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.

.0. 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.

.2. 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.

.3. 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`.

4. 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.

5. 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.

6. 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.

8. 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.

9. 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.

0. 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.

8. 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.

9. 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.

0. 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.

1. 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.

2. 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.

3. 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.

4. 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.

5. 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.