1 Java class design

Exam objectives covered in this chapter	What you need to know
[1.1] Use access modifiers: private, protected, and public	How to use appropriate access modifiers to design classes How to limit accessibility of classes, interfaces, enums, methods, and variables by using the appropriate access modifiers The correct combination of access modifiers and the entities (classes, interfaces, enums, methods, and variables) to which they can be applied The implications of modifying the access modifier of a Java entity
[1.2] Override methods	The conditions and requirements that make a subclass override a base class method How to differentiate among overloaded, overridden, and hidden methods
[1.3] Overload constructors and methods	The need and right rules to overload constructors and methods
[1.4] Use the `instanceof` operator and casting	Understand the right use of the `instanceof` operator, and implicit and explicit object casting and their implications Compilation errors and runtime exceptions associated with the use of the `instanceof` operator and casting
[1.5] Use virtual method invocation	The methods that can and can't be invoked virtually

Exam objectives covered in this chapter	What you need to know
[1.6] Override methods from the `Object` class to improve the functionality of your class	The need to override methods from class `Object`—differentiate correct, incorrect, appropriate, and inappropriate overriding
[1.7] Use package and import statements	How to package classes and use `package`, `import`, and `static import` statements

Classes and interfaces are building blocks of an application. Efficient and effective class design makes a significant impact on the overall application design. Imagine if, while designing your classes, you didn't consider effective packaging, correct overloaded or overridden methods, or access protection—you might lose on extensibility, flexibility, and usability of your classes. For example, if you didn't override methods `hashCode()` and `equals()` correctly in your classes, your *seemingly* "equal" objects might *not* be considered equal by collection classes like `HashSet` or `HashMap`. Or, say, imagine if you didn't use the right access modifiers to protect your classes and their members, they could be subject to unwanted manipulation by other classes from the same or different packages. The creation of overloaded methods is another domain, which is an important class design decision. It eases instance creation and use of methods.

Class design decisions require an insight into understanding correct and appropriate implementation practices. When armed with adequate information you'll be able to select the best practices and approach to designing your classes. The topics covered in this chapter will help you design better classes by taking you through multiple examples. This chapter covers

- Access modifiers
- Method overloading
- Method overriding
- Virtual method invocation
- Use of the `instanceof` operator and casting
- Override methods from class `Object` to improve the functionality of your class
- How to create packages and use classes from other packages

Let's get started with how to control access to your classes and their members, using access modifiers.

1.1 Java access modifiers

[1.1] Use access modifiers: private, protected, and public

When you design applications and create classes, you need to answer multiple questions:

- How do I restrict other classes from accessing certain members of a class?
- How do I prevent classes from modifying the state of objects of a class, both within the same and separate packages?

Java access modifiers answer all these questions. Access modifiers control the accessibility of a class or an interface, including its members (methods and variables), by other classes and interfaces within the same or separate packages. By using the appropriate access modifiers, you can limit access to your class or interface, and its members.

Access modifiers can be applied to classes, interfaces, and their members (instance and class variables and methods). Local variables and method parameters can't be defined using access modifiers. An attempt to do so will prevent the code from compiling.

In this section, we'll cover all of the access modifiers—`public`, `protected`, and `private`—as well as *default* access, which is the result when you don't use an access modifier. You'll also discover the effects of changing the access levels of existing types on other code.

NOTE Access modifiers are also covered in the OCA Java SE 7 Programmer I exam (1Z0-803). If you've written this exam recently, then perhaps you might like to skip sections 1.1.1–1.1.4.

To understand all of these access modifiers, we'll use the same set of classes: `Book`, `CourseBook`, `Librarian`, `StoryBook`, and `House`. Figure 1.1 depicts these classes using UML notation.

Classes `Book`, `CourseBook`, and `Librarian` are defined in the package `library`. Classes `StoryBook` and `House` are defined in the package `building`. Classes `StoryBook` and `CourseBook` (defined in separate packages) extend class `Book`. Using these classes,

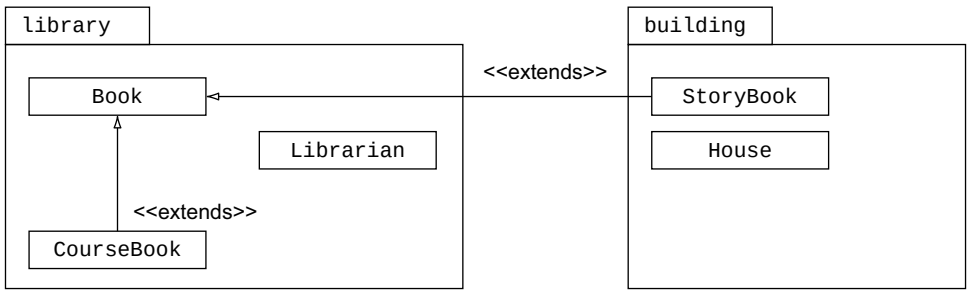

Figure 1.1 A set of classes and their relationships to help understand access modifiers

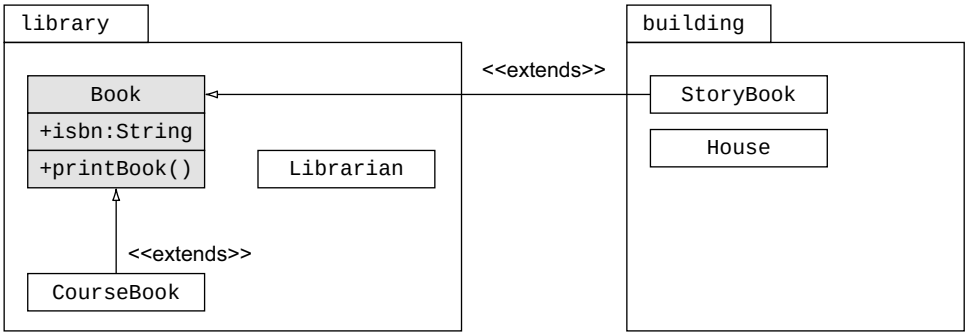

Figure 1.2 Understanding the `public` access modifier

you'll see how the accessibility of a class and its members varies with different access modifiers, from unrelated to derived classes, across packages.

As we cover each of the access modifiers, we'll add a set of instance variables and a method to class `Book` with the relevant access modifier.

1.1.1 Public access modifier

This is the least restrictive access modifier. Classes and interfaces defined using the `public` access modifier are accessible across all packages, from derived to unrelated classes.

To understand the `public` access modifier, let's define class `Book` as a public class and add a public instance variable (`isbn`) and a public method (`printBook()`) to it. Figure 1.2 shows the UML notation.

Examine the following definition of class `Book`:

```
package library;
public class Book {
    public String isbn;
    public void printBook() {}
}
```

public class Book

public variable isbn

public method printBook()

The `public` access modifier is said to be the least restrictive, so let's try to access the public class `Book` and its public members from class `House`. We'll use class `House` because `House` and `Book` are defined in separate packages and they're unrelated. Class `House` doesn't enjoy any advantages of being defined in the same package or being a derived class.

Here's the code for class `House`:

```
package building;
import library.Book;
public class House {
```

```
public House() {
    Book book = new Book();
    String value = book.isbn;
    book.printBook();
}
}
```

Book is accessible to House

isbn is accessible to House

printBook() is accessible to House

In the preceding example, class `Book` and its public members—instance variable `isbn` and method `printBook()`—are accessible to class `House`. They're also accessible to the other classes: `StoryBook`, `Librarian`, and `CourseBook`. Figure 1.3 shows the classes that can access a public class and its members.

1.1.2 Protected access modifier

The members of a class defined using the `protected` access modifier are accessible to

- Classes and interfaces defined in the same package
- All derived classes, even if they're defined in separate packages

EXAM TIP Members of an interface are implicitly public. If you define interface members as protected, the interface won't compile.

Let's add a protected instance variable `author` and method `modifyTemplate()` to class `Book`. Figure 1.4 shows the class representation.

Figure 1.3 Classes that can access a public class and its members

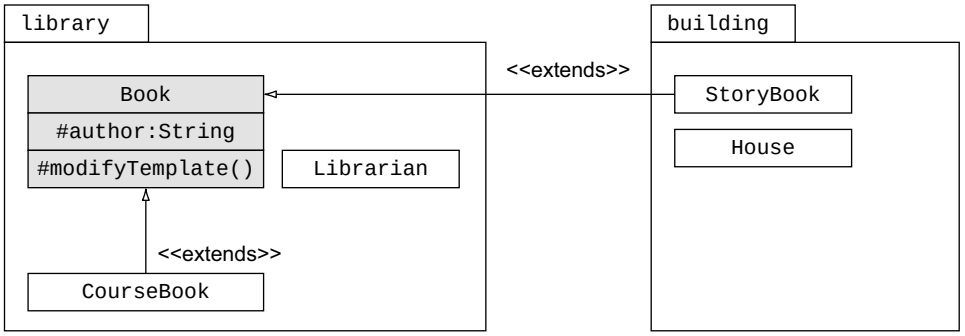

Figure 1.4 Understanding the `protected` access modifier

Here's the code for class `Book` (I've deliberately left out its public members because they aren't required in this section):

```
package library;
public class Book {
    protected String author;
    protected void modifyTemplate(){}
}
```

protected variable author

protected method modifyTemplate()

Figure 1.5 illustrates how classes from the same and separate packages, derived classes, and unrelated classes access class `Book` and its protected members.

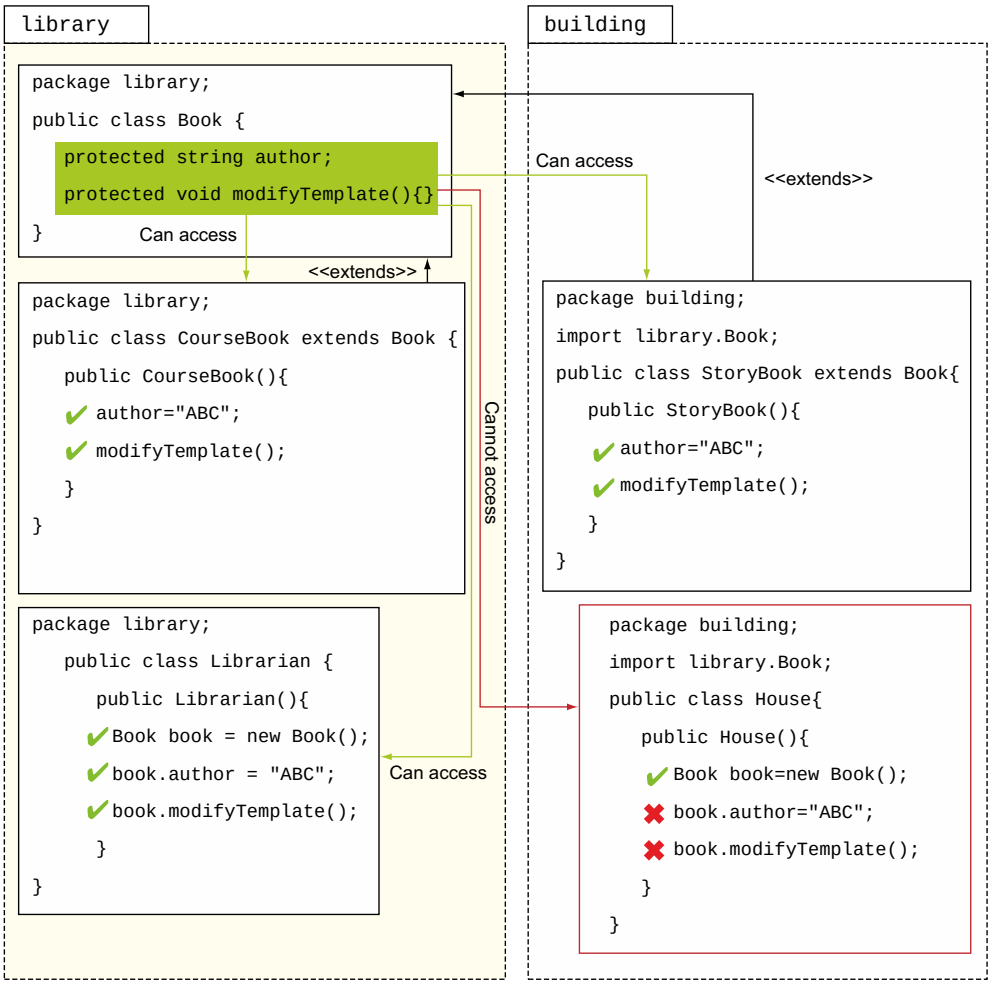

Figure 1.5 Access of protected members of class `Book` in unrelated and derived classes, from the same and separate packages

Class `House` fails compilation for trying to access method `modifyTemplate()`and variable `author`, as follows:

```
House.java:8: modifyTemplate()has protected access in library.Book
        book.modifyTemplate();
             ^
```

A derived class inherits the protected members of its base class, irrespective of the packages in which they are defined.

Notice that the derived classes `CourseBook` and `StoryBook` inherit class `Book`'s protected member variable `author` and method `modifyTemplate()`. If class `StoryBook` tries to instantiate `Book` using a reference variable and then tries to access its protected variable `author` and method `modifyTemplate()`, it won't compile:

```
package building;
import library.Book;
class StoryBook extends Book {
    StoryBook () {
        Book book = new Book();
        String v = book.author;
        book.modifyTemplate();
    }
}
```

Book and StoryBook defined in separate packages

Protected members of Book aren't accessible in StoryBook if accessed using an instance of Book.

EXAM TIP A concise but not too simple way of stating the previous rule is this: A derived class can inherit and access protected members of its base class, regardless of the package in which it's defined. A derived class in a separate package can't access protected members of its base class using reference variables.

Figure 1.6 shows the classes that can access protected members of a class or an interface.

1.1.3 Default access (package access)

The members of a class defined without using any explicit access modifier are defined with *package accessibility* (also called *default accessibility*). The members with package access are *only* accessible to classes and interfaces defined in the same package. The default access is also referred to as *package-private*. Think of a package as your home, classes as rooms, and things in rooms as variables with default access. These things

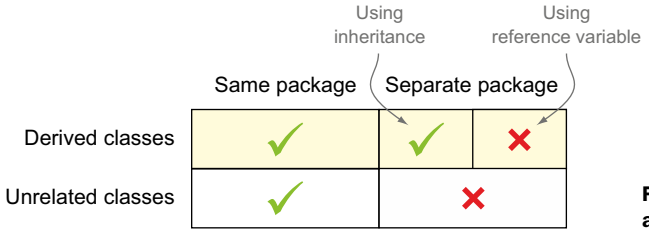

Figure 1.6 Classes that can access protected members

aren't limited to one room—they can be accessed across all the rooms in your home. But they're still private to your home—you wouldn't want them to be accessed outside your home. Similarly, when you define a package, you might want to make accessible members of classes to all the other classes across the same package.

NOTE While the package-private access is as valid as the other access levels, in real projects, it often appears as the result of inexperienced developers forgetting to specify the access modifier of Java components.

Let's define an instance variable `issueCount` and a method `issueHistory()`with default access in class `Book`. Figure 1.7 shows the class representation with these new members.

Here's the code for class `Book` (I've deliberately left out its public and protected members because they aren't required in this section):

```
package library;
public class Book {
    int issueCount;
    void issueHistory () {}
}
```

public class Book

issueCount with default access

issueHistory() with default access

You can see how classes from the same package and separate packages, derived classes, and unrelated classes access class `Book` and its members (instance variable `issueCount` and method `issueHistory()`) in figure 1.8.

Because classes `CourseBook` and `Librarian` are defined in the same package as class `Book`, they can access the members `issueCount` and `issueHistory()`. Because classes `House` and `StoryBook` aren't defined in the same package as class `Book`, they can't access the members `issueCount` and `issueHistory()`. Class `StoryBook` fails compilation with the following error message:

```
StoryBook.java:6: issueHistory () is not public in library.Book; cannot be
     accessed from outside package
        book.issueHistory ();
            ^
```

Figure 1.7 Understanding class representations for the default access

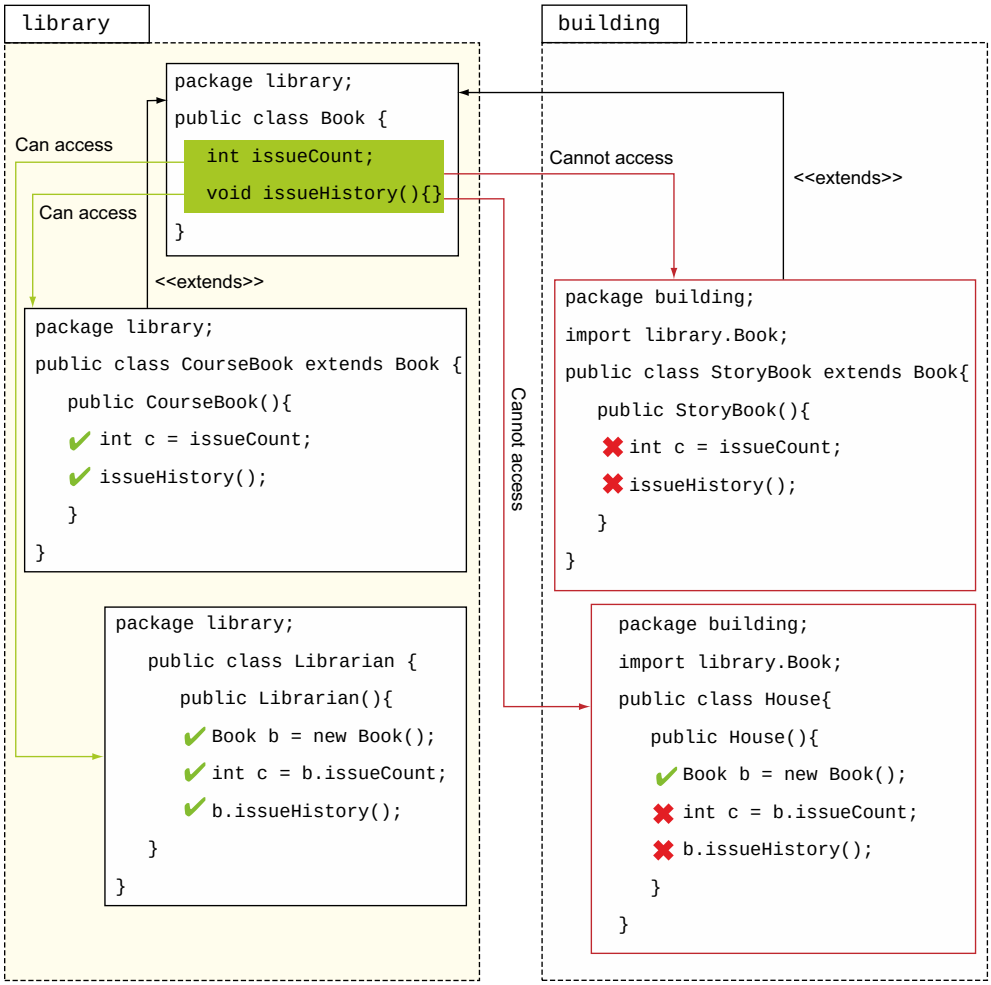

Figure 1.8 Access of members with default access to class Book in unrelated and derived classes from the same and separate packages

Class `House` is unaware of the existence of `issueHistory()`—it fails compilation with the following error message:

```
House.java:9: cannot find symbol
symbol  : method issueHistory ()
location: class building.House
        issueHistory ();
```

Defining a class Book with default access

What happens if you define a class with default access? What will happen to the accessibility of its members if the class itself has default (package) accessibility?

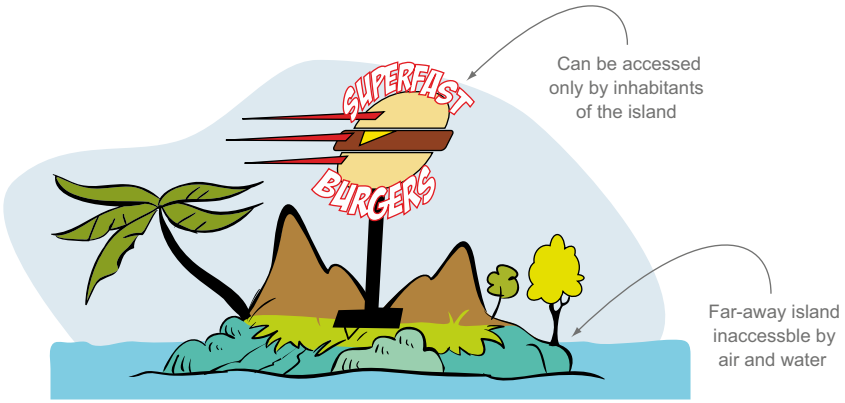

Figure 1.9 This Superfast Burgers cannot be accessed from outside the island because the island is inaccessible by air and water.

Consider this situation: Assume that Superfast Burgers opens a new outlet on a beautiful island and offers free meals to people from all over the world, which obviously includes inhabitants of the island. But the island is inaccessible by all means (air and water). Would the existence of this particular Superfast Burger outlet make any sense to people who don't inhabit the island? An illustration of this example is shown in figure 1.9.

The island is like a package in Java, and the Superfast Burger outlet is like a class defined with default access. In the same way that the Superfast Burger outlet can't be accessed from outside the island on which it exists, a class defined with default (package) access is visible and accessible only within the package in which it's defined. It can't be accessed from outside its package.

Let's redefine class `Book` with default (package) access as follows:

```
package library;
class Book {
    //.. class members
}
```

Book now has default access

The behavior of class `Book` remains the same for classes `CourseBook` and `Librarian`, which are defined in the same package. But class `Book` can't be accessed by classes `House` and `StoryBook`, which reside in a separate package.

Let's start with class `House`. Examine the following code:

```
package building;
import library.Book;
public class House {}
```

Book isn't accessible in House

Figure 1.10 Classes that can access members with default (package) access

Class `House` fails compilation with the following error message:

```
House.java:2: library.Book is not public in library; cannot be accessed from
     outside package
import library.Book;
```

Here's the code of class `StoryBook`:

```
package building;
import library.Book;
class StoryBook extends Book {}
```

Book isn't accessible in StoryBook

StoryBook cannot extend Book

Figure 1.10 shows which classes can access members of a class or an interface with default (package) access.

Because a lot of programmers are confused about which members are made accessible by using the `protected` access modifier and no modifier (default), the following exam tip offers a simple and interesting rule to help you remember their differences.

EXAM TIP Default access can be compared to package-private (accessible only within a package) and protected access can be compared to package-private + *kids* (kids refers to derived classes). Kids can access protected members only by inheritance and not by reference (accessing members by using the dot operator on an object).

1.1.4 The private access modifier

The `private` access modifier is the most restrictive access modifier. The members of a class defined using the `private` access modifier are accessible only to them. For example, the internal organs of your body (heart, lungs, etc.) are private to your body. No one else can access them. It doesn't matter whether the class or interface in question is from another package or has extended the class—private members *aren't* accessible outside the class in which they're defined.

EXAM TIP Members of an interface are implicitly public. If you define interface members as private, the interface won't compile.

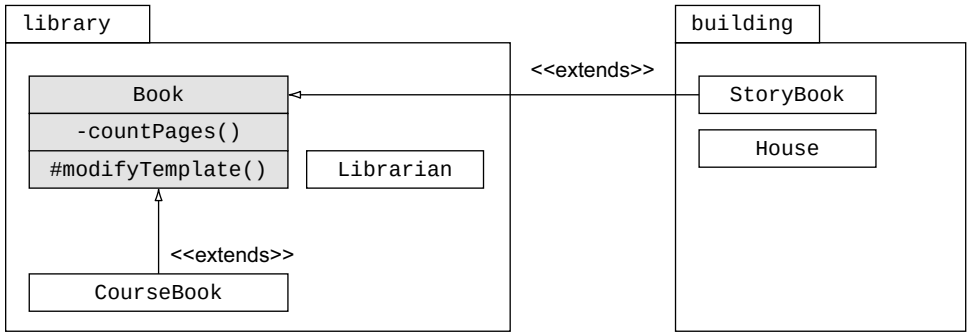

Figure 1.11 Understanding the `private` access modifier

Let's see the private members in action by adding a private method `countPages()` to class `Book`. Figure 1.11 depicts the class representation using UML.

Examine the following definition of class `Book`:

```
package library;
class Book {
    private void countPages () {}
    protected void modifyTemplate(){
        countPages ();
    }
}
```

Private method

Only Book can access its own private method countPages()

None of the classes defined in any of the packages (whether derived or not) can access the private method `countPages()`. But let's try to access it from class `CourseBook`. I chose class `CourseBook` because both of these classes are defined in the same package, and class `CourseBook` extends class `Book`. Here's the code of `CourseBook`:

```
package library;
class CourseBook extends Book {
    CourseBook () {
        countPages ();
    }
}
```

CourseBook extends Book

CourseBook cannot access private method countPages()

Because class `CourseBook` tries to access private members of class `Book`, it will not compile. Similarly, if any of the other classes (`StoryBook`, `Librarian`, or `House`) try to access private method `countPages()` of class `Book`, it will not compile. Figure 1.12 shows the classes that can access the private members of a class.

NOTE For your real projects, it *is* possible to access private members of a class outside them, using *Java Reflection*. But Java Reflection isn't on the exam. So don't consider it when answering questions on the accessibility of private members.

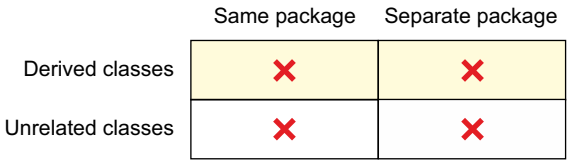

Figure 1.12 No classes can access private members of another class.

1.1.5 Access modifiers and Java entities

Can every access modifier be applied to all the Java entities? The simple answer is *no*. Table 1.1 lists the Java entities and the access modifiers that can be used with them.

Table 1.1 Java entities and the access modifiers that can be applied to them

Entity name	`public`	`protected`	`private`
Top-level class, interface, enum	✓	✗	✗
Nested class, interface, enum	✓	✓	✓
Class variables and methods	✓	✓	✓
Instance variables and methods	✓	✓	✓
Method parameters and local variables	✗	✗	✗

What happens if you try to code the combinations for an X above? None of these combinations will compile. Here's the code:

```
protected class MyTopLevelClass {}
private class MyTopLevelClass {}
protected interface TopLevelInterface {}
protected enum TopLevelEnum {}
```

Won't compile—top-level class, interface, and enums can't be defined with protected and private access.

```
void myMethod(private int param) {}
void myMethod(int param) {
    public int localVariable = 10;
}
```

Won't compile—method parameters and local variables can't be defined using any explicit access modifiers.

Watch out for these combinations on the exam. It's simple to insert these small and invalid combinations in any code snippet and still make you believe that you're being tested on a rather complex topic like threads or concurrency.

EXAM TIP Watch out for invalid combinations of a Java entity and an access modifier. Such code won't compile.

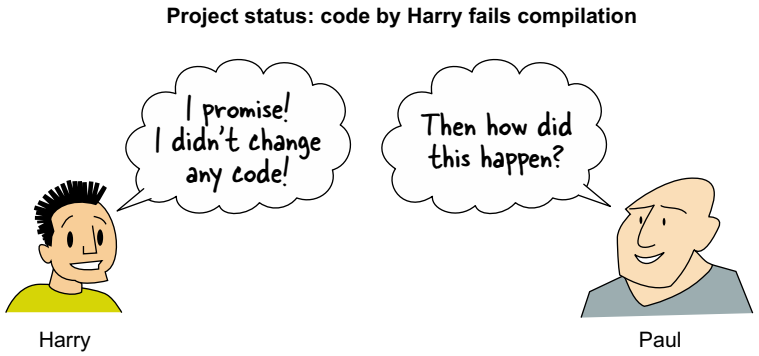

Figure 1.13 A change in the access modifier of a member of a class can break the code of other classes.

1.1.6 Effects of changing access modifiers for existing entities

Shreya, a programmer, changed the access modifier of a member in her class, `Book`, and see what Harry (another programmer) had to go through the next morning (figure 1.13).

Let's analyze what happened. Why did Harry's code break when Shreya changed her own code? As shown in figure 1.14, Harry's class `StoryBook` extends class `Book` created by Shreya. Before the modifications, Harry's class `StoryBook` accessed the protected member `author` from its parent class `Book`. But when Shreya modified the access modifier of the member `author` from `protected` to default access, it could no longer be accessed by class `StoryBook` because they reside in separate packages. So, even though Harry didn't change his code, it didn't compile.

You can change the access modifier of a member in two ways:

- Accessibility is decreased—for example, a public member is made private
- Accessibility is increased—for example, a private member is made public

WHEN ACCESSIBILITY OF AN ENTITY IS DECREASED (MORE RESTRICTIVE)

As shown in figure 1.14, when an entity is made more restrictive, there are chances that other code that uses that entity *might* break.

> **Impact of decreasing accessibility in real-life projects**
>
> Decreasing the accessibility of entities can affect the overall application in a big way. This is especially important for designing APIs and maintaining software. Many Java developers make the mistake of carelessly decreasing the accessibility of methods or fields, which can result in access issues with other components in a system.

WHEN ACCESSIBILITY OF AN ENTITY IS INCREASED (LESS RESTRICTIVE)

There are no issues when an entity is made less restrictive, say, when access of an entity is changed from default to `protected` or `public`. With increased access, an

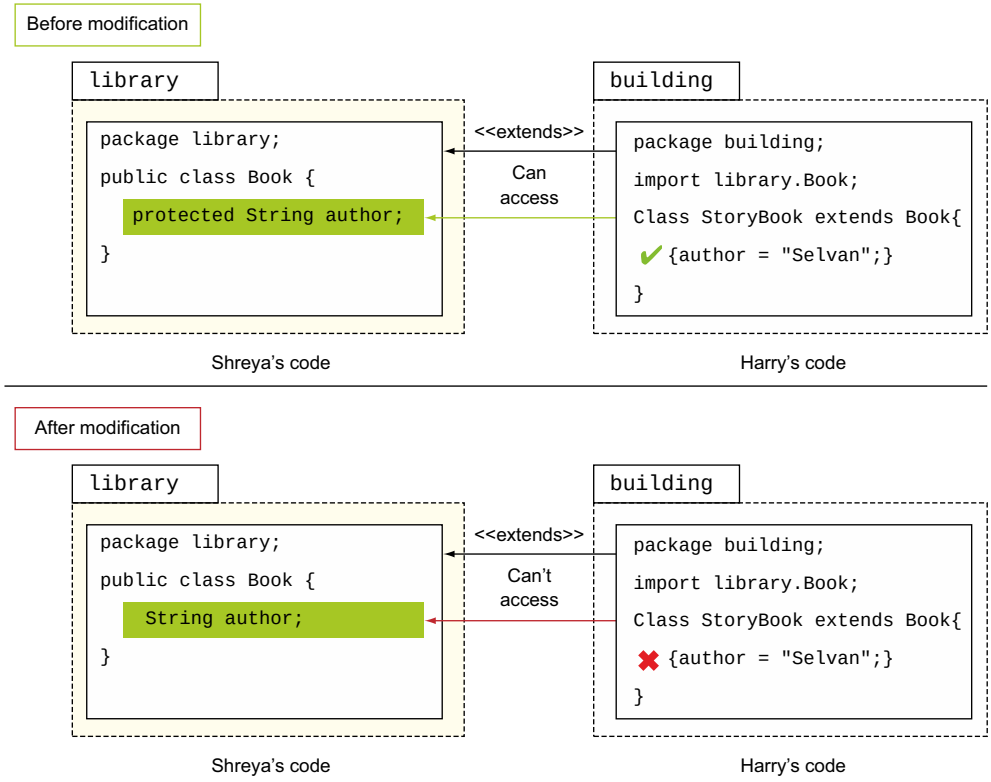

Figure 1.14 Code before and after modification showing why Harry's code failed to compile, even though he didn't change a bit of it.

entity may become visible to other classes, interfaces, and enums to which it wasn't visible earlier.

Apart from being an important exam topic, you're sure to encounter issues related to access modifiers at your workplace in real projects. Let's see whether you can spot a similar issue in the first "Twist in the Tale" exercise.

About the "Twist in the Tale" exercises

For these exercises, I've tried to use modified code from the examples already covered in the chapter. The "Twist in the Tale" title refers to modified or tweaked code. These exercises will help you understand how even small code modifications can change the behavior of your code. They should also encourage you to carefully examine all of the code on the exam. The reason for these exercises is that on the exam, you may be asked more than one question that seems to require the same answer. But on closer inspection, you'll realize that the questions differ slightly, and this will change the behavior of the code and the correct answer option. All answers to "Twist in the Tale" exercises are in the appendix.

Twist in the Tale 1.1

Here are the classes written by Shreya and Harry (residing in separate source code files) that work without any issues:

```
package library;                  // Class written by Shreya
public class Book {
    protected String author;
}
```

```
package building;                 // Class written by Harry
import library.Book;
class StoryBook extends Book {
    { author = "Selvan"; }
}
```

On Friday evening, Shreya modified her code and checked it in to the organization's version control system. Do you think Harry would be able to run his code without any errors when he checks out the modified code on Monday morning, and why? Here's the modified code:

```
package library;                  // Class written by Shreya
class Book {
    protected String author;
}
```

```
package building;                 // Class written by Harry
import library.Book;
class StoryBook extends Book {
    { author = "Selvan"; }
}
```

In the next section, we'll cover the need and semantics of defining overloaded methods. You can compare overloaded methods with any action that you might specify with multiple, different, or additional details. Let's get started with understanding the need of defining overloaded methods.

1.2 Overloaded methods and constructors

[1.3] Overload constructors and methods

Overloaded methods are methods with the same name but different method parameter lists. In this section, you'll learn how to create and use overloaded methods.

Imagine that you're delivering a lecture and need to instruct the audience to take notes using paper, a Smartphone, or a laptop—whichever is available to them for the day. One way to do this is to give the audience a list of instructions like

- Take notes using paper.
- Take notes using Smartphones.
- Take notes using laptops.

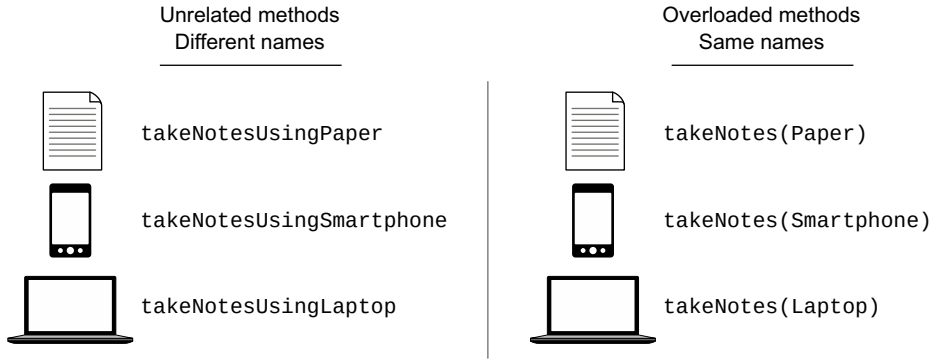

Figure 1.15 Real-life examples of overloaded methods

Another method is to instruct them to "take notes" and then provide them with the paper, a Smartphone, or a laptop they're supposed to use. Apart from the simplicity of the latter method, it also gives you the flexibility to add other media on which to take notes (such as one's hand, some cloth, or the wall) without needing to remember the list of all the instructions.

This second approach—providing one set of instructions (with the same name) but a different set of input values—can be compared to overloaded methods in Java, as shown in figure 1.15.

The implementation of the example shown in figure 1.15 in code is as follows:

```
class Paper {}
class Smartphone {}
class Laptop {}

class Lecture {
    void takeNotes(Paper paper) {}
    void takeNotes(Smartphone phone) {}
    void takeNotes(Laptop laptop) {}
}
```

Overloaded method—takeNotes()

Overloaded methods are usually referred to as methods that are defined in the same class, with the same name, but with a different method argument list. A derived class can also overload the methods inherited from its base class as follows:

```
class Paper {}
class Smartphone {}
class Laptop {}

class Lecture {
    void takeNotes(Paper paper) {}
    void takeNotes(Smartphone phone) {}
    void takeNotes(Laptop laptop) {}
}
class Canvas {}
class FineArtLecture extends Lecture {
    void takeNotes(Canvas canvas) {}
}
```

takeNotes() in FineArtLecture overloads takeNotes() from Lecture by specifying a different parameter list.

Overloaded methods make it easier to add methods with similar functionality that work with a different set of input values. Let's work with an example from the Java API classes that we all use frequently: `System.out.println()`. Method `println()` accepts multiple types of method parameters:

```
int intVal = 10;
boolean boolVal = false;
String name = "eJava";

System.out.println(intVal);
System.out.println(boolVal);
System.out.println(name);
```

Prints an int value

Prints a boolean value

Prints a string value

When you use method `println()`, you know that whatever you pass to it as a method argument will be printed to the console. Wouldn't it be crazy to use methods like `printlnInt()`, `printlnBool()`, and `printlnString()` for the same functionality? I think so, too.

Let's examine in detail the method parameters passed to overloaded methods, their return types, and their access and nonaccess modifiers.

NOTE The exam will test you on how you can define correct overloaded methods, which overloaded methods get invoked when you use a set of arguments, and also whether a compiler is unable to resolve the call.

1.2.1 Argument list

Overloaded methods accept different lists of arguments. The argument lists can differ in terms of

- The change in the number of parameters that are accepted
- The change in the type of the method parameters that are accepted
- The change in the positions of the parameters that are accepted (based on parameter type, not variable names)

Let's work with some examples to verify these points.

CHANGE IN THE NUMBER OF METHOD PARAMETERS

Overloaded methods that define a different number of method parameters are the simplest among all the method types. Let's work with an example of an overloaded method, `calcAverage()`, which accepts a different count of method parameters:

```
class Result {
    double calcAverage(int marks1, int marks2) {
        return (marks1 + marks2)/2;
    }
    double calcAverage(int marks1, int marks2, int marks3) {
        return (marks1 + marks2 + marks3)/3;
    }
}
```

Two method arguments

Three method arguments

CHANGE IN THE TYPE OF METHOD PARAMETERS

In the following example, the difference is in the argument list—due to the change in the type of parameters it accepts—to calculate the average of integer and decimal numbers:

```
class Result {
    double calcAverage(int marks1, double marks2) {
        return (marks1 + marks2)/2;
    }
    double calcAverage(double marks1, double marks2) {
        return (marks1 + marks2)/2;
    }
}
```

Arguments—int and double

Arguments—double and double

When you define overloaded methods with object references as parameters, their classes might or might not share an inheritance relationship. When the classes don't share an inheritance relationship, there isn't any confusion with the version of the method that will be called:

```
class Employee {}
class Engineer extends Employee {}
class CEO extends Employee {}
class Travel {
    static String bookTicket(Engineer val) {
        return "economy class";
    }
    static String bookTicket(CEO val) {
        return "business class";
    }
}
```

Engineer and CEO aren't in the same inheritance tree

For the preceding code, if you call method `bookTicket()` by passing it a `CEO` object, it will call the method that accepts a parameter of type `CEO`—no confusion here. Now, what happens if you define overloaded methods that accept object references of classes which share an inheritance relationship? For example (modifications in code are in **`bold`**)

```
class Employee {}
class CEO extends Employee {}
class Travel {
    static String bookTicket(Employee val) {
        return "economy class";
    }
    static String bookTicket(CEO val) {
        return "business class";
    }
}
```

Method parameters—CEO extends Employee

Which of these methods do you think would be called if you pass a `CEO` object to method `bookTicket()`? Can a `CEO` object be assigned to both `CEO` and `Employee`?

```
class TravelAgent {
    public static void main(String... args) {
        System.out.println(Travel.bookTicket(new CEO()));
    }
}
```

Prints "business class"

The preceding code calls overloaded method `bookTicket()`that accepts a `CEO`, because without any explicit reference variable, `new CEO()` is referred to using a `CEO` variable. Now, try to determine the output of the following code:

```
class TravelAgent {
    public static void main(String... args) {
        Employee emp = new CEO();
        System.out.println(Travel.bookTicket(emp));
    }
}
```

Prints "economy class"

The preceding code prints "economy class" and not "business class" because the type of the `reference` variable `emp` is `Employee`. The overloaded methods are bound at compile time and not runtime. To resolve the call to the overloaded methods, the compiler considers the type of variable that's used to refer to an object.

EXAM TIP Calls to the overloaded methods are resolved during compilation.

Using the preceding `Employee` and `CEO` example, figure 1.16 shows a fun way to remember calls to the overloaded methods are resolved during compilation.

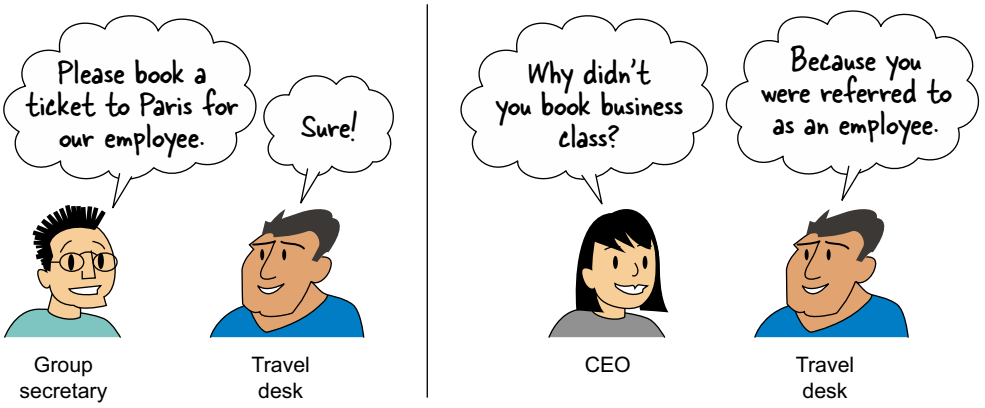

Figure 1.16 Overloaded methods are resolved during compilation.

For the overloaded method `bookTicket()` that defines the method parameter of either `Engineer` or `CEO`, watch out for exam questions that try to call it using a reference variable of `Employee`:

```
class Employee {}
class Engineer extends Employee {}
class CEO extends Employee {}
class Travel {
    static String bookTicket(Engineer val) {
        return "economy class";
    }
    static String bookTicket(CEO val) {
        return "business class";
    }
    public static void main(String args[]) {
        Employee emp = new CEO();
        System.out.println(bookTicket(emp));
    }
}
```

Accepts Engineer

Accepts CEO

Won't compile—Travel doesn't define method that accepts Employee

Change in the positions of method parameters

The methods are correctly overloaded if they only change the positions of the parameters that are passed to them, as follows:

```
double calcAverage(double marks1, int marks2) {
    return (marks1 + marks2)/2;
}
double calcAverage(int marks1, double marks2) {
    return (marks1 + marks2)/2;
}
```

Arguments—double and int

Arguments—int and double

Although you might argue that the arguments being accepted are the same, with only a difference in their positions, the Java compiler treats them as different argument lists. Therefore, the previous code is a valid example of overloaded methods. But an issue arises when you try to execute this method using values that can be passed to both versions of the overloaded method. In this case, the code in method `main()` will fail to compile:

```
class MyClass {
    static double calcAverage(double marks1, int marks2) {
        return(marks1 + marks2)/2;
    }
    static double calcAverage(int marks1, double marks2) {
        return(marks1 + marks2)/2;
    }
    public static void main(String[] args) {
        calcAverage(2, 3);
    }
}
```

❶ **Method parameters—double and int**

❷ **Method parameters—int and double**

❸ **Compiler can't determine overloaded calcAverage() that should be called**

In the previous code, ❶ defines the `calcAverage()` method, which accepts two method parameters: a `double` and an `int`. The code at ❷ defines overloaded method

`calcAverage()`, which accepts two method parameters: first an `int` and then a `double`. Because an `int` literal value can be passed to a variable of type `double`, literal values `2` and `3` can be passed to both overloaded methods, declared at ❶ and ❷. Because this *method call* is dubious, the code at ❸ fails to compile, with the following message:

```
MyClass.java:10: error: reference to calcAverage is ambiguous, both method
calcAverage(double,int) in MyClass and method calcAverage(int,double) in
MyClass match
           calcAverage(2, 3);
          ^
1 error
```

EXAM TIP For primitive method arguments, if a *call* to an overloaded method is dubious, the code won't compile.

Here's an interesting question: Would an overloaded method with the following signature solve this specific problem?

```
static double calcAverage(int marks1, int marks2)
```

Yes, it will. Because the type of literal integer value is `int`, the compiler will be able to resolve the call `calcAverage(2, 3)` to `calcAverage(int marks1, int marks2)` and compile successfully.

1.2.2 When methods can't be defined as overloaded methods

The overloaded methods give you the flexibility of defining methods with the same name that can be passed a different set of arguments. But it doesn't make sense to define overloaded methods with a difference in only their return types or access or nonaccess modifiers.

RETURN TYPE

Methods can't be defined as overloaded methods if they only differ in their return types, as follows:

```
class Result {
    double calcAverage(int marks1, int marks2) {
        return (marks1 + marks2)/2;
    }
    int calcAverage(int marks1, int marks2) {
        return (marks1 + marks2)/2;
    }
}
```

Return type of calcAverage() is double

Return type of calcAverage() is int

The methods defined in the preceding code aren't correctly overloaded methods—they won't compile.

EXAM TIP When the Java compiler differentiates methods, it doesn't consider their return types. So you can't define overloaded methods with the same parameter list and different return types.

ACCESS MODIFIER

Methods can't be defined as overloaded methods if they only differ in their access modifiers, as follows:

```
class Result {
    public double calcAverage(int marks1, int marks2) {
        return (marks1 + marks2)/2;
    }
    protected double calcAverage(int marks1, int marks2) {
        return (marks1 + marks2)/2;
    }
}
```

Access—public

Access—protected

NONACCESS MODIFIER

Methods can't be defined as overloaded methods if they only differ in their nonaccess modifiers, as follows:

```
class Result {
    public synchronized double calcAverage(int marks1, int marks2) {
        return (marks1 + marks2)/2;
    }
    public final double calcAverage(int marks1, int marks2) {
        return (marks1 + marks2)/2;
    }
}
```

Nonaccess modifier—synchronized

Nonaccess modifier—final

Let's revisit the rules for defining overloaded methods.

Rules to remember for defining overloaded methods

Here's a quick list of rules to remember for the exam for defining and using overloaded methods:

- A class can overload its own methods and methods inherited from its base class.
- Overloaded methods must be defined with the same name.
- Overloaded methods must be defined with different parameter lists.
- Overloaded methods might define a different return type or access or nonaccess modifier, but they can't be defined with only a change in their return types or access or nonaccess modifiers.

In the next section, we'll create overloaded versions of special methods, called *constructors*, which are used to create objects of a class.

1.2.3 Overloaded constructors

While creating instances of a class, you might need to assign default values to some of its variables and assign explicit values to the rest. You can do so by overloading the

constructors. *Overloaded constructors* follow the same rules as discussed in the previous section on overloaded methods:

- Overloaded constructors must be defined using a different argument list.
- Overloaded constructors can't be defined by a mere change in their access modifiers.

EXAM TIP Watch out for exam questions that use nonaccess modifiers with constructors.

Using nonaccess modifiers with constructors is illegal—the code won't compile. Here's an example of class `Employee`, which defines four overloaded constructors:

```
class Employee {
    String name;
    int age;
    Employee() {
        name = "John";
        age = 25;
    }
    Employee(String newName) {
        name = newName;
        age = 25;
    }
    Employee(int newAge, String newName) {
        name = newName;
        age = newAge;
    }
    Employee(String newName, int newAge) {
        name = newName;
        age = newAge;
    }
}
```

❶ **No-argument constructor**

❷ **Constructor with one String argument**

❸ **Constructor with two arguments—int and String**

❹ **Constructor with two arguments—String and int**

In the previous code, the code at ❶ defines a constructor that doesn't accept any arguments, and the code at ❷ defines another constructor that accepts a single argument. Note the constructors defined at ❸ and ❹. Both of these accept two arguments, `String` and `int`. But the placement of these two arguments is different in ❸ and ❹, which is acceptable and valid for overloaded constructors and methods.

INVOKING AN OVERLOADED CONSTRUCTOR FROM ANOTHER CONSTRUCTOR

It's common to define multiple constructors in a class. Unlike overloaded methods, which can be invoked using the name of a method, overloaded constructors are invoked by using the keyword `this`—an implicit reference, accessible to an object, to refer to itself. For instance

```
class Employee {
    String name;
    int age;
    Employee() {
        this(null, 0);
    }
```

❶ **No-argument constructor**

❷ **Invokes constructor that accepts two arguments**

```
    Employee(String newName, int newAge) {
        name = newName;
        age = newAge;
    }
}
```

❸ **Constructor that accepts two arguments**

The code at ❶ defines a no-argument constructor. At ❷, this constructor calls the overloaded constructor by passing to it values `null` and `0`. ❸ defines an overloaded constructor that accepts two arguments.

Because a constructor is defined using the name of its class, it's a common mistake to try to invoke a constructor from another constructor using the class's name:

```
class Employee {
    String name;
    int age;
    Employee() {
        Employee(null, 0);
    }
    Employee(String newName, int newAge) {
        name = newName;
        age = newAge;
    }
}
```

Won't compile—you can't invoke a constructor within a class by using the class's name.

Also, when you invoke an overloaded constructor using the keyword `this`, it must be the first statement in your constructor:

```
class Employee {
    String name;
    int age;
    Employee() {
        System.out.println("No-argument constructor");
        this(null, 0);
    }
    Employee(String newName, int newAge) {
        name = newName;
        age = newAge;
    }
}
```

Won't compile—call to overloaded constructor must be first statement in constructor.

That's not all: you can't call a constructor from any other method in your class. None of the other methods of class `Employee` can invoke its constructor.

Rules to remember for defining overloaded constructors

Here's a quick list of rules to remember for the exam for defining and using overloaded constructors:

- Overloaded constructors must be defined using different argument lists.
- Overloaded constructors can't be defined by just a change in the access modifiers.
- Overloaded constructors can be defined using different access modifiers.

(continued)

- A constructor can call another overloaded constructor by using the keyword `this`.
- A constructor can't invoke a constructor by using its class's name.
- If present, the call to another constructor must be the first statement in a constructor.

INSTANCE INITIALIZERS

Apart from constructors, you can also define an *instance initializer* to initialize the instance variables of your class. An instance initializer is a code block defined within a class, using a pair of `{}`. You can define multiple instance initializers in your class. *Each* instance initializer is invoked when an instance is created, in the order they're defined in a class. They're invoked before a class constructor is invoked.

Why do you think you need an instance initializer if you can initialize your instances using constructors? Multiple reasons exist:

- For a big class, it makes sense to place the variable initialization just after its declaration.
- *All* the initializers are invoked, irrespective of the constructor that's used to instantiate an object.
- Initializers can be used to initialize variables of anonymous classes that can't define constructors. (You'll work with anonymous classes in the next chapter.)

Here's a simple example:

```
class Pencil {
    public Pencil() {
        System.out.println("Pencil:constructor");
    }
    public Pencil(String a) {
        System.out.println("Pencil:constructor2");
    }
    {
        System.out.println("Pencil:init1");
    }
    {
        System.out.println("Pencil:init2");
    }

    public static void main(String[] args) {
        new Pencil();
        new Pencil("aValue");
    }
}
```

Added to both overloaded constructors

The output of the preceding code is

```
Pencil:init1
Pencil:init2
Pencil:constructor
```

```
Pencil:init1
Pencil:init2
Pencil:constructor2
```

The next "Twist in the Tale" exercise hides an important concept within its code, which you can get to know only if you try to compile and execute the modified code.

Twist in the Tale 1.2

Let's modify the definition of class `Employee` used in the section on overloaded constructors as follows:

```
class Employee {
    String name;
    int age;
    Employee() {
        this("Shreya", 10);
    }
    Employee (String newName, int newAge) {
        this();
        name = newName;
        age = newAge;
    }
    void print(){
        print(age);
    }
     void print(int age) {
         print();
     }
}
```

What is the output of this modified code, and why?

The instance initializer blocks are executed after an implicit or explicit call to the parent class's constructor:

```
class Instrument {
    Instrument() {
        System.out.println("Instrument:constructor");
    }
}
class Pencil extends Instrument {
    public Pencil() {
        System.out.println("Pencil:constructor");
    }
    {
        System.out.println("Pencil:instance initializer");
    }
    public static void main(String[] args) {
        new Pencil();
    }
}
```

Figure 1.17 The order of execution of constructors and instance initializers in parent and child classes

The output of the preceding code is

```
Instrument:constructor
Pencil:instance initializer
Pencil:constructor
```

Figure 1.17 shows a fun way of remembering the order of execution of a parent class constructor, instance initializers, and a class constructor. Paul, our programmer, was having a very hard time remembering the order of execution of all these code blocks. He literally had to stand upside down to get the order right.

EXAM TIP If a parent or child class defines static initializer block(s), they execute before all parent and child class constructors and instance initializers—first for the parent and then for the child class.

Now that you've seen how to create the overloaded variants of methods and constructors, let's dive deep into method overriding. These two concepts, overloading and overriding, seem to be confusing for a lot of programmers. Let's get started by clearing the cobwebs.

1.3 Method overriding and virtual method invocation

[1.2] Override methods

[1.5] Use virtual method invocation

Do you celebrate a festival or an event in exactly the same manner as celebrated by your parents? Or have your modified it? Perhaps you celebrate the same festivals and events, but in your *own* unique manner. In a similar manner, classes can inherit

behavior from other classes. But they can redefine the behavior that they inherit—this is also referred to as *method overriding*.

Method overriding is an object-oriented programming (OOP) language feature that enables a derived class to define a specific implementation of an existing base class method to extend its own behavior. A derived class can *override* an instance method defined in a base class by defining an instance method with the same method signature/method name and number and types of method parameters. Overridden methods are also synonymous with *polymorphic methods*. The *static* methods of a base can't be overridden, but they can be *hidden* by defining methods with the same signature in the derived class.

A method that can be overridden by a derived class is called a *virtual method*. But beware: Java has always shied away from using the term *virtual methods* and you will not find a mention of this term in Java's vocabulary. This term is used in other OO languages like C and C++. *Virtual method invocation* is the invocation of the correct overridden method, which is based on the type of the object referred to by an object reference and not by the object reference itself. It's determined at runtime, not at compilation time.

The exam will question you on the need for overridden methods; the correct syntax of overridden methods; the differences between overloaded, overridden, and hidden methods; common mistakes while overriding methods; and virtual method invocation. Let's get started with the need for overridden methods.

NOTE A base class method is referred to as the *overridden method* and the derived class method is referred to as the *overriding method*.

1.3.1 Need of overridden methods

In the same way we inherit our parents' behaviors but redefine some of the inherited behavior to suit our own needs, a derived class can inherit the behavior and properties of its base class but still be different in its own manner—by defining new variables and methods. A derived class can also choose to define a different course of action for its base class method by overriding it. Here's an example of class `Book`, which defines a method `issueBook()` that accepts `days` as a method parameter:

```
class Book {
    void issueBook(int days) {
        if (days > 0)
            System.out.println("Book issued");
        else
            System.out.println("Cannot issue for 0 or less days");
    }
}
```

Following is another class, `CourseBook`, which inherits class `Book`. This class needs to override method `issueBook()` because a `CourseBook` can't be issued if it's only for

reference. Also, a `CourseBook` can't be issued for 14 or more days. Let's see how this is accomplished by overriding method `issueBook()`:

```
class CourseBook extends Book {
    boolean onlyForReference;
    CourseBook(boolean val) {
        onlyForReference = val;
    }
    @Override
    void issueBook(int days) {
        if (onlyForReference)
            System.out.println("Reference book");
        else
            if (days < 14)
                super.issueBook(days);
            else
                System.out.println("days >= 14");
    }
}
```

1 Annotation— @Override

2 Overrides issueBook() in base class Book

3 Calls issueBook() defined in Book

The code at ❶ uses the annotation `@Override`, which notifies the compiler that this method overrides a base class method. Though optional, this annotation can come in very handy if you try to override a method incorrectly. The code at ❷ defines method `issueBook()` with the same name and method parameters as defined in class `Book`. The code at ❸ calls method `issueBook()` defined in class `Book`; however, it isn't mandatory to do so. It depends on whether the derived class wants to execute the same code as defined by the base class.

NOTE Whenever you intend to override methods in a derived class, use the annotation `@Override`. It will warn you if a method can't be overridden or if you're actually overloading a method rather than overriding it.

The following example can be used to test the preceding code:

```
class BookExample  {
    public static void main(String[] args) {
        Book b = new CourseBook(true);
        b.issueBook(100);
        b = new CourseBook(false);
        b.issueBook(100);
        b = new Book();
        b.issueBook(100);
    }
}
```

Prints "Reference book"

Prints "days >= 14"

b now refers to a Book instance

Prints "Book issued"

Figure 1.18 represents the compilation and execution process of class `BookExample`, as Step 1 and Step 2:

- Step 1: The compile time uses the reference type for the method check.
- Step 2: The runtime uses the instance type for the method invocation.

Now let's move on to how to correctly override a base class method in a derived class.

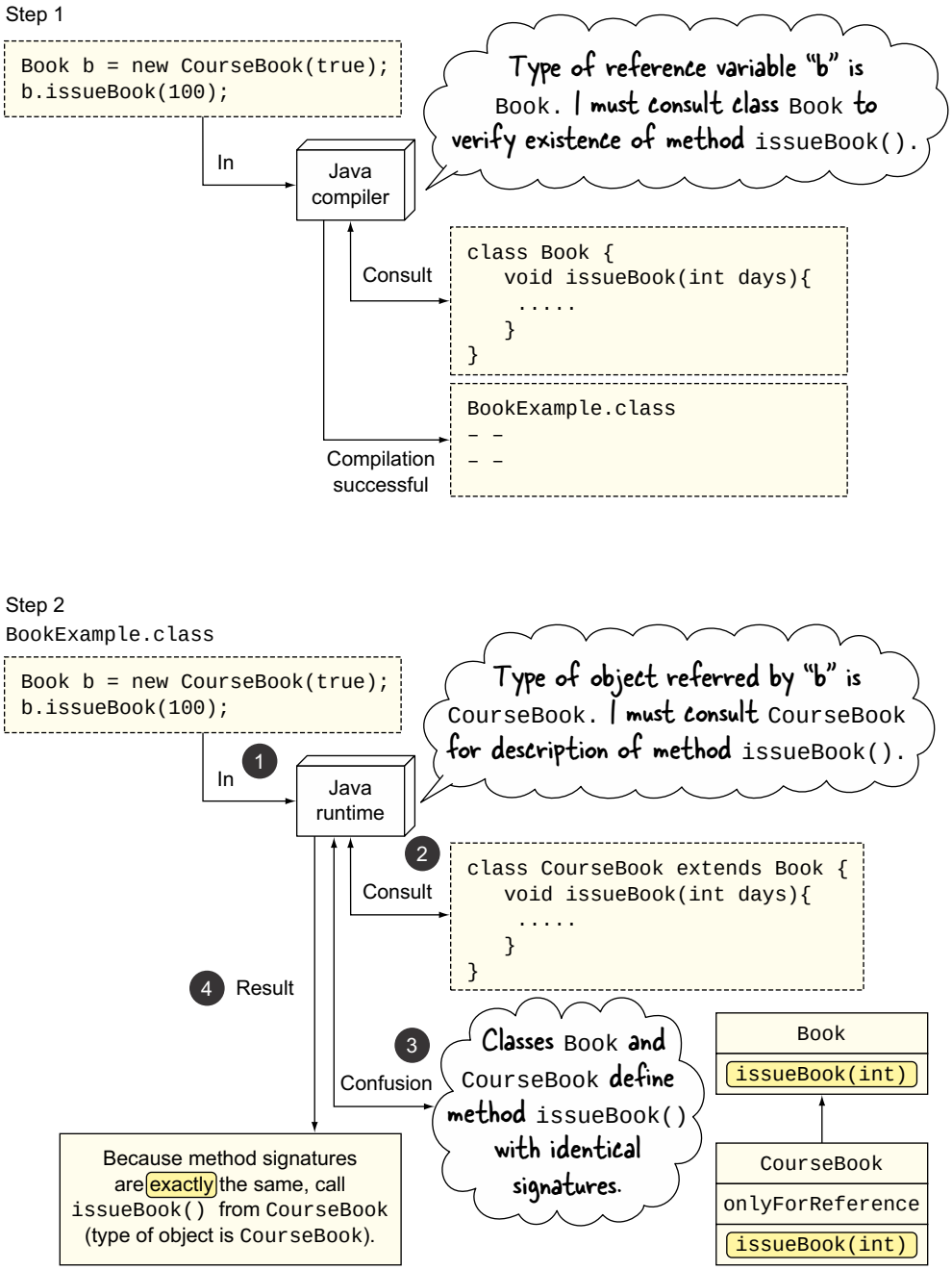

Figure 1.18 To compile `b.issueBook()`, the compiler refers only to the definition of class Book. To execute `b.issueBook()`, the Java Runtime Environment (JRE) uses the actual method implementation of `issueBook()` from class `CourseBook`.

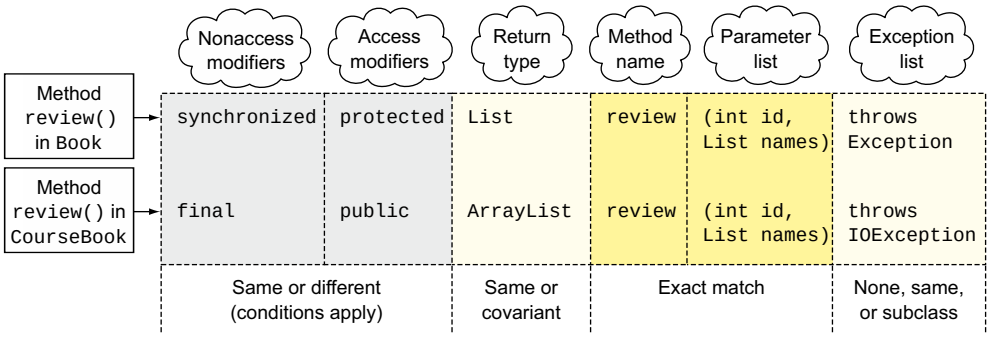

Figure 1.19 Comparing parts of a method declaration for a base class method and overriding method

1.3.2 Correct syntax of overriding methods

Let's start with an example of overridden method `review()`, as follows:

```
class Book {
    synchronized protected List review(int id,
                                       List names) throws Exception {
        return null;
    }
}
class CourseBook extends Book {
    @Override
    final public ArrayList review(int id,
                                  List names) throws IOException {
        return null;
    }
}
```

Method review() in base class Book

CourseBook extends Book

Overridden method review() in derived class CourseBook

Figure 1.19 shows the components of a method declaration: access modifiers, nonaccess modifiers, return type, method name, parameter list, and a list of exceptions that can be thrown (method declaration isn't the same as method signature). The figure also compares the `review` method defined in base class `Book` with overriding method `review()` defined in class `CourseBook` with respect to these identified parts.

Table 1.2 compares the method components shown in figure 1.19.

Table 1.2 Comparison of method components and their acceptable values for an overriding method

Method component	Value in class Book	Value in class CourseBook	Overriding method review() in class CourseBook
Access modifier	`protected`	`public`	Define same access or less restrictive access than method `review()` in the base class.

Table 1.2 Comparison of method components and their acceptable values for an overriding method

Method component	Value in class Book	Value in class CourseBook	Overriding method review() in class CourseBook
Nonaccess modifier	`synchronized`	`final`	Overriding method can use any nonaccess modifier for an overridden method. A nonabstract method can also be overridden to an abstract method. But a final method in the base class cannot be overridden. A static method cannot be overridden to be nonstatic.
Return type	`List`	`ArrayList`	Define the same or a subtype of the return type used in the base class method (covariant return types).
Method name	`review`	`review`	Exact match.
Parameter list	`(int id, List names)`	`(int id, List names)`	Exact match.
Exceptions thrown	`throws Exception`	`throws IOException`	Throw none, same, or a subclass of the exception thrown by the base class method.

EXAM TIP The rule listed in table 1.2 on exceptions in overriding methods only applies to checked exceptions. An overriding method can throw any unchecked exception (`RuntimeException` or `Error`) even if the overridden method doesn't. The unchecked exceptions aren't part of the method signature and aren't checked by the compiler.

Chapter 6 includes a detailed explanation on overridden and overriding methods that throw exceptions. Let's walk through a couple of invalid combinations that are important and very likely to be on the exam.

NOTE Though a best practice, I've deliberately not preceded the definition of the overriding methods with the annotation `@Override` because you might not see it on the exam.

ACCESS MODIFIERS

A derived class can assign the same or more access but not a weaker access to the overriding method in the derived class:

```
class Book {
    protected void review(int id, List names) {}
}
class CourseBook extends Book {
    void review(int id, List names) {}
}
```

Won't compile; overriding methods in derived classes can't use a weaker access.

NONACCESS MODIFIERS

A derived class can't override a base class method marked `final`:

```
class Book {
    final void review(int id, List names) {}
}
class CourseBook extends Book {
    void review(int id, List names) {}
}
```

Won't compile; final methods can't be overridden.

ARGUMENT LIST AND COVARIANT RETURN TYPES

When the overriding method returns a subclass of the return type of the overridden method, it's known as a *covariant return type*. To override a method, the parameter list of the methods in the base and derived classes must be *exactly* the same. It you try to use covariant types in the argument list, you'll end up overloading the methods and not overriding them. For example

```
class Book {
    void review(int id, List names) throws Exception {
        System.out.println("Base:review");
    }
}
class CourseBook extends Book {
    void review(int id, ArrayList names) throws IOException {
        System.out.println("Derived:review");
    }
}
```

(1) Argument list—int and List

Argument list—int and ArrayList

At (1) method `review()` in base class `Book` accepts an object of type `List`. Method `review()` in derived class `CourseBook` accepts a subtype `ArrayList` (`ArrayList` implements `List`). These methods aren't overridden—they're overloaded:

```
class Verify {
    public static void main(String[] args)throws Exception {
        Book book = new CourseBook();
        book.review(1, null);
    }
}
```

Calls review in Book; prints "Base:review"

(1) Reference variable of type Book used to refer to object CourseBook.

The code at (1) uses a reference variable of type `Book` to refer to an object of type `CourseBook`. The compilation process assigns execution of method `review()` from base class `Book` to the reference variable `book`. Because method `review()` in class `CourseBook` *doesn't* override the `review` method in class `Book`, the JRE doesn't have any confusion regarding whether to call method `review()` from class `Book` or from class `CourseBook`. It moves forward with calling `review()` from `Book`.

EXAM TIP It's the reference variable type that dictates which overloaded method will be chosen. This choice is made at compilation time.

EXCEPTIONS THROWN

An overriding method must either declare to throw no exception, the same exception, or a subtype of the exception declared to be thrown by the base class method, or else it will fail to compile. This rule, however, doesn't apply to error classes or runtime exceptions. For example

```
class Book {
    void review() throws Exception {}
    void read() throws Exception {}
    void close() throws Exception {}
    void write() throws NullPointerException {}
    void skip() throws IOException {}
    void modify() {}
}
class CourseBook extends Book {
    void review() {}
    void read() throws IOException {}
    void close() throws Error {}
    void write() throws RuntimeException {}
    void skip() throws Exception {}
    void modify() throws IOException {}
}
```

Compiles; declares to throw no exception.

Compiles; declares to throw IOException, a subclass of Exception.

Compiles; an overriding method can declare to throw any Error.

Compiles; an overriding method can declare to throw any RuntimeException.

Doesn't compile; declares to throw Exception, a superclass of IOException. Overriding method can't declare to throw broader exceptions than declared to be thrown by overridden method.

Doesn't compile; declares to throw IOException. Overriding method can't declare to throw a checked exception if overridden method doesn't.

EXAM TIP An overriding method can declare to throw *any* `RuntimeException` or `Error`, even if the overridden method doesn't.

To remember this preceding point, let's compare exceptions with monsters. Figure 1.20 shows a fun way to remember the exceptions (monsters) that can be on the list of an

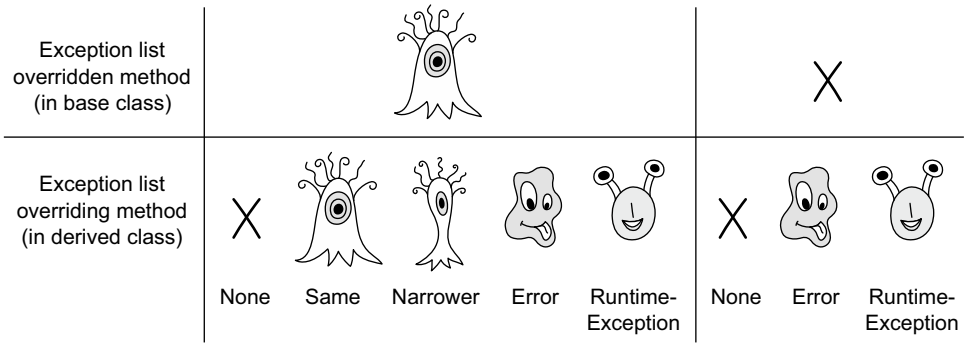

Figure 1.20 Comparing exceptions to monsters. When an overridden method declares to throw a checked exception (monster), the overriding method can declare to throw none, the same, or a narrower checked exception. An overriding method can declare to throw any `Error` or `RuntimeException`.

overriding method, when the overridden method doesn't declare to throw a checked exception and when it declares to throw a checked exception.

1.3.3 Can you override all methods from the base class or invoke them virtually?

The simple answer is no. You can override only the following methods from the base class:

- Methods accessible to a derived class
- Nonstatic base class methods

METHODS ACCESSIBLE TO A BASE CLASS

The accessibility of a method in a derived class depends on its access modifier. For example, a private method defined in a base class isn't available to any of its derived classes. Also, a method with default access in a base class isn't available to a derived class in another package. A class can't override the methods that it can't access.

ONLY NONSTATIC METHODS CAN BE OVERRIDDEN

If a derived class defines a static method with the same name and signature as the one defined in its base class, it *hides* its base class method and *doesn't* override it. You can't override static methods. For example

```
class Book {
    static void printName() {
        System.out.println("Book");
    }
}
class CourseBook extends Book {
    static void printName() {
        System.out.println("CourseBook");
    }
}
```

Static method in base class

Static method in derived class

Method `printName()` in class `CourseBook` hides `printName()` in class `Book`. It doesn't override it. Because the static methods are bound at compile time, the method `printName()` that's called depends on the type of the reference variable:

```
class BookExampleStaticMethod  {
    public static void main(String[] args) {
        Book base = new Book();
        base.printName();

        Book derived = new CourseBook();
        derived.printName();
   }
}
```

Prints "Book"

Prints "Book"

```
class Book{
    public static void printName(){
        ....
    }
    public int issueBook(int days){
        ....
    }
    public int returnBook(int days){
        ....
    }
}
```

```
class CourseBook extends Book{
    public static void printName(){
        ....
    }
    public int issueBook(int days){
        ....
    }
    public int issueBook(){
        ....
    }
    public int returnBook(int a, int b){
        ....
    }
}
```

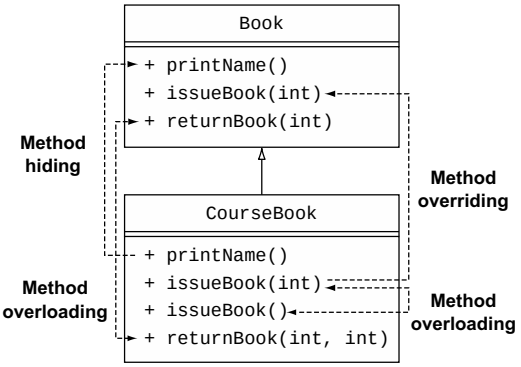

Figure 1.21 Identifying method overriding, method overloading, and method hiding in a base and derived class

1.3.4 Identifying method overriding, overloading, and hiding

It's easy to get confused with method overriding, overloading, and hiding. Figure 1.21 identifies these methods in classes `Book` and `CourseBook`. On the left are the class definitions, and on the right their UML representations.

EXAM TIP When a class extends another class, it can overload, override, or hide its base class methods. A class can't override or hide its own methods—it can only overload its own methods.

Let's check out the correct code for defining a static or nonstatic method in a derived class that overrides or hides a static or nonstatic method in a base class using the next "Twist in the Tale" exercise.

Twist in the Tale 1.3

Let's modify the code of classes `Book` and `CourseBook` and define multiple combinations of static and nonstatic method `print()` in both these classes as follows:

a
```
class Book{
    static void print(){}
}
```

```
    class CourseBook extends Book{
        static void print(){}
    }
b   class Book{
        static void print(){}
    }
    class CourseBook extends Book{
        void print(){}
    }
c   class Book{
        void print(){}
    }
    class CourseBook extends Book{
        static void print(){}
    }
d   class Book{
        void print(){}
    }
    class CourseBook extends Book{
        void print(){}
    }
```

Your task is to first tag them with one of the options and then compile them on your system to see if they're correct. On the actual exam, you'll need to verify (without a compiler) if a code snippet compiles or not:

- Overridden `print()` method
- Hidden `print()` method
- Compilation error

1.3.5 Can you override base class constructors or invoke them virtually?

The simple answer is no. Constructors aren't inherited by a derived class. Because only inherited methods can be overridden, constructors cannot be overridden by a derived class. If you attempt an exam question that queries you on overriding a base class constructor, you know that it's trying to trick you.

EXAM TIP Constructors can't be overridden because a base class constructor isn't inherited by a derived class.

Now that you know why and how to override methods in your own classes, let's see in the next section why it's important to override the methods of class `java.lang.Object`.

1.4 Overriding methods of class Object

[1.6] Override methods from the Object class to improve the functionality of your class

All the classes in java—classes from the Java API, user-defined classes, or classes from any other API—extend class `java.lang.Object`, either implicitly or explicitly. Because this section talks about overriding the methods from class `Object`, let's take a look at its nonfinal and final methods in figure 1.22.

You might write a Java class to be used in your small in-house project or a commercial project, or it could be a part of a library that may be released to be used by other programmers. As you have less control over who uses your class and how it's used, the importance of correctly overriding methods from class `Object` rises. It's important to override the nonfinal `Object` class methods so that these classes can be used efficiently by other users. Apart from being able to be used as desired, incorrect overriding of these methods can also result in increased debug time.

Because the final methods can't be overridden, I'll discuss the nonfinal methods of class `Object` in this section. These methods—`clone()`, `equals()`, `hashCode()`, `toString()`, and `finalize()`—define a contract, a set of rules on how to override these methods, specified by the Java API documentation.

1.4.1 Overriding method toString()

Method `toString()` is called when you try to print out the value of a reference variable or use a reference variable in a concatenation operator. The default implementation of method `toString()` returns the name of the class, followed by `@` and the hash

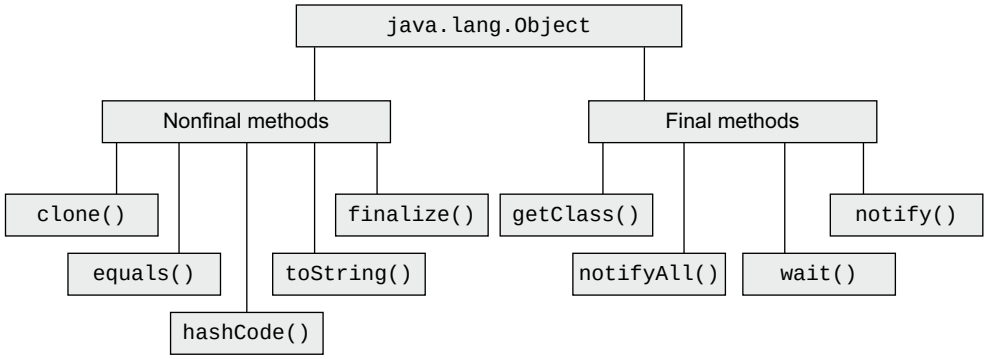

Figure 1.22 Categorization of final and nonfinal methods of class `java.lang.Object`

code of the object it represents. Following is the code of method `toString()`, as defined in class `Object` in the Java API:

toString() as defined in java.lang.Object

```
public String toString() {
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

Following is an example of class `Book`, which doesn't override method `toString()`. In this case, a request to print the reference variable of this class will call method `toString()` defined in class `Object`:

```
class Book {
    String title;
}
class PrintBook {
    public static void main(String[] args) {
        Book b = new Book();
        System.out.println(b);
    }
}
```

Prints a value similar to Book@45a877

Let's override method `toString()` in class `Book`. The contract of method `toString()` specifies that it should return a *concise* but *informative* textual representation of the object that it represents. This is usually accomplished by using the value of the instance variables of an object:

```
class Book {
    String title;
    @Override
    public String toString() {
        return title;
    }
}
class Test {
    public static void main(String[] args) {
        Book b = new Book();
        b.title = "Java Certification";
        System.out.println(b);
    }
}
```

toString() uses title to represent Book

Prints book title, "Java Certification"

If a class defines a lot of instance variables, method `toString()` *might* include only the important ones—that is, the ones that provide its concise description. In the following example, class `Book` defines multiple instance variables and uses a few of them in method `toString()`:

```
class Book {
    String title;
    String isbn;
    String[] author;
    java.util.Date publishDate;
```

Instance variables to store a Book's state

```
    double price;
    int version;
    String publisher;
    boolean eBookReady;
    @Override
    public String toString() {
        return title + ", ISBN:"+isbn + ", Lead Author:"+author[0];
    }
}
class Test {
    public static void main(String[] args) {
        Book b = new Book();
        b.title = "Java Smart Apps";
        b.author = new String[]{"Paul", "Larry"};
        b.isbn = "9810-9643-987";
        System.out.println(b);
    }
}
```

Instance variables to store a Book's state

toString uses title, isbn, and the first element of array author to describe a Book.

Prints "Java Smart Apps, ISBN:9810-9643-987, Lead Author:Paul"

You have overridden method `toString()` inappropriately if it returns any text that's specific to a particular class, for example, the name of a class or a value of a static variable:

```
class Book {
    String title;
    static int bookCopies = 1000;
    @Override
    public String toString() {
        return title + ", Copies:" + bookCopies;
    }
}
class CourseBook extends Book {
    static int bookCopies = 99999;
}
class BookOverrideToString {
    public static void main(String[] args) {
        CourseBook b = new CourseBook();
        b.title = "Java Smart Apps";
        System.out.println(b);
    }
}
```

❶ **Overridden toString() uses static variable of Book.**

❷ **Static variable bookCopies also defined in CourseBook**

❸ **Prints "Java Smart Apps, Copies:1000"**

In this code, ❶ shows inappropriate overriding of method `toString()` because it uses a static variable. The code at ❷ defines a static variable `bookCopies` in class `CourseBook`. Because static members are bound at compile time, method `toString()` will refer to the variable `bookCopies` defined in class `Book`, even if the object it refers to is of the type `CourseBook`. ❸ prints the value of the `static` variable defined in class `Book`.

Overriding methods of class `Object` is an important concept. Let it sink in. The next "Twist in the Tale" exercise will ensure that you get the hang of correct overriding of method `toString()`, before moving on to the next section.

Twist in the Tale 1.4

Which of the following classes—`Book1`, `Book2`, `Book3`, or `Book4`—shows an appropriate overridden method `toString()`?

```
class Book1 {
    String title;
    int copies = 1000;
    public String toString() {
        return "Class Book, Title: " + title;
    }
}
class Book2 {
    String title;
    int copies = 1000;
    public String toString() {
        return ""+copies * 11;
    }
}
class Book3 {
    String title;
    int copies = 1000;
    public String toString() {
        return title;
    }
}
class Book4 {
    String title;
    int copies = 1000;
    public String toString() {
        return getClass().getName() + ":" + title;
    }
}
```

1.4.2 Overriding method equals()

Method `equals()` is used to determine whether two objects of a class should be considered equal or not. Figure 1.23 shows a conversation between two objects, wondering whether they're equal or not.

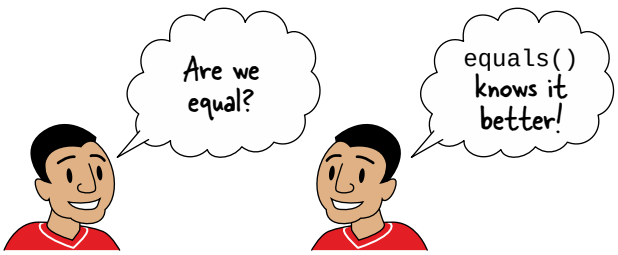

Figure 1.23 Applying a twist on Shakespeare's quote: "Equal or not equal, that is the question." Method `equals()` returns a `boolean` value that determines whether two objects should be considered equal or not.

The default implementation of method `equals()` in class `Object` compares the object references and returns `true` if both reference variables refer to the same object, or `false` otherwise. In essence, it only returns `true` if an object is compared to itself. Following is the default implementation of method `equals()` in class `java.lang.Object`:

```
public boolean equals(Object obj) {
    return (this == obj);
}
```

The exam will question you on the following points:

- The need to override method `equals()`
- Overriding method `equals()` correctly
- Overriding method `equals()` incorrectly

THE NEED TO OVERRIDE METHOD EQUALS()

You need to override method `equals()` for objects that you wish to *equate logically*, which normally depends on the state of an object (that is, the value of its instance variables). The goal of overriding method `equals()` is to check for *equality* of the objects, not to check for the *same* variable references. For two objects of the same class, say, `object1` and `object2`, `equals()` checks whether object1 is *logically* equal to `object2`, but `object1` isn't necessarily pointing to the *exact same object* as `object2`.

For example, class `String` overrides method `equals()` to check whether two `String` objects define the exact same sequence of characters:

```
String name1 = "Harry";
String name2 = new String ("Harry");
System.out.println(name1.equals(name2));
```

Prints "true"

In the preceding code, `name1` and `name2` refer to separate `String` objects but define the exact same sequence of characters—`"Harry"`. So `name1.equals(name2)` returns `true`.

AN EXAMPLE

You might need to find out whether the same undergraduate course is or is not offered by multiple universities. In an application, you can represent a university using a class, say, `University`, and each course being offered using a class, say, `Course`. Assuming that each university offers a list of courses, you can override method `equals()` in class `Course` to determine if two `Course` objects can be considered equal, as follows:

```
class Course {
    String title;
    int duration;
    public boolean equals(Object o) {
        if (o != null && o instanceof Course) {
            Course c = (Course)o;
            return (title.equals(c.title) && duration==c.duration);
        }
        else
            return false;
    }
}
```

RULES FOR OVERRIDING METHOD EQUALS()

Method `equals()` defines an elaborate contract (set of rules), as follows (straight from the Java API documentation):

1. It's *reflexive*—For any non-null reference value `x`, `x.equals(x)` should return `true`. This rule states that an object should be equal to itself, which is reasonable.
2. It's *symmetric*—For any non-null reference values `x` and `y`, `x.equals(y)` should return `true` if and only if `y.equals(x)` returns `true`. This rule states that two objects should be comparable to each other in the same way.
3. It's *transitive*—For any non-null reference values `x`, `y`, and `z`, if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals(z)` should return `true`. This rule states that while comparing objects, you shouldn't selectively compare the values based on the type of an object.
4. It's *consistent*—For any non-null reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return `true` or consistently return `false`, provided no information used in `equals` comparisons on the objects is modified. This rule states that method `equals()` should rely on the value of instance variables that can be accessed from the memory and shouldn't try to rely on values like the IP address of a system, which may be assigned a separate value upon reconnection to a network.
5. For any non-null reference value `x`, `x.equals(null)` should return `false`. This rule states that a non-null object can never be equal to `null`.

Quite a lot of rules to remember! Let's use an interesting way to remember all these rules, by comparing `equals()` to *love*. So when you see "`x.equals(x)`," read it as "`x.loves(x)`." Read "`if x.equals(y)` returns `true`, `y.equals(x)` must return `true`" as "if x loves y, y loves x." All these rules are shown in figure 1.24. They'll make more sense when you cover them using these examples.

CORRECT AND INCORRECT OVERRIDING OF METHOD EQUALS()

To override method `toString()` correctly, follow the method overriding rules defined in section 1.3. Note that the type of parameter passed to `equals()` is `Object`. Watch out for exam questions that *seem to* override `equals()`, passing it to a parameter type of the class in which it's defined. In the following example, class `Course` doesn't *override* method `equals()`, it *overloads* it:

```
class Course {
    String title;
    Course(String title) {
        this.title = title;
    }
    public boolean equals(Course o) {
        return title.equals(o.title);
    }
    public static void main(String args[]) {
        Object c1 = new Course("eJava");
        Object c2 = new Course("eJava");
        System.out.println(c1.equals(c2));
    }
}
```

Course doesn't override toString(), it overloads it.

Prints "false"

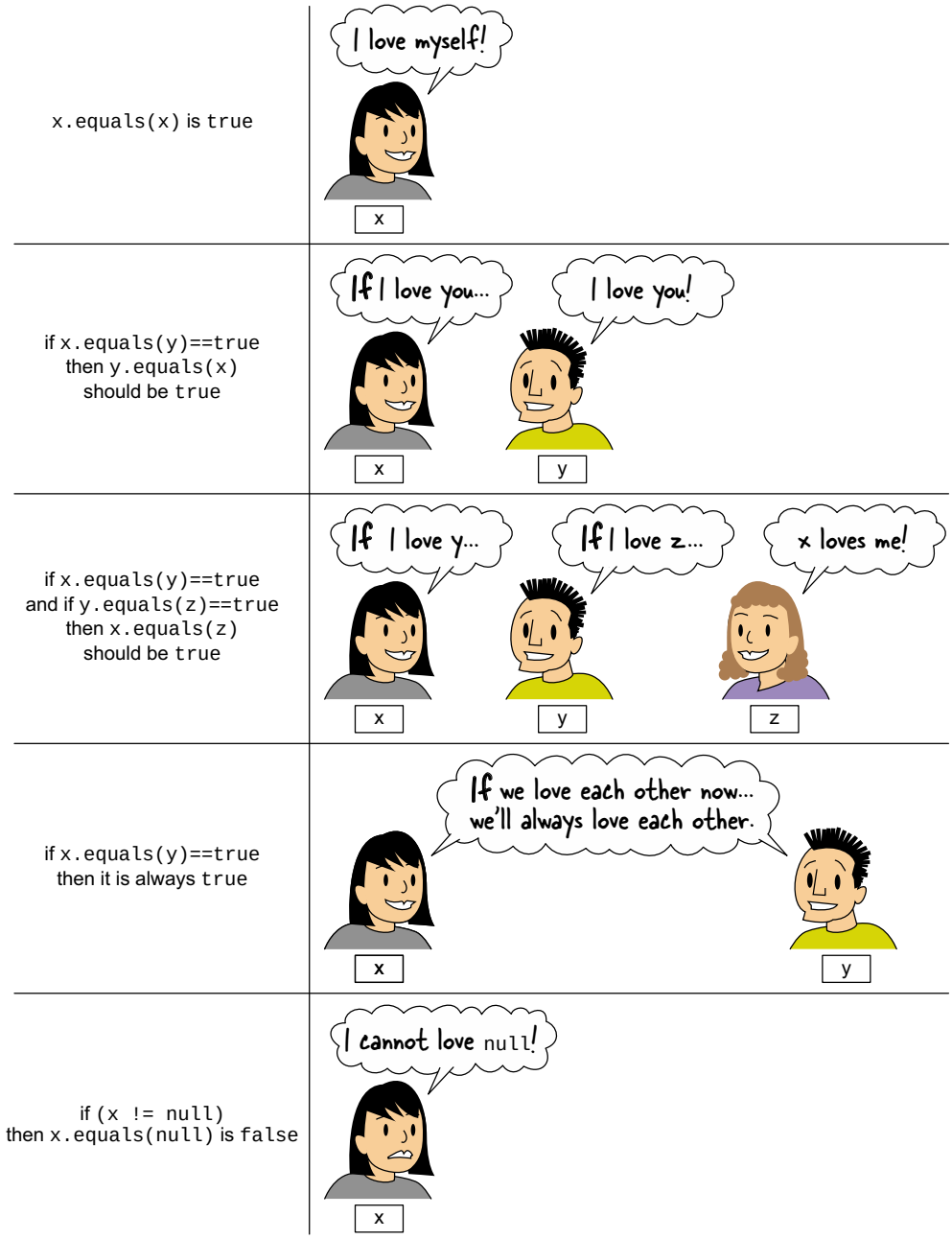

Figure 1.24 A fun way to remember all the rules of the `equals()` contract by comparing `equals()` with *love*.

EXAM TIP Use `Object` as the parameter type to `equals()`. Using any other type will *overload* `equals()`.

APPROPRIATE AND INAPPROPRIATE OVERRIDING OF METHOD EQUALS()

If you don't follow the contract of method `equals()` while overriding it in your classes, you'll be overriding it inappropriately. An inappropriately overridden method `equals()` doesn't mean compilation failure.

EXAM TIP An inappropriately overridden method `equals()` doesn't mean compilation failure.

In the following code, class `Course` doesn't comply with the symmetric and reflexive rules while overriding method `equals()`. Class `University` shows how these rules aren't adhered to:

```
class Course {
    String title;
    Course(String title) {
        this.title = title;
    }
    public boolean equals(Object o) {
        return (title.equals(o));
    }
}
class University {
    public static void main(String[] args) {
        Course c1 = new Course("level1");
        String s1 = "level1";
        System.out.println(c1.equals(s1));
        System.out.println(s1.equals(c1));

        System.out.println(c1.equals(c1));
    }
}
```

Compares title of course with object passed to equals

❶ **Shows violation of symmetric rule—c1.equals(s1) prints "true" but s1.equals(c1) prints "false"**

❷ **Shows violation of reflexive rule—c1.equals(c1) prints "false"**

The code at ❶ prints `true` for `c1.equals(s1)` and `false` for `s1.equals(c1)`, which is a clear violation of `equals()`'s symmetric contract, which states that for any non-null reference values `x` and `y`, `x.equals(y)` should print `true` if and only if `y.equals(x)` returns `true`. The `Course` object will not evaluate to `true` in `String`'s method `equals()` because `equals()` in `String` first verifies if the object being compared to it is a `String` object *before* checking to see if their character sequence is the same. At ❷, `c1.equals(c1)` prints `false`, violating the reflexive rule that states that an object should be equal to itself.

Let's work with another example, where class `JavaCourse` violates the *transitive* rule while overriding method `equals()`. Class `JavaCourse` extends class `Course` and defines method `equals()`, which compares its object to both an object of `Course` and `JavaCourse`. Method `equals()` compares the common attributes, if the object being compared is that of the base class `Course`. It also compares all the attributes, if the object being compared is of class `JavaCourse`:

```
class Course {
    String title;
    Course(String title) {
        this.title = title;
    }
    public boolean equals(Object o) {
        if (o instanceof Course) {
            Course c = (Course)o;
            return (title.equals(c.title));
        }
        else
            return false;
    }
}
class JavaCourse extends Course {
    int duration = 0;
    JavaCourse(String title, int duration) {
        super(title);
        this.duration = duration;
    }
    public boolean equals(Object o) {
        if (o instanceof JavaCourse) {
            return (super.equals(o) &&
                   ((JavaCourse)o).duration == duration);
        }
        else if(o instanceof Course) {
            return (super.equals(o));
        }
        else
            return false;
    }
}
```

In the following code for class `University2`, `c1` is equal to `c2` and `c2` is equal to `c3` because these comparisons only check the course `title`. But `c1` isn't equal to `c3` because the course `duration`s aren't the same. Therefore, the overridden method `equals()` in class `JavaCourse` fails the *transitive* rule:

```
class University2 {
    public static void main(String[] args) {
        Course c1 = new JavaCourse("level1", 2);
        Course c2 = new Course("level1");
        Course c3 = new JavaCourse("level1", 12);

        System.out.println(c1.equals(c2));
        System.out.println(c2.equals(c3));
        System.out.println(c1.equals(c3));
    }
}
```

Inappropriate overriding of method `equals()` can result in bizarre behavior. Use of `equals()` by collection classes is explained in detail in chapter 4.

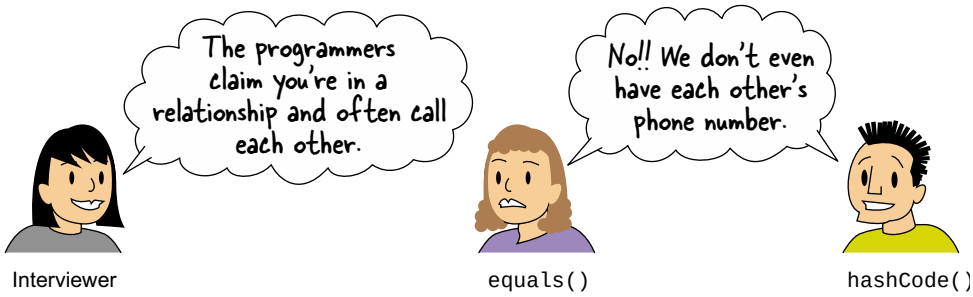

Figure 1.25 Methods `hashCode()` and `equals()` don't call each other.

1.4.3 Overriding method hashCode()

First, method `hashCode()` isn't called by method `equals()` or vice versa. The contract of methods `equals()` and `hashCode()` mentions that both these methods should be overridden if one of them is overridden. This makes a lot of programmers believe that perhaps these methods are called by each other, which isn't the case. Figure 1.25 shows a fun way to remember that methods `equals()` and `hashCode()` deny being in a relationship and calling each other.

THE NEED TO OVERRIDE METHOD HASHCODE()

Method `hashCode()` returns a hash-code value for an object, which is used to efficiently store and retrieve values in collection classes that use hashing algorithms, such as `HashMap`. Hashing algorithms identify the *buckets* in which they would store the objects and from which they would retrieve them. A well-written method `hashCode()` ensures that objects are evenly distributed in these buckets. Objects with the same hash-code values are stored in the same bucket. To retrieve an object, its bucket is identified using its hash-code value. If the bucket contains multiple objects, method `equals()` is used to find the target object.

To understand how this works, let's create a class called `MyNumber`, which contains a primitive `long` as its field. It returns a sum of all the individual digits of its field as its method `hashCode()`, as follows:

```
class MyNumber {
    long number;
    MyNumber(long number) {this.number = number;}
    public int hashCode() {
        int sum = 0;
        long num = number;
        do {
            sum += num % 10; num /= 10;
        }
        while( num != 0 );
        return sum;
    }
}
```

Let's assume you add the following keys and values in a `HashMap`:

```
Map<MyNumber, String> map = new HashMap<>();
MyNumber num1 = new MyNumber(1200);
MyNumber num2 = new MyNumber(2500);
MyNumber num3 = new MyNumber(57123);
map.put(num1, "John");
map.put(num2, "Mary");
map.put(num3, "Sam");
```

Hash-code value 3

Hash-code value 7

Hash-code value 18

With the preceding keys, each bucket contains only one entry. When you request the `HashMap` to retrieve a value, it would find the corresponding bucket using the key's hash-code value and then it retrieves the value. Now let's add another key-value pair:

```
MyNumber num4 = new MyNumber(57123);
map.put(num4, "Kim");
```

Hash-code value 18

Now the bucket with the hash-code value `18` has two `String` values. In this case, `HashMap` would use the `hashCode()` value to identify the bucket and then call method `equals()` to find the correct object. This explains why distinct hash-code values for distinct values are preferred.

NOTE Chapter 4 explains in detail how the hashing algorithms in collection classes use methods `hashCode()` and `equals()`.

OVERRIDING METHOD HASHCODE() CORRECTLY

Here's the signature of method `hashCode()` as defined in class `Object`:

```
public native int hashCode();
```

To correctly override method `hashCode()`, you must follow the rules already discussed in section 1.3. Watch out for exam questions that use the incorrect case for `hashCode()`—the correct name uses uppercase C. Figure 1.26 shows a fun way to remember this simple, but important, exam point.

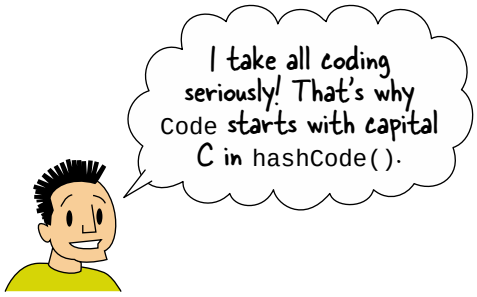

Figure 1.26 The correct case of `hashCode()` includes a capital C.

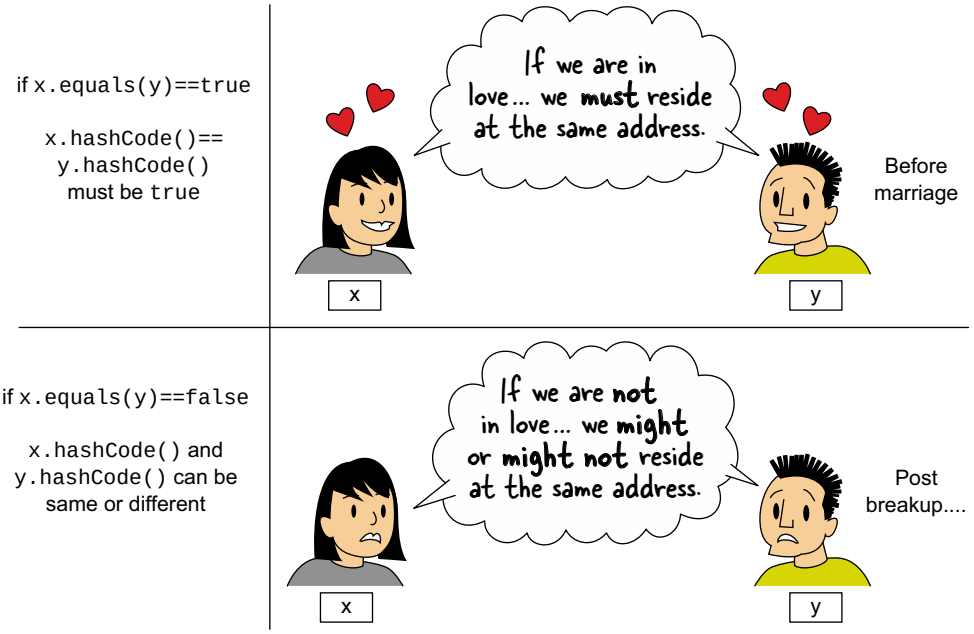

Figure 1.27 Comparing `equals()` with being in love and `hashCode()` with an address. If two objects are equal, they *must* return the same `hashCode()`. But if two objects return the same `hashCode()`, they *might not* be equal.

To override method `hashCode()` correctly, you must also abide by its contract, as mentioned in the Java documentation. For the exam, the following rules are important:

1. If two objects are equal according to method `equals(Object)`, then calling method `hashCode()` on each of the two objects must produce the same integer result.
2. It's not required that if two objects are unequal according to method `equals(java.lang.Object)`, that calling method `hashCode()` on each of the two objects must produce distinct integer results.

Let's use a fun analogy to remember these rules, as shown in figure 1.27. Let's compare method `equals()` to being in love and method `hashCode()` to a physical address. When two objects are in love with each other, they *must* reside at the same address (this is what they think *before* they marry). Later, if they fall out of love, they might or might not continue to reside at the same address.

Figure 1.27 will make more sense as you work with the code examples in this section. Let's revisit the previous example, including `equals()` and verifying the rules:

```
class MyNumber {
    long number;
    MyNumber(long number) {this.number = number;}
```

```
    public int hashCode() {
        int sum = 0;
        long num = number;
        do {
            sum += num % 10; num /= 10;
        }
        while( num != 0 );
        return sum;
    }
    public boolean equals(Object o) {
        if (o != null && o instanceof MyNumber)
            return (number == ((MyNumber)o).number);
        else
            return false;
    }
    public static void main(String args[]) {
        MyNumber n1 = new MyNumber(9);
        MyNumber n2 = new MyNumber(18);
        MyNumber n3 = new MyNumber(18);
        System.out.println
             (n1.equals(n2)+":"+n1.hashCode()+":"+n2.hashCode());
        System.out.println
             (n2.equals(n3)+":"+n2.hashCode()+":"+n3.hashCode());
    }
}
```

Prints "false:9:9"

Prints "true:9:9"

The preceding code abides by both the rules of the `hashCode()` contract, when `n2.equals(n3)` returns `true`, `n2.hashCode()` and `n3.hashCode()` return the same value. But when `n1.equals(n2)` returns `false`, `n1.hashCode()` and `n2.hashCode()` might not return distinct values.

Let's modify the preceding code, so that `hashCode()` returns distinct values when `equals()` returns `false`:

```
class MyNumber {
    long number;
    MyNumber(long number) {this.number = number;}
    public int hashCode() {
        return (int)number;
    }
    public boolean equals(Object o) {
        if (o != null && o instanceof MyNumber)
            return (number == ((MyNumber)o).number);
        else
            return false;
    }
    public static void main(String args[]) {
        MyNumber n1 = new MyNumber(9);
        MyNumber n2 = new MyNumber(18);
        MyNumber n3 = new MyNumber(18);
        System.out.println
             (n1.equals(n2)+":"+n1.hashCode()+":"+n2.hashCode());
        System.out.println
             (n2.equals(n3)+":"+n2.hashCode()+":"+n3.hashCode());
    }
}
```

Prints "false:9:18"

Prints "true:18:18"

NOTE Using a system-dependent value (like a memory address) is *allowed* in `hashCode()`. But objects of such classes can't be used as keys in distributed systems because *equal* objects (across systems) will return different hash-code values.

OVERRIDING METHOD HASHCODE() INAPPROPRIATELY

Inappropriate overriding isn't the same as incorrect overriding—the former won't fail compilation but can have issues with object retrieval. On the exam, watch out for questions that will show code for `hashCode()`, `equals()`, or both, and query what happens when the class instances are used as keys in collection classes, like `HashMap`. In this section, you'll work with examples that override `hashCode()` correctly—syntactically, but not appropriately.

In the previous section you learned why it's important for method `hashCode()` that two objects return the same value, if they're equal as per method `equals()`. Failing this condition, an object value will never be able to be retrieved from a `HashMap`. Let's see what happens when class `MyNumber` doesn't return the same `hashCode()` values for its *equal* objects:

```
class MyNumber {
    int primary, secondary;
    MyNumber(int primary, int secondary) {
        this.primary = primary;
        this.secondary = secondary;
    }
    public int hashCode() {
        return secondary;
    }
    public boolean equals(Object o) {
        if (o != null && o instanceof MyNumber)
            return (primary == ((MyNumber)o).primary);
        else
            return false;
    }
    public static void main(String args[]) {
        Map<MyNumber, String> map = new HashMap<>();
        MyNumber num1 = new MyNumber(2500, 100);
        MyNumber num2 = new MyNumber(2500, 200);
        System.out.println(num1.equals(num2));
        map.put(num1, "Shreya");
        System.out.println(map.get(num2));
    }
}
```

❶ **Doesn't print same hashCode() value for equal objects**

❷ **Prints "true"—objects num1 and num2 are considered equal.**

❸ **Prints "null"**

In the preceding code, even though the code at ❷ prints `true`, confirming that objects `num1` and `num2` are considered equal by `equals()`, the code at ❸ prints `null`. The reason for this? The `hashCode()` in `MyNumber` doesn't return the same values for its equal objects. In method `hashCode()`, the code at ❶ uses `secondary` to calculate its value, which isn't used by `equals()`.

Another rule of method `hashCode()` is that when it's invoked on the same object more than once during the execution of a Java application, `hashCode()` *must* consistently return the same integer, provided no information used in the `equals()` comparisons on the object is modified. This integer doesn't need to remain consistent from one execution of an application to another execution of the same application.

Let's see what happens when `hashCode()` doesn't return the same integer value when it's invoked on the same instance during the execution of a Java application:

```
class MyNumber {
    int number;
    MyNumber(int number) {this.number = number;}
    public int hashCode() {
        return ((int)(Math.random() * 100));
    }
    public boolean equals(Object o) {
        if (o != null && o instanceof MyNumber)
            return (number == ((MyNumber)o).number);
        else
            return false;
    }
    public static void main(String args[]) {
        Map<MyNumber, String> map = new HashMap<>();
        MyNumber num1 = new MyNumber(2500);
        map.put(num1, "Shreya");
        System.out.println(map.get(num1));
    }
}
```

Prints random hash-code values on each invocation

Prints "null" (most probably)

In the preceding code, when you add key-value `num1`, `"Shreya"` to `HashMap`, you most likely won't be able to retrieve `Shreya` using the same key, `num1`. This is because each call to `num1.hashCode()` might return a different value (the chances of returning the same `hashCode()` values aren't ruled out, but are very low).

INEFFICIENT OVERRIDING OF HASHCODE()

In real projects, always strive for generating distinct values in `hashCode()`. Distinct `hashCode()` values and faster object access are directly related in collection objects that use hashing functions to retrieve and store values. Here's an example of inefficient overriding of method `hashCode()`:

```
class MyNumber {
    long number;
    MyNumber(long number) {this.number = number;}
    public int hashCode() {
        return 1654;
    }
}
```

In the preceding code, method `hashCode()` returns the same hash-code value for all the objects of `MyNumber`. This essentially stores all the values in the same bucket, if

objects of the above class are used as keys in class `HashMap` (or in similar classes that use hashing), and reduces it to a linked list, drastically reducing its efficiency.

EXAM TIP Read the questions on method `hashCode()` carefully. You might be questioned on incorrect, inappropriate, or inefficient overriding of `hashCode()`.

EFFECTS OF USING MUTABLE OBJECTS AS KEYS

Java recommends using immutable objects as keys for collection classes that use the hashing algorithm. What if you don't? The exam might query you on this important question.

Revisiting the example used in the previous section, what happens if the value of the field `number` is changed during the course of the application? In this case, you'll never be able to retrieve the corresponding value in the `HashMap`, because the `HashMap` will not be able to look for the right bucket:

```
class MyNumber {
    int number;
    MyNumber(int number) {this.number = number;}
    public int hashCode() {
        return number;
    }
    public boolean equals(Object o) {
        if (o != null && o instanceof MyNumber)
            return (number == ((MyNumber)o).number);
        else
            return false;
    }
    public static void main(String args[]) {
        Map<MyNumber, String> map = new HashMap<>();
        MyNumber num1 = new MyNumber(2500);
        map.put(num1, "Shreya");
        num1.number = 100;
        System.out.println(map.get(num1));
    }
}
```

Add value Shreya to HashMap using key num1.

Modify field number of key num1, which is used by equals() and hashCode().

Prints "null"—can't locate object with modified key.

In the preceding code, the field used to determine the hash code of an object is modified in `main()`. With the modified key, `HashMap` won't be able to retrieve its corresponding object.

In the next section, you'll cover when, why, and how you can cast an instance to another type and use the `instanceof` operator.

1.5 Casting and the instanceof operator

[1.4] Use the instanceof operator and casting

Imagine that you enroll yourself for flying classes, where you expect to be trained by an experienced pilot. Even though your trainer might also be a swimming champion,

you need not know about it. You need not care about the characteristics and behavior that's not related to flying. Now think of a situation when you do care about the swimming skills of your instructor. Imagine that when you're attending the flying classes, your friend enquires whether your flying instructor also conducts swimming classes and, if yes, whether she would be willing to assist your friend. In this case, a *need* arises to enquire about the swimming skills (additional *existing* skills) of your flying instructor.

Similarly, in Java, you can refer to an object of a derived class using a reference variable of its base class or implemented interface. But you might need to access the members of the derived class, which aren't defined in its base class or the implemented interface. Here's when *casting* can help. Casting shows how an object of a type can be used as an object of another type, either implicitly or explicitly. The `instanceof` operator is used to logically test whether an object is a valid type of a class or an interface.

1.5.1 Implicit and explicit casting

Let's start with the definitions of the interface `Printable` and classes `ShoppingItem` and `Book` to show implicit and explicit casting. Class `Book` extends class `ShoppingItem` and implements the interface `Printable` as follows:

```
public interface Printable {
    void print();
}
public class ShoppingItem {
    public void description() {
        System.out.println("Shopping Item");
    }
}
public class Book extends ShoppingItem implements Printable {
    public void description() {
        System.out.println("Book");
    }
    public void print() {
        System.out.println("Printing book");
    }
}
```

Figure 1.28 shows the inheritance relationship between these classes.

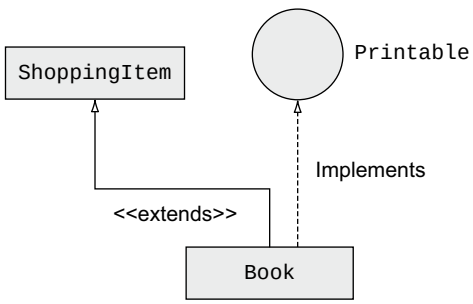

Figure 1.28 Relationship between classes `ShoppingItem` and `Book` and the interface `Printable`

Now let's create variables of type `Printable` and `ShoppingItem` and assign to them objects of the type `Book`:

```
class Shopping {
    public static void main(String args[]) {
        Book book = new Book();
        Printable printable = book;
        printable.print();

        ShoppingItem shoppingItem = book;
        shoppingItem.description();
    }
}
```

❶ **Implicit casting**

❷ **Acceptable**

The code at ❶ shows how an object of type `book` is *implicitly* referred to, or casted to, type `Printable`. The code at ❷ shows how an object of type `book` is *implicitly* referred to, or casted to, type `ShoppingItem`. Objects of subclasses can be implicitly casted to their base classes or the interfaces that they implement.

As shown in the preceding code block for the class `Book`, you can see that `Book` defines a method `description()`. Let's try to access it using the `printable` variable:

```
class Shopping {
    public static void main(String args[]) {
        Printable printable = new Book();
        printable.description();
    }
}
```

❶ **Won't compile—can't access method description() in Printable.**

The code at ❶ fails to compile with the following message:

```
Shopping.java:4: error: cannot find symbol
        printable.description();
                  ^
  symbol:   method description()
  location: variable printable of type Printable
1 error
```

Because the type of the reference variable `printable` is `Printable`, the compiler refers to the definition of the interface `Printable` when you call method `description()` on `printable`. Figure 1.29 shows what happens behind the scenes.

But you know that the actual object is of type `Book`. Is there a way to treat the reference variable `printable` as a `Book`? Yes, there is! You need to inform the compiler you know what you're doing by using an explicit cast, as follows (see also figure 1.30):

```
class Shopping {
    public static void main(String args[]) {
        Printable printable = new Book();
        ((Book)printable).description();
    }
}
```

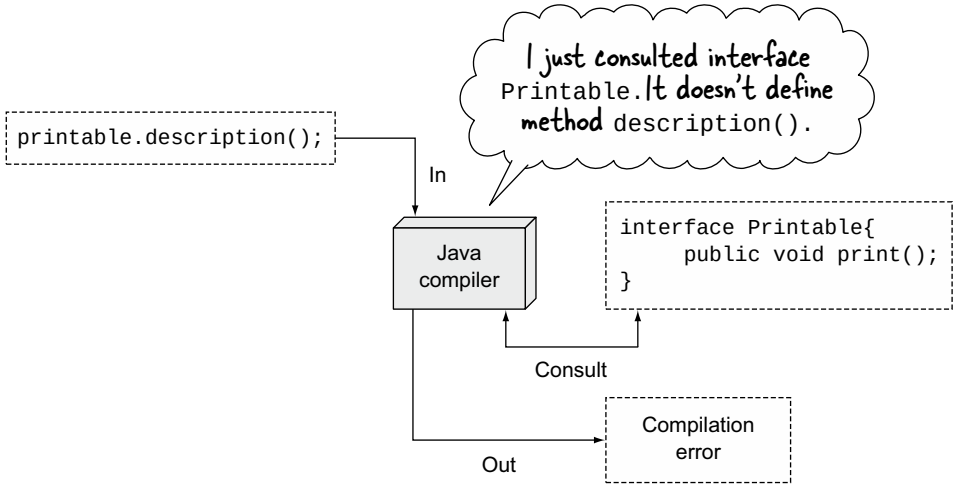

Figure 1.29 The Java compiler doesn't compile code if you try to access `description()`, defined in class `Book`, by using a variable of the interface `Printable`.

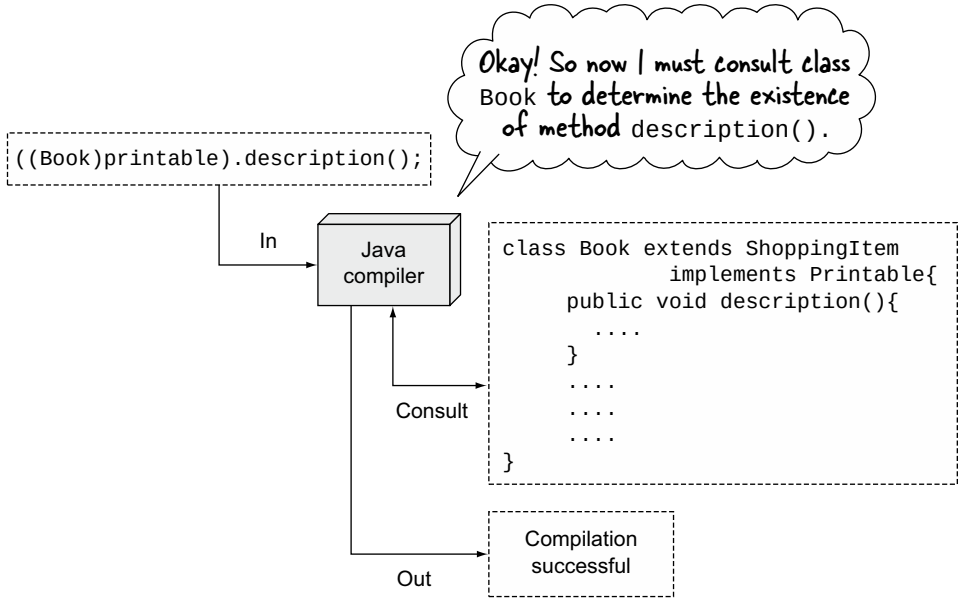

Figure 1.30 Explicit casting can be used to access `description()` defined in class `Book` by using a variable of the interface `Printable`.

```
interface Printable {
    void print();
}
class ShoppingItem {
    public void description() {
        System.out.println("Shopping Item");
    }
}
class Chair extends ShoppingItem {
    public void description() {
        System.out.println("Chair");
    }
}
class Book extends ShoppingItem
                implements Printable {
    public void description() {
        System.out.println("Book");
    }
    public void print() {
        System.out.println("Printing book");
    }
}
```

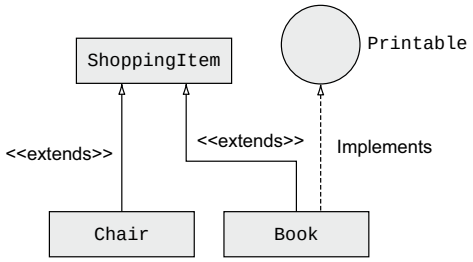

Figure 1.31 Set of classes and interfaces, with their UML representation

In the preceding code, `(Book)` is placed just before the name of the variable, `printable`, to cast it to `Book`. Note how a pair of parentheses surrounds `(Book)printable`. Casting in this line of code is another method of telling the compiler that you know that the actual object being referred to is `Book`, even though you're using a reference variable of type `Printable`.

1.5.2 Combinations of casting

To work with a combination of casting, let's work with a set of classes and interfaces, as shown in figure 1.31.

ASSIGNMENTS WITH IMPLICIT CASTING

Implicit upcasting is allowed. You can assign a reference variable of a derived class to a reference variable of its own type, its base classes, and the interfaces that it implements as follows:

```
public class UpcastWithImplicitCasting {
    public static void main(String[] arguments) {
        Book book = new Book();

        Chair chair = book;
        ShoppingItem shoppingItem = book;
        Printable printable = book;
        Object object = book;

        Chair chair2 = new Chair();
        Printable printable2 = chair;
    }
}
```

Won't compile—both Book and Chair extend ShoppingItem, but don't belong to a single line of inheritance.

Okay—a book is a ShoppingItem.

Okay—a book is Printable.

Okay—a book is an Object.

Won't compile—Chair doesn't implement Printable.

Implicit downcasting isn't allowed. You can't assign reference variables of a base class to reference variables of its derived classes or to the interfaces that it doesn't implement. For example

```
public class DowncastWithImplicitCasting {
    public static void main(String[] arguments) {
        ShoppingItem shoppingItem3 = new ShoppingItem();

        Book book3 = shoppingItem3;
        Chair chair3 = shoppingItem3;
        Printable printable3 = shoppingItem3;

        Object object3 = shoppingItem3;
    }
}
```

Won't compile—a ShoppingItem isn't necessarily a book.

Won't compile—a ShoppingItem isn't necessarily a chair.

Won't compile—a ShoppingItem isn't Printable.

Okay—a chair is an Object.

EXAM TIP In the absence of explicit casting, you'll never get `ClassCastException`—a `RuntimeException`.

Assignment with explicit casting

Both implicit and explicit upcasting are allowed. So, for the exam, let's focus on explicit downcasting.

Java recommends programming to an interface, which implies using reference variables of a base class or implementing interfaces to refer to the actual objects. But you might need to cast an object referred by a base class to its specific type. You can downcast an object to a type that falls in its inheritance tree using explicit casting. For a nonfinal class, you can explicitly cast its object to any interface type, even if the class doesn't implement the interface. Let's see what happens when you accept a method parameter of type `ShoppingItem` and try to cast it explicitly to other types:

```
public class DowncastWithExplicitCasting {
    static void downCast(ShoppingItem item) {
        Book book = (Book)item;
        Chair chair = (Chair)item;
        Printable printable = (Printable)item;
    }
    public static void main(String args[]) {
        ShoppingItem item = new ShoppingItem();
        downCast(item);
    }
}
```

❶ **Compiles with casting—will throw ClassCastException; can't downcast instance of parent object to subclass type.**

❷ **Compiles with casting—will throw ClassCastException; ShoppingItem doesn't implement Printable.**

The code at ❶ and ❷ compiles with an explicit cast. But its individual lines will fail at runtime. At runtime, Java can determine the exact type of the object being casted. It throws a `ClassCastException` if you're trying to cast types that aren't allowed.

NOTE For the exam, you need to be very clear whether an explicit cast will result in a compilation error or a runtime exception (`ClassCastException`).

Does the preceding code make you wonder why an explicit cast from a `ShoppingItem` instance to `Printable` is permitted, even though `ShoppingItem` doesn't implement `Printable`? It's to allow subclasses of `ShoppingItem` to implement `Printable` and use the reference variable of type `Printable` to refer to its instances. So what happens if you try to cast a final class's instance to an interface it doesn't implement? The code won't compile:

```
interface Printable {}
final class Engineer {}
class Factory {
    public static void main(String[] args) {
        Engineer engineer = new Engineer();
        Printable printable = (Printable)engineer;
    }
}
```

Won't compile—can't cast final class Engineer's instance to Printable.

EXAM TIP Class `String` is defined as a final class. Watch out for questions that explicitly cast `String` objects to interfaces they don't implement. They won't compile.

What about casting `null` to a type? You can explicitly cast `null` to any type without a compilation error or runtime exception (`ClassCastException`):

```
static void castNull() {
    Book book = (Book)null;
    Chair chair = (Chair)null;
    Printable printable = (Printable)null;
}
```

EXAM TIP You can explicitly cast `null` to any type. It won't generate a compilation error or throw a `ClassCastException`.

ACCESS OF MEMBERS WITH EXPLICIT CASTING

You can access methods and variables of explicitly casted variables in single or multiple lines of code:

```
public class AccesMembersWithExplicitCasting {
    static void accessMember(ShoppingItem item) {
        Book book = (Book)item;
        book.description();

        ((Book)item).description();
    }
}
```

❶ **Cast a reference variable and access its method in multiple steps.**

❷ **Cast objects and call their members in a single step.**

Here the code at ❶ casts a reference `item` to `Book` in one line and then accesses its method `description()`. At ❷, note how the object referred by `item` is casted-enclosed within `()` to call its member method `description()`. The inclusion in `()` is due to the fact that the dot operator has precedence over the casting parentheses.

EXAM TIP If you cast an instance to a class outside its inheritance tree, you'll get a compiler error. If you cast an instance to a class within its inheritance tree, but the types don't match at runtime, the code will throw a `ClassCastException`.

Points to remember for casting

- An instance can be implicitly casted to its superclasses or interfaces that it implements.
- An instance of a nonfinal class can be explicitly casted to any interface at compile time.
- Classes in the same inheritance tree can be casted to each other using explicit casting at compile time.
- Objects of classes that don't form part of the same inheritance tree cannot be casted.
- Casting to an interface is successful at runtime if the class implements the interface.
- Casting to a derived class type is successful at runtime if the casted object is actually a type of the derived class to which it's casted.

In the previous examples, you learned how mismatching of objects and explicit casting can throw a `ClassCastException`. In the next section, you'll see how you can prevent this by using the `instanceof` operator to safely cast objects to a type.

1.5.3 Using the instanceof operator

The `instanceof` operator is used to logically test whether an object is a valid type of a class or an interface. You should proceed with explicit casting only if this operator returns `true`, or you risk running into a `ClassCastException` at runtime. For example, consider `equals()`, which defines a method parameter of type `Object`. When you override `equals()` to determine the equality of objects of your class, you might need to query the state of the accepted argument before you move forward with an explicit cast:

```
class Course {
    String title;
    Course(String t) {title = t;}
    public boolean equals(Object obj) {
        if (obj instanceof Course) {
            Course c = (Course)obj;
            return (title.equals(c.title));
        }
        else
            return false;
    }
}
```

❶ Use instanceof to verify if obj is an instance of Course.

❷ Explicitly cast obj to Course.

The code at ❶ ensures that the type of the accepted method parameter—that is, `obj`—is `Course`, before it moves forward with the explicit casting of `obj` to `Course` ❷.

In the absence of this check, the code at ❷ would execute for all non-null method parameters, which can result in a `ClassCastException` if the object passed to `equals()` isn't of type `Course`.

EXAM TIP The operator `instanceof` returns `false` if the reference variable being compared to is `null`.

In the previous example, the type of method parameter to `equals()` is `Object`, which is the parent class of all classes. But if the `instanceof` operator uses inconvertible types, the code won't compile. In the following example, the `instanceof` operator uses a reference variable of type `Course` to test whether the object that it refers to can be an *instance of* class `Student`. Because `Course` and `Student` are unrelated, class `Test` won't compile:

```
class Course {}
class Student {}
public class TestInstanceof {
    public static void main(String[] args) {
        Course c = new Course();
        Student s = new Student();
        System.out.println(c instanceof Student);
    }
}
```

Won't compile—can't use instanceof to compare inconvertible types.

EXAM TIP The `instanceof` operator *never* throws a runtime exception; it returns either `true` or `false`. If the `instanceof` operator uses inconvertible types, the code won't compile.

The `instanceof` operator is preceded by a value (literal value or a variable name) and is followed by a class, interface, or enum name. It's acceptable to use the literal value `null` with the `instanceof` operator:

```
class Course {
  public static void main(String[] args) {
        System.out.println(null instanceof Course);
    }
}
```

Prints "false"—null can't be an instance of any class.

EXAM TIP The literal value `null` isn't an instance of any class. So `<referenceVariable> instanceof <ClassName>` will return `false` whenever the `<referenceVariable>` is `null`.

Using instanceof versus getClass in method equals()

Using `instanceof` versus `getClass` is a common subject of debate about proper use and object orientation in general (including performance aspects, design patterns, and so on). Though important, this discussion is beyond the scope of this book. If you're interested in further details, refer to Josh Bloch's book *Effective Java*.

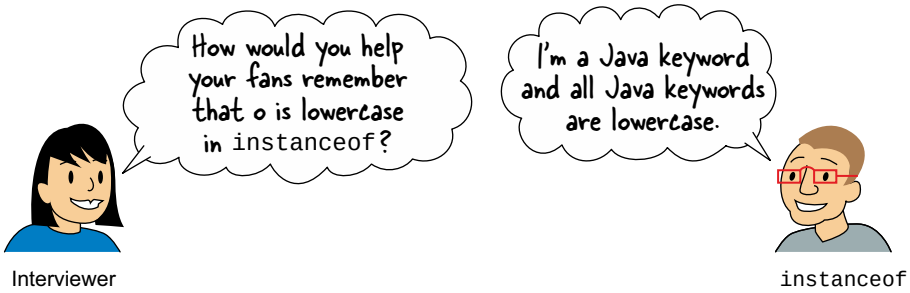

Figure 1.32 Remember that o in `instanceof` is lowercase.

Note that o in `instanceof` is lowercase; take a look at figure 1.32 for a fun way of remembering this.

Next we'll move forward with defining the Java classes and interfaces in named packages. This is a common requirement in real Java applications. So let's get started.

1.6 Packages

[1.7] Use package and import statements

In this section, you'll learn what Java packages are and how to create them. You'll use the `import` statement, which enables you to use simple names for classes and interfaces defined in separate packages.

1.6.1 The need for packages

You can use packages to group together a related set of enums, classes, and interfaces. Packages also provide namespace management. You can create separate packages to define classes for separate projects, such as Android games and online healthcare systems. Further, you can create subpackages within these packages, such as separate subpackages for GUIs, database access, networking, and so on.

NOTE In real-life projects, you'll never work with a package-less class or interface. Almost all organizations that develop software have strict package-naming rules, which are often documented.

If you don't include an explicit `package` statement in a class or an interface, it's part of a *default* package.

1.6.2 Defining classes in a package using the package statement

You can define classes and interfaces in a package by using the `package` statement as the first statement in your class or interface (only comments can precede the `package` statement). Here's an example:

```
package certification;
class ExamQuestion {
    //..code
}
```

The class in the previous code defines a class `ExamQuestion` in the `certification` package. You can define an interface, `MultipleChoice`, in a similar manner:

```
package certification;
interface MultipleChoice {
    //..code
}
```

Figure 1.33 shows the UML representation of the `certification` package, with class `ExamQuestion` and interface `MultipleChoice`.

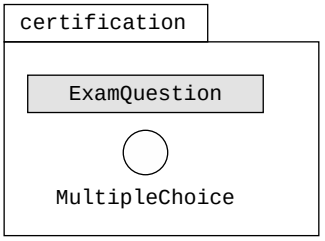

Figure 1.33 A UML representation of the `certification` package, class `ExamQuestion`, and interface `MultipleChoice`

The name of the package in the previous examples is `certification`. You may use such names for small projects that contain only a few classes and interfaces, but it's common for organizations to use subpackages to define *all* their classes. For example, if folks at Oracle define a class to store exam questions for a Java Associate exam, they might use the package name `com.oracle.javacert.associate`. For subpackages, the `package` statement includes the complete package name. Figure 1.34 shows its UML representation along with the corresponding class definition.

NOTE A fully qualified name for a class or interface is formed by prefixing its name with its package name (separated by a period). The fully qualified name of the `ExamQuestion` class is `certification.ExamQuestion` in figure 1.33 and `com.oracle.javacert.associate.ExamQuestion` in figure 1.34.

```
package com.oracle.javacert.associate;
class ExamQuestion {
    // variables and methods
}
```

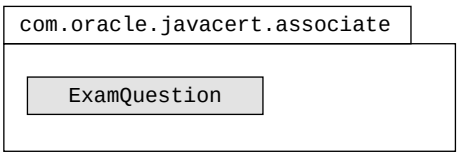

Figure 1.34 A subpackage and its corresponding class definition

Rules to remember about packages

Here are a few important rules about packages:

- Per Java naming conventions, package names should all be in lowercase.
- The package and subpackage names are separated using a dot (`.`).
- Package names follow the rules defined for valid identifiers in Java.
- For packaged classes and interfaces, the `package` statement is the first statement in a Java source file (a .java file). The exception is that comments can appear before or after a `package` statement.
- There can be a maximum of one `package` statement per Java source code file (.java file).
- All the classes and interfaces defined in a Java source code file will be defined in the same package. There's no way to define them in different packages.

DIRECTORY STRUCTURE AND PACKAGE HIERARCHY

The hierarchy of the classes defined in packages should match the hierarchy of the directories in which these classes and interfaces are defined in the code. For example, class `ExamQuestion` in the `certification` package should be defined in a directory with the name certification. The name of the certification directory and its location are governed by the rules shown in figure 1.35.

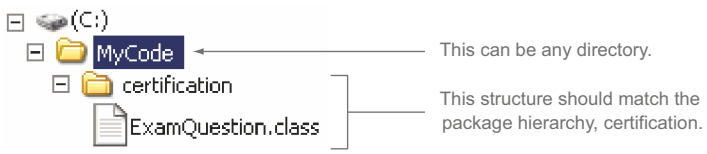

Figure 1.35 Matching directory structure and package hierarchy

For the package example shown in figure 1.35, note that there isn't any constraint on the location of the base directory in which the directory structure is defined. Examine figure 1.36.

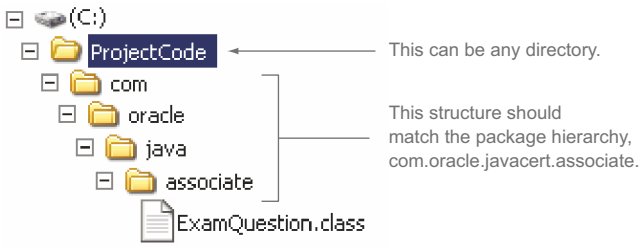

Figure 1.36 There's no constraint on the location of the base directory to define directories corresponding to package hierarchy.

SETTING THE CLASS PATH FOR CLASSES IN PACKAGES

To enable the JRE to find your classes, interfaces, and enums defined in packages, add the base directory that contains your Java code to the class path.

For example, to enable the JRE to locate the `certification.ExamQuestion` class from the previous examples, add the directory C:\MyCode to the class path. To enable the JRE to locate class `com.oracle.javacert.associate.ExamQuestion`, add the directory C:\ProjectCode to the class path.

You don't need to bother setting the class path if you're working with an integrated development environment (IDE). But I strongly encourage you to learn how to work with a simple text editor and how to set a class path. This can be particularly helpful with your projects at work. I've also witnessed many interviewers querying candidates on the need for class paths.

1.6.3 Using simple names with import statements

The `import` statement enables you to use *simple names* instead of using *fully qualified names* for classes and interfaces defined in separate packages. Let's work with an example, in which classes `LivingRoom` and `Kitchen` are defined in the package `home` and classes `Cubicle` and `ConferenceHall` are defined in the package `office`. Class `Cubicle` uses (is associated to) class `LivingRoom` in the package `home`, as shown in figure 1.37.

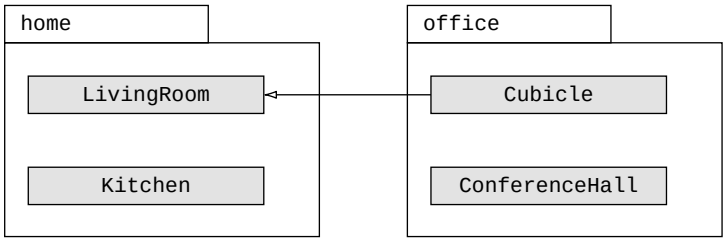

Figure 1.37 A UML representation of classes `LivingRoom` and `Cubicle`, defined in separate packages, with their associations

Class `Cubicle` can refer to class `LivingRoom` without using an `import` statement:

```
package office;
class Cubicle {
    home.LivingRoom livingRoom;
}
```

For no import statement, use fully qualified name to refer to LivingRoom from package home

Class `Cubicle` can use the simple name for class `LivingRoom` by using the `import` statement:

```
package office;
import home.LivingRoom;

class Cubicle {
    LivingRoom livingRoom;
}
```

Import statement

No need to use fully qualified name of LivingRoom

NOTE The `import` statement doesn't embed the contents of the imported class in your class, which means that importing more classes doesn't increase the size of your own class. It lets you use the simple name for a class or interface defined in a separate package.

1.6.4 Using packages without using the import statement

Classes in the `java.lang` package are automatically imported in all the Java classes, interfaces, and enums. To use simple names for classes, interfaces, and enums from other packages, you should use the `import` statement. It's possible to use a class or interface from a package without using the `import` statement by using its fully qualified name:

```
class AnnualExam {
    certification.ExamQuestion eq;
}
```

Missing import statement

Define a variable of ExamQuestion by using its fully qualified name.

But using a fully qualified class name can clutter your code if you use multiple variables of interfaces and classes defined in other packages. *Don't* use this approach in real projects.

For the exam, it's important to note that you can't use the `import` statement to use multiple classes or interfaces with the same names from different packages. For example, the Java API defines class `Date` in two commonly used packages: `java.util` and `java.sql`. To define variables of these classes in a class, use their fully qualified names with the variable declaration:

```
class AnnualExam {
    java.util.Date date1;
    java.sql.Date date2;
}
```

Missing import statement

Variable of type java.util.Date

Variable of type java.sql.Date

An attempt to use an `import` statement to import both these classes in the same class will not compile:

```
import java.util.Date;
import java.sql.Date;
class AnnualExam { }
```

Code to import classes with same name from different packages won't compile

In the preceding code, you want to use a shortcut (`Date`) but your shortcut refers to either `java.util.Date` or `java.sql.Date`. So the Java compiler has no way of knowing which is which (both have `Date` as their simple name), therefore the compiler error.

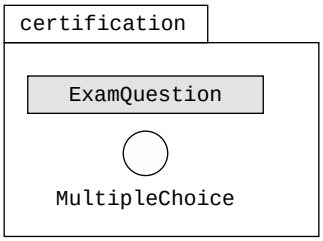

Figure 1.38 A UML representation of the `certification` package

1.6.5 Importing a single member versus all members of a package

You can import either a single member or all members (classes and interfaces) of a package using the `import` statement. First, revisit the UML notation of the `certification` package, as shown in figure 1.38.

Examine the following code for class `AnnualExam`:

```
import certification.ExamQuestion;
class AnnualExam {
    ExamQuestion eq;
    MultipleChoice mc;
}
```

Imports only ExamQuestion

Compiles okay

Will not compile

By using the wildcard character, an asterisk (`*`), you can import all of the public members, classes, and interfaces of a package. Compare the previous class definition with the following definition of class `AnnualExam`:

```
import certification.*;
class AnnualExam {
    ExamQuestion eq;
    MultipleChoice mc;
}
```

Imports all classes and interfaces from certification

Compiles okay

Also compiles okay

When you work with an IDE, it may automatically add `import` statements for classes and interfaces that you reference in your code.

1.6.6 The import statement doesn't import the whole package tree

You can't import classes from a subpackage by using an asterisk in the `import` statement. For example, the UML notation in figure 1.39 depicts the package `com.oracle.javacert` with class `Schedule` and two subpackages, `associate` and `webdeveloper`. The `associate` package contains class `ExamQuestion`, and the `webdeveloper` package contains class `MarkSheet`.

The following `import` statement will import only the `Schedule` class; it won't import classes `ExamQuestion` and `MarkSheet`:

```
import com.oracle.javacert.*;
```

Imports Schedule only

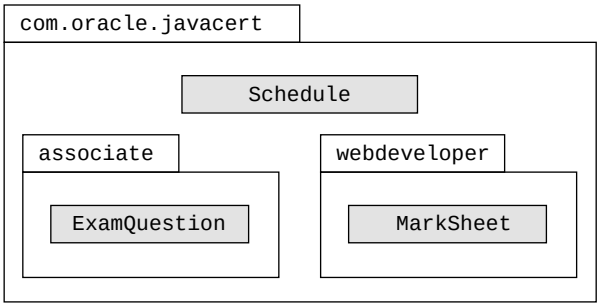

Figure 1.39 A UML representation of the `com.oracle.javacert` package and its subpackages

Similarly, the following `import` statement will import all the classes from the `associate` and `webdeveloper` packages:

```
import com.oracle.javacert.associate.*;
import com.oracle.javacert.webdeveloper.*;
```

Imports ExamQuestion only

Imports MarkSheet only

1.6.7 Importing classes from the default package

What happens if you don't include a `package` statement in your class or interface? In this case, they become part of a *default, no-name* package. This default package is automatically imported in the Java classes and interfaces defined within the *same* directory on your system.

For example, classes `Person` and `Office`, which aren't defined in an explicit package, can use each other if they're defined in the same directory:

```
class Person {
    // code
}
class Office {
    Person p;
}
```

Not defined in an explicit package

Person accessible in Office

EXAM TIP Members of a named package can't access classes and interfaces defined in the *default* package.

1.6.8 Static imports

You can import an individual static member of a class or an interface, or all its static members, by using the `import static` statement. Though accessible using an instance, the static members are usually accessed by prefixing their name with the class or interface names. By using `static import`, you can drop the prefix and just use the name of the static variable or method.

In the following code, class `ExamQuestion` defines a public static variable named `marks` and a public static method named `print()`:

```
package certification;
public class ExamQuestion {
    static public int marks;
    public static void print() {
        System.out.println(100);
    }
}
```

Public static variable marks

Public static method print()

Variable `marks` can be accessed in class `AnnualExam` using the `import static` statement. The order of the keywords `import` and `static` can't be reversed:

```
package university;
import static certification.ExamQuestion.marks;
class AnnualExam {
    AnnualExam() {
        marks = 20;
    }
}
```

Correct statement is import static, not static import

Access variable marks without prefixing it with its class name

EXAM TIP This feature is called *static imports*, but syntax is `import static`.

To use all public and static members of class `ExamQuestion` in class `AnnualExam` without importing each of them individually, you can use an asterisk with the `import static` statement:

```
package university;
import static certification.ExamQuestion.*;
class AnnualExam {
    AnnualExam() {
        marks = 20;
        print();
    }
}
```

Imports all static members of ExamQuestion

Uses marks and print() without prefixing them with their class names

Because variable `marks` and method `print()` are public, they're accessible to class `AnnualExam`. By using `import static` you don't have to prefix them with their class name. But if they were defined using any other access modifier, they wouldn't be accessible in `AnnualExam` because both these classes are defined in separate packages and `AnnualExam` doesn't inherit `ExamQuestion`.

1.7 Summary

This chapter covers the basic building blocks of the Java class design, starting with access modifiers, and then overloading and overriding methods, creating packages, and using classes from other packages.

As a Java programmer, you should understand the role of access modifiers in designing your classes. We covered how access modifiers enable a class to control who can access it, to what extent, and how.

Efficient design and implementation of an application also depends on correct and appropriate overloaded and overridden methods. You witnessed multiple examples on the need for overloading and overriding methods, including the correct ingredients. We also covered why all the methods of a base class can't be overridden.

A discussion of the nonfinal methods of class `java.lang.Object`, which is the parent class of all the Java classes, showed you why and how to override its methods. The methods of class `Object` are called by various other classes and JRE, which makes it crucial for a developer to override the relevant methods from class `Object` before shipping them off to be used by other people.

You also learned how you can use casting to refer to specific behavior of derived class objects when they're referred to their base class references. The `instanceof` operator is used to logically test whether an object is a valid type of a class or an interface.

In the final section, we worked with `package` and `import` statements. It's important to group your classes, interfaces, enums, and other Java entities depending on their functionality. In real programming projects, you'd always work with classes organized in packages.

REVIEW NOTES

This section lists the main points covered in this chapter.

Java access modifiers

- The access modifiers control the accessibility of your class and its members outside the class and package.
- Access modifiers defined by Java are `public`, `protected`, and `private`. In the absence of an explicit access modifier, a member is defined with the *default* access level.
- The `public` access modifier is the least restrictive access modifier.
- Classes and interfaces defined using the `public` access modifier are accessible to related and unrelated classes outside the package in which they're defined.
- The members of a class defined using the `protected` access modifier are accessible to classes and interfaces defined in the same package and to all derived classes, even if they're defined in separate packages.
- The members of a class defined without using an explicit access modifier are defined with package accessibility (also called default accessibility).
- The members with package access are accessible only to classes and interfaces defined in the same package.
- A class defined using default access can't be accessed outside its package.
- The private members of a class are only accessible to itself.
- The `private` access modifier is the most restrictive access modifier.
- A top-level class, interface, or enum can only be defined using the `public` or default access. They can't be defined using `protected` or `private` access.

- Method parameters and local variables can never be defined using an explicit access modifier. They don't have access control–only scope. Either they're in scope or out of scope.
- If accessibility of an existing Java entity or its member is decreased, it can break others' code.

Overloaded methods and constructors

- Overloaded methods are methods with the same name but different method parameter lists.
- A class can overload its own methods and inherited methods from its base class.
- Overloaded methods accept different lists of arguments.
- The argument lists of overloaded methods can differ in terms of change in the number, type, or position of parameters that they accept.
- Overloaded methods are bound at compile time. Unlike overridden methods they're not bound at runtime.
- A call to correctly overloaded methods can also fail compilation if the compiler is unable to resolve the call to an overloaded method.
- Overloaded methods might define a different return type or access or nonaccess modifier, but they can't be defined with only a change in their return types or access or nonaccess modifiers.
- Overloaded constructors must be defined using different argument lists.
- Overloaded constructors can't be defined by just a change in the access modifiers.
- Overloaded constructors can be defined using different access modifiers.
- A constructor can call another overloaded constructor by using the keyword `this`.
- A constructor can't invoke another constructor by using its class's name.
- If present, the call to another constructor must be the first statement in a constructor.

Method overriding and virtual method invocation

- Method overriding is an OOP language feature that enables a derived class to define a specific implementation of an existing base class method to extend its own behavior.
- A derived class can override an instance method defined in a base class by defining an instance method with the same method signature.
- Whenever you intend to override methods in a derived class, use the annotation `@Override`. It will warn you if a method can't be overridden or if you're actually overloading a method rather than overriding it.
- Overridden methods can define the same or covariant return types.
- A derived class can't override a base class method to make it less accessible.
- Overriding methods must define exactly the same method parameters; the use of a subclass or parent class results in overloading methods.

- Static methods can't be overridden. They're not polymorphic and they're bound at compile time.
- In a derived class, a static method with the same signature as that of a static method in its base class hides the base class method.
- A derived class can't override the base class methods that aren't accessible to it, such as private methods.
- Constructors cannot be overridden because a base class constructor isn't inherited by a derived class.
- A method that can be overridden by a derived class is called a virtual method.
- Virtual method invocation is invocation of the correct method–determined using the object type and not its reference.

Java packages

- You can use packages to group together a related set of classes and interfaces.
- The package and subpackage names are separated using a period.
- Classes and interfaces in the same package can access each other.
- An `import` statement allows the use of simple names for classes and interfaces defined in other packages.
- You can't use the `import` statement to access multiple classes or interfaces with the same names from different packages.
- You can import either a single member or all members (classes and interfaces) of a package using the `import` statement.
- You can't import classes from a subpackage by using the wildcard character, an asterisk (`*`), in the `import` statement.
- A class from the default package can't be used in any named package, regardless of whether it's defined within the same directory or not.
- You can import an individual static member of a class or all its static members by using an `import static` statement.
- An `import` statement can't be placed before a `package` statement in a class. Any attempt to do so will cause the compilation of the class to fail.
- The members of the default package are accessible only to classes or interfaces defined in the same directory on your system.

SAMPLE EXAM QUESTIONS

Q 1-1. Which of the following points should you incorporate in your application design?

a. Create related classes in a single package.
b. Don't make derived classes overload methods from their base class.
c. Expose the functionality of your classes using public methods.
d. Create private methods to work as helper methods for the public methods.

Q 1-2. What is the output of the following code?

```
class Wood {
    public Wood() {
        System.out.println("Wood");
    }
    {
        System.out.println("Wood:init");
    }
}
class Teak extends Wood {
    {
        System.out.println("Teak:init");
    }
    public Teak() {
        System.out.println("Teak");
    }
    public static void main(String args[]) {
        new Teak();
    }
}
```

a
```
Wood:init
Wood
Teak:init
Teak
```

b
```
Wood
Wood:init
Teak:init
Teak
```

c
```
Wood:init
Teak:init
Wood
Teak
```

d
```
Wood
Wood:init
Teak
Teak:init
```

Q 1-3. Examine the following code and select the answer options that are correct individually.

```
class Machine {
    void start() throws Exception { System.out.println("start machine"); }
}
class Laptop {
    void start() { System.out.println("Start Laptop"); }
    void start(int ms) { System.out.println("Start Laptop:"+ms); }
}
```

a Class `Laptop` overloads method `start()`.

b Class `Laptop` overrides method `start()`.

c Class `Machine` overrides method `start()`.

d Class `Machine` won't compile.
e Class `Laptop` won't compile.

Q 1-4. Given that classes `Class1` and `Class2` exist in separate packages and source code files, examine the code and select the correct options.

```
package pack1;
public class Class1 {
    protected String name = "Base";
}
```

```
package pack2;
import pack1.*;
class Class2 extends Class1{
    Class2() {
        Class1 cls1 = new Class1();      //line 1
        name = "Derived";                //line 2
        System.out.println(cls1.name);   //line 3
    }
}
```

a `Class2` can extend `Class1` but it can't access the `name` variable on line 2.
b `Class2` can't access the `name` variable on line 3.
c `Class2` can't access `Class1` on line 1.
d `Class2` won't compile.
e Line 3 will print `Base`.
f Line 3 will print `Derived`.

Q 1-5. Select the correct option.

a The declaration of private variables to store the state of an object is encouraged.
b The protected members of a class aren't accessible outside the package in which the class is defined.
c The public members of a class that's defined with default access can be accessed outside the package.
d If you change the signature or implementation of a private method, other classes that use this method cease to compile.

Q 1-6. Given the following code

```
interface Scavenger{}
class Bird{}
class Parrot extends Bird{}
class Vulture extends Bird implements Scavenger{}
```

```
class BirdSanctuary {
    public static void main(String args[]) {
        Bird bird = new Bird();
        Parrot parrot = new Parrot();
```

```
        Vulture vulture = new Vulture();
        //INSERT CODE HERE
    }
}
```

In which of the following options will the code, when inserted at `//INSERT CODE HERE`, throw a `ClassCastException`?

a. `Vulture vulture2 = (Vulture)parrot;`
b. `Parrot parrot2 = (Parrot)bird;`
c. `Scavenger sc = (Scavenger)vulture;`
d. `Scavenger sc2 = (Scavenger)bird;`

Q 1-7. Assuming that all of the following classes are defined in separate source code files, select the incorrect statements.

```
package solarfamily;
public class Sun {
    public Sun() {}
}
```

```
package stars;
public class Sun {
    public Sun() {}
}
```

```
package skyies;
import stars.Sun;           // line1
import solarfamily.Sun;     // line2
class Sky {
    Sun sun = new Sun();    // line 3
}
```

a. Code compilation fails at line 1.
b. Code compilation fails at line 2.
c. Code compilation fails at line 3.
d. The code compiles successfully and class `Sky` creates an object of class `Sun` from the `stars` package.
e. The code compiles successfully and class `Sky` creates an object of class `Sun` from the `solarfamily` package.

Q 1-8. Select the correct options.

```
class Color {
    String name;
    Color(String name) {this.name = name;}
    public String toString() {return name;}
    public boolean equals(Object obj) {
        return (obj.toString().equals(name));
    }
}
```

a Class `Color` overrides method `toString()` correctly.
b Class `Color` overrides method `equals()` correctly.
c Class `Color` fails to compile.
d Class `Color` throws an exception at runtime.
e None of the above.

Q 1-9. Given the following code

```
class Book {
    String isbn;
    Book(String isbn) {this.isbn = isbn;}
    public int hashCode() {
        return 87536;
    }
}
```

Select the correct option.

a Objects of the class `Book` can never be used as keys because the corresponding objects wouldn't be retrievable.
b Method `hashCode()` is inefficient.
c Class `Book` will not compile.
d Though objects of class `Book` are used as keys, they will throw an exception when the corresponding values are retrieved.

Q 1-10. What is the output of the following code?

```
class Wood {
    String wood = "Wood";
    public Wood() {
        wood = "Wood";
    }
    {
        wood = "init:Wood";
    }
}
class Teak extends Wood {
    String teak;
    {
        teak = "init:Teak";
    }
    public Teak() {
        teak = "Teak";
    }
    public static void main(String args[]) {
        Teak teak = new Teak();
        System.out.println(teak.wood);
        System.out.println(teak.teak);
    }
}
```

a
```
init:Wood
init:Teak
```
b
```
init:Wood
Teak
```
c
```
Wood
init:Teak
```
d
```
Wood
Teak
```

Q 1-11. Given the following code

```
class Cloth {}
class Shirt extends Cloth implements Resizable{}
class Shorts extends Cloth {}
interface Resizable {}

class Factory {
    public static void main(String sr[]) {
        Shirt s = new Shirt();
        //INSERT CODE HERE
        System.out.println(res);
    }
}
```

Which options will print `true`?

a
```
boolean res = new Cloth() instanceof Shirt;
```
b
```
boolean res = new Shirt() instanceof Resizable;
```
c
```
boolean res = null instanceof Factory;
```
d
```
Cloth cloth = new Cloth();
Shirt shirt = new Shirt();
boolean res = shirt instanceof cloth;
```

ANSWERS TO SAMPLE EXAM QUESTIONS

A 1-1. a, c, d

[1.1] Use access modifiers: private, protected, and public
[1.3] Overload constructors and methods
[1.7] Use package and import statements

Explanation: Option (a) is correct. A package enables you to create a namespace to group related classes and interfaces together.

Option (b) is incorrect. A base class overloads its base class method, as required. Making derived classes overload their base class methods doesn't make it an incorrect or inefficient design.

Options (c) and (d) are also correct. The functionality of your classes should be exposed using the public methods. The private methods are called within the class in which they're defined. They usually work as helper methods.

A 1-2. a

[1.3] Overload constructors and methods

Explanation: When a class is compiled, the contents of its initializer block are added to its constructor, just before its own contents. For example, here's the decompiled code for class `Wood`. As you can see, the contents of its initializer block are added to its constructor:

```
class Wood
{
    public Wood()
    {
        System.out.println("Wood:init");
        System.out.println("Wood");
    }
}
```

A 1-3. a

[1.2] Override methods
[1.3] Overload constructors and methods

Explanation: Class `Laptop` correctly overloads the method `start()` by defining a different parameter list.

Options (b) and (c) are incorrect because classes `Laptop` and `Machine` are unrelated. A derived class can override its base class method.

Method `start()` qualifies as a valid overridden method in class `Laptop`, if `Laptop` extends class `Machine`. It's acceptable for an overriding method to not throw any checked exception, even if the base class method is throwing a checked exception.

Options (d) and (e) are incorrect because both classes will compile successfully.

A 1-4. b, d

[1.7] Use package and import statements

Explanation: A derived class can access a protected member of its base class, across packages, directly. But if the base and derived classes are in separate packages, then you can't access protected members of the base class by using reference variables of class `Base` in a derived class. So, `Class2` doesn't compile.

Options (e) and (f) are incorrect because `Class2` won't compile.

A 1-5. a

[1.1] Use access modifiers: private, protected, and public
[1.7] Use package and import statements

Explanation: Option (b) is incorrect because the protected members of a class are accessible by the derived classes, outside the package in which the class is defined.

Option (c) is incorrect because a class with default access isn't visible outside the package within which it's defined. If the class isn't visible itself, it doesn't matter whether its members are accessible or not.

Option (d) is incorrect because a private method can't be used outside the class in which it's defined.

A 1-6. b, d

[1.4] Use the instanceof operator and casting

Explanation: `ClassCastException` is thrown at runtime. So the options that don't fail to compile are eligible to be considered for the following question: Will they throw a `ClassCastException`?

Option (a) is incorrect because it fails to compile.

Option (b) is correct because classes `Bird` and `Parrot` are in the same hierarchy tree, so an object of base class `Bird` can be explicitly casted to its derived class `Parrot` at compilation. But the JVM can determine the type of the objects at runtime. Because an object of a derived class can't refer to an object of its base class, this line throws a `ClassCastException` at runtime.

Option (c) is incorrect because class `Vulture` implements the interface `Scavenger`, so this code will also execute without the explicit cast.

Option (d) is correct. An instance of a nonfinal class can be casted to any interface type using an explicit cast during the compilation phase. But the exact object types are validated during runtime and a `ClassCastException` is thrown if the object's class doesn't implement that interface. Class `Bird` doesn't implement the interface `Scavenger` and so this code fails during runtime, throwing a `ClassCastException`.

A 1-7. b

[1.7] Use package and import statements

Explanation: Class `Sky` fails with the following error message:

```
Sky.java:3: error: stars.Sun is already defined in a single-type import
import solarfamily.Sun;
^
1 error
```

A 1-8. a

[1.2] Override methods

Explanation: Class `Color` overrides method `toString()` correctly, but not method `equals()`. According to the contract of method `equals()`, for any non-null reference

values `x` and `y`, `x.equals(y)` should return `true` if and only if `y.equals(x)` returns `true`—this rule states that two objects should be comparable to each other in the same way. Class `Color` doesn't follow this rule. Here's the proof:

```
class TestColor {
    public static void main(String args[]) {
        Color color = new Color("red");
        String string = "red";

        System.out.println(color.equals(string));   // prints true
        System.out.println(string.equals(color));   // prints false
    }
}
```

A 1-9. b

[1.6] Override the hashCode, equals, and toString methods from the Object class to improve the functionality of your class

Explanation: Method `hashCode()` returns the same hash code for all the objects of this class. This essentially makes all the values be stored in the same bucket if objects of the preceding classes are used as keys in class `HashMap` (or similar classes that use hashing), and reduces it to a linked list, drastically reducing its efficiency.

Option (a) in incorrect. `Book` instances can be used to retrieve corresponding key values but only in limited cases—when you use the same keys (instances) to store and retrieve values. Even though `hashCode()` will return the same value for different `Book` instances, `equals()` will always compare the reference variables and not their values, returning `false`.

A 1-10. d

[1.3] Overload constructors and methods

Explanation: When a class is compiled, the contents of its initializer block are added to its constructor just before its own contents. For example, here's the decompiled code for class `Wood`. As you can see, the contents of its initializer block are added to its constructor:

```
class Wood
{
    public Wood()
    {
        wood = "Wood";           // initial initialization
        wood = "init:wood";      // re-assignment by the initializer block
        wood = "Wood";           // re-assignment by the constructor
    }
    String wood;
}
```

A 1-11. b

[1.4] Use the instanceof operator and casting

Explanation: Option (a) prints `false`.

Option (c) prints `false`. It doesn't fail to compile because `null` is a valid literal value that can be used for objects.

Option (d) fails to compile. The `instanceof` operator must be followed by the name of an interface, class, or enum.