**24.** B, C, E. *value1* is a `final` instance variable. It can only be set once: in the variable declaration, an instance initializer, or a constructor. Option A does not compile because the `final` variable was already set in the declaration. *value2* is a `static` variable. Both instance and static initializers are able to access `static` variables, making options B and E correct. *value3* is an instance variable. Options D and F do not compile because a `static` initializer does not have access to instance variables.

**25.** A, E. The `100` parameter is an `int` and so calls the matching `int` constructor. When this constructor is removed, Java looks for the next most specific constructor. Java prefers autoboxing to varargs, and so chooses the `Integer` constructor. The `100L` parameter is a `long`. Since it can't be converted into a smaller type, it is autoboxed into a `Long` and then the constructor for `Object` is called.

**26.** A. This code is correct. Line 8 creates a lambda expression that checks if the age is less than 5. Since there is only one parameter and it does not specify a type, the parentheses around the type parameter are optional. Line 10 uses the `Predicate` interface, which declares a `test()` method.

**27.** C. The interface takes two `int` parameters. The code on line 7 attempts to use them as if one is a `StringBuilder`. It is tricky to use types in a lambda when they are implicitly specified. Remember to check the interface for the real type.

**28.** A, D, F. `removeIf()` expects a `Predicate`, which takes a parameter list of one parameter using the specified type. Options B and C are incorrect because they do not use the `return` keyword. It is required inside braces for lambda bodies. Option E is incorrect because it is missing the parentheses around the parameter list. This is only optional for a single parameter with an inferred type.

**29.** A, F. Option B is incorrect because it does not use the `return` keyword. Options C, D, and E are incorrect because the variable `e` is already in use from the lambda and cannot be redefined. Additionally, option C is missing the `return` keyword and option E is missing the semicolon.

# Chapter 5: Class Design

**1.** B. All interface methods are implicitly `public`, so option B is correct and option A is not. Interface methods may be declared as `static` or `default` but are never implicitly added, so options C and F are incorrect. Option D is incorrect—`void` is not a modifier; it is a return type. Option E is a tricky one, because prior to Java 8 all interface methods would be assumed to be `abstract`. Since Java 8 now includes default and static methods and they are never abstract, you cannot assume the `abstract` modifier will be implicitly applied to all methods by the compiler.

**2.** E. The code will not compile because the parent class `Mammal` doesn't define a no-argument constructor, so the first line of a `Platypus` constructor should be an explicit call to `super(int age)`. If there was such a call, then the output would be `MammalPlatypus`, since the super constructor is executed before the child constructor.

3. A, B, D, E. The blank can be filled with any class or interface that is a supertype of `TurtleFrog`. Option A is a superclass of `TurtleFrog`, and option B is the same class, so both are correct. `BrazilianHornedFrog` is not a superclass of `TurtleFrog`, so option C is incorrect. `TurtleFrog` inherits the `CanHope` interface, so option D is correct. All classes inherit `Object`, so option E is correct. Finally, `Long` is an unrelated class that is not a superclass of `TurtleFrog`, and is therefore incorrect.

4. C, E. The code doesn't compile, so option A is incorrect. Option B is also not correct because the rules for overriding a method allow a subclass to define a method with an exception that is a subclass of the exception in the parent method. Option C is correct because the return types are not covariant; in particular, `Number` is not a subclass of `Integer`. Option D is incorrect because the subclass defines a method that is more accessible than the method in the parent class, which is allowed. Finally, option E is correct because the method is declared as `static` in the parent class and not so in the child class. For nonprivate methods in the parent class, both methods must use `static` (hide) or neither should use `static` (override).

5. A, D, E, F. First off, options B and C are incorrect because protected and public methods may be overridden, not hidden. Option A is correct because private methods are always hidden in a subclass. Option D is also correct because static methods cannot be overridden, only hidden. Options E and F are correct because variables may only be hidden, regardless of the access modifier.

6. D. The code fails to compile because `Beetle`, the first concrete subclass, doesn't implement `getNumberOfSections()`, which is inherited as an abstract method; therefore, option D is correct. Option B is incorrect because there is nothing wrong with this interface method definition. Option C is incorrect because an abstract class is not required to implement any abstract methods, including those inherited from an interface. Option E is incorrect because the code fails at compilation-time.

7. B, C. A reference to an object requires an explicit cast if referenced with a subclass, so option A is incorrect. If the cast is to a superclass reference, then an explicit cast is not required. Because of polymorphic parameters, if a method takes the superclass of an object as a parameter, then any subclass references may be used without a cast, so option B is correct. All objects extend `java.lang.Object`, so if a method takes that type, any valid object, including `null`, may be passed; therefore, option C is correct. Some cast exceptions can be detected as errors at compile-time, but others can only be detected at runtime, so D is incorrect. Due to the nature of polymorphism, a `public` instance method can be overridden in a subclass and calls to it will be replaced even in the superclass it was defined, so E is incorrect.

8. F. The interface variable amount is correctly declared, with `public` and `static` being assumed and automatically inserted by the compiler, so option B is incorrect. The method declaration for `eatGrass()` on line 3 is incorrect because the method has been marked as `static` but no method body has been provided. The method declaration for `chew()` on line 4 is also incorrect, since an interface method that provides a body must be marked as `default` or `static` explicitly. Therefore, option F is the correct answer since this code contains two compile-time errors.

**9.** A. Although the definition of methods on lines 2 and 5 vary, both will be converted to `public abstract` by the compiler. Line 4 is fine, because an interface can have `public` or default access. Finally, the class `Falcon` doesn't need to implement the interface methods because it is marked as `abstract`. Therefore, the code will compile without issue.

**10.** B, C, E, F. Option A is wrong, because an abstract class may contain concrete methods. Since Java 8, interfaces may also contain concrete methods in form of static or default methods. Although all variables in interfaces are assumed to be `public static final`, abstract classes may contain them as well, so option B is correct. Both abstract classes and interfaces can be extended with the `extends` keyword, so option C is correct. Only interfaces can contain default methods, so option D is incorrect. Both abstract classes and interfaces can contain static methods, so option E is correct. Both structures require a concrete subclass to be instantiated, so option F is correct. Finally, though an instance of an object that implements an interface inherits `java.lang.Object`, the interface itself doesn't; otherwise, Java would support multiple inheritance for objects, which it doesn't. Therefore, option G is incorrect.

**11.** A, D, E. Interface variables are assumed to be `public static final`; therefore, options A, D, and E are correct. Options B and C are incorrect because interface variables must be public—interfaces are implemented by classes, not inherited by interfaces. Option F is incorrect because variables can never be abstract.

**12.** B. This code compiles and runs without issue, outputting `false`, so option B is the correct answer. The first declaration of `isBlind()` is as a default interface method, assumed `public`. The second declaration of `isBlind()` correctly overrides the default interface method. Finally, the newly created `Owl` instance may be automatically cast to a `Nocturnal` reference without an explicit cast, although adding it doesn't break the code.

**13.** A. The code compiles and runs without issue, so options E and F are incorrect. The `printName()` method is an overload in `Spider`, not an override, so both methods may be called. The call on line 8 references the version that takes an `int` as input defined in the `Spider` class, and the call on line 9 references the version in the `Arthropod` class that takes a `double`. Therefore, `SpiderArthropod` is output and option A is the correct answer.

**14.** C. The code compiles without issue, so option A is wrong. Option B is incorrect, since an abstract class could implement `HasVocalCords` without the need to override the `makeSound()` method. Option C is correct; any class that implements `CanBark` automatically inherits its methods, as well as any inherited methods defined in the parent interface. Because option C is correct, it follows that option D is incorrect. Finally, an interface can extend multiple interfaces, so option E is incorrect.

**15.** B. Concrete classes are, by definition, not abstract, so option A is incorrect. A concrete class must implement all inherited abstract methods, so option B is correct. Option C is incorrect; a superclass may have already implemented an inherited interface, so the concrete subclass would not need to implement the method. Concrete classes can be both `final` and not `final`, so option D is incorrect. Finally, abstract methods must be overridden by a concrete subclass, so option E is incorrect.

16. E. The code doesn't compile, so options A and B are incorrect. The issue with line 9 is that `layEggs()` is marked as `final` in the superclass `Reptile`, which means it cannot be overridden. There are no errors on any other lines, so options C and D are incorrect.
17. B. This may look like a complex question, but it is actually quite easy. Line 2 contains an invalid definition of an abstract method. Abstract methods cannot contain a body, so the code will not compile and option B is the correct answer. If the body `{}` was removed from line 2, the code would still not compile, although it would be line 8 that would throw the compilation error. Since `dive()` in `Whale` is abstract and `Orca` extends `Whale`, then it must implement an overridden version of `dive()`. The method on line 9 is an overloaded version of `dive()`, not an overridden version, so `Orca` is an invalid subclass and will not compile.
18. E. The code doesn't compile because line 6 contains an incompatible override of the `getNumberOfGills(int input)` method defined in the `Aquatic` interface. In particular, `int` and `String` are not covariant returns types, since `int` is not a subclass of `String`. Note that line 5 compiles without issue; `getNumberOfGills()` is an overloaded method that is not related to the parent interface method that takes an `int` value.
19. A, C, F. First off, `Cobra` is a subclass of `Snake`, so option A can be used. `GardenSnake` is not defined as a subclass of `Snake`, so it cannot be used and option B is incorrect. The class `Snake` is not marked as `abstract`, so it can be instantiated and passed, so option C is correct. Next, `Object` is a superclass of `Snake`, not a subclass, so it also cannot be used and option D is incorrect. The class `String` is unrelated in this example, so option E is incorrect. Finally, a `null` value can always be passed as an object value, regardless of type, so option F is correct.
20. A. The code compiles and runs without issue, so options C, D, and E are incorrect. The trick here is that the method `fly()` is marked as `private` in the parent class `Bird`, which means it may only be hidden, not overridden. With hidden methods, the specific method used depends on where it is referenced. Since it is referenced within the `Bird` class, the method declared on line 2 was used, and option A is correct. Alternatively, if the method was referenced within the `Pelican` class, or if the method in the parent class was marked as `protected` and overridden in the subclass, then the method on line 9 would have been used.

# Chapter 6: Exceptions

1. B. Runtime exceptions are also known as unchecked exceptions. They are allowed to be declared, but they don't have to be. Checked exceptions must be handled or declared. Legally, you can handle `java.lang.Error` subclasses, but it's not a good idea.
2. B, D. In a method declaration, the keyword `throws` is used. To actually throw an exception, the keyword `throw` is used and a new exception is created.
3. C. A `try` statement is required to have a `catch` clause and/or `finally` clause. If it goes the `catch` route, it is allowed to have multiple `catch` clauses.

4. B. The second line tries to cast an `Integer` to a `String`. Since `String` does not extend `Integer`, this is not allowed and a `ClassCastException` is thrown.
5. A, B, D. `java.io.IOException` is thrown by many methods in the `java.io` package, but it is always thrown programmatically. The same is true for `NumberFormatException`; it is thrown programmatically by the wrapper classes of `java.lang`. The other three exceptions are all thrown by the JVM when the corresponding problem arises.
6. C. The compiler tests the operation for a valid type but not a valid result, so the code will still compile and run. At runtime, evaluation of the parameter takes place before passing it to the `print()` method, so an `ArithmeticException` object is raised.
7. C. The `main()` method invokes `go` and `A` is printed on line 3. The `stop` method is invoked and `E` is printed on line 14. Line 16 throws a `NullPointerException`, so `stop` immediately ends and line 17 doesn't execute. The exception isn't caught in `go`, so the `go` method ends as well, but not before its `finally` block executes and `C` is printed on line 9. Because `main()` doesn't catch the exception, the stack trace displays and no further output occurs, so `AEC` was the output printed before the stack trace.
8. E. The order of `catch` blocks is important because they're checked in the order they appear after the `try` block. Because `ArithmeticException` is a child class of `RuntimeException`, the `catch` block on line 7 is unreachable. (If an `ArithmeticException` is thrown in `try` try block, it will be caught on line 5.) Line 7 generates a compiler error because it is unreachable code.
9. B. The `main()` method invokes `start` on a new `Laptop` object. Line 4 prints `Starting up`; then line 5 throws an `Exception`. Line 6 catches the exception, line 7 prints `Problem`, and then line 8 calls `System.exit`, which terminates the JVM. The `finally` block does not execute because the JVM is no longer running.
10. E. The `parseName` method is invoked within `main()` on a new `Dog` object. Line 4 prints `1`. The `try` block executes and `2` is printed. Line 7 throws a `NumberFormatException`, so line 8 doesn't execute. The exception is caught on line 9, and line 10 prints `4`. Because the exception is handled, execution resumes normally. `parseName` runs to completion, and line 17 executes, printing `5`. That's the end of the program, so the output is `1245`.
11. A. The `parseName` method is invoked on a new `Cat` object. Line 4 prints `1`. The `try` block is entered, and line 6 prints `2`. Line 7 throws a `NumberFormatException`. It isn't caught, so `parseName` ends. `main()` doesn't catch the exception either, so the program terminates and the stack trace for the `NumberFormatException` is printed.
12. A, B, D, G. The `main()` method invokes run on a new `Mouse` object. Line 4 prints `1` and line 6 prints `2`, so options A and B are correct. Line 7 throws a `NullPointerException`, which causes line 8 to be skipped, so C is incorrect. The exception is caught on line 9 and line 10 prints `4`, so option D is correct. Line 11 throws the exception again, which causes `run()` to immediately end, so line 13 doesn't execute and option E is incorrect. The `main()` method doesn't catch the exception either, so line 18 doesn't execute and option F is incorrect. The uncaught `NullPointerException` causes the stack trace to be printed, so option G is correct.

**13.** A, B, C, E. Classes listed in the `throws` part of a method declaration must extend `java.lang.Throwable`. This includes `Error`, `Exception`, and `RuntimeException`. Arbitrary classes such as `String` can't go there. Any Java type, including `Exception`, can be declared as the return type. However, this will simply return the object rather than throw an exception.

**14.** A, C, D, E. A method that declares an exception isn't required to throw one, making option A correct. Runtime exceptions can be thrown in any method, making options C and E correct. Option D matches the exception type declared and so is also correct. Option B is incorrect because a broader exception is not allowed.

**15.** A, B, D, E. `ArrayIndexOutOfBoundsException`, `IllegalArgumentException`, and `NumberFormatException` are runtime exceptions. Sorry, you have to memorize them. Any class that extends `RuntimeException` is a runtime (unchecked) exception. Classes that extend `Exception` but not `RuntimeException` are checked exceptions.

**16.** B. `IllegalArgumentException` is used when an unexpected parameter is passed into a method. Option A is incorrect because returning `null` or `-1` is a common return value for this scenario. Option D is incorrect because a `for` loop is typically used for this scenario. Option E is incorrect because you should find out how to code the method and not leave it for the unsuspecting programmer who calls your method. Option C is incorrect because you should run!

**17.** A, C, D, E. The method is allowed to throw no exceptions at all, making option A correct. It is also allowed to throw runtime exceptions, making options D and E correct. Option C is also correct since it matches the signature in the interface.

**18.** A, B, C, E. Checked exceptions are required to be handled or declared. Runtime exceptions are allowed to be handled or declared. Errors are allowed to be handled or declared, but this is bad practice.

**19.** C, E. Option C is allowed because it is a more specific type than `RuntimeException`. Option E is allowed because it isn't in the same inheritance tree as `RuntimeException`. It's not a good idea to catch either of these. Option B is not allowed because the method called inside the `try` block doesn't declare an `IOException` to be thrown. The compiler realizes that `IOException` would be an unreachable `catch` block. Option D is not allowed because the same exception can't be specified in two different `catch` blocks. Finally, option A is not allowed because it's more general than `RuntimeException` and would make that block unreachable.

**20.** A, E. The code begins normally and prints `a` on line 13, followed by `b` on line 15. On line 16, it throws an exception that's caught on line 17. Remember, only the most specific matching `catch` is run. Line 18 prints `c`, and then line 19 throws another exception. Regardless, the `finally` block runs, printing `e`. Since the `finally` block also throws an exception, that's the one printed.