Select the one correct answer.

(a) The program will fail to compile.
(b) The program will compile and print `0` when run.
(c) The program will compile and print `1` when run.
(d) The program will compile and print `2` when run.
(e) The program will compile and print `3` when run.

5.31 Which of the following statements about the following program are true?

```
public interface HeavenlyBody { String describe(); }

class Star {
  String starName;
  public String describe() { return "star " + starName; }
}

class Planet extends Star {
  String name;
  public String describe() {
    return "planet " + name + " orbiting star " + starName;
  }
}
```

Select the three correct answers:

(a) The code will fail to compile.
(b) The code defines a `Planet` *is-a* `Star` relationship.
(c) The code will fail to compile if the name `starName` is replaced with the name `bodyName` throughout the declaration of the `Star` class.
(d) The code will fail to compile if the name `starName` is replaced with the name `name` throughout the declaration of the `Star` class.
(e) An instance of `Planet` is a valid instance of `HeavenlyBody`.
(f) The code defines a `Planet` *has-a* `Star` relationship.

5.13 Enum Types

An *enum type* is a special-purpose class that defines *a finite set of symbolic names and their values*. These symbolic names are usually called *enum constants* or *named constants*. An enum type is also synonymously referred to as an *enum class*.

Before the introduction of enum types in the Java programming language, such constants were typically declared as `final`, `static` variables in a class (or an interface) declaration:

```
public class MachineState {
  public static final int BUSY = 1;
  public static final int IDLE = 0;
  public static final int BLOCKED = -1;
}
```

Such constants are not type-safe, as *any* `int` value can be used where we need to use a constant declared in the `MachineState` class. Such a constant must be qualified by

the class (or interface) name, unless the class is extended (or the interface is implemented). When such a constant is printed, only its value (e.g., `0`), and not its name (e.g., `IDLE`) is printed. A constant also needs recompiling if its value is changed, as the values of such constants are compiled into the client code.

An enum type in Java is a special kind of class that is much more powerful than the approach outlined above for defining named constants.

Declaring Type-Safe Enums

The canonical form of declaring an enum type is shown below.

```
access_modifier enum enum_type_name     // Enum header
{                                       // Enum body
  EC1, EC2, ..., ECk                    // Enum constants
}
```

The following declaration is an example of an enum type:

```
public enum MachineState                // Enum header
{                                       // Enum body
  BUSY, IDLE, BLOCKED                   // Enum constants

}
```

The keyword `enum` is used to declare an enum type, as opposed to the keyword `class` for a class declaration. The basic notation requires the *enum type name* in the enum header, and *a comma-separated list of enum constants* (EC_1, *EC2, ..., ECk*) can be specified in the enum body. In the example enum declaration, the name of the enum type is `MachineState`. It defines three enum constants with explicit names: `BUSY`, `IDLE`, and `BLOCKED`. An enum constant can be any legal Java identifier, but the convention is to use uppercase letters in the name.

Essentially, an enum declaration defines a *reference type* that has a *finite number of permissible values* referenced by the enum constants, and the compiler ensures they are used in a type-safe manner.

Analogous to a top-level class, a top-level enum type can be declared with either `public` or package accessibility. However, an enum type declared as a `static` member of a reference type can be declared with any accessibility.

As we shall see later, other member declarations can be specified in the body of an enum type. If this is the case, the enum constant list must be terminated by a semicolon (;). Analogous to a class declaration, an enum type is compiled to Java bytecode that is placed in a separate class file.

The Java SE Platform API contains numerous enum types. We mention two enum types here: `java.time.Month` and `java.time.DayOfWeek`. As we would expect, the `Month` enum type represents the months from `JANUARY` to `DECEMBER`, and the `DayOfWeek` enum type represents the days of the week from `MONDAY` to `SUNDAY`. Examples of their usage can be found in §17.2, p. 1027.

Some additional examples of enum types are given below.

```
public enum MarchingOrders { LEFT, RIGHT }

public enum TrafficLightState { RED, YELLOW, GREEN; }

enum MealType { BREAKFAST, LUNCH, DINNER }
```

Using Type-Safe Enums

Example 5.24 illustrates using enum constants. An enum type is essentially used as any other reference type, and the restrictions are noted later in this section. Enum constants are actually `final`, `static` variables of the enum type, and they are implicitly initialized with instances of the enum type when the enum type is loaded at runtime. Since the enum constants are `static` members, they can be accessed using the name of the enum type—analogous to accessing `static` members in a class.

Example 5.24 shows a machine client that uses a machine whose state is an enum constant. In this example, we see that an enum constant can be passed as an argument, as shown at (1), and we can declare references whose type is an enum type, as shown at (3), but we *cannot* create new constants (i.e., objects) of the enum type `MachineState`. An attempt to do so at (5) results in a compile-time error.

The text representation of an enum constant is its name, as shown at (4). Note that it is not possible to pass a value of a type other than a `MachineState` enum constant in the call to the method `setState()` of the `Machine` class, as shown at (2).

Example 5.24 *Using Enums*

```
// File: MachineState.java
public enum MachineState { BUSY, IDLE, BLOCKED }
```

```
// File: Machine.java
public class Machine {

  private MachineState state;

  public void setState(MachineState state) { this.state = state; }
  public MachineState getState()           { return this.state; }
}
```

```
// File: MachineClient.java
public class MachineClient {
  public static void main(String[] args) {

    Machine machine = new Machine();
    machine.setState(MachineState.IDLE);           // (1) Passed as a value.
    // machine.setState(1);                        // (2) Compile error!
```

```
        MachineState state = machine.getState();        // (3) Declaring a reference.
        System.out.println(
            "Current machine state: " + state           // (4) Printing the enum name.
        );

        // MachineState newState = new MachineState();  // (5) Compile error!

        System.out.println("All machine states:");
        for (MachineState ms : MachineState.values()) { // (6) Traversing over enum
          System.out.println(ms + ":" + ms.ordinal());  //     contants.
        }

        System.out.println("Comparison:");
        MachineState state1 = MachineState.BUSY;
        MachineState state2 = state1;
        MachineState state3 = MachineState.BLOCKED;

        System.out.println(state1 + " == " + state2 + ": " +
                           (state1 == state2));                          // (7)
        System.out.println(state1 + " is equal to " + state2 + ": " +
                           (state1.equals(state2)));                     // (8)
        System.out.println(state1 + " is less than " + state3 + ": " +
                           (state1.compareTo(state3) < 0));              // (9)
      }
    }
```

Output from the program:

```
Current machine state: IDLE
All machine states:
BUSY:0
IDLE:1
BLOCKED:2
Comparison:
BUSY == BUSY: true
BUSY is equal to BUSY: true
BUSY is less than BLOCKED: true
```

Declaring Enum Constructors and Members

An enum type can declare constructors and other members as in an ordinary class, but the enum constants must be declared before any other declarations (see the declaration of the enum type `Meal` in Example 5.25). The list of enum constants must be terminated by a semicolon (;) if followed by any constructor or member declaration. Each enum constant name can be followed by an argument list that is passed to the constructor of the enum type having the matching parameter signature.

In Example 5.25, the enum type `Meal` contains a constructor declaration at (2) with the following signature:

```
Meal(int, int)
```

Each enum constant is specified with an argument list with the signature `(int, int)` that matches the non-zero argument constructor signature. The enum constant list is also terminated by a semicolon, as the enum declaration contains other members: two fields for the meal time at (3), and three instance methods to retrieve meal time information at (4).

When the enum type is loaded at runtime, the constructor is run for each enum constant, passing the argument values specified for the enum constant. For the `Meal` enum type, three objects are created that are initialized with the specified argument values, and are referenced by the three enum constant names, respectively. Note that each enum constant is a `final`, `static` reference that stores the reference value of an object of the enum type, and methods of the enum type can be called on this object by using the enum constant name. This is illustrated at (5) in Example 5.25 by calling methods on the object referenced by the enum constant `Meal.BREAKFAST`.

A default constructor is created if no constructors are provided for the enum type, analogous to a class. As mentioned earlier, an enum type cannot be instantiated using the `new` operator. The constructors cannot be called explicitly. Thus the only access modifier allowed for a constructor is `private`, as a constructor is understood to be implicitly declared `private` if no access modifier is specified.

Static initializer blocks can also be declared in an enum type, analogous to those in a class (§10.7, p. 545).

Example 5.25 *Declaring Enum Constructors and Members*

```
// File: Meal.java
public enum Meal {
  BREAKFAST(7,30), LUNCH(12,15), DINNER(19,45);             // (1)

  // Non-zero argument constructor                            (2)
  Meal(int hh, int mm) {
    this.hh = hh;
    this.mm = mm;
  }

  // Fields for the meal time:                                (3)
  private int hh;
  private int mm;

  // Instance methods:                                        (4)
  public int getHour() { return this.hh; }
  public int getMins() { return this.mm; }
  public String getTimeString() {                           // "hh:mm"
    return String.format("%02d:%02d", this.hh, this.mm);
  }
}
```

```
// File: MealAdministrator.java
public class MealAdministrator {
  public static void main(String[] args) {

    System.out.printf(                                        // (5)
        "Please note that no eggs will be served at %s, %s.%n",
        Meal.BREAKFAST, Meal.BREAKFAST.getTimeString()
    );

    System.out.println("Meal times are as follows:");
    Meal[] meals = Meal.values();                             // (6)
    for (Meal meal : meals) {                                 // (7)
      System.out.printf("%s served at %s%n", meal, meal.getTimeString());
    }

    Meal formalDinner = Meal.valueOf("DINNER");               // (8)
    System.out.printf("Formal dress is required for %s at %s.%n",
        formalDinner, formalDinner.getTimeString()
    );
  }
}
```

Output from the program:

```
Please note that no eggs will be served at BREAKFAST, 07:30.
Meal times are as follows:
BREAKFAST served at 07:30
LUNCH served at 12:15
DINNER served at 19:45
Formal dress is required for DINNER at 19:45.
```

Implicit Static Methods for Enum Types

All enum types implicitly have the following `static` methods, and methods with these names *cannot* be declared in an enum type declaration:

`static` *`EnumTypeName`*`[] values()`

Returns an array containing the enum constants of this enum type, *in the order they are specified*.

`static` *`EnumTypeName`* `valueOf(String name)`

Returns the enum constant with the specified `name`. An `IllegalArgumentException` is thrown if the specified `name` does not match the name of an enum constant. The specified `name` is *not* qualified with the enum type name.

The `static` method `values()` is called at (6) in Example 5.25 to create an array of enum constants. This array is traversed in the `for(:)` loop at (7), printing the information about each meal.

The static method valueOf() is called at (8) in Example 5.25 to retrieve the enum constant that has the specified name "DINNER". A print statement is used to print the information about the meal denoted by this enum constant.

Inherited Methods from the java.lang.Enum Class

All enum types are subtypes of the java.lang.Enum class which implements the default behavior. All enum types are comparable (§14.4, p. 761) and serializable (§20.5, p. 1261).

All enum types inherit the following selected final methods from the java.lang.Enum class, and these methods therefore *cannot* be overridden by an enum type:

```
final int compareTo(E o)
```

The *natural order* of the enum constants in an enum type is according to their *ordinal values* (see the ordinal() method below). The compareTo() method in the Comparable interface is discussed in §14.4, p. 761.

```
final boolean equals(Object other)
```

Returns true if the specified object is equal to this enum constant (§14.2, p. 744).

```
final int hashCode()
```

Returns a hash code for this enum constant (§14.3, p. 753).

```
final String name()
```

Returns the name of this enum constant, exactly as it is declared in its enum declaration.

```
final int ordinal()
```

Returns the *ordinal value* of this enum constant (i.e., its position in its enum type declaration). The first enum constant is assigned an ordinal value of zero. If the ordinal value of an enum constant is less than the ordinal value of another enum constant of the same enum type, the former occurs before the latter in the enum type declaration.

Note that the equality test implemented by the equals() method is based on reference equality (==) of the enum constants, not on object value equality. Comparing two enum references for equality meanss determining whether they store the reference value of the same enum constant—that is, whether the references are aliases. Thus for any two enum references, meal1 and meal2, the expressions meal1.equals(meal2) and meal1 == meal2 are equivalent.

The java.lang.Enum class also overrides the toString() method from the Object class. The toString() method returns the name of the enum constant, but it is *not* final and can be overridden by an enum type—but that is rarely done. Examples in this subsection illustrate the use of these methods.

Extending Enum Types: Constant-Specific Class Bodies

Constant-specific class bodies define anonymous classes (§9.6, p. 521) inside an enum type—they implicitly extend the enclosing enum type creating new subtypes. The enum type `Meal` in Example 5.26 declares constant-specific class bodies for its constants. The following skeletal code declares the constant-specific class body for the enum constant `BREAKFAST`:

```
BREAKFAST(7,30) {                    // (1) Start of constant-specific class body
  @Override
  public double mealPrice(Day day) {  // (2) Overriding abstract method
    ...
  }
  @Override
  public String toString() {          // (3) Overriding method from the Enum class
    ...
  }
}                                     // (4) End of constant-specific class body
```

The constant-specific class body, as the name implies, is a class body that is specific to a particular enum constant. As for any class body, it is enclosed in curly brackets, `{ }`. It is declared immediately after the enum constant and any constructor arguments. In the code above, it starts at (1) and ends at (4). Like any class body, it can contain member declarations. In the above case, the body contains two method declarations: an implementation of the method `mealPrice()` at (2) that overrides the `abstract` method declaration at (7) in the enclosing enum supertype `Meal`, and an implementation of the `toString()` method at (3) that overrides the one inherited by the `Meal` enum type from the superclass `java.lang.Enum`. The `@Override` annotation used on these overriding methods ensures that the compiler will issue an error message if such a method declaration does not satisfy the criteria for overriding.

The constant-specific class body is an anonymous class—that is, a class with no name. Each constant-specific class body defines a distinct, albeit anonymous, subtype of the enclosing enum type. In the code above, the constant-specific class body defines a subtype of the `Meal` enum type. It inherits members of the enclosing enum supertype that are not private, overridden, or hidden. When the enum type `Meal` is loaded at runtime, this constant-specific class body is instantiated, and the reference value of the instance is assigned to the enum constant `BREAKFAST`. Note that the type of the enum constant is `Meal`, which is the supertype of the anonymous subtype represented by the constant-specific class body. Since supertype references can refer to subtype objects, this assignment is legal.

Each enum constant overrides the `abstract` method `mealPrice()` declared in the enclosing enum supertype—that is, provides an implementation for the method. The compiler will report an error if this is not the case. Although the enum type declaration specifies an `abstract` method, the enum type declaration is *not* declared `abstract`—contrary to an `abstract` class. Given that the references `meal` and `day` are of the enum types `Meal` and `Day` from Example 5.26, respectively, the method call

```
meal.mealPrice(day)
```

will execute the `mealPrice()` method from the constant-specific body of the enum constant denoted by the reference `meal`.

Two constant-specific class bodies, associated with the enum constants `BREAKFAST` and `LUNCH`, override the `toString()` method from the `java.lang.Enum` class. Note that the `toString()` method is not overridden in the `Meal` enum type, but in the anonymous classes represented by two constant-specific class bodies. The third enum constant, `DINNER`, relies on the `toString()` method inherited from the `java.lang.Enum` class.

Constructors, `abstract` methods, and `static` methods cannot be declared in a constant-specific class body. Instance methods declared in constant-specific class bodies are only accessible if they override methods in the enclosing enum supertype.

Example 5.26 *Declaring Constant-Specific Class Bodies*

```
// File: Day.java
public enum Day {
  MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}
```

```
// File: Meal.java
public enum Meal {
  // Each enum constant defines a constant-specific class body
  BREAKFAST(7,30) {                                                    // (1)
    @Override
    public double mealPrice(Day day) {                                 // (2)
      double breakfastPrice = 10.50;
      if (day.equals(Day.SATURDAY) || day == Day.SUNDAY)
        breakfastPrice *= 1.5;
      return breakfastPrice;
    }
    @Override
    public String toString() {                                         // (3)
      return "Breakfast";
    }
  },                                                                   // (4)
  LUNCH(12,15) {
    @Override
    public double mealPrice(Day day) {                                 // (5)
      double lunchPrice = 20.50;
      switch (day) {
        case SATURDAY: case SUNDAY:
          lunchPrice *= 2.0;
      }
      return lunchPrice;
    }
    @Override
    public String toString() {
      return "Lunch";
    }
  },
```

```
  DINNER(19,45) {
    @Override
    public double mealPrice(Day day) {                                  // (6)
      double dinnerPrice = 25.50;
      if (day.compareTo(Day.SATURDAY) >= 0 && day.compareTo(Day.SUNDAY) <= 0)
        dinnerPrice *= 2.5;
      return dinnerPrice;
    }
  };

  // Abstract method implemented in constant-specific class bodies.
  abstract double mealPrice(Day day);                                   // (7)

  // Enum constructor:
  Meal(int hh, int mm) {
    this.hh = hh;
    this.mm = mm;
  }

  // Instance fields: Time for the meal.
  private int hh;
  private int mm;

  // Instance methods:
  public int getHour() { return this.hh; }
  public int getMins() { return this.mm; }
  public String getTimeString() {                               // "hh:mm"
    return String.format("%02d:%02d", this.hh, this.mm);
  }
}

// File: MealPrices.java
public class MealPrices {

  public static void main(String[] args) {                              // (8)
    System.out.printf(
        "Please note that %s, %s, on %s costs $%.2f.%n",
        Meal.BREAKFAST.name(),                                          // (9)
        Meal.BREAKFAST.getTimeString(),
        Day.MONDAY,
        Meal.BREAKFAST.mealPrice(Day.MONDAY)                            // (10)
    );

    System.out.println("Meal prices on " + Day.SATURDAY + " are as follows:");
    Meal[] meals = Meal.values();
    for (Meal meal : meals) {
      System.out.printf(
          "%s costs $%.2f.%n", meal, meal.mealPrice(Day.SATURDAY)       // (11)
      );
    }
  }
}
```

Output from the program:

```
Please note that BREAKFAST, 07:30, on MONDAY costs $10.50.
```

```
Meal prices on SATURDAY are as follows:
Breakfast costs $15.75.
Lunch costs $41.00.
DINNER costs $63.75.
```

In Example 5.26, the `mealPrice()` method declaration at (2) uses both the `equals()` method and the `==` operator to compare enum constants for equality. The `mealPrice()` method declaration at (5) uses enum constants in a `switch` statement. Note that the `case` labels in the `switch` statement are enum constant names, without the enum type name. The `mealPrice()` method declaration at (6) uses the `compareTo()` method to compare enum constants.

The `main()` method at (8) in Example 5.26 demonstrates calling the `mealPrice()` method in the constant-specific class bodies. The `mealPrice()` method is called at (10) and (11). Example 5.26 also illustrates the difference between the `name()` and the `toString()` methods of the enum types. The `name()` method is called at (9), and the `toString()` method is called implicitly at (11) on `Meal` enum values. The `name()` method always prints the enum constant name exactly as it was declared. Which `toString()` method is executed depends on whether the `toString()` method from the `java.lang.Enum` class is overridden. Only the constant-specific class bodies of the enum constants `BREAKFAST` and `LUNCH` override this method. The output from the program confirms this to be the case.

Enum Values in Exhaustive `switch` Expressions

A `switch` expression is always *exhaustive*—that is, the cases defined in the `switch` expression must cover *all* values of the selector expression type or the code will not compile (§4.2, p. 160).

The `switch` expression below at (1) is exhaustive, as all values of the enum type `Day` are covered by the two switch rules at (2) and (3). Deleting any one or both of these switch rules will result in a compile-time error.

```
Day day = Day.MONDAY;             // See Example 5.26, p. 295, for enum type Day.
String typeOfDay = switch (day) {                                          // (1)
  case MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY -> "Weekday";          // (2)
  case SATURDAY, SUNDAY                             -> "Weekend";          // (3)
};
```

The `switch` expression below at (4) is also exhaustive, as all values of the enum type `Day` are collectively covered by the `case` label at (5) and the `default` label at (6). Deleting the `case` label at (5) does *not* result in a compile-time error, as the `default` label will then cover all values of the selector expression—but it may result in a wrong value being returned by the `switch` expression. However, deleting the `default` label at (6) will result in a compile-time error, as the `case` label at (5) only covers a subset of the values for the `Day` enum type. Compile-time checking of exhaustiveness of a `switch` expression whose switch rules are defined by the arrow notation results in robust and secure code.

```
typeOfDay = switch (day) {                                          // (4)
  case SATURDAY, SUNDAY -> "Weekend";                               // (5)
  default -> "Weekday";                                             // (6)
}
```

Declaring Type-Safe Enums, Revisited

We have seen declarations of enum types as top-level types, but they can also be nested as `static` member and `static` local types (§9.1, p. 491). Although nested enum types are implicitly `static`, they can be declared with the keyword `static`. The following skeletal code shows the two enum types `Day` and `Meal` declared as `static` member types in the class `MealPrices`:

```
public class MealPrices {
  public enum Day { /* ... */ }                          // Static member

  public static enum Meal { /* ... */ }                  // Static member

  public static void main(String[] args) { /* ... */ }   // Static method
}
```

An enum type cannot be explicitly extended using the `extends` clause. An enum type is implicitly `final`, unless it contains constant-specific class bodies. If it declares constant-specific class bodies, it is implicitly extended. No matter what, it cannot be explicitly declared `final`.

An enum class is either *implicitly* `final` if its declaration contains no enum constants that have a class body, or *implicitly* `sealed` if its declaration contains at least one enum constant that has a class body. Since an enum type can be either implicitly `final` or implicitly `sealed`, it can implement a sealed interface—in which case, it must be specified in the `permits` clause of the sealed interface (p. 311).

An enum type *cannot* be declared `abstract`, regardless of whether each `abstract` method is overridden in the constant-specific class body of every enum constant.

Like a class, an enum can implement interfaces.

```
public interface ITimeInfo {
  public int getHour();
  public int getMins();
}

public enum Meal implements ITimeInfo {
  // ...
  @Override public int getHour() { return this.hh; }
  @Override public int getMins() { return this.mm; }
  // ...
}
```

The Java Collections Framework provides a special-purpose *set* implementation (`java.util.EnumSet`) and a special-purpose *map* implementation (`java.util.EnumMap`) for use with enum types. These implementations provide better performance for enum types than their general-purpose counterparts, and are worth checking out.