Answer

1. (A) and (B)
   The `extends` clause is used to specify that a class extends another class. A subclass can be declared `abstract` regardless of whether the superclass was declared `abstract`. Private, overridden, and hidden members from the superclass are not inherited by the subclass. A class cannot be declared both `abstract` and `final`, since an abstract class needs to be extended to be useful, and a final class cannot be extended. The accessibility of the class is not limited by the accessibility of its members. A class with all the members declared `private` can still be declared `public`.

2. (B) and (E)
   The `Object` class has a public method named equals, but it does not have any method named length. Since all classes are subclasses of the `Object` class, they all inherit the `equals()` method. Thus, all Java objects have a public method named equals. In Java, a class can only extend a single superclass, but there is no limit on how many classes can extend a superclass.

3. (A), (B), and (D)
   `Bar` is a subclass of `Foo` that overrides the method `g()`. The statement `a.j = 5` is not legal, since the member `j` in the class `Bar` cannot be accessed through a `Foo` reference. The statement `b.i = 3` is not legal either, since the private member `i` cannot be accessed from outside of the class `Foo`.

4. (G)
   It is not possible to invoke the `doIt()` method in `A` from an instance method in class `C`. The method in `C` needs to call a method in a superclass two levels up in the inheritance hierarchy. The `super.super.doIt()` strategy will not work, since `super` is a keyword and cannot be used as an ordinary reference, nor accessed like a field. If the member to be accessed had been a field, the solution would be to cast the this reference to the class of the field and use the resulting reference to access the field. Field access is determined by the declared type of the reference, whereas the instance method to execute is determined by the actual type of the object denoted by the reference.

5. (B) and (G)
   `(A)` and `(1)` do not have covariant return types. `(B)` overrides `(2)`. The instance method in `(C)` cannot override the static method at `(4)`. The static method in `(D)` and the static method at `(4)` do not have compatible return types. The static method in `(E)` cannot override the instance method at `(3)`. The instance method in `(F)` and the instance method at `(5)` do not have compatible return types. The instance method in `(G)` overrides the instance method at `(6)`, and they have covariant return types.

6. (G)
   In the class `Car`, the static method `getModelName()` hides the static method of the same name in the superclass `Vehicle`. In the class `Car`, the instance method `getRegNo()` overrides the instance method of the same name in the superclass `Vehicle`. The declared type of the reference determines the method to execute when a static method is called, but the actual type of the object at runtime determines the method to execute when an overridden method is called.

7. (A), (C), and (D)
   Fields in interfaces declare named constants, and are always public, static, and final. None of these modifiers are mandatory in a constant declaration. All named constants must be explicitly initialized in the declaration.

8. (D)
   The code will compile without errors. The class `MyClass` declares that it implements the interfaces `Interface1` and `Interface2`. Since the class is declared abstract, it does not need to implement all

abstract method declarations defined in these interfaces. Any non-abstract subclasses of `MyClass` must provide the missing method implementations. The two interfaces share a common abstract method declaration `void g()`. `MyClass` provides an implementation for this abstract method declaration that satisfies both `Interface1` and `Interface2`. Both interfaces provide declarations of constants named `VAL_B`. This can lead to an ambiguity when referring to `VAL_B` by its simple name from `MyClass`. The ambiguity can be resolved by using fully qualified names: `Interface1.VAL_B` and `Interface2.VAL_B`. However, there are no problems with the code as it stands.

9. (A) and (C)
Declaration (B) fails, since it contains an illegal forward reference to its own named constant. The field type is missing in declaration (D). Declaration (E) tries illegally to use the protected modifier, even though named constants always have public accessibility. Such constants are implicitly public, static, and final.

10. (E)
Only the assignment `I1 b = obj3` is valid. The assignment is allowed, since `C3` extends `C1`, which implements `I1`. The assignment `obj2 = obj1` is not legal, since `C1` is not a subclass of `C2`. The assignments `obj3 = obj1` and `obj3 = obj2` are not legal, since neither `C1` nor `C2` is a subclass of `C3`. The assignment `I1 a = obj2` is not legal, since `C2` does not implement `I1`. Assignment `I2 c = obj1` is not legal, since `C1` does not implement `I2`.

11. (C)
Only `A a = d` is legal. The reference value in `d` can be assigned to `a`, since `D` implements `A`. The statements `c = d` and `d = c` are illegal, since there is no subtype-supertype relationship between `C` and `D`. Even though a cast is provided, the statement `d = (D) c` is illegal. The object referred to by `c` cannot possibly be of type `D`, since `D` is not a subclass of `C`. The statement `c = b` is illegal, since assigning a reference value of a reference of type `B` to a reference of type `C` requires a cast.

12. (A)
The program will print all the letters `I`, `J`, `C`, and `D`, when run. The object referred to by the reference `x` is of class `D`. Class `D` extends class `C` and class `C` implements interface `I`. This makes `I`, `J`, and `C` supertypes of class `D`. The reference value of an object of class `D` can be assigned to any reference of its supertypes and is, therefore, an instanceof these types.

13. (C)
The program will print 1 when run. The `f()` methods in `A` and `B` are private and are not accessible by the subclasses. Because of this, the subclasses cannot overload or override these methods, but simply define new methods with the same signature. The object being called is of the class `C`. The reference used to access the object is of the type `B`. Since `B` contains a method `g()`, the method call will be allowed at compile time. During execution it is determined that the object is of the class `C`, and dynamic method lookup will cause the overridden method `g()` in `B` to be executed. This method calls a method named `f`. It can be determined during compilation that this can only refer to the `f()` method in `B`, since the method is private and cannot be overridden. This method returns the value 1, which is printed.