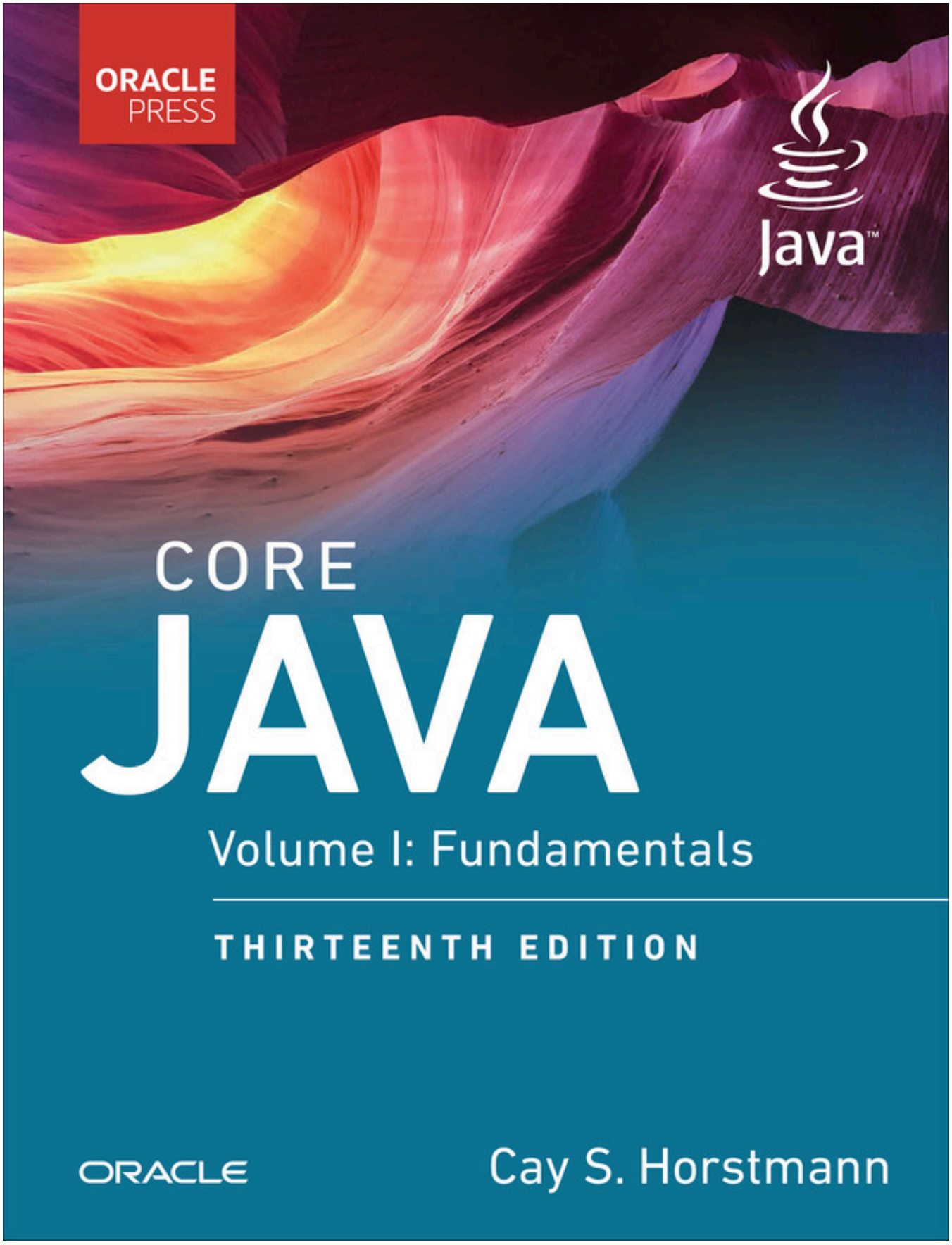
ORACLE
PRESS
Java™
CORE
JAVA
Volume I: Fundamentals
THIRTEENTH EDITION
ORACLE
Cay S. Horstmann

omit the `public` specifier for the subclass method. The compiler then complains that you try to supply a more restrictive access privilege.

5.1.7. Preventing Inheritance: Final Classes and Methods

Occasionally, you want to prevent someone from forming a subclass of one of your classes. Classes that cannot be extended are called *final* classes, and you use the `final` modifier in the definition of the class to indicate this. For example, suppose we want to prevent others from subclassing the `Executive` class. Simply declare the class using the `final` modifier, as follows:

```
public final class Executive extends Manager
{
   . . .
}
```

You can also make a specific method in a class `final`. If you do this, then no subclass can override that method. (All methods in a `final` class are automatically `final`.) For example:

```
public class Employee
{
   . . .
   public final String getName()
   {
      return name;
   }
   . . .
}
```

Note: Recall that fields can also be declared as `final`. A final field cannot be changed after the object has been constructed. However, if a class is declared `final`, only the methods, not the fields, are automatically `final`.

There is only one good reason to make a method or class `final`: to make sure its semantics cannot be changed in a subclass. For example, the `getTime` and `setTime` methods of the `Calendar` class are `final`. This indicates that the designers of the `Calendar` class have taken over responsibility for the conversion between the `Date` class and the calendar state. No subclass should be allowed to mess up this arrangement. Similarly, the `String` class is a `final` class. That means nobody can define a subclass of `String`. In other words, if you have a `String` reference, you know it refers to a `String` and nothing but a `String`.

If you call a method in a constructor, you should declare it as `final`. Otherwise, it can be overridden in a subclass, and it can access a partially constructed subclass instance. Here is an example. For debugging purposes, the `Employee` constructor displays a description of the constructed object.

```
public class Employee
{
   public Employee(String name, double salary, int year, int month, int day)
   {
      this.name = name;
      this.salary = salary;
      hireDay = LocalDate.of(year, month, day);
      System.out.println("Constructed " + description());
   }

   public String description()
   {
      return "An employee with a salary of " + salary;
   }

   . . .
}
```

Now a new class is added to the hierarchy of employee classes—executives with titles:

```
public class Executive
{
   private String title;

   public Executive(String name, String title, double salary,
      int year, int month, int day)
   {
      super(name, salary, year, month, day);
      this.title = title;;
   }

   public String getTitle()
   {
      return title;
   }

   public String description()
   {
      if (title.length() >= 30)
         return "An executive with an impressive title";
      else
         return "An executive with a title of " + title;
   }
}
```

When an `Executive` is constructed, its constructor first calls the `Manager` constructor, which calls the `Employee` constructor, which calls the `description` method. Because of

polymorphism, that is the `description` method in the `Executive` class! Unfortunately, the `Executive` constructor hasn't finished yet. The `title` instance variable is still `null`, causing a `NullPointerException`.

Calling a method in a constructor is inherently dangerous. The constructor must have done enough work for the method to succeed. If the method can be overridden, this becomes very difficult to ensure. Therefore, it is best to call only `final` or `private` methods in a constructor.

Tip: Since Java 21, if you compile with the `-Xlint` or `-Xlint:this-escape` flag, the compiler issues a warning when the constructor of a public class calls a method that is not `final` or `private`.

The name of the flag is a bit unfortunate since there are other situations where the `this` reference can "escape" from a constructor that the compiler does not currently detect.

C++ Note: In C++, method calls in a constructor are not polymorphic. For example, if you call `getDescription` in an `Employee` constructor, it always invokes `Employee::getDescription`.

Some programmers believe that you should declare all methods as `final` unless you have a good reason to want polymorphism. In fact, in C++ and C#, methods do not use polymorphism unless you specifically request it. That may be a bit extreme, but I agree that it is a good idea to think carefully about final methods and classes when you design a class hierarchy.

In the early days of Java, some programmers used the `final` keyword hoping to avoid the overhead of dynamic binding. If a method is not overridden, and it is short, then a compiler can optimize the method call away—a process called *inlining*. For example, inlining the call `e.getName()` replaces it with the field access `e.name`. This is a worthwhile improvement—CPUs hate branching because it interferes with their strategy of prefetching instructions while processing the current one. However, if `getName` can be overridden in another class, then the compiler cannot inline it because it has no way of knowing what the overriding code may do.

Fortunately, the just-in-time compiler in the virtual machine can do a better job than a traditional compiler. It knows exactly which classes extend a given class, and it can check whether any class actually overrides a given method. If a method is short, frequently called, and not actually overridden, the just-in-time compiler can inline it. What happens if the virtual machine loads another subclass that overrides an inlined method? Then the optimizer must undo the inlining. That takes time, but it happens rarely.

Note: Enumerations and records are always final—you cannot extend them.

5.1.8. Casting

Recall from Chapter 3 that the process of forcing a conversion from one type to another is called casting. The Java programming language has a special notation for casts. For example,

```
double x = 3.405;
int nx = (int) x;
```

converts the value of the expression `x` into an integer, discarding the fractional part.

Just as you occasionally need to convert a floating-point number to an integer, you may need to convert an object reference from one class to another. Let's again use the example of an array containing a mix of `Employee` and `Manager` objects:

```
var staff = new Employee[3];
staff[0] = new Manager("Carl Cracker", 80000, 1987, 12, 15);
staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);
```

To actually make a cast of an object reference, use a syntax similar to what you use for casting numeric expressions. Surround the target class name with parentheses and place it before the object reference you want to cast. For example:

```
Manager boss = (Manager) staff[0];
```

There is only one reason why you would want to make a cast—to use an object in its full capacity after its actual type has been temporarily forgotten. For example, in the `ManagerTest` class, the `staff` array had to be an array of `Employee` objects because *some* of its elements were regular employees. We would need to cast the managerial elements of the array back to `Manager` to access any of its new variables. (Note that in the sample code for the first section, I made a special effort to avoid the cast. I initialized the `boss` variable with a `Manager` object before storing it in the array. I needed the correct type to set the bonus of the manager.)

As you know, in Java every variable has a type. The type describes the kind of object the variable refers to and what it can do. For example, `staff[i]` refers to an `Employee` object (so it can also refer to a `Manager` object).

The compiler checks that you do not promise too much when you store a value in a variable. If you assign a subclass reference to a superclass variable, you are promising less, and the compiler will simply let you do it. If you assign a superclass reference to a

subclass variable, you are promising more. Then you must use a cast so that your promise can be checked at runtime.

What happens if you try to cast down an inheritance chain and are "lying" about what an object contains?

```
Manager boss = (Manager) staff[1]; // ERROR
```

When the program runs, the Java runtime system notices the broken promise and generates a `ClassCastException`. If you do not catch the exception, your program terminates. Thus, it is good programming practice to find out whether a cast will succeed before attempting it. Simply use the `instanceof` operator. For example:

```
if (staff[i] instanceof Manager)
{
   boss = (Manager) staff[i];
   . . .
}
```

Finally, the compiler will not let you make a cast if there is no chance for the cast to succeed. For example, the cast

```
String c = (String) staff[i];
```

is a compile-time error because `String` is not a subclass of `Employee`.

To sum up:

- You can cast only within an inheritance hierarchy.
- Use `instanceof` to check before casting from a superclass to a subclass.

Note: The test

```
x instanceof C
```

does not generate an exception if `x` is `null`. It simply returns `false`. That makes sense: `null` refers to no object, so it certainly doesn't refer to an object of type `C`.

Actually, converting the type of an object by a cast is not usually a good idea. In our example, you do not need to cast an `Employee` object to a `Manager` object for most purposes. The `getSalary` method will work correctly on both objects of both classes. The dynamic binding that makes polymorphism work locates the correct method automatically.

The only reason to make the cast is to use a method that is unique to managers, such as `setBonus`. If for some reason you find yourself wanting to call `setBonus` on `Employee` objects,

ask yourself whether this is an indication of a design flaw in the superclass. It may make sense to redesign the superclass and add a `setBonus` method. Remember, it takes only one uncaught `ClassCastException` to terminate your program. In general, it is best to minimize the use of casts and the `instanceof` operator.

C++ Note: Java uses the cast syntax from the "bad old days" of C, but it works like the safe `dynamic_cast` operation of C++. For example,

```
Employee[] staff = ...;
Manager boss = (Manager) staff[i]; // Java
```

is equivalent to

```
Employee* staff[] = ...;
Manager* boss = dynamic_cast<Manager*>(staff[i]); // C++
```

There is one important difference. If the `dynamic_cast` fails, it yields a null pointer instead of throwing an exception.

5.1.9. Pattern Matching for `instanceof`

The code

```
if (staff[i] instanceof Manager)
{
   Manager boss = (Manager) staff[i];
   boss.setBonus(5000);
}
```

is rather verbose. Do we really need to mention the subclass `Manager` three times?

As of Java 16, there is an easier way. You can declare the subclass variable right in the `instanceof` test:

```
if (staff[i] instanceof Manager boss)
{
   boss.setBonus(5000);
}
```

If `staff[i]` is an instance of the `Manager` class, then the variable `boss` is set to `staff[i]`, and you can use it *as a* `Manager`. You skip the cast.

If `staff[i]` doesn't refer to a `Manager`, `boss` is not set, and the `instanceof` operator yields the value `false`. The body of the `if` statement is skipped.