Object-Oriented Programming 5

Chapter Topics

- Implementing inheritance and its implications in object-oriented programming
- Implications of the subtype–supertype relationship
- Differentiating between inheritance (*is-a*) and aggregation (*has-a*)
- Distinguishing between static and dynamic types of a reference
- Overriding and hiding superclass members
- Understanding method overriding and comparing it with method overloading
- Using the `super` reference to access superclass members
- Using the `this()` and `super()` calls for constructor chaining
- Declaring `abstract` classes and methods
- Using the `final` modifier to declare classes, methods, fields, and local variables
- Declaring interfaces, and the implications of multiple interface inheritance
- Understanding the implications of subtyping for reference arrays
- Understanding conversions when assigning, casting, and passing references
- Understanding resolution of overloaded methods
- Identifying the type of objects and pattern matching using the `instanceof` operator
- Taking advantage of polymorphism and how dynamic method lookup works
- Declaring and using enum types
- Declaring and using record classes
- Declaring and using sealed classes and interfaces

Java SE 17 Developer Exam Objectives

[3.2] Create classes and records, and define and use instance and static fields and methods, constructors, and instance and static initializers	§5.14, p. 299
❍ *Record classes are covered in this chapter.*	
❍ *For creating normal classes and defining class members and constructors, see* Chapter 3, p. 97.	
❍ *For instance and static initializers, see* Chapter 10.5, p. 540.	
[3.3] Implement overloading, including var-arg methods	§5.1, p. 202
❍ *Comparing overloading and overriding is covered in this chapter.*	

- *For overloading methods, including varargs methods, see* **Chapter 3, p. 97**.

[3.5] Implement inheritance, including abstract and sealed classes. Override methods, including that of Object class. Implement polymorphism and differentiate object type versus reference type. Perform type casting, identify object types using instanceof operator and pattern matching - *Inheritance, abstract and sealed classes, overriding methods, polymorphism and type casting, static and dynamic types of a reference, and* `instanceof` *type comparison and pattern match operators are covered in this chapter.* - *For overriding methods of the* `Object` *class, see* **§14.1, p. 743**.	**§5.1, p. 191** *to* **§5.4, p. 218** **§5.11, p. 269** **§5.12, p. 278** **§5.15, p. 311**
[3.6] Create and use interfaces, identify functional interfaces, and utilize private, static, and default interface methods - *Only nonfunctional interfaces are covered in this chapter.* - *For functional interfaces, see* **§13.1, p. 675**.	**§5.6, p. 237**
[3.7] Create and use enumerations with fields, methods and constructors	**§5.13, p. 287**

Java SE 11 Developer Exam Objectives

[3.5] Create and use subclasses and superclasses, including abstract classes	**§5.1, p. 191** *to* **§5.4, p. 218**
[3.6] Utilize polymorphism and casting to call methods, differentiating object type versus reference type	**§5.11, p. 269** **§5.12, p. 278**
[3.7] Create and use interfaces, identify functional interfaces, and utilize private, static, and default methods	**§5.6, p. 237**

- *Only nonfunctional interfaces are covered in this chapter.*
- *For functional interfaces, see* §13.1, p. 675.

[3.8] Create and use enumerations — §5.13, p. 287

This chapter delves into two of the main pillars of object-oriented programming (OOP): inheritance and polymorphism. The discussion of the other two OOP concepts, abstraction and encapsulation, can be found in §1.2, p. 5, and §6.1, p. 324, respectively.

Many Java features relating to OOP are covered in this chapter: subtype–supertype relationship, abstract classes and methods, final declarations, interfaces, reference conversions, enum types, record classes, and sealed classes and interfaces. There is a lot of ground to cover in this chapter, so take a deep breath and dive in.

5.1 Implementing Inheritance

Inheritance is one of the fundamental mechanisms for code reuse in OOP. It allows new classes to be derived from existing ones. The new class (also called a *subclass*, *subtype*, *derived class*, or *child class*) can inherit members from the old class (also called a *superclass*, *supertype*, *base class*, or *parent class*). The subclass can add new behavior and properties and, under certain circumstances, modify its inherited behavior.

A subclass specifies the name of its superclass in the subclass header using the `extends` clause.

Click here to view code image

```
class TubeLight extends Light { ... }   // TubeLight is a subclass of Light.
```

The subclass specifies only the additional new and modified members in its class body. The rest of its declaration is made up of its inherited members. If no `extends` clause is specified in the header of a class declaration, the class implicitly inherits from the `java.lang.Object` class. Selected methods inherited by all objects from the `Object` class are covered in connection with object comparison (Chapter 14, p. 741) and thread synchronization (§22.4, p. 1396).

Inheritance of members is closely tied to their declared *accessibility*. If a superclass member is accessible by its simple name in the subclass (without the use of any extra syntax, like `super`), that member is considered inherited. Conversely, private, overrid-

den, and hidden members of the superclass are *not* inherited. Inheritance should not be confused with the *existence* of such members in the state of a subclass object.

The declaration of the `Light` class at (1) in **Example 5.1** implicitly extends the `java.lang.Object` class, as no explicit `extends` clause is specified. Also in **Example 5.1**, the subclass `TubeLight` at (2) explicitly uses the `extends` clause and specifies only members other than those that it already inherits from the superclass `Light` (which, in turn, inherits from the `Object` class). Members of the superclass `Light`, which are accessible by their simple names in the subclass `TubeLight`, are inherited by the subclass, as evident from the output in **Example 5.1**.

Members of the superclass that have `private` access are not inherited by the subclass and can only be accessed indirectly. The `private` field `indicator` of the superclass `Light` is not inherited, but exists in the subclass object and is indirectly accessible through `public` methods.

Using appropriate access modifiers, the superclass can limit which members can be accessed directly, and therefore, inherited by its subclasses. As shown in **Example 5.1**, the subclass can use the inherited members as if they were declared in its own class body. This is not the case for members that are declared as `private` in the superclass, as shown at (3), (4), and (5). Members that have package accessibility in the superclass are also not inherited by subclasses in other packages, as these members are accessible by their simple names only in subclasses within the same package as the superclass.

Since constructors are *not* members of a class, they are *not* inherited by a subclass.

Example 5.1 *Extending Classes: Inheritance and Accessibility*

Click here to view code image

```
// File: Utility.java
class Light {                              // (1)
  // Instance fields:
            int     noOfWatts;           // Wattage
  private   boolean indicator;           // On or off
  protected String  location;            // Placement

  // Static field:
  private static int counter;            // Number of Light objects created

  // Non-zero argument constructor:
  Light(int noOfWatts, boolean indicator, String location) {
    this.noOfWatts = noOfWatts;
    this.indicator = indicator;
    this.location  = location;
```

```
    ++counter;                          // Increment counter.
  }

  // Instance methods:
  public  void    switchOn()  { indicator = true; }
  public  void    switchOff() { indicator = false; }
  public  boolean isOn()      { return indicator; }
  private String  getLocation() { return location; }

  // Static methods:
  public static int getCount() { return counter; }
}
//_____________________________________________________________________________
class TubeLight extends Light {         // (2) Subclass uses the extends clause.
  // Instance fields:
  private int tubeLength;               // Length in millimeters
  private int tubeDiameter;             // Diameter in millimeters

  // Non-zero argument constructor
  TubeLight(int noOfWatts, boolean indicator, String location,
            int tubeLength, int tubeDiameter) {
    super(noOfWatts, indicator, location);  // Calling constructor in superclass.
    this.tubeLength = tubeLength;
    this.tubeDiameter = tubeDiameter;
  }
  // Instance methods:
  public int getTubeLength() { return tubeLength; }

  public void printInfo() {
    System.out.println("From the subclass:");
    System.out.println("Tube length (mm): "  + getTubeLength());
    System.out.println("Tube diameter (mm): " + tubeDiameter);
    System.out.println();
    System.out.println("From the superclass:");
    System.out.println("Wattage: "      + noOfWatts);     // Inherited.
//  System.out.println("Indicator: "    + indicator);     // (3) Not inherited.
    System.out.println("Location: "     + location);      // Inherited.
//  System.out.println("Counter: "   + counter);          // (4) Not inherited.
    switchOn();                                           // Inherited

    switchOff();                                          // Inherited
    System.out.println("Indicator: "    + isOn());        // Inherited.
//  System.out.println("Location: " + getLocation());     // (5) Not inherited.
    System.out.println("Number of lights: " + getCount());// Inherited.

  }
}
//_____________________________________________________________________________
public class Utility {
```

```
    public static void main(String[] args) {
      TubeLight loftLight = new TubeLight(18, true, "Loft", 590, 26);
      loftLight.printInfo();
    }
  }
```

Output from the program:

Click here to view code image

```
From the subclass:
Tube length (mm): 590
Tube diameter (mm): 26

From the superclass:
Wattage: 18
Location: Loft
Indicator: false
Number of lights: 1
```

In Java, a class can extend only *one* class; that is, it can have only one direct superclass. This kind of inheritance is sometimes called *single* or *linear implementation inheritance*. The name is appropriate, as the subclass inherits the *implementation* of its superclass. Java only allows *single inheritance of implementation* using the `extends` clause. *Multiple inheritance of implementation* occurs when a class inherits multiple implementations from the interfaces it implements, but this is *not* allowed in Java (p. 240).

The inheritance relationship can be depicted as an *inheritance hierarchy* (also called a *class hierarchy*), where each subclass is connected by the inheritance arrow to its direct superclass. The `java.lang.Object` class is always at the top (the *root*) of any Java inheritance hierarchy, as all classes extend (either directly or indirectly) this class. The path from a subclass in the inheritance hierarchy to the root traces all the superclasses that the subclass inherits from. Classes up in the hierarchy are more *generalized* (often called *broader*), as they abstract the class behavior. Classes lower in the hierarchy are more *specialized* (often called *narrower*), as they customize the inherited behavior by additional properties and behavior. **Figure 5.1** illustrates the inheritance relationship between the class `Light`, which represents the more general abstraction, and its more specialized subclasses. The class `SpotLightBulb` inherits from the classes `LightBulb`, `Light`, and `Object`.

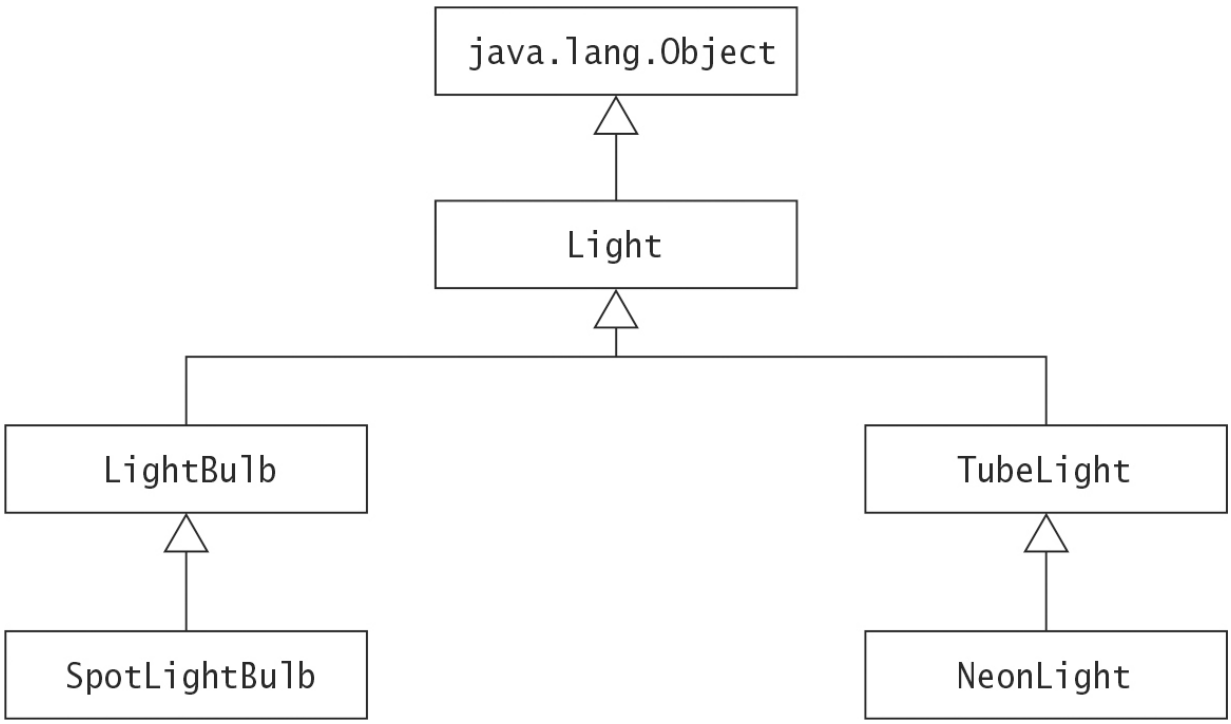

Figure 5.1 *Inheritance Hierarchy*

Relationships: is-a and has-a

The inheritance relationship between a subclass and its superclass is embodied by the *is-a* relationship. Since a subclass inherits from its superclass, a subclass object *is-a* superclass object, and can be used wherever an object of the superclass can be used. It has particular consequences for how objects can be used. An object of the `TubeLight` class *is-an* object of the superclass `Light`. Referring to **Figure 5.1**, an object of the `TubeLight` class can be used wherever an object of the superclass `Light` can be used.

The inheritance relationship is *transitive*: If class `B` extends class `A` and class `C` extends class `B`, then class `C` will also inherit from class `A` via class `B`. In **Figure 5.1**, an object of the `SpotLightBulb` class *is-an* object of the class `Light`. The *is-a* relationship does not hold between peer classes: An object of the `LightBulb` class is *not* an object of the class `TubeLight`, and vice versa.

Whereas inheritance defines the relationship *is-a* between a superclass and its subclasses, *aggregation* defines the relationship *has-a* (also called the *whole–part* relationship) between an instance of a class and its constituents (also called *parts)*. Aggregation comprises the *usage* of objects.

A class declaration with instance field references implements an aggregate object. The class `Light` is declared with three instance fields: an instance field to store its wattage (`noOfWatts`), an instance field to store whether it is on or off (`indicator`), and an instance field reference (`location`) to a `String` object to store its location (in fact, its only constituent object). An instance of class `Light` *has* (or *uses*) an object of class `String`. In Java, a composite object cannot contain other objects. It can only store *reference values* of its constituent objects in its fields. This relationship defines an *aggrega-*

tion hierarchy (also called *object hierarchy*) that embodies the *has-a* relationship. Constituent objects can be shared between objects. If their lifetimes are dependent on the lifetime of the aggregate object, then this relationship is called *composition*, and implies strong ownership of the parts by the composite object.

Choosing between inheritance and aggregation to model relationships can be a crucial design decision. A good design strategy advocates that inheritance should be used only if the relationship *is-a* is unequivocally maintained throughout the lifetime of the objects involved; otherwise, aggregation is the best choice. A *role* is often confused with an *is-a* relationship. For example, given the class `Employee`, it would not be a good idea to model the roles that an employee can play (such as manager or cashier) by inheritance, if these roles change intermittently. Changing roles would involve a new object to represent the new role every time this happens.

Code reuse is also best achieved by aggregation when there is no *is-a* relationship. Enforcing an artificial *is-a* relationship that is not naturally present is usually not a good idea. Methods that contradict the abstraction represented by the subclass can be invoked. Using aggregation in such a case results in a better solution.

Both inheritance and aggregation promote encapsulation of *implementation*. Changes in the implementation of constituent objects generally have minimal impact on the clients of the composite object, since these clients do not directly deal with the underlying objects. However, changing the *contract* of a superclass can have consequences for the subclasses (called the *ripple effect*) as well as for clients that are dependent on a particular behavior of the subclasses. For this reason, aggregation provides stronger encapsulation than inheritance.

The Subtype–Supertype Relationship

A class defines a *reference type*, a data type whose objects can be accessed only by references. Therefore, the inheritance hierarchy can be regarded as a *type hierarchy*, embodying the *subtype–supertype relationship* between reference types. The *subclass– superclass* relationship is a special case of the subtype–supertype relationship that is between classes. The subclass–superclass relationship allows *single inheritance of type*, meaning that the subclass inherits the *type* of its direct superclass. This is in contrast to a class that implements several interfaces, resulting in *multiple inheritance of type*—that is, a class inherits the *type* of all interfaces it implements.

In the context of Java, the subtype–supertype relationship implies that the reference value of a subtype object can be assigned to a supertype reference because a subtype object can be substituted for a supertype object. This assignment involves a *widening reference conversion*, as references are assigned *up* the inheritance hierarchy. Using the

reference types in **Example 5.1**, the following code assigns the reference value of an object of the subtype `TubeLight` to the reference `light` of the supertype `Light`:

Click here to view code image

```
Light light = new TubeLight(36, false, "Basement",
                            1200, 26);      // (1) widening reference conversion
```

An implicit widening conversion takes place under assignment, as the reference value of a narrower type (subtype `TubeLight`) object is being assigned to a reference of a broader type (supertype `Light`). We can now use the reference `light` to invoke those methods on the subtype object that are inherited from the supertype `Light`:

Click here to view code image

```
light.switchOn();                           // (2)
```

Note that the compiler only knows about the *static type* (also called the *declared type*) of the reference `light`, which is `Light`, and ensures that only methods from this type can be called using the reference `light`. However, at runtime, the reference `light` will refer to an object of the subtype `TubeLight` when the call to the method `switchOn()` is executed. It is the *type of the object* that the reference refers to at run-time that determines which method is executed. The subtype object inherits the `switchOn()` method from its supertype `Light`, so this method is executed. The type of the object that the reference refers to at runtime is called the *dynamic type* (a.k.a. the *runtime type* or the *object type*) of the reference.

One might be tempted to invoke methods exclusive to the `TubeLight` subtype via the supertype reference `light`:

Click here to view code image

```
light.getTubeLength();                      // (3) Compile-time error!
```

This code will not work, as the compiler does not know which object the reference `light` will denote at runtime; it only knows the declared type of the reference. As the declaration of the class `Light` does not have a method called `getTubeLength()`, this method call at (3) results in a compile-time error. Eliciting subtype-specific behavior using a supertype reference requires a *narrowing reference conversion* with an explicit cast (**p. 278**).

Overriding Instance Methods

Under certain circumstances, a subclass can *override instance methods* from its superclass. Overriding such a method allows the subclass to provide its *own* implementation of the method. The overridden method in the superclass is *not* inherited by the subclass. When the method is invoked on an object of the subclass, it is the method implementation in the subclass that is executed. The new method in the subclass must abide by the following rules of method overriding:

- The new method definition in the subclass must have the same *method signature*. In other words, the method name, and the types and the number of parameters, including their order, must be the same as in the overridden method of the superclass. Whether parameters in the overriding method should be `final` is at the discretion of the subclass. A method's signature does not comprise the `final` modifier of parameters, only their types and order.
- The return type of the overriding method can be a *subtype* of the return type of the overridden method (called *covariant return*, **p. 201**).
- The new method definition cannot *narrow* the accessibility of the method, but it can *widen* it.
- The new method definition can throw either all or none, or a subset of the *checked* exceptions (including their subclasses) that are specified in the `throws` clause of the overridden method in the superclass (§7.5, **p. 388**)—that is, the overriding method may not throw any checked exception that is *wider* than the checked exceptions thrown by the overridden method. However, the overriding method may throw any *unchecked* exceptions.

The criteria for overriding methods also apply to interfaces, where a subinterface can override `abstract` and `default` method declarations from its superinterfaces (**p. 237**).

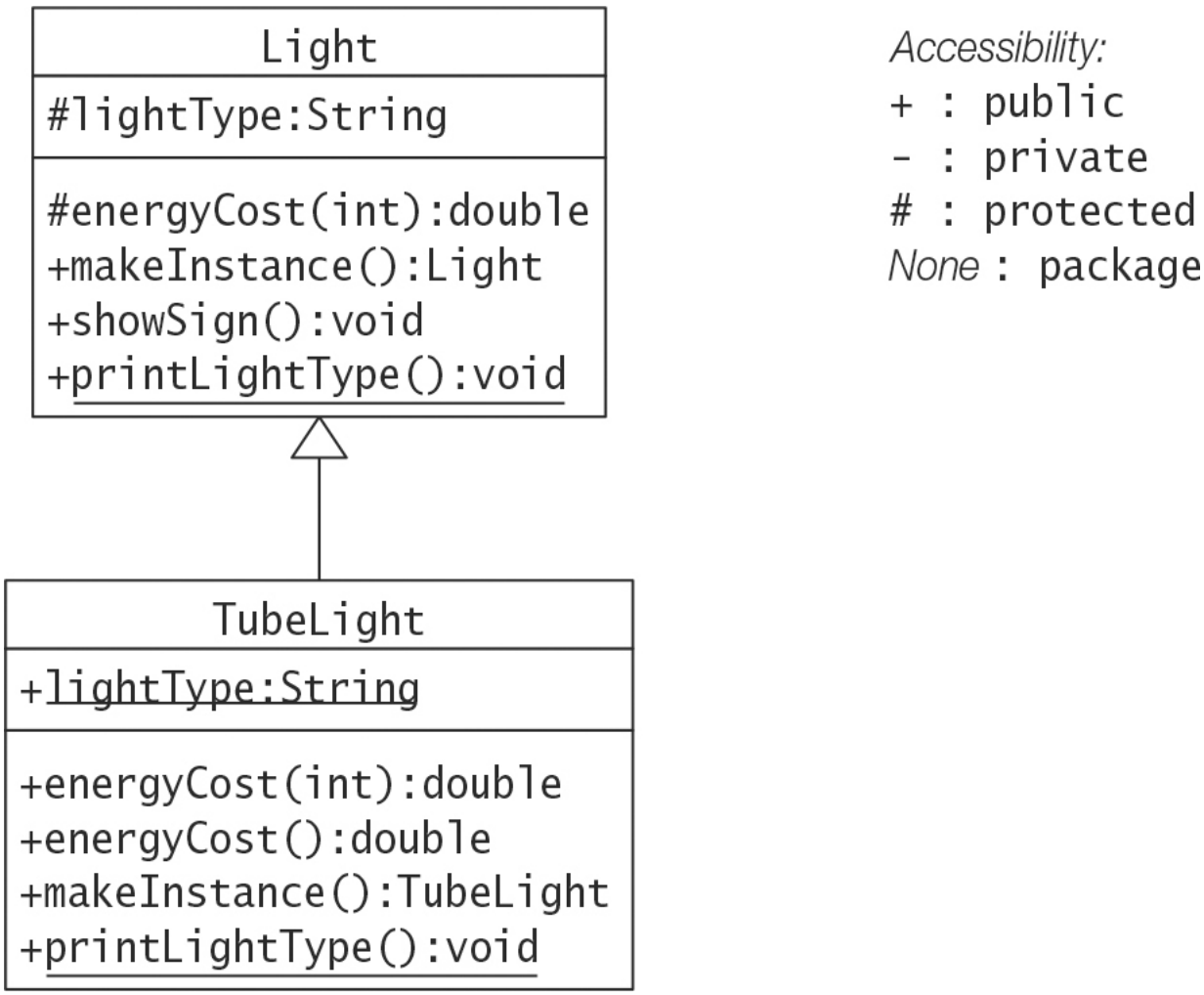

Figure 5.2 *Inheritance Hierarchy for* Example 5.2

The canonical examples of method overriding in Java are the `equals()`, `hashCode()`, and `toString()` methods of the `Object` class. The wrapper classes and the `String` class all override these methods (**Chapter 8**, **p. 423**). Overriding these methods in user-defined classes is thoroughly covered in connection with object comparison (**Chapter 14**, **p. 741**).

Example 5.2 illustrates overriding, overloading, and hiding of members in a class. **Figure 5.2** gives an overview of the two main classes in **Example 5.2**. The new definition of the `energyCost()` method at (7) in the subclass `TubeLight` has the same signature and the same return type as the method at (2) in the superclass `Light`. The new definition specifies a subset of the exceptions (`ZeroHoursException`) thrown by the overridden method (the exception class `InvalidHoursException` is a superclass of `NegativeHoursException` and `ZeroHoursException`). The new definition also widens the accessibility (`public`) from what it was in the overridden definition (`protected`). The overriding method declares the parameter to be `final`, but this has no bearing on overriding the method.

Click here to view code image

```
// The overridden method in the superclass Light:
protected double energyCost(int noOfHours)      // (2) Instance method
```

```
    throws InvalidHoursException { ... }

// The overriding method in the subclass TubeLight:
@Override
public double energyCost(final int noOfHours)   // (7) Overriding instance
    throws ZeroHoursException { ... }           //     method at (2).
```

The astute reader will have noticed the `@Override` annotation preceding the method definition at (7). The compiler will now report an error if the method definition at (7) does *not* override an inherited method. The annotation helps to ensure that the method definition overrides the inherited method, rather than overloading another method silently.

Invocation of the method `energyCost()` on an object of subclass `TubeLight` using references of the subclass and the superclass at (15) and (16), respectively, results in the new definition at (7) being executed, since both references are aliases of the `TubeLight` object created at (12).

Click here to view code image

```
tubeLight.energyCost(50);                   // (15) Invokes method at (7).
light1.energyCost(50);                      // (16) Invokes method at (7).
```

Not surprisingly, the invocation of the method `energyCost()` on an object of superclass `Light`, using a reference of the superclass at (17), results in the overridden definition at (2) being executed:

Click here to view code image

```
light2.energyCost(50);                      // (17) Invokes method at (2).
```

Example 5.2 *Overriding, Overloading, and Hiding*

Click here to view code image

```
// File: Client2.java
// Exceptions
class InvalidHoursException extends Exception {}
class NegativeHoursException extends InvalidHoursException {}
class ZeroHoursException extends InvalidHoursException {}

class Light {

  protected String lightType = "Generic Light";   // (1) Instance field
```

```
  protected double energyCost(int noOfHours)      // (2) Instance method
      throws InvalidHoursException {
    System.out.print(">> Light.energyCost(int): ");
    if (noOfHours < 0)
      throw new NegativeHoursException();
    double cost = 00.20 * noOfHours;
    System.out.println("Energy cost for " + lightType + ": " + cost);
    return cost;
  }

  public Light makeInstance() {                   // (3) Instance method
    System.out.print(">> Light.makeInstance(): ");
    return new Light();
  }

  public void showSign() {                        // (4) Instance method
    System.out.print(">> Light.showSign(): ");
    System.out.println("Let there be light!");
  }

  public static void printLightType() {           // (5) Static method
    System.out.print(">> Static Light.printLightType(): ");
    System.out.println("Generic Light");
  }
}
//_____________________________________________________________________________
class TubeLight extends Light {

  public static String lightType = "Tube Light";  // (6) Hiding field at (1).

  @Override
  public double energyCost(final int noOfHours)   // (7) Overriding instance
      throws ZeroHoursException {                 //     method at (2).
    System.out.print(">> TubeLight.energyCost(int): ");
    if (noOfHours == 0)
      throw new ZeroHoursException();
    double cost = 00.10 * noOfHours;
    System.out.println("Energy cost for " + lightType + ": " + cost);
    return cost;
  }

  public double energyCost() {          // (8) Overloading method at (7).
    System.out.print(">> TubeLight.energyCost(): ");
    double flatrate = 20.00;
    System.out.println("Energy cost for " + lightType + ": " + flatrate);
    return flatrate;
  }
```

```
  @Override
  public TubeLight makeInstance() {     // (9) Overriding instance method at (3).
    System.out.print(">> TubeLight.makeInstance(): ");
    return new TubeLight();
  }

  public static void printLightType() { // (10) Hiding static method at (5).
    System.out.print(">> Static TubeLight.printLightType(): ");
    System.out.println(lightType);
  }
}
//_____________________________________________________________________________

public class Client2 {
  public static void main(String[] args)          // (11)
      throws InvalidHoursException {

    TubeLight tubeLight = new TubeLight();        // (12)
    Light     light1    = tubeLight;              // (13) Aliases.
    Light     light2    = new Light();            // (14)

    System.out.println("Invoke overridden instance method:");
    tubeLight.energyCost(50);                     // (15) Invokes method at (7).
    light1.energyCost(50);                        // (16) Invokes method at (7).
    light2.energyCost(50);                        // (17) Invokes method at (2).

    System.out.println(
        "\nInvoke overridden instance method with covariant return:");
    System.out.println(
        light2.makeInstance().getClass());        // (18) Invokes method at (3).
    System.out.println(
        tubeLight.makeInstance().getClass());     // (19) Invokes method at (9).

    System.out.println("\nAccess hidden field:");
    System.out.println(tubeLight.lightType);      // (20) Accesses field at (6).
    System.out.println(light1.lightType);         // (21) Accesses field at (1).
    System.out.println(light2.lightType);         // (22) Accesses field at (1).

    System.out.println("\nInvoke hidden static method:");
    tubeLight.printLightType();                   // (23) Invokes method at (10).
    light1.printLightType();                      // (24) Invokes method at (5).
    light2.printLightType();                      // (25) Invokes method at (5).

    System.out.println("\nInvoke overloaded method:");
    tubeLight.energyCost();                       // (26) Invokes method at (8).
  }
}
```

Output from the program:

Click here to view code image

```
Invoke overridden instance method:
>> TubeLight.energyCost(int): Energy cost for Tube Light: 5.0
>> TubeLight.energyCost(int): Energy cost for Tube Light: 5.0
>> Light.energyCost(int): Energy cost for Generic Light: 10.0

Invoke overridden instance method with covariant return:
>> Light.makeInstance(): class Light
>> TubeLight.makeInstance(): class TubeLight

Access hidden field:
Tube Light
Generic Light
Generic Light

Invoke hidden static method:
>> Static TubeLight.printLightType(): Tube Light
>> Static Light.printLightType(): Generic Light
>> Static Light.printLightType(): Generic Light

Invoke overloaded method:
>> TubeLight.energyCost(): Energy cost for Tube Light: 20.0
```

Here are a few more facts to note about overriding.

- A subclass must use the keyword `super` to invoke an overridden method in the superclass.
- A `final` method cannot be overridden because the modifier `final` prevents method overriding. An attempt to override a `final` method will result in a compile-time error. An `abstract` method, in contrast, requires the non-`abstract` subclasses to override the method so as to provide an implementation. Abstract and final methods are discussed in §5.4, p. 224, and §5.5, p. 226, respectively.
- The access modifier `private` for a method means that the method is not accessible outside the class in which it is defined; therefore, a subclass cannot override it. However, a subclass can give its own definition of such a method, which may have the same signature as the method in its superclass.
- A subclass within the same package as the superclass can override any non-`final` methods declared in the superclass. However, a subclass in a different package can override only the non-`final` methods that are declared as either `public` or `protected` in the superclass.
- An instance method in a subclass cannot override a `static` method in the superclass. The compiler will flag such an attempt as an error. A `static` method is class

specific and not part of any object, while overriding methods are invoked on behalf of objects of the subclass. However, a `static` method in a subclass can *hide* a `static` method in the superclass, as we shall see (p. 203), but it cannot hide an instance method in the superclass.

- Constructors, since they are not inherited, cannot be overridden.

Covariant `return` in Overriding Methods

In Example 5.2, the definition of the method `makeInstance()` at (9) overrides the method definition at (3).

Click here to view code image

```
// The overridden method in the superclass Light:
public Light makeInstance() { ... }     // (3) Instance method

// The overriding method in the subclass TubeLight:
@Override
public TubeLight makeInstance() { ... } // (9) Overriding instance method at (3).
```

Note that the method signatures are the same, but the return type at (9) is a subtype of the return type at (3). The method at (9) returns an object of the subtype `Tube-Light`, whereas the method at (3) returns an object of the supertype `Light`. This is an example of *covariant return*.

Depending on whether we call the method `makeInstance()` on an object of the subtype `TubeLight` or an object of the supertype `Light`, the respective method definition will be executed. The code at (18) and (19) illustrates which object is returned by the method, depending on which method definition is executed.

Note that covariant return applies only to *reference* types, not to primitive types. For example, changing the return type of the `energyCost()` method at (7) to `float` will result in a compile-time error. There is no subtype–supertype relationship between primitive types.

Overriding versus Overloading

Method overriding should not be confused with *method overloading* (§3.6, p. 108). Method overriding always requires the same method signature (name and parameter types) and the same or covariant return types. Overloading occurs when the method names are the same, but the parameter lists differ. Therefore, to overload methods, the parameters must differ in either type, order, or number. As the return type is not a part of the method signature, use of different return types is not sufficient to overload

methods. Although methods can be overloaded, as shown at (1) and (2) in **Example 5.3**, a call to such an overloaded method can be *ambiguous*, as shown at (3). It is not possible to determine which method should be called, (1) or (2). Disambiguation can be done by making the types of the arguments in the call more specific, as shown at (4) and (5).

Example 5.3 *Ambiguous Call to Overloaded Methods*

Click here to view code image

```
public class Overloader {

  static final void callMe(long x, Integer y) {       // (1) Overloaded by (2)
    System.out.println("long, Integer");
  }
  static void callMe(Integer x, long y) {       // (2) Overloaded by (1)
    System.out.println("Integer, long");
  }

  public static void main(String[] args) {
//  callMe(20, 17);                // (3) Ambiguous call: Box 1st or 2nd argument?
    callMe(20, Integer.valueOf(17));   // (4) Calls (1): long, Integer
    callMe(Integer.valueOf(20), 17);   // (5) Calls (2): Integer, long
  }
}
```

Output from the program:

```
long, Integer
Integer, long
```

Only non-`final` instance methods in the superclass that are directly accessible from the subclass using their simple name can be overridden. In contrast, both non-`private` instance and `static` methods can be overloaded in the class they are defined in or are in a subclass of the class they are defined in.

Depending on the arguments passed in a call to an overloaded method, the appropriate method implementation is invoked. In **Example 5.2**, the method `energy-Cost()` at (2) in superclass `Light` is overridden in subclass `TubeLight` at (7) and overloaded at (8). When invoked at (26), the overloaded declaration at (8) is executed.

For overloaded and `static` methods, which method implementation will be executed at runtime is determined at *compile time*. In contrast, for overridden methods, the method implementation to be executed is determined at *runtime* (**p. 278**).

Hiding Fields

A subclass cannot override inherited *fields* of the superclass, but it can *hide* them. The subclass can define fields with the same name as in the superclass. If this is the case, the fields in the superclass cannot be accessed in the subclass by their simple names; therefore, they are not inherited by the subclass. A hidden `static` field can always be accessed by using the superclass name in the subclass declaration. Additionally, the keyword `super` can be used in non-`static` code in the subclass declaration to access hidden `static` fields.

The following distinction between invoking instance methods on an object and accessing fields of an object must be noted. When an instance method is invoked on an object using a reference, it is the *dynamic type* of the reference (i.e., *the type of the current object* denoted by the reference at runtime), not the declared type of the reference, that determines which method implementation will be executed. In **Example 5.2** at (15), (16), and (17), this is evident from invoking the overridden method `energyCost()`: The method from the class corresponding to the current object is executed, regardless of the declared reference type.

When a field of an object is accessed using a reference, it is the *declared type* of the reference, not the type of the current object denoted by the reference, that determines which field will actually be accessed. In **Example 5.2** at (20), (21), and (22), this is evident from accessing the hidden field `lightType`: The field accessed is the one declared in the class corresponding to the declared reference type, regardless of the object denoted by the reference at runtime.

In contrast to method overriding, where an instance method cannot override a `static` method, there are no such restrictions on the hiding of fields. The field `lightType` is `static` in the subclass, but not in the superclass. The declared type of the fields need not be the same either—only the field name matters in the hiding of fields.

Hiding Static Methods

Only instance methods in an object can be overridden. However, a `static` method in a subclass can *hide* a `static` method from the superclass. Hiding a `static` method is analogous to overriding an instance method except for one important aspect: Calls to `static` methods are *bound at compile time* as opposed to runtime for calls to overridden instance methods, and therefore do *not* exhibit polymorphic behavior exhibited by calls to overridden instance methods (**p. 278**).

When hiding a `static` method, the compiler will flag an error if the signatures are the same, but the other requirements regarding the return type, `throws` clause, and acces-

sibility are not met. If the signatures are different, the method name is overloaded, not hidden.

A `static` method in the subclass can only hide a `static` method in the superclass. Analogous to an overridden instance method, a hidden superclass `static` method is *not* inherited. Analogous to accessing hidden `static` fields, a hidden `static` method in the superclass can always be invoked by using the superclass name or by using the keyword `super` in non-`static` code in the subclass declaration.

Example 5.2 illustrates invocation of `static` methods using references. Analogous to accessing fields, the `static` method invoked at (23), (24), and (25) is determined by the *declared type* of the reference. At (23), the declared reference type is `TubeLight`; therefore, the `static` method `printLightType()` at (10) in this class is invoked. At (24) and (25), the declared reference type is `Light`, and the hidden `static` method `printLightType()` at (5) in that class is invoked. This is borne out by the output from the program.

In summary, a subclass can do any of the following:

- Access inherited members using their simple names
- Override a non-`final` instance method with the same signature as in the superclass
- Overload a `static` or instance method with the same name as in the superclass
- Hide a `static` non-`final` method with the same signature as in the superclass
- Hide a `static` or instance field with the same name as in the superclass
- Declare new members that are not in the superclass
- Declare constructors that can invoke constructors in the superclass implicitly, or use the `super()` construct with the appropriate arguments to invoke them explicitly (p. 209)

Table 5.1 provides a comparison between overriding, hiding, and overloading of methods.

Table 5.1 *Overriding, Hiding, and Overloading of Methods*

Comparison criteria	Overriding of instance methods/Hiding of `static` methods	Overloading of instance and `static` methods
Method name	Must be the same.	Must be the same.
Argument list	Must be the same.	Must be different.

Return type	Can be the same type or a covariant type.	Can be different.
`throws` clause	Can be restrictive about checked exceptions thrown. Must not throw new checked exceptions, but can include subclasses of exceptions thrown.	Can be different.
Accessibility	Can make it less restrictive, but not more restrictive.	Can be different.
`final` modifier	A `final` instance/`static` method cannot be overridden/hidden. Compile-time error if `final` instance/`static` method is defined in a subclass.	Can be overloaded in the same class or in a subclass.
`private` modifier	A `private` instance/`static` method cannot be overridden/hidden. No compile-time error if `private` instance/`static` method is defined in a subclass.	Can be overloaded only in the same class.
Declaration context	An instance/`static` method can only be overridden/hidden in a subclass.	An instance or `static` method can be overloaded in the same class or in a subclass.
Binding of method call/polymorphic behavior	Calls to overridden instance methods are bound at runtime, and therefore exhibit polymorphic behavior. Calls to hidden `static` methods are bound at compile time and do *not* exhibit polymorphic behavior.	Calls are bound at compile time and do *not* exhibit polymorphic behavior.

5.2 The Object Reference `super`

The `this` reference can be used in non-`static` code to refer to the current object (§3.5, p. 106). The keyword `super`, in contrast, can be used in non-`static` code to access fields and invoke methods from the superclass. The keyword `super` provides a reference to the current object as an instance of its superclass. In method invocations with `super`, the method from the superclass is invoked regardless of what the actual type of the current object is or whether the current class overrides the method. This approach is typically used to invoke methods that are overridden and to access members that are hidden to the subclass. Unlike the `this` keyword, the `super` keyword cannot be used as an ordinary reference. For example, it cannot be assigned to other references or cast to other reference types.

Example 5.4 uses the superclass `Light` and its subclass `TubeLight` from **Example 5.2**, which are also shown in **Figure 5.2**. In **Example 5.4**, the class `NeonLight` extends the class `TubeLight`. The declaration of the method `demonstrate()` at (11) in the class `NeonLight` makes use of the `super` keyword to access members higher in its inheritance hierarchy. This is the case when the `showSign()` method is invoked at (12). This method is defined at (4) in the class `Light`, rather than in the direct superclass `TubeLight` of the subclass `NeonLight`. The overridden method `energyCost()` at (7) and its overloaded version at (8) in the class `TubeLight` are invoked, using the object reference `super` at (13) and (14), respectively.

The superclass `Light` has a field named `lightType` and a method named `energyCost` defined at (1) and (2), respectively. One might be tempted to use the syntax `super.super.energyCost(20)` in the subclass `NeonLight` to invoke this method, but this is not a valid construct. One might also be tempted to cast the `this` reference to the class `Light` and try again, as shown at (15). The output shows that the method `energyCost()` at (7) in the class `TubeLight` was executed, not the one from the class `Light`. The reason is that a cast simply changes the type of the reference (in this case to `Light`), not the class of the object (which is still `NeonLight`). Method invocation is determined by the class of the current object, resulting in the inherited method `energyCost()` in the class `TubeLight` being executed. There is no way to invoke the method `energyCost()` in the class `Light` from the subclass `NeonLight`, without declaring a reference of the type `Light`.

At (16), the keyword `super` is used to access the field `lightType` at (6) in the class `TubeLight`, but the keyword `super` is redundant in this case. At (17), the field `lightType` from the class `Light` is accessed successfully by casting the `this` reference, because it is the type of the reference that determines which field is accessed. From non-`static` code in a subclass, it is possible to directly access fields in a class higher in the inheritance hierarchy by casting the `this` reference. However, it is futile to cast the

`this` reference to invoke instance methods in a class higher in the inheritance hierarchy, as illustrated earlier for the overridden method `energyCost()`.

Finally, the calls to the `static` methods at (18) and (19) using the `super` and `this` references, respectively, exhibit runtime behavior analogous to accessing fields, as discussed previously.

Example 5.4 *Using the* `super` *Keyword*

Click here to view code image

```
// File: Client3.java
//Exceptions
class InvalidHoursException extends Exception {}
class NegativeHoursException extends InvalidHoursException {}
class ZeroHoursException extends InvalidHoursException {}

class Light {

  protected String lightType = "Generic Light";   // (1) Instance field

  protected double energyCost(int noOfHours)      // (2) Instance method
      throws InvalidHoursException {
    System.out.print(">> Light.energyCost(int): ");
    if (noOfHours < 0)
      throw new NegativeHoursException();
    double cost = 00.20 * noOfHours;
    System.out.println("Energy cost for " + lightType + ": " + cost);
    return cost;
  }

  public Light makeInstance() {                   // (3) Instance method
    System.out.print(">> Light.makeInstance(): ");
    return new Light();
  }

  public void showSign() {                        // (4) Instance method
    System.out.print(">> Light.showSign(): ");
    System.out.println("Let there be light!");
  }

  public static void printLightType() {           // (5) Static method
    System.out.print(">> Static Light.printLightType(): ");
    System.out.println("Generic Light");
  }
}
//_____________________________________________________________________
```

```
class TubeLight extends Light {

  public static String lightType = "Tube Light";  // (6) Hiding field at (1).

  @Override
  public double energyCost(final int noOfHours)   // (7) Overriding instance
      throws ZeroHoursException {                 //     method at (2).
    System.out.print(">> TubeLight.energyCost(int): ");
    if (noOfHours == 0)
      throw new ZeroHoursException();
    double cost = 00.10 * noOfHours;
    System.out.println("Energy cost for " + lightType + ": " + cost);
    return cost;
  }

  public double energyCost() {          // (8) Overloading method at (7).
    System.out.print(">> TubeLight.energyCost(): ");

    double flatrate = 20.00;
    System.out.println("Energy cost for " + lightType + ": " + flatrate);
    return flatrate;
  }

  @Override
  public TubeLight makeInstance() {     // (9) Overriding instance method at (3).
    System.out.print(">> TubeLight.makeInstance(): ");
    return new TubeLight();
  }

  public static void printLightType() { // (10) Hiding static method at (5).
    System.out.print(">> Static TubeLight.printLightType(): ");
    System.out.println(lightType);
  }
}
//_____________________________________________________________________________
class NeonLight extends TubeLight {
  // ...
  public void demonstrate()                         // (11)
      throws InvalidHoursException {
    super.showSign();                               // (12) Invokes method at (4)
    super.energyCost(50);                           // (13) Invokes method at (7)
    super.energyCost();                             // (14) Invokes method at (8)

    ((Light) this).energyCost(50);                  // (15) Invokes method at (7)

    System.out.println(super.lightType);            // (16) Accesses field at (6)
    System.out.println(((Light) this).lightType); // (17) Accesses field at (1)

    super.printLightType();                         // (18) Invokes method at (10)
```

```
    ((Light) this).printLightType();              // (19) Invokes method at (5)
  }
}
//_____________________________________________________________________________
public class Client3 {
  public static void main(String[] args)
      throws InvalidHoursException {
    NeonLight neonRef = new NeonLight();
    neonRef.demonstrate();
  }
}
```

Output from the program:

Click here to view code image

```
>> Light.showSign(): Let there be light!
>> TubeLight.energyCost(int): Energy cost for Tube Light: 5.0
>> TubeLight.energyCost(): Energy cost for Tube Light: 20.0
>> TubeLight.energyCost(int): Energy cost for Tube Light: 5.0
Tube Light
Generic Light
>> Static TubeLight.printLightType(): Tube Light
>> Static Light.printLightType(): Generic Light
```

5.3 Chaining Constructors Using `this()` and `super()`

A basic understanding of constructors (§3.7, p. 109) is beneficent for the discussion in this section.

The `this()` Constructor Call

Constructors cannot be inherited or overridden. They can be overloaded, but only in the same class. Since a constructor always has the same name as the class, each parameter list must be different when defining more than one constructor for a class. In Example 5.5, the class `Light` has three overloaded constructors. In the constructor at (3), the `this` reference is used to access the fields shadowed by the parameters. In the `main()` method at (4), the appropriate constructor is invoked depending on the arguments in the constructor call, as illustrated by the program output.

Example 5.5 *Constructor Overloading*

Click here to view code image

Click here to view code image

```
// File: DemoConstructorCall.java
class Light {
  // Fields:
  private int     noOfWatts;      // wattage
  private boolean indicator;      // on or off
  private String  location;       // placement

  // Constructors:
  Light() {                                   // (1) No-argument constructor
    noOfWatts = 0;
    indicator = false;
    location  = "X";
    System.out.println("Returning from no-argument constructor no. 1.");
  }
  Light(int watts, boolean onOffState) {                      // (2)
    noOfWatts = watts;
    indicator = onOffState;
    location  = "X";
    System.out.println("Returning from constructor no. 2.");
  }
  Light(int noOfWatts, boolean indicator, String location) {  // (3)
    this.noOfWatts = noOfWatts;
    this.indicator = indicator;
    this.location  = location;
    System.out.println("Returning from constructor no. 3.");
  }
}
//_____________________________________________________________________________
public class DemoConstructorCall {
  public static void main(String[] args) {                    // (4)
    System.out.println("Creating Light object no. 1.");
    Light light1 = new Light();
    System.out.println("Creating Light object no. 2.");
    Light light2 = new Light(250, true);
    System.out.println("Creating Light object no. 3.");
    Light light3 = new Light(250, true, "attic");
  }
}
```

Output from the program:

Click here to view code image

```
Creating Light object no. 1.
Returning from no-argument constructor no. 1.
Creating Light object no. 2.
Returning from constructor no. 2.
```

```
Creating Light object no. 3.
Returning from constructor no. 3.
```

Example 5.6 illustrates the use of the `this()` construct, which is used to implement *local chaining* of constructors in the class when an instance of the class is created. The first two constructors at (1) and (2) from **Example 5.5** have been rewritten using the `this()` construct in **Example 5.6** at (1) and (2), respectively. The `this()` construct can be regarded as being locally overloaded, since its parameters (and hence its signature) can vary, as shown in the body of the constructors at (1) and (2). The `this()` call invokes the local constructor with the corresponding parameter list. In the `main()` method at (4), the appropriate constructor is invoked depending on the arguments in the constructor call when each of the three `Light` objects are created. Calling the no-argument constructor at (1) to create a `Light` object results in the constructors at (2) and (3) being executed as well. This is confirmed by the output from the program. In this case, the output shows that the constructor at (3) completed first, followed by the constructor at (2), and finally by the no-argument constructor at (1) that was called first. Bearing in mind the definition of the constructors, the constructors are invoked in the *reverse* order; that is, invocation of the no-argument constructor immediately leads to invocation of the constructor at (2) by the call `this(0, false)`, and its invocation leads to the constructor at (3) being called immediately by the call `this(watt, ind, "X")`, with the completion of the execution in the reverse order of their invocation. Similarly, calling the constructor at (2) to create an instance of the `Light` class results in the constructor at (3) being executed as well.

If the `this()` call is specified, it must occur as the *first* statement in a constructor. The `this()` call can be followed by any other relevant code. This restriction is due to Java's handling of constructor invocation in the superclass when an object of the subclass is created. This mechanism is explained in the next subsection.

Example 5.6 *The* `this()` *Constructor Call*

Click here to view code image

```
// File: DemoThisCall.java
class Light {
  // Fields:
  private int     noOfWatts;
  private boolean indicator;
  private String  location;

  // Constructors:
  Light() {                                  // (1) No-argument constructor
    this(0, false);
    System.out.println("Returning from no-argument constructor no. 1.");
```

```
  }
  Light(int watt, boolean ind) {                              // (2)
    this(watt, ind, "X");
    System.out.println("Returning from constructor no. 2.");
  }
  Light(int noOfWatts, boolean indicator, String location) { // (3)
    this.noOfWatts = noOfWatts;
    this.indicator = indicator;
    this.location  = location;
    System.out.println("Returning from constructor no. 3.");
  }
}
//_____________________________________________________________________________
public class DemoThisCall {
  public static void main(String[] args) {                    // (4)
    System.out.println("Creating Light object no. 1.");
    Light light1 = new Light();                               // (5)
    System.out.println("Creating Light object no. 2.");
    Light light2 = new Light(250, true);                      // (6)
    System.out.println("Creating Light object no. 3.");
    Light light3 = new Light(250, true, "attic");             // (7)
  }
}
```

Output from the program:

Click here to view code image

```
Creating Light object no. 1.
Returning from constructor no. 3.
Returning from constructor no. 2.
Returning from no-argument constructor no. 1.
Creating Light object no. 2.
Returning from constructor no. 3.
Returning from constructor no. 2.
Creating Light object no. 3.
Returning from constructor no. 3.
```

The `super()` Constructor Call

The constructor call `super()` is used in a subclass constructor to invoke a constructor in the *direct* superclass. This allows the subclass to influence the initialization of its inherited state when an object of the subclass is created. A `super()` call in the constructor of a subclass will result in the execution of the relevant constructor from the superclass, based on the signature of the call. Since the superclass name is known in the sub-

class declaration, the compiler can determine the superclass constructor invoked from the signature of the parameter list.

A constructor in a subclass can access the class's inherited members by their simple names. The keyword `super` can also be used in a subclass constructor to access inherited members via its superclass. One might be tempted to use the `super` keyword in a constructor to specify initial values for inherited fields. However, the `super()` construct provides a better solution to initialize the inherited state.

In **Example 5.7**, the constructor at (3) of the class `Light` has a `super()` call (with no arguments) at (4). Although the constructor is not strictly necessary, as the compiler will insert one—as explained later—it is included here for expositional purposes. The constructor at (6) of the class `TubeLight` has a `super()` call (with three arguments) at (7). This `super()` call will match the constructor at (3) of the superclass `Light`. This is evident from the program output.

Example 5.7 *The* `super()` *Constructor Call*

Click here to view code image

```
// File: ChainingConstructors.java
class Light {
  // Fields:
  private int     noOfWatts;
  private boolean indicator;
  private String  location;

  // Constructors:
  Light() {                                    // (1) No-argument constructor
    this(0, false);
    System.out.println(
    "Returning from no-argument constructor no. 1 in class Light");
  }
  Light(int watt, boolean ind) {                               // (2)
    this(watt, ind, "X");
    System.out.println(
    "Returning from constructor no. 2 in class Light");
  }
  Light(int noOfWatts, boolean indicator, String location) {  // (3)
    super();                                                   // (4)
    this.noOfWatts = noOfWatts;
    this.indicator = indicator;
    this.location  = location;
    System.out.println(
        "Returning from constructor no. 3 in class Light");
  }
```

```
  }
  //_____________________________________________________________________
  class TubeLight extends Light {
    // Instance variables:
    private int tubeLength;
    private int colorNo;

    // Constructors:
    TubeLight(int tubeLength, int colorNo) {                    // (5)
      this(tubeLength, colorNo, 100, true, "Unknown");
      System.out.println(
             "Returning from constructor no. 1 in class TubeLight");
    }
    TubeLight(int tubeLength, int colorNo, int noOfWatts,
              boolean indicator, String location) {             // (6)
      super(noOfWatts, indicator, location);                    // (7)
      this.tubeLength = tubeLength;
      this.colorNo    = colorNo;
      System.out.println(
             "Returning from constructor no. 2 in class TubeLight");
    }
  }
  //_____________________________________________________________________
  public class ChainingConstructors {
    public static void main(String[] args) {
      System.out.println("Creating a TubeLight object.");
      TubeLight tubeLightRef = new TubeLight(20, 5);            // (8)
    }
  }
```

Output from the program:

Click here to view code image

```
Creating a TubeLight object.
Returning from constructor no. 3 in class Light
Returning from constructor no. 2 in class TubeLight
Returning from constructor no. 1 in class TubeLight
```

The `super()` construct has the same restrictions as the `this()` construct: If used, the `super()` call must occur as the *first* statement in a constructor, and it can only be used in a constructor declaration. This implies that `this()` and `super()` calls cannot both occur in the same constructor. The `this()` construct is used to *chain* constructors in the *same* class. The constructor at the end of such a chain can invoke a superclass constructor using the `super()` construct. Just as the `this()` construct leads to chaining of constructors in the same class, so the `super()` construct leads to chaining of subclass

constructors to superclass constructors. This chaining behavior guarantees that all superclass constructors are called, starting with the constructor of the class being instantiated, all the way to the top of the inheritance hierarchy, which is always the `Object` class. Note that the body of the constructor is executed in the reverse order to the call order, as the `super()` call can occur only as the first statement in a constructor. This order of execution ensures that the constructor from the `Object` class is completed first, followed by the constructors in the other classes down to the class being instantiated in the inheritance hierarchy. This is called (subclass–superclass) *constructor chaining*. The output from **Example 5.7** clearly illustrates this chain of events when an object of the class `TubeLight` is created.

If a constructor at the end of a `this()` chain (which may not be a chain at all if no `this()` call is invoked) does not have an explicit call to `super()`, the call `super()` (without the parameters) is implicitly inserted by the compiler to invoke the no-argument constructor of the superclass. In other words, if a constructor has neither a `this()` call nor a `super()` call as its first statement, the compiler inserts a `super()` call to the no-argument constructor in the superclass. The code

Click here to view code image

```
class A {
  A() {}              // No-argument constructor.
  // ...
}

class B extends A {   // No constructors.
  // ...
}
```

is equivalent to

Click here to view code image

```
class A {
  A() { super(); }    // (1) Call to no-argument superclass constructor inserted.
  // ...
}

class B extends A {
  B() { super(); }    // (2) Default constructor inserted.
  // ...
}
```

where the compiler inserts a `super()` call in the no-argument constructor for class `A` at (1) and inserts the default constructor for class `B` at (2). The `super()` call at (2) will result in a call to the no-argument constructor in `A` at (1), and the `super()` call at (1) will result in a call to the no-argument constructor in the superclass of `A`—that is, the `Object` class.

If a superclass defines just non-zero argument constructors (i.e., only constructors with parameters), its subclasses cannot rely on the implicit `super()` call being inserted. This will be flagged as a compile-time error. The subclasses must then explicitly call a superclass constructor, using the `super()` construct with the right arguments.

Click here to view code image

```
class NeonLight extends TubeLight {
  // Field
  private String sign;

  NeonLight() {                                // (1)
    super(10, 2, 100, true, "Roof-top");       // (2) Cannot be commented out.
    sign = "All will be revealed!";
  }
  // ...
}
```

The preceding declaration of the subclass `NeonLight` provides a no-argument constructor at (1). The call of the constructor at (2) in the superclass `TubeLight` cannot be omitted. If it is omitted, any insertion of a `super()` call (with no arguments) in this constructor will try to match a no-argument constructor in the superclass `Tube-Light`, which provides only non-zero argument constructors. The class `NeonLight` will not compile unless an explicit valid `super()` call is inserted at (2).

If the superclass provides just non-zero argument constructors (i.e., it does not have a no-argument constructor), this has implications for its subclasses. A subclass that relies on its default constructor will fail to compile because the default constructor of the subclass will attempt to call the (nonexistent) no-argument constructor in the superclass. A constructor in a subclass must explicitly use the `super()` call, with the appropriate arguments, to invoke a non-zero argument constructor in the superclass. This call is necessary because the constructor in the subclass cannot rely on an implicit `super()` call to the no-argument constructor in the superclass.

Review Questions

5.1 Given the following class declarations:

Click here to view code image

```
// Classes
class Foo {
  private int i;
  public void f() { /* ... */ }
  public void g() { /* ... */ }
}

class Bar extends Foo {
  public int j;
  public void g() { /* ... */ }
}

public class Main {
  public static void main(String[] args) {
    Foo a = new Bar();
    Bar b = new Bar()
    // (1) INSERT STATEMENT HERE
  }
}
```

Which of the following statements can be inserted at (1) without causing a compile-time error?

Select the two correct answers.

a. `b.f();`

b. `a.j = 5;`

c. `a.g();`

d. `b.i = 3;`

5.2 Given the following code:

Click here to view code image

```
class A {
  void doIt() {}
```

```
}

class B extends A {
  void doIt() {}
}

class C extends B {
  void doIt() {}
  void callUp() {
    // (1) INSERT EXPRESSION HERE
  }
}
```

insert the expression that would call the `doIt()` method in `A`.

Select the one correct answer.

a. `doIt();`

b. `super.doIt();`

c. `super.super.doIt();`

d. `this.super.doIt();`

e. `A.this.doIt();`

f. `((A) this).doIt();`

g. It is not possible.

5.3 What would be the result of compiling and running the following program?

Click here to view code image

```
class A {
  int max(int x, int y) { (x>y) ? x : y; }
}

class B extends A {
  int max(int x, int y) { return super.max(y, x) - 10; }
}

class C extends B {
  int max(int x, int y) { return super.max(x+10, y+10); }
}
```

```
public class UserClass {
  public static void main(String[] args) {
    B b = new C();
    System.out.println(b.max(13, 29));
  }
}
```

Select the one correct answer.

a. The code will fail to compile.

b. The code will compile, but it will throw an exception at runtime.

c. The code will compile and print `13` at runtime.

d. The code will compile and print `23` at runtime.

e. The code will compile and print `29` at runtime.

f. The code will compile and print `39` at runtime.

5.4 What would be the result of compiling and running the following program?

Click here to view code image

```
class Vehicle {
  static public String getModelName() { return "Volvo"; }
  public long getRegNo() { return 12345; }
}

class Car extends Vehicle {
  static public String getModelName() { return "Toyota"; }
  public long getRegNo() { return 54321; }
}

public class TakeARide {
  public static void main(String[] args) {
    Car c = new Car();
    Vehicle v = c;

    System.out.println("|" + v.getModelName() + "|" + c.getModelName() +
                       "|" + v.getRegNo()     + "|" + c.getRegNo() + "|");
  }
}
```

Select the one correct answer.

a. The code will fail to compile.

b. The code will compile and print `|Toyota|Volvo|12345|54321|` at runtime.

c. The code will compile and print `|Volvo|Toyota|12345|54321|` at runtime.

d. The code will compile and print `|Toyota|Toyota|12345|12345|` at runtime.

e. The code will compile and print `|Volvo|Volvo|12345|54321|` at runtime.

f. The code will compile and print `|Toyota|Toyota|12345|12345|` at runtime.

g. The code will compile and print `|Volvo|Toyota|54321|54321|` at runtime.

5.5 Which constructors can be inserted at (1) in `MySub` without causing a compile-time error?

Click here to view code image

```
class MySuper {
  int number;
  MySuper(int i) { number = i; }
}

class MySub extends MySuper {
  int count;
  MySub(int count, int num) {
    super(num);
    this.count = count;
  }

  // (1) INSERT CONSTRUCTOR HERE
}
```

Select the one correct answer.

a. `MySub() {}`

b. `MySub(int count) { this.count = count; }`

c. `MySub(int count) { super(); this.count = count; }`

d. `MySub(int count) { this.count = count; super(count); }`

e. `MySub(int count) { this(count, count); }`

f. `MySub(int count) { super(count); this(count, 0); }`

5.6 Which of the following statements is true?

Select the one correct answer.

a. A `super()` or `this()` call must always be provided explicitly as the first statement in the body of a constructor.

b. If both a subclass and its superclass do not have any declared constructors, the implicit default constructor of the subclass will call `super()` when run.

c. If neither `super()` nor `this()` is specified as the first statement in the body of a constructor, `this()` will implicitly be inserted as the first statement.

d. If `super()` is the first statement in the body of a constructor, `this()` can be declared as the second statement.

e. Calling `super()` as the first statement in the body of a constructor of a subclass will always work, since all superclasses have a default constructor.

5.7 What will the following program print when run?

Click here to view code image

```
public class MyClass {
  public static void main(String[] args) {
    B b = new B("Test");
  }
}

class A {
  A() { this("1", "2"); }

  A(String s, String t) { this(s + t); }

  A(String s) { System.out.println(s); }
}

class B extends A {
  B(String s) { System.out.println(s); }

  B(String s, String t) { this(t + s + "3"); }
```

```
    B() { super("4"); };
  }
```

Select the one correct answer.

a. It will just print `Test`.

b. It will print `Test` followed by `Test`.

c. It will print `123` followed by `Test`.

d. It will print `12` followed by `Test`.

e. It will print `4` followed by `Test`.

5.4 Abstract Classes and Methods

The keyword `abstract` is used in the following contexts in Java:

- Declaring `abstract` classes
- Declaring `abstract` methods in classes (**p. 224**), in interfaces (**p. 240**), and in enum types (**p. 294**)

Abstract Classes

A *concrete* class is one that defines, by virtue of its `public` methods, a *contract* for services it guarantees its clients and provides the *implementation for all the methods* necessary to fulfill that contract. Clients can readily instantiate a concrete class and use its objects.

In certain cases, a class might want to define the contract for the services, but only provide *partial implementation* for its contract. Such a design decision might be necessary if the abstraction the class represents is so general that certain aspects need to be specialized by subclasses to be of practical use, but at the same time guarantee that these *will* be implemented by the subclasses. This design strategy can be implemented by using *abstract classes*. Clients cannot instantiate an `abstract` class, but now its concrete subclasses must be instantiated to provide the necessary objects.

The class `Vehicle` might be declared as an `abstract` class with a partially implemented contract to represent the general abstraction of a vehicle, as creating instances of the class would not make much sense. Its non-`abstract` subclasses, like `Car` or `Bus`, would then provide the implementation necessary to fulfill the contract of the superclass `Vehicle`, making the abstraction more concrete and useful.

The Java SE Platform API contains many `abstract` classes. The `abstract` class `java.lang.Number` is the superclass of wrapper classes that represent numeric values as objects (§8.3, p. 434). The Java Collections Framework makes heavy use of `abstract` classes in implementing commonly used collection data structures (§15.1, p. 783).

Declaring an `abstract` *Class*

An `abstract` class is declared with the modifier `abstract` in its class header. In Example 5.8, the class `Light` at (1) is declared as an `abstract` class. It also declares an `abstract` method `energyCost()` at (2), which has no method body and is essentially a method header (p. 224).

Click here to view code image

```
abstract class Light {                                          // (1) Abstract class
  //...
  // Abstract instance method:
  protected abstract double energyCost(int noOfHours)           // (2) Method header
      throws InvalidHoursException;                             // No method body
}
```

If a class has one or more `abstract` methods, it must be declared as `abstract`, as it is *incomplete*. In Example 5.8, if the `abstract` keyword is omitted from the header of the class `Light` at (1), the compiler will issue an error, as the class declares an `abstract` method and is therefore incomplete.

Like a normal class, an `abstract` class can declare class members, constructors, and initializers. The `abstract` class `Light` in Example 5.8 declares three instance fields—one non-zero argument constructor and three instance methods—in addition to the `abstract` method at (2).

A class that is declared `absract` cannot be instantiated, regardless of whether it has `abstract` methods or not.

Click here to view code image

```
Light porchLight = new Light(21, true, "Porch");    // (5) Compile-time error!
```

The UML class diagram for the inheritance relationship in Example 5.8 is depicted in Figure 5.3. Note that an `abstract` class name and an `abstract` method name are shown in *italics* to distinguish them from concrete classes and methods.

Extending an `abstract` *Class*

A class might choose the design strategy with `abstract` methods to dictate certain behavior, but allow its subclasses the freedom to provide the relevant implementation. An `abstract` class forces its subclasses to provide the subclass-specific functionality stipulated by its `abstract` methods, which is needed to fully implement its abstraction. In other words, subclasses of the `abstract` class have to take a stand and provide implementations of any inherited `abstract` methods before objects can be created. In **Example 5.8**, since the class `Light` is `abstract`, it forces its *concrete* (i.e., non-`abstract`) subclass to provide an implementation for the `abstract` method `energyCost()`. The concrete subclass `TubeLight` provides an implementation for this method at (3).

Click here to view code image

```
class TubeLight extends Light {
  // ...
  // Implementation of the abstract method from the superclass.
  @Override public double energyCost(int noOfHours) {      // (3)
    return  0.15 * noOfHours;
  }
}
```

Creating an object of a subclass results in the fields of all its superclasses, whether these classes are `abstract` or not, to be created and initialized—that is, they *exist* in the subclass object. As with normal classes, the inheritance relationship between classes allows references of the `abstract` superclass type to be declared and used to refer to objects of their subclasses—that is, these references exhibit polymorphic behavior (**p. 278**).

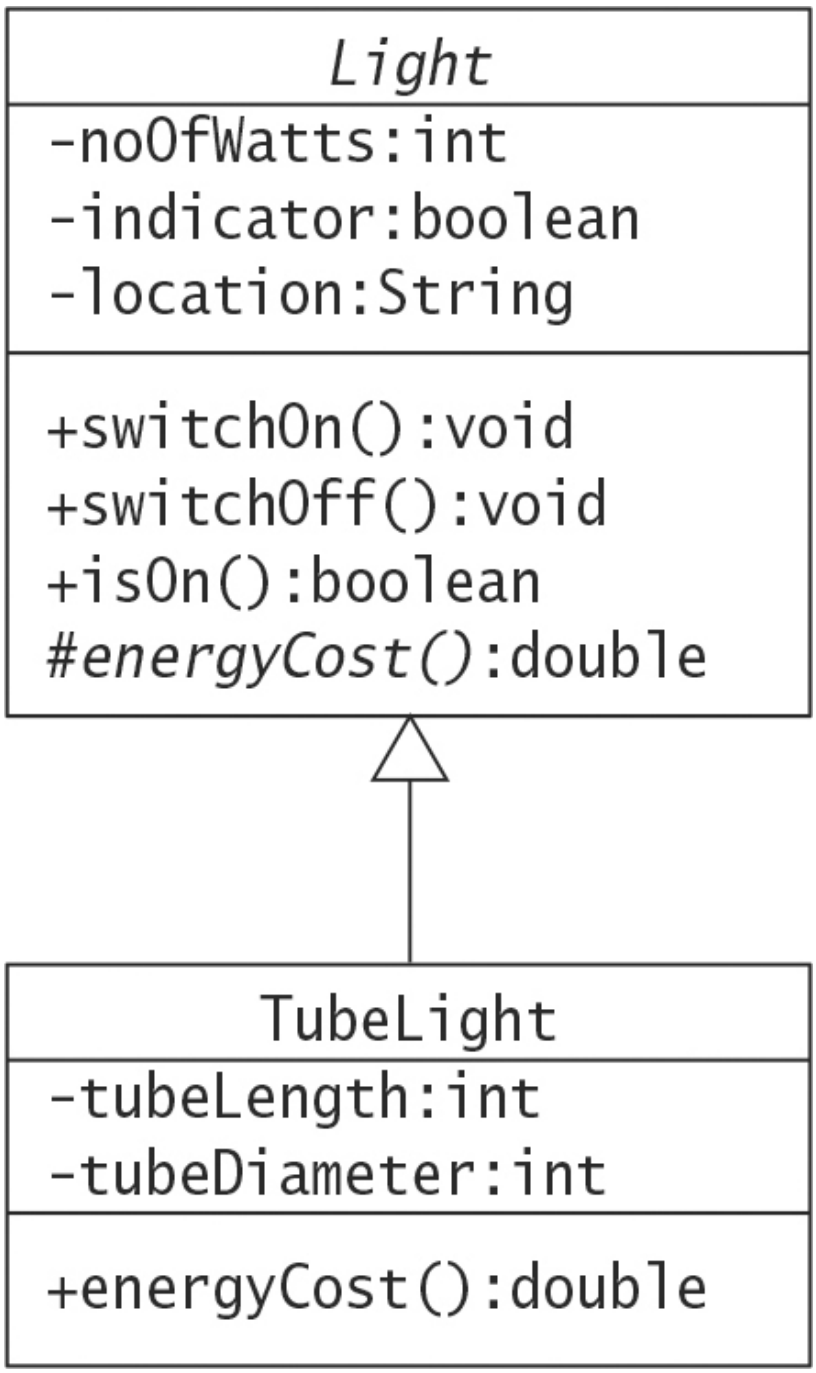

Figure 5.3 *Class Diagram for* Example 5.8

In **Example 5.8**, the class `Factory` creates an instance of the subclass `TubeLight` at (6). The `private` fields declared in the `abstract` superclass `Light` can be accessed indirectly by invoking the `public` methods it provides on objects of the subclass `TubeLight.` The subclass reference `cellarLight` is used to invoke `public` methods in the superclass in the following code:

Click here to view code image

```
TubeLight cellarLight = new TubeLight(18, true, "Cellar", 590, 26);  // (6)
cellarLight.switchOff();                        // Method in superclass
System.out.println(cellarLight.isOn());         // Method in superclass: false
```

The subclass reference `cellarLight` of course can be used to invoke subclass-specific methods, as shown at (7).

Click here to view code image

```
System.out.printf("Energy cost ($): %2.2f%n",
    cellarLight.energyCost(40));              // (7) Using subclass reference
```

References of an `abstract` superclass can be declared and assigned reference values of its subclass objects, as shown at (8). Superclass references can be used to manipulate subclass objects, as shown at (9), where the `energyCost()` method from the subclass `TubeLight` is executed.

Click here to view code image

```
Light nightLight = new TubeLight(15, false, "Bedroom", 850, 15);     // (8)
System.out.printf("Energy cost ($): %2.2f%n",
    nightLight.energyCost(30));             // (9) Using superclass reference
                                            // Invokes method in subclass
                                            // Requires throws clause at (4)
```

Note that using the subclass reference `cellarLight` at (7) to invoke the method `energyCost()` cannot throw a checked exception, as readily seen from its declaration in the subclass `TubeLight`. However, using the superclass reference `nightLight` at (9) to invoke the method `energyCost()` can throw a checked exception, as seen from the method declaration in the superclass `Light`. At compile time, only the static type of the reference is known, namely `Light`, and the method `energyCost()` in this class throws a checked `InvalidHoursException` (§7.2, p. 374). The `throws` clause in the `main()` method at (4) specifies this exception—otherwise, the code will not compile.

In the code below, the class `AbstractArt` at (2) must be declared as `abstract` as it does not implement the `abstract` method `paint()` from its superclass `Art` at (1).

Click here to view code image

```
abstract class Art { abstract void paint(); }           // (1) Abstract class

abstract class AbstractArt extends Art {}               // (2) Must be abstract

class MinimalistArt extends AbstractArt {               // (3) Concrete class
  @Override void paint() { System.out.println(":-)"); } // (4) Concrete method
}

abstract class PostModernMinimalistArt
                                extends MinimalistArt { // (5) Abstract class
  @Override void paint() { System.out.println(":-("); } // (6) Concrete method
                                                        //     overrides (4)
```

```
}

class ArtsyFartsy extends PostModernMinimalistArt {}     // (7) Concrete class
```

Analogous to a normal class, an `abstract` class can only extend a single non-`final` class that can be either concrete or `abstract`. In the code above, the `abstract` class `AbstractArt` at (2) extends the `abstract` class `Art,` and the `abstract` class `PostModernMinimalistArt` at (5) extends the concrete class `MinimalistArt.`

A non-`final` concrete class, which by definition has no `abstract` methods, can be *considered incomplete* by declaring it as `abstract`. The `PostModernMinimalistArt` class at (5) is declared `abstract` and considered incomplete, even though it is concrete. It cannot be instantiated. However, its subclass `ArtsyFartsy` at (7) is a concrete class, as it inherits the concrete method `paint()` from its `abstract` superclass `PostModernMinimalistArt`.

A class cannot be declared both `final` and `abstract`—that would be a contradiction in terms: A `final` class cannot be extended, but an `abstract` class is incomplete or considered to be incomplete and must be extended.

An `abstract` class should not be used to implement a class that cannot be instantiated. The recommended practice is to only provide a zero-argument constructor that is `private`, thus making sure that it is never invoked in the class.

In many ways `abstract` classes and interfaces are similar, and interfaces can be used with advantage in many cases. However, if private state should be maintained with instance members, then `abstract` classes are preferred, as interfaces do not have any notion of state.

Analogous to a normal class, an `abstract` class can implement multiple interfaces (p. 240).

Example 5.8 *Using Abstract Classes*

Click here to view code image

```
// File: Factory.java
// Checked exceptions:
class InvalidHoursException extends Exception {}
class NegativeHoursException extends InvalidHoursException {}
class ZeroHoursException extends InvalidHoursException {}

abstract class Light {                                    // (1) Abstract class
  // Fields:
```

```
  private int     noOfWatts;       // Wattage
  private boolean indicator;       // On or off
  private String  location;        // Placement

  // Non-zero argument constructor:
  Light(int noOfWatts, boolean indicator, String location) {
    this.noOfWatts = noOfWatts;
    this.indicator = indicator;
    this.location  = location;
  }

  // Instance methods:
  public void switchOn()  { indicator = true; }
  public void switchOff() { indicator = false; }
  public boolean isOn()   { return indicator; }

  // Abstract instance method:
  protected abstract double energyCost(int noOfHours)      // (2) Method header
      throws InvalidHoursException;                        // No method body
}
//_____________________________________________________________________________
class TubeLight extends Light {
  // Instance fields:
  private int tubeLength;                                  // millimeters
  private int tubeDiameter;                                // millimeters

  // Non-zero argument constructor
  TubeLight(int noOfWatts, boolean indicator, String location,
            int tubeLength, int tubeDiameter) {
    super(noOfWatts, indicator, location);  // Calling constructor in superclass.
    this.tubeLength = tubeLength;
    this.tubeDiameter = tubeDiameter;
  }

  // Implementation of the abstract method from the superclass.
  @Override public double energyCost(int noOfHours) {      // (3)
    return  0.15 * noOfHours;
  }
}
//_____________________________________________________________________________
public class Factory {
  public static void main(String[] args) throws InvalidHoursException {  // (4)
//  Light porchLight = new Light(21, true, "Porch");    // (5) Compile-time error!
    TubeLight cellarLight = new TubeLight(18, true, "Cellar", 590, 26);  // (6)
    cellarLight.switchOff();
    System.out.println(cellarLight.isOn());        // false
    System.out.printf("Energy cost ($): %2.2f%n",
        cellarLight.energyCost(40));               // (7) Using subclass reference
    Light nightLight = new TubeLight(15, false, "Bedroom", 850, 15);     // (8)
```

```
    System.out.printf("Energy cost ($): %2.2f%n",
        nightLight.energyCost(30));             // (9) Using superclass reference
                                                // Invokes method in subclass
                                                // Requires throws clause in (4)
  }
}
```

Output from the program:

```
false
Energy cost ($): 6.00
Energy cost ($): 4.50
```

Abstract Methods in Classes

In this subsection we discuss in more detail declaring and overriding `abstract` methods in classes.

Declaring an `abstract` *Method*

An `abstract` method in an `abstract` class has the following syntax:

Click here to view code image

```
access_modifier abstract return_type method_name (formal_parameter_list)
                                                 throws_clause;
```

An `abstract` method does not have an implementation; that is, no method body is defined for an `abstract` method, and only the *method header* is provided in the class declaration. The keyword `abstract` is mandatory in the header of an `abstract` method declared in a class. Its class is then incomplete and must be explicitly declared as `abstract`. Subclasses of an `abstract` class must then override the `abstract` method to provide the method implementation; otherwise, they must also be declared as `abstract`.

Overriding an `abstract` *Method*

When overriding an `abstract` method from the superclass, the notation `@Override` should always be used in the overriding method in the subclass. The compiler will issue an error if the override criteria are not satisfied.

The accessibility of an `abstract` method declared in a top-level class cannot be `private`, as subclasses would not be able to override the method and provide an imple-

mentation. Thus an `abstract` method in a top-level class can only have `public`, `protected`, or package accessibility.

In **Example 5.8**, the `abstract` instance method in the `abstract` superclass `Light` has the following declaration:

Click here to view code image

```
protected abstract double energyCost(int noOfHours)      // (2) Method header
    throws InvalidHoursException;                        // No method body
```

It has `protected` access and has type `double` as the return type. Its method signature is `energyCost(int)`, and it throws the checked `InvalidHoursException`.

The implementation of the `abstract` method in the subclass `TubeLight` has the following declaration:

Click here to view code image

```
@Override public double energyCost(int noOfHours) {      // (3)
  return  0.15 * noOfHours;
}
```

It has `public` access and has type `double` as the return type. Its method signature is `energyCost(int)`, and it has no `throws` clause. Widening the access to `public` access and throwing no checked exceptions are allowed according to the override criteria.

Since an `abstract` method must be overridden to provide an implementation, only an instance method can be declared as `abstract`. Since `static` methods cannot be overridden, declaring an `abstract static` method makes no sense, and the compiler will report an error.

An `abstract` method can be overloaded just like a normal method. The following method declaration in either the superclass or the subclass overloads the method named `energyCost`, as it has a different signature: `energyCost()`.

Click here to view code image

```
public double energyCost() {          // Overloaded
  return 1.75;
}
```

If an attempt to override or overload an `abstract` method fails, the compiler will issue an error. If either of these two methods is declared in a subclass of the `Light` class, the compiler will issue an error.

Click here to view code image

```
@Override
double energyCost(int numOfHours) {   // Not overridden! Narrows accessibility!
  return 2.0 * numOfHours;
}

public Double energyCost(int numOfHours) {  // Not overloaded! Duplicate method!
  return 3.5 * numOfHours;
}
```

An `abstract` method or a non-`final` concrete method in a class can be overridden by an `abstract` method in a subclass. This is governed by the same rules for method overriding.

A method cannot be both `final` and `abstract`—that would be a contradiction in terms: A `final` method cannot be overridden, but an `abstract` method must be overridden to provide an implementation.

For a discussion of `abstract` methods in top-level interfaces, see §5.6, p. 240. Abstract methods can also be declared in an enum type, if the enum type contains constant-specific class bodies that implement these methods (p. 294).

5.5 Final Declarations

The keyword `final` can be used in the following contexts in Java:

- Declaring `final` classes (p. 225)
- Declaring `final` members in classes (p. 226), and in enum types (p. 294)
- Declaring `final` local variables in methods and blocks (p. 231)
- Declaring `final static` variables in interfaces (p. 254)

Final Classes

A class can be declared `final` to indicate that it cannot be extended—that is, one cannot define subclasses of a `final` class. In other words, the class behavior cannot be changed by extending the class. This implies that one cannot override or hide any methods declared in such a class. Its methods behave as `final` methods (p. 226). A `final` class marks the lower boundary of its *implementation inheritance hierarchy*.

A *concrete* class is a class that has only *concrete* methods—that is, methods that are non-`abstract`, and therefore have an implementation. Only a concrete class can be declared `final`. If it is decided that the class `TubeLight` at (12) in **Example 5.9** may not be extended, it can be declared `final`. Any attempt to specify the class name `TubeLight` in an `extends` clause will result in a compile-time error.

Click here to view code image

```
final class TubeLight extends Light {                  // (12)
  // ...
}

class NeonLight extends TubeLight {                     // Compile-time error!
  // ...
}
```

A `final` class must be complete, whereas an `abstract` class is considered incomplete. Classes, therefore, cannot be both `final` and `abstract` at the same time. Interfaces are implicitly `abstract`, and therefore cannot be declared as `final`. A `final` class and an interface with only `abstract` methods represent two diametrical strategies when it comes to providing an implementation: the former with all the methods implemented and the latter with none. An `abstract` class or an interface with partial implementation represents a compromise between these two strategies.

The Java SE Platform API includes many `final` classes—for example, the `java.lang.String` class and the wrapper classes for primitive values.

Final Methods in Classes

A `final` method in a class is a *concrete* method (i.e., has an implementation) and cannot be overridden or hidden in any subclass. Any normal class can declare a `final` method. The class need not be declared `final`.

In **Example 5.9**, the non-`final` class `Light` defines the `final` method `setWatts()` at (10).

Click here to view code image

```
class Light {                                       // (1)
  // ...
  public final void setWatts(int watt) {            // (10) Final instance method
    noOfWatts = watt;
  }
}
```

The subclass `TubeLight` attempts to override the `final` method `setWatts()` from the superclass `Light` at (14), which is not permitted.

Click here to view code image

```
class TubeLight extends Light {                  // (12)
  // ...
  @Override
  public void setWatts(int watt) {   // (14) Cannot override final method at (10)!
    noOfWatts = 2*watt;
  }
}
```

A call to a `final` method is bound at compile time; as such, a method cannot be overridden by a subclass and therefore there is no dynamic lookup to perform at runtime. A method that is declared `final` allows the compiler to do code optimization by replacing its method call with the code of its method body, a technique called *inlining*. Use of the `final` keyword for exceptional performance enhancement is not recommended, as the compiler is too smart to fall for that trick.

A subclass cannot override or hide a `private` method from its superclass. However, it can define a new method using the same method signature as the `private` method, without regard to the other criteria for overriding: the return type and the `throws` clause. Defining a new method using the same method signature is *not* possible with a `final` method which cannot be overridden or hidden. Declaring a `private` method as `final` in addition is redundant.

Although a constructor cannot be overridden, it also *cannot* be declared `final`.

The `java.lang.Object` class—the superclass of all classes—defines a number of `final` methods (`notify()`, `notifyAll()`, and `wait()`—mostly to synchronize the running of threads) that all objects inherit, but cannot override.

Example 5.9 *Using the* `final` *Modifier*

Click here to view code image

```
import java.util.Arrays;
import static java.lang.System.out;

class Light {                                     // (1)
  // Static final variables:
  public static final double KWH_PRICE = 3.25;    // (2) Constant static variable
  public static final String MANUFACTURER;        // (3) Blank final static field
```

```
  static {                                        // Static initializer block
    MANUFACTURER = "Ozam";                        // (4) Initializes (3)
  }

  // Instance variables:
  int noOfWatts;                                  // (5)
  final String color;                             // (6) Blank final instance field
  final String energyRating;                      // (7) Blank final instance field

  {                                               // Instance initializer block
    color = "off white";                          // (8) Initializes (6)
  }

  // Constructor:
  Light() {
    energyRating = "A++";                         // (9) Initializes (7)
  }

  public final void setWatts(int watt) {          // (10) Final instance method
    this.noOfWatts = watt;
  }

  public static void setKWH(double rate) {
//  KWH_PRICE = rate;                             // (11) Not OK. Final field.
  }
}
//_____________________________________________________________________________
class TubeLight extends Light {                   // (12)
  static StringBuilder color = new StringBuilder("green");      // (13) Hiding (6)

//@Override
//public void setWatts(int watt) {   // (14) Cannot override final method at (10)!
//  noOfWatts = 2*watt;
//}
}
//_____________________________________________________________________________
public class Warehouse {
  public static void main(final String[] args) { // (15) Final parameter

    final Light workLight = new Light(); // (16) Non-blank final local variable.
    workLight.setWatts(100);             // (17) OK. Changing object state.
//  workLight.color = "pink";            // (18) Not OK. Final instance field.
//  workLight = new Light();             // (19) Not OK. Changing final reference.

    final Light alarmLight;              // (20) Blank final local variable.
//  alarmLight.setWatts(200);            // (21) Not OK. Not initialized.

    Light carLight;                      // (22) Non-final local variable.
```

```
//  carLight.setWatts(10);              // (23) Not OK. Not initialized.

    out.println("Accessing final static fields in class Light:");
    out.println("KWH_PRICE:    " + Light.KWH_PRICE);
    out.println("MANUFACTURER: " + Light.MANUFACTURER);

    out.println("Accessing final instance fields in an object of class Light:");
    out.println("noOfWatts:    " + workLight.noOfWatts);
    out.println("color:        " + workLight.color);
    out.println("energyRating: " + workLight.energyRating);

    out.println("Fun with final parameter args:");
    out.println(Arrays.toString(args));  // Print array.
    out.println("args length: " + args.length);
    args[0] = "1";                       // (24) OK. Modifying array state.
    out.println(Arrays.toString(args));  // Print array.
//  args = null;                         // (25) Not OK. Final parameter.
//  args.length = 10;                    // (26) Not OK. Final instance field.
  }
}
```

Output from the program when run with the following command:

Click here to view code image

```
>java Warehouse One Two Three
Accessing final static fields in class Light:
KWH_PRICE:    3.25
MANUFACTURER: Ozam
Accessing final instance fields in an object of class Light:
noOfWatts:    100
color:        off white
energyRating: A++
Fun with final parameter args:
[One, Two, Three]
args length: 3
[1, Two, Three]
```

Final Variables

A `final` *variable* is a variable whose value cannot be changed during its lifetime once it has been initialized. A `final` variable is declared with the keyword `final` in its declaration. Any attempt to reassign a value will result in a compile-time error.

Declaring a variable as `final` has the following implications:

- A `final` variable of a primitive data type cannot change its value once it has been initialized.
- A `final` variable of a reference type cannot change its reference value once it has been initialized. This effectively means that a `final` reference will always refer to the same object. However, the keyword `final` has no bearing on whether the *state of the object* denoted by the reference can be changed.
- The compiler may perform code optimizations for `final` variables, as certain assumptions can be made about such code.

For all paths of execution through the code, the compiler checks that a `final` variable is assigned only once before it is accessed, and when in doubt, the compiler issues an error (**p. 232**).

A *blank* `final` *variable* is a `final` variable whose declaration does not specify an initializer expression.

A *constant variable* is a `final` variable of either a primitive type or type `String` that is initialized with a *constant expression*. Constant variables can be used as `case` labels of a `switch` statement.

We distinguish between two kinds of `final` variables: `final` *fields* that are members in a class declaration and `final` *local variables* that can be declared in methods and blocks. A `final` *field* is either `static` or non-`static` depending on whether it is declared as a `static` or non-`static` member in the class.

Example 5.9 provides examples of different kinds of `final` variables. The declarations from **Example 5.9** are shown here.

Click here to view code image

```
class Light {                                        // (1)
  public static final double KWH_PRICE = 3.25;       // (2) Constant static variable
  public static final String MANUFACTURER;           // (3) Blank final static field

  int noOfWatts;                                     // (5) Non-final instance field
  final String color;                                // (6) Blank final instance field
  final String energyRating;                         // (7) Blank final instance field
  // ...
}

public class Warehouse {
  public static void main(final String[] args) { // (15) Final parameter

    final Light workLight = new Light(); // (16) Non-blank final local variable.
```

```
      final Light alarmLight;              // (20) Blank final local variable.

      Light carLight;                      // (22) Non-final local variable.
      // ...
    }
  }
```

In the next two subsections we elaborate on `final` fields in classes and `final` local variables that can be declared in methods and blocks.

Final Fields in Classes

A `final` field must be explicitly initialized only once with an initializer expression, either in its declaration or in an initializer block. A `final` instance field can also be initialized in a constructor. After the class completes initialization, the `final static` fields of the class are all guaranteed to be initialized. After a constructor completes execution, the `final` instance fields of the current object are all guaranteed to be initialized. The compiler ensures that the class provides the appropriate code to initialize the `final` fields.

In **Example 5.9**, the class `Light` defines two `final static` fields at (2) and (3). The field `KWH_PRICE` is a constant variable that is initialized by the constant expression `3.25` in the declaration at (2). The field `MANUFACTURER` is a blank `final static` field that is initialized in the `static` initializer block at (4). All `static` initializer blocks are executed at class initialization. Note that a blank `final static` field *must* be initialized in a `static` initializer block; otherwise, the compiler will issue an error. An attempt to change the value of the `final static` field `KWH_PRICE` at (11) in a method results in a compile-time error.

In **Example 5.9**, the class `Light` also defines two `final` instance fields at (6) and (7). The instance field `color` is a blank `final` instance field that is initialized in the instance initializer block at (8), and the instance field `energyRating` is a blank `final` instance field that is initialized in the constructor at (9). If either the assignment at (8) in the instance initializer block or the assignment at (9) in the constructor is removed, it will result in a compile-time error.

Note that a blank `final` instance field *must* be initialized either in an instance initializer block or in a constructor; otherwise, the code will not compile. Each time an object is created using the `new` operator with a constructor call, all instance initializer blocks are executed, but not necessarily all constructors, as this depends on constructor chaining. If the class has several constructors, code must make sure that a blank `final` instance field is initialized no matter which constructor is called to initialize the object.

Since `final static` fields are initialized at class initialization time and `final` instance fields are initialized at object creation time, the compiler will issue an error if an attempt is made to assign a value to a `final` field inside any method.

Analogous to a non-`final` field, a `final` field with the same name *can* be hidden in a subclass—in contrast to `final` methods that can neither be overridden nor hidden. In **Example 5.9**, the blank `final` instance field `color` of type `String` at (6) in class `Light` is hidden by the `static` field `color` of type `StringBuilder` at (13) in subclass `TubeLight`.

Fields that are `final` and `static` are commonly used to define *manifest constants* (also called *named constants*). For example, the minimum and maximum values of the numerical primitive types are defined by `final static` fields in the respective wrapper classes. The `final static` field `Integer.MAX_VALUE` defines the maximum `int` value. The field `java.lang.System.out` is a `final static` field. Fields defined in an interface are implicitly `final static` fields (**p. 254**), as are the enum constants of an enum type (**p. 287**).

Final Local Variables in Methods

A `final` local variable need not be initialized in its declaration—that is, it can be a blank `final` local variable—but it must be initialized in the code before it is accessed. However, the compiler does not complain as long as the local variable is not accessed—this is in contrast to `final` fields that must be explicitly assigned a value, whether they are accessed or not.

In **Example 5.9**, the `main()` method at (15) in the class `Warehouse` defines a `final` local reference `workLight` at (16). The state of the object denoted by the reference `workLight` is changed at (17), but an attempt to change the value of the `final` instance field `color` of this object at (18) does not succeed. The compiler also reports an error at (19), since the reference value of the `final` local variable `work-Light` cannot be changed. A blank `final` local reference `alarmLight` is declared at (20). The compiler reports an error when an attempt is made to use this reference at (21) before it is initialized. The non-`final` local reference `carLight` at (22) also cannot be accessed at (22) before it is initialized. However, note that the compiler does not complain as long as the local variables at (20) and (22) are not accessed.

Click here to view code image

```
    final Light workLight = new Light(); // (16) Final local variable.
    workLight.setWatts(100);             // (17) OK. Changing object state.
//  workLight.color = "pink";            // (18) Not OK. Final instance field.
//  workLight = new Light();             // (19) Not OK. Changing final reference.
```

```
    final Light alarmLight;              // (20) Blank final local variable.
//  alarmLight.setWatts(200);            // (21) Not OK. Not initialized.

    Light carLight;                      // (22) Non-final local variable.
//  carLight.setWatts(10);               // (23) Not OK. Not initialized.
```

Local variables in certain contexts are considered to be *implicitly declared* `final`: an exception parameter of a multi-`catch` clause (§7.6, p. 397) and a local variable that refers to a resource in a `try`-with-resources statement (§7.7, p. 407).

`final` Parameters

A formal parameter can be declared with the keyword `final` preceding the parameter declaration in the method header. A `final` parameter is a special case of a blank `final` local variable. It is initialized with the value of the corresponding argument in the method call, before the code in the body of the method is executed. Declaring parameters as `final` prevents their values from being changed inadvertently in the method. A formal parameter's declaration as `final` does not affect the caller's code.

In the declaration of the `main()` method at (15) in **Example 5.9**, the parameter `args` is declared as `final`. The array `args` will be initialized with the program arguments on the command line when execution starts.

Click here to view code image

```
public static void main(final String[] args) {  // (15) Final parameter
  // ...
  out.println("Fun with final parameter args:");
  out.println(Arrays.toString(args));          // (24) Print array.
  out.println("args length: " + args.length); // (25) Access final field
  args[0] = "1";                               // (26) OK. Modifying array state.
  out.println(Arrays.toString(args));          // Print array.
//args = null;                                 // (27) Not OK. Final parameter.
//args.length = 10;                           // (28) Not OK. Final instance field.
}
```

The state of the array object `args` is printed at (24). Its `final` field `length` is accessed at (25). The state of the `args` array is changed at (26), and is printed again. However, the compiler will not allow the reference value of the `final` parameter `args` to be changed at (27), nor will it allow the value of the `final` instance field `args.length` to be changed at (28).

Definite Assignment Analysis for Final Variables

The *name* of a `final` variable can occur in the following contexts: in its declaration when it is declared, in the context of an assignment when it is assigned a value, and in the context where its value is accessed in an expression.

The analysis performed by the compiler determines whether the blank `final` variable is initialized *before* its value is accessed. This involves checking whether a blank `final` variable has been assigned a value on any possible path of execution to where the value of the variable is accessed. In technical terms, this means a blank `final` variable must be *definitely assigned* before any access.

Click here to view code image

```
final int k;              // Declaration: blank final local variable
k = 10;                   // (1) Assignment
System.out.println(k);    // (2) Access: k is definitely assigned.

boolean status = true;    // Non-constant variable.
final int j;              // Declaration: blank final local variable
if (status) {             // (3) Value of conditional expression not evaluated
  j = 5;                  // (4) Conditional assignment
}
System.out.println(j);    // (5) Access: j is not definitely assigned!
```

The blank `final` local variable `k` is accessed at (2) after it has been assigned a value at (1). It is *definitely assigned* before access at (2). The situation is different at (5). The compiler can deduce that the blank `final` local variable `j` can be assigned a value in the `if` block, depending on the conditional expression in the `if` statement. In other words, the structure of the `if` statement is considered in the analysis, but not the non-constant conditional expression. The compiler cannot guarantee that the assignment at (4) will be executed. The conclusion being that the blank `final` local variable `j` is *not definitely assigned* before access at (5). The statement at (5) does not compile. However, if the conditional expression in the `if` statement at (3) is replaced with the `boolean` literal `true`, the compiler can determine that the assignment at (4) will be executed and the variable `j` at (5) becomes definitely assigned before access.

The flow analysis performed is conservative. The compiler does not always evaluate all expressions, particularly method calls. However, it does evaluate `boolean-` valued constant expressions, and it also takes into consideration the syntax of statements and expressions, including the use of the conditional operators `!`, `&&`, `||`, and `? :`.

For each assignment to a blank `final` variable, the compiler must also determine that it is assigned *exactly once*—in other words, it has not already been assigned a value.

This means that, given an assignment to a blank `final` local variable, there should not be any other assignment to this variable on any possible path of execution prior to this assignment. In technical terms, this means a blank `final` variable must be *definitely unassigned* before any assignment.

Click here to view code image

```
int n = 5;
final int k;            // Declaration: blank final local variable
if (n >= 4) {           // (1) Value of conditional expression not evaluated
  k = 6;                // (2) Conditional assignment: k is definitely unassigned
}
k = 12;                 // (3) Assignment: k is not definitely unassigned!
System.out.println(k);  // (4) Access: k is definitely assigned
```

The blank `final` local variable `k` in the assignment at (2) is definitely unassigned, as there is no other assignment on a path of execution before this assignment. However, the variable `k` at (3) is *not* definitely unassigned, as the assignment at (2) in the `if` statement can be executed on a path of execution before the assignment at (3). The compiler reports that the variable `k` at (3) may already have been assigned a value. The assignment at (3) does not compile. Note that the variable `k` at (4) is definitely assigned before the access, because of the assignment at (3). Removing the assignment statement at (3) makes the variable `k` at (4) *not* definitely assigned, as in the previous example.

If the code is modified by replacing the `if` statement at (3) with an `if-else` statement, where the assignment at (3) is done in the `else` block, the variable k at (3) also becomes definitely unassigned—there is no assignment to the variable `k` before this assignment. Regardless of which assignment executes in the `if-else` statement, the variable `k` at (4) is definitely assigned before access.

Click here to view code image

```
int n = 5;
final int k;            // Declaration: blank final local variable
if (n >= 4) {           // (1) Value of conditional expression not evaluated
  k = 6;                // (2) Conditional assignment: k is definitely unassigned
} else {
  k = 12;               // (3) Conditional assignment: k is definitely unassigned
}
System.out.println(k);  // (4) Access: k is definitely assigned
```

The two rules of definitely assigned before an access and definitely unassigned before an assignment help to determine correct usage of blank `final` variables. For non-`fi-`

`nal` *local variables*, the rule for definitely assigned before an access is sufficient, as the value of such variables can be changed.

The examples below shed more light on the usage of `final` variables.

Click here to view code image

```
{
  final int i = 10;
  i++;               // Not OK. Side effect changes the value of i.
}

{
  for (int i = 0; i < 10; i++) {
    final int j = i; // OK. Final variable goes out of scope after each iteration.
  }
}

{
  final int j;
  for (int i = 0; i < 10; i++) {
    j = i;  // Not OK. Loop either not executed and j not initialized
            // or each iteration will assign a new value to j.
  }
  System.out.println(j); // Not OK. Not guaranteed that j is initialized.
}
```

Review Questions

5.8 Which one of the following class declarations is a valid declaration of a class that cannot be instantiated?

Select the one correct answer.

a.

Click here to view code image

```
class Ghost          { abstract void haunt(); }
```

b.

Click here to view code image

```
abstract class Ghost { void haunt(); }
```

c.

Click here to view code image

```
abstract class Ghost { void haunt() {}; }
```

d.

Click here to view code image

```
abstract Ghost       { abstract void haunt(); }
```

e.

Click here to view code image

```
abstract class Ghost { abstract haunt(); }
```

5.9 Given the following classes:

Click here to view code image

```
public class Animal {
  // (1)
}

public abstract class Cat extends Animal {
  // (2)
}
```

and the following method definition:

Click here to view code image

```
public abstract void eat();
```

where can this abstract method be inserted?

Select the one correct answer.

a. It can be inserted at (1).

b. It can be inserted at (2).

c. It can be inserted at both (1) and (2).

d. It cannot be inserted into any of the classes.

5.10 Which statement is true about an `abstract` class?

Select the one correct answer.

a. An `abstract` class cannot have a constructor.

b. An `abstract` class cannot have concrete methods.

c. An `abstract` class cannot override methods from its supertypes.

d. An `abstract` class cannot be instantiated with the `new` operator.

5.11 Which statement is true about the following classes?

Click here to view code image

```
abstract class LivingOrganism {
  public abstract void feed();
}

class Bacteria extends LivingOrganism {
  @Override
  public void feed()  {  }
  public void feed(int quantity)  {  }        // (1)
}

public class Lab {
  public static void main(String[] args) {
    LivingOrganism org = new Bacteria();      // (2)
    org.feed();
    ((Bacteria)org).feed(10);                 // (3)
  }
}
```

Select the one correct answer.

a. Class `Bacteria` cannot overload the `feed()` method at (1).

b. An `@Overload` annotation can be specified for the `feed()` method at (1).

c. An `@Override` annotation can be specified for the `feed()` method at (1).

d. An instance of `Bacteria` can be assigned to the `org` variable at (2).

e. Class `Lab` at (3) cannot invoke the `feed()` method at (1).

5.12 Which of the following statements are true? Select the two correct answers.

a. In Java, the `extends` clause is used to specify the inheritance relationship.

b. The subclass of a non-`abstract` class can be declared as `abstract`.

c. All members of the superclass are inherited by the subclass.

d. A `final` class can be `abstract`.

e. A class in which all the members are declared `private` cannot be declared as `public`.

5.13 Which one of the following class declarations is a valid declaration of a class that cannot be extended?

Select the one correct answer.

a. `class Link { }`

b. `abstract class Link { }`

c. `final class Link { }`

d. `abstract final class Link { }`

5.14 Given the following source code, which comment line can be uncommented without introducing errors?

Click here to view code image

```
abstract class MyClass {
  abstract void f();
  final    void g() {}
//final    void h() {}                          // (1)

  protected static int i;
```

```
  private          int j;
}

final class MyOtherClass extends MyClass {
//MyOtherClass(int n) { m = n; }               // (2)

  public static void main(String[] args) {
    MyClass mc = new MyOtherClass();
  }

  void f() {}
  void h() {}
//void k() { i++; }                            // (3)
//void l() { j++; }                            // (4)

  int m;
}
```

Select the one correct answer.

a. (1)

b. (2)

c. (3)

d. (4)

5.15 Which of the following statements are true about modifiers? Select the two correct answers.

a. Abstract classes can declare `final` methods.

b. Fields can be declared as `abstract`.

c. Non-`abstract` methods can be declared in `abstract` classes.

d. Abstract classes can declare `default` methods.

e. Abstract classes can be declared as `final`.

5.16 Which statement is true about the following classes?

Click here to view code image

```
public abstract class Animal {
  public static final MAX_SIZE = 10;
  public abstract void measure(int size);
}

public class Cat extends Animal {
  private int size;
  @Override
  public final void measure(int size) {
    this.size = (size < Animal.MAX_SIZE) ? size : Animal.MAX_SIZE;
  }
}
```

Select the one correct answer.

a. The compilation fails because of the `final` variable declaration in the abstract class.

b. The compilation fails because of the `final` method overriding an `abstract` method.

c. The compilation fails because `@Override` annotation cannot be applied to a `final` method.

d. The compilation succeeds.

5.17 Which statement is true about `final` classes and methods?

Select the one correct answer.

a. A `final` class cannot contain `abstract` methods.

b. An `abstract` class cannot contain `final` methods.

c. A `final` class cannot contain `final` methods.

d. A `final` method cannot override an `abstract` method.

5.6 Interfaces

An interface defines a *contract* for services that classes can implement. Objects of such classes guarantee that this contract will be honored.

Before diving into interfaces, an overview of the inheritance relationship between *classes* can be useful. The `extends` clause in a class definition only allows *linear inheritance* between classes—that is, a subclass can only extend one superclass. A superclass

reference can refer to objects of its own type and of its subclasses strictly according to the linear inheritance hierarchy. Note that this inheritance relationship between classes comprises both *inheritance of type* (i.e., a subclass inherits the type of its superclass and can act as such) and *inheritance of implementation* (i.e., a subclass inherits methods and fields from its superclass). Since this relationship is linear, it rules out *multiple inheritance of implementation*, in which a subclass can inherit implementation directly from more than one direct superclass.

As we shall see in this section, *interfaces* not only allow new named reference types to be introduced, but their usage can result in both *multiple inheritance of type* and *multiple inheritance of implementation*. As we shall also see, multiple inheritance of type does not pose any problems, but multiple inheritance of implementation does and is disallowed by the compiler.

Defining Interfaces

A top-level interface has the following syntax:

Click here to view code image

```
access_modifier interface interface_name
                          optional_type_parameter_list
                          optional_extends_interface_clause // Interface header
{ // Interface body
    abstract_method_declarations
    default_method_declarations
    static_method_declarations
    private_instance_method_declarations
    private_static_method_declarations
    constant_declarations
    member_type_declarations
}
```

In the interface header, the name of the interface is preceded by the keyword `interface`. In addition, the interface header can specify the following information:

- The *access modifier* can be `public` or `private`. Lack of an access modifier implies package accessibility.
- The *optional type parameter list* specifies a comma-separated list of any formal type parameters enclosed by angle brackets (`<>`) for declaring a *generic interface* (**§11.2, p. 572**).
- The *optional extends interface clause* specifies a comma-separated list of any superinterfaces that the interface extends (**p. 244**).

The interface body can contain *member declarations* that include any of the following:

- *Abstract method declarations* (p. 240)
- *Default method declarations* (p. 246)
- *Static method declarations* (p. 251)
- *Private instance and static method declarations* (p. 252)
- *Constant declarations* (p. 254)
- *Member type declarations* (§9.1, p. 491)

An interface is `abstract` by definition, which means that it cannot be instantiated. Declaring an interface as `abstract` is superfluous and seldom done in practice. It is the only non-access modifier that can be specified for a top-level interface.

The member declarations can appear in any order in the interface body, which can be empty. Since interfaces are meant to be implemented by classes, interface members implicitly have `public` access and the `public` modifier can be omitted. The following declaration is an example of a bare-bones interface that has an empty body:

```
interface Playable { }
```

Interfaces with empty bodies can be used as *markers* to *tag* classes as having a certain property or behavior. Such interfaces are also called *ability* interfaces. The Java SE Platform API provides several examples of such marker interfaces—for example, `java.lang.Cloneable`, `java.io.Serializable` (§20.5, p. 1261), and `java.util.EventListener`. However, *annotations* (Chapter 25, p. 1555) are a better solution than marker interfaces for attaching metadata to class definitions.

Table 5.2 and Table 5.3 summarize the salient properties of member declarations that can be included in an interface, and which we will elaborate on in this section.

Table 5.2 *Summary of Member Declarations in an Interface (Part I)*

Member declarations	Abstract instance method	Default instance method	Static method	Private instance method	Private static method
Access modifier:	Implicitly `public`	Implicitly `public`	Implicitly `public`	`private` mandatory	`private` mandatory
Non-access modifier:	Implicitly `abstract`	`default` mandatory	`static` mandatory	None	`static` mandatory

		tory	tory		tory
Implemented:	No	Yes	Yes	Yes	Yes
Can be inherited:	If not overrid-den	If not overrid-den	No	No	No

Table 5.3 *Summary of Member Declarations in an Interface (Part II)*

Member declarations	Constant	Member type declaration
Access modifier:	Implicitly `public`	Implicitly `public`
Non-access modifier:	Implicitly `static` and `fi-nal`	Implicitly `static`
Implemented:	Yes	Yes
Can be inherited:	If not hidden	If not hidden

Abstract Methods in Interfaces

An interface defines a *contract* by specifying a set of `abstract` and `default` method declarations, but provides implementations only for the `default` methods—not for the `abstract` methods. The `abstract` methods in an interface are all implicitly `abstract` and `public` by virtue of their definitions. Only the modifiers `abstract` and `public` are allowed, but these are invariably omitted. An `abstract` method declaration has the following simple form in a top-level interface:

Click here to view code image

```
return_type method_name (formal_parameter_list) throws_clause;
```

An `abstract` method declaration is essentially a method header terminated by a semicolon (`;`). Note that an `abstract` method is an *instance method* whose implementation will be provided by a class that implements the interface in which the `abstract` method is declared. The `throws` *clause* is discussed in §7.5, p. 388.

The interface `Playable` shown below declares an `abstract` method `play()`. This method is implicitly declared to be `public` and `abstract`. The interface `Playable` defines a contract: To be `Playable`, an object must implement the method `play()`.

Click here to view code image

```
interface Playable {
  void play();              // Abstract method: no implementation
}
```

An interface that has no direct superinterfaces implicitly includes a `public abstract` method declaration for each `public` instance method from the `java.lang.Object` class (e.g., `equals()`, `toString()`, `hashCode()`). These methods are *not* inherited from the `java.lang.Object` class, as only `abstract` method declarations are included in the interface. Their inclusion allows these methods to be called using an interface reference, and their implementation is always guaranteed at runtime as they are either inherited or overridden by all classes.

In contrast to the syntax of `abstract` methods in top-level interfaces, `abstract` methods in top-level classes must be explicitly specified with the keyword `abstract`, and can have `public`, `protected`, or package accessibility.

Functional interfaces, meaning interfaces with a single `abstract` method, are discussed together with lambda expressions in **§13.1, p. 675**.

The rest of this chapter provides numerous examples of declaring, implementing, and using interfaces.

Implementing Interfaces

A class can implement, wholly or partially, any number of interfaces. A class specifies the interfaces it implements as a comma-separated list of unique interface names in an `implements` clause in the class header. Implementing an interface essentially means that the class must provide implementation for the `abstract` methods declared in the interface. In fact, an `abstract` method declaration is overridden when an implementation is provided by a class. Optionally, the class can also override any `default` methods if necessary.

The criteria for overriding methods in classes also apply when implementing `abstract` and any `default` methods from interfaces. A class must provide a method implementation with the same method signature and (covariant) return type as the declaration in the interface, and it can neither narrow the `public` access of the method nor

specify new exceptions in the method's `throws` clause, as attempting to do so would amount to altering the interface's contract, which is illegal.

Best practice advocates using the `@Override` annotation on the implementations of `abstract` and `default` methods (**Example 5.10**). The compiler checks that such a method satisfies the criteria for overriding another method. This check ensures that the method is not inadvertently overloaded by catching the error at compile time. The annotations also aid in the readability of the code, making obvious which methods are overridden.

It is not enough for a class to provide implementations of methods declared in an interface. The class must also specify the interface name in its `implements` clause in order to reap the benefits of interfaces.

In **Example 5.10**, the class `Stack` at (2) implements the interface `IStack`. It both specifies the interface name using the `implements` clause in its class header at (2) and provides the implementation for the `abstract` methods in the interface at (3) and (4). Changing the `public` access of these methods in the class will result in a compile-time error, as this would narrow their accessibility.

Example 5.10 *Implementing Interfaces*

Click here to view code image

```
// File: StackUser.java
interface IStack {                                                    // (1)
  void   push(Object item);
  Object pop();
}
//_____________________________________________________________________________
class Stack implements IStack {                                       // (2)
  protected Object[] elements;
  protected int      tos;  // top of stack

  public Stack(int capacity) {
    elements = new Object[capacity];
    tos      = -1;
  }

  @Override
  public void push(Object item) { elements[++tos] = item; }           // (3)

  @Override
  public Object pop() {                                               // (4)
    Object objRef = elements[tos];
    elements[tos] = null;
```

```
    tos--;
    return objRef;
  }
  public Object peek() { return elements[tos]; }
}
//_____________________________________________________________________
interface ISafeStack extends IStack {                                  // (5)
  boolean isEmpty();
  boolean isFull();
}
//_____________________________________________________________________
class SafeStack extends Stack implements ISafeStack {                  // (6)

  public SafeStack(int capacity) { super(capacity); }
  @Override public boolean isEmpty() { return tos < 0; }               // (7)
  @Override public boolean isFull()  { return tos >= elements.length-1; } // (8)
}
//_____________________________________________________________________
public class StackUser {

  public static void main(String[] args) {                             // (9)
    SafeStack  safeStackRef  = new SafeStack(10);
    Stack      stackRef      = safeStackRef;
    ISafeStack isafeStackRef = safeStackRef;
    IStack     istackRef     = safeStackRef;
    Object     objRef        = safeStackRef;

    safeStackRef.push("Dollars");                                      // (10)
    stackRef.push("Kroner");
    System.out.println(isafeStackRef.pop());
    System.out.println(istackRef.pop());
    System.out.println(objRef.getClass());
  }
}
```

Output from the program:

```
Kroner
Dollars
class SafeStack
```

A class can choose to implement only some of the `abstract` methods of its interfaces (i.e., give a partial implementation of its interfaces). The class must then be declared as `abstract`. Note that `abstract` methods cannot be declared as `static`, because they comprise the contract fulfilled by the *objects* of the class implementing the interface. Abstract methods are always implemented as instance methods.

The interfaces that a class implements and the classes that it extends (directly or indirectly) are called *supertypes* of the class. Conversely, the class is a *subtype* of its supertypes. A class can now inherit from multiple interfaces. Even so, regardless of how many interfaces a class implements directly or indirectly, it provides just a *single* implementation of any `abstract` method declared in multiple interfaces.

Single implementation of an `abstract` method is illustrated by the following code, where the `Worker` class at (5) provides only one implementation of the `doIt()` method that is declared in both interfaces, at (1) and (2). The class `Worker` fulfills the contract for both interfaces, as the `doIt()` method declarations at (1) and (2) have the same method signature and return type. However, the class `Combined` at (3) declares that it implements the two interfaces, but does not provide any implementation of the `doIt()` method; consequently, it must be declared as `abstract`.

Click here to view code image

```
interface IA { int doIt(); }                        // (1)

interface IB { int doIt(); }                        // (2)

abstract class Combined implements IA, IB { }       // (3)

public class Worker implements IA, IB {             // (4)
  @Override
  public int doIt() { return 0; }                   // (5)
}
```

Multiple declarations of the same `abstract` method in several interfaces implemented by a class do *not* result in multiple inheritance of implementation, as no implementation is inherited for such methods. The `abstract` method `doIt()` has only one implementation in the `Worker` class. Multiple inheritance of type is also *not* a problem, as the class is a subtype of all interfaces it implements.

If the `doIt()` methods in the two previous interfaces at (1) and (2) had the same signatures but different return types, the `Worker` class would not be able to implement both interfaces. This is illustrated by the code below. The `doIt()` methods at (1) and (2) have the same signature, but different return types. The `ChallengedWorker` class provides two implementations of the `doIt()` method at (5) and (6), which results in compile-time errors because a class cannot have two methods with the same signature but different return types. Removing either implementation from the `ChallengedWorker` class will be flagged as a compile-time error because the `ChallengedWorker` class will not be implementing both interfaces. There is no way the `ChallengedWorker` class can implement both interfaces, given the declarations shown in the code. In addition, the

`abstract` class `Combined` at (3) will not compile because it will be inheriting two methods with conflicting `abstract` method declarations. In fact, the compiler complains of duplicate methods.

Click here to view code image

```
interface IA { int doIt(); }                          // (1)

interface IB { double doIt(); }                       // (2)

abstract class Combined implements IA, IB { }         // (3) Compile-time error!

public class ChallengedWorker implements IA, IB {     // (4)
  @Override
  public int doIt() { return 0; }                     // (5) Compile-time error!
  @Override
  public double doIt() {                              // (6) Compile-time error!
    System.out.println("Sorry!");
    return = 0.0;
  }
}
```

An enum type (p. 298) can also implement interfaces. The discussion above on classes implementing interfaces also applies to enum types, but with the following caveat. Since an enum type can never be `abstract`, each `abstract` method of its declared interfaces must be implemented as an instance member either in the enum type or in *all* constant-specific class bodies defined in the enum type— otherwise, the code will not compile.

Extending Interfaces

An interface can extend other interfaces, using the `extends` clause. Unlike when extending classes, an interface can extend several interfaces. The interfaces extended by an interface (directly or indirectly) are called *superinterfaces*. Conversely, the interface is a *subinterface* of its superinterfaces. Since interfaces define new reference types, superinterfaces and subinterfaces are also supertypes and subtypes, respectively.

A subinterface inherits from its superinterfaces all members of those superinterfaces, *except* for the following:

- Any `abstract` or `default` methods that it overrides from its superinterfaces (p. 240 and p. 246)
- Any `static` methods declared in its superinterfaces (p. 251)
- Any `static` constants that it hides from its superinterfaces (p. 254)

- Any `static` member types that it hides from its superinterfaces (§9.2, p. 495)

Barring any conflicts, a subinterface inherits `abstract` and `default` method declarations that are not overridden, as well as constants and `static` member types that it does not hide in its superinterfaces. In addition, `abstract`, `static`, and `default` method declarations can be overloaded, analogous to method overloading in classes. For a detailed discussion of overriding `abstract` methods from multiple superinterfaces, see §11.12, p. 621.

Example 5.10 illustrates the relationships between classes and interfaces. In **Example 5.10**, the interface `ISafeStack` extends the interface `IStack` at (5). The class `SafeStack` both extends the `Stack` class and implements the `ISafeStack` interface at (6). Inheritance hierarchies for classes and interfaces defined in **Example 5.10** are shown in **Figure 5.4**.

In UML, an interface resembles a class. One way to differentiate between them is to use an *«interface»* stereotype, as in **Figure 5.4**. The association between a class and any interface it implements is called a *realization* in UML. Realization is depicted in a similar manner to extending classes, but is indicated by an unbroken inheritance arrow. In **Figure 5.4**, there are two explicit realizations: The class `Stack` implements the `IStack` interface and the class `SafeStack` implements the `ISafeStack` interface.

It is instructive to consider how the class `SafeStack` transitively also implements the `IStack` interface in **Example 5.10**. This is evident from the diamond shape of the inheritance hierarchy in **Figure 5.4**. The class `SafeStack` inherits the implementations of the `push()` and `pop()` methods from its superclass `Stack`, which itself implements the

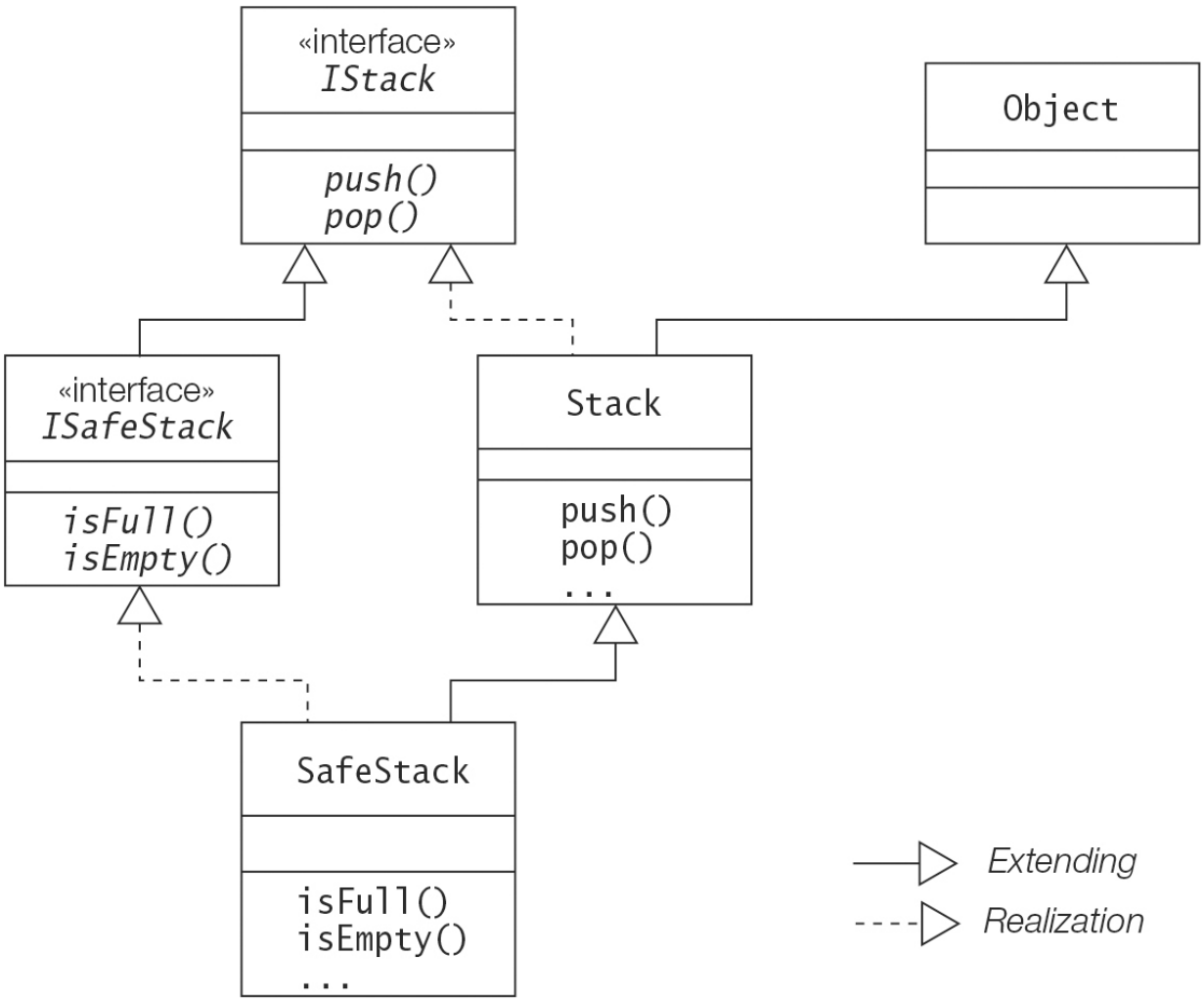

Figure 5.4 *Inheritance Hierarchies for Classes and Interfaces in* Example 5.10

`IStack` interface in which these two methods are declared. The class `SafeStack` also implements the `IStack` interface via the `ISafeStack` interface. The class `SafeStack` provides its own implementation of the `isFull()` and `isEmpty()` methods declared in the `ISafeStack` interface, and has inherited implementations of the `push()` and `pop()` methods whose declarations the `ISafeStack` interface inherits from its superinterface `IStack`. Note that there is only one *implementation* of a method that is inherited by the class `SafeStack`—from its superclass `Stack`.

Thinking in terms of types, every reference type in Java is a subtype of the `java.lang.Object` class. In turn, any interface type is also a subtype of the `Object` class, but it does *not* inherit any implementation from the `Object` class. As mentioned earlier, an interface that has *no direct superinterfaces* implicitly declares a `public abstract` method for *each* `public` instance method in the `Object` class. These `abstract` method declarations are inherited by all subinterfaces of such an interface. Note that this does not mean any *implementation* is inherited by the subinterfaces. The implicit `public abstract` method declarations in an interface allow `public` instance methods in the `Object` class to be invoked on objects referenced by an interface reference. All classes implement these methods, whether they are inherited or overridden from the `Object` class. Any interface can also provide *explicit* `public abstract` method declarations for *non-final* `public` instance methods in the `Object` class.

Click here to view code image

```
interface IStack {                                                    // (1)
  void   push(Object item);
  Object pop();
  @Override boolean equals(Object other);        // public method in Object class.
  @Override String toString();                   // public method in Object class.
//@Override Class getClass(); // Compile-time error! final method in Object class.
}
```

Interface References

Although interfaces cannot be instantiated, references of an interface type can be declared. The reference value of an object can be assigned to references of the object's supertypes. In **Example 5.10**, an object of the class `SafeStack` is created in the `main()` method of the class `StackUser` at (9). The reference value of the object is assigned to references of all the object's supertypes, which are used to manipulate the object. The references are aliases to the same `SafeStack` object, but they can only be used to manipulate this object as an object of the reference type. For example, calling the method `isFull()` on this object using the `stackRef` reference will be flagged as a compile-time error, as the class `Stack` does not provide such a method.

Polymorphic behavior of supertype references is discussed later in this chapter (p. 278).

Default Methods in Interfaces

Only interfaces can define `default` methods. A `default` method is an *instance method* declared with the keyword `default` and whose implementation is provided by the interface. However, a `default` method in a top-level interface always has `public` access, whether the keyword `public` is specified or not.

Click here to view code image

```
default return_type method_name (formal_parameter_list) throws_clause {
    implementaion_of_method_body
}
```

A class implementing an interface can optionally decide to override any `default` method in the interface, as can a subinterface of the interface. If a `default` method is not overridden to provide a new implementation, the `default` implementation provided by the interface is inherited by the class or the subinterface.

No other non-access modifiers, such as `abstract`, `final`, or `static`, are allowed in a `default` method declaration. A `default` method is not `abstract` because it provides an implementation; is not `final` because it can be overridden; and is not `static` because it can be invoked only on instances of a class that implements the interface in which the `default` method is declared.

Note that a `default` method can only be implemented in terms of the values of its local variables (including those passed as parameters) and calls to other methods accessible in the interface, but without reference to any persistent state, as there is none associated with an interface.

The keyword `default` in the context of a `default` method should not be confused with default or package accessibility of a method in a class, which is implied in the absence of any access modifier.

Example 5.11 illustrates the use of `default` methods. The `default` method `printSlogan()` at (1) in the interface `ISlogan` is overridden at (2) in the class `JavaGuru`, and is inherited by the class `JavaGeek` at (3). The output from the program shows that this is the case.

Example 5.11 *Default Methods in Interfaces*

Click here to view code image

```
// File: JavaParty.java
interface ISlogan {
  default void printSlogan() {                        // (1)
    System.out.println("Happiness is getting certified!");
  }
}
//_____________________________________________________________________________
class JavaGuru implements ISlogan {
  @Override
  public void printSlogan() {                         // (2) overrides (1)
    System.out.println("Happiness is catching all the exceptions!");
  }
}
//_____________________________________________________________________________
class JavaGeek implements ISlogan { }                 // (3) inherits (1)
//_____________________________________________________________________________
public class JavaParty {
  public static void main(String[] args) {
    JavaGuru guru = new JavaGuru();
    guru.printSlogan();                               // (4)
    JavaGeek geek = new JavaGeek();
    geek.printSlogan();                               // (5)
```

```
  }
}
```

Output from the program:

Click here to view code image

```
Happiness is catching all the exceptions!
Happiness is getting certified!
```

Overriding Default Methods

Overriding a `default` method from an interface does not necessarily imply that a new implementation is being provided. The `default` method can also be overridden by providing an `abstract` method declaration, as illustrated by the code below. The `default` method `printSlogan()` at (1) in the interface `ISlogan` is overridden by an `abstract` method declaration at (2) and (3) in the interface `INewSlogan` and the `abstract` class `JavaMaster`, respectively. This strategy effectively forces the subtypes of the interface `INewSlogan` and of the `abstract` class `JavaMaster` to provide a new concrete implementation for the method, as one would expect for an `abstract` method.

Click here to view code image

```
interface ISlogan {
  default void printSlogan() {         // (1) Default method.
    System.out.println("Happiness is getting certified!");
  }
}

interface INewSlogan extends ISlogan {
  @Override
  abstract void printSlogan();         // (2) overrides (1) with abstract method.
}

abstract class JavaMaster implements ISlogan {
  @Override
  public abstract void printSlogan();  // (3) overrides (1) with abstract method.
}
```

Conflict Resolution for Default Methods

Conflicts with multiple inheritance of implementation can arise when `default` methods are inherited from unrelated interfaces.

Example 5.12 illustrates the case where two interfaces define a `default` method with the same signature. The `default` method `printSlogan()` is declared at (1) and (2) in the interfaces `ICheapSlogan` and `IFunnySlogan`, respectively. The two method declarations have the same signature. The interface `IAvailableSlogan` at (3) tries to extend the two interfaces `ICheapSlogan` and `IFunnySlogan`. If this were allowed, the interface `IAvailableSlogan` would inherit two implementations of a method with the same signature, which of course is not allowed—so the compiler flags it as an error. By the same token, the compiler flags an error at (4), indicating that the `abstract` class `Wholesaler` cannot inherit two `default` methods with the same signature.

A way out of this dilemma is to override the conflicting `default` methods. The `abstract` class `RetailSeller` that implements the interfaces `ICheapSlogan` and `IFunnySlogan` overrides the conflicting methods by providing an `abstract` method declaration of the `default` method `printSlogan()` at (5). Similarly, the class `NetSeller` that implements the interfaces `ICheapSlogan` and `IFunnySlogan` overrides the conflicting methods by providing an implementation of the `default` method `printSlogan()` at (6).

The upshot of this solution is that clients of the classes `RetailSeller` and `NetSeller` now have to deal with the new declarations of the `printSlogan()` method provided by these classes. One such client is the class `MultipleInheritance` at (10), which calls the method `printSlogan()` on an instance of class `NetSeller` at (11). Not surprisingly, the program output shows that the method in the `NetSeller` class was executed.

What if the class `NetSeller` wanted to invoke the `default` method `printSlogan()` in the interfaces it implements? The overridden `default` method can be called by the overriding subtype (in this case, `NetSeller`) using the keyword `super` in conjunction with the fully qualified name of the interface and the name of the method, as shown at (8) and (9). This syntax works for calling overridden `default` methods in the *direct* superinterface, but not at any higher level in the inheritance hierarchy. The class `NetSeller` can call only `default` methods in its direct superinterfaces `ICheapSlogan` and `IFunnySlogan`. It would not be possible for the class `NetSeller` to call any `default` methods inherited by these superinterfaces, even if they had any.

Example 5.12 *Inheriting Default Method Implementations from Superinterfaces*

Click here to view code image

```
// File: MultipleInheritance.java
interface ICheapSlogan {
  default void printSlogan() {          // (1)
    System.out.println("Override, don't overload.");
  }
```

```
}
//_____________________________________________________________________
interface IFunnySlogan {
  default void printSlogan() {          // (2)
    System.out.println("Catch exceptions, not bugs.");
  }
}
//_____________________________________________________________________
interface IAvailableSlogan              // (3) Compile-time error.
          extends ICheapSlogan, IFunnySlogan { }
//_____________________________________________________________________
abstract class Wholesaler               // (4) Compile-time error.
               implements ICheapSlogan, IFunnySlogan { }
//_____________________________________________________________________
abstract class RetailSeller implements ICheapSlogan, IFunnySlogan {
  @Override                             // Abstract method.
  public abstract void printSlogan();   // (5) overrides (1) and (2).
}
//_____________________________________________________________________
class NetSeller implements ICheapSlogan, IFunnySlogan {
  @Override                             // Concrete method.
  public void printSlogan() {           // (6) overrides (1) and (2).
    System.out.println("Think outside of the class.");
  }

  public void invokeDirect() {          // (7)
    ICheapSlogan.super.printSlogan();   // (8) calls ICheapSlogan.printSlogan()
    IFunnySlogan.super.printSlogan();   // (9) calls IFunnySlogan.printSlogan()
  }
}
//_____________________________________________________________________
public class MultipleInheritance {      // (10)
  public static void main(String[] args) {
    NetSeller seller = new NetSeller();
    seller.printSlogan();               // (11)
    seller.invokeDirect();
  }
}
```

Output from the program:

Click here to view code image

```
Think outside of the class.
Override, don't overload.
Catch exceptions, not bugs.
```

Example 5.13 illustrates the case where a concrete method at (1) in the class `Slogan` and a `default` method at (2) in the interface `ISlogan` have the same signature. The subclass `MySlogan` at (3) extends the superclass `Slogan` and implements the super-interface `ISlogan`. The class `MySlogan` compiles even though it looks like two implementations of the `printSlogan()` method are being inherited. In this special case there is no multiple inheritance of implementation, as only the implementation from the superclass `Slogan` is inherited. The implementation of the `default` method at (2) is ignored. This is borne out by the program output when the method `print-Slogan()` is called at (5) on an object of class `MySlogan`.

Interface Evolution

Augmenting an existing interface with an `abstract` method will break all classes that implement this interface, as they will no longer implement the old interface. These classes will need to be modified and recompiled in order to work with the augmented interface.

Augmenting an existing interface with a `default` method does not pose this problem. Classes that implement the old interface will work with the augmented interface *without modifying or recompiling* them. Thus interfaces can evolve without affecting classes that worked with the old interface. This is also true for `static` methods (p. 251) added to existing interfaces.

Example 5.13 *Inheriting Method Implementations from Supertypes*

Click here to view code image

```
// File: MultipleInheritance2.java
class Slogan {
  public void printSlogan() {                          // (1) Concrete method
    System.out.println("Superclass wins!");
  }
}
//_____________________________________________________________________________
interface ISlogan {
  default void printSlogan() {                         // (2) Default method
    System.out.println("Superinterface wins!");
  }
}
//_____________________________________________________________________________
class MySlogan extends Slogan implements ISlogan { } // (3)
//_____________________________________________________________________________
public class MultipleInheritance2 {                    // (4)
  public static void main(String[] args) {
    MySlogan slogan = new MySlogan();
```

```
        slogan.printSlogan();                          // (5)
    }
}
```

Output from the program:

```
Superclass wins!
```

Static Methods in Interfaces

A common practice in designing APIs has been to provide an interface that classes can implement and a separate *utility class* providing `static` methods for common operations on objects of these classes. Typical examples are the `java.util.Collection` interface and the `java.util.Collections` utility class (**Chapter 15**, **p. 781**). Another example is the `Path` interface and the `Paths` utility class in the `java.nio.file` package (**Chapter 21**, **p. 1285**). However, now an interface can also declare `static` methods, and there is no need for a separate utility class.

Static method declarations in a top-level interface are declared analogous to `static` method declarations in a class. However, a `static` method in a top-level interface always has `public` access, whether the keyword `public` is specified or not. As with `static` methods in a class, the keyword `static` is mandatory; otherwise, the code will not compile. Without the keyword `static`, the method declaration is identical to that of an instance method, but such instance methods cannot be declared in an interface and the compiler will flag an error.

Click here to view code image

```
static return_type method_name (formal_parameter_list) throws_clause {
    implementaion_of_method_body
}
```

Static methods in an interface differ from those in a class in one important respect: Static methods in an interface *cannot* be inherited, unlike `static` methods in classes. This essentially means that such methods cannot be invoked directly by calling the method in subinterfaces or in classes that extend or implement interfaces containing such methods, respectively. A `static` method can be invoked only by using its qualified name—that is, the name of the interface in which it is declared, together with its simple name, using the dot notation (`.`).

Example 5.14 illustrates the use of `static` methods in interfaces. The `static` method `getNumOfCylinders()` at (1) is declared in the `IMaxEngineSize` interface. There are

two implementations of the method `getEngineSize()`, at (2) and (3), in the interface `IMaxEngineSize` and its subinterface `INewEngineSize`, respectively. The class `CarRace` implements the subinterface `INewEngineSize`.

It is not possible to invoke the `static` method `getNumOfCylinders()` directly, as shown at (4). It is also not possible to invoke directly the `static` method `getEngine-Size()` from either interface, as shown at (6). The respective implementations of the `static` methods can be invoked only by using their qualified names, as shown at (5), (7), and (8). It does not matter that a `static` method is redeclared in a subinterface; the `static` method is not inherited. Each `static` method declaration in **Example 5.14** is a new method.

Example 5.14 *Static Methods in Interfaces*

Click here to view code image

```
// File: CarRace.java
import static java.lang.System.out;
interface IMaxEngineSize {
  static int getNumOfCylinders() { return 6; }        // (1) Static method
  static double getEngineSize() { return 1.6; }       // (2) Static method
}
//_____________________________________________________________________________
interface INewEngineSize extends IMaxEngineSize {
  static double getEngineSize() { return 2.4; }       // (3) Static method
}
//_____________________________________________________________________________
public class CarRace implements INewEngineSize {
  public static void main(String[] args) {
//  out.println("No. of cylinders: " +
//              getNumOfCylinders());                  // (4) Compile-time error.
    out.println("No. of cylinders: " +
        IMaxEngineSize.getNumOfCylinders());         // (5)
//  out.println("Engine size: " + getEngineSize());   // (6) Compile-time error.
    out.println("Max engine size: " + IMaxEngineSize.getEngineSize()); // (7)
    out.println("New engine size: " + INewEngineSize.getEngineSize()); // (8)
  }
}
```

Output from the program:

```
No. of cylinders: 6
Max engine size: 1.6
New engine size: 2.4
```

Private Methods in Interfaces

Private methods in interfaces are no different from `private` methods in classes. As such, they can only be accessed inside the interface, acting as *helper* or *auxiliary methods* for non-`abstract` methods declared in the interface. They allow code to be shared between the non-`abstract` methods in the interface, thus promoting code reuse and avoiding code duplicity.

Private methods in an interface are governed by the following rules:

- Not surprisingly, a `private` method must be declared with the `private` modifier in its header.
- A `private` method can only be accessed inside the interface itself.
- A `private` method cannot be declared with the `abstract` or the `default` modifier, but can be declared either as a `private` instance method or a `private static` method.
- A `private static` method can be invoked in any non-`abstract` method in the interface.
- A `private` instance method can only be invoked in `default` and `private` instance methods of the interface, and not in any `static` methods of the interface.

The above rules stem from accessibility of a `private` member in a type declaration in which the code is accessible in static and non-static contexts.

Example 5.15 illustrates declaring `private` methods in an interface. The `default` method `getSalePrice()` at (2) and the `static` method `startSale()` at (5) call the `private` instance method `wrapUp()` at (4) and the `private static` method `showSaleItems()` at (7), respectively.

Example 5.15 *Private Methods in Interfaces*

Click here to view code image

```
/** Interface with private methods. */
public interface IShopper {

  // Abstract method:
  double getItemPrice();                                  // (1)

  // Default method:
  default double getSalePrice(double price) {             // (2)
    var salePrice = (80.0/100.0)*price;
    System.out.println("Default method: " + "Sale price is " + salePrice);
    wrapUp();                    // (3) Calls the private instance method at (4)
    return salePrice;
```

```
  }

  // Private instance method:
  private void wrapUp() {                                  // 4)
    System.out.println("Private method: " + "Wrapping up!");
  }

  // Static method:
  static void startSale() {                                // (5)
    System.out.println("Static method: " + "Amazing savings!");
    showSaleItems();             // (6) Calls the private static method at (7)
  }

  // Private static method:
  private static void showSaleItems() {                    // (7)
    System.out.println("Private static method: " + "Sorry. No items on sale!");
  }
}
```

```
/** Class Shopper implements IShopper interface */
public class Shopper implements IShopper {

  @Override
  public double getItemPrice() {
    return 100.00;
  }

  public static void main(String[] args) {
    Shopper customer = new Shopper();
    var price = customer.getItemPrice();
    System.out.println("Item price: " + price);
    var salePrice = customer.getSalePrice(price); // Calls default method at (2)
    System.out.println();
    IShopper.startSale();                         // Calls static method at (5)
  }
}
```

Output from the program:

Click here to view code image

```
Item price: 100.0
Default method: Sale price is 80.0
Private method: Wrapping up!
```

```
Static method: Amazing savings!
Private static method: Sorry. No items on sale!
```

Constants in Interfaces

A field declaration in an interface defines a *named constant*. Naming conventions recommend using uppercase letters, with multiple words in the name being separated by underscores. Such constants are considered to be `public`, `static`, and `final`. These modifiers are usually omitted from the declaration, but can be specified in any order. Such a constant must be initialized with an initializer expression.

An interface constant can be accessed by any client (a class or interface) using its qualified name, regardless of whether the client extends or implements its interface. However, if the client is a class that implements this interface or is an interface that extends this interface, then the client can also access such constants directly by their simple names. Such a client inherits the interface constants. Typical usage of constants in interfaces is illustrated in **Example 5.16**, showing access both by the constant's simple name and its qualified name in the print statements at (1) and (2), respectively.

Example 5.16 *Constants in Interfaces*

Click here to view code image

```
// File: Client.java
interface Constants {
  double PI_APPROXIMATION = 3.14;
  String AREA_UNITS       = "sq.cm.";
  String LENGTH_UNITS     = "cm.";
}
//_____________________________________________________________________________
public class Client implements Constants {
  public static void main(String[] args) {
    double radius = 1.5;

    // (1) Using simple name:
    System.out.printf("Area of circle is %.2f %s%n",
               PI_APPROXIMATION * radius*radius, AREA_UNITS);

    // (2) Using qualified name:
    System.out.printf("Circumference of circle is %.2f %s%n",
             2.0 * Constants.PI_APPROXIMATION * radius, Constants.LENGTH_UNITS);
  }
}
```

Output from the program:

Click here to view code image

```
Area of circle is 7.06 sq.cm.
Circumference of circle is 9.42 cm.
```

Extending an interface that has constants is analogous to extending a class that has `static` variables. This is illustrated in **Figure 5.5** and **Example 5.17**. Note the diamond shape of the inheritance hierarchy, indicating the presence of multiple inheritance paths through which constants can be inherited. The constants `IDLE` and `BUSY` at (1) and (2) in the interface `IBaseStates` are inherited by the subinterface `IAllStates` via both the interface `IExtStatesA` and the interface `IExtStatesB`. In such cases, the constant is considered to be inherited only once and can be accessed by its simple name, as shown at (12) in **Example 5.17**.

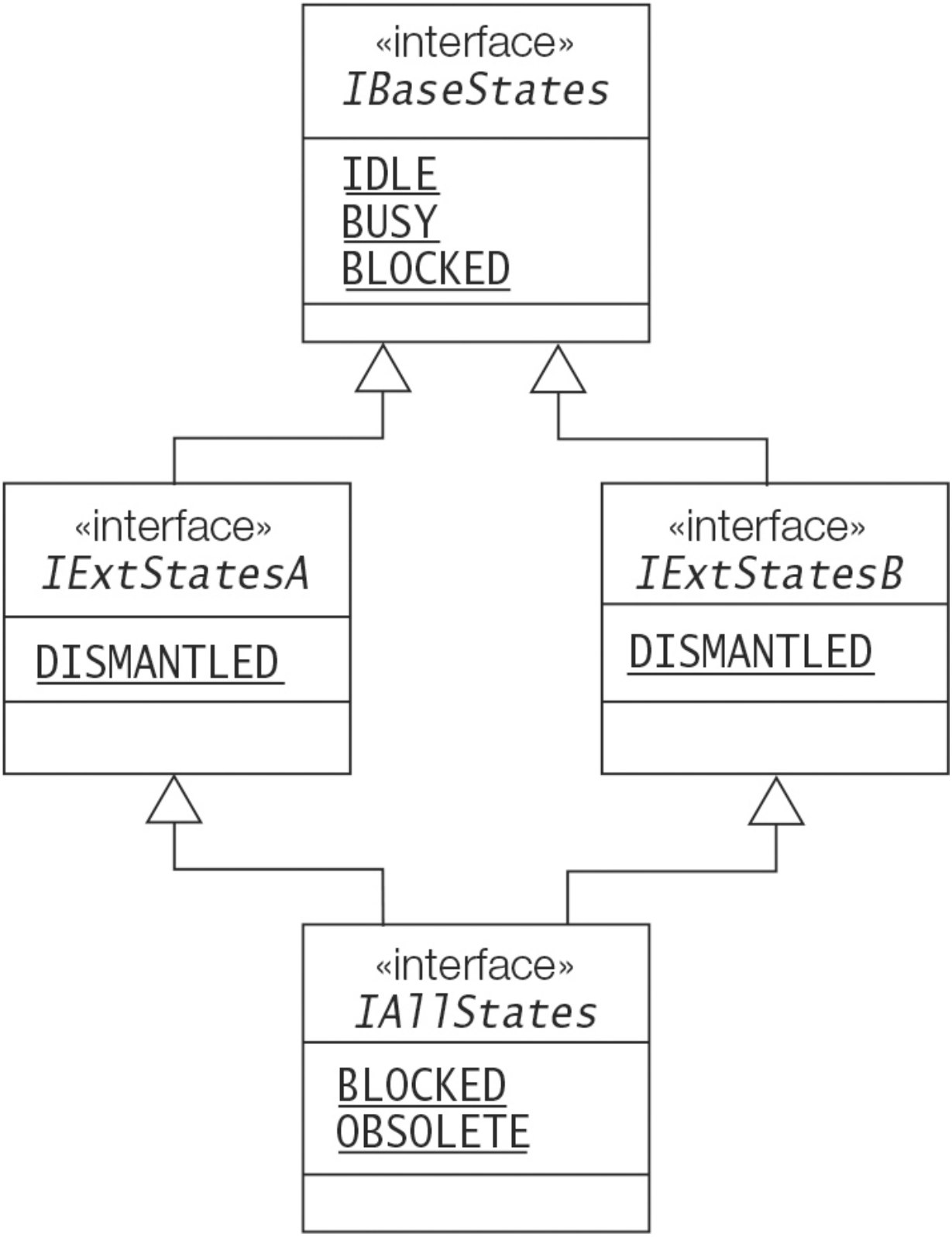

Figure 5.5 *Inheritance Relationships for Interface Constants*

Constants can be *hidden* by the subinterfaces. The declaration of the constant `BLOCKED` at (6) in the interface `IAllStates` hides the declaration of the constant at (2) in the interface `IBaseStates`. The new declaration can be accessed by its simple name in a class implementing the interface `IAllStates`, as shown at (10) in **Example 5.17**. The hidden constant declaration can always be accessed by using its qualified name as shown at (11) in **Example 5.17**.

In the case of multiple inheritance of interface constants, any name conflicts can be resolved by using the qualified name to access the constants. This is illustrated by the

constant `DISMANTLED`, which is declared in both the `IExtStatesA` and `IExtStatesB` interfaces. Both declarations are inherited by the subinterface `IAllStates`. Such declarations are said to be *ambiguous*. The compiler will report an error only if such constants are accessed by their simple names, as shown at (7) and (8) for the constant `DISMANTLED`. Only the qualified name can be used to disambiguate such constants and resolve the conflict, as shown at (7a) and (8a) for the constant `DISMANTLED.`

When defining a *set of related constants*, the recommended practice is to use an enum type, rather than named constants in an interface.

Example 5.17 *Inheriting Constants in Interfaces*

Click here to view code image

```
// File: Factory.java
import static java.lang.System.out;

interface IBaseStates {
  String IDLE = "idle";                                  // (1)
  String BUSY = "busy";                                  // (2)
  String BLOCKED = "blocked";                            // (3)
}
//_____________________________________________________________________________
interface IExtStatesA extends IBaseStates {
  String DISMANTLED = "dismantled";                      // (4)
}
//_____________________________________________________________________________
interface IExtStatesB extends IBaseStates {
  String DISMANTLED = "kaput";                           // (5)
}
//_____________________________________________________________________________
interface IAllStates extends IExtStatesB, IExtStatesA {
  String BLOCKED = "out of order";                       // (6) hides (3)
//String OBSOLETE = BLOCKED + ", " +
//                  DISMANTLED + " and scrapped.";       // (7) Ambiguous
  String OBSOLETE = BLOCKED + ", " +
         IExtStatesB.DISMANTLED + " and scrapped";       // (7a)
}
//_____________________________________________________________________________
public class Factory implements IAllStates {
  public static void main(String[] args) {
//  out.println("Machine A is " + DISMANTLED);              // (8) Ambiguous.
    out.println("Machine A is " + IExtStatesB.DISMANTLED); // (8a)
    out.println("Machine B is " + OBSOLETE);            // (9) IAllStates.OBSOLETE
    out.println("Machine C is " + BLOCKED);             // (10) IAllStates.BLOCKED
    out.println("Machine D is " + IBaseStates.BLOCKED);// (11)
    out.println("Machine E is " + BUSY);                // (12) Simple name
```

```
  }
}
```

Output from the program:

Click here to view code image

```
Machine A is kaput
Machine B is out of order, kaput and scrapped
Machine C is out of order
Machine D is blocked
Machine E is busy
```

Review Questions

5.18 Which modifiers are not allowed for methods in an interface ? Select the two correct answers.

a. `public`

b. `protected`

c. `private`

d. `default`

e. `abstract`

f. `static`

g. `final`

5.19 Which method call can be inserted at both (1) and (2) so that the following code will still compile?

Click here to view code image

```
interface INewSlogan {
  String SLOGAN = "Trouble shared is trouble halved!";
  static void printSlogan() { System.out.println(SLOGAN); }
}
//_____________________________________________________________________
public class Firm implements INewSlogan {
```

```
  public static void main(String[] args) {
    Firm co = new Firm();
    INewSlogan sl = co;
    // (1) INSERT STATEMENT EXPRESSION HERE
  }

  void testSlogan() {
    Firm co = new Firm();
    INewSlogan sl = co;
    // (2) INSERT STATEMENT EXPRESSION HERE
  }
}
```

Select the one correct answer.

a. `printSlogan();`

b. `co.printSlogan();`

c. `sl.printSlogan();`

d. `Firm.printSlogan();`

e. `INewSlogan.printSlogan();`

5.20 What will be the result of compiling and running the following program?

Click here to view code image

```
interface IJogger {
  default boolean justDoIt(String msg) { return false; }  // (1)
  static  boolean justDoIt(int i)      { return true; }   // (2)
}

class Athlete implements IJogger {
  public boolean justDoIt(String msg)  { return true; }   // (3)
  public boolean justDoIt(int i)       { return false; }  // (4)
}

public class RaceA {
  public static void main(String[] args) {
    Athlete athlete = new Athlete();
    IJogger jogger = athlete;
    System.out.print(jogger.justDoIt("Run"));             // (5)
    System.out.println("|" + athlete.justDoIt(10));      // (6)
```

```
  }
}
```

Select the one correct answer.

a. The program will fail to compile.

b. `true|true`

c. `true|false`

d. `false|true`

e. `false|false`

5.21 Which statement is true about the following code?

Click here to view code image

```
abstract class MyClass implements Interface1, Interface2 {
  public void f() { }
  public void g() { }
}

interface Interface1 {
  int VAL_A = 1;
  int VAL_B = 2;

  void f();
  void g();
}

interface Interface2 {
  int VAL_B = 3;
  int VAL_C = 4;

  void g();
  void h();
}
```

Select the one correct answer.

a. `MyClass` implements only `Interface1`; the implementation for `void h()` from `Interface2` is missing.

b. The declarations of `void g()` in the two interfaces are in conflict, so the code will fail to compile.

c. The declarations of `int VAL_B` in the two interfaces are in conflict, so the code will fail to compile.

d. Nothing is wrong with the code; it will compile without errors.

5.7 Arrays and Subtyping

Table 5.4 summarizes the types found in Java. Only primitive data and reference values can be stored in variables. Only class and array types can be explicitly instantiated to create objects.

Table 5.4 *Types and Values*

Types	Values
Primitive data types	Primitive data values
Class, interface, enum, and array types (*reference types*)	Reference values

Arrays and Subtype Covariance

Arrays are objects in Java. Array types (`boolean[]`, `Object[]`, `Stack[]`) implicitly augment the inheritance hierarchy. The inheritance hierarchy depicted in Figure 5.4, for example, can be augmented by the corresponding array types to produce the *type hierarchy* shown in Figure 5.6. An array type is shown as a "class" with the `[]` notation appended to the name of the element type. The class `SafeStack` is a subclass of the class `Stack`. The corresponding array types, `SafeStack[]` and `Stack[]`, are shown as the subtype and the supertype, respectively, in the type hierarchy. Figure 5.6 also shows array types corresponding to some of the primitive data types.

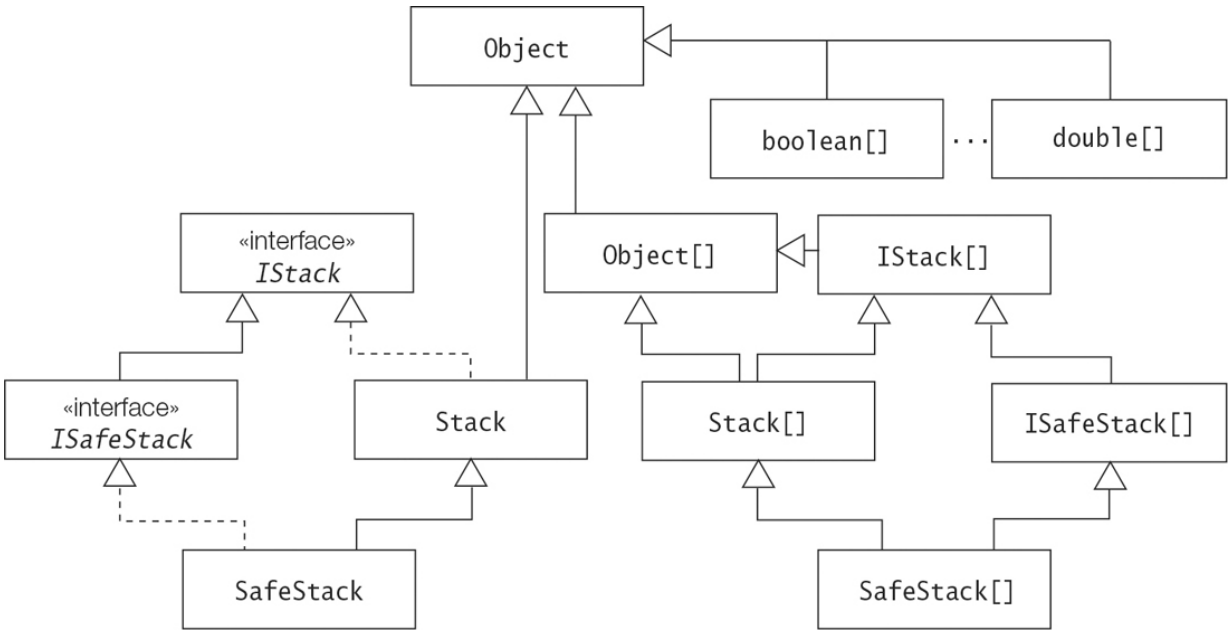

Figure 5.6 *Reference Type Hierarchy: Arrays and Subtype Covariance*

From the type hierarchy in **Figure 5.6**, the following facts are apparent:

- *All* reference types are subtypes of the `Object` type. This applies to classes, interfaces, enums, and array types, as these are all reference types.
- All arrays of reference types are also subtypes of the array type `Object[]`, but arrays of primitive data types are not. Note that the array type `Object[]` is also a subtype of the `Object` type.
- If a non-generic reference type is a subtype of another non-generic reference type, the corresponding array types also have an analogous subtype–supertype relationship. This is called the *subtype covariance relationship*.
- There is no subtype–supertype relationship between a type and its corresponding array type.

We can create an array of an interface type, but we cannot instantiate an interface (as is the case with `abstract` classes). In the following declaration statement, the reference `arrayOfISafeStack` has type `ISafeStack[]` (i.e., an array of the interface type `ISafeStack`):

Click here to view code image

```
ISafeStack[] arrayOfISafeStack = new ISafeStack[5];
```

The array creation expression creates an array whose element type is `ISafeStack`. The array object can accommodate five references of the type `ISafeStack`. The declaration statement does not initialize these references to refer to any objects; instead, they are initialized to the default value `null`.

Array Store Check

An array reference exhibits polymorphic behavior like any other reference, subject to its location in the type hierarchy (p. 278). However, a runtime check is necessary when objects are inserted in an array, as illustrated below.

The following assignment is valid, as a supertype reference (`Stack[]`) can refer to objects of its subtype (`SafeStack[]`):

Click here to view code image

```
Stack[] arrayOfStack = new SafeStack[2];      // (1)
```

Since `Stack` is a supertype of `SafeStack`, the following assignment is also valid:

Click here to view code image

```
arrayOfStack[0] = new SafeStack(10);          // (2)
```

The assignment at (2) assigns the reference value of a new `SafeStack` object to the reference at index `0` in the `SafeStack[]` object created at (1).

Since the type of `arrayOfStack[i]`, (`0 £ i < 2`), is `Stack`, it should be possible to make the following assignment as well:

Click here to view code image

```
arrayOfStack[1] = new Stack(20);              // (3) ArrayStoreException
```

At compile time there are no problems, as the compiler cannot deduce that the array variable `arrayOfStack` will actually denote a `SafeStack[]` object at runtime. However, the assignment at (3) results in an `ArrayStoreException` being thrown at runtime because an array of `SafeStack` objects cannot possibly contain objects of its supertype `Stack`.

The array store check at runtime ensures that an object being stored in the array is assignment compatible (p. 264) with the element type of the array. To make the array store check feasible at runtime, the array retains information about its declared element type at runtime.

5.8 Reference Values and Conversions

A review of conversions (§2.3, p. 43) is recommended before proceeding with this section.

Reference values, like primitive values, can be assigned, cast, and passed as arguments. Conversions can occur in the following contexts:

- Assignment
- Method invocation
- Casting

The rule of thumb for the primitive data types is that widening conversions are permitted, but narrowing conversions require an explicit cast. The rule of thumb for reference values is that widening conversions up the type hierarchy are permitted, but narrowing conversions down the hierarchy require an explicit cast. In other words, conversions that are from a subtype to its supertypes are allowed, but other conversions require an explicit cast or are otherwise illegal. There is no notion of promotion for reference values.

5.9 Reference Value Assignment Conversions

In the context of assignments, the following conversions are permitted (Table 2.17, p. 47):

- Widening primitive and reference conversions (`long` ← `int`, `Object` ← `String`)
- Boxing conversion of primitive values, followed by optional widening reference conversion (`Integer` ← `int`, `Number` ← `Integer` ← `int`)
- Unboxing conversion of a primitive value wrapper object, followed by optional widening primitive conversion (`long` ← `int` ← `Integer`)

In addition, for assignment conversions only, the following conversion is also possible:

- Narrowing conversion for constant expressions of non-`long` integer types, with optional boxing (`Byte` ← `byte` ← `int`)

Note that these rules imply that a widening conversion *cannot* be followed by any boxing conversion, but the converse is permitted.

Widening reference conversions typically occur during assignment *up* the type hierarchy, with implicit conversion of the source reference value to that of the destination reference type:

Click here to view code image

```
Object obj = "Up the tree";    // Widening reference conversion: Object <-- String
String str1 = obj;       // Not OK. Narrowing reference conversion requires a cast.
String str2 = Integer.valueOf(10); // Illegal. No relation between
                                   //          String and Integer.
```

The source value can be a primitive value, in which case the value is boxed in a wrapper object corresponding to the primitive type. If the destination reference type is a supertype of the wrapper type, a widening reference conversion can occur:

Click here to view code image

```
Integer iRef = 10;  // Only boxing
Number num = 10L;   // Boxing, followed by widening: Number <--- Long <--- long
Object obj = 100;   // Boxing, followed by widening: Object <--- Integer <--- int
```

More examples of boxing during assignment can be found in §2.3, p. 45.

Example 5.18 *Assigning and Passing Reference Values*

Click here to view code image

```
// See Example 5.10, p. 241, for type declarations.
interface IStack
interface ISafeStack extends IStack
class Stack implements IStack
class SafeStack extends Stack implements ISafeStack
```

Click here to view code image

```
public class ReferenceConversion {

  public static void main(String[] args) {
    // Reference declarations:
    Object     objRef;
    Stack      stackRef;
    SafeStack  safeStackRef;
    IStack     iStackRef;
    ISafeStack iSafeStackRef;

    // SourceType is a class type:
    safeStackRef  = new SafeStack(10);
    objRef        = safeStackRef;    // (1) Always possible
    stackRef      = safeStackRef;    // (2) Subclass to superclass assignment
    iStackRef     = stackRef;        // (3) Stack implements IStack
    iSafeStackRef = safeStackRef;    // (4) SafeStack implements ISafeStack
```

```
  // SourceType is an interface type:
  objRef    = iStackRef;           // (5) Always possible
  iStackRef = iSafeStackRef;       // (6) Sub- to superinterface assignment

  // SourceType is an array type:
  Object[]     objArray          = new Object[3];
  Stack[]      arrayOfStack      = new Stack[3];
  SafeStack[]  arrayOfSafeStack  = new SafeStack[5];
  ISafeStack[] arrayOfISafeStack = new ISafeStack[5];
  int[]        intArray          = new int[10];

  // Reference value assignments:
  objRef     = objArray;           // (7) Always possible
  objRef     = arrayOfStack;       // (8) Always possible
  objArray   = arrayOfStack;       // (9) Always possible
  objArray   = arrayOfISafeStack;  // (10) Always possible
  objRef     = intArray;           // (11) Always possible
  //  objArray   = intArray;       // (12) Compile-time error:
                                   //      int[] not subtype of Object[]
  arrayOfStack = arrayOfSafeStack; // (13) Subclass array to superclass array
  arrayOfISafeStack = arrayOfSafeStack; // (14) SafeStack implements
                                        //      ISafeStack

  // Method invocation conversions:
  System.out.println("First call:");
  sendParams(stackRef, safeStackRef, iStackRef,
             arrayOfSafeStack, arrayOfISafeStack);                  // (15)
  //  Call Signature: sendParams(Stack, SafeStack, IStack,
  //                             SafeStack[], ISafeStack[]);

  System.out.println("Second call:");
  sendParams(arrayOfISafeStack, stackRef, iSafeStackRef,
             arrayOfStack, arrayOfSafeStack);                       // (16)
  //  Call Signature: sendParams(ISafeStack[], Stack, ISafeStack,
  //                             Stack[], SafeStack[]);
}

public static void sendParams(Object objRefParam, Stack stackRefParam,
    IStack iStackRefParam, Stack[] arrayOfStackParam,
    IStack[] arrayOfIStackParam) {                                   // (17)
  //  Signature: sendParams(Object, Stack, IStack, Stack[], IStack[])
  //  Print class name of object denoted by the reference at runtime.
  System.out.println(objRefParam.getClass());
  System.out.println(stackRefParam.getClass());
  System.out.println(iStackRefParam.getClass());
  System.out.println(arrayOfStackParam.getClass());
  System.out.println(arrayOfIStackParam.getClass());
```

```
  }
}
```

Output from the program:

Click here to view code image

```
First call:
class SafeStack
class SafeStack
class SafeStack
class [LSafeStack;
class [LSafeStack;
Second call:
class [LSafeStack;
class SafeStack
class SafeStack
class [LSafeStack;
class [LSafeStack;
```

The rules for reference value assignment are stated in this section, based on the following code:

Click here to view code image

```
SourceType srcRef;
// srcRef is appropriately initialized.
DestinationType destRef = srcRef;
```

If an assignment is legal, the reference value of `srcRef` is said to be *assignable* (or *assignment compatible*) to the reference of `DestinationType`. The rules are illustrated by concrete cases from **Example 5.18**. Note that the code in **Example 5.18** uses reference types from **Example 5.10**, **p. 241**.

- If the `SourceType` is a *class type*, the reference value in `srcRef` may be assigned to the `destRef` reference, provided the `DestinationType` is one of the following:
  - `DestinationType` is a superclass of the subclass `SourceType`.
  - `DestinationType` is an interface type that is implemented by the class `SourceType`.

  Click here to view code image

```
objRef        = safeStackRef;    // (1) Always possible
stackRef      = safeStackRef;    // (2) Subclass to superclass assignment
```

```
iStackRef     = stackRef;        // (3) Stack implements IStack
iSafeStackRef = safeStackRef;    // (4) SafeStack implements ISafeStack
```

- If the `SourceType` is an *interface type*, the reference value in `srcRef` may be assigned to the `destRef` reference, provided the `DestinationType` is one of the following:
  - `DestinationType` is the `Object` class.
  - `DestinationType` is a superinterface of the subinterface `SourceType`.

  Click here to view code image

```
objRef    = iStackRef;     // (5) Always possible
iStackRef = iSafeStackRef; // (6) Subinterface to superinterface assignment
```

- If the `SourceType` is an *array type*, the reference value in `srcRef` may be assigned to the `destRef` reference, provided the `DestinationType` is one of the following:
  - `DestinationType` is the `Object` class.
  - `DestinationType` is an array type, where the element type of the `SourceType` is assignable to the element type of the `DestinationType`.

  Click here to view code image

```
objRef     = objArray;           // (7) Always possible
objRef     = arrayOfStack;       // (8) Always possible
objArray   = arrayOfStack;       // (9) Always possible
objArray   = arrayOfISafeStack;  // (10) Always possible
objRef     = intArray;           // (11) Always possible
// objArray   = intArray;        // (12) Compile-time error:
                                 //      int[] not subtype of Object[]
arrayOfStack = arrayOfSafeStack; // (13) Subclass array to superclass array
arrayOfISafeStack = arrayOfSafeStack; // (14) SafeStack implements
                                      //      ISafeStack
```

The rules for assignment are enforced at compile time, guaranteeing that no type conversion error will occur during assignment at runtime. Such conversions are *type-safe*. The reason the rules can be enforced at compile time is that they concern the *declared type* of the reference (which is always known at compile time) rather than the actual type of the object being referenced (which is known at runtime).

5.10 Method Invocation Conversions Involving References

The conversions for reference value assignment are also applicable to *method invocation conversions*, except for the narrowing conversion for constant expressions of non-`long` integer type (**Table 2.17**, **p. 47**). This is reasonable, as parameters in Java are passed by value (**§3.10**, **p. 127**), requiring that values of the actual parameters must be assignable to formal parameters of the compatible types.

In **Example 5.18**, the method `sendParams()` at (17) has the following signature, showing the types of the formal parameters:

Click here to view code image

```
sendParams(Object, Stack, IStack, Stack[], IStack[])
```

The method call at (15) has the following signature, showing the types of the actual parameters:

Click here to view code image

```
sendParams(Stack, SafeStack, IStack, SafeStack[], ISafeStack[]);   // Compile time
```

At runtime, the actual type of the objects whose reference values are passed at (15) is the following:

Click here to view code image

```
sendParams(SafeStack, SafeStack, SafeStack, SafeStack[], SafeStack[]);  // Runtime
```

Note that the assignment of the values of the actual parameters to the corresponding formal parameters is legal, according to the rules for assignment discussed earlier. The method call at (16) provides another example of the parameter-passing conversion. It has the following signature:

Click here to view code image

```
sendParams(ISafeStack[], Stack, ISafeStack, Stack[], SafeStack[]); // Compile time
```

At runtime, the actual type of the objects whose reference values are passed at (16) is the following:

Click here to view code image

```
sendParams(SafeStack[], SafeStack, SafeStack, SafeStack[], SafeStack[]);// Runtime
```

Analogous to assignment, the rules for parameter-passing conversions are based on the reference type of the parameters and are enforced at compile time. It is instructive to compare the type of a formal parameter with the type of its corresponding actual parameter at compile time, and the type of the object whose reference value is actually passed at runtime. The output in **Example 5.18** shows the class of the actual objects

referenced by the formal parameters at runtime, which in this case turns out to be either `SafeStack` or `SafeStack[]`. The characters `[L` in the output indicate a one-dimensional array of a class or interface type (see the `Class.getName()` method in the Java SE Platform API documentation).

Overloaded Method Resolution

In this subsection, we take a look at some aspects regarding *overloaded method resolution*—that is, how the compiler determines which overloaded method will be invoked by a given method call at runtime.

Resolution of overloaded methods selects the *most specific* method for execution. One method is considered more specific than another method if all actual parameters that can be accepted by the one method can be accepted by the other method. If more than one such method is present, the call is described as *ambiguous*. The following overloaded methods illustrate this situation:

Click here to view code image

```
private static void flipFlop(String str, int i, Integer iRef) { // (1)
    out.println(str + " ==> (String, int, Integer)");
}
private static void flipFlop(String str, int i, int j) {        // (2)
    out.println(str + " ==> (String, int, int)");
}
```

Their method signatures are as follows:

Click here to view code image

```
flipFlop(String, int, Integer)                                  // See (1)
flipFlop(String, int, int)                                      // See (2)
```

The following method call is ambiguous:

Click here to view code image

```
flipFlop("(String, Integer, int)",
         Integer.valueOf(4), 2020);                             // (3) Ambiguous call
```

It has the call signature:

Click here to view code image

```
flipFlop(String, Integer, int)                          // See (3)
```

The method at (1) can be called with the second argument unboxed and the third argument boxed, as can the method at (2) with only the second argument unboxed. In other words, for the call at (3), none of the methods is more specific than the other.

Example 5.19 illustrates a simple case of how method resolution is done to choose the most specific overloaded method. The method `testIfOn()` is overloaded at (1) and (2) in the class `Overload`. The call `client.testIfOn(tubeLight)` at (3) *satisfies* the parameter lists in both implementations given at (1) and (2), as the reference `tube-Light` can also be assigned to a reference of its superclass `Light`. The *most specific* method, (2), is chosen, resulting in `false` being written on the terminal. The call `client.testIfOn(light)` at (4) satisfies only the parameter list in the implementation given at (1), resulting in `true` being written on the terminal. This is also the case at (5). The object referred to by the argument in the call at runtime is irrelevant; rather, it is the *type* of the argument that is important for overloaded method resolution.

Example 5.19 *Choosing the Most Specific Method (Simple Case)*

Click here to view code image

```
class Light { /* ... */ }

class TubeLight extends Light { /* ... */ }

public class Overload {
  boolean testIfOn(Light aLight)         { return true; }    // (1)
  boolean testIfOn(TubeLight aTubeLight) { return false; }   // (2)

  public static void main(String[] args) {

    TubeLight tubeLight = new TubeLight();
    Light     light     = new Light();
    Light     light2    = new TubeLight();

    Overload client = new Overload();
    System.out.println(client.testIfOn(tubeLight)); // (3) ==> method at (2)
    System.out.println(client.testIfOn(light));     // (4) ==> method at (1)
    System.out.println(client.testIfOn(light2));    // (5) ==> method at (2)
  }
}
```

Output from the program:

```
false
true
true
```

The algorithm used by the compiler for the resolution of overloaded methods incorporates the following phases:

1. The compiler performs overload resolution without permitting boxing, unboxing, or the use of a variable arity call.
2. If phase (1) fails, the compiler performs overload resolution allowing boxing and unboxing, but excluding the use of a variable arity call.
3. If phase (2) fails, the compiler performs overload resolution combining a variable arity call, boxing, and unboxing.

Example 5.20 provides some insight into how the compiler determines the most specific overloaded method using these three phases. The example has six overloaded declarations of the method `action()`. The signature of each method is given by the local variable `signature` in each method. The first formal parameter of each method is the *signature of the call* that invoked the method. The printout from each method allows us to see which method call resolved to which method. The `main()` method contains 10 calls, (8) to (17), of the `action()` method. In each call, the first argument is the signature of that method call.

An important point to note is that the compiler chooses a *fixed arity* call over a variable arity call, as seen in the calls from (8) to (12):

Click here to view code image

```
(String) => (String)                                (8) calls (1)
(String, int) => (String, int)                      (9) calls (2)
(String, Integer) => (String, int)                 (10) calls (2)
(String, int, byte) => (String, int, int)          (11) calls (3)
(String, int, int) => (String, int, int)           (12) calls (3)
```

An unboxing conversion (`Integer` to `int`) takes place for the call at (10). A widening primitive conversion (`byte` to `int`) takes place for the call at (11).

Variable arity calls are chosen from (13) to (17):

Click here to view code image

```
(String, int, long) => (String, Number[])          (13) calls (5)
(String, int, int, int) => (String, Integer[])     (14) calls (4)
```

```
(String, int, double) => (String, Number[])        (15) calls (5)
(String, int, String) => (String, Object[])        (16) calls (6)
(String, boolean) => (String, Object[])            (17) calls (6)
```

When a variable arity call is chosen, the method determined has the most specific variable arity parameter that is applicable for the actual argument. For example, in the method call at (14), the type `Integer[]` is more specific than either `Number[]` or `Object[]`. Note also the boxing of the elements of the implicitly created array in the calls from (13) to (17).

Example 5.20 *Overloaded Method Resolution*

Click here to view code image

```
import static java.lang.System.out;

class OverloadResolution {

  public void action(String str) {                  // (1)
    String signature = "(String)";
    out.println(str + " => " + signature);
  }

  public void action(String str, int m) {           // (2)
    String signature = "(String, int)";
    out.println(str + " => " + signature);
  }

  public void action(String str, int m, int n) {    // (3)
    String signature = "(String, int, int)";
    out.println(str + " => " + signature);
  }

  public void action(String str, Integer... data) { // (4)
    String signature = "(String, Integer[])";
    out.println(str + " => " + signature);
  }

  public void action(String str, Number... data) {  // (5)
    String signature = "(String, Number[])";
    out.println(str + " => " + signature);
  }

  public void action(String str, Object... data) {  // (6)
    String signature = "(String, Object[])";
    out.println(str + " => " + signature);
  }
```

```
  public static void main(String[] args) {
    OverloadResolution ref = new OverloadResolution();
    ref.action("(String)");                                  // (8)  calls (1)
    ref.action("(String, int)",       10);                   // (9)  calls (2)
    ref.action("(String, Integer)",   Integer.valueOf(10));  // (10) calls (2)
    ref.action("(String, int, byte)",     10, (byte)20);     // (11) calls (3)
    ref.action("(String, int, int)",      10,  20);          // (12) calls (3)
    ref.action("(String, int, long)",     10,  20L);         // (13) calls (5)
    ref.action("(String, int, int, int)", 10,  20,  30);     // (14) calls (4)
    ref.action("(String, int, double)",   10,  20.0);        // (15) calls (5)
    ref.action("(String, int, String)",   10,  "what?");     // (16) calls (6)
    ref.action("(String, boolean)",       false);            // (17) calls (6)
  }
}
```

For output from the program, see the explanation for **Example 5.20** in the text.

5.11 Reference Casting and the `instanceof` Operator

In this section we explore *type casting of references* and the `instanceof` operator that can be used to perform either *type comparison* or *pattern matching*.

The Cast Operator

The basic form of the type cast expression for reference types has the following syntax:

Click here to view code image

```
(destination_type) reference_expression
```

where the *reference expression* evaluates to a reference value of an object of some reference type. A type cast expression checks that the reference value refers to an object whose type is compatible with the *destination type*, meaning that its type is a subtype of the *destination type*. The construct `(` *destination_type* `)` is usually called the *cast operator*. The result of a type cast expression for references is always a reference value of an object. The literal `null` can be cast to any reference type.

The code below illustrates the various scenarios that arise when using the cast operator. In this discussion, it is the type cast expression that is important, not the evaluation of the assignment operator in the declaration statements. At (1), the cast is from the superclass `Object` to the subclass `String`; the code compiles and at runtime this cast is permitted, as the reference `obj` will denote an object of class `String`. At (2), the cast is from the superclass `Object` to the subclass `Integer`; the code compiles, but at run-

time this cast results in a `ClassCastException`, since the reference `obj` will denote an object of class `String`, which cannot be converted to an `Integer`. At (3), the cast is from the class `String` to the class `Integer`. As these two classes are unrelated, the compiler flags an error for the cast.

Click here to view code image

```
Object  obj = new String("Cast me!");
String  str = (String) obj;         // (1) Cast from Object to String.
Integer iRef1 = (Integer) obj;      // (2) Cast from Object to Integer, but
                                    //     ClassCastException at runtime.
Integer iRef2 = (Integer) str;      // (3) Compile-time error!
                                    //     Cast between unrelated types.
```

The following conversions can be applied to the operand of a cast operator:

- Both widening and narrowing reference conversions, followed optionally by an unchecked conversion (§11.2, p. 575)
- Both boxing and unboxing conversions

Boxing and unboxing conversions that can occur during casting are illustrated by the following code. Again, it is the type cast expression that is important in this discussion, rather than whether the assignment operator requires one in the declaration statements.

Click here to view code image

```
// (1) Boxing and casting: Number <-- Integer <-- int:
Number num = (Number) 100;
// (2) Casting, boxing, casting: Object <-- Integer <-- int <-- double:
Object obj = (Object) (int) 10.5;
// (3) Casting, unboxing, casting: double <--- int <-- Integer <-- Object:
double d = (double) (Integer) obj;
```

Note that the resulting object at (1) and (2) is an `Integer`, but the resulting value at (3) is a `double`. The boxing conversions from `int` to `Integer` at (1) and (2) are implicit, and the unboxing conversion from `Integer` to `int` at (3) is also implicit.

Casting to a supertype is always permitted, but redundant, since a supertype reference can always refer to an object of its subtype.

The `instanceof` Type Comparison Operator

The binary `instanceof` operator can be used for comparing *types*. It has the following syntax when used as a type comparison operator (note that the keyword is composed of lowercase letters only):

Click here to view code image

```
reference_expression instanceof destination_type
```

The `instanceof` type comparison operator returns `true` if the left-hand operand (i.e., the reference value that results from the evaluation of *reference expression*) can be a *subtype* of the right-hand operand (*destination type*). It always returns `false` if the left-hand operand is `null`. If the `instanceof` operator returns `true`, the corresponding type cast expression `(` *destination_type* `)` applied to the left-hand operand *reference_expression* will always be valid:

Click here to view code image

```
(destination_type) reference_expression
```

Both the type cast expression and the `instanceof` type comparison operator require a compile-time check and a runtime check. The compile-time check determines whether there is a subtype–supertype relationship between the source and destination types. Given that the type of the *reference expression* is *source type*, the compiler determines whether a reference of *source type* and a reference of *destination type* can refer to objects of a reference type that is a *common subtype* of both *source type* and *destination type* in the type hierarchy. If this is not the case, then obviously there is no relationship between the types, and neither the cast nor the `instanceof` type comparison operator application would be valid. At runtime, the *reference expression* evaluates to a reference value of an object. The `instanceof` type comparison operator determines whether the type of this object is a subtype of the *destination type*.

With the classes `Light` and `String` as *source type* and *destination type*, respectively, there is no subtype–supertype relationship between *source type* and *destination type*. The compiler would reject casting a reference of type `Light` to type `String` or applying the `instanceof` operator, as shown at (2) and (3) in **Example 5.21**. References of the classes `Light` and `TubeLight` can refer to objects of the class `TubeLight` (or its subclasses) in the inheritance hierarchy depicted in **Figure 5.1**, **p. 194**. Therefore, it makes sense to apply the `instanceof` type comparison operator or to cast a reference of the type `Light` to the type `TubeLight` as shown at (4) and (5), respectively, in **Example 5.21**.

At runtime, the result of applying the `instanceof` type comparison operator at (4) is `false` because the reference `light1` of the class `Light` will actually denote an object of the subclass `LightBulb`, and this object cannot be denoted by a reference of the peer class `TubeLight`. Applying the cast at (5) results in a `ClassCastException` for the same reason. This is the reason why cast conversions are said to be *unsafe*, as they may throw a `ClassCastException` at runtime. Note that if the result of the `instanceof` type comparison operator is `false`, the cast involving the operands will throw a `ClassCastException`.

In **Example 5.21**, the result of applying the `instanceof` type comparison operator at (6) is also `false` because the reference `light1` will still denote an object of the class `LightBulb`, whose objects cannot be denoted by a reference of its subclass `SpotLightBulb`. Thus applying the cast at (7) causes a `ClassCastException` to be thrown at runtime.

The situation shown at (8), (9), and (10) illustrates typical usage of the `instanceof` type comparison operator to determine which object a reference is denoting so that it can be cast for the purpose of carrying out some specific action—as we will see later, the `instanceof` pattern match operator provides a better solution for this particular case (**p. 274**). The reference `light1` of the class `Light` is initialized to an object of the subclass `NeonLight` at (8). The result of the `instanceof` type comparison operator at (9) is `true` because the reference `light1` will denote an object of the subclass `NeonLight`, whose objects can also be denoted by a reference of its superclass `TubeLight`. By the same token, the cast at (10) is valid. If the result of the `instanceof` type comparison operator is `true`, the cast involving the operands will be valid as well.

So far in **Example 5.21**, the *static* type of the reference in the left-hand operand of the `instanceof` comparison operator has been a *supertype* of the destination type. At (12), the *static* type (`SpotLightBulb`) of the reference in the left-hand operand of the `instanceof` comparison operator is a *subtype* of the destination type (`Light`). Since the *dynamic* type of the object is `SpotLightBulb`, which is a subtype of the destination type `Light`, the `instanceof` comparison operator returns `true`. The reference value of the object can be trivially assigned to a reference of the supertype `Light`, and the cast shown at (13) is actually redundant. This behavior of the `instanceof` comparison operator is allowed because of backward compatibility.

Example 5.21 *The* `instanceof` *Type Comparison and Cast Operators*

Click here to view code image

```
// See Figure 5.1, p. 194, for inheritance hierarchy.
class Light { /* ... */ }
class LightBulb extends Light { /* ... */ }
```

```
class SpotLightBulb extends LightBulb { /* ... */ }
class TubeLight extends Light { /* ... */ }
class NeonLight extends TubeLight { /* ... */ }

public class WhoAmI {
  public static void main(String[] args) {
    boolean result1, result2, result3, result4;
    Light light1 = new LightBulb();                    // (1)
    //  String str = (String) light1;                  // (2) Compile-time error!
    //  result1 = light1 instanceof String;            // (3) Compile-time error!

    result2 = light1 instanceof TubeLight;             // (4) false: peer class.
    //  TubeLight tubeLight1 = (TubeLight) light1;     // (5) ClassCastException!

    result3 = light1 instanceof SpotLightBulb;         // (6) false: superclass.
    //  SpotLightBulb spotRef = (SpotLightBulb) light1;// (7) ClassCastException!

    light1 = new NeonLight();                          // (8)
    if (light1 instanceof TubeLight) {                 // (9) true.
      TubeLight tubeLight2 = (TubeLight) light1;       // (10) OK.
      // Can now use tubeLight2 to access an object of the class NeonLight,
      // but only those members that the object inherits or overrides
      // from the superclass TubeLight.
    }

    SpotLightBulb light2 = new SpotLightBulb();        // (11)
    result4 = light2 instanceof Light;                 // (12) true.
    Light light = (Light) light2;                      // (13) OK. Redundant cast.
  }
}
```

As we have seen, the `instanceof` type comparison operator effectively determines whether the reference value in the reference on the left-hand side refers to an object whose class is a subtype of the type specified on the right-hand side. At runtime, it is the type of the actual object denoted by the reference on the left-hand side that is compared with the type specified on the right-hand side. In other words, what matters at runtime is the type of the actual object denoted by the reference, not the declared or static type of the reference.

Example 5.22 provides more examples of the `instanceof` type comparison operator. It is instructive to go through the print statements and understand the results printed by the program. The following facts should be noted:

- The literal `null` is not an instance of any reference type, as shown in the print statements at (1), (2), and (16).

- An instance of a superclass is not an instance of its subclass, as shown in the print statement at (4).
- An instance of a class is not an instance of a totally unrelated class, as shown in the print statement at (10).
- An instance of a class is not an instance of an interface type that the class does not implement, as shown in the print statement at (6).
- Any array of a non-primitive type is an instance of both `Object` and `Object[]` types, as shown in the print statements at (14) and (15), respectively.

Example 5.22 *Using the* `instanceof` *Type Comparison Operator*

Click here to view code image

```
// See Figure 5.4, p. 245, for inheritance hierarchy.
interface IStack
interface ISafeStack extends IStack
class Stack implements IStack
class SafeStack extends Stack implements ISafeStack
```

Click here to view code image

```
public class Identification {
  public static void main(String[] args) {
    Object obj = new Object();
    Stack stack = new Stack(10);
    SafeStack safeStack = new SafeStack(5);
    IStack iStack;

    String strFormat = "(%d)  %-25s instance of %-25s: %s%n";
    System.out.printf(strFormat, 1,
        null, Object.class, null instanceof Object);      // Always false.
    System.out.printf(strFormat, 2,
        null, IStack.class, null instanceof IStack);      // Always false.

    System.out.printf(strFormat, 3, stack.getClass(), Object.class,
        stack instanceof Object);     // true: instance of subclass of Object.
    System.out.printf(strFormat, 4, obj.getClass(), Stack.class,
        obj instanceof Stack);        // false: Object not subtype of Stack.
    System.out.printf(strFormat, 5, stack.getClass(), Stack.class,
        stack instanceof Stack);      // true: instance of Stack.
    System.out.printf(strFormat, 6, obj.getClass(), IStack.class,
        obj instanceof IStack);       // false: Object does not implement IStack.
    System.out.printf(strFormat, 7, safeStack.getClass(), IStack.class,
        safeStack instanceof IStack); // true: SafeStack implements IStack.

    obj = stack;           // No cast required: assigning subclass to superclass.
```

```
    System.out.printf(strFormat, 8, obj.getClass(), Stack.class,
        obj instanceof Stack);        // true: instance of Stack.
    System.out.printf(strFormat, 9, obj.getClass(), IStack.class,
        obj instanceof IStack);       // true: Stack implements IStack.
    System.out.printf(strFormat, 10, obj.getClass(), String.class,
        obj instanceof String);       // false: no relationship.
    iStack = (IStack) obj; // Cast required: assigning superclass to subclass.
    System.out.printf(strFormat, 11, iStack.getClass(), Object.class,
        iStack instanceof Object);       // true: instance of subclass of Object.
    System.out.printf(strFormat, 12, iStack.getClass(), Stack.class,
        iStack instanceof Stack);        // true: instance of Stack.

    String[] strArray = new String[10];
//  System.out.printf(strFormat, 13, strArray.getClass(), String.class,
//      strArray instanceof String);     // Compile-time error: no relationship.
    System.out.printf(strFormat, 14, strArray.getClass(), Object.class,

        strArray instanceof Object);     // true: array subclass of Object.
    System.out.printf(strFormat, 15, strArray.getClass(), Object[].class,
        strArray instanceof Object[]);   // true: array subclass of Object[].
    System.out.printf(strFormat, 16, strArray[0], Object.class,
        strArray[0] instanceof Object);  // false: strArray[0] is null.
    System.out.printf(strFormat, 17, strArray.getClass(), String[].class,
        strArray instanceof String[]);   // true: array of String.

    strArray[0] = "Amoeba strip";
    System.out.printf(strFormat, 18, strArray[0].getClass(), String.class,
        strArray[0] instanceof String);  // true: strArray[0] instance of String.
  }
}
```

Output from the program:

Click here to view code image

```
(1)  null                    instance of class java.lang.Object   : false
(2)  null                    instance of interface IStack         : false
(3)  class Stack             instance of class java.lang.Object   : true
(4)  class java.lang.Object  instance of class Stack              : false
(5)  class Stack             instance of class Stack              : true
(6)  class java.lang.Object  instance of interface IStack         : false
(7)  class SafeStack         instance of interface IStack         : true
(8)  class Stack             instance of class Stack              : true
(9)  class Stack             instance of interface IStack         : true
(10)  class Stack             instance of class java.lang.String   : false
(11)  class Stack             instance of class java.lang.Object   : true
(12)  class Stack             instance of class Stack              : true
(14)  class [Ljava.lang.String; instance of class java.lang.Object   : true
```

```
(15)  class [Ljava.lang.String; instance of class [Ljava.lang.Object;: true
(16)  null                      instance of class java.lang.Object   : false
(17)  class [Ljava.lang.String; instance of class [Ljava.lang.String;: true
(18)  class java.lang.String    instance of class java.lang.String   : true
```

The `instanceof` Pattern Match Operator

The `instanceof` operator can also be used for *pattern matching*, as explained below. It has the following syntax when used as a *pattern match operator*:

$$\textit{reference\_expression}\ \texttt{instanceof}\ \overbrace{\textit{destination\_type}\ \ \textit{pattern\_variable}}^{\textit{type\_pattern}}$$

Note that the syntax of the `instanceof` pattern match operator augments the syntax of the `instanceof` type comparison operator with a *pattern variable*. The *type pattern* comprises a *local reference variable declaration*. The `instanceof` pattern match operator essentially combines the `instanceof` type comparison operator with a type cast and initialization of a *pattern variable*.

As with the `instanceof` type comparison operator, a compile-time check determines whether there is a subtype–supertype relationship between the static type of the left-hand operand and the *destination type*. However, as opposed to the `instanceof` type comparison operator, the compiler flags an error if the *destination type* is *not* a *subtype* of the *static* type of the left-hand operand.

At runtime, if the reference value of the object resulting from the evaluation of the *reference expression* can be *cast* to the *destination type* (i.e., the dynamic type of the resulting object is a *subtype* of the *destination type*), the object is said to *match* the *type pattern*. Only in this case does the `instanceof` pattern match operator return `true` and, as a side effect, *the reference value of the object is cast to the destination type and assigned to the pattern variable*. The *pattern variable* is never created or initialized if the `instanceof` pattern match operator returns `false`. It always returns `false` if the left-hand operand is `null`.

We assume the following types in the discussion below. See **Example 5.10, p. 241**, for type declarations.

Click here to view code image

```
interface IStack                    (defines methods push() and pop())
interface ISafeStack extends IStack (with boolean methods isEmpty() and isFull())
```

```
class Stack implements IStack
class SafeStack extends Stack implements ISafeStack
```

The `instanceof`-and-cast idiom is commonly used to elicit the subtype-specific behavior of an object referenced by a supertype reference. The code below at (1) and (2) is equivalent. However, this idiom can be expressed compactly with the `instanceof` pattern match operator as it type checks the object and casts its reference value that is assigned to the pattern variable. When this operator returns `true`, it is said to *introduce* a *pattern variable* into a well-defined scope, which in this case is the `if` block.

Click here to view code image

```
IStack stack = new SafeStack(20); // Supertype reference denotes subtype object.

// Using the instanceof type comparison operator. (1)
if (stack instanceof SafeStack) {                  // Correct subtype?
  SafeStack safestack = (SafeStack) stack;         // Cast to subtype.
  System.out.println(safestack.isFull());          // Call subtype-specific method.
}

// Using the instanceof pattern match operator.   (2)
if (stack instanceof SafeStack safestack) {
  System.out.println(safestack.isFull());
}
```

In contrast to the `instanceof` type comparison operator, the destination type in the `instanceof` pattern match operator *must* be a subtype of the static type of the left-hand operand. In other words, the reference value of the object is always cast to a subtype by the operator. In the code below, the compiler complains that the expression type (`ISafeStack`) cannot be a subtype of the pattern types `ISafeStack` and `IStack`, respectively, at (1) and (2), in the context of the `instanceof` pattern match operator.

Click here to view code image

```
ISafeStack safestack = new SafeStack(20);
if (safestack instanceof ISafeStack stack) {     // (1) Compile-time error!
    stack.push("Hi");
}
if (safestack instanceof IStack stack) {         // (2) Compile-time error!
    stack.push("Howdy");
}
```

A pattern variable *cannot* shadow another local variable by the same name—that is, it cannot be redeclared in a pattern if the local variable of the same name is still in scope.

Click here to view code image

```
IStack stack = new SafeStack(20);
SafeStack safestack = new SafeStack(5);       // Local variable safestack

if (stack instanceof SafeStack safestack) {   // Error: safestack redeclared.
  System.out.println(safestack.isFull());
}
```

A pattern variable introduced by the `instanceof` pattern match operator can shadow a *field* of the same name in the class.

Click here to view code image

```
// Field declaration:
private static IStack myStack = new Stack(10);

...
IStack stack = new SafeStack(20);
if (stack instanceof SafeStack myStack) {    // Local variable introduced.
  System.out.println(myStack.isFull());      // Shadows field reference.
}
myStack.push("Hello");                       // Field reference.
```

A pattern variable is not `final` in its scope unless it is declared `final`—in other words, the pattern variable can be declared with the modifier `final`, as shown at (1). Changing the value of a `final` pattern variable is flagged as a compile-time error, as shown at (2).

Click here to view code image

```
IStack stack = new SafeStack(
if (stack instanceof final SafeStack safestack) {  // (1) final pattern variable.
  safestack = new SafeStack(100);                  // (2) Compile-time error!
  System.out.println(safestack.isFull());
}
```

Scope of Pattern Variables

We first examine the scope of a pattern variable in an `if-else` statement. If the `instanceof` pattern match operator returns `true` in the conditional of an `if` statement, the pattern variable is introduced and its scope is the `if` block. Not surprisingly, this is also the case for the `if-else` statement. The pattern variable is *not* accessible in the `else` block.

Click here to view code image

```
IStack stack = new SafeStack(20);
if (stack instanceof SafeStack safestack) {
  System.out.println(safestack.isFull());   // Pattern variable in scope.
} else {

  System.out.println(safestack.isEmpty());  // Compile-time error!
}
```

It is important to keep in mind that the `instanceof` pattern match operator introduces a pattern variable if and only if it returns `true`. In the `if-else` statement below, the `if` block is only executed if the conditional is `true`—that is, if the `instanceof` pattern match operator returns `false`. In that case, no pattern variable is introduced, and thus no pattern variable is ever accessible in the `if` block. However, if the conditional is `false`, the `instanceof` pattern match operator must be `true`, and thus introduces a pattern variable that is guaranteed to be accessible in the `else` block.

Click here to view code image

```
IStack stack = new SafeStack(20);
if (!(stack instanceof SafeStack safestack)) { // Logical complement operator (!)
  System.out.println(safestack.isFull());      // Compile-time error.
} else {
  System.out.println(safestack.isFull());      // Pattern variable in scope.
}
```

In fact, if the `if` block does *not complete normally* (e.g., by executing a `return`, a `break`, or a `continue` statement) as shown at (1), the pattern variable introduced is accessible in the `else` block and in code after the `if-else` statement.

Click here to view code image

```
IStack stack = new SafeStack(20);
if (!(stack instanceof SafeStack safestack)) {
  System.out.println("No safestack here");

  return;                                     // (1) Does not complete normally.
} else {
  System.out.println(safestack.isFull());     // Pattern variable in scope.
}
System.out.println(safestack.isEmpty());      // Pattern variable still in scope.
```

The `instanceof` pattern match operator can also introduce a pattern variable in certain boolean expressions. The conditional in the `if` statement below uses the conditional-AND operator (`&&`). The short-circuit evaluation of the `&&` operator ensures that the right-hand operand is only executed if the left-hand operand evaluates to `true` , thereby introducing the pattern variable that is then in scope in the right-hand operand. The pattern variable will be in scope in the `if` block if the conditional evaluates to `true` —that is, both operands of the `&&` operator return `true` . Applying the logical complement (`!`) operator to the conditional expression below works the same way as we have seen earlier with the `if-else` statement.

Click here to view code image

```
IStack stack = new SafeStack(20);
if (stack instanceof SafeStack safestack && safestack.isFull()) {
  System.out.println("safestack is full");

  Object obj = safestack.pop();
}
```

The conditional-OR operator (`||`) does *not* introduce a pattern variable in the `if` block. Because of short-circuit evaluation of the `||` operator, the right-side operand of the conditional is only evaluated if the left-side operand is `false` , but then no pattern variable has been introduced in the boolean expression by the `instanceof` pattern match operator.

Click here to view code image

```
IStack stack = new SafeStack(20);
if (stack instanceof SafeStack safestack || safestack.isFull()) { // Compile-time
                                                                  error.
  System.out.println("safestack is full");
  Object obj = safestack.pop();                                // Compile-time error.
}
```

Using the same pattern variable in `instanceof` expressions below results in a compile-time error, as this is analogous to redeclaring a local variable.

Click here to view code image

```
IStack stack = new SafeStack(20);
if (stack instanceof SafeStack safestack &&  // Compile-time error!
    stack instanceof ISafeStack safestack) { // Duplicate variable safestack
  System.out.println(safestack);
}
```

The `instanceof` pattern match operator can be used in the conditional (ternary) operator, and it's semantics are analogous to those of the `if-else` statement:

Click here to view code image

```
IStack stack = new SafeStack(20);
boolean result = stack instanceof SafeStack safestack ? safestack.isEmpty()
                                                       : false;
```

The `instanceof` pattern match operator can also introduce a pattern variable in loops—for example, in `for` and `while` statements. This allows conditional processing of objects where loop termination is controlled by the `boolean` value of the `instanceof` pattern match operator and the pattern variable referring to the next object for processing.

Using the `instanceof` pattern match operator makes the code concise and safe, as it combines three tasks into a single operation: subtype checking, reference casting, and assignment to a local variable.

5.12 Polymorphism

A supertype reference can denote an object of its subtypes—conversely, a subtype object can act as an object of its supertypes. This is because there is an *is-a* relationship between a subtype and its supertypes.

Since a supertype reference can denote objects of different types at different times during execution, a supertype reference is said to exhibit *polymorphic* behavior (meaning *has many forms*). A subtype object also exhibits polymorphic behavior, since it can act as any object of its supertypes.

When a non-`private` instance method is called, the method definition executed is determined by *dynamic method lookup*, based on *the type of the object* denoted by the reference at runtime. Dynamic method lookup (also known as *late binding*, *dynamic binding*, and *virtual method invocation*) is the process of determining which method definition a method call signature denotes at runtime. It is performed starting in the class of the object on which the method is invoked, and moves up the inheritance hierarchy to find the supertype with the appropriate method definition.

For *a call to an overridden method using a supertype reference*, dynamic method lookup can determine a different method definition to execute at different times depending on the type of the object that is denoted by the supertype reference during execution. The overridden method is said to exhibit polymorphic behavior. Exploiting this polymorphic behavior is illustrated next.

The inheritance hierarchy depicted in **Figure 5.7** is implemented in **Example 5.23**. Note that both the `draw()` and `area()` methods are overridden in **Figure 5.7**. The `abstract` method `draw()` in the `IDrawable` interface at (1) is implemented in all subclasses of the `abstract` class `Shape`. It is also implemented by the class `Graph`. The invocation of the `draw()` method in the three loops at (3), (4), and (6) in **Example 5.23** relies on the polymorphic behavior of references and dynamic method lookup. The array `drawables` holds `IDrawable` references that can be assigned the reference value of an object whose class implements the `IDrawable` interface: a `Square`, a `Circle`, a `Rectangle`, and a `Graph`, as shown at (2). At runtime, dynamic lookup determines the `draw()` method implementation that will execute, based on the type of the object denoted by each element in the array.

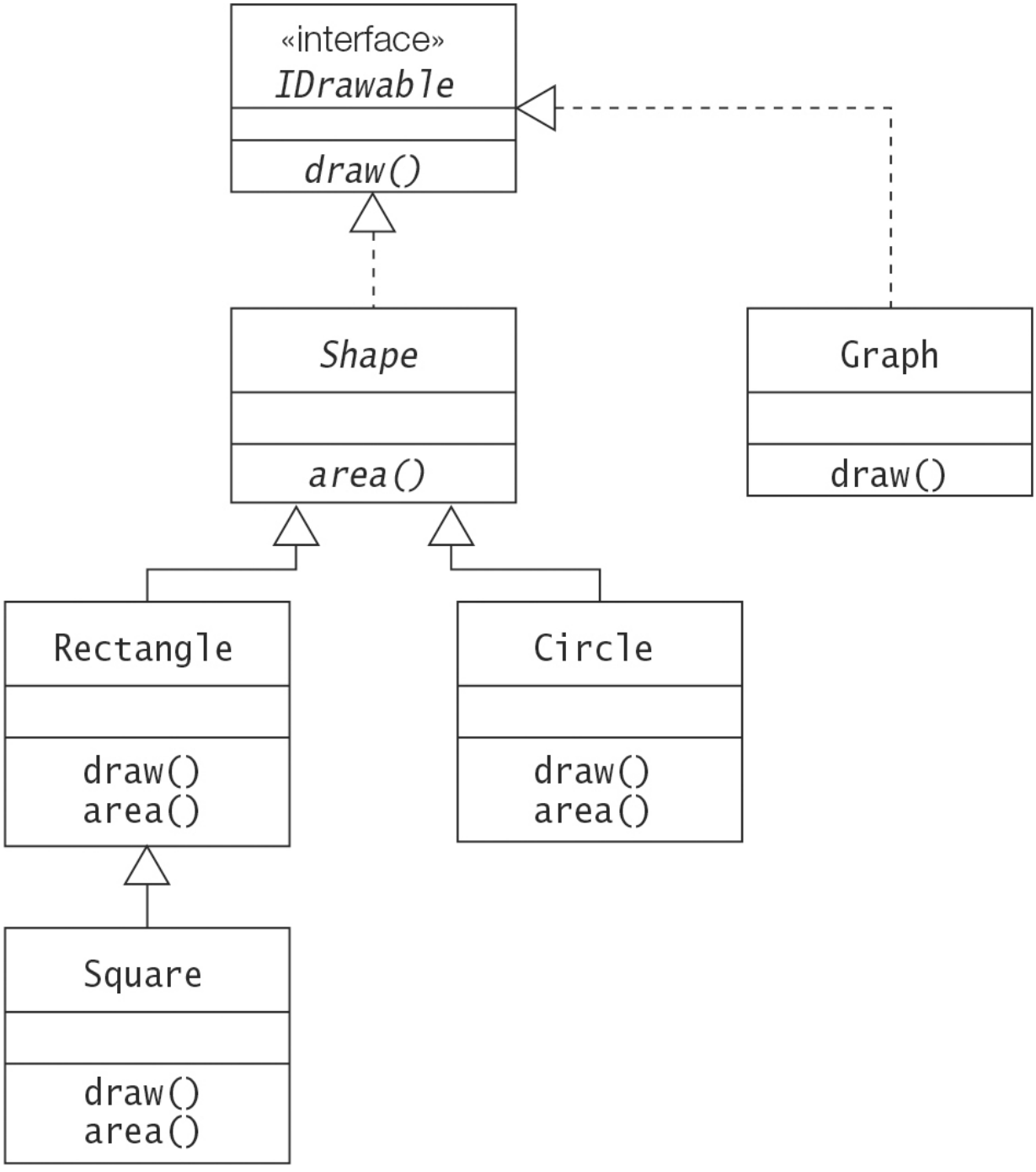

Figure 5.7 *Type Hierarchy to Illustrate Polymorphism*

Example 5.23 *Using Polymorphism*

Click here to view code image

```
// File: PolymorphRefs.java
interface IDrawable {                                                    // (1)
  void draw();
}
//_____________________________________________________________________________
abstract class Shape implements IDrawable {
  abstract public void area();
}
//_____________________________________________________________________________
class Circle extends Shape {
  @Override public void draw() { System.out.println("Drawing a Circle."); }
  @Override public void area() {
    System.out.println("Computing area of a Circle.");
  }
}
//_____________________________________________________________________________
class Rectangle extends Shape {
  @Override public void draw() { System.out.println("Drawing a Rectangle."); }
  @Override public void area() {
    System.out.println("Computing area of a Rectangle.");
  }
}
//_____________________________________________________________________________
class Square extends Rectangle {
  @Override public void draw() { System.out.println("Drawing a Square."); }
  @Override public void area() {
    System.out.println("Computing area of a Square.");
  }
}
//_____________________________________________________________________________
class Graph implements IDrawable {
  @Override public void draw() { System.out.println("Drawing a Graph."); }
}
//_____________________________________________________________________________
public class PolymorphRefs {
  public static void main(String[] args) {
    IDrawable[] drawables
        = {new Square(), new Circle(), new Rectangle(), new Graph()};     // (2)

    System.out.println("Draw drawables:");
    for (IDrawable drawable : drawables) {                               // (3)
      drawable.draw();
    }

    System.out.println("Only draw shapes:");
    for (IDrawable drawable : drawables) {                               // (4)
      if (drawable instanceof Shape) {                                   // (5)
```

```
          drawable.draw();
        }
      }

      System.out.println("Only compute area of shapes:");
      for (IDrawable drawable : drawables) {                          // (6)
        if (drawable instanceof Shape shape) {                        // (7)
          shape.area();                                               // (8)
        }
      }
    }
  }
```

Output from the program:

Click here to view code image

```
Draw drawables:
Drawing a Square.
Drawing a Circle.
Drawing a Rectangle.
Drawing a Graph.
Only draw shapes:
Drawing a Square.
Drawing a Circle.
Drawing a Rectangle.
Only compute area of shapes:
Computing area of a Square.
Computing area of a Circle.
Computing area of a Rectangle.
```

In the `for(:)` loop at (3) in **Example 5.23**, depending on the type of the object denoted by the loop variable `drawable`, the call to the `draw()` method will result in the `draw()` method of this object to be executed.

The `instanceof` type comparison operator returns `true` if the reference specified as its left-hand operand at runtime denotes an object whose type is the same as or is a subtype of the reference type specified in its right-hand operand. If the types of the operands are unrelated, the compiler issues an error.

Click here to view code image

```
IDrawable d1 = new Rectangle();
System.out.println(d1 instanceof IDrawable); // true. Rectangle is an IDrawable.
System.out.println(d1 instanceof Shape);     // true. Rectangle is a Shape.
System.out.println(d1 instanceof Rectangle); // true. Rectangle is a Rectangle.
```

```
System.out.println(d1 instanceof Circle);    // false. Rectangle is not a Circle.
System.out.println(d1 instanceof Graph);     // false. Rectangle is not a Graph.
// System.out.println(d1 instanceof String); // Unrelated. Compile-time error.
```

In **Example 5.23**, the `abstract` method `area()` of the `abstract` class `Shape` is implemented by all subclasses of the superclass `Shape`. The code below does not compile, as the compiler determines that the method `area()` is not defined by the interface `IDrawable`.

Click here to view code image

```
IDrawable d2 = new Square();  // Subtype object denoted by supertype reference.
d2.area();        // Method not defined for type IDrawable. Compile-time error!
```

In order to elicit subtype-specific behavior in an object that is denoted by a super-type reference, the reference must be *cast* to the subtype. The cast is specified as `(` *reference_type* `)`. The cast in this case applies a narrowing reference conversion from a supertype to a subtype. The cast appeases the compiler. At runtime, the type of the object still determines which `area()` method will be executed.

Click here to view code image

```
((Square) d2).area();         // Prints "Computing area of a Square."
((Shape) d2).area();          // Prints "Computing area of a Square."
```

The code above executes normally because the reference `d2` at runtime denotes an object whose type defines the appropriate `area()` method. The code below shows that the cast alone is not enough to guarantee that the execution will proceed normally, even though there was no compile-time error. The `Graph` object denoted by the reference `d3` at runtime does not define the `area()` method.

Click here to view code image

```
IDrawable d3 = new Graph();       // No compile-time error.
((Shape) d3).area();              // Throws a ClassCastException!
```

Guaranteeing the correct subtype and casting to a subtype reference safely can be accomplished using the `instanceof` pattern match operator.

Click here to view code image

```
if (d2 instanceof Shape shape) {           // true
  shape.area();                            // Prints "Computing area of a Square."
```

```
} else {
  System.out.println(d2.getClass().getName() + " is not a Shape." );
}

if (d3 instanceof Shape shape) {           // false
  shape.area();
} else {
  System.out.println(d3.getClass().getName() +
                     " is not a Shape." ); // Prints "Graph is not a Shape."
}
```

In the `for(:)` loop at (4) in **Example 5.23**, we are only interested in drawing objects from the `drawables` array that are of type `Shape`. The binary `instanceof` type comparison operator in the conditional of the `if` statement at (5) is used to determine whether the type of an object in the `drawables` array is of type `Shape`. The `draw()` method is only called on objects that satisfy this condition.

The `for(:)` loop at (6) in **Example 5.23** uses the `instanceof` pattern match operator at (7) to select those `IDrawable` objects whose type is `Shape`, and invoke the `area()` method on them.

A `private` instance method does not exhibit polymorphic behavior. A call to such a method can occur only within the class and gets bound to the `private` method implementation at compile time.

Overloaded instance methods do not exhibit polymorphic behavior, as their calls are bound at compile time, unless an overloaded method is also overridden and invoked by a supertype reference.

Static methods also do not exhibit polymorphic behavior, as these methods do not belong to objects.

Polymorphism is achieved through inheritance and interface implementation. Code relying on polymorphic behavior will still work without any change if new subclasses or new classes implementing the interface are added. If no obvious *is-a* relationship is present, polymorphism is best achieved by using aggregation with interface implementation.

Polymorphism and dynamic method lookup form a powerful programming paradigm that simplifies client definitions, encourages object decoupling, and supports dynamically changing relationships between objects at runtime.

Review Questions

5.22 Given the following class and reference declarations, what can be said about the statement `y = (Sub) x`?

Click here to view code image

```
// Class declarations:
class Super {}
class Sub extends Super {}

// Reference declarations:
  Super x = null;
  Sub y = null;
```

Select the one correct answer.

a. It is illegal at compile time.

b. It is legal at compile time, but it might throw a `ClassCastException` at runtime.

c. It is definitely legal at runtime, but the cast operator `(Sub)` is not strictly needed.

d. It is definitely legal at runtime, and the cast operator `(Sub)` is needed.

5.23 Given the following code:

Click here to view code image

```
class A { }
class B extends A { }
class C extends B { }
```

Which `boolean` expression is `true` only when the reference `o` refers to an object of class `B`, and not to an object of class `A` or class `C`?

Select the one correct answer.

a. `(o instanceof B) && (!(o instanceof A))`

b. `(o instanceof B) && (!(o instanceof C))`

c. `!((o instanceof A) || (o instanceof B))`

d. `(o instanceof B)`

e. `(o instanceof B) && !((o instanceof A) || (o instanceof C))`

5.24 What will the following program print when run?

Click here to view code image

```
interface I{}
interface J{}
class C implements I {}
class D extends C implements J {}

public class RQ07A100 {
  public static void main(String[] args) {
    I x = new D();
    if (x instanceof I) System.out.print("I");
    if (x instanceof J) System.out.print("J");
    if (x instanceof C) System.out.print("C");
    if (x instanceof D) System.out.print("D");
    System.out.println();
  }
}
```

Select the one correct answer.

a. The program will not print any letters.

b. `ICD`

c. `IJD`

d. `IJCD`

e. `ID`

5.25 What will be the result of compiling and running the following program?

Click here to view code image

```
public class RQ800A20 {
  static void compute(int... ia) {                          // (1)
    System.out.print("|");
    for(int i : ia) {
      System.out.print(i + "|");
    }
```

```
    System.out.println();
  }
  static void compute(int[] ia1, int... ia2) {                 // (2)
    compute(ia1);
    compute(ia2);
  }
  static void compute(int[] ia1, int[]... ia2d) {               // (3)
    for(int[] ia : ia2d) {
      compute(ia);
    }
  }
  public static void main(String[] args) {
    compute(new int[] {10, 11}, new int[] {12, 13, 14});       // (4)
    compute(15, 16);                                           // (5)
    compute(new int[] {17, 18}, new int[][] {{19}, {20}});     // (6)
    compute(null, new int[][] {{21}, {22}});                   // (7)
  }
}
```

Select the one correct answer.

a. The program will fail to compile because of errors in one or more calls to the `compute()` method.

b. The program compiles, but it throws a `NullPointerException` when run.

c. The program compiles and prints:

```
|10|11|
|12|13|14|
|15|16|
|19|
|20|
|21|
|22|
```

d. The program compiles and prints:

```
|12|13|14|
|15|16|
|10|11|
|19|
|20|
|21|
|22|
```

5.26 Given the following code:

Click here to view code image

```
public class RQ43 {
  public static void main(String[] args) {
    Object x = "acme";
    if (x instanceof String s && s.length() > 5) {
      x = s.equals("acme") ? "1" : "2";
    } else {
      x = s.equals("acme") ? "3" : "4";
    }
    System.out.println(x);
  }
}
```

What is the result?

Select the one correct answer.

a. `1`

b. `2`

c. `3`

d. `4`

e. The program will throw an exception at runtime.

f. The program will fail to compile.

5.27 Which option will compile without errors?

Select the one correct answer.

a.

Click here to view code image

```
Integer x = Integer.valueOf(42);
if (x instanceof Integer s) {
    System.out.print(s.intValue());
```

b.

Click here to view code image

```
var x = Integer.valueOf(42);
if (x instanceof Integer s) {
    System.out.print(s.intValue());
}
```

c.

Click here to view code image

```
var x = Integer.valueOf(42);
if (x instanceof String s) {
    System.out.print(s.toUpperCase());
}
```

d.

Click here to view code image

```
Number x = Integer.valueOf(42);
if (x instanceof Integer s) {
    System.out.print(s.intValue());
}
```

e.

Click here to view code image

```
Integer x = Integer.valueOf(42);
if (x instanceof Number s) {
    System.out.print(s.intValue());
}
```

5.28 Which of the following statements are true? Select the two correct answers.

a. The `instanceof` pattern match operator does not throw a `NullPointerException` if its left operand is `null`.

b. The `instanceof` pattern match operator throws a `NullPointerException` if its left operand is `null`.

c. A pattern variable is only introduced when the `instanceof` pattern match operator returns `true`.

d. A pattern variable is only introduced when the `instanceof` pattern match operator returns `false`.

5.29 What will be the result of compiling and running the following program?

Click here to view code image

```
class A           { int f() { return 0; } }
class B extends A { int f() { return 1; } }
class C extends B { int f() { return 2; } }

public class Polymorphism {
  public static void main(String[] args) {
    A ref1 = new C();
    B ref2 = (B) ref1;
    System.out.println(ref2.f());
  }
}
```

Select the one correct answer.

a. The program will fail to compile.

b. The program will compile, but it will throw a `ClassCastException` at runtime.

c. The program will compile and print `0` when run.

d. The program will compile and print `1` when run.

e. The program will compile and print `2` when run.

5.30 What will be the result of compiling and running the following program?

Click here to view code image

```
class A {
  private int f() { return 0; }
  public int g() { return 3; }
}
class B extends A {
  private int f() { return 1; }
  public int g() { return f(); }
}
```

```
class C extends B {
  public int f() { return 2; }
}

public class Polymorphism2 {
  public static void main(String[] args) {
    A ref1 = new C();
    B ref2 = (B) ref1;
    System.out.println(ref2.g());
  }
}
```

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?

Click here to view code image

```
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:

Click here to view code image

```
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.

Click here to view code image

```
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:

Click here to view code image

```
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 byte-code 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.

Click here to view code image

```
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*

Click here to view code image

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

Click here to view code image

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

  private MachineState state;
```

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

Click here to view code image

```
// 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:

Click here to view code image

```
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*

Click here to view code image

```
// 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);
  }
}
```

Click here to view code image

```
// 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:

Click here to view code image

```
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:

Click here to view code image

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

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

Click here to view code image

```
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.

Click here to view code image

```
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` :

Click here to view code image

```
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*

Click here to view code image

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

Click here to view code image

```
// 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:

Click here to view code image

```
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.

Click here to view code image

```
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.

Click here to view code image

```
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`:

Click here to view code image

```
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.

Click here to view code image

```
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.

5.14 Record Classes

Exchanging *plain data* between programs is a common task. *Plain data objects* allow efficient *aggregation* of data and provide transparent *access* to it. In the literature, such data objects are referred to by different names; the acronym *POJOs* (*Plain Old Java Objects*) is common in the Java community.

A *plain data class* can always be implemented as a normal class, but this is tedious and repetitive, entailing boilerplate code and full weight of the OOP programming model. In **Example 5.27**, the class `CD_v1` represents the data for a CD. A full-fledged implementation of this data class requires declaration of its instance fields that constitute the data, an appropriate constructor to initialize the data fields, *get methods* to allow access to the data, and overriding the pertinent methods from the `Object` class to make `CD` objects more meaningful and useful. (For details on overriding methods from the `Object` class, see **Chapter 14**, p. 741). Although most IDEs can generate the code neces-

sary to implement such data objects, they lack the incremental validation required when changes are made to the data model of the normal class.

Example 5.27 *The* `CD` *Class*

Click here to view code image

```
package record.basics;
// The different genres in music.
public enum Genre {POP, JAZZ, OTHER}
```

Click here to view code image

```
package record.basics;
import java.time.Year;
import java.util.Objects;

/** A class that represents a CD. */
public class CD_v1 {

  // Instance fields:
  private final String artist;      // Name of the artist.
  private final String title;       // Title of the CD.
  private final int    noOfTracks;  // Number of tracks on the CD.
  private final Year   year;        // Year the CD was released.
  private final Genre  genre;       // Music genre of the CD.

  // Non-zero argument constructor:
  public CD_v1(String artist, String title, int noOfTracks,
               Year year, Genre genre) {
    this.artist     = artist;
    this.title      = title;
    this.noOfTracks = noOfTracks;
    this.year       = year;
    this.genre      = genre;
  }

  // Get methods:
  public String getArtist()     { return this.artist; }
  public String getTitle()      { return this.title; }
  public int    getNoOfTracks() { return this.noOfTracks; }
  public Year   getYear()       { return this.year; }
  public Genre  getGenre()      { return this.genre; }

  // Overridden methods from the Object class:
  @Override public String toString() {
    return String.format("<%s, \"%s\", %d, %s, %s>",
```

```
          this.artist, this.title, this.noOfTracks, this.year, this.genre);
    }

    @Override public boolean equals(Object obj) {
      return (this == obj)
          || (obj instanceof CD_v1 other
              && this.artist.equals(other.artist)
              && this.title.equals(other.title)
              && this.noOfTracks == other.noOfTracks
              && this.year == other.year
              && this.genre == other.genre);
    }

    @Override public int hashCode() {
      return Objects.hash(this.artist, this.title, this.noOfTracks,
                          this.year, this.genre);
    }
  }
```

Record Class Basics

A *record class* in Java is a special-purpose class that simplifies declaration and handling of *an aggregate of values* that comprise the state of a plain data object. A record class defines immutable fields and the compiler generates the *get methods* (also called *getter* or *accessor methods*) necessary to access the values of fields in instances of the record class. A record class generally does not provide any heavy processing of the data; the primarily goal is to allow users to access the data.

The basic syntax for a record class is shown below. The *record header* of a top-level record class can specify the `public` access modifier, as in a top-level normal class. The contextual keyword `record` is used in the header to declare a record class. It has this special meaning only in this context. A record class has a *name*, similar to a normal class. The header specifies a comma-separated *component list* that comprises *field declarations* (called *record components*), where each field declaration specifies the *name* and the *type* of the field. For the basic record class declaration, the record body can be left empty.

Click here to view code image

```
access_modifier record record_name(component_list)      // Record header
{ }                                                     // Empty record body
```

The `CD_v0` class in **Example 5.27** can be declared as a record class, as shown in **Example 5.28**. The `public` record class `CD` declared at (1) in **Example 5.28** is equivalent to the normal class `CD_v1` in **Example 5.27**. The compiler automatically generates

the necessary fields, contructor, and method declarations for the record class `CD`. The client class `DataUser` in **Example 5.28** utilizes data objects of record class `CD`.

Example 5.28 *The* `CD` *Record Class*

Click here to view code image

```
package record.basics;
import java.time.Year;

/** A record class that represents a CD. */
public record CD(String artist, String title, int noOfTracks,          // (1)
                 Year year, Genre genre) { /* Empty body */ }
```

Click here to view code image

```
package record.basics;
import java.time.Year;

public class DataUser {
  public static void main(String[] args) {
    // Some ready-made CDs:                                              (2)
    CD cd0 = new CD("Jaav",      "Java Jive",       8, Year.of(2017), Genre.POP);
    CD cd1 = new CD("Jaav",      "Java Jam",        6, Year.of(2017), Genre.JAZZ);
    CD cd2 = new CD("Funkies",   "Lambda Dancing", 10, Year.of(2018), Genre.POP);
    CD cd3 = new CD("Genericos", "Keep on Erasing", 8, Year.of(2018), Genre.JAZZ);
    CD cd4 = new CD("Genericos", "Hot Generics",   10, Year.of(2018), Genre.JAZZ);

    // An array of CDs.
    CD[] cdArray = {cd0, cd1, cd2, cd3, cd4};

    System.out.println("     Artist    Title           No. Year Genre");
    for(int i = 0; i < cdArray.length; ++i) {
      CD cd = cdArray[i];
      String cdToString = String.format("%-10s%-16s%-4d%-5s%-5s",
          cd.artist(), cd.title(), cd.noOfTracks(),                    // (3)
          cd.year(), cd.genre());
      System.out.printf("cd%d: %s%n", i, cdToString);
    }

    System.out.println();
    System.out.println(cd0.toString());                                // (4)

    CD cdX = new CD("Jaav",      "Java Jive",       8, Year.of(2017), Genre.POP);
    System.out.println("cd0.equals(cdX): " + cd0.equals(cdX));          // (5)
```

```
    }
}
```

Output from the program:

Click here to view code image

```
      Artist    Title           No. Year Genre
cd0: Jaav      Java Jive       8   2017 POP
cd1: Jaav      Java Jam        6   2017 JAZZ
cd2: Funkies   Lambda Dancing  10  2018 POP
cd3: Genericos Keep on Erasing 8   2018 JAZZ
cd4: Genericos Hot Generics    10  2018 JAZZ

cd0: CD[artist=Jaav, title=Java Jive, noOfTracks=8, year=2017, genre=POP]
cd0.equals(cdX): true
```

The compiler automatically generates the necessary declarations for a record class when it is declared according to the simplified syntax form, in particular, when the record body is empty.

- Each record component in the component list results in a `private`, `final` *instance field* in the record. In other words, these *component fields* pertain to objects of the record class, and their values constitute the state of a record. From the `CD` record class declaration, the following component fields are created: `String artist`, `String title`, `int noOfTracks`, `Year year`, and `Genre genre`. The component fields in the `CD` record class are equivalent to the instance field declarations in the `CD_v0` class in **Example 5.27**.
  Fields being `final` means that once a `final` field is initialized, its value cannot be changed. However, if a field refers to a mutable object, this object can be modified, but the reference value of the object assigned to the field cannot be changed. Record classes are thus said to be *shallowly immutable*.
  The component fields in the `CD` record class have immutable values, as objects of `String`, `Year`, and `Genre` are immutable, as are primitive values. The `CD` record class is therefore immutable. (See also **§6.7**, **p. 356**, for a discussion on immutability.)
  As we shall see, the *order* of the record components in the component list has implications in the record class.
- Each record component results in a `public` zero-argument *get method* (also known as a *getter* method). It is important to note that the get method has the same name and return type as the corresponding record component. This does not create any problem, as method names and field names are in different *name spaces*.

The compiler thus ensures that each record component has a corresponding `private`, `final` component field and a read-only `public` zero-argument get method that returns the value of this field.

The following `public` zero-argument get methods are created for the `CD` record class: `String artist()`, `String title()`, `int noOfTracks()`, `Year year()`, and `Genre genre()`, each having the name and the return type as declared by the corresponding record component. As expected, calling a get method returns the value of the corresponding component field. In **Example 5.28**, the get methods are called at (3) to obtain the values of individual component fields in a `CD` record. Although their names are different, the get methods of the `CD` record class are equivalent to the get methods of the `CD_v0` class in **Example 5.27**.

- In order to initialize all component fields properly, the compiler automatically creates a `public` constructor—called the *implicit canonical constructor*—that has a parameter list that corresponds to the record components in *name, type* and *order*. A call to the canonical constructor in a `new` expression results in the argument values being assigned to the corresponding component fields in the same order. The implicit canonical constructor of a record class functions analogously to the default constructor in a normal class.

  The implementation of the implicit canonical constructor for the `CD` record class is equivalent to the non-zero argument normal constructor implemented in the `CD_v0` class in **Example 5.27**.

  Creating records is no different from creating objects of a normal class. In **Example 5.28**, `CD` records are constructed by calling the implicit canonical constructor at (2) in the `new` expression. Note that the order of the specified values corresponds to that of the record components in the component list.

  Click here to view code image

```
CD cd0 = new CD("Jaav",      "Java Jive",       8, Year.of(2017), Genre.POP);
...
```

- The compiler also creates an implementation of the `toString()` method that is overridden from the `Object` class. The `toString()` method creates a default text representation of a record that includes the name of the record class and a textual list with name-value pairs for each component field in the order specified in the component list.

  In **Example 5.28**, the `toString()` method is explicitly called in the print statement at (4), resulting in the following output:

  Click here to view code image

```
CD[artist=Jaav, title=Java Jive, noOfTracks=8, year=2017, genre=POP]
```

- An implementation of the `equals()` method (§14.2, p. 744) that is overridden from the `Object` class is also created to compare two `CD` records for equality, based on the values of corresponding component fields in the two records. The method compares the component fields in the same order as specified in the component list, and returns `true` if the values in corresponding fields all match—analogous to the implementation of the `equals()` method in the `CD_v0` class in Example 5.27. Overriding of the `equals()` method is essential if records are to be used in collections (§15.12, p. 866).
  In Example 5.28, two records `cd0` and `cdX` are compared for equality with the `equals()` method, showing that the two records are equal, since they have the same state.
- Lastly, the compiler generates an implementation of the `hashCode()` method (§14.3, p. 753) that is overridden from the `Object` class. The method returns a hash code that is computed based on the values of the component fields, in the same order as specified in the component list—analogous to the implementation of the `hashCode()` method in the `CD_v0` class in Example 5.27. Overriding of the `hashCode()` method is essential if records are to be used in sets and maps (§15.8, p. 830).

Restrictions on Record Declarations

A record class is implemented as an implicitly `final` direct subclass of the `java.lang.Record` abstract class that defines the default inherited behavior of such classes. In particular, the `Record` class defines `abstract` methods that override the `equals()`, `hashCode()`, and `toString()` methods of the `Object` class. This means that every record class must implement these methods either explicitly or implicitly. We have seen in Example 5.28 that the compiler provides the implementation of these methods automatically when any of these methods is not declared explicitly in the record class.

A record class cannot have an explicit `extends` clause to declare a direct superclass, not even its direct superclass `Record`. Being `final` also means that a record class cannot be declared `abstract`. However, a record class is seldom declared explicitly `final`. As such, a record class is solely defined by its state and cannot be extended by any subclass.

Note also that the name of a record component in the component list cannot be the same as the name of any of the methods in the `Object` class, except `equals`.

Augmenting Basic Record Class Declaration

In the rest of this section, we explore how the basic declaration of a record class can be augmented and customized. The general syntax of a record class is shown below.

Click here to view code image

```
access_modifier record record_name(component_list) optional_implements_clause {
    optional_constructor_declarations
    optional_member_declarations
}
```

The basic declaration of a record class can be augmented with an `implements` clause in order to implement any interfaces. Constructors can be customized, as can the get methods corresponding to the component fields. New instance methods can be declared, as can static members and initializers, but new instance fields and initializers are not allowed in the record body.

Record Constructors

In this section we discuss three kinds of constructors that can be specified in a record class declaration:

- A *normal canonical record constructor*, if declared, supersedes the implicit canonical record constructor.
- If a normal canonical record constructor is not specified, a *compact canonical record constructor* can be declared. In this case, the implicit canonical record constructor is generated. Note that either the normal canonical record constructor or the compact canonical record constructor can be specified, but not both.
- Any number of *non-canonical record constructors* can be specified, whether or not any canonical constructor is provided.

Normal Canonical Record Constructor

In a basic record class declaration, the compiler generates the *implicit canonical record constructor* discussed earlier to ensure that component fields of a record are initialized properly. This canonical record constructor can be implemented *explicitly* when there is a need to process the argument values before they are assigned to the component fields. This explicit canonical record constructor is called the *normal canonical record constructor* as its declaration resembles the constructor of a normal class. If a record class implements the normal canonical record constructor, the implicit counterpart is *not* generated.

The normal canonical constructor must have the *same parameter list* as the implicit canonical record constructor, in *name*, *type*, and *order*. In the constructor, the compo-

nent fields must be accessed with the `this` reference to distinguish them from the parameter names, and *all* component fields must be initialized in the constructor. In **Example 5.29**, the record class `CD` provides an implementation of the normal canonical constructor at (2). The parameter values are sanitized at (3) before being assigned to the component fields at (4). Running the `DataUser` class in **Example 5.28** with the record class `CD` in **Example 5.29** produces the same results.

Here are a few other restrictions to note regarding the normal canonical constructor:

- It must be at least as accessible as the record class. For example, for a top-level record class that has `public` accessibility, it must be declared `public`.
- It cannot be generic (§11.7, p. 593), or have a `throws` clause in its header (§7.5, p. 388). In other words, any checked exception thrown by the constructor must be handled in the body of the constructor.
- It cannot invoke another record constructor with the `this()` expression to chain constructors locally (p. 209), as is the case with normal constructors.

Example 5.29 *Normal Canonical Record Constructor*

Click here to view code image

```
package record.constructors.canonical;
import java.time.Year;

/** A record class that represents a CD. */
public record CD(String artist, String title, int noOfTracks,          // (1)
                 Year year, Genre genre) {

  // Normal canonical record constructor                               // (2)
  public CD(String artist, String title, int noOfTracks, Year year, Genre genre) {
    // Sanitize the parameter values:                                    (3)
    artist = artist.strip();
    title = title.strip();
    noOfTracks = noOfTracks < 0 ? 0 : noOfTracks;
    year =  year.compareTo(Year.of(2022)) > 0? Year.of(2022) : year;
    genre = genre == null ? Genre.OTHER : genre;
    // Initialize all component fields:                                  (4)
    this.artist     = artist;
    this.title      = title;
    this.noOfTracks = noOfTracks;
    this.year       = year;
    this.genre      = genre;
  }
}
```

Compact Canonical Record Constructor

The *compact canonical record constructor* is a more concise form of the normal canonical record constructor. It eliminates the need for a parameter list and the initialization of the component fields. The parameter list is derived from the component list declared in the record header, and the initialization of *all* component fields is left to the *implicit* canonical record constructor. In fact, the component fields cannot be accessed in the compact constructor, as the component names refer to the parameter names of the implicit canonical constructor. Note that the implicit canonical record constructor is generated when the compact constructor is provided in the record class declaration. The compact constructor is called in a `new` expression, exactly as any other constructor, passing the argument values in the call.

In **Example 5.30**, the compact canonical constructor for the record class `CD` is implemented at (2). It has no parameter list and it does *not* initialize the component fields. It only sanitizes the values passed to the constructor. Again running the `DataUser` class in **Example 5.28** with the record class `CD` in **Example 5.30** produces the same results.

The *implicit* canonical record constructor is called before exiting from the compact canonical constructor; therefore, a `return` statement is not allowed in the compact constructor. The compact constructor primarily functions as a validator for the data values before they are assigned to the component fields by the *implicit* canonical record constructor.

Previous restrictions mentioned for the normal canonical record constructor also apply to the compact canonical record constructor.

Example 5.30 *Compact Record Constructor and Non-Canonical Constructor*

Click here to view code image

```
package record.constructors.compact;
import java.time.Year;

/** A record class that represents a CD. */
public record CD(String artist, String title, int noOfTracks,              // (1)
                 Year year, Genre genre) {

  // Compact canonical record constructor                                    (2)
  public CD {
    // Sanitize the values passed to the constructor:                       (3)
    artist = artist.strip();
    title = title.strip();
    noOfTracks = noOfTracks < 0 ? 0 : noOfTracks;
    year =  year.compareTo(Year.of(2022)) > 0? Year.of(2022) : year;
```

```
    genre = genre == null ? Genre.OTHER : genre;

    // Cannot explicitly assign to component fields:                    (4)
    // this.artist = artist;                    // Compile-time error!
  }

  // A non-canonical record constructor                                 (5)
  public CD() {
    this(" Anonymous  ", "  No title  ", 0, Year.of(2022), Genre.OTHER);  // (6)
  }
}
```

Normal Non-Canonical Record Constructors

A record class declaration can specify any number of *non-canonical record constructors*—that is, constructors whose signature is *not* the same as that of the canonical constructor.

In **Example 5.30**, a non-canonical constructor for the record class `CD` is implemented at (5). It is a zero-argument non-canonical record constructor to create a record with default values for the component fields.

The first statement in the non-canonical constructor at (6) is an explicit invocation of the canonical constructor with the `this()` expression. The signature of the `this()` expression matches that of the canonical constructor in type and in order. The rule is that chaining of constructors with the `this()` expression (**p. 209**) must ultimately lead to the canonical constructor being invoked so that all component fields are initialized. In **Example 5.30**, the constructor invocation at (6) leads to the compact constructor being invoked, and whose execution in turn leads to the implicit canonical record constructor being invoked. Use of this non-canonical constructor is illustrated in the following print statement:

Click here to view code image

```
System.out.println(new CD());        // Calls the non-canonical constructor
```

Output from the print statement shows that the record was created after the string values were sanitized:

Click here to view code image

```
CD[artist=Anonymous, title=No title, noOfTracks=0, year=2022, genre=OTHER]
```

Non-canonical constructors are primarily used for creating specialized records.

The restrictions noted for the normal and the compact canonical constructors do *not* apply to a non-canonical constructor: It can be less accessible than its record class, be generic, and can specify a `throws` clause, but it must invoke another record constructor with the `this()` expression.

Member Declarations

The component fields of a record class are always automatically generated based on the field components specified in the component list. Thus a record class *cannot* declare any new instance fields in addition to those specified in the component list.

However, new instance methods can be declared in a record class. In **Example 5.31**, three new instance methods are declared at (8) in the record class to determine the genre of a CD.

Automatic generation of any get method or methods overridden from the `Object` class (`equals()`, `hashCode()`, and `toString()`) will be suppressed in a record class, if the record class provides an implementation for any of these methods.

In **Example 5.31**, the `CD` record class provides an implementation for the `title()` method at (9) that returns the uppercase version of the string in the `title` field. This explicit method will be invoked when called by a client to get the value of the `title` field in a `CD` record. An explicit get method can be annotated with the `@Override` annotation to enable validation of its method signature against that of the default get method that would otherwise be generated. Such an explicit get method must fulfill the following criteria:

- It must be an instance method that is declared `public`.
- It must have no formal parameters or a `throws` clause.
- Its return type must be the same as the declared type of its corresponding record component.
- It must not be a generic method.

When the object referenced by a field is mutable, its get method can be explicitly implemented to return a copy of the object so that the original object is never modified, maintaining the immutability of the record. As values of individual component fields cannot be changed, a new record with any modified values must be created.

Also, `static` fields, methods, and initializers (**§10.7**, **p. 545**) can be declared. In **Example 5.31**, the `CD` record class declares `static` fields to create some ready-made `CD` records. Argument values passed in the constructor call to create `CD` records will

be sanitized by the compact constructor at (2) before being assigned to the component fields by the implicit canonical record constructor.

Nested static types can also be declared in a record class (**Chapter 9**, **p. 489**). In **Example 5.31**, the enum type `Genre` is declared as a `static` member of the `CD` record class at (11), and is used in the `CD` record class.

In **Example 5.31**, the client class `DataUser2` uses some of the explicitly implemented methods in the `CD` record class. The `CD.isOther()` method is used at (13) to filter an array of CDs created at (12). The output from the program shows that the values passed in the constructor calls were sanitized.

An uppercase version of the string in the `title` field is printed when the `title()` method is explicitly invoked on a `CD` record. However, note that the automatically generated `toString()` method, called at (14), does not use the uppercase version of the title. The reason is that the fields are accessed directly by the generated methods in the record class, and not by calling the get methods. The `toString()` method accesses the `title` field directly, thereby accessing the original string in the field. In order to use the uppercase version of the string in all contexts, it can be converted to uppercase in the compact constructor by replacing line (4a) with (4b) so that the uppercase version of the string is assigned to the `title` field—and thereby making the `title()` method declared at (9) redundant.

Example 5.31 *Other Member Declarations in a Record Class*

Click here to view code image

```
package record.customize;
import java.time.Year;

/** A record class that represents a CD. */
public record CD(String artist, String title, int noOfTracks,              // (1)
                 Year year, Genre genre) {

  // Compact canonical record constructor                                    (2)
  public CD {
    // Sanitize the values passed to the constructor:                       (3)
    artist = artist.strip();
    title = title.strip();                                                 // (4a)
//  title = title.strip().toUpperCase();                                   // (4b)
    noOfTracks = noOfTracks < 0 ? 0 : noOfTracks;
    year =  year.compareTo(Year.of(2022)) > 0? Year.of(2022) : year;
    genre = genre == null ? Genre.OTHER : genre;

// Cannot explicitly assign to component fields:                            (5)
```

```
  // this.artist = artist;                                    // Compile-time error!
    }

    // A non-canonical record constructor                                    (6)
    public CD() {
      this(" Anonymous  ", "  No title  ", 0, Year.of(2022), Genre.OTHER);  // (7)
    }

    //  New instance methods:                                                (8)
    public boolean isPop()   { return this.genre == Genre.POP; }
    public boolean isJazz()  { return this.genre == Genre.JAZZ; }
    public boolean isOther() { return this.genre == Genre.OTHER; }

    // Customize a get method:                                               (9)
    @Override public String title() {
       return this.title.toUpperCase();
    }

    // Static fields with some ready-made CDs:                               (10)
    public final static CD cd0
        = new CD("   Jaav",      "Java Jive",       8, Year.of(2017), Genre.POP);
    public final static CD cd2
        = new CD("Funkies ",    " Lambda Dancing  ", 10, Year.of(2024), null);
    public final static CD cd4
        = new CD("Genericos", "Hot Generics",   -5, Year.of(2018), Genre.JAZZ);

    // Declare a nested type:
    public enum Genre {POP, JAZZ, OTHER}                                  //  (11)
}
```

Click here to view code image

```
package record.customize;

public class DataUser2 {
  public static void main(String[] args) {
    CD[] cdArray = {CD.cd0, CD.cd2, CD.cd4, new CD()};                    // (12)
    for(int i = 0; i < cdArray.length; ++i) {
      CD cd = cdArray[i];
      if (cd.isOther()) {                                                 // (13)
        System.out.println(cd.toString());                                // (14)
      }
    }
    System.out.println("Title: " + cdArray[1].title());                   // (15)
  }
}
```

Output from the program:

Click here to view code image

```
CD[artist=Funkies, title=Lambda Dancing, noOfTracks=10, year=2022, genre=OTHER]
CD[artist=Anonymous, title=No title, noOfTracks=0, year=2022, genre=OTHER]
Title: LAMBDA DANCING
```

Other Aspects of Record Classes

Some other aspects of record classes are mentioned below, and are covered in detail elsewhere in the book.

Implementing Interfaces

Records can implement interfaces, which is no different from a normal class implementing interfaces. As a record class is implicitly `final`, it cannot extend other classes. The `CD` record class below implements the `Serializable` interface and the `Comparable<CD>` interface. **Example 20.6**, **p. 1263**, uses the `CD` record class to demonstrate record serialization.

Click here to view code image

```
public record CD(String artist, String title, int noOfTracks,          // (1)
    Year year, Genre genre) implements Serializable and Comparable<CD> {

  @Override public int compareTo(CD other) { /* See Example 16.1, p. 883. */ }

  public enum Genre implements Serializable {POP, JAZZ, OTHER}
}
```

The `CD` record class in **Example 16.1**, **p. 883**, implements the `Comparable<CD>` interface so that `CD` records can be compared.

Record Serialization

Example 20.6, **p. 1263**, demonstrates serializing records to external storage. It is important to note that serialization of records cannot be customized, as with objects of normal classes, and that a record is always deserialized using the canonical constructor.

Generic Record Classes

Generic record classes can be delcared, analogous to generic classes (**Chapter 11**, **p. 563**). The generic record class `Container` below has one type parameter (`T`):

Click here to view code image

```
record Container<T>(T item) { /* Empty body */ }
```

We can create records by parameterizing the generic record class:

Click here to view code image

```
Container<String> p0 = new Container<>("Hi");
Container<Integer> p1 = new Container<>(Integer.valueOf(10));
```

Nested Record Classes

Record classes can be declared as nested record classes—in particular, as `static` *member types* and as `static` *local types*. For details on nested record classes, see **Chapter 9**, **p. 489**.

Direct Permitted Subtypes of Sealed Interfaces

A record class is only allowed in the `permits` clause of a *sealed interface*, if it implements the sealed interface. As a record class is implicitly `final`, it also fulfills the second criterion for a permitted direct subtype (**p. 311**).

Finally, it is worth noting that records in Java are designed so that the record state solely defines the representation of the record. Record classes require less effort to implement than normal classes and have an efficient memory footprint and runtime performance. Also, their immutability provides thread safety in concurrent applications.

5.15 Sealed Classes and Interfaces

Design by inheritance promotes code reuse in the OOP model—where subtypes can inherit code from their supertypes. Such an inheritance hierarchy is depicted in **Figure 5.8**, where the subclasses `PrintedBook`, `Ebook`, and `Audiobook` can inherit and reuse code from the superclass `Book`. The inheritance hierarchy of the `Book` superclass can also be regarded as a *domain model* for books (albeit greatly simplified for this exposition). However, this domain model as represented by the inheritance hierarchy in **Figure 5.8** does not give the `Book` superclass much control over which classes can be classified as books. It can easily be exploited by clients to create their own subclasses of the `Book` superclass, thereby undermining the domain model represented by the inheritance hierarchy in **Figure 5.8**. If the `PrintedBook` subclass is not declared `final`, it can also be freely extended to define more kinds of books.

The superclass `Book` in **Figure 5.8** cannot control which subclasses belong to the book domain, as it cannot control its extensibility. When it comes to domain modeling using inheritance, modifiers such as `abstract` and `final` are not adequate, nor are the accessibility modifiers to control the use of the superclass. In this section, we look at how sealed classes and interfaces aid in domain modeling that promotes accessibility of the superclass and allows control of extensibility of its subclasses. Sealed classes also have the additional benefits of resulting in secure hierarchies and aiding the compiler to provide better analysis of potential problems in the code.

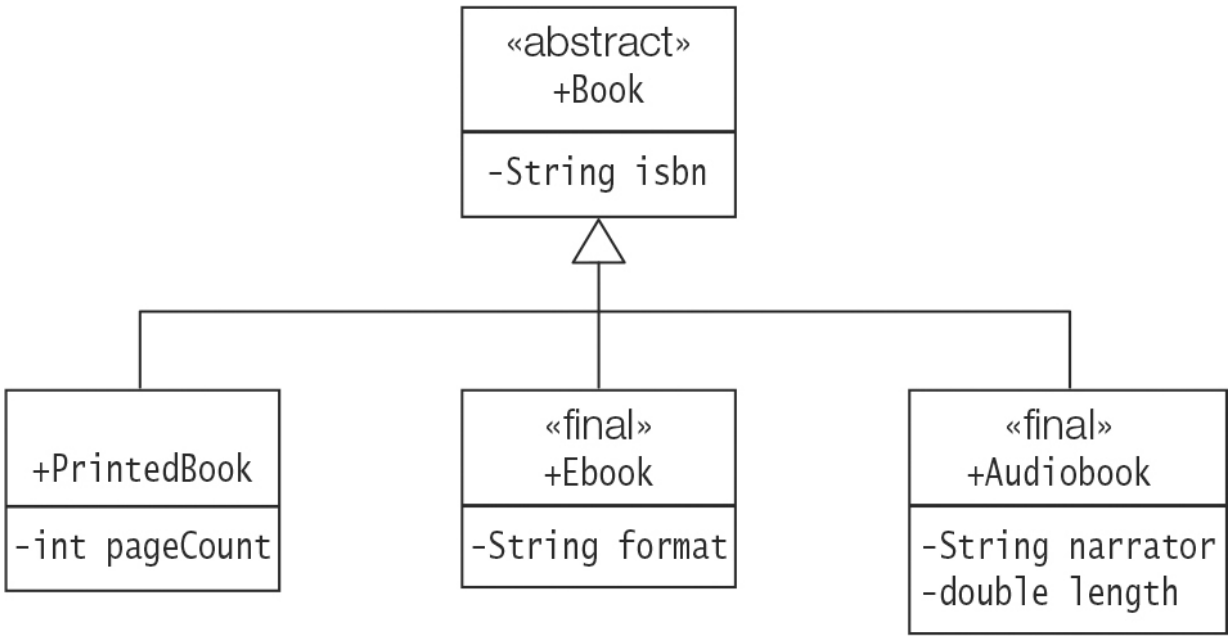

Figure 5.8 *A Domain Model as Represented by Inheritance Hierarchy*

Sealed Classes

Sealing of classes and interfaces allows fine-grained control over the inheritance hierarchy of reference types in a given domain. A sealed class or interface only allows specific classes or interfaces to extend or implement its definition. To implement sealed classes, the language has introduced two new modifiers, `sealed` and `non-sealed`, and the `permits` clause—these three identifiers are *contextual keywords* that have special meaning only in the context of defining sealed types. As we shall see, the modifier `final` is also an integral part of defining sealed types.

We first look at sealed classes, and later we discuss sealed interfaces and compare the two (**p. 315**).

Going back to **Figure 5.8** with the simplified domain model of books, we want to ensure that only the subclasses `PrintedBook`, `Ebook`, and `Audiobook` can *directly extend* the abstract superclass `Book` (i.e., ensure that the three subclasses are the only *permitted direct subclasses* of the `sealed` class `Book`). **Figure 5.9** shows the modeling of the inheritance hierarchy with the required constraints. In this example, the classes in-

volved are located in the same package. Skeletal code for the classes in shown in the discussion below, and complete class declarations can be found in **Example 5.32**.

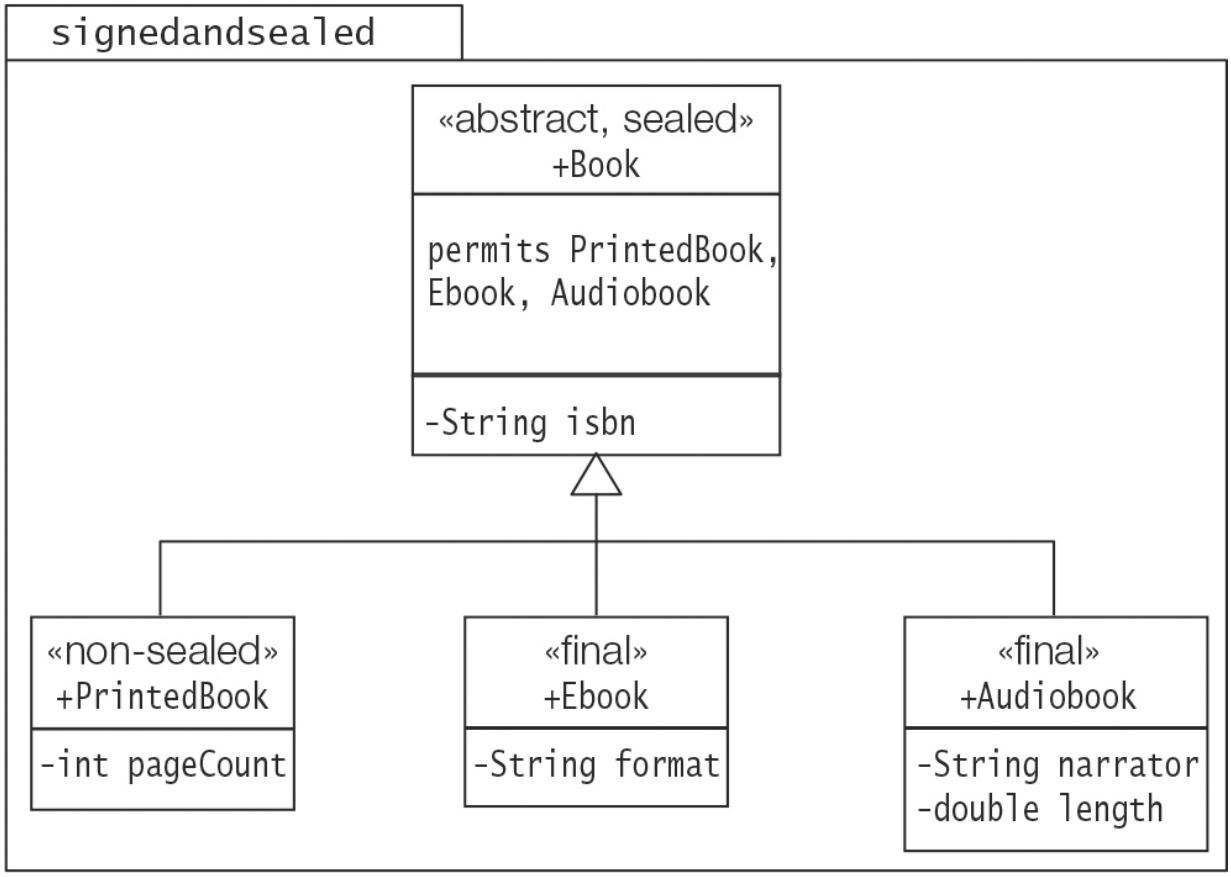

Figure 5.9 *Sealed Classes for Domain Modeling of Inheritance Hierarchy*

A sealed class is declared using the modifier `sealed` and the optional `permits` clause in the class header. The `permits` clause is optional if a sealed class and its permitted direct subclasses are declared in the same compilation unit (**p. 317**). If a class specifies the `permits` clause, the class must be declared `sealed`. The `permits` clause, if specified, must appear after any `extends` and `implements` clauses in the class header.

The superclass `Book` is declared `sealed` and explicitly specifies a `permits` clause with a list of its permitted direct subclasses, as shown at (1) below. Keep in mind that since these subclasses are `public`, they are declared in separate compilation units, but belong to the same package.

Click here to view code image

```
public abstract sealed class Book permits PrintedBook, Ebook, Audiobook {/*(1)*/}
```

Each *permitted direct subclass* of a sealed class is bound by the following *contract*:

- Each permitted direct subclass *must extend* its direct `sealed` superclass. Note that the sealing relationship is defined between two consecutive levels in the inheritance

hierarchy: between a superclass and its direct subclasses. This is also the case for the permitted direct subclasses of the `Book` superclass:

Click here to view code image

```
public non-sealed class PrintedBook extends Book {/*...*/}

public final class Ebook extends Book {/*...*/}

public final class Audiobook extends Book {/*...*/}
```

- Each permitted direct subclass must be declared either `sealed`, `non-sealed`, or `final`. This avoids any unwanted subclassing to the inheritance hierarchy of a sealed class.
  A permitted direct subclass that is declared `final` cannot be extended, as in the case of a `final` normal class. This implies that the inheritance hierarchy of a `final` permitted subclass cannot be extended. The permitted direct subclasses `Ebook` and `Audiobook` are declared `final`. Any attempt to extend them would result in a compile-time error.

  Click here to view code image

```
class EComicBook extends Ebook {}   // Compile-time error!
```

A permitted direct subclass that is declared `non-sealed` can be *freely extended*. This implies that the inheritance hierarchy of a `non-sealed` permitted subclass can be extended. Note that a class cannot be declared `non-sealed` unless it is a permitted direct subclass of a direct `sealed` superclass.
The permitted direct subclass `PrintedBook` is declared `non-sealed`. Client code can readily extend this class:

Click here to view code image

```
class Hardcover extends PrintedBook {}
class Paperback extends PrintedBook {}
```

A `sealed` permitted direct subclass applies sealing to its own permitted direct subclasses, just like its direct `sealed` superclass. This implies that the inheritance hierarchy of a `sealed` permitted subclass can be extended, but in a restricted way. For example, if the permitted subclass `PrintedBook` was declared `sealed`, its permitted direct subclasses would be subject to the same rules for declaring permitted subclasses, as shown below:

Click here to view code image

```
public sealed class PrintedBook extends Book permits Hardcover, Paperback {}
final class Hardcover extends PrintedBook {}
```

```
final class Paperback extends PrintedBook {}
```

- Each permitted direct subclass must be *accessible* by the direct `sealed` superclass. If the `sealed` superclass is in a *named module*, the permitted subclasses must also be in the same module (**Chapter 19**, **p. 1161**). Note that this does not mean that they have to be in the same package in the named module. Their *fully qualified names* can be used in the `permits` clause to access the subclasses in their respective packages (**§6.3**, **p. 326**).
  If the `sealed` superclass is in *the unnamed module*, the permitted subclasses must all be in either the *same named package* or *the unnamed package* as the `sealed` superclass. In **Example 5.32**, the `sealed` superclass `Book` is in a named package (but in the unnamed module), as are the permitted subclasses, specified using the `package` statement (**§6.3**, **p. 326**).

Example 5.32 *Declaring* `sealed`, `non-sealed`, *and* `final` *Classes*

Click here to view code image

```
package signedandsealed;

public abstract sealed class Book permits PrintedBook, Ebook, Audiobook  { // (1)
  private String isbn;
  protected Book(String isbn) {
    this.isbn = isbn;
  }
  public String getIsbn() { return this.isbn; }
}
```

Click here to view code image

```
package signedandsealed;

public non-sealed class PrintedBook extends Book {                        // (2)
  private int pageCount;
  protected PrintedBook(String isbn, int pageCount) {
    super(isbn);
    this.pageCount = pageCount;
  }
  public int getPageCount() { return this.pageCount; }
}
```

Click here to view code image

```
package signedandsealed;
```

```
public final class Ebook extends Book {                                    // (3)
  private String format;
  public Ebook(String isbn, String format) {
    super(isbn);
    this.format = format;
  }
  public String getFormat() { return this.format; }
}
```

Click here to view code image

```
package signedandsealed;

public final class Audiobook extends Book {                                // (4)
  private String narrator;
  private double length;
  public Audiobook(String isbn, String narrator, double length) {
    super(isbn);
    this.narrator = narrator;
    this.length = length;
  }
  public String getNarrator() { return this.narrator; }
  public double getLength() { return this.length; }
}
```

Sealed Interfaces

Analogous to `sealed` classes, `sealed` interfaces can also be defined. However, a `sealed` interface can have *both* subinterfaces and subclasses as its *permitted direct subtypes*. The permitted subtypes can be subclasses that *implement* the `sealed` interface and subinterfaces that *extend* the `sealed` interface. This is in contrast to a `sealed` class that can have direct subclasses, but not direct subinterfaces—as classes cannot be extended or implemented by interfaces.

Figure 5.10 shows the book domain from **Figure 5.9** that has been augmented with a `sealed` interface that specifies its permitted direct subtypes in a `permits` clause:

Click here to view code image

```
public sealed interface Subscribable permits Ebook, Audiobook, VIPSubcribable {}
```

The `sealed` superinterface `Subscribable` has two `final` permitted direct subclasses (that implement the superinterface) and one `non-sealed` direct subinterface (that extends the superinterface). The declarations of the permitted direct subclasses `Ebook`

and `Audiobook` have been updated accordingly so that they implement the direct superinterface `Subscribable`.

Click here to view code image

```
public final class Ebook extends Book implements Subscribable {}
public final class Audiobook extends Book implements Subscribable {}
public non-sealed interface VIPSubcribable extends Subscribable {}
```

The rest of the type declarations from **Figure 5.9** remain the same in **Figure 5.10**:

Click here to view code image

```
public abstract sealed class Book permits PrintedBook, Ebook, Audiobook {}
public non-sealed class PrintedBook extends Book {}
```

Note that it is perfectly possible for a class or an interface to be a permitted direct subtype of more than one direct supertype—as is the case for the `Ebook` and the `Audiobook` subclasses in **Figure 5.10**.

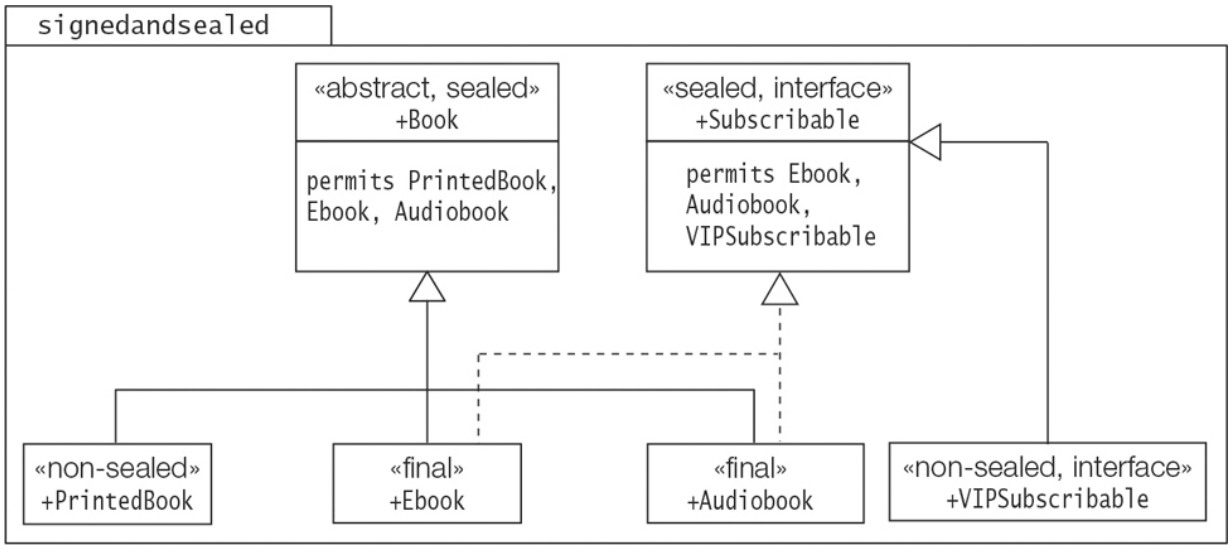

Figure 5.10 *Sealed Classes and Interfaces*

We see from the discussion above that the permitted direct subtypes of a `sealed` superinterface abide by the same *contract* rules as for `sealed` superclasses:

- A permitted direct subclass or subinterface must extend or implement its direct superinterface, respectively.
- Any permitted subclass of a sealed interface must be declared either `sealed`, `non-sealed` or `final`, but any permitted subinterface can only be declared either `sealed` or `non-sealed`. The modifier `final` is not allowed for interfaces.
- The same rules for locality also apply for `sealed` interfaces and their permitted direct subtypes: All are declared either in the same named module or in the same

package (named or unnamed) in the unnamed module.

Enum and Record Types as Permitted Direct Subtypes

By definition, an enum type (p. 287) is either *implicitly* `final` (has *no* enum constants that have a class body, as shown at (2)) or *implicitly* `sealed` (has *at least one* enum constant with a class body that constitutes an implicitly declared direct subclass, as shown at (3) for the constant `DVD_R` that has an empty class body). Thus an enum type can be specified as a permitted direct subtype of a `sealed` superinterface, as shown at (1). The modifiers `final` and `sealed` cannot be explicitly specified in the enum type declaration.

Click here to view code image

```
sealed interface MediaStorage permits CD, DVD {}       // (1) Sealed interface
enum CD implements MediaStorage {CD_ROM, CD_R, CD_W}   // (2) Implicitly final
enum DVD implements MediaStorage {DVD_R {}, DVD_RW}    // (3) Implicitly sealed
```

Analogously, a record class (p. 299) is *implicitly* `final`, and can be specified as a permitted direct subtype of a `sealed` superinterface. The `sealed` interface `MediaStorage` at (1a) now permits the record class `HardDisk` as a direct subtype. Again note that the modifier `final` cannot be specified in the header of the `HardDisk` record class declared at (4).

Click here to view code image

```
sealed interface MediaStorage permits CD, DVD, HardDisk {}// (1a) Sealed interface
record HardDisk(double capacity) implements MediaStorage {}// (4) Implicitly final
```

Deriving Permitted Direct Subtypes of a Sealed Supertype

The `permits` clause of a `sealed` class or interface can be omitted, if all its permitted direct subtypes are declared in the *same compilation unit*—that is, are in the same source file. Since only one reference type can be declared `public` in a compilation unit, the accessibility of the other type declarations cannot be `public`.

The compiler can infer the permitted direct subtypes of any `sealed` class or interface in a compilation unit by checking whether a type declaration fulfills the contract for declaring a permitted direct subtype; that is, it extends/implements a `sealed` supertype and is declared either (implicitly) `sealed`, `non-sealed`, or (implicitly) `final`.

Click here to view code image

```
// File: Book.java (compilation unit)
public abstract sealed class Book {}        // Permitted subclasses are derived.
non-sealed class PrintedBook extends Book {}

sealed interface Subscribable {}            // Permitted subtypes are derived.
final class Ebook extends Book implements Subscribable {}
final class Audiobook extends Book implements Subscribable {}
non-sealed interface VIPSubcribable extends Subscribable {}
```

Given the type declarations in the compilation unit `Book.java`, the compiler is able to infer that `PrintedBook`, `Ebook`, and `Audiobook` are permitted direct subclasses of the `sealed` superclass `Book`, whereas `Ebook`, `Audiobook`, and `VIPSubcribable` are permitted direct subtypes of the `sealed` superinterface `Subscribable`.

Using Sealed Classes and Interfaces

As we have seen, sealed classes and interfaces restrict which other classes or interfaces can extend or implement them. Apart from providing the programmer with a language feature to develop improved domain models for libraries, the compiler can also leverage the sealed types to provide better analysis of potential problems in the code at compile time.

Example 5.33 is a client that uses the `sealed` classes and their permitted direct subclasses defined in **Example 5.32**. Typically, a reference of the `sealed` superclass is used to process a collection of objects of its permitted subtypes. The array `books` is such a collection created at (2). The objects of the permitted subtypes are processed by the enhanced `for(:)` loop at (3). Typically, a cascading `if-else` statement is used in the loop body to distinguish each object in the collection and process it accordingly. Any object of a permitted subclass can be processed by the code as long as the cascading `if-else` statement guarantees that the checks are exhaustive—that is, all permitted subtypes are covered. The compiler cannot help to ensure that this is the case, as it cannot extract the necessary information from the cascading `if-else` statement for this analysis. Note that the conditionals in the cascading `if-else` statement use the `instanceof` pattern match operator to make the code at least less verbose.

Example 5.33 *Using Sealed Classes*

Click here to view code image

```
package signedandsealed;

public class BookAdmin {
  public static void main(String[] args) {
```

```
    // Create some books:                                          // (1)
    PrintedBook pBook = new PrintedBook("888-222", 340);
    Ebook eBook = new Ebook("999-777", "epub");
    Audiobook aBook = new Audiobook("333-555", "Fry", 300.0);

    // Create a book array:                                        // (2)
    Book[] books = {pBook, eBook, aBook};

    // Process the books:                                          // (3)
    for (Book book : books) {
      if (book instanceof PrintedBook pb) {                       // (4)
        System.out.printf("Printed book: %s, %d%n",
                          pb.getIsbn(), pb.getPageCount());
      } else if (book instanceof Ebook eb) {
        System.out.printf("Ebook: %s, %s%n", eb.getIsbn(), eb.getFormat());
      } else if (book instanceof Audiobook ab) {
        System.out.printf("Audiobook: %s, %s, %.1f%n",
                          ab.getIsbn(), ab.getNarrator(), ab.getLength());
      }
    }
  }
}
```

Output from the program:

Click here to view code image

```
Printed book: 888-222, 340
Ebook: 999-777, epub
Audiobook: 333-555, Fry, 300.0
```

Review Questions

5.32 What will be the result of attempting to compile and run the following code?

Click here to view code image

```
public enum Drill {
  ATTENTION("Attention!"), EYES_RIGHT("Eyes right!"),
  EYES_LEFT("Eyes left!"), AT_EASE("At ease!");

  private String command;

  Drill(String command) {
    this.command = command;
```

```
  }
  public static void main(String[] args) {
    System.out.println(ATTENTION);             // (1)
    System.out.println(AT_EASE);               // (2)
  }
}
```

Select the one correct answer.

a. The code compiles, but it reports a `ClassNotFoundException` when run, since an enum type cannot be run as a stand-alone application.

b. The compiler reports errors at (1) and (2), as the constants must be qualified by the enum type name `Drill`.

c. The compiler reports errors at (1) and (2), as the constants cannot be accessed in a static context.

d. The code compiles and prints:

```
ATTENTION
AT_EASE
```

e. The code compiles and prints:

```
Attention!
At ease!
```

f. None of the above

5.33 What will be the result of compiling and running the following code?

Click here to view code image

```
import java.util.Arrays;

public enum Priority {
  ONE(1) { public String toString() { return "LOW"; } },        // (1)
  TWO(2),
  THREE(3) { public String toString() { return "NORMAL"; } },   // (2)
  FOUR(4),
  FIVE(5) { public String toString() { return "HIGH"; } };      // (3)

  private int pValue;
```

```
  Priority(int pValue) {
    this.pValue = pValue;
  }

  public static void main(String[] args) {
    System.out.println(Arrays.toString(Priority.values()));
  }
}
```

Select the one correct answer.

a. The compiler reports syntax errors at (1), (2), and (3).

b. The code compiles and prints:
`[LOW, TWO, NORMAL, FOUR, HIGH]`

c. The code compiles and prints:
`[ONE, TWO, THREE, FOUR, FIVE]`

d. None of the above

5.34 What will be the result of compiling and running the following code?

Click here to view code image

```
public enum TrafficLight {
  RED("Stop"), YELLOW("Caution"), GREEN("Go");

  private String action;

  TrafficLight(String action) {
    this.action = action;
  }

  public static void main(String[] args) {
    TrafficLight green = new TrafficLight("Go");
    System.out.println(GREEN.equals(green));
  }
}
```

Select the one correct answer.

a. The code will compile and print `true`.

b. The code will compile and print `false`.

c. The code will fail to compile, as an enum type cannot be instantiated with the `new` operator.

d. The code will fail to compile, as the enum type does not provide the `equals()` method.

5.35 Given the following code:

Click here to view code image

```
public class RQ54 {
  public static void main(String[] args) {
    Product p = new Product(101, "Tea", 1.99);
    System.out.println(p);
  }
}
```

Which two record class definitions will cause this program to produce the following output?

Click here to view code image

```
Product[id=101, name=TEA, price=1.99]
```

Select the two correct answers.

a.

Click here to view code image

```
public record Product(int id, String name, double price) {
  public Product() {
    this.id = id;
    this.name = name.toUpperCase();
    this.price = price;
  }
}
```

b.

Click here to view code image

```
public record Product(int id, String name, double price) {
  public Product(int newId, String newName, double newPrice) {
```

```
    id = newId;
    name = newName.toUpperCase();
    price = newPrice;
  }
}
```

c.

Click here to view code image

```
public record Product() {
  public Product(int id, String name, double price) {
    this.id = id;
    this.name = name.toUpperCase();
    this.price = price;
  }
}
```

d.

Click here to view code image

```
public record Product(int id, String name, double price) {
  public Product {
    name = name.toUpperCase();
  }
}
```

e.

Click here to view code image

```
public record Product(int id, String name, double price) {
  public String toString(){
    return "Product[id="+id+", name="+name.toUpperCase()+",
            price="+price+"]";
  }
}
```

f.

Click here to view code image

```
 public record Product(int id, String name, double price) {
  public String name() {
    return name.toUpperCase();
  }
  public String toString(){
    return "Product[id="+id+", name="+name+", price="+price+"]";
  }
}
```

5.36 Given the following code:

Click here to view code image

```
public record Employee(String name, double salary) { }
```

and

Click here to view code image

```
public class RQ55 {
  public static void main(String[] args) {
    Employee e1 = new Employee("Bob", 1234);
    Employee e2 = new Employee("Bob", 1234);
    System.out.print(e1.equals(e2));
    System.out.print(e1 == e2);
  }
}
```

What is the result?

Select the one correct answer.

a. `falsefalse`

b. `truetrue`

c. `falsetrue`

d. `truefalse`

5.37 Which statement is true?

Select the one correct answer.

a. A record class cannot declare new fields, other than the ones in its header.

b. A record class provides get and set methods for each component field.

c. A record class cannot have an `extends` clause.

d. A record class directly extends the class `Object`.

5.38 Which statement is true?

Select the one correct answer.

a. A class marked as `sealed` cannot be `abstract`.

b. A class marked as `non-sealed` cannot be `abstract`.

c. A class marked as `sealed` cannot be freely extended.

d. A class that extends a `sealed` class cannot be `sealed`.

e. A class that extends a `sealed` class cannot be `non-sealed`.

5.39 Given the following code:

Click here to view code image

```
public sealed interface X permits Y, Z { }
```

Which of the following options provides a correct definition of the permitted subtypes `Y` and `Z`?

Select the one correct answer.

a.

Click here to view code image

```
public final class Y implements X { }
public sealed interface Z extends X { }
```

b.

Click here to view code image

```
public final class Y implements X { }
public interface Z extends X { }
```

c.

Click here to view code image

```
public non-sealed class Y implements X, Z { }
public sealed interface Z extends X permits Y { }
```

d.

Click here to view code image

```
public class Y implements X, Z { }
public non-sealed interface Z extends X { }
```