two different generic values and returns a generic value, taking a total of three generic arguments. Next, `ToDoubleFunction` takes exactly one

generic value and returns a `double` value, requiring one generic argument. The `ToIntBiFunction` interface takes two generic values and returns an `int` value, for a total of two generic arguments. For these reasons, Options A, C, and D are incorrect. The correct answer is Option B. `DoubleFunction` takes a `double` value and returns a generic result, taking exactly one generic argument, not two.

28. D. While lambda expressions can use primitive types as arguments, the functional interface in this class uses the wrapper classes, which are not compatible. For this reason, Option A is incorrect. Option B is also incorrect, since the number of arguments and return type does not match the functional interface. Furthermore, the method reference `System.out::print` on the right-hand side of the lambda expression is invalid here, since it returns a method reference, not a `double` value. Option C is incorrect because `2*w` is of type `double`, which cannot be returned as an `Integer` without an explicit cast. Option D is the correct answer. It takes exactly two arguments because the return value `int` can be implicitly autoboxed to `Integer`.

29. A. `BooleanSupplier` is the only functional interface that does not involve `double`, `int`, or `long`, making Option A the correct answer. The rest of the functional interfaces are not found in `java.util.function`. Java does not have built-in support for primitive functional interfaces that include `char`, `float`, or `short`.

30. D. The code does not compile because the lambda expression `p -> p*100` is not compatible with the `DoubleToIntFunction` functional interface. The input to the functional interface is `double`, meaning `p*100` is also `double`. The functional interface requires a return value of `int`, and since `double` cannot be implicitly cast to `int`, the code does not compile, making Option D the correct answer. If the correct cast was applied to make `(p*100)` an `int`, then the rest of the class would compile and `250` would be printed at runtime, making Option B correct.

31. B. The `ToDoubleFunction` interface takes a generic value, not a `double` value, making Option D incorrect. It also uses the method name `accept()`. The rest of the functional interfaces all take a `double` value. `DoubleConsumer` contains the `accept()` method, making Option A incorrect. `DoublePredicate` contains the `test()` method, making Option B the correct answer. Finally, `DoubleUnaryOperator` contains the `applyAsDouble()` method, making Option C incorrect.

32. D. To start with, line 5 does not compile because `Function` takes two generic arguments, not one. Second, the assignment statement on line 7 does not end with a semicolon (`;`), so it also does not compile. Finally, the `forEach()` method on line 10 requires a `Consumer`, not a `Function`, so this line does not compile. For these three reasons, Option D is the correct answer.

33. D. The `DoubleToLongFunction` interface takes a `double` argument and returns a `long` value. Option A is compatible since the `int` value `1` can be implicitly cast to `long`, and `2L` is already a `long`. Option B is also compatible, since the `double` value `10.0*e` is explicitly cast to `int` then implicitly cast to `long`. Next, Option C is compatible because an explicit cast of the `double` to a `long` value is used. Option D cannot be assigned and is the correct answer. Although the `Double` class does have a `longValue()` method, the left-hand side of the lambda expression must use the primitive `double`, not the wrapper `Double`. This lambda expression violates the signature of the functional interface, since it allows `Double` values to be sent to the interface, including those that could be `null`.

34. C. The `DoublePredicate` interface takes a `double` value and returns a `boolean` value. `LongUnaryOperator` takes a `long` value and returns a `long` value. `ToIntBiFunction` takes two generic values and returns an `int` value. The only choice that is not an existing functional interface is `ShortSupplier`. Recall that Java only includes primitive functional interfaces that operate on `double`, `int`, or `long`. For this reason, Option C is the correct answer.

35. A. The method reference `System.out::println` takes a single input and does not return any data. `Consumer<Sheep>` is compatible with this behavior, making Option A the correct answer and Option D incorrect. Option B is incorrect because `void` cannot be used as a generic argument. Option C is incorrect since `System.out::println()` does not return any data and `UnaryOperator` requires a return value.

36. C. The code does not compile, making Options A and B incorrect. The local variable `MAX_LENGTH` is neither `final` nor effectively final, meaning it cannot be used inside the lambda expression. The `MAX_LENGTH` variable starts off with an initial value of `2`, but then is modified with the increment assignment (`+=`) operator to a value of `5`, disqualifying its ability to be considered effectively final by the compiler. Since the lambda does not compile, Option C is the correct answer. If the code was rewritten so that the `MAX_LENGTH` variable was marked `final` and assigned a value of `5` from the start, then it would output `2`, and Option A would be correct.

37. B. To begin with, all of the functional interfaces in the list of choices take two values. The difference is in the name of the method they use. `BiConsumer` uses `accept()`, making Option A incorrect. Option B is correct because `BiFunction` includes the `apply()` method. Option C is incorrect, since `BiPredicate` uses the `test()` method. `DoubleBinaryOperator` is almost correct but the name of the method is `applyAsDouble()`, not `apply()`, making Option D incorrect. For the exam, you should be aware of which primitive functional interfaces use a different method name than the generic ones.

38. B. To start with, `IntFunction<Integer>` takes an `int` value and returns an `Integer`. The first statement uses `Integer` instead of `int` as the input argument and is therefore not compatible. The second statement is compatible, since the return type `null` can be used as an `Integer` return type. The last statement is also valid. An `int` can be autoboxed to `Integer`. For these reasons, Option B is the correct answer.

39. C. The primitive `Supplier` functional interfaces, such as `BooleanSupplier` and `LongSupplier`, do not have a `get()` method. Instead, they have methods such as `getAsBoolean()` and `getAsLong()`, respectively. For this reason, the first line of the `checkInventory()` method does not compile, making Option C the correct answer. If the method call was changed to `getAsBoolean()`, then the rest of the code would compile without issue, print `Plenty!` at runtime, and Option A would be the correct answer.

40. B. Java only supports a single return data type or `void`. Therefore, it is not possible to define a functional interface that returns two data types, making Option A incorrect. Although Java does not include built-in support for primitive functional interfaces that include `float`, `char`, or `short`, there is nothing to prevent a developer from creating them in their own project, making Option B the true statement and the correct answer. Option C is incorrect because a functional interface that takes no values and returns `void` is possible. In fact, `Runnable` is one such example. Option D is also incorrect, since `IntFunction<R>` takes a primitive argument as input and a generic argument for the return type.

Chapter 15: Java Stream API

1. D. Option A is incorrect because it doesn't print out one line. The `peek()` method is an intermediate operation. Since there is no terminal operation, the stream pipeline is not executed, so the `peek()` method is never executed. Options B and C are incorrect because they correctly output one line using a method reference and lambda, respectively, and don't use any bad practices. Option D is the answer. It does output one line. However, it is bad practice to have a `peek()` method that has side effects like modifying a variable.

2. A. This code generates an infinite stream of integers: 1, 2, 3, 4, 5, 6, 7, etc. The `Predicate` checks if the element is greater than 5. With `anyMatch()`, the stream pipeline ends once element 6 is hit and the code prints `true`. For both the `allMatch()` and `noneMatch()` operators, they see that the first element in the stream does not match and the code prints `false`. Therefore, Option A is correct.

3. B. Only the `average()` method returns an `OptionalDouble`. This reflects that it doesn't make sense to calculate an average when you don't have any numbers. By contrast, counting without any numbers gives the `long` number `0` and summing gives the `double` number `0.0`. Since only one method matches the return type, Option B is correct.

4. C. The `map()` method can fill in the blank. The lambda converts a `String` to an `int` and Java uses autoboxing to turn that into an `Integer`. The `mapToInt()` method can also fill in the blank and Java doesn't even need to autobox. There isn't a `mapToObject()` in the stream API. Note there is a similarly named `mapToObj()` method on `IntStream`. Since both `map()` and `mapToInt()` work here, Option C is correct.

5. D. The `average()` method returns an `OptionalDouble`. This interface has a `getAsDouble()` method rather than a `get()` method, so the code does compile. However, the stream is empty, so the optional is also empty. When trying to get the value, the code throws a `NoSuchElementException`, making Option D correct.

6. D. Option A is incorrect because `anyMatch()` returns a `boolean`. Option B is incorrect because `filter()` is an intermediate operation, not a terminal operation, and therefore returns a `Stream`. Both of these methods do take a `Predicate` as a parameter. While `findAny()` does return an `Optional`, it doesn't take any parameters. Therefore, Option C is incorrect, and Option D is the answer.

7. B. This code builds a list with two elements. It then uses that list as a source for the stream, sorts the stream as it goes by, and grabs the first sorted element. This does not change the original list. The first element in the sorted stream is `1.2`, but the first element of `list` remains as `5.4`. This makes Option B correct.

8. B. Primitive streams, like `LongStream`, declare an `average()` method, while summary statistics classes, like `LongSummaryStatistics`, declare a `getAverage()` method, making Options C and D incorrect. The `average()` method returns an `OptionalDouble` object, which declares a `getAsDouble()` method rather than a `get()` method. Therefore, Option A is incorrect, and Option B is correct.

9. B. Since the result of the `collect()` is not stored in a variable or used in any way, all the code needs to do is compile. There is no `Collectors.toArrayList()` method. If you want to specify an `ArrayList`, you can call `Collectors.toCollection(ArrayList::new)`. The `Collectors.toList()` method does in fact exist and compile. While there is a `Collectors.toMap()` method, it requires two parameters to specify the key and value functions, respectively. Since only one can compile, Option B is correct.

10. C. As tempting as it is, you can't actually convert a `Map` into a `Stream` directly, which means you can't call the `map()` method on it either. However, you can build a `Stream` out of the keys or values or key/value pairs. Since this code doesn't compile, Option C is correct.

11. D. I is incorrect because `isPresent()` returns `false` for an empty `Optional`. II is incorrect because `of()` throws a `NullPointerException` if you try to pass a null reference. III doesn't throw an exception as the `ofNullable()` is designed to allow a null reference. However, it returns `false` because no value is present. Since none of the choices are correct, Option D is the answer.

12. A. This code does compile. Remember that imports are implied, including the `static` import for `Collectors`. The collector tries to use the number of characters in each stream element as the key in a map. This works fine for the first two elements, `speak` and `bark`, because they are of length 5 and 4, respectively. When it gets to `meow`, it sees another key of 4. The merge function says to use the first one, so it chooses `bark` for the value. Similarly, `growl` is 5 characters, but the first value of `speak` is used. There are only two distinct lengths, so Option A is correct.

13. C. For the primitive stream that contains the `int` primitives, the interface names are incorrect. They should be `IntStream` and `IntSummaryStatistics`, making Option C correct. If this was fixed, Option B would be the answer.

14. B. This code does compile. As an intermediate operation, you are allowed to call `peek()` many times in a stream pipeline. You can even call it multiple times in a row. While it is common to write `System.out::println` directly as a parameter to `peek()`, nothing prevents you from creating a `Consumer` variable. Since the `forEach()` method also takes a `Consumer`, we can reuse it. The three `peek()` intermediate operations and one `forEach()` operation total four lines of output. The `map()` operation could be omitted since it simply passes the input through.

15. B. `Character` objects are allowed in a `Stream`, so line `z1` compiles, making Option C incorrect. Line `z2` also compiles since `findAny()` returns an `Optional` and `ifPresent()` is declared on `Optional`. Therefore, Option D is also incorrect. Now let's look at the `Stream`. The source has three elements. The intermediate operation sorts the elements and then we request one from `findAny()`. The `findAny()` method is not guaranteed to return a specific element. Since we are not using parallelization, it is highly likely that the code will print `a`. However, you need to know this is not guaranteed, making Option B the answer.

16. A. The `sorted()` method takes an optional `Comparator` as the parameter, which takes two `String` parameters and returns an `int`. Option A is correct because the lambda implements this interface. Option B is incorrect because the method reference doesn't take any parameters, nor does it return an `int`.

17. D. The `Optional` class has an `isPresent()` method that doesn't take any parameters. It returns a `boolean` and is commonly used in `if` statements. There is also an `ifPresent()` method that takes a `Consumer` parameter and runs it only if the `Optional` is non-empty. The methods `isNotNull()` and `forEach()`are not declared in `Optional`. Therefore, Option D is correct.

18. C. The first intermediate operation, `limit(1)`, turns the infinite stream into a stream with one element: `true`. The `partitioningBy()` method returns a map with two keys, `true` and `false`, regardless of whether any elements actually match. If there are no matches, the value is an empty list, making Option C correct.

19. B. The `flatMap()` method is used to turn a stream of streams into a one-dimensional stream. This means it gets rid of the empty `list` and flattens the other two. Option A is incorrect because this is the output you'd get using the regular `map()` method. Option B is correct because it flattens the elements. Notice how it doesn't matter that all three elements are different types of `Collection` implementations.

20. D. The `sorted()` method allows an optional `Comparator` to be passed as a reference. However, `Comparator.reverseOrder()` does not implement the `Comparator` interface. It takes zero parameters instead of the required two. Since it cannot be used as a method reference, the code does not compile, and Option D is correct.

21. D. Option A is incorrect because the `findAny()` might not return `1`. The result could be any of the three numbers. Option B is incorrect because there is no `first()` method available as a terminal operation. Option C is tempting because there is a `min()` method. However, since we are working with a `Stream`, this method requires a `Comparator` as a parameter. Therefore, Option D is the answer.

22. C. `List` doesn't have a `filter()` method, so Option A is incorrect. `Stream` does have `filter()` and `map()` methods. However, `Stream` doesn't have an `ifPresent()` method. This makes IV incorrect, so Options B and D are incorrect. Both `Collection` and `String` have an `isEmpty()` method, so either can be used with the `Optional`, making Option C the answer.

23. D. This code generates an infinite stream of the number 1. The `Predicate` checks if the element is greater than 5. This will never be true. With `allMatch()`, the stream pipeline ends after checking the first element. It doesn't match, so the code prints `false`. Both `anyMatch()` and `noneMatch()` keep checking and don't find any matches. However, they don't know if a future stream element will be different, so the code executes infinitely until the process is terminated. Therefore, Option D is correct.

24. D. Both `Collectors.groupingBy()` and `Collectors.partitioningBy()` are useful for turning a stream into a `Map`. The other two methods do not exist. However, when using a condition, you should use `partitioningBy()` as it automatically groups using a `Boolean` key. Therefore, Option D is correct.

25. B. Option A is incorrect because we are working with primitives rather than objects. Option C compiles but outputs the stream references rather than the contents. Option B is correct because it flattens the `int` primitives into one stream.

26. D. The summary statistics classes provide getters in order to access the data. The `getAverage()` method returns a `double` and not an `OptionalDouble`. Option D is the only option to compile.

27. D. Option A doesn't compile because the `get()` method on `Optional` doesn't take any parameters. Options B and C do compile, but both print `Cupcake` since the `Optional` is not empty. Therefore, Option D is correct.

28. C. The first line generates an infinite stream. The stream pipeline has a filter that lets all these elements through. Since `sorted()` requires all the elements be available to sort, it never completes, making Option C correct.

29. A. The `mapToDouble()` method compiles. However, it converts `9` into `9.0` rather than the single digit `9`. The `mapToInt()` method does not compile because a `long` cannot be converted into an `int` without casting. The `mapToLong()` method is not available on `LongStream` so it does not compile. It is available on `DoubleStream`, `IntStream`, and `Stream` implementations. Since none of the options outputs the single digit `9`, Option A is correct.

30. A. The `filter()` method either passes along a given element or doesn't, making Option D incorrect. The `flatMap()` method doesn't pass along any elements for empty streams. For non-empty streams, it flattens the elements, allowing it to return zero or more elements. This makes Option B incorrect. Finally, the `map()` method applies a one-to-one function for each element. It has to return exactly one element, so Option A is the correct answer.

31. D. First, we sort the stream. Option B is incorrect because `findFirst()` is guaranteed to return the first element. However, the `findFirst()` method returns an `Optional`. Therefore, the output of this code is `Optional[a]` rather than the letter `a`, making Option D correct.

32. C. There is not a stream pipeline method called `sort()`. There is one called `sorted()`. Since the code does not compile, Option C is correct. If this was fixed, Option A would be correct since the `Comparator` sorts in ascending order.

33. B. This code compiles. It creates a stream of `Ballot` objects. Then it creates a map with the contestant's name as the key and the sum of the scores as the value. For `Mario`, this is 10 + 9, or `19`, so Option B is correct.

34. D. Both `anyMatch()` and `allMatch()` take a `Predicate` as a parameter. This code does not compile because the parameter is missing.

35. D. The `flatMap()` method works with streams rather than collections. The code does not compile because the `x` is not a stream, making Option D correct. If this was fixed, Option B would be the answer.

36. C. The `groupingBy()` collector always returns a `Map` (or a specific implementation class of `Map`), so III can't be right. The other two are definitely possible. To get I, you can group using a `Function` that returns an `Integer` such as `s.collect(Collectors.groupingBy(String::length))`. To get II, you need to group using a `Function` that returns a `Boolean` and specify the type, such as `s.collect(Collectors.groupingBy(String::isEmpty, Collectors.toSet()))`. Notice that autoboxing is used for both. Therefore, Option C is correct.

37. D. There is no built-in method to map a value to a `boolean` primitive. Therefore, Options B and C don't even compile, so they are incorrect. Option A does compile as it maps a `Runner` to a `Boolean`. However, it doesn't actually `filter()` the stream to eliminate any values, so the output is not the same. It prints `3` instead of `1`. None of these are correct, making Option D the answer.

38. A. Option A is the answer because there is a `getCount()` method that returns a `long` rather than a method named `getCountAsLong()`. Option B is incorrect because there is in fact a `getMax()` method. Option C is incorrect because `toString()` is declared on `Object`, which means it is inherited by all classes.

39. C. The `main()` method has warnings, but it does compile, making Option D incorrect. The warnings are both about not declaring the generic type for `Optional`. Option A does not compile because the `orElse()` method expects an `Exception` as the alternate value passed as a parameter. `IllegalArgumentException::new` is a `Supplier` instead. Options B and C both compile as both methods expect a `Supplier` as the parameter. However, `orElseGet()` simply returns the exception from the method rather than throwing it. Option C actually throws the exception the `Supplier` created and is the correct answer.

40. B. Option A happens to output the same result for both pairs. It outputs a blank line in `withFlatMap()` because empty streams are removed and in `withoutFlatMap()` because the `filter()` method removes the empty list. Option B outputs different results. The `withFlatMap()` method outputs `lastall queued up` since it flattens the streams. By contrast, the `withoutFlatMap()` method outputs `[last, all queued up]` since it leaves the structure intact. Since the output is different. Option B produces different results so it is the answer.

Chapter 16: Exceptions and Assertions

1. D. If no exception is thrown, then the `catch` block will not be executed. The `try` block is always visited first, followed by the `finally` block, which is guaranteed to execute regardless of whether an exception is thrown. For these reasons, Option D is the correct answer, with the statements in the correct order.

2. C. Unlike a try-with-resources statement, in which the `catch` and `finally` blocks are optional, a `try` statement requires a `catch` or `finally` block to be used, or both. For this reason, Option C is the correct answer.

3. D. The code does not compile because the `throw` keyword is incorrectly used in the `toss()` method declaration. The keyword `throws` should have been used instead. For this reason, Option D is the correct answer. Since `LostBallException` inherits `Throwable` and the `main()` method handles `Throwable`, `LostBallException` is handled by the `main()` method, making Option B incorrect. Option C is also incorrect because `ArrayStoreException` is an unchecked exception that extends `RuntimeException` and is not required to be handled or declared. Finally, if `throws` was used instead of `throw`, the entire application would compile without issue and print `Caught!`, making Option A the correct answer.

4. A. The only symbol permitted to separate exception types in a multi-catch statement is a single pipe character (`|`). For this reason, Option A is correct.

5. D. In Java, `assert` is a keyword, meaning it cannot be used as a variable, class, or method name. For this reason, line 5 does not compile. Line 6 also does not compile because the `assert` statement is not a method and does not support parentheses around both expressions. Because neither of these lines compile, Option D is the correct answer.

6. C. First off, `Error` is an unchecked exception. It is recommended that `Error` not be caught by most application processes, making Option A incorrect. `IllegalStateException` inherits `RuntimeException`, both of which are unchecked, making Options B and D, respectively, incorrect. Option C is correct because `ParseException` must be handled or declared.

7. D. The `Exception` class contains multiple constructors, including one that takes `Throwable`, one that takes `String`, and a no-argument constructor. The first `WhaleSharkException` constructor compiles, using the `Exception` constructor that takes a `String`. The second `WhaleSharkException` constructor also compiles. The two statements, `super()` and `new Exception()`, actually call the same constructor in the `Exception` class, one after another. The last `WhaleSharkException` compiles with the compiler inserting the default no-argument constructor `super()`, because it exists in the `Exception` class. For these reasons, all of the constructors compile, and Option D is the correct answer.

8. B. The `UnsupportedOperationException` class is an unchecked exception, which means it inherits from `RuntimeException`. While `Error` also is an unchecked exception, the diagram indicates that the class does not inherit from `Error`. There is only one class between `Throwable` and `RuntimeException` in the diagram, and since we know `RuntimeException` inherits `Exception`, the other missing class must be `Exception`. For these reasons, Option B is the correct answer.

9. C. The code does not compile because the variable `b` is used twice in the `main()` method, both in the method declaration and in the `catch` block, making Option C the correct answer. If a different variable name was used in one of the locations, the program would print one line, `complete`, making Option A the correct answer. Note that while an exception is created inside the `turnOn()` method, it is not thrown.

10. D. First off, unless assertions are enabled at runtime, no assertion statement guarantees an assertion will be thrown at runtime, making Option D the correct answer. Next, Option A does not compile because the `assert` statement is not a method and does not take arguments in parentheses. It's also invalid because it requires a `boolean` expression for the first expression, not a numeric one. An additional value can be specified, but it requires a colon separator (`:`). Option B would be the correct answer and trigger an error if assertions are enabled, since `0==1` evaluates to `false`. Option C is incorrect. Even if assertions were enabled, it would not trigger an error since `0==0` evaluates to `true`.

11. C. The class does not compile because in line `r2`, brackets `{}` are used instead of parentheses `()` in the try-with-resources statement, making Option C the correct answer. If this line

was fixed to use parentheses `()`, then the rest of the class would compile without issue and print `This just in!` at runtime, making Option A the correct answer.

12. C. When both a `try` block and `close()` method throw exceptions, the one in the `try` block is the primary, while the one in the `close()` method is suppressed. For this reason, Option A is a true statement. Option B is also a true statement, since a `catch` block is not required when using a try-with-resources statement. Option C is the correct answer, since resources are closed in reverse order in which they are created, not the other way around. Option D is a true statement because multiple resources can be declared within a single set of parentheses, each separated by a semicolon (`;`).

13. A. The program compiles without issue, so Option D is incorrect. The narrower `SpellingException` and `NullPointerException`, which inherit from `Exception`, are correctly presented in the first `catch` block, with the broader `Exception` being in the second `catch` block. The if-then statement evaluates to true, and a new `SpellingException` instance is created, but it is not thrown because it is missing the `throw` keyword. For this reason, the `try` block ends without any of the `catch` blocks being executed. The `finally` block is then called, making it the only section of code in the program that prints a line of text. For this reason, Option A is the correct answer.

14. C. First off, the `try` block is capable of throwing two checked exceptions, `MissingMoneyException` and `MissingFoodException`. The `catch` block uses the `Exception` class to handle this, since both have `Exception` as a supertype. It then rethrows the `Exception`. For this reason, `Exception` would be appropriate in the blank, making the first statement correct. The compiler is also smart enough to know that there are only two possible subclasses of `Exception` that can actually be thrown in the `main()` method, so declaring `MissingMoneyException` and `MissingFoodException` together also allows the code to compile, making the third statement correct. The second statement, only inserting `MissingMoneyException`, would not allow the code to compile because the `main()` method could throw a checked `MissingFoodException` that was not handled. For these reasons, Option C is the correct answer.

15. C. First off, `Closeable` extends `AutoCloseable`, making Option A incorrect. The difference between the two is that the `close()` method in `AutoCloseable` throws `Exception`, while the `close()` method in `Closeable` throws `IOException`, making Option D incorrect. Since `IOException` is a subclass of `Exception`, both `close()` methods can throw an `IOException`, making Option B incorrect. On the other hand, `Exception` is not a subclass of `IOException`. For this reason, the `close()` method in a class that implements `Closeable` cannot throw an instance of the `Exception` class, because it is an invalid override using a broader exception type, making Option C the correct answer.

16. B. Option A does not compile because a multi-catch expression uses a single variable, not two variables. Option C does not compile because it is not possible to throw this checked `IOException` in the `try` block. Option D does not compile because multi-catch blocks cannot contain two exceptions in which one is a subclass of the other. If it did, one of the two exceptions would be redundant. Option B is the correct answer and the only expression that allows the class to compile. While both exceptions in the multi-catch block inherit from `Exception`, neither is a subclass of the other.

17. C. First off, the order of exceptions in a multi-catch does not matter, only that they not be subclasses of one another, making Options A and B incorrect. Option C is the correct answer because a multi-catch variable is effectively final. Java forbids reassigning of multi-catch variables since it is unclear what the precise reference type is. Option D is incorrect because a multi-catch with a single exception type is just a regular `catch` block. A regular `catch` variable is not required to be effectively final and can be reassigned within the `catch` block.

18. D. The code does not compile, so Option A is incorrect. The first compilation error is that `Shelf` does not implement `AutoCloseable`, meaning a try-with-resources statement cannot be used. Even though `Shelf` does implement `Closing`, an interface that uses the same `abstract` method signature as `AutoCloseable`, the JVM requires `AutoCloseable` be implemented to use try-with-resources. The second compilation problem is that `throws` is used instead of `throw` inside the `try` block. Remember that `throws` is only used in method signatures. The third compilation issue is that the order of exceptions in the two `catch` blocks are reversed. Since `Exception` will catch all `IllegalArgumentException` instances, the second `catch` block is unreachable. The final compilation error is that the `shelf` variable is used in the `finally` block, which is out of scope. Remember that the scope of try-with-resources variables ends when the `try` statement is complete. For these four reasons, Option D is the correct answer.

19. A. Option A is the correct answer. Any `catch` or `finally` blocks used with a try-with-resources statement are executed after the declared resources have been closed, not before. Options B and C are true statements, since `Closeable` extends `AutoCloseable` and the requirement for try-with-resources is that they must be of type `AutoCloseable`. Finally, Option D is a true statement and one of the primary motivations for using try-with-resources statements.

20. D. The optional second parameter of an `assert` statement, when used, must return a value. The second `assert` statement uses `System.out.print()` as its second parameter, which has a return type of `void`. For this reason, the code does not compile, making Option D the correct answer. Other than this one line, the rest of the class compiles without issue.

21. D. Only one of the classes, `MissingResourceException`, inherits from the unchecked `RuntimeException` class, making Option D the correct answer. In fact, `IOException` and `SQLException` extend the checked `Exception` class directly. The `NotSerializableException` is also checked, since it is a subclass of `IOException` via `ObjectStreamException`.

22. D. The code does not compile, making Options A and B incorrect. The declaration of `weatherTracker` uses an anonymous inner class that correctly overrides the `close()` method. Remember that overridden methods cannot throw any new or broader checked exceptions than the inherited method. Alternatively, they can avoid throwing inherited checked exceptions or declare new unchecked exceptions, such as `RuntimeException`. The compilation error is in the `catch` block of the `main()` method, where the `weatherTracker` variable is out of scope. In try-with-resources statements, the resources are only accessible in the `try` block. For this reason, Option D is the correct answer.

23. A. Asserts can be enabled by using the command-line options `-ea` or `-enableassertions` and disabled by using `-da` or `-disableassertions`. Passing `-di` does not enable or disable assertions, making Option A the correct answer.

24. A. The application compiles without issue and prints `Hello`, making Option A the correct answer. The `ReadSign` and `MakeSign` classes are both correctly implemented, with both overridden versions of `close()` dropping the checked exception. The try-with-resources statement is also correctly implemented for two resources and does not cause any compilation errors or runtime exceptions. Note that the semicolon (`;`) after the second resource declaration is optional.

25. B. The code compiles, so Option D is incorrect. The order of evaluation for a try-with-resources statement is that the resources are closed before any associated `catch` or `finally` blocks are executed. For this reason, `2` is printed first, followed by `1`. The `ArithmeticException` is then caught and `3` is printed. The last value printed is `4`, since the `finally` block runs at the end. For these reasons, Option B is the correct answer.

26. B. First off, Option A is an incorrect statement because the `AutoCloseable` interface does not define a `default` implementation of `close()`. Next, the `close()` method should be idempotent, which means it is able to be run multiple times without triggering any side effects. For this reason, Option B is correct. After being run once, future calls to `close()` should not change any data. Option C is incorrect because the `close()` method is fully capable of throwing exceptions. In fact, the signature of the method in `AutoCloseable` throws a checked `Exception`, although classes that override it may choose to drop the checked exception. Option D is incorrect because the return type of `close()` is `void`, which means no return value can be returned.

27. D. The `play()` method compiles without issue, rethrowing a wrapped exception in the `catch` block. While the `main()` method does declare `RuntimeException`, it does not declare or catch the `Exception` thrown by the calls to `play()`. Even though the `play()` method does not appear to actually throw an instance of `Exception`, because it is declared, the `main()` method must catch or declare it. Since the checked exception is not handled, the `main()` method does not compile, and Option D is the correct answer. If the `main()` method was changed to declare the appropriate checked exception, then the rest of the code would compile, and exactly one exception would be printed, making Option A the correct answer.

28. B. Assertions are often used to check method post conditions, test control flow invariants, and validate class invariants, making Options A, C, and D true statements. Option B is the correct answer. An assertion should never modify any data because it may be disabled at runtime, leading to unintended side effects.

29. B. A multi-catch block cannot contain two exceptions in which one is a subclass of the other, since it is a redundant expression. Since `CarCrash` is a subclass of `RuntimeException` and `RuntimeException` is a subclass of `Exception`, line `w2` contains a compilation error, making Option B the correct answer. The rest of the lines of code do not contain any compilation errors.

30. B. First off, a class must inherit from `RuntimeException` or `Error` to be considered an unchecked exception. `Dopey` and `Grumpy` both are subclasses of `Exception`, but not `RuntimeException`, making them both checked exceptions. Since `IOException` is a checked exception, the subclass `Happy` is also a checked exception. `Sleepy` extends `IllegalStateException`, which is an unchecked exception that extends `RuntimeException`. Finally, `Sneezy` extends `Throwable`, which does not inherit `RuntimeException` or `Error`, making it a checked exception. Therefore, there are a total of four checked exceptions and one unchecked exception within the classes listed here. Since there are no compilation errors in any of the class declarations, Option B is the correct answer, with the first and third statement being true.

31. D. The `close()` method in each of the resources throws an `Exception`, which must be handled or declared in the `main()` method. The `catch` block supports `TimeException`, but it is too narrow to catch `Exception`. Since there are no other `catch` blocks present and the `main()` method does not declare `Exception`, the try-with-resources statement does not compile, and Option D is the correct answer. If the `catch` block was modified to handle `Exception` instead of `TimeException`, the code would compile without issue and print `3215` at runtime, closing the resources in the reverse order in which they were declared and making Option B the correct answer.

32. A. The try-catch block already catches `Exception`, so the correct answer would be the one that is not a subclass of `Exception`. In this case, `Error` extends `Throwable` and is the only choice that allows the code to compile. Because `IllegalStateException` and `RingException` both inherit from `Exception`, Options B and C, respectively, are incorrect. Finally, Option D is incorrect because there is an answer choice that allows the code to compile.

33. B. Option A does not compile because the assignment `(age=2)` has a value of `int`, not `boolean`, which is required for an `assert` statement. Option B compiles without issue and is the correct answer. Even though `Error` and `10` are different data types, `String` and `int` respectively, the second `assert` parameter only needs to be a value, so both are allowed. Option C does not compile because the usage of the lambda expression does not match a functional interface. Option D is incorrect because a `return` statement is not allowed in the second expression of an `assert` statement.

34. D. The `MissedCallException` is a checked exception since it extends `Exception` and does not inherit `RuntimeException`. For this reason, the first `catch` block fails to compile, since the compiler detects that it is not possible to throw this checked exception inside the `try` block, making Option D the correct answer. Note that if `MissedCallException` was changed to extend the checked `RuntimeException`, then the code would compile and the `RuntimeException` from the `finally` block would replace the `ArrayIndexOutOfBoundsException` from the `try` block in the message reported to the caller, making Option C the correct answer.

35. D. The `catch` variable `d` is of type `BearException` that cannot be assigned an instance of the superclass `RuntimeException` without an explicit cast. For this reason, the first `catch` block does not compile in `tellStory()`. The second `catch` block also does not compile, albeit for a slightly different reason. A `catch` variable in a multi-catch block must be effectively final because the precise type is not known until runtime. Therefore, the compiler does not allow the variable `e` to be reassigned. For these two reasons, Option D is the correct answer. Note that the first `catch` block does allow the `catch` variable `d` to be reassigned, it just must be to a class that inherits `BearException` or is an instance of `BearException`.

36. D. The `play()` method declares two checked exceptions, `OutOfTuneException` and `FileNotFoundException`, which are handled in the `main()` method's `catch` block using the `Exception` type. The `catch` block then rethrows the `Exception`. The compiler is smart enough to know that only two possible checked exceptions can be thrown here, but they both must be handled or declared. Since the `main()` method only declares one of the two checked exceptions, `FileNotFoundException` is not handled, and the code does not compile. For this reason, Option D is the correct answer. Note that the `main()` could have also handled or declared `Exception`, since both checked exceptions inherit it. If the `main()` method had declared `Exception`, then `Song finished!` would have been printed followed by a stack trace, making Option C the correct answer.

37. C. The code compiles without issue, making Option D incorrect. Option A is incorrect because assertions are not enabled by default in Java. Therefore, the code will run without throwing any exceptions. Option B is also incorrect because the command enables assertions everywhere but disables them for the `Falcon` class. Option C is the correct answer, with the command disabling assertions everywhere except in the `Falcon` class, causing an `AssertionError` to be thrown at runtime.

38. C. The `Closeable` interface defines a `close()` method that throws `IOException`. The overridden implementation of `MyDatabase`, which implements `Closeable`, declares a `SQLException`. This is a new checked exception not found in the inherited method signature. Therefore, the method override is invalid and the `close()` method in `MyDatabase` does not compile, making Option C the correct answer.

39. D. The code does not compile because the `close()` method throws an `Exception` that is not handled or declared in the `main()` method, making Option D the correct answer. When a try-with-resources statement is used with a `close()` method that throws a checked exception, it must be handled by the method or caught within the try-with-resources statement.

40. B. The code compiles without issue, making Option C incorrect. In the `climb()` method, two exceptions are thrown. One is thrown by the `close()` method and the other by the `try` block. The exception thrown in the `try` block is considered the primary exception and reported to the caller on line `e1`, while the exception thrown by the `close()` method is suppressed. For this reason, `java.lang.RuntimeException` is thrown to the `main()` method, and Option B is the correct answer.

Chapter 17: Use Java SE 8 Date/Time API

1. C. The date and time classes added in Java 8 are in the `java.time` package, making Option C correct. The older date classes are in the `java.util` package.

2. A. The `Duration` class is used to reflect an amount of time using small units like minutes. Since it just uses units of time, it does not involve time zones. The `LocalTime` class contains units of hours, minutes, seconds, and fractional seconds. The `LocalDateTime` class contains all the data in a `LocalTime` and adds on a year, month, and date. Neither of these classes uses time zones. There is a `ZonedDateTime` class when you need to use time zones. Since none of the three classes listed includes a time zone, Option A is correct.

3. A. A `Period` is measured in days, weeks, months, or years. A `Duration` is measured in smaller units like minutes or seconds. Only `Duration` has a `getSeconds()` method, making Option A correct.

4. D. To compare times in different time zones, you can subtract the time zone from the time to convert to GMT. This makes it 02:00 in Berlin because we subtract 1 from 3. Similarly, it is 02:00 in Helsinki due to subtracting 2 from 4. Finally, it is 04:00 in Warsaw because we subtracted 1 from 5. We have a tie because it is the same time in Berlin and Helsinki, so Option D is correct.

5. B. On a normal night, adding three hours to 1 a.m. makes it 4 a.m. However, this date begins daylight savings time. This means we add an extra hour to skip the 2 a.m. hour. This makes `later` contain 05:00 instead of 04:00. Therefore, the code prints `5`, and Option B is correct.

6. C. `LocalDate` allows passing the month as an `int` or `Month enum` parameter. However, `Month.MARCH` is an `enum`. It cannot be assigned to an `int` variable, so the declaration of `month` does not compile, and Option C is correct.

7. C. Both `LocalDate` and `DateTimeFormatter` have a `format()` method. This makes II incorrect. While it is tricky, you do need to know that the `format()` method can be called on either object. Since I and III are correct, Option C is the answer.

8. C. Converting to GMT by subtracting the time zone offset, it is 17:00 for the Phoenix time since 10 minus negative 7 is 17. In GMT, the Vancouver time is 16:00 due to subtracting negative 8 from 8. Remember that subtracting a negative number is the same as adding. Therefore, the Vancouver time is an hour earlier than the Phoenix time, and Option C is correct.

9. C. While there is no 2 a.m. on the clock that night, Java adjusts the time to 3 a.m. automatically and changes the time zone. It does not throw an exception, so Option D is incorrect. Option B is a valid expression, since any value after the time adjustment is just a normal time on the clock. Since both A and B are valid expressions, Option C is the correct answer.

10. B. Line 12 creates a `Period` representing a year, six months, and three days. Adding this to `waffleDay` gives us the year 2018, the month of September, and a day of 28. This new date is stored in `later` on line 13 and represents September 28, 2018. Line 14 has no effect as the return value is ignored. Line 17 checks that you know that `isBefore()` returns `false` if the value is an exact match. Since `thisOne` is an exact match but `thatOne` is a whole day before, the output is `false true`, making Option B correct.

11. D. `Duration` is supposed to be used with objects that contain times. While it has an `ofDays()` method, this is a convenience method to represent a large number of seconds. This means that calling `Duration.ofDays(1)` is fine. However, this code throws an `UnsupportedTemporalTypeException` when you try to pass it the `minus()` method on `LocalDate`, making Option D correct. Note that the question asks about a possible result rather than the definitive result because the format of dates varies by region.

12. C. The `DateTimeFormatter` is created with `ofLocalizedDate()`. It knows how to format date fields but not time fields. Line 18 is fine because a `LocalDate` clearly has date fields. Line 19 is also fine. Since a `LocalDateTime` has both date and time fields, the formatter just looks at the date fields. Line 20 is a problem. A `LocalTime` object does not have any date fields so the formatter throws an `UnsupportedTemporalTypeException`, making Option C the answer.

13. D. This question is tricky. It appears to be about daylight savings time. However, the result of `z.plusHours(1)` is never stored in a variable or used. Since `ZonedDateTime` is immutable, the time remains at 01:00. The code prints out `1`, making none of these correct and Option D the answer.

14. D. For dates, a lowercase `m` means minute while an uppercase `M` means month. This eliminates Options A and C. A lowercase `h` means hour. Therefore, Option B is incorrect, and Option D is the answer.

15. D. There are three overloads for `LocalTime.of()`. Options A, B, and C are all valid overloads. Option D is not because Java only allows passing one fractional second parameter. Java does support nanoseconds, but not the further granularity of picoseconds.

16. C. The `LocalDate` class represents a date using year, month, and day fields. There is a `getYear()` method to get the year. The `Period` class holds units of years, months, and days. It has a `getYears()` method. There is not a date/time class called `ZonedDate`. There is a class called `ZonedDateTime`, which does have a `getYear()` method. Since only `LocalDate` and `Period` have a method to get the year, Option C is correct.

17. A. `Duration` is used for units of time a day and smaller, making Option B a true statement. `Period` is used for units of time a day and larger, making Option C a true statement. While both represent the same length of time, they output different values when calling `toString()`. The `Duration` object outputs `PT24H`, and the `Period` object outputs `P1D`. This shows that `Duration` is providing the `ofDays()` method as a convenience instead of requiring the programmer to type 24 hours. Option A is the answer.

18. B. The first thing to notice is that this is a `LocalTime` object. Since there is no date component, Options C and D are incorrect. Four parameters are passed to this `LocalTime` method. The first three are the hour, minute, and second. The fourth is nanoseconds, which are fractions of a second. While you aren't expected to do calculations with nanoseconds, you should know that a fraction of a second is at the end of the output. Option A is incorrect because `.4` is 40 percent of a second. That's far larger than 4 nanoseconds. Therefore, Option B is correct.

19. B. `LocalDate` starts counting months from one, so month 2 is February. This rules out Options A and C. The pattern specifies that the date should appear before the month, making Option B correct.

20. A. The `ChronoUnit enum` contains values for various measures of time including `HOURS`, so Option A is correct.

21. B. Adding three hours to 13:00 makes it 16:00. While this date happens to be the start of daylight savings time, the change occurs at 2 a.m. This code uses 13:00, which is 1 p.m. Since the calculation does not cross 2 a.m., the fact that it is the date that starts daylight savings time is irrelevant. Option B is correct because the hour is `16` and the time is 16:00.

22. B. This code correctly subtracts a day from `montyPythonDay`. It then outputs a `LocalDateTime` value. Option A is incorrect because it omits the time. Option B is correct because it represents one day earlier than the original date and includes a time in the output.

23. D. There is a `DateTimeFormatter` class, but not a `DateFormatter` class. The `DateTimeFormatter` class is used for formatting dates, times, or both. Since the provided code does not compile, nothing can fill in the blank to make the code print `2017-01-15`, and Option D is the answer.

24. B. There are many overloads for `LocalDateTime.of()`. Option A is a valid overload because it uses date and time objects. Options C and D are also valid overloads, showing you can pass the month as an `int` or `Month enum`. Option B is the answer. Java doesn't allow combining a `LocalDate` object with time fields directly.

25. C. In the first time change of the year, clocks "spring ahead" and skip the 02:00–03:00 hour entirely. This means 1:59 is followed by 03:00 on March 12, 2017. By contrast, July 4 is a normal day and 1:59 is followed by 02:00. In the second time change of the year, clocks "fall back" and repeat an hour, so 1:59 is followed by 01:00. Granted, you can't tell whether this is the first or second 1:59 from the image. If this information is relevant to a question's context, the question will specify this fact. In this case, 03:00, 02:00, 02:00 is not a choice. Option C is the answer.

26. D. February has 28 or 29 days, depending on the year. There is never a February 31. Java throws a `DateTimeException` when you try to create an invalid date, making Option D correct.

27. A. This one is tricky. In order to determine GMT, you need to subtract the time zone offset from the hour. In this case, the time zone offset is negative 10. Since subtracting a negative number is like adding a positive number, this means we are adding 16 and 10. That gives us 26. However, there are only 24 hours in a day. We've crossed a time zone boundary, so we can remove a whole 24-hour day. Subtracting 24 from 26 gives us 2. This means it is 02:00 in GMT, and Option A is correct. It's also a day later in GMT, but the question didn't ask that.

28. D. An `Instant` represents a specific moment in time using GMT. Since there are no time zones included, Options A and C are incorrect. This code correctly adds one day to the `instant`, making Option D correct.

29. D. Make sure to pay attention to date types. This code attempts to add a month to a `LocalTime` value. There is no `plusMonths()` method on `LocalTime`, so Option D is correct.

30. D. The format of the pattern is incorrect. You can't just put literal text in there. Most of the characters of `Holiday:` are not defined formatting symbols. The code throws an `IllegalArgumentException`, so Option D is correct.

31. A. To compare times in different time zones, you can subtract the time zone from the time. This makes it 09:00 in Bangkok because we subtract 7 from 16. Similarly, it is 14:00 in Dubai due to subtracting 4 from 18. Finally, it is 12:00 in Kuala Lumpur because we subtracted 8 from 20. Notice how we used a 24-hour clock to make comparing times easier. The earliest time is in Bangkok, so Option A is correct.

32. C. Line 12 creates a `Period` representing three days. `Period` objects do not chain, so only the last method call, which is to `ofDays(3)`, is used in determining the value. Adding three days sets `later` to March 28, 2017. Line 14 has no effect as the return value is ignored. March 28, 2017, is before both `thisOne` and `thatOne`, so Option C is correct.

33. B. The `TemporalUnit` interface does not define a `DAYS` constant, making II and IV incorrect. The `until()` and `between()` methods have the same behavior. They determine how many units of time are between two dates. One takes both dates as parameter and the other is an instance method on the date. Since I and III are equivalent, Option B is the answer. Note that while we don't have date times in this question, the `until()` and `between()` methods work the same way for them.

34. A. The `DateTimeFormatter` class is used to format all of these objects. The method will throw an exception if called with a `LocalDate` since the formatter only knows about time fields. However, it will still compile, making Option A correct.

35. B. This code begins by correctly creating four objects. It then adds a month to `date`. Since Java 8 date/time classes are immutable, this does not affect the value of `iceCreamDay`. Therefore, `iceCreamDay` remains in July. Since months count from one, Option B is correct.

36. A. Java 8 date and time classes are immutable. They use a `static` factory method to get the object reference rather than a constructor. This makes Options B and D incorrect. Further, there is not a `ZonedDate` class. There is a `ZonedDateTime` class. As a result, Option C is incorrect, and Option A is the answer.

37. B. The first line of code correctly creates a `LocalDate` object representing March 3, 2017. The second line adds two days to it, making it March 5. It then subtracts a day, making it March 4. Finally, it subtracts yet another day ending at March 3. The outcome of all this is that we have two dates that have the same value, and Option B is correct.

38. C. An `Instant` represents a specific moment in time using GMT. Since `LocalDateTime` does not have a time zone, it cannot be converted to a specific moment in time. Therefore, the `toInstant()` call does not compile, and Option C is correct.

39. A. While it is traditional to include the year when outputting a date, it is not required. This code correctly prints the month followed by a decimal point. After the decimal point, it prints the day of the month followed by the hours and minutes. Happy Pi Day!

40. C. Normally, adding an hour would result in 02:00 in the same time zone offset of –05:00. Since the hour is repeated, it is 01:00 again. However, the time zone offset changes instead. Therefore, Option C is correct.

Chapter 18: Java I/O Fundamentals

1. B. `Writer` is an abstract class, making Option B the correct answer. Note that `InputStream`, `OutputStream`, and `Reader` are also abstract classes.

2. D. `File` uses `mkdir()` and `mkdirs()` to create a directory, not `createDirectory()`, making Option A incorrect. Note there is a `createDirectory()` method in the NIO.2 `Files` class. The `getLength()` method also does not exist, as the correct method is called `length()`. Next, there is a `listFiles()` method used to read the contents of a directory, but there is no `listFile()` method. That leaves us with `renameTo()`, which does exist and is used to rename file system paths.

3. C. The `skip()` method just reads a single byte and discards the value. The `read()` method can be used for a similar purpose, making Option C the correct answer. Option A is incorrect because there is no `jump()` method defined in `InputStream`. Options B and D are incorrect because they cannot be used to skip data, only to mark a location and return to it later, respectively.

4. D. `Serializable` is a marker or tagging interface, which means it does not contain any methods and is used to provide information about an object at runtime. Therefore, Option D is the correct answer because the interface does not define any `abstract` methods.

5. C. Given a valid instance of `Console`, `reader()` returns an instance of `Reader`, while `writer()` returns an instance of `PrintWriter`. `Reader` and `PrintWriter` was not an answer choice though, making Option C the next best choice since `PrintWriter` inherits `Writer`. Options A and B are incorrect because `PrintReader` is not defined in the `java.io` library. Option D is incorrect because the type of the instance returned by `reader()` is `Reader`, which does not inherit `StringReader`.

6. D. `BufferedWriter` is a wrapper class that requires an instance of `Writer` to operate on. In the `Smoke` class, a `FileOutputStream` is passed, which does not inherit `Writer`, causing the class not to compile, and making Option D the correct answer. If `FileWriter` was used instead of `FileOutputStream`, the code would compile without issue and print `13`, making Option B the correct answer.

7. A. The `File` class is used to read both files and directories within a file system, making Option A the correct answer. The other three classes do not exist. Note there is an NIO.2 interface, `java.nio.file.Path`, used to read both file and path information.

8. C. `FileOutputStream` and `FileReader` are both low-level streams that operate directly on files, making Options A and B incorrect. `ObjectInputStream` is a high-level stream that can only wrap an existing `InputStream`. For this reason, Option C is the correct answer. `PrintWriter` can operate on other streams, but it can also operate on files. Since the question asks which class can only wrap low-level streams, Option D is incorrect.

9. D. The code compiles, so Option C is incorrect. The `FileInputStream` does not support marks, though, leading to an `IOException` at runtime when the `reset()` method is called.

For this reason, Option D is the correct answer. Be suspicious of any code samples that call the `mark()` or `reset()` method without first calling `markSupported()`.

10. C. The absolute path is the full path from the root directory to the file, while the relative path is the path from the current working directory to the file. For this reason, Option C is the correct answer.

11. D. The difference between the two methods is that `writeSecret1()` does not take any steps to ensure the `close()` method is called after the resource is allocated. On the other hand, `writeSecret2()` uses a try-with-resources block, which automatically tries to close the resource after it is used. Without a try-with-resources statement or an equivalent `finally` block, any exception thrown by the `write()` method would cause the resource not to be closed in the `writeSecret1()` method, possibly leading to a resource leak. For this reason, Option D is the correct answer. Option A is incorrect since they are not equivalent to each other. Finally, Options B and C are incorrect because both compile without issue.

12. A. The constructor for `Console` is `private`. Therefore, attempting to call `new Console()` outside the class results in a compilation error, making Option A the correct answer. The correct way to obtain a `Console` instance is to call `System.console()`. Even if the correct way of obtaining a `Console` had been used, and the `Console` was available at runtime, `stuff` is `null` in the `printItinerary()` method. Referencing `stuff.activities` results in a `NullPointerException`, which would make Option B the correct answer.

13. A. While you might not be familiar with `FilterOutputStream`, the diagram shows that the two classes must inherit from `OutputStream`. Options B and C can be eliminated as choices since `PrintOutputStream` and `Stream` are not the name of any `java.io` classes. Option D can also be eliminated because `OutputStream` is already in the diagram, and you cannot have a circular class dependency. That leaves us with the correct answer, Option A, with `BufferedOutputStream` and `PrintStream` both extending `FilterOutputStream`. Note that `ByteArrayOutputStream` and `FileOutputStream` referenced in Options C and D, respectively, do not extend `FilterOutputStream`, although knowing this fact was not required to solve the problem.

14. D. The `Cereal` class does not implement the `Serializable` interface; therefore, attempting to write the instance to disk, or calling `readObject()` using `ObjectInputStream`, will result in a `NotSerializableException` at runtime. For this reason, Option D is the correct answer. If the class did implement `Serializable`, then the value of `name` would be `CornLoops`, since none of the constructor, initializers, or setters methods are used on deserialization, making Option B the correct answer.

15. B. An `OutputStream` is used to write bytes, while a `Writer` is used to write character data. Both can write character data, the `OutputStream` just needs the data converted to bytes first. For this reason, Option A is incorrect. Option B is the correct answer, with `Writer` containing numerous methods for writing character or `String` data. Both interfaces contain a `flush()` method, making Option C incorrect. Finally, because both can be used with a `byte` array, Option D is incorrect.

16. C. First off, the code compiles without issue. The first method call to `mkdirs()` creates two directories, `/templates` and `/templates/proofs`. The next `mkdir()` call is unnecessary, since `/templates/proofs` already exists. That said, calling it on an existing directory is harmless and does not cause an exception to be thrown at runtime. Next, a file `draft.doc` is created in the `/templates` directory. The final two lines attempt to remove the newly created directories. The first call to `delete()` is successful because `/templates/proofs` is an empty directory. On the other hand, the second call to `delete()` fails to delete the directory `/templates` because it is non-empty, containing the file `draft.doc`. Neither of these calls trigger an exception at runtime, though, with `delete()` just returning a `boolean` value indicating whether the call was successful. Therefore, our program ends without throwing any exceptions, and Option C is the correct answer.

17. D. To answer the question, you need to identify three of the four ways to call the system-independent file name separator. For example, the file name separator is often a forward-slash (/) in Linux-based systems and a backward-slash (\) in Windows-based systems. Option A is valid because it is the fully qualified name of the property. Option B is also valid because `File.separator` and `File.separatorChar` are equivalent. While accessing a `static` variable using an instance is discouraged, as shown in Option B, it is allowed. Option C is valid and a common way to read the character using the `System` class. Finally, Option D is the correct answer and one call that cannot be used to get the system-dependent name separator character. Note that `System.getProperty("path.separator")` is used to separate sets of paths, not names within a single path.

18. D. The first compilation error is that the `FileReader` constructor call is missing the `new` keyword. The second compilation error is that the `music` variable is marked `final`, but then modified in the `while` loop. The third compilation problem is that the `readMusic()` method fails to declare or handle an `IOException`. Even though the `IOException` thrown by `readLine()` is caught, the one thrown by the implicit call to `close()` via the try-with-resources block is not caught. Due to these three compilation errors, Option D is the correct answer.

19. C. Both of the methods do exist, making Option D incorrect. Both methods take the same arguments and do the exact same thing, making Option C the correct answer. The `printf()` was added as a convenience method, since many other languages use `printf()` to accomplish the same task as `format()`.

20. C. `FileWriter` and `BufferedWriter` can be used in conjunction to write large amounts of text data to a file in an efficient manner, making Option C the correct answer. While you can write text data using `FileOutputStream` and `BufferedOutputStream`, they are primarily used for binary data. Since there is a better choice available, Option A is incorrect. Option B is incorrect since `FileOutputWriter` and `FileBufferedWriter` are not classes that exist within the `java.io` API. Option D is incorrect since `ObjectOutputStream` is a high-level binary stream. Also, while it can write `String` data, it writes it in a binary format, not a text format.

21. D. The code compiles and runs without issue, so Options A and B are incorrect. The problem with the implementation is that checking if `ios.readObject()` is `null` is not the recommended way of iterating over an entire file. For example, the file could have been written with `writeObject(null)` in-between two non-`null` records. In this case, the reading of the

file would stop on this `null` value, before the end of the file has been reached. For this reason, Option D is the correct answer. Note that the valid way to iterate over all elements of a file using `ObjectInputStream` is to continue to call `readObject()` until an `EOFException` is thrown.

22. D. `BufferedInputStream` is the complement of `BufferedOutputStream`. Likewise, `BufferedReader` and `FileReader` are the complements of `BufferedWriter` and `FileWriter`, respectively. On the other hand, `PrintWriter` does not have an accompanying `PrintReader` class within the `java.io` API, making Option D the correct answer. Remember that this is also true of `PrintStream`, as there is no `PrintInputStream` class.

23. C. The `File getParent()` method returns a `String`, not a `File` object. For this reason, the code does not compile on the last line since there is no `getParent()` method defined in the `String` class, and Option C is correct. If the first method call on the last line was changed to `getParentFile()`, then the code would compile and run without issue, outputting `/ - null` and making Option B the correct answer. The `File` class does not require the location to exist within the file system in order to perform some operations, like `getParent()`, on the path.

24. D. All three statements about the program are correct. If `System.console()` is available, then the program will ask the user a question and then print the response if one is entered. On the other hand, if `System.console()` is not available, then the program will exit with a `NullPointerException`. It is strongly recommended to always check whether or not `System.console()` is `null` after requesting it. Finally, the user may choose not to respond to the program's request for input, resulting in the program hanging indefinitely and making the last statement true.

25. C. The code contains two compilation errors. First, the `File list()` method returns a list of `String` values, not `File` values, so the call to `deleteTree()` with a `String` value does not compile. Either the call would have to be changed to `f.listFiles()` or the lambda expression body would have to be updated to `deleteTree(new File(s))` for the code to work properly. Next, there is no `deleteDirectory()` method in the `File` class. Directories are deleted with the same `delete()` method used for files, once they have been emptied. With those two sets of corrections, the method would compile and be capable of deleting a directory tree. Notice we continually used the phrase "capable of deleting a directory tree." While the corrected code is able to delete a directory tree, it may fail in some cases, such as if the file system is read-only.

26. C. `System.err`, `System.in`, and `System.out` are each valid streams defined in the `System` class. `System.info` is not, making Option C the correct answer.

27. D. The code compiles without issue, making Options B and C incorrect. The `BufferedWriter` uses the existing `FileWriter` object as the low-level stream to write the file to disk. By using the try-with-resources block, though, the `BufferedWriter` calls `close()` before executing any associated `catch` or `finally` blocks. Since closing a high-level stream automatically closes the associated low-level stream, the `w` object is already closed by the time the `finally` block is executed. For this reason, the `flush()` command triggers an `IOException` at runtime, making Option D the correct answer. Note that the call to `w.close()`, if that line was reached, does not trigger an exception, because calling `close()` on already closed streams is innocuous.

28. B. The `Console` class contains a `reader()` method that returns a `Reader` object. The `Reader` class defines a `read()` method, but not a `readLine()` method. For this reason, Option B is the correct answer. Recall that a `BufferedReader` is required to call the `readLine()` method. Options A, C, and D are valid ways to read input from the user.

29. C. The code compiles without issue, since `InputStream` and `OutputStream` both support reading/writing `byte` arrays, making Option A incorrect. Option D is also incorrect. While it is often recommended that an I/O array be a power of 2 for performance reasons, it is not required, making Option D incorrect. This leaves us with Options B and C. The key here is the `write()` method used does not take a length value, available in the `chirps` variable, when writing the file. The method will always write the entire `data` array, even when only a handful of bytes were read into the `data` array, which may occur during the last iteration of the loop. The result is that files whose bytes are a multiple of `123` will be written correctly, while all other files will be written with extra data appended to the end of the file. For this reason, Option C is the correct answer. If the `write(data)` call was replaced with `write(data,0,chirps)`, which does take the number of bytes read into consideration, then all files would copy correctly, making Option B the correct answer.

30. C. The class name has three components that tell you what it would do if it was a `java.io` stream. First, `Buffered` tells you it can be used to handle large data sets efficiently. Next, `File` tells you it is involved in reading or writing files. Finally, `Reader` tells you it is used to read character data. Therefore, the class would be useful for reading large files of character data from disk efficiently, making Option C the correct answer. Option A is incorrect because it refers to a small file over a network. Options B and D are incorrect because both involve binary data.

31. A. The code compiles and runs without issue. The first two values of the `ByteArrayInputStream` are read. Next, the `markSupported()` value is tested. Since `-1` is not one of the possible answers, we assume that `ByteArrayInputStream` does support marks. Two values are read and three are skipped, but then `reset()` is called, putting the stream back in the state before `mark()` was called. In other words, everything between `mark()` and `reset()` can be ignored. The last value read is `3`, making Option A the correct answer.

32. C. Line 5 creates a `File` object, but that does not create a file in the file system unless `cake.createNewFile()` is called. Line 6 also does not necessarily create a file, although the call to `flush()` will on line 7. Note that this class does not properly close the file resource, potentially leading to a resource leak. Line 8 creates a new `File` object, which is used to create a new directory using the `mkdirs()` method. Recall from your studies that `mkdirs()` is similar to `mkdir()`, except that it creates any missing directories in the path. Since directories can have periods (`.`) in their name, such as a directory called `info.txt`, this code compiles and runs without issue. Since two file system objects, a file and a directory, are created, Option C is the correct answer.

33. B. Since the file is stored on disk, `FileInputStream` is an appropriate choice. Next, because the data is quite large, a `BufferedInputStream` would help improve access. Finally, since the data is a set of Java values, `ObjectInputStream` would allow various different formats to be read. The only one that does not help in this process is `BufferedReader`, Option B. `BufferedReader` should be used with text-based `Reader` streams, not binary `InputStream` objects.

34. B. The `flush()` method is defined on classes that inherit `Writer` and `OutputStream`, not `Reader` and `InputStream`. For this reason, the `r.flush()` in both methods does not compile, making Option B the correct answer and Option C incorrect. The methods are not equivalent even if they did compile, since `getNameSafely()` ensures the resource is closed properly by using a try-with-resources statement, making Option A incorrect for two reasons. Finally, Option D would be correct if the calls to `flush()` were removed.

35. A. First off, the class compiles without issue. Although there are built-in methods to print a `String` and read a line of input, the developer has chosen not to use them for most of the application. The application first prints `Pass`, one character at a time. The `flush()` method does not throw an exception at runtime. In fact, it helps make sure the message is presented to the user. Next, the user enters `badxbad` and presses Enter. The stream stops reading on the `x`, so the value stored in the `StringBuilder` is `bad`. Finally, this value is printed to the user, using the `format()` method along with `Result:` as a prefix. For these reasons, Option A is the correct answer.

36. B. The `readPassword()` returns a `char` array for security reasons. If the data was stored as a `String`, it would enter the shared JVM string pool, potentially allowing a malicious user to access it, especially if there is a memory dump. By using a `char` array, the data can be immediately cleared after it is written and removed from memory. For this reason, Option B is the correct answer. The rest of the statements are not true.

37. A. The `BufferedInputStream` constructor in the `readBook()` method requires an `InputStream` as input. Since `FileReader` does not inherit `InputStream`, the `readBook()` method does not compile, and Option A is the correct answer. If `FileReader` was changed to `FileInputStream`, then the code would compile without issue. Since `read()` is called twice per loop iteration, the program would print every other byte, making Option C correct. Remember that `InputStream read()` returns `-1` when the end of the stream is met. Alternatively, we use `EOFException` with `ObjectInputStream readObject()` to determine when the end of the file has been reached.

38. B. Generally speaking, classes should be marked with the `Serializable` interface if they contain data that we might want to save and retrieve later. Options A, C, and D describe the type of data that we would want to store over a long period of time. Option B, though, defines a class that manages transient or short-lived data. Application processes change quite frequently, and trying to reconstruct a process is often considered a bad idea. For these reasons, Option B is the best answer.

39. D. The `receiveText()` method compiles and runs without issue. The method correctly checks that the `mark()` method is supported before attempting to use it. Based on the implementation with `reset()`, the pointer is in the same location before/after the if-then statement. On the other hand, the `sendText()` method does not compile. The `skip()` method is defined on `InputStream` and `Reader`, not `OutputStream` and `Writer`, making Option D the correct answer. If this line was removed, the rest of the code would compile and run without issue, printing `You up?` at runtime and making Option A the correct answer.

40. B. The class compiles and runs without issue, so Option D is incorrect. The class defines three variables, only one of which is serializable. The first variable, `chambers`, is serializable, with the value `2` being written to disk and then read from disk. Note that constructors and instance initializers are not executed when a class is deserialized. The next variable, `size`, is `transient`. It is discarded when it is written to disk, so it has the default object value of `null` when read from disk. Finally, the variable `color` is `static`, which means it is shared by all instances of the class. Even though the value was `RED` when the instance was serialized, this value was not written to disk, since it was not part of the instance. The constructor call `new Value()` between the two try-with-resources blocks sets this value to `BLUE`, which is the value printed later in the application. For these reasons, the class prints `2,null,BLUE`, making Option B the correct answer.

Chapter 19: Java File I/O (NIO.2)

1. C. A symbolic link is a file that contains a reference to another file or directory within the file system, making Options A and B incorrect. Further, there is no such thing as an irregular file. Option C is the correct answer because a regular file is not a directory and contains content, unlike a symbolic link or resource. Option D is also incorrect because all symbolic links are stored as files, not directories, even when their target is a directory.

2. C. The NIO.2 `Path` interface contains the methods `getRoot()` and `toRealPath()`. On the other hand, the method `isDirectory()` is found in the NIO.2 `Files` class, while the method `listFiles()` is found in the `java.io.File` class. For these reasons, Option C is the correct answer.

3. A. The code does not compile because there is no NIO.2 class `File` that contains an `isHidden()` method, making Option A the correct answer. There is a `java.io.File` class, but that does not contain an `isHidden()` method either. The correct call is `Files.isHidden()`. Remember to check `File` vs. `Files` as well as `Path` vs. `Paths` on the real exam. If the correct method call was used, the program would print `Found!`, and Option C would be the correct answer.

4. D. A breadth-first traversal is when all elements of the same level, or distance from the starting path, are visited before moving on to the next level. On the other hand, a depth-first traversal is when each element's entire path, from start to finish, is visited before moving onto another path on the same level. Both `walk()` and `find()` use depth-first traversals, so Option D is the correct answer.

5. A. Reading an attribute interface is accomplished in a single trip to the underlying file system. On the other hand, reading multiple file attributes using individual `Files` methods requires a round-trip to the file system for each method call. For these reasons, Option A is the correct answer. Option B is incorrect because nothing guarantees it will perform faster, especially if the `Files` method is only being used to read a single attribute. For multiple calls, it is expected to be faster, but the statement uses the word *guarantees*, which is incorrect. Option C is also incorrect because both have built-in support for symbolic links. Finally, Option D is incorrect because this discussion has nothing to do with memory leaks.

6. B. First off, the class compiles without issue. It is not without problems, though. The `Files.isSameFile()` method call on line `j1` first checks if the `Path` values are equivalent in terms of `equals()`. One is absolute and the other is relative, so this test will fail. The `isSameFile()` method then moves on to verify that the two `Path` values reference the same file system object. Since we know the directory does not exist, the call to `isSameFile()` on line `j1` will produce a `NoSuchFileException` at runtime, making Option B the correct answer.

7. B. A cycle is caused when a path contains a symbolic link that references the path itself, or a parent of the parent, triggering an infinitely deep traversal. That said, `Files.walk()` does not follow symbolic links by default. For this reason, the cycle is never activated, and the code would print a number at runtime, making Option B the correct answer. If the `FOLLOW_LINKS` enum value was used in the call to `Files.walk()`, then it would trigger a cycle resulting in a `FileSystemLoopException` at runtime, and Option A would be the correct answer.

8. B. The methods `length()` and `getLength()` do not exist in the `Files` class, making Options A and C incorrect. Recall that the `java.io.File` method retrieves the size of a file on disk. The NIO.2 `Files` class includes the `Files.size()` method to accomplish this same function. For this reason, Option B is the correct answer.

9. D. The code compiles without issue, making Option C incorrect. Even though `tricks` would be dropped in the normalized path `/bag/of/disappear.txt`, there is no `normalize()` call, so `path.subpath(2,3)` returns `tricks` on line 5. On line 6, the call to `getName()` throws an `IllegalArgumentException` at runtime. Since `getName()` is zero-indexed and contains only one element, the call on line 6 throws an `IllegalArgumentException`, making Option D the correct answer. If `getName(0)` had been used instead of `getName(1)`, then the program would run without issue and print `/home/tricks`, and Option A would have been the correct answer.

10. A. The NIO.2 `Files` class contains the method `isSameFile()`. The methods `length()` and `mkdir()` are found in `java.io.File`, with the NIO.2 equivalent versions being `Files.size()` and `Files.createDirectory()`, respectively. In addition, the `relativize()` method is found in NIO.2 `Path`, not `Files`. Since only `isSameFile()` is found in NIO.2 `Files`, Option A is the correct answer.

11. B. First off, the code compiles without issue, so Option D is incorrect. The enum value `REPLACE_EXISTING` does not use a type, although this compiles correctly if a `static` import of `StandardCopyOption` is used. The `AtomicMoveNotSupportedException` in Option A is only possible when the `ATOMIC_MOVE` option is passed to the `move()` method. Similarly, the `FileAlreadyExistsException` in Option C is only possible when the `REPLACE_EXISTING` option is not passed to the `move()` method. That leaves us with the correct answer of Option B. A `DirectoryNotEmptyException` can occur regardless of the options passed to the `Files.move()` method.

12. D. The `Path` method `getFileName()` returns a `Path` instance, not a `String`. For this reason, the code does not compile, regardless of which line of code is inserted into the blank, making Option D the correct answer. Statements I and III are two valid ways to create a `Path` instance. If the method was updated to use `Path` as the return type, then Option B would be the correct answer. Statement II would cause the method to not compile, because `Path` is an interface and requires a class to be instantiated.

13. A. The code compiles without issue, but that's about it. The class may throw an exception at runtime, since we have not said whether or not the source file exists nor whether the target file already exists, is a directory, or is write-protected. For these reason, Option B is incorrect. Option C is also incorrect because the implementation is a flawed copy method. On a regular file, the code will copy the contents but the line breaks would be missing in the target file. In order to correctly copy the original file, a line break would have to be written after each time `temp` is written. Since it is the only correct statement, Option A is the correct answer.

14. C. First off, there is no `Files.readLines()` method, making Options B and D immediately incorrect. The `Files.readAllLines()` method returns a `List<String>`, while the `Files.lines()` method returns a `Stream<String>`. For this reason, Option C is the correct answer, and Option A is incorrect.

15. A. The program compiles and runs without issue, making Options C and D incorrect. Like `String` instances, `Path` instances are immutable. For this reason, the `resolve()` operation on line 7 has no impact on the `lessTraveled` variable. Since one `Path` ends with `/spot.txt` and the other does not, they are not equivalent in terms of `equals()`, making Option A the correct answer. If lines 6 and 7 were combined, such that the result of the `resolve()` operation was stored in the `lessTraveled` variable, then `normalize()` would reduce `lessTraveled` to a `Path` value that is equivalent to `oftenTraveled`, making Option B the correct answer.

16. C. Options A, B, and D are each advantages of using NIO.2. As you may remember, using an attribute view to read multiple attributes at once is more efficient than a single attribute call since it involves fewer round trips to the file system. Option C is the correct answer. Neither API provides a single method to delete a directory tree.

17. C. The `Files.delete()` method has a return type of `void`, not `boolean`, resulting in a compilation error and making Option C the correct answer. There is another method, `Files.deleteIfExists()`, which returns `true` if the file is able to be deleted. If it was used here instead, the file would compile and print a list of `true` values, making Option A the correct answer. As stated in the description, the directory tree is fully accessible, so none of the `Files.deleteIfExists()` would return `false`.

18. D. First off, `DosFileAttributes` and `PosixFileAttributes` extend `BasicFileAttributes`, which means they are compatible with the `readAttributes()` method signature. Second, they produce instances that inherit the interface `BasicFileAttributes`, which means they can be assigned to a variable `b` of type `BasicFileAttributes` without an explicit cast. For this reason, all three interfaces are permitted, and Option D is the correct answer.

19. D. The `relativize()` method requires that both path values be absolute or relative. Based on the details provided, `p1` is a relative path, while `p2` is an absolute path. For this reason, the code snippet produces an exception at runtime, making Option D the correct answer. If the first path was modified to be absolute by dropping the leading dot (`.`) in the path expression, then the output would match the values in Option A.

20. C. First off, `p2` is an absolute path, which means that `p1.resolve(p2)` just returns `p2`. For this reason, Option B is incorrect. Since `p1` is a relative path, it is appended onto `p2`, making Option C correct and Option A incorrect.

21. B. The code does not compile because `Files.list()` returns a `Stream<Path>`, not a `List<Path>`, making Option B the correct answer. Note that `java.io.File` does include a `list()` method that returns an array of `String` values and a `listFiles()` method that returns an array of `File` values, but neither is applicable here.

22. C. For this problem, remember that the path symbols can be applied to simplify the path before needing to apply any symbolic links in the file system. The paths in Options A and B can both be reduced from `/objC/bin/../backwards/../forward/Sort.java` and `/objC/bin/../forward/./Sort.java`, respectively, to `/objC/forward/Sort.java` just using the path symbols. Because of the symbolic link, this references the same file as `/java/Sort.java`. For these reasons, Options A and B match our target path. Option C can be reduced from `/objC/bin/../java/./forward/Sort.java` to `/objC/java/forward/Sort.java`, which does not match the desired path for the file. The symbolic link is not followed since it exists in the `/objC` directory, not in the `/objC/java` directory. This causes a stack trace to be printed at runtime since the path does not exist, making Option C the correct answer. Option D can be reduced from `/objC/bin/../../java/Sort.java` to `/java/Sort.java`, which matches the target path without using the symbolic link.

23. B. We need to empty the `/objC` directory before we can delete it. First, the `Heap.exe` file would have to be deleted before the `bin` directory could be removed, for a total of two calls to `Files.delete()`. Next, the `Heap.m` file is easily deleted with a single call to `Files.delete()`. Calling `Files.delete()` on the symbolic link `forward` deletes the link itself and leaves the target of the symbolic link intact. With a total of four calls, Option B is the correct answer. Option A is incorrect because Java requires directories to be empty before they can be deleted. Option C is also incorrect. It might make sense if `Files.delete()` traversed symbolic links on a delete, but since this is not the case, it is an incorrect answer. Option D is incorrect because there is no `Files.deleteSymbolicLink()` method defined in the Java NIO.2 API.

24. C. Since `System.out` is a `PrintStream` that inherits `OutputStream` and implements `Closeable`, line `y1` compiles without issue. On the other hand, the `Files.copy()` does not compile because there is no overloaded version of `Files.copy()` that takes an `OutputStream` as the first parameter. For this reason, Option C is the correct answer. If the order of the arguments in the `Files.copy()` call was switched, then the code would compile and print the contents of the file at runtime, making Option D the correct answer.

25. B. To begin with, the `BasicFileAttributeView` class contains methods to read and write file data, while the `BasicFileAttributes` class only contains methods to read file data. The advantage of using a `BasicFileAttributeView` is to also modify file data, so Option D is incorrect. Next, The `BasicFileAttributeView` does not include a method to modify the hidden attribute. Instead, a `DosFileAttributeView` is required, making Option A incorrect. Option B is the correct answer because `BasicFileAttributeView` includes a `setTimes()` method to modify the file date values. Finally, Option C is incorrect because both read file information in a single round-trip.

26. A. Trick question! The code does not compile, therefore no `Path` values are printed, and Option A is the correct answer. The key here is that `toRealPath()` interacts with the file system and therefore throws a checked `IOException`. Since this checked exception is not handled inside the lambda expression, the class does not compile. If the lambda expression was fixed to handle the `IOException`, then the expected number of `Path` values printed would be six, and Option C would be the correct answer. A `maxDepth` value of `1` causes the `walk()` method to visit two total levels, the original `/flower` and the files it contains.

27. D. The first statement returns a `null` value, since the path `..` does not have a parent. That said, it does not throw an exception at runtime, since it is not operated upon. The second and third statements both return paths representing the root (`/`) at runtime. Remember that calling `getRoot()` on a root path returns the root path. The fourth statement throws a `NullPointerException` at runtime since `getRoot()` on a relative path returns `null`, with the call to `getParent()` triggering the exception. Since the fourth statement is the only one to produce a `NullPointerException` at runtime, Option D is the correct answer.

28. C. The code compiles without issue, so Options A and B are incorrect. While many of the `Files` methods do throw `IOException`, most of the `Path` methods do not throw a checked exception. The lack of indent of the `return` statement on line 6 is intentional and does not prevent the class from compiling. If the input argument `p` is `null` or not an absolute path, then the if-then clause is skipped, and it is returned to the caller unchanged. Alternatively, if the input argument is an absolute path, then calling `toAbsolutePath()` has no effect. In both cases, the return value of the method matches the input argument, making Option C the correct answer.

29. D. Option A is incorrect because both methods take exactly one `Path` parameter, along with an optional vararg of `FileAttribute` values. Option B is also incorrect because both methods will throw a `FileAlreadyExistsException` if the target exists and is a file. Option C is incorrect since both methods declare a checked `IOException`. The correct answer is Option D. The method `createDirectory()` creates a single directory, while `createDirectories()` may create many directories along the path value.

30. C. The `toAbsolutePath()` combines the current working directory and relative path to form a `/hail/../jungle/.././rain..` path. The `normalize()` method removes the path symbols and leaves a `/rain..` value. Note that the last double period (`..`) is not removed because it is part of a path name and not interpreted as a path symbol. The result is then appended with `snow.txt` and we are left with `/rain../snow.txt`, making Option C the correct answer.

31. A. The program compiles and runs without issue, so Options C and D are incorrect. The process breaks apart the inputted path value and then attempts to reconstitute it. There is only one problem. The method call `getName(0)` does not include the root element. This results in the `repaired` variable having a value of `tissue/heart/chambers.txt`, which is not equivalent to the original path. The program prints `false`, and Option A is the correct answer.

32. B. Unlike `Files.delete()`, the `Files.deleteIfExists()` method does not throw an exception if the path does not exist, making Option B the correct answer. Options A, C, and D describe situations in which the Java process encounters a path in a state that cannot be deleted. In each of these situations, an exception would be thrown at runtime.

33. D. The code does not compile because `Path` is an interface and does not contain a `get()` method. Since the first line contains a compilation error, Option D is the correct answer. If the code was corrected to use `Paths.get()`, then the output would be `true false true`, and Option B would be the correct answer. The normalized path of both is `/desert/sand.doc`, which means they would be equivalent, in terms of `equals()`, and point to the same path in the file system. On the other hand, the non-normalized values are not equivalent, in terms of `equals()`, since the objects represent distinct path values.

34. C. First off, the `Files.getFileAttributeView()` method requires a reference to a subclass of `FileAttributeView`, such as `BasicFileAttributeView.class`. The parameter must also be compatible with the reference assignment to `vw`. For these two reasons, this line of code does not compile. Next, `BasicFileAttributeView` does not contain a `creationTime()` method, so `vw.creationTime()` results in a compilation error. For the exam, remember that view classes do contain access to attributes, but only through the `readAttributes` method, such as `vw.readAttributes().creationTime()`. Since these are the only two lines that contain compilation errors, Option C is the correct answer. Note that we purposely omitted all `import` statements in this question, since this may happen on the real exam.

35. B. The program compiles and runs without issue, making Options C and D incorrect. The first variable, `halleysComet`, is created with `normalize()` being applied right away, leading to a value of `stars/m1.meteor`. The second variable, `lexellsComet`, starts with a value of `./stars/../solar/`. The `subpath()` call reduces it to its first two components, `./stars`. The `resolve()` method then appends `m1.meteor`, resulting in a value of `./stars/m1.meteor`. Finally, `normalize()` further reduces the value to `stars/m1.meteor`. Since this matches our first `Path`, the program prints `Same!`, and Option B is the correct answer.

36. D. Both stream statements compile without issue, making Options A and B incorrect. The two statements are equivalent to one another and print the same values at runtime. For this reason, Option C is incorrect, and Option D is correct. There are some subtle differences in the implementation besides one using `walk()` with a `filter()` and the other using `find()`. The `walk()` call does not include a depth limit, but since `Integer.MAX_VALUE` is the default value, the two calls are equivalent. Furthermore, the `walk()` statement prints a stream of absolute paths stored as `String` values, while the `find()` statement prints a stream of `Path` values. If the input `p` was a relative path, then these two calls would have very different results, but since we are told `p` is an absolute path, the application of `toAbsolutePath()` does not change the results.

37. A. The code does not compile because `Files.lines()` and `Files.readAllLines()` throw a checked `IOException`, which must be handled or declared. For the exam, remember that other than a handful of test methods, like `Files.exists()`, most methods in the NIO.2 `Files` class that operate on file system records declare an `IOException`. Now, if the exceptions were properly handled or declared within the class, then `jonReads()` would likely take more time to run. Like all streams, `Files.lines()` loads the contents of the file in a lazy manner, meaning the time it takes for `jenniferReads()` to run is constant regardless of the file size. Note the stream isn't actually traversed since there is no terminal operation. Alternatively, `Files.readAllLines()` reads the entire contents of the file before returning a list of `String` values. The larger the file, the longer it takes `jonReads()` to execute. Since the original question says the file is significantly large, then if the compilation problems were corrected, `jonReads()` would likely take longer to run, and Option C would be the correct answer.

38. C. The first `copy()` method call on line `q1` compiles without issue because it matches the signature of a `copy()` method in `Files`. It also does not throw an exception because the `REPLACE_EXISTING` option is used and we are told the file is fully accessible within the file system. On the other hand, the second `copy()` method on line `q2` does not compile. There is a version of `Files.copy()` that takes an `InputStream`, followed by a `Path` and a list of copy options. Because `BufferedReader` does not inherit `InputStream`, though, there is no matching `copy()` method and the code does not compile. For this reason, Option C is the correct answer.

39. C. The `Files.isSameFile()` throws a checked `IOException`. Even though accessing the file system can be skipped in some cases, such as if the `Path` instances are equivalent in terms of `equals()`, the method still declares `IOException` since it may access the file system to determine if the two `Path` instances refer to the same file. For this reason, Option C is the correct answer. The rest of the methods listed do not throw any checked exceptions, even though they do access the file system, instead returning `false` if the file does not exist.

40. B. The program compiles and runs without issue, making Options C and D incorrect. The program uses `Files.list()` to iterate over all files within a single directory. For each file, it then iterates over the lines of the file and counts the sum. For this reason, Option B is the correct answer. If the `count()` method had used `Files.walk()` instead of `Files.lines()`, then the class would still compile and run, and Option A would be the correct answer. Note that we had to wrap `Files.lines()` in a try-catch block because using this method directly within a lambda expression without a try-catch block leads to a compilation error.

Chapter 20: Java Concurrency

1. A. The `ExecutorService` interface defines the two `submit()` methods shown in Options C and D. Because `ExecutorService` extends `Executor`, it inherits the `execute(Runnable)` method presented in Option B. That leaves us with the correct answer, Option A, because `ExecutorService` does not define nor inherit an overloaded method `execute()` that takes a `Callable` parameter.

2. B. The class compiles and runs without throwing an exception, making the first statement true. The class defines two values that are incremented by multiple threads in parallel. The first `IntStream` statement uses an atomic class to update a variable. Since updating an atomic numeric instance is thread-safe by design, the first number printed is always `10`, and the second statement is true. The second `IntStream` statement uses an `int` with the pre-increment operator (`++`), which is not thread-safe. It is possible two threads could update and set the same value at the same time, a form of race condition, resulting in a value less than `5`. For this reason, the third statement is not true. Since only the first two statements are true, Option B is the correct answer.

3. C. Option A is incorrect, although it would be correct if `Executors` was replaced with `ExecutorService` in the sentence. While an instance of `ExecutorService` can be obtained

from the `Executors` class, there is no method in the `Executors` class that performs a task directly. Option B is also incorrect, but it would be correct if `start()` was replaced with `run()` in the sentence. It is recommended that you override the `run()` method, not the `start()` method, to execute a task using a custom `Thread` class. Option C is correct, and one of the most common ways to define an asynchronous task. Finally, Option D is incorrect because Options A and B are incorrect.

4. D. Trick question! `ExecutorService` does not contain any of these methods. In order to obtain an instance of a thread executor, you need to use the `Executors` factory class. For this reason, Option D is the correct answer. If the question had instead asked which `Executors` method to use, then the correct answer would be Option C. Options A and B do not allow concurrent processes and should not be used with a `CyclicBarrier` expecting to reach a limit of five concurrent threads. Option C, on the other hand, will create threads as needed and is appropriate for use with a `CyclicBarrier`.

5. C. `CopyOnWriteArrayList` makes a copy of the array every time it is modified, preserving the original list of values the iterator is using, even as the array is modified. For this reason, the `for` loop using `copy1` does not throw an exception at runtime. On the other hand, the `for` loops using `copy2` and `copy3` both throw `ConcurrentModificationException` at runtime since neither allows modification while they are being iterated upon. Finally, the `ConcurrentLinkedQueue` used in `copy4` completes without throwing an exception at runtime. For the exam, remember that the `Concurrent` classes order read/write access such that access to the class is consistent across all threads and processes, while the synchronized classes do not. Because exactly two of the `for` statements produce exceptions at runtime, Option C is the correct answer.

6. C. Resource starvation is when a single active thread is perpetually unable to gain access to a shared resource. Livelock is a special case of resource starvation, in which two or more active threads are unable to gain access to shared resources, repeating the process over and over again. For these reasons, Option C is the correct answer. Deadlock and livelock are similar, although in a deadlock situation the threads are stuck waiting, rather than being active or performing any work. Finally, a race condition is an undesirable result when two tasks that should be completed sequentially are completed at the same time.

7. B. The class does not compile because the `Future.get()` on line 8 throws a checked `InterruptedException` and `ExecutionException`, neither of which is handled nor declared by the `submitReports()` method. If the `submitReports()` and accompanying `main()` methods were both updated to declare these exceptions, then the application would print `null` at runtime, and Option A would be the correct answer. For the exam, remember that `Future` can be used with `Runnable` lambda expressions that do not have a return value but that the return value is always `null` when completed.

8. A. Options B and C are both proper ways to obtain instances of `ExecutorService`. Remember that `newSingleThreadExecutor()` is equivalent to calling `newFixedThreadPool()` with a value of `1`. Option D is the correct way to request a single-threaded `ScheduledExecutorService` instance. The correct answer is Option A. The method `newFixedScheduledThreadPool()` does not exist in the `Executors` class, although there is one called `newScheduledThreadPool()`.

9. A. The code compiles without issue but hangs indefinitely at runtime. The application defines a thread executor with a single thread and 12 submitted tasks. Because only one thread is available to work at a time, the first thread will wait endlessly on the call to `await()`. Since the `CyclicBarrier` requires four threads to release it, the application waits endlessly in a frozen condition. Since the barrier is never reached and the code hangs, the application will never output `Ready`, making Option A the correct answer. If `newCachedThreadPool()` had been used instead of `newSingleThreadExecutor()`, then the barrier would be reached three times, and Option C would be the correct answer.

10. D. First off, `BlockingDeque` is incorrect since it is an interface, not a class. Next, `ConcurrentLinkedDeque` does support adding elements to both ends of an ordered data structure but does not include methods for waiting a specified amount of time to do so, referred to as blocking. `ConcurentSkipListSet` is also incorrect, since its elements are sorted and not just ordered, and it does not contain any blocking methods. That leaves the correct answer, Option D. A `LinkedBlockingDeque` includes blocking methods in which elements can be added to the beginning or end of the queue, while waiting at most a specified amount of time.

11. A. The `findAny()` method can return any element of the stream, regardless of whether the stream is serial or parallel. While on serial streams this is likely to be the first element in the stream, on parallel streams the result is less certain. For this reason, Option A is the correct answer. When applied to an ordered stream, the rest of the methods always produce the same results on both serial and parallel streams. For this reason, these operations can be costly on a parallel stream since it has to be forced into a serial process.

12. D. The `static` method `Array.asList()` returns a `List` instance, which inherits the `Collection` interface. While the `Collection` interface defines a `stream()` and `parallelStream()` method, it does not contain a `parallel()` method. For this reason, the second stream statement does not compile, and Option D is the correct answer. If the code was corrected to use `parallelStream()`, then the arrays would be consistently printed in the same order, and Option C would be the correct answer. Remember that the `forEachOrdered()` method forces parallel streams to run in sequential order.

13. D. To start with, the `ForkJoinTask` is the parent class of `RecursiveAction` and `RecursiveTask` and does not contain a `compute()` method, neither abstract nor concrete, making Options A and C automatically incorrect. The `RecursiveTask` class contains the abstract `compute()` method that utilizes a generic return type, while the `RecursiveAction` class contains the abstract `compute()` method that uses a `void` return type. For this reason, Option D is the correct answer.

14. B. An accumulator in a serial or parallel reduction must be associative and stateless. In a parallel reduction, invalid accumulators tend to produce more visible errors, where the result may be processed in an unexpected order. Option A is not associative, since `(a-b)-c` is not the same as `a-(b-c)` for all values `a`, `b`, and `c`. For example, using values of `1`, `2`, and `3` results in two different values, `-4` and `2`. Option C is not stateless, since a class or instance variable `i` is modified each time the accumulator runs. That leaves us with Option B, which is the correct answer since it is both stateless and associative. Even though it ignores the input parameters, it meets the qualifications for performing a reduction.

15. B. The code does not compile because `Callable` must define a `call()` method, not a `run()` method, so Option B is the correct answer. If the code was fixed to use the correct method name, then it would complete without issue, printing `Done!` at runtime, and Option A would be the correct answer.

16. C. Part of synchronizing access to a variable is ensuring that read/write operations are atomic, or happen without interruption. For example, an increment operation requires reading a value and then immediately writing it. If any thread interrupts this process, then data could be lost. In this regard, Option C shows proper synchronized access. Thread 2 reads a value and then writes it without interruption. Thread 1 then reads the new value and writes it. The rest of the answers are incorrect because one thread writes data to the variable in-between another thread reading and writing to the same variable. Because a thread is writing data to a variable that has already been written to by another thread, it may set invalid data. For example, two increment operations running at the same time could result in one of the increment operations being lost.

17. D. The code compiles and runs without issue. The two methods `hare()` and `tortoise()` are nearly identical, with one calling `invokeAll()` and the other calling `invokeAny()`. The key is to know that both methods operate synchronously, waiting for a result from one or more tasks. Calling the `invokeAll()` method causes the current thread to wait until all tasks are finished. Since each task is one second long and they are being executed in parallel, the `hare()` method will take about one second to complete. The `invokeAny()` method will cause the current thread to wait until at least one task is complete. Although the result of the first finished thread is often returned, it is not guaranteed. Since each task takes one second to complete, though, the shortest amount of time this method will return is after one second. In this regard, the `tortoise()` method will also take about one second to complete. Since both methods take about the same amount of time, either may finish first, causing the output to vary at runtime and making Option D the correct answer. Note that after this program prints the two strings, it does not terminate, since the `ExecutorService` is not shut down.

18. B. `ConcurrentSkipListMap` implements the `SortedMap` interface, in which the keys are kept sorted, making Option B the correct answer. While the other answers define ordered data structures, none are guaranteed to be sorted. Remember, if you see `SkipList` as part of a concurrent class name, it means it is sorted in some way, such as a sorted set or map.

19. D. The `synchronized` block used in the `getQuestion()` method requires an object to synchronize on. Without it, the code does not compile, and Option D is the correct answer. What if the command was fixed to synchronize on the current object, such as using `synchronized(this)`? Each task would obtain a lock for its respective object, then wait a couple of seconds before requesting the lock for the other object. Since the locks are already held, both wait indefinitely, resulting in a deadlock. In this scenario, Option A would be the correct answer since a deadlock is the most likely result. We say most likely because even with corrected code, a deadlock is not guaranteed. It is possible, albeit very unlikely, for the JVM to wait five seconds before starting the second task, allowing enough time for the first task to finish and avoiding the deadlock completely.

20. B. The `ScheduledExecutorService` does not include a `scheduleAtFixedDelay()` method, so Option A is incorrect. The `scheduleAtFixedRate()` method creates a new task for the associated action at a set time interval, even if previous tasks for the same action are still active. In this manner, it is possible multiple threads working on the same action could be executing at the same time, making Option B the correct answer. On the other hand, `scheduleWithFixedDelay()` waits until each task is completed before scheduling the next task, guaranteeing at most one thread working on the action is active in the thread pool.

21. D. The application compiles, so Option B is incorrect. The `stroke` variable is thread-safe in the sense that no write is lost since all writes are wrapped in a `synchronized` method, making Option C incorrect. Even though the method is thread-safe, the value of `stroke` is read while the threads may still be executing. The result is it may output `0`, `1000`, or anything in-between, making Option D the correct answer. If the `ExecutorService` method `awaitTermination()` is called before the value of `stroke` is printed and enough time elapses, then the result would be `1000`, and Option A would be the correct answer.

22. B. A race condition is an undesirable result when two tasks that should be completed sequentially are completed at the same time. The result is often corruption of data in some way. If two threads are both modifying the same `int` variable and there is no synchronization, then a race condition can occur with one of the writes being lost. For this reason, Option B is the correct answer. Option A is the description of resource starvation. Options C and D are describing livelock and deadlock, respectively.

23. A. The code compiles, so Option C is incorrect. The application attempts to count the elements of the `sheep` array, recursively. For example, the first two elements are totaled by one thread and added to the sum of the remainder of the elements in the array, which is calculated by another thread. Unfortunately, the class contains a bug. The `count` value is not marked `static` and not shared by all of the `CountSheep` subtasks. The value of `count` printed in the `main()` menu comes from the first `CountSheep` instance, which does not modify the `count` variable. The application prints `0`, and Option A is the correct answer. If `count` was marked `static`, then the application would sum the elements correctly, printing `10`, and Option B would be the correct answer.

24. D. First off, certain stream operations, such as `limit()` or `skip()`, force a parallel stream to behave it a serial manner, so Option A is incorrect. Option B is also incorrect. Although some operations could take less time to execute, there is no guarantee any operation will actually be faster. For example, the JVM may only allocate a single thread to a parallel stream. In other words, parallel streams may improve performance but do not guarantee it. Option C is incorrect because parallel stream operations are not synchronized. It is up to the developer to provide synchronization or use a concurrent collection if required. Finally, Option D is the correct answer. The `BaseStream` interface, which all streams inherit, includes a `parallel()` method. Of course, the results of an operation may change in the presence of a parallel stream, such as when a stateful lambda expression is used, but they all can be made parallel.

25. A. The code compiles and runs without issue. The JVM will fall back to a single-threaded process if all of the conditions for performing the parallel reduction are not met. The stream used in the `main()` method is not parallel, but the `groupingbyConcurrent()` method can still be applied without throwing an exception at runtime. Although performance will suffer from not using a parallel stream, the application will still process the results correctly. Since the process groups the data by year, Option A is the correct answer.

26. A. The code compiles and runs without issue. The three-argument `reduce()` method returns a generic type, while the one-argument `reduce()` method returns an `Optional`. The `concat1()` method is passed an identity `"a"`, which it applies to each element, resulting in the reduction to `aCataHat`. The lambda expression in the `concat2()` method reverses the order of its inputs, leading to a value of `HatCat`. Therefore, Option A is the correct answer.

27. A. The code compiles without issue, so Options B and C are incorrect. The `f1` declaration uses the version of `submit()` in `ExecutorService`, which takes a `Runnable` and returns a `Future<?>`. The call `f1.get()` waits until the task is finished and always returns `null`, since `Runnable` expressions have a `void` return type. The `f2` declaration uses an overloaded version of `submit()`, which takes a `Callable` expression and returns a generic `Future` object. Since the `double` value can be autoboxed to a `Double` object, the line compiles without issue with `f2.get()` returning `3.14159`. For these reasons, Option A is the correct answer. Option D is incorrect because no exception is expected to be thrown at runtime.

28. C. The class compiles without issue, making Options A and D incorrect. The class attempts to create a synchronized version of a `List<Integer>`. The `size()` and `addValue()` help synchronize the read/write operations. Unfortunately, the `getValue()` method is not synchronized so the class is not thread-safe, and Option C is the correct answer. It is possible that one thread could add to the `data` object while another thread is reading from the object, leading to an unexpected result. Note that the synchronization of the `size()` method is valid, but since `ThreadSafeList.class` is a shared object, this will synchronize all instances of the class to the same object. This could result in a substantial performance cost if enough threads are creating `ThreadSafeList` objects.

29. D. The post-decrement operator (`--`) decrements a value but returns the original value. It is equivalent to the atomic `getAndDecrement()` method. The pre-increment operator (`++`) increments a value and then returns the new value. It is equivalent to the `incrementAndGet()` atomic operation. For these reasons, Option D is the correct answer.

30. B. When a `CyclicBarrier` goes over its limit, the barrier count is reset to zero. The application defines a `CyclicBarrier` with a barrier limit of 5 threads. The application then submits 12 tasks to a cached executor service. In this scenario, a cached thread executor will use between 5 and 12 threads, reusing existing threads as they become available. In this manner, there is no worry about running out of available threads. The barrier will then trigger twice, printing five `1`s for each of the sets of threads, for a total of ten `1`s. For this reason, Option B is the correct answer. The application then hangs indefinitely, as discussed in the next question.

31. D. The application does not terminate successfully nor produce an exception at runtime, making Options A and B incorrect. It hangs at runtime because the `CyclicBarrier` limit is five, while the number of tasks submitted and awaiting activation is 12. This means that 2 of the tasks will be left over, stuck in a deadlocked state waiting for the barrier limit to be reached but with no more tasks available to trigger it. For this reason, Option D is the correct answer. If the number of tasks was a multiple of the barrier limit, such as 10 instead of 12, then the application will still hang because the `ExecutorService` is never shut down. The `isShutdown()` in the application `finally` block does not trigger a shutdown. Remember that it is important to shut down an `ExecutorService` after you are finished with it, else it can prevent a program from terminating. In this case, Option C would be the correct answer.

32. C. The code does not compile because the blocking methods `offerLast()` and `pollFirst()` each throw a checked `InterruptedException` that are not handled by the lambda expressions, so Option C is the correct answer. If the lambda expressions were wrapped with try-catch blocks, then the process would first add all items to the queue, then remove them all of them, resulting in an output of `0`. In this case, Option A would be the correct answer. Even though the tasks are completed in parallel, each stream does not terminate until all tasks are done. Note that 10 seconds is more than enough time under normal circumstances to add/remove elements from the queue.

33. A. First of all, the `for` loops using `copy1` and `copy4` both throw `ConcurrentModificationException` at runtime since neither allows modification while they are being iterated upon. Next, `CopyOnWriteArrayList` makes a copy of the array every time it is modified, preserving the original list of values the iterator is using, even as the array is modified. For this reason, the `for` loop using `copy2` completes without throwing an exception or creating an infinite loop. Finally, the `ConcurrentLinkedDeque` used in `copy3` completes without producing an exception or infinite loop. The `Concurrent` collections order read/write access such that access to the class is consistent across all threads and processes, even iterators. Because the values are inserted at the head of the queue using `push()` and the underlying data structure is ordered, the new values will not be iterated upon and the loop finishes. Since none of the `for` statements produce an infinite loop at runtime, Option A is the correct answer. If `push()` had been used instead of `offer()` in the third loop, with new values being inserted at the tail of the queue instead of at the head, then the `for` loop would have entered an infinite loop, and Option B would be the correct answer.

34. B. Options A, C, and D are the precise requirements for Java to perform a concurrent reduction using the `collect()` method, which takes a `Collector` argument. Recall from your studies that a `Collector` is considered concurrent and unordered if it has the `Collector.Characteristics` enum values `CONCURRENT` and `UNORDERED`, respectively. Option B is the correct answer because elements of a stream are not required to implement `Comparable` in order to perform a parallel reduction.

35. D. The class compiles and runs without issue, making Options A and B incorrect. The purpose of the fork/join framework is to use parallel processing to complete subtasks across multiple threads concurrently. Unfortunately, calling the `compute()` method inside of an existing `compute()` does not spawn a new thread. The result is that this task is completed using a single thread, despite a pool of threads being available. For this reason, Option D is the correct answer. In order to properly implement the fork/join framework, the `compute()` method would need to be rewritten. The `f1.compute()` call should be replaced with `f1.fork()` to spawn a separate task, followed by `f2.compute()` to process the data on the

current thread, and ending in `f1.join()` to retrieve the results of the first task completed while `f2.compute()` was being processed. If the code was rewritten as described, then Option C would be the correct answer.

36. D. The `shutdown()` method prevents new tasks from being added but allows existing tasks to finish. In addition to preventing new tasks from being added, the `shutdownNow()` method also attempts to stop all running tasks. Neither of these methods guarantee any task will be stopped, making Option D the correct answer. Option C is incorrect because there is no `halt()` method in `ExecutorService`.

37. B. First off, the class uses a synchronized list, which is thread-safe and allows modification from multiple threads, making Option D incorrect. The process generates a list of numbers from `1` to `5` and sends them into a parallel stream where the `map()` is applied, possibly out of order. This results in elements being written to `db` in a random order. The stream then applies the `forEachOrdered()` method to its elements, which will force the parallel stream into a single-threaded state. At runtime, line `p1` will print the results in order every time as `12345`. On the other hand, since the elements were added to `db` in a random order, the output of line `p2` is random and cannot be predicted ahead of time. Since the results may sometimes be the same, Option B is the correct answer. Part of the reason that the results are indeterminate is that the question uses a stateful lambda expression, which based on your studies should be avoided in practice!

38. C. The program compiles and does not throw an exception at runtime, making Options B and D incorrect. The class attempts to add and remove values from a single `cookie` variable in a thread-safe manner but fails to do so because the methods `deposit()` and `withdrawal()` synchronize on different objects. The instance method `deposit()` synchronizes on the `bank` object, while the `static` method `withdrawal()` synchronizes on the `static Bank.class` object. Even though method calls of the same type are protected, calls across the two different methods are not. Since the compound assignment operators (`+=`) and (`-=`) are not thread-safe, it is possible for one call to modify the value of `cookies` while the other is already operating on it, resulting in a loss of information. For this reason, the output cannot be predicted, and Option C is the correct answer. If the two sets of calls were properly synchronized on the same object, then the `cookies` variable would be protected from concurrent modifications, and Option A would be the correct answer.

39. A. The code attempts to search for a matching element in an array recursively. While it does not contain any compilation problems, it does contain an error. Despite creating `Thread` instances, it is not a multi-threaded program. Calling `run()` on a `Thread` runs the process as part of the current thread. To be a multi-threaded execution, it would need to instead call the `start()` method. For this reason, the code completes synchronously, waiting for each method call to return before moving on to the next and printing `true` at the end of the execution, making Option A the correct answer. On the other hand, if `start()` had been used, then the application would be multi-threaded but then the result may not be ready by the time the `println()` method is called, resulting in a value that cannot be predicted ahead of time. In this case, Option D would be the correct answer.

40. C. Line 13 does not compile because the `execute()` method has a return type of `void`, not `Future`. Line 15 does not compile because `scheduleAtFixedRate()` requires four arguments that include an initial delay and period value. For these two reasons, Option C is the correct answer.

Chapter 21: Building Database Applications with JDBC

1. C. `Connection` is an interface for communicating with the database. `Driver` is tricky because you don't write code that references it directly. However, you are still required to know it is a JDBC interface. `DriverManager` is used in JDBC code to get a `Connection`. However, it is a concrete class rather than an interface. Since `Connection` and `Driver` are JDBC interfaces, Option C is correct.

2. D. Database-specific implementation classes are not in the `java.sql` package. The implementation classes are in database drivers and have package names that are specific to the database. Therefore, Option D is correct. The `Driver` interface is in the `java.sql` package. Note that these classes may or may not exist. You are not required to know the names of any database-specific classes, so the creators of the exam iare free to make up names.

3. D. All JDBC URLs begin with the protocol `jdbc` followed by a colon as a delimiter. Option D is the only one that does both of these, making it the answer.

4. A. The `Driver` interface is responsible for getting a connection to the database, making Option A the answer. The `Connection` interface is responsible for communication with the database but not making the initial connection. The `Statement` interface knows how to run the SQL query, and the `ResultSet` interface knows what was returned by a `SELECT` query.

5. B. The requirement to include a `java.sql.Driver` file in the driver jar file was introduced in JDBC 4.0. A 3.0 driver is allowed, but not required, to include this file. JDBC 3.0 also requires a call to `Class.forName()`. As a result, Option B best fills in the blanks.

6. C. `Connection` is an interface. Since interfaces do not have constructors, Option D is incorrect. The `Connection` class doesn't have a `static` method to get a `Connection` either, making Option A incorrect. The `Driver` class is also an interface without `static` methods, making Option B incorrect. Option C is the answer because `DriverManager` is the class used in JDBC to get a `Connection`.

7. B. The `DriverManager.getConnection()` method can be called with just a URL. It is also overloaded to take the URL, username, and password, making Option B correct.

8. D. `CallableStatement` and `PreparedStatement` are interfaces that extend the `Statement` interface. You don't need to know that for the exam. You do need to know that a database driver is required to provide the concrete implementation class of `Statement` rather than the JDK. This makes Option D correct. Note that while Derby is provided with Java, it is in a separate jar from the "main" JDK.

9. C. A JDBC URL has three components separated by colons. All three of these URLs meet those criteria. For the data after the component, the database driver specifies the format. Depending on the driver, this might include an IP address and port. Regardless, it needs to include the database name or alias. I and II could both be valid formats because they

mention the database box. However, III only has an IP address and port. It does not have a database name or alias. Therefore III is incorrect and Option C correct.

10. C. The requirement to include a `java.sql.Driver` file in the driver jar was introduced in JDBC 4.0. A call to `Class.forName()` was made optional with JDBC 4.0. As a result, Option C best fills in the blanks.

11. A. Scroll sensitive is a result set type parameter, and updatable is a concurrency mode. The result set type parameter is passed to `createStatement()` before the concurrency mode. If you request options that the database driver does not support, it downgrades to an option it does support rather than throwing an exception. Statements I and III are correct, making Option A the answer.

12. B. JDBC 4.0 allows, but does not require, a call to the `Class.forName()` method. However, since it is in the code, it needs to be correct. This method is expecting a fully qualified class name of a database driver, not the JDBC URL. As a result, the `Class.forName()` method throws a `ClassNotFoundException`, and Option B is the answer.

13. B. There are two `ResultSet` concurrency modes: `CONCUR_READ_ONLY` and `CONCUR_UPDATABLE`. All database drivers support read-only result sets, but not all support updatable ones. Therefore, Option B is correct.

14. D. This code is missing a call to `rs.next()`. As a result, `rs.getInt(1)` throws a `SQLException` with the message `Invalid cursor state - no current row`. Therefore, Option D is the answer.

15. D. The `execute()` method is allowed to run any type of SQL statements. The `executeUpdate()` method is allowed to run any type of the SQL statement that returns a row count rather than a `ResultSet`. Both `DELETE` AND `UPDATE` SQL statements are allowed to be run with either `execute()` or `executeUpdate()`. They are not allowed to be run with `executeQuery()` because they do not return a `ResultSet`. Therefore, Option D is the answer.

16. C. `Connection` is an interface rather than a concrete class. Therefore, it does not have a constructor and line `s2` does not compile. As a result, Option C is the answer. Option A would be the answer if the code `new Connection()` was changed to `DriverManager.getConnection()`.

17. A. There are three `ResultSet` type options: `TYPE_FORWARD_ONLY`, `TYPE_SCROLL_INSENSITIVE`, and `TYPE_SCROLL_SENSITIVE`. Only one of these is in the list, making Option A correct.

18. B. Unlike arrays, JDBC uses one-based indexes. Since `num_pages` is in the second column, the parameter needs to be `2`, ruling out Options A and C. Further, there is not a method named `getInteger()` on the `ResultSet` interface, ruling out Option D. Since the proper method is `getInt()`, Option B is the answer.

19. D. Option A does not compile because you have to pass a column index or column name to the method. Options B and C compile. However, there are not columns named `0` or `1`. Since these column names don't exist, the code would throw a `SQLException` at runtime. Option D is correct as it uses the proper column name.

20. B. The parameters to `createStatement()` are backward. However, they still compile because both are of type `int`. This means the code to create the `Statement` does compile, and Option A is incorrect. Next comes the code to create the `ResultSet`. While both `execute()` and `executeQuery()` can run a `SELECT` SQL statement, they have different return types. Only `executeQuery()` can be used in this example. The code does not compile because the `execute()` method returns a `boolean`, and Option B is correct. If this was fixed, Option D would be the answer because `rs.next()` is never called.

21. D. Since this code opens `Statement` using a try-with-resources, `Statement` gets closed automatically at the end of the block. Further, closing a `Statement` automatically closes a `ResultSet` created by it, making Option D the answer. Remember that you should close any resources you open in code you write.

22. C. Option A is incorrect because `Driver` is an interface while `DriverManager` is a concrete class. The inverse isn't true either; `DriverManager` doesn't implement `Driver`. Option B is incorrect because the `Connection` implementation comes from the database driver jar. Option C is correct. You can turn off auto-commit mode, but it defaults to on. Option D is incorrect because you need to call `rs.next()` or an equivalent method to point to the first row.

23. C. The requirement to include a `java.sql.Driver` file in the `META-INF` directory was introduced in JDBC 4.0. Older drivers are not required to provide it, making Option B incorrect. A file named `jdbc.driver` has never been a requirement. Option A is incorrect and is simply here to trick you. All drivers are required to implement the `Connection` interface, making Option C the answer.

24. D. First, `rs.next()` moves the cursor to point to the first row, which contains the number `10`. Line `q1` moves the cursor to immediately before the first row. This is the same as the position it was in before calling `rs.next()` in the first place. It is a valid position but isn't a row of data. Line `q2` tries to retrieve the data at this position and throws a `SQLException` because there isn't any data, making Option D the answer.

25. B. This code shows how to properly update a `ResultSet`. Note that it calls `updateRow()` so the changes get applied in the database. This allows the `SELECT` query to see the changes and output `10`. Option B is correct. Remember that unlike this code, you should always close a `ResultSet` when you open it in real code.

26. C. There is no `ResultSet` method named `prev()`. Therefore, the code doesn't compile, and Option C is correct. If `prev()` was changed to `previous()`, the answer would be Option B because `updateRow()` is never called. Remember that unlike this code, you should always close a `ResultSet` when you open it in real code.

27. D. While the code turns off automatic committing, there is a `commit()` statement after the first two inserts that explicitly commits those to the database. Then automatic commit is turned back on and the third commit is made, making Option D the answer.

28. A. The `count(*)` function in SQL always returns a number. In this case, it is the number zero. This means line `r1` executes successfully because it positions the cursor at that row. Line `r2` also executes successfully and prints `0`, which is the value in the row. Since the code runs successfully, Option A is the answer.

29. B. The cursor starts out at position zero, right before the first row. Line 6 moves the cursor to position five. Line 7 tries to move the cursor ten rows before that position which is row negative five. Since you can't move back before row zero, the cursor is at row zero instead. Then line 8 moves the cursor forward five positions from row zero, leaving it at row five and making Option B the answer.

30. C. JDBC 4.0 allows, but does not require, a call to the `Class.forName()` method. Since the database does not exist, `DriverManager.getConnection()` throws a `SQLException`, and Option C is the answer.

31. D. When running a query on a `Statement`, Java closes any already open `ResultSet` objects. This means that `rs1` is closed on line 8. Therefore, it throws a `SQLException` on line 9 because we are trying to call `next()` on a closed `ResultSet`, and Option D is correct.

32. B. The code turns off automatic committing, so the inserts for `red` and `blue` are not immediately made. The `rollback()` statement actually prevents them from being committed. Then automatic commit is turned back on and one insert is made, making Option B the answer.

33. A. This code correctly obtains a `Connection` and `Statement`. It then runs a query, getting back a `ResultSet` without any rows. The `rs.next()` call returns `false`, so nothing is printed, making Option A correct.

34. B. Since the `ResultSet` type allows scrolling, the code does not throw a `SQLException` at runtime. Immediately after getting the `ResultSet`, the cursor is positioned at the end immediately after Scott's row. The next two lines try to move forward one row. This has no effect since the cursor is already at the end. Then `previous()` moves the cursor to point to the last row, which is Scott's row. The second `previous()` call moves the cursor up one more row to point to Elena's row, making Option B the answer.

35. B. When passing a negative number to `absolute()`, Java counts from the end instead of the beginning. The last row is Scott's row, so the first print statement outputs `Scott`. When passing a positive number to `absolute()`, Java counts from the beginning, so `Jeanne` is output. Therefore, Option B is correct.

36. D. When creating the `Statement`, the code doesn't specify a result set type. This means it defaults to `TYPE_FORWARD_ONLY`. The `absolute()` method can only be called on scrollable result sets. The code throws a `SQLException`, making Option D the answer.

37. B. This code does not compile because the `ResultSet` options need to be supplied when creating the `Statement` object rather than when executing the query. Since the code does not compile, Option B is correct.

38. B. The code turns off automatic committing, so the inserts for `red` and `blue` are not immediately made. The `rollback()` statement says to prevent any changes made from occurring. This gets rid of `red` and `blue`. Then automatic commit is turned back on and the one insert for `green` is made. The final rollback has no effect since the commit was automatically made. Since there was one row added, Option B is the answer.

39. D. Line 18 doesn't compile because `beforeFirst()` has a `void` return type. Since the code doesn't compile, it doesn't print `true` at all, and Option D is correct. If line 18 called `rs.beforeFirst()` without trying to print the result, Option B would be the answer. All the other statements are valid and return `true`.

40. B. When manually closing database resources, they should be closed in the reverse order from which they were opened. This means that the `ResultSet` object is closed before the `Statement` object and the `Statement` object is closed before the `Connection` object. This makes Option B the answer.

Chapter 22: Localization

1. D. Oracle defines a locale as a geographical, political, or cultural region. Time zones often span multiple locales, so Option D is correct.

2. C. Currencies vary in presentation by locale. For example, `9,000` and `9.000` both represent nine thousand, depending on the locale. Similarly, for dates, `01-02-1991` and `02-01-1991` represent January 2, 1991, depending on the locale. This makes Option C the answer.

3. C. The `Locale` object provides `getDefault()` and `setDefault()` methods for working with the default locale, so Option C is correct. There is no `get()` method declared on `Locale`.

4. A. Internationalization means the program is designed so it can be adapted for multiple languages. By extracting the town names, this is exactly what has happened here, making Option A correct. Localization means the program actually supports multiple locales. There's no mention of multiple locales here, so Option B is incorrect. Similarly, there is no mention of multiple languages, making Option D incorrect. Finally, specialization is not a term relevant to properties, making Option C incorrect.

5. A. The `Properties` class is a `Map`, making III correct. `Hashtable` and `HashMap` are concrete classes rather than interfaces, so I and II are incorrect. While a `Properties` object is a `Hashtable`, this is not an interface. Since only III is correct, Option A is the answer.

6. C. Java supports properties file resource bundles and Java class resource bundles. Properties file resource bundles contain `String` keys and `String` values. Java class resource bundles contain `String` keys and any type of classes as values. Since both are valid, Option C is correct.

7. B. Calling `Locale.setDefault()` changes the default locale within the program. It does not change any settings on the computer. The next time you run a Java program, it will have the original default locale rather than the one you changed it to.

8. B. Line 18 prints the value for the property with the key `mystery`, which is `bag`. Line 19 prints a space. Line 20 doesn't find the key `more` so it prints `null`. Therefore, it prints `bag null`, and Option B is correct.

9. C. There is not a built-in class called `JavaResourceBundle`, making Options A and B incorrect. The `ListResourceBundle` class is used to programmatically create a resource bundle. It requires one method to be implemented named `getContents()`, making Option D incorrect and Option C correct. This method returns a 2D array of key/value pairs.

10. A. When both a language and country code are present, the language code comes first. The language code is in all lowercase letters and the country code is in all uppercase.

11. C. Java starts out by looking for a Java file with the most specific match, which is language and country code. Since this is happening at runtime, it is looking for the corresponding file with a `.class` extension. This matches Option C, making it the answer. If this file was not found, Java would then look for a `.properties` file with the name, which is Option D. If neither was found, it would continue dropping components of the name, eventually getting to Options A and B in that order.

12. A. This class correctly creates a Java class resource bundle. It extends `ListResourceBundle` and creates a 2D array as the property contents. Since `count` is an `int`, it is autoboxed into an `Integer`. In the `main()` method, it gets the resource bundle without a locale and requests the `count` key. Since `Integer` is a Java `Object`, it calls `getObject()` to get the value. The value is not incremented each time because the `getContents()` method is only called once. Therefore, Option A is correct.

13. A. A `Locale` can consist of a language only, making Option A correct. It cannot consist of a country code without a language, so Option B is incorrect. Finally, if both a language and country code are present, the language code is first, making Option C incorrect.

14. A. Java supports properties file resource bundles and Java class resource bundles. Properties file resource bundles require `String` values, making Option B incorrect. Java class resource bundles allow any type of classes as values. Since the question asks about defining values, it is the `.java` source code rather than the `.class` bytecode file, making Option A the answer.

15. C. At least one matching resource bundle must be available at the time of the call to `getBundle()`. While the requested key determines which of the resource bundles is used, at least one must exist simply to get the `ResourceBundle` reference, so Option C is the answer.

16. D. The `Properties` class implements `Map`. While the `get()` method, inherited from the superclass, is available, it returns an `Object`. Since `Object` cannot be cast to `String`, it does not compile, and Option D is the answer.

17. D. Java supports properties file resource bundles and Java class resource bundles. Both require `String` as the key format, so Option D is the answer.

18. A. Java starts out by looking for a Java file with the most specific match, which is language and country code. Since there is no such matching file, it drops the country code and looks for a match by language code. Java looks for bytecode files before properties files. Therefore, Option A is the answer. If it wasn't present, Option B would be the next choice. Options C and D would never be considered, as a locale doesn't just have a country code.