Chapter7 Beyond Classes

Tuesday, October 14, 2025 12:01 PM

Declaring and Using an Interface

In Java, an interface is defined with the `interface` keyword, analogous to the `class` keyword used when defining a class. Refer to Figure 7.1 for a proper interface declaration.

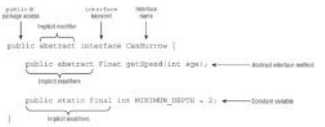


FIGURE 7.1 Defining an interface

Inheriting an Interface

Like an abstract class, when a concrete class inherits an interface, all of the inherited abstract methods must be implemented. We illustrate this principle in Figure 7.3. How many abstract methods does the concrete `Swan` class inherit?

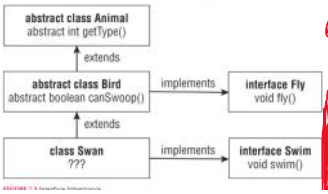


TABLE 7.1 Interface member types

	Membership type	Required modifiers	Implicit modifiers	Has value or body?
Constant variable	Class	—	`public static final`	Yes
`abstract` method	Instance	—	`public abstract`	No
`default` method	Instance	`default`	`public`	Yes
`static` method	Class	`static`	`public`	Yes
`private` method	Instance	`private`	—	Yes
`private static` method	Class	`private static`	—	Yes

```
public interface Schedule {
   default void wakeUp()        { checkTime(7);  }
   private void haveBreakFast() { checkTime(9);  }
   static void workOut()        { checkTime(18); }
   private static void checkTime(int hour) {
      if (hour > 17) {
         System.out.println("You're late!");
      } else {
         System.out.println("You have "+(17-hour)+" hours left "
               + "to make the appointment");
      }
   }
}
```

Using these rules, which of the following methods do not compile?

```
public interface ZooTrainTour {
   abstract int getTrainName();
   private static void ride() {}
   default void playHorn() { getTrainName(); ride(); }
   public static void slowDown() { playHorn(); }
   static void speedUp() { ride(); }
}
```

The `ride()` method is `private` and `static`, so it can be accessed by any `default` or `static` method within the interface declaration. The `getTrainName()` is `abstract`, so it can be accessed by a `default` method associated with the instance. The `slowDown()` method is `static`, though, and cannot call a `default` or `private` method, such as `playHorn()`, without an explicit reference object. Therefore, the `slowDown()` method does not compile.

Unlike a class, which can extend only one class, an interface can extend multipl interfaces.

```
public interface Nocturnal {
   public int hunt();
}

public interface CanFly {
   public void flap();
}

public interface HasBigEyes extends Nocturnal, CanFly {}

public class Owl implements HasBigEyes {
   public int hunt() { return 5; }
   public void flap() { System.out.println("Flap!"); }
}
```

```
public interface CanRun {}
public class Cheetah extends CanRun {}   // DOES NOT COMPILE

public class Hyena {}
public interface HasFur extends Hyena {} // DOES NOT COMPILE
```

Be wary of exam questions that mix class and interface declarations.

The following is an example of an incompatible declaration:

```
public interface Herbivore { public void eatPlants(int plantsLeft); }

public interface Omnivore  { public int eatPlants(int foodRemaining); }

public class Tiger implements Herbivore, Omnivore { // DOES NOT COMP
   // Doesn't matter!
}
```

The implementation of `Tiger` doesn't matter in this case since it's impossible to write a version of `Tiger` that satisfies both inherited `abstract` methods. The code does not compile, regardless of what is declared inside the `Tiger` class.

```
public interface Walk {
   public default int getSpeed() { return 5; }
}

public interface Run {
   public default int getSpeed() { return 10; }
}

public class Cat implements Walk, Run {}  // DOES NOT COMPILE
```

```
public class Cat implements Walk, Run {
   public int getSpeed() {
      return 1;
   }

   public int getWalkSpeed() {
      return Walk.super.getSpeed();
   } }
```

This is an area where a `default` method `getSpeed()` exhibits properties of both an instance and `static` method. We use the interface name to indicate which method we want to call, but we use the `super` keyword to show that we are following instance inheritance, not class inheritance. Note that calling `Walk.this.getSpeed()` would not have worked. A bit confusing, we know, *but you need to be familiar with this syntax for the exam.*

`abstract int getTrainName();`
is an **abstract instance method that must be implemented by any class** implementing `ZooTrainTour`.

Even though it has no body, **it's still a perfectly valid method call** in code that will later be executed on an actual object that does implement it.

So in the line:

```
java                                                    Copy code
getTrainName();
```

inside `playHorn()`, we're saying:

> "When an object that implements this interface calls `playHorn()`, execute `getTrainName()` on that same object."

That's 100 % fine, because by the time `playHorn()` actually runs, the object is guaranteed to have a concrete implementation of `getTrainName()`.

ENUMS

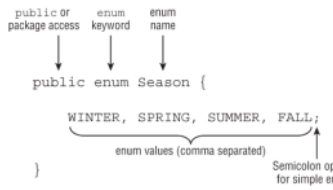


FIGURE 7.4 Defining a simple enum

```
System.out.print("begin,");
var firstCall = SeasonWithVisitors.SUMMER;   // Prints 4 times
System.out.print("middle,");
var secondCall = SeasonWithVisitors.SUMMER;  // Doesn't print anythi
System.out.print("end");
```

One thing that you can't do is extend an enum.

```
public enum ExtendedSeason extends Season {} // DOES NOT COMPILE
```

```
for(var season: Season.values()) {
   System.out.println(season.name() + " " + season.ordinal());
}
```

The `ordinal()` method returns an `int` value, which denotes the order in which the value is declared in the enum:

```
WINTER 0
SPRING 1
SUMMER 2
FALL 3
```

```
if (Season.SUMMER == 2) {} // DOES NOT COMPILE
```

An enum provides a useful `valueOf()` method for converting from a `String` to

RULES!

The following list includes the implicit modifiers for interfaces that you need to know for the exam:

- Interfaces are implicitly `abstract`.
- Interface variables are implicitly `public`, `static`, and `final`.
- Interface methods without a body are implicitly `abstract`.
- Interface methods without the `private` modifier are implicitly `public`.

The last rule applies to `abstract`, `default`, and `static` interface methods, which we cover in the next section.

Default Interface Method Definition Rules

1. A `default` method may be declared only within an interface.
2. A `default` method must be marked with the `default` keyword and include a method body.
3. A `default` method is implicitly `public`.
4. A `default` method cannot be marked `abstract`, `final`, or `static`.
5. A `default` method may be overridden by a class that implements the interface.
6. If a class inherits two or more `default` methods with the same method signature, then the class must override the method.

Static Interface Method Definition Rules

1. A `static` method must be marked with the `static` keyword and include a method body.
2. A `static` method without an access modifier is implicitly `public`.
3. A `static` method cannot be marked `abstract` or `final`.
4. A `static` method is not inherited and cannot be accessed in a class implementing the interface without a reference to the interface name.

While Table 7.2 might seem like a lot to remember, here are some quick tips for the exam:

- Treat `abstract`, `default`, and non-`static` `private` methods as belonging to an instance of the interface.
- Treat `static` methods and variables as belonging to the interface class object.
- All `private` interface method types are only accessible within the interface declaration.

ENUMS

There are a few things to notice here. On line 23, the list of enum values ends with a semicolon (`;`). While this is optional for a simple enum, it is required if there is anything in the enum besides the values. Lines 25–33 are regular Java

Sealed classes

Specifying the Subclass Modifier

While some types, like interfaces, have a certain number of implicit modifiers, sealed classes do not. *Every class that directly extends a sealed class must specify exactly one of the following three modifiers:* `final`, `sealed`, *or* `non-sealed`. Remember this rule for the exam!

Location of direct subclasses	`permits` clause
In a different file from the sealed class	Required
In the same file as the sealed class	Permitted, but not required
Nested inside of the sealed class	Permitted, but not required

For this reason, interfaces that extend a sealed interface can only be marked `sealed` or `non-sealed`. They cannot be marked `final`.

Sealed Class Rules

- Sealed classes are declared with the `sealed` and `permits` modifiers.
- Sealed classes must be declared in the same package or named module as their direct subclasses.
- Direct subclasses of sealed classes must be marked `final`, `sealed`, or `non-sealed`. For interfaces that extend a sealed interface, only `sealed` and `non-sealed` modifiers are permitted.
- The `permits` clause is optional if the sealed class and its direct subclasses are declared within the same file or the subclasses are nested within the sealed class.
- Interfaces can be sealed to limit the classes that implement them or the interfaces that extend them.

Records

Members Automatically Added to Records

- **Constructor** : A constructor with the parameters in the same order as the record declaration
- **Accessor method** : One accessor for each field
- `equals()` : A method to compare two elements that returns `true` if each field is equal in terms of `equals()`
- `hashCode()` : A consistent `hashCode()` method using all of the fields
- `toString()` : A `toString()` implementation that prints each field of the record in a convenient, easy-to-read format

The first line of an overloaded constructor must be an explicit call to another constructor via `this()`. If there are no other constructors, the long constructor must be called. Contrast this with what you learned about in Chapter 6, where calling `super()` or `this()` was often optional in constructor declarations. Also, unlike compact constructors, you can only transform the data on the first line. After the first line, all of the fields will already be assigned, and the object is immutable.

```
public record Crane(int numberEggs, String name) {
   public Crane(int numberEggs, String firstName, String lastName) {
```

Tips

Another way to think of it is that a `private` interface method is only accessible to non-`static` methods defined within the interface. A `private static` interface method, on the other hand, can be accessed by any method in the interface. For both types of `private` methods, a class inheriting the interface cannot directly invoke them.

```
public sealed class Snake permits Cobra {  // DOES NOT COMPILE
   final class Cobra extends Snake {}
}
```

This code does not compile because `Cobra` requires a reference to the `Snake` namespace. The following fixes this issue:

```
public sealed class Snake permits Snake.Cobra {
   final class Cobra extends Snake {}
}
```

Records

Fun fact: it is legal to have a record without any fields. It is simply declared with the `record` keyword and parentheses:

```
public record Crane() {}
```

This is not the kind of thing you'd use in your own code, but it could come up on the exam.

```
System.out.print("begin,");
var firstCall = SeasonWithVisitors.SUMMER;   // Prints 4 times
System.out.print("middle,");
var secondCall = SeasonWithVisitors.SUMMER;  // Doesn't print anythi
System.out.print("end");
```

```
public enum SeasonWithTimes {
   WINTER {
      public String getHours() { return "10am-3pm"; }
   },
   SPRING {
      public String getHours() { return "9am-5pm"; }
   },
   SUMMER {
      public String getHours() { return "9am-7pm"; }
   },
   FALL {
      public String getHours() { return "9am-5pm"; }
   };
   public abstract String getHours();
}
```

What's going on here? It looks like we created an `abstract` class and a bunch of tiny subclasses. In a way, we are. The enum itself has an `abstract` method. This means that each and every enum value is required to implement this method. If we forget to implement the method for one of the values, we get a compiler error:

```
The enum constant WINTER must implement the abstract method getHours
```

```
SPRING 1
SUMMER 2
FALL 3

if (Season.SUMMER == 2) {} // DOES NOT COMPILE
```

An enum provides a useful `valueOf()` method for converting from a `String` to an enum value. This is helpful when working with older code or parsing user input. The `String` passed in must match the enum value exactly, though.

```
Season s = Season.valueOf("SUMMER"); // SUMMER
Season t = Season.valueOf("summer"); // IllegalArgumentException
```

compact constructors, you can only transform the data on the first line. After the first line, all of the fields will already be assigned, and the object is immutable.

```
public record Crane(int numberEggs, String name) {
   public Crane(int numberEggs, String firstName, String lastName) {
      this(numberEggs + 1, firstName + " " + lastName);
      numberEggs = 10; // NO EFFECT (applies to parameter, not instar
      this.numberEggs = 20; // DOES NOT COMPILE
   }
}
```

Only the long constructor, with fields that match the record declaration, supports setting field values with a `this` reference. Compact and overloaded constructors do not.

For the exam, you should be aware of the following rules when working with pattern matching and records:

- If any field declared in the record is included, then all fields must be included.
- The order of fields must be the same as in the record.
- The names of the fields do not have to match.
- At compile time, the type of the field must be compatible with the type declared in the record.
- The pattern may not match at runtime if the record supports elements of various types.

- Overloaded and compact constructors
- Instance methods including overriding any provided methods (accessors, `equals()`, `hashCode()`, `toString()`)
- Nested classes, interfaces, annotations, enums, and records

SEALD CLASSES

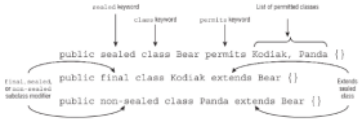


FIGURE 7.8 Defining a sealed class

Compiling Sealed Classes

Let's say we create a `Penguin` class and compile it in a new package without any other source code. With that in mind, does the following compile?

```
// Penguin.java
package zoo;
public sealed class Penguin permits Emperor {}
```

No, it does not! Why? The answer is that a sealed class needs to be declared (and compiled) in the same package as its direct subclasses. But what about the subclasses themselves? They must each extend the sealed class. For example, the following two declarations do not compile:

```
// Penguin.java
package zoo;
public sealed class Penguin permits Emperor {}  // DOES NOT COMPILE

// Emperor.java
package zoo;
public final class Emperor {}
```

```
// Sealed interface
public sealed interface Swims permits Duck, Swan, Floats {}

// Classes permitted to implement sealed interface
public final class Duck implements Swims {}
public final class Swan implements Swims {}

// Interface permitted to extend sealed interface
public non-sealed interface Floats extends Swims {}
```

Nested Classes

A *nested class* is a class that is defined within another class. A nested class can come in one of four flavors, with all supporting instance and `static` variables members.

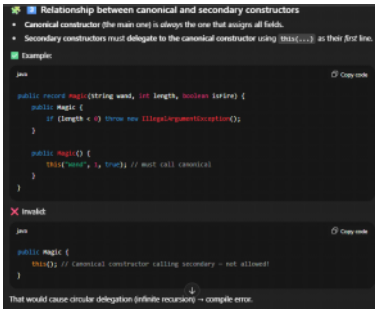
- *Inner class*: A non-`static` type defined at the member level of a class
- *Static nested class*: A `static` type defined at the member level of a class
- *Local class*: A class defined within a method body
- *Anonymous class*: A special case of a local class that does not have a name

Inner classes have the following properties:

- Can be declared `public`, `protected`, package, or `private`
- Can extend a class and implement interfaces
- Can be marked `abstract` or `final`
- Can access members of the outer class, including `private` members

Records

```
public record Crane(int numberEggs, String name) {
   public Crane(int numberEggs, String name) {}  // DOES NOT COMPILE
}
```



Local Classes

Local classes have the following properties:

- Do not have an access modifier.
- Can be declared `final` or `abstract`.
- Can include instance and `static` members.
- Have access to all fields and methods of the enclosing class (when defined in an instance method).
- Can access `final` and effectively final local variables.

Compact Constructors

Luckily, the authors of Java added the ability to define a compact constructor for records. A *compact constructor* is a special type of constructor used for records to process validation and transformations succinctly. It takes no parameters and implicitly sets all fields. Figure 7.7 shows an example of a compact constructor.

FIGURE 7.7 Declaring a compact constructor

While compact constructors can modify the constructor parameters, *they cannot modify the fields of the record.* For example, this does not compile:

```
public record Crane(int numberEggs, String name) {
   public Crane {
      this.numberEggs = 10;  // DOES NOT COMPILE
   }
}
```

Like enums, that means you can't extend or inherit a record.

```
public record BlueCrane() extends Crane {}  // DOES NOT COMPILE
```

Permitted modifiers	Inner class	`static` nested class	Local class	Anonymous class
Access modifiers	All	All	None	None
`abstract`	Yes	Yes	Yes	No
`final`	Yes	Yes	Yes	No

```
if(c instanceof Couple(Bear a, Bear b)) {
   System.out.print(a.name() + " " + b.name());
}
if(c instanceof Couple(Bear(String firstName, List<String> f),
      Bear b)) {
   System.out.print(firstName + " " + b.name());
}
if(c instanceof Couple(Bear(String name, List<String> f1),
      Bear(String name, List<String> f2))) {
   System.out.print(name + " " + name);
}
```

```
public record Crane(int numberEggs, String name) {
   private static int TYPE = 10;
   public int size;                          // DOES NOT COMPILE
   private final boolean friendly = true;  // DOES NOT COMPILE
}
```

Records also do not support instance initializers. All initialization for the fields of a record must happen in a constructor. They do support `static` initializers, though.

```
public record Crane(int numberEggs, String name) {
  static { System.out.print("Hello Bird!"); }
  { System.out.print("Goodbye Bird!"); }  // DOES NOT COMPILE
  { this.name = "Big"; }                  // DOES NOT COMPILE
}
```

	Inner class	`static` nested class	Local class	Anonymous class
Can include instance and `static` members?	Yes	Yes	Yes	Yes
Can extend a class or implement any number of interfaces?	Yes	Yes	Yes	No—must have exactly one superclass or one interface
Can access instance members of enclosing class?	Yes	No	Yes (if declared in an instance method)	Yes (if declared in an instance method)
Can access local variables of	N/A	N/A	Yes (if `final` or effectively	Yes (if `final` or effectively final)

Nested Classes

FIGURE 7.7 Declaring a compact constructor

```
if(c instanceof Couple(Bear a, Bear b)) {
   System.out.print(a.name() + " " + b.name());
}
if(c instanceof Couple(Bear(String firstName, List<String> f),
      Bear b)) {
   System.out.print(firstName + " " + b.name());
}
if(c instanceof Couple(Bear(String name, List<String> f1),
      Bear(String name, List<String> f2))) {
   System.out.print(name + " " + name);
}
```

```
public record Crane(int numberEggs, String name) {
   private static int TYPE = 10;
   public int size;                        // DOES NOT COMPILE
   private final boolean friendly = true;  // DOES NOT COMPILE
}
```

Records also do not support instance initializers. All initialization for the fields of a record must happen in a constructor. They do support `static` initializers, though.

```
public record Crane(int numberEggs, String name) {
  static { System.out.print("Hello Bird!"); }
  { System.out.print("Goodbye Bird!"); }  // DOES NOT COMPILE
  { this.name = "Big"; }                  // DOES NOT COMPILE
}
```

Nested Classes

Instantiating an Instance of an Inner Class

There is another way to instantiate `Room` that looks odd at first. OK, well, maybe not just at first. This syntax isn't used often enough to get used to it:

```
20:    public static void main(String[] args) {
21:       var home = new Home();
22:       Room room = home.new Room();  // Create the inner class ins
23:       room.enter();
24:    }
```

Let's take a closer look at lines 21 and 22. We need an instance of `Home` to create a `Room`. We can't just call `new Room()` inside the `static main()` method, because Java won't know which instance of `Home` it is associated with. Java solves this by calling `new` as if it were a method on the `home` variable. We can shorten lines 21–23 to a single line:

```
21:       new Home().new Room().enter();  // Sorry, it looks ugly to
```

```
1:  public class A {
2:     private int x = 10;
3:     class B {
4:        private int x = 20;
5:        class C {
6:           private int x = 30;
7:           public void allTheX() {
8:              System.out.println(x);        // 30
9:              System.out.println(this.x);   // 30
10:             System.out.println(B.this.x); // 20
11:             System.out.println(A.this.x); // 10
12:    } } }
13:    public static void main(String[] args) {
14:       A a = new A();
15:       A.B b = a.new B();
16:       A.B.C c = b.new C();
17:       c.allTheX();
18: }}
```

```
public class Fox {
   private class Den {}
   public void goHome() {
      new Den();
   }
   public static void visitFriend() {
      new Den();  // DOES NOT COMPILE
   }
}

public class Squirrel {
   public void visitFox() {
      new Den();  // DOES NOT COMPILE
   }
}
```

Let's take a look at an example:

```
1: public class Park {
2:    static class Ride {
3:       private int price = 6;
4:    }
5:    public static void main(String[] args) {
6:       var ride = new Ride();
7:       System.out.println(ride.price);
8: } }
```

Line 6 instantiates the nested class. Since the class is `static`, you do not need an instance of `Park` to use it. You are allowed to access `private` instance variables, as shown on line 7.

```
11: class Emu1 {
12:    String name = "Emmy";
13:    static Feathers createFeathers() {
14:       return new Feathers("grey");
15:    }
16:    record Feathers(String color) {
17:       void fly() {
18:          System.out.print(name + "  is flying");  // DOES NOT
19:       } } }
20:
21: class Emu2 {
22:    String name = "Emmy";
23:    static Feathers createFeathers() {
24:       return new Feathers("grey");  // DOES NOT COMPILE
25:    }
26:    class Feathers {
27:       void fly() {
28:          System.out.print(name + "  is flying");
29:       } } }
```

Line 14 compiles without issue because the record is implicitly `static`. Line 24 does not compile, though, as the class version of `Feathers` is not `static` and would require an instance of `Emu2` to create. Likewise, the outer variable, `name`, is only visible to the nested class if it is not `static`, as shown by line 28 compiling and line 18 not compiling.

Local Classes

	Inner class	`static` nested class	Local class	Anonymous class
Can include instance and `static` members?	Yes	Yes	Yes	Yes
Can extend a class or implement any number of interfaces?	Yes	Yes	Yes	No—must have exactly one superclass or one interface
Can access instance members of enclosing class?	Yes	No	Yes (if declared in an instance method)	Yes (if declared in an instance method)
Can access local variables of enclosing method?	N/A	N/A	Yes (if `final` or effectively final)	Yes (if `final` or effectively final)

Polymorphism

- A reference with the same type as the object
- A reference that is a superclass of the object
- A reference of an interface the object implements or inherits

We can summarize this principle with the following two rules:

1. The type of the object determines which properties exist within the object in memory.
2. The type of the reference to the object determines which methods and variables are accessible to the Java program.

We summarize these concepts into a set of rules for you to memorize for the exam:

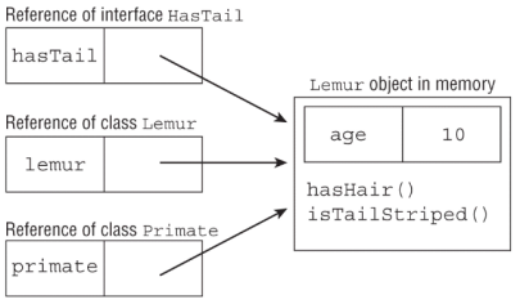
1. Casting a reference from a subtype to a supertype doesn't require an explicit cast.
2. Casting a reference from a supertype to a subtype requires an explicit cast.
3. At runtime, an invalid cast of a reference to an incompatible type results in a `ClassCastException` being thrown.
4. The compiler disallows casts to unrelated types.

Earlier, we made the statement that local variable references are allowed if they are `final` or effectively final. As an illustrative example, consider the following:

```
public void processData() {
   final int length = 5;
   int width = 10;
   int height = 2;
   class VolumeCalculator {
      public int multiply() {
         return length * width * height; // DOES NOT COMPILE
      }
   }
   width = 2;
}
```

The `length` and `height` variables are `final` and effectively final, respectively, so neither causes a compilation issue. On the other hand, the `width` variable is reassigned during the method, so it cannot be effectively final. For this reason, the local class declaration does not compile.

Polymorphism



Disallowed Casting

Let's try an example. Do you think the following program compiles?

```
1: interface Canine {}
2: interface Dog {}
3: class Wolf implements Canine {}
4:
5: public class BadCasts {
6:    public static void main(String[] args) {
7:       Wolf wolfy = new Wolf();
8:       Dog badWolf = (Dog)wolfy;
9:    } }
```

This limitation aside, the compiler can enforce one rule around interface casting. The compiler does not allow a cast from an interface reference to an object reference if the object type cannot possibly implement the interface, such as if the class is marked `final`. For example, what if we changed line 3 of our previous code?

```
3: final class Wolf implements Canine {}
```

Line 8 no longer compiles. The compiler recognizes that there are no possible subclasses of `Wolf` capable of implementing the `Dog` interface.

EXAM

1. ~~A~~B,E, D

1. Which of the following are valid record declarations? (Choose all that apply)

```
public record Iguana(int age) {
   private static final int age = 10; }

public final record Gecko() {}

public abstract record Chameleon()  {
   private static String name; }

public record BeardedDragon(boolean fun) {
   @Override public boolean fun() { return false; } }

public record Reptile(long size) {
   public Reptile {
      if(size == 1) throw new IllegalArgumentException();
   } }

public record Newt(double age) extends Reptile {
   public Newt(double age) {
      age = this.age % 2 == 0 ? 5 : 10;
   } }
```

A. Iguana
B. Gecko
C. Chameleon
D. BeardedDragon
E. Reptile
F. Newt
G. None of the above

1. B, D, E. `Iguana` does not compile, as it declares a `static` field with the same name as an instance field. Records are implicitly `final` and cannot be marked `abstract`, which is why `Gecko` compiles and `Chameleon` does not, making option B correct. Notice in `Gecko` that records are not required to declare any fields. `BeardedDragon` also compiles, as records may override any accessor methods, making option D correct. `Reptile` compiles as it contains a valid compact constructor, making option E correct. `Newt` does not compile because it cannot extend another record. It also does not compile because the compact constructor tries to read `this.age`, which is not permitted.

2. A,B,D,E
3. ~~C~~ D

3. What is the result of the following program?

```
11: public class Favorites {
```

!!ENUM construtors have implicit constructors private.

F. Next
G. None of the above

2. A,B,D,E
3. C, D

3. What is the result of the following program?

```
11: public class Favorites {
12:    enum Flavors {
13:       VANILLA, CHOCOLATE, STRAWBERRY
14:       public Flavors() {}
15:    }
16:    public static void main(String[] args) {
17:       for(final var e : Flavors.values())
18:          System.out.print((e.ordinal() % 2) + " ");
19:    } }
```

!!ENUM construtors have implicit constructors private.

Line 14 is incorrect

A. 0 1 0
B. 1 0 1
C. Exactly one line of code does not compile.
D. More than one line of code does not compile.
E. The code compiles but produces an exception at runtime.
F. None of the above.

4. B, C

4. What is the output of the following program?

```
public sealed class ArmoredAnimal permits Armadillo {
   public ArmoredAnimal(int size) {}
   @Override public String toString() { return "Strong"; }
   public static void main(String[] a) {
      var c = new Armadillo(10, null);
      System.out.println(c);
   }
}
class Armadillo extends ArmoredAnimal {
   @Override public String toString() { return "Cute"; }
   public Armadillo(int size, String name) {
      super(size);
   }
}
```

!!A class which extends seald must be : final or sealed or non-sealed.

A. Strong
B. Cute
C. The program does not compile.
D. The code compiles but produces an exception at runtime.
E. None of the above.

5. F, E

5. Which statement about the following program is correct?

```
1:  interface HasExoskeleton {
2:     double size = 2.0f;
3:     abstract int getNumberOfSections();
4:  }
5:  abstract class Insect implements HasExoskeleton {
6:     abstract int getNumberOfLegs();
7:  }
8:  public class Beetle extends Insect {
9:     int getNumberOfLegs() { return 6; }
10:    int getNumberOfSections(int count) { return 1; }
11: }
```

!! If subclasses don't implement abstract methods, the error will occur at declaration class.
(Line 8)

A. It compiles without issue.
B. The code will produce a `ClassCastException` if called at runtime.
C. The code will not compile because of line 2.
D. The code will not compile because of line 5.
E. The code will not compile because of line 8.
F. The code will not compile because of line 10.

6. D,E
7. B, E

7. What is the output of the following program?

```
1: interface Aquatic {
2:    int getNumOfGills(int p);
3: }
4: public class ClownFish implements Aquatic {
5:    String getNumOfGills() { return "14"; }
6:    int getNumOfGills(int input) { return 15; }
7:    public static void main(String[] args) {
8:       System.out.println(new ClownFish().getNumOfGills(-1));
9: } }
```

7. E. The inherited interface method `getNumOfGills(int)` is implicitly `public`; therefore, it must be declared `public` in any concrete class that implements the interface. Since the method uses the package (default) modifier in the `ClownFish` class, line 6 does not compile, making option E the correct answer. If the method declaration were corrected to include `public` on line 6, then the program would compile and print `15` at runtime, and option B would be the correct answer.

A. 14
B. 15
C. The code will not compile because of line 4.
D. The code will not compile because of line 5.
E. The code will not compile because of line 6.
F. None of the above.

8. C,D,E B, G

8. Given the following, select the statements that can be inserted into the blank line so that the code will compile and print `true` at runtime? (Choose all that apply.)

```
record Walrus(List<String> diet) {}
record Exhibit(Walrus animal, String location) {}

var e = new Exhibit(new Walrus(List.of("Wally")), "Artic");
System.out.print(e instanceof _____);
```

8. B, E, G. Options A and F do not compile because they are not compatible with `List<String>`. Option C does not compile because the reference type of `w` is `Object`, which doesn't have an `animal()` method. Option D does not compile because the variable `i` is used twice in the same pattern matching statement. Option H does not compile because you can't use `null` in a pattern matching statement. Options B, E, and G correctly compile and print `true` at runtime.

A. `Exhibit(Walrus(List<Integer> z), Object a)`
B. `Exhibit(Walrus(List m), Object n)`
C. `Object w && w.animal().diet().size() == 0`
D. `Exhibit(Walrus(var i), var i)`
E. `Exhibit(var p, var q)`
F. `Exhibit(List<?> g, var h)`
G. `Exhibit(var x, CharSequence y)`
H. `Exhibit(Walrus(null), var v)`
I. None of the above

9. A,E,F
10. G-> A,B,C,E (The grill is wrong)

10. What types can be inserted in the blanks on the lines marked `X` and `Z` that allow the code to compile? (Choose all that apply.)

```
interface Walk { private static List move() { return null; } }
interface Run extends Walk { public ArrayList move(); }
class Leopard implements Walk {
   public _____ move() {  // X
      return null;
   }
}
class Panther implements Run {
   public  move(_____) {  // Z
      return null;
   }
}
```

A. `Integer` on the line marked `X`
B. `ArrayList` on the line marked `X`
C. `List` on the line marked `X`
D. `List` on the line marked `Z`
E. `ArrayList` on the line marked `Z`
F. None of the above, since the `Run` interface does not compile.
G. Does not compile for a different reason.

11. B

12. A,F,E

E. `ArrayList` on the line marked `2`
F. None of the above, since the `Run` interface does not compile.
G. Does not compile for a different reason.

11. B

12. A,F,E 0

12. Which variables or members are accessible from within the `hiss()` method? (Choose all that apply.)

```
13: public class BoaConstrictor {
14:    private Body body;
15:    BoaConstrictor(Body b) { this.body = b; }
16:    private long tail = 10;
17:    record Body(int stripes) {
18:       private static int counter = 0;
19:       int counter() { return counter; }
20:       Body {
21:          stripes = stripes + counter++;
22:       }
23:       private void hiss() {} } }
```

A. `counter()`
B. `tail`
C. `body`
D. `stripes()`
E. `stripes`
F. `counter`
G. Line 15 does not compile.
H. Line 17 does not compile.
I. Lines 20–22 do not compile.

Two key facts

1. **Record accessors are auto-generated.**
   For `record Body(int stripes)`, the compiler generates a public method `int stripes()` automatically. That's where `stripes()` comes from— you don't see it written, but it exists.
2. **Nested records are implicitly `static`.**
   Since `Body` is a nested record, it is implicitly `static`. A `static` nested type **cannot access instance members of the enclosing class** (like `tail` and `body`) without an explicit `BoaConstrictor` instance.

13. E G

13. What is the result of the following program?

```
public class Weather {
   enum Seasons {
      WINTER, SPRING, SUMMER, FALL
   }

   public static void main(String[] args) {
      Seasons w = null;
      switch (w) {
         case Seasons.SPRING -> System.out.print("s");
         case Seasons.WINTER -> System.out.print("w");
         case Seasons.SUMMER -> System.out.print("m");
         default -> System.out.println("missing data"); }
   } }
```

It prints an NullPointerException!!

A. `s`
B. `w`
C. `m`
D. `missing data`
E. Exactly one line of code does not compile.
F. More than one line of code does not compile.
G. The code compiles but produces an exception at runtime.

14. A,C,D,E

14. Which statements about sealed classes are correct? (Choose all that apply.)
A. A sealed interface restricts which subinterfaces may extend it.
B. A sealed class cannot be indirectly extended by a class that is not listed in its `permits` clause.
C. A sealed class can be extended by an `abstract` class.
D. A sealed class can be extended by a subclass that uses the `non-sealed` modifier.
E. A sealed interface restricts which subclasses may implement it.
F. A sealed class cannot contain any nested subclasses.
G. None of the above.

14. A, C, E. A sealed interface restricts which interfaces may extend it, or which classes may implement it, making options A and E correct. Option B is incorrect. For example, a `non-sealed` subclass allows classes not listed in the `permits` clause to indirectly extend the sealed class. Option C is correct. While a sealed class is commonly extended by a subclass marked `final`, it can also be extended by a sealed or `non-sealed` subclass marked `abstract`. Option D is incorrect, as the modifier is `non-sealed`, not `unsealed`. Finally, option F is incorrect, as sealed classes can contain nested subclasses.

15. ~~B,C~~ F

15. Which line allows the code to print `Not scared` at runtime?

```
public class Ghost {
   public static void boo() {
      System.out.println("Not scared");
   }
   protected final class Spirit {
      public void boo() {
         System.out.println("Booo!!!");
      }
   }
   public static void main(String… haunt) {
      var g = new Ghost().new Spirit() {};
      ____________________________;
   } }
```

A. `g.boo()`
B. `g.super.boo()`
C. `new Ghost().boo()`
D. `g.Ghost.boo()`
E. `new Spirit().boo()`
F. None of the above

15. F. Trick question—the code does not compile! For this reason, option F is correct. The `Spirit` class is marked `final`, so it cannot be extended. The `main()` method uses an anonymous class that inherits from `Spirit`, which is not allowed. If `Spirit` were not marked `final`, then option C would be correct. Option A would print `Booo!!!`, while options B, D, and E would not compile for various reasons.

16. C E

16. The following code appears in a file named `Ostrich.java`. What is the result of compiling the source file?

```
1: public class Ostrich {
2:    private int count;
3:    static class OstrichWrangler {
4:       public int stampede() {
5:          return count;
6:       } } }
```

A. The code compiles successfully, and one bytecode file is generated: `Ostrich.class`.
B. The code compiles successfully, and two bytecode files are generated: `Ostrich.class` and `OstrichWrangler.class`.
C. The code compiles successfully, and two bytecode files are generated: `Ostrich.class` and `Ostrich$OstrichWrangler.class`.
D. A compiler error occurs on line 3.
E. A compiler error occurs on line 5.

16. E. The `OstrichWrangler` class is a `static` nested class; therefore, it cannot access the instance member `count`. For this reason, line 5 does not compile, and option E is correct.

17. E,G
18. E
19. F
20. ~~D~~ H

20. What is the output of this code?

```
13: record Gorilla(int x, Double y) {
14:    Gorilla {}
15:    Gorilla() { this(1,2.0); }
16: }
17: record Family(Gorilla parent1, Gorilla parent2) {}
18:
19: var family = new Family(
20:    new Gorilla(1, null), new Gorilla(0, 1.2));
21: System.out.print(switch (family) {
22:    case Family(var a, var b) -> "1";
23:    case Family(Gorilla c, Gorilla (int d, double e)) -> "2";
24:    case Family(Gorilla (int f, Double g), var h) -> "3";
25:    case Family(Gorilla i, Gorilla (int j, Double k)) -> "4";
26:    case Family(Object m, Object n) -> "5";
27:    case null -> "6";
28:    default -> "7";
29: });
```

A. 1
B. 2
C. 3
D. 4
E. 5
F. 6
G. 7
H. None of the above

20. H. The record declarations compile but the `switch` expression does not, making option H correct. First, the second `case` statement does not compile, as `double` is not compatible with `Double`. Next, the pattern matching `case` statement on line 22 dominates the ones on lines 23–25. If three of them were to be removed (including the second one), then the code would compile and print the value associated with the remaining one.

21. F
22. C,D

22. Which of the following can be inserted in the `rest()` method? (Choose all that apply.)

```
public class Lion {
   class Cub {}
   static class Den {}
   static void rest() {
      _____;
   } }
```

A. `Cub a = Lion.new Cub()`
B. `Lion.Cub b = new Lion().Cub()`
C. `Lion.Cub c = new Lion().new Cub()`
D. `var d = new Den()`
E. `var e = Lion.new Cub()`
F. `Lion.Den f = Lion.new Den()`
G. `Lion.Den g = new Lion.Den()`
H. `var h = new Cub()`

23. D
24. B,D,E -> Why D is not correct?
25. F

25. What does the following program print?

```
1:  public class Zebra {
2:     private int x = 24;
3:     public int hunt() {
4:        String message = "x is ";
5:        abstract class Stripes {
6:           private int x = 0;
7:           public void print() {
8:              System.out.print(message + Zebra.this.x);
9:           }
10:       }
11:       var s = new Stripes() {};
12:       s.print();
13:       return x;
14:    }
15:    public static void main(String[] args) {
16:       new Zebra().hunt();
17:    } }
```

A. `x is 0`
B. `x is 24`
C. Line 6 generates a compiler error.
D. Line 8 generates a compiler error.
E. Line 11 generates a compiler error.
F. None of the above.

25. B. `Zebra.this.x` is the correct way to refer to `x` in the `Zebra` class. Line 5 defines an abstract local class within a method. While line 11 defines a concrete anonymous class that extends the `Stripes` class. The code compiles without issue and prints `x is 24` at runtime, making option B the correct answer.

26. C
27. B,C,D,G
28. A,B,D
29. F

29. How many lines of the following program contain a compilation error?

```
1:  class Primate {
2:     protected int age = 2;
3:     { age = 1; }
4:     public Primate() {
5:        this().age = 3;
6:     }
7:  }
8:  public class Orangutan {
9:     protected int age = 4;
10:    { age = 5; }
11:    public Orangutan() {
12:       this().age = 6;
13:    }
14:    public static void main(String[] bananas) {
15:       final Primate x = (Primate)new Orangutan();
16:       System.out.println(x.age);
17:    }
18: }
```

A. None, and the program prints `1` at runtime.
B. None, and the program prints `3` at runtime.
C. None, but it causes a `ClassCastException` at runtime.
D. 1
E. 2
F. 3

30. X E, C

30. Assuming the following classes are declared as top-level types in the same file, which classes contain compiler errors? (Choose all that apply.)

```
sealed class Bird {
   public final class Flamingo extends Bird {}
}

sealed class Monkey {}

class EmperorTamarin extends Monkey {}

non-sealed class Mandrill extends Monkey {}

sealed class Friendly extends Mandrill permits Silly {}

final class Silly {}
```

A. `Bird`
B. `Monkey`
C. `EmperorTamarin`
D. `Mandrill`
E. `Friendly`
F. `Silly`
G. All of the classes compile without issue.

30. C, E. `Bird` and its nested `Flamingo` subclass compile without issue. The `permits` clause is optional if the subclass is nested or declared in the same file. For this reason, `Monkey` and its subclass `Mandrill` also compile without issue. `EmperorTamarin` does not compile, as it is missing a `non-sealed`, `sealed`, or `final` modifier, making option C correct. `Friendly` also does not compile since it lists a subclass `Silly` that does not extend it, making option E correct. While the `permits` clause is optional, the `extends` clause is not. `Silly` compiles just fine. Even though it does not extend `Friendly`, the compiler error is in the sealed class.