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.

16. 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.

17. 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.

18. 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.

19. 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.

20. 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`.

15. 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.

16. 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.

17. 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.

18. 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.

19. 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.

20. 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.