Object Lifetime 10

Chapter Topics

- Understanding automatic garbage collection and guidelines for facilitating garbage collection
- Using static and instance initializers, in particular declaration order, forward references, and exception handling
- Understanding the role played by initializers and constructors in initializing objects, classes, and interfaces

Java SE 17 Developer Exam Objectives	
[3.1] Declare and instantiate Java objects including nested class objects, and explain the object life-cycle including creation, reassigning references, and garbage collection ❍ *Object lifecycle and garbage collection are covered in this chapter.* ❍ *For constructing objects, see* Chapter 3, p. 97. ❍ *For nested classes, see* Chapter 9, p. 489.	§10.1, p. 533 *to* §10.4, p. 537
[3.2] Create classes and records, and define and use instance and static fields and methods, constructors, and instance and static initializers ❍ *Initializers and object construction are covered in this chapter.* ❍ *For declaring normal classes and defining class members and constructors, see* Chapter 3, p. 97. ❍ *For records, see* §5.14, p. 299.	§10.5, p. 540 *to* §10.9, p. 555

Java SE 11 Developer Exam Objectives	
[3.1] Declare and instantiate Java objects including nested class objects, and explain objects' lifecycles (including creation, dereferencing by reassignment, and garbage collection)	§10.1, p. 533 *to*

❍ *Object lifecycle and garbage collection are covered in this chapter.* §10.4, p. 537

❍ *For constructing objects, see* Chapter 3, p. 97.

❍ *For nested classes, see* Chapter 9, p. 489.

Java SE 11 Developer Exam Objectives

[3.3] Initialize objects and their members using instance and static initialiser statements and constructors — §10.5, p. 540 *to* §10.9, p. 555

❍ *Initializers and object construction are covered in this chapter.*

❍ *For initializing objects using initialization statements and constructors, see* Chapter 3, p. 97.

One of the most important features of Java is its *automatic memory management*, also know as *automatic garbage collection*. This chapter provides a non-technical discussion of this important topic about reclaiming storage of unused objects in a program at runtime. In addition, we look at the role of initializer expressions, initializer blocks, and constructors to initialize the state of an object when it is created with the `new` operator, and which includes the construction of the state from its superclasses.

10.1 Garbage Collection

Efficient memory management is essential in a runtime system. Storage for objects is allocated in a designated part of the memory called the *heap*, which has a finite size. Garbage collection (GC) is a process of managing the heap efficiently, by reclaiming memory occupied by objects that are no longer needed and making it available for new objects. Java provides automatic garbage collection, meaning that the runtime environment can take care of memory management without the program having to take any special action. Objects allocated on the heap (through the `new` operator) are administered by the automatic garbage collector. The automatic garbage collection scheme guarantees that a reference to an object is always valid while the object is needed by the program. Specifically, the object will not be reclaimed if it will result in a *dangling reference*—that is, a reference to an object that no longer exists.

Having an automatic garbage collector frees the programmer from the responsibility of writing code for deleting objects. By relying on the automatic garbage collector, a Java program also forfeits any significant influence on the garbage collection of its objects (p. 537). However, this price is insignificant when compared to the cost of putting the code for object management in place and plugging all the memory leaks. Time-critical applications should recognize that the automatic garbage collector runs as a background task and may have a negative impact on their performance.

10.2 Reachable Objects

An automatic garbage collector essentially performs two tasks:

- Decides if and when memory needs to be reclaimed
- Finds objects that are no longer needed by the program and reclaims their storage

A program has no guarantees that the automatic garbage collector will be run during its execution. Consequently, a program should not rely on the scheduling of the automatic garbage collector for its behavior (p. 537).

To understand how the automatic garbage collector finds objects whose storage should be reclaimed, we need to look at the activity happening in the JVM. Java provides *thread-based multitasking*, meaning that several threads can be executing concurrently in the JVM, each doing its own task. A *thread* is an independent path of execution through the program code. A thread is alive if it has not completed its execution. Each live thread has its own JVM stack (§7.1, p. 365). The JVM stack contains activation frames of methods that are currently active. Local references declared in a method can always be found in the method's activation frame, stored on the JVM stack associated with the thread in which the method is called. Objects, in contrast, are always created on the heap. If an object has a field reference, the field will be found inside the object in the heap, and the object denoted by the field reference will also be found in the heap.

An example of how memory is organized during execution is depicted in **Figure 10.1**, which shows two live threads (t_1 and t_2) and their respective JVM stacks with the activation frames. The diagram indicates which objects in the heap are referenced by local references in the method activation frames. It also identifies field references in objects, which refer to other objects in the heap. Some objects have several aliases.

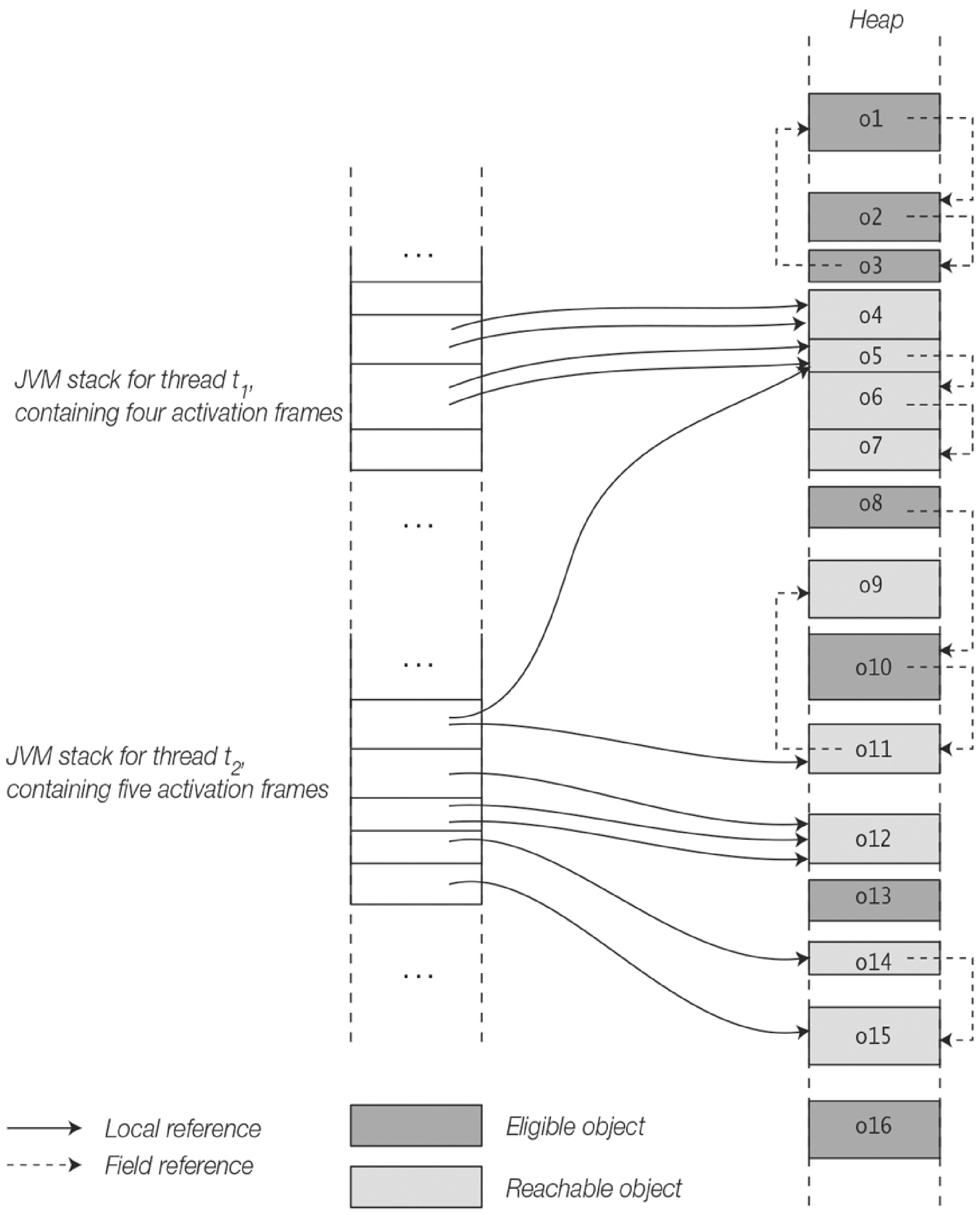
An object in the heap is said to be *reachable* if it is referenced by any *local* reference in a JVM stack. Likewise, any object that is denoted by a reference in a reachable object is said to be reachable. Reachability is a transitive relationship. Thus a reachable object has at least one chain of reachable references from the JVM stack. Any reference that makes an object reachable is called a *reachable reference*. An object that is not reachable is said to be *unreachable*.

A reachable object is *alive*, and is *accessible* by a live thread. Note that an object can be accessible by more than one thread. Any object that is *not* accessible by a live thread is a candidate for garbage collection. When an object becomes unreachable and is waiting for its memory to be reclaimed, it is said to be *eligible* for garbage collection. An object is eligible for garbage collection if all references denoting it are in eligible objects. Eligible objects do not affect the future course of program execution. When the garbage collector runs, it finds and reclaims the storage of eligible objects, although garbage collection does not necessarily occur as soon as an object becomes unreachable.

In **Figure 10.1**, the objects `o4`, `o5`, `o11`, `o12`, `o14`, and `o15` all have reachable references. Objects `o13` and `o16` have no reachable references, and therefore, are eligible for garbage collection.

From the preceding discussion we can conclude that if a compound object becomes unreachable, its constituent objects also become unreachable, barring any reachable references to the constituent objects. Although the objects `o1`, `o2`, and `o3` in **Figure 10.1** form a circular list, they do not have any reachable references. Thus these objects are all eligible for garbage collection. Conversely, the objects `o5`, `o6`, and `o7` form a linear list, but they are all reachable, as the first object in the list, `o5`, is reachable. The objects `o8`, `o10`, `o11`, and `o9` also form a linear list (in that order), but not all objects in the list are reachable. Only the objects `o9` and `o11` are reachable, as object `o11` has a reachable reference. The objects `o8` and `o10` are eligible for garbage collection.

The *lifetime* of an object is the time from its creation to the time it is garbage collected. Under normal circumstances, an object is accessible from the time when it is created to the time when it becomes unreachable. The lifetime of an object can also include a period when it is eligible for garbage collection, waiting for its storage to be reclaimed.



Heap
o1
o2
o3
o4
o5
o6
o7
o8
o9
o10
o11
o12
o13
o14
o15
o16
JVM stack for thread t_1, containing four activation frames
JVM stack for thread t_2, containing five activation frames
Local reference
Field reference
Eligible object
Reachable object

Figure 10.1 *Memory Organization at Runtime*

10.3 Facilitating Garbage Collection

The automatic garbage collector determines which objects are not reachable, and therefore, are eligible for garbage collection. It will certainly go to work if there is an imminent memory shortage. Even so, automatic garbage collection should not be perceived as a license for creating a plethora of objects and then forgetting about them. Nevertheless, certain programming practices can help in minimizing the overhead associated with garbage collection during program execution.

Certain objects, such as files and network connections, can tie up resources and should be disposed of properly when they are no longer needed. In most cases, the `try` -with-resources statement (§7.7, p. 407) provides a convenient facility for such purposes, as it will always ensure proper closing of any `AutoCloseable` resources.

To optimize its memory footprint, a live thread should retain access to an object for only as long as the object is needed for its execution. The program can allow objects to become eligible for garbage collection as early as possible by removing all references to an object when it is no longer needed.

Objects that are created and accessed by local references in a method are eligible for garbage collection when the method terminates, unless reference values to these objects are exported out of the method. This can occur if a reference value is returned from the method, passed as an argument to another method that records the reference value, or thrown as an exception. However, a method need not always leave objects to be garbage collected after its termination. It can facilitate garbage collection by taking suitable action—for example, by nulling references.

Click here to view code image

```
import java.io.*;

class WellBehavedClass {
  // ...
  void wellBehavedMethod() {
    FileReader reader;
    long[] bigArray = new long[20000];
    // ... uses local variables ...
    // Does clean-up (before starting something extensive)
    reader = null;                    // (1)
    bigArray = null;                  // (2)

    // Start some other extensive activity
    // ...
  }
  // ...
}
```

In this code, the local variables are set to `null` after use at (1) and (2), before starting some other extensive activity. This makes the objects denoted by the local variables eligible for garbage collection from this point onward, rather than after the method terminates. This optimization technique of nulling references should be used only as a last resort when resources are scarce.

Here are some other techniques to facilitate garbage collection:

- When a method returns a reference value and the object denoted by the value is not needed, not assigning this value to a reference facilitates garbage collection.
- If a reference is assigned a new value, the object that was previously denoted by the reference can become eligible for garbage collection.
- Removing reachable references to a compound object can make the constituent objects become eligible for garbage collection, as explained earlier.

10.4 Invoking Garbage Collection Programmatically

Although Java provides facilities to invoke the garbage collector explicitly, there are no guarantees that it will be run. The program can request that garbage collection be performed, but there is no way to force garbage collection to be activated.

The `System.gc()` method can be used to request garbage collection.

```
static void gc()
```

Requests that garbage collection be run.

Alternatively, corresponding methods in the `Runtime` class can be used. A Java application has a unique `Runtime` object that can be used by the application to interact with the JVM. An application can obtain this object by calling the method `Runtime.getRuntime()`. The `Runtime` class provides several methods related to memory issues:

```
static Runtime getRuntime()
```

Returns the `Runtime` object associated with the current application.

```
void gc()
```

Requests that garbage collection be run. There are no guarantees that it will be run. It is recommended to use the more convenient `static` method `System.gc()`.

```
long freeMemory()
```

Returns the amount of free memory (bytes) in the JVM that is available for new objects.

```
long totalMemory()
```

Returns the total amount of memory (bytes) available in the JVM, including both memory occupied by current objects and memory available for new objects.

The following points regarding automatic garbage collection should be noted:

- Trying to initiate garbage collection programmatically does not guarantee that it will actually be run.
- Garbage collection might not even be run if the program execution does not warrant it. Thus any memory allocated during program execution might remain allocated after program termination, but will eventually be reclaimed by the operating system.
- There are also no guarantees about the order in which the objects will be garbage collected. Therefore, the program should not make any assumptions based on this criteria.
- Garbage collection does not guarantee that there will be enough memory for the program to run. A program can rely on the garbage collector to run when memory gets very low, and it can expect an `OutOfMemoryError` to be thrown if its memory demands cannot be met.

Review Questions

10.1 Which of the following statements is true?

Select the one correct answer.

a. Objects can be explicitly destroyed using the keyword `delete`.

b. An object will be garbage collected immediately after it becomes unreachable.

c. If object `obj1` is accessible from object `obj2,` and object `obj2` is accessible from `obj1`, then `obj1` and `obj2` are not eligible for garbage collection.

d. If object `obj1` can access object `obj2` that is eligible for garbage collection, then `obj1` is also eligible for garbage collection.

10.2 Identify the location in the following program where the object, initially referenced by `arg1`, is eligible for garbage collection.

Click here to view code image

```
public class MyClass {
  public static void main(String[] args) {
    String msg;
    String pre = "This program was called with ";
    String post = " as first argument.";
    String arg1 = new String((args.length > 0) ? "'" + args[0] + "'" :
                             "<no argument>");
    msg = arg1;
    arg1 = null;             // (1)
    msg = pre + msg + post;  // (2)
    pre = null;              // (3)
    System.out.println(msg);
    msg = null;              // (4)
    post = null;             // (5)
    args = null;             // (6)
  }
}
```

Select the one correct answer.

a. After the line at (1).

b. After the line at (2).

c. After the line at (3).

d. After the line at (4).

e. After the line at (5).

f. After the line at (6).

10.3 How many objects are eligible for garbage collection when control reaches (1)?

Click here to view code image

```
public class Elements {
  public static void main(String[] args) {
    int[] array = new int[4];
    for (int i = 0; i < 4; i++) {
      array[i] = i;
    }
    array[0] = array[1] = array[2] = array[3] = 0;
    System.gc();                            // (1);
  }
}
```

Select the one correct answer.

a. 0

b. 1

c. 4

d. It’s hard to say.

10.4 How many objects are eligible for garbage collection when control reaches (1)?

Click here to view code image

```
public class Link {
  private Link next;
  Link(Link next) { this.next = next; }

  public static void main(String[] args) {
    Link p = null;
    for (int i = 0; i < 5; i++) {
      p = new Link(p);
    }
    System.gc();                            // (1)
  }
}
```

Select the one correct answer.

a. 0

b. 5

c. 10

d. It’s hard to say.

10.5 How many objects are eligible for garbage collection when control reaches (1)?

```
public class Song {
```

Click here to view code image

```
public class Album {
  private Song[] songs = { new Song(), new Song() };
  public Song[] getSongs() {
    return songs;
  }

  public void removeAll() {
    songs = null;
  }
}
```

Click here to view code image

```
public class Test {
  public static void main(String[] args) {
    Album album = new Album();
    Song[] songs = album.getSongs();
    album.removeAll();
    System.out.println(album.getSongs());
    // Line (1)
  }
}
```

Select the one correct answer.

a. 0

b. 1

c. 2

d. 3

e. 4

10.5 Initializers

Initializers can be used to set initial values for fields in objects and classes. There are three kinds of initializers:

- *Field initializer expressions*
- *Static initializer blocks*
- *Instance initializer blocks*

Subsequent sections in this chapter provide details on these initializers, concluding with a discussion of the procedure involved in constructing the state of an object when the object is created using the `new` operator.

For brevity, we have dispensed with the access modifier for fields in most class declarations in the rest of this chapter, as field accessibility by clients is not of primary concern here.

10.6 Field Initializer Expressions

Initialization of fields can be specified in field declaration statements using initializer expressions. The value of the initializer expression must be assignment compatible with the declared field. We distinguish between static and non-static field initializers.

Click here to view code image

```
class ConstantInitializers {
         int minAge = 12;              // (1) Non-static

  static double pensionPoints = 10.5; // (2) Static
  // ...
}
```

The fields of an object are initialized with the values of initializer expressions when the object is created by using the `new` operator. In the previous example, the declaration at (1) will result in the field `minAge` being initialized to `12` in every object of the class `ConstantInitializers` created with the `new` operator. If no explicit initializer expressions are specified, default values are assigned to the fields.

When a class is loaded, it is initialized, meaning its static fields are initialized with the values of the initializer expressions. The declaration at (2) will result in the static field `pensionPoints` being initialized to `10.5` when the class is loaded by the JVM. Again, if no explicit initializers are specified, default values are assigned to the static fields.

An initializer expression for a static field cannot refer to non-static members by their simple names. The keywords `this` and `super` cannot occur in a static initializer expression.

Since a class is always initialized before it can be instantiated, an instance initializer expression can always refer to any static member of a class, regardless of the member declaration order. In the following code, the instance initializer expression at (1) refers to the static field `NO_OF_WEEKS` declared and initialized at (2). Such a *forward reference* is legal. More examples of forward references are given in the next subsection.

Click here to view code image

```
class MoreInitializers {
         int noOfDays    = 7 * NO_OF_WEEKS;    // (1) Non-static
  static int NO_OF_WEEKS = 52;                 // (2) Static
  // ...
}
```

Initializer expressions can also be used to define constants in interfaces (§5.6, p.254). Such initializer expressions are implicitly static, as they define values of `static final` fields in an interface.

Initializer expressions are used to initialize local variables as well (§3.4, p.102). A local variable is initialized with the value of the initializer expression every time the local variable declaration is executed.

Declaration Order of Initializer Expressions

When an object is created using the `new` operator, instance initializer expressions are executed in the order in which the instance fields are declared in the class.

Java requires that the declaration of a field must occur *before its usage* in any initializer expression if the field is *used on the right-hand side of an assignment* in the initializer expression. This essentially means that the declaration of a field must occur before the value of the field is *read* in an initializer expression. Using the field on the left-hand side of an assignment in the initializer expression does not violate the *declaration-before-reading rule*, as this constitutes a *write* operation. This rule applies when the usage of the field is by its *simple name*.

There is one caveat to the declaration-before-reading rule: It does not apply if the initializer expression defines an anonymous class, as the usage then occurs in a different class that has its own accessibility rules in the enclosing context. The restrictions outlined earlier help to detect initialization anomalies at compile time.

In the next code example, the initialization at (2) generates a compile-time error because the field `width` in the initializer expression violates the declaration-before-reading rule. Because the usage of the field `width` in the initializer expression at (2) does not occur on the left-hand side of the assignment, this is an illegal forward reference. To remedy the error, the declaration of the field `width` at (4) can be moved in front of the declaration at (2). In any case, we can use the keyword `this` as shown at (3), but it will read the default value `0` in the field `width`.

Click here to view code image

```
class NonStaticInitializers {
  int length  = 10;                    // (1)
//double area = length * width;        // (2) Not OK. Illegal forward reference.
  double area = length * this.width;   // (3) OK, but width has default value 0.
  int width   = 10;                    // (4)

  int sqSide = height = 20;            // (5) OK. Legal forward reference.
  int height;                          // (6)
}
```

The forward reference at (5) is legal. The usage of the field `height` in the initializer expression at (5) occurs on the left-hand side of the assignment. The initializer expression at (5) is evaluated as `(sqSide = (height = 20))`. Every object of the class `NonStaticInitializers` will have the fields `height` and `sqSide` set to the value `20`.

The declaration-before-reading rule is equally applicable to static initializer expressions when static fields are referenced by their simple names.

Example 10.1 shows why the order of field initializer expressions can be important. The initializer expressions in this example are calls to methods defined in the class, and methods are not subject to the same access rules as initializer expressions. The call at (2) to the method `initMaxGuests()` defined at (4) is expected to return the maximum number of guests, but the field `occupancyPerRoom` at (3) will not have been explicitly initialized at this point; therefore, its default value `0` will be used in the method `initMaxGuests()`, which will return a logically incorrect value. The program output shows that after object creation, the occupancy per room is correct, but the maximum number of guests is wrong.

Example 10.1 *Initializer Expression Order and Method Calls*

Click here to view code image

```
// File: TestOrder.java
class Hotel {
  private int noOfRooms        = 12;                                   // (1)
  private int maxNoOfGuests    = initMaxGuests();                    // (2) Bug
  private int occupancyPerRoom = 2;                                  // (3)
  public int initMaxGuests() {                                       // (4)
    System.out.println("occupancyPerRoom: " + occupancyPerRoom);
    System.out.println("maxNoOfGuests:    " + noOfRooms * occupancyPerRoom);
    return noOfRooms * occupancyPerRoom;
  }

  public int getMaxGuests() { return maxNoOfGuests; }                // (5)

  public int getOccupancy() { return occupancyPerRoom; }             // (6)
}
//________________________________________________________________________
public class TestOrder {
  public static void main(String[] args) {
    Hotel hotel = new Hotel();                                       // (7)
    System.out.println("AFTER OBJECT CREATION");
    System.out.println("occupancyPerRoom: " + hotel.getOccupancy()); // (8)
    System.out.println("maxNoOfGuests:    " + hotel.getMaxGuests()); // (9)
  }
}
```

Output from the program:

```
occupancyPerRoom: 0
maxNoOfGuests:    0
AFTER OBJECT CREATION
occupancyPerRoom: 2
maxNoOfGuests:    0
```

Exception Handling and Initializer Expressions

Initializer expressions in named classes and interfaces must not result in any uncaught checked exception (§7.2, p.374). If any checked exception is thrown during execution of an initializer expression, it must be caught and handled by code called from the initializer expression. This restriction does not apply to instance initializer expressions in anonymous classes.

Example 10.2 illustrates exception handling for initializer expressions in named classes. The static initializer expression at (3) calls the static method `createHotelPool()` at (4), which catches and handles the checked `TooManyHotelsException` defined at (2). If the method `createHotelPool()` were to use the `throws` clause to specify the checked exception, instead of catching and handling it within a `try`-`catch` block, the initializer expression at (3), which called the method, would have to handle the exception. However, the initializer expression cannot specify any exception handling, as the compiler would complain.

The instance initializer expression at (5) calls the method `initMaxGuests()` at (6), which can throw the unchecked `RoomOccupancyTooHighException`. If thrown, this exception will be caught and handled in the `main()` method. Program output confirms that an unchecked `RoomOccupancyTooHighException` was thrown during program execution.

Example 10.2 *Exceptions in Initializer Expressions*

Click here to view code image

```
// File: ExceptionsInInitializers.java
class RoomOccupancyTooHighException
      extends RuntimeException {}                     // (1) Unchecked Exception
class TooManyHotelsException
      extends Exception {}                            // (2) Checked Exception
//_____________________________________________________________________________
class Hotel {
  // Static Members
  private static int noOfHotels = 12;
  private static Hotel[] hotelPool = createHotelPool();   // (3)

  private static Hotel[] createHotelPool() {              // (4)
    try {
      if (noOfHotels > 10)
        throw new TooManyHotelsException();
    } catch (TooManyHotelsException e) {
      noOfHotels = 10;
      System.out.println("No. of hotels adjusted to " + noOfHotels);
    }
    return new Hotel[noOfHotels];
  }
  // Instance Members
  private int noOfRooms        = 215;
  private int occupancyPerRoom = 5;
  private int maxNoOfGuests    = initMaxGuests();         // (5)

  private int initMaxGuests() {                           // (6)
    if (occupancyPerRoom > 4)
      throw new RoomOccupancyTooHighException();
    return noOfRooms * occupancyPerRoom;
  }
}
//_____________________________________________________________________________
public class ExceptionsInInitializers {
  public static void main(String[] args) {
    try { new Hotel(); }
    catch (RoomOccupancyTooHighException exception) {
      exception.printStackTrace();
    }
  }
}
```

Output from the program:

Click here to view code image

```
No. of hotels adjusted to 10
RoomOccupancyTooHighException
      at Hotel.initMaxGuests(ExceptionsInInitializers.java:29)
      at Hotel.<init>(ExceptionsInInitializers.java:25)
      at ExceptionsInInitializers.main(ExceptionsInInitializers.java:36)
```

10.7 Static Initializer Blocks

Java allows static initializer blocks to be defined in a class. Although such blocks can include arbitrary code, they are primarily used for initializing static fields. The code in a static initializer block is executed only once, when the class is loaded and initialized.

Local variables in static and instance initializer blocks can be declared with the reserved type name `var`—that is, local variable type inference using `var` is permitted for local variable declarations in initializer blocks. Code in the rest of this chapter shows many examples of such declarations.

The syntax of a static initializer block comprises the keyword `static` followed by a local block that can contain arbitrary code, as shown at (3) in the declaration of the following class:

Click here to view code image

```
class MatrixData {

  static final int ROWS = 12, COLUMNS = 10;          // (1)
  static long[][] matrix = new long[ROWS][COLUMNS];  // (2)
  // ...
  static {                                           // (3) Static initializer
    for (int i = 0; i < matrix.length; i++)
      for (int j = 0; j < matrix[i].length; j++)
        matrix[i][j] = 2*i + j;
    }
    // ...
  }
```

When the class `MatrixData` is first loaded, the `static final` fields at (1) are initialized. Then the array of arrays `matrix` of specified size is created at (2), followed by the execution of the static block at (3).

Note that the static initializer block is not contained in any method. A class can have more than one static initializer block. Initializer blocks are *not* members of a class, and they cannot have a `return` statement because they cannot be called directly.

When a class is initialized, the initializer expressions in `static` field declarations and `static` initializer blocks are executed in the order in which they are specified in the class. In the class `MatrixData`, the initializer expressions at (1) and (2) are executed before the static initializer block at (3).

Similar restrictions apply to static initializer blocks as for static initializer expressions: The keywords `this` and `super` cannot occur in a static initializer block, as such a block defines a static context.

Declaration Order of Static Initializers

In the class `ScheduleV1` below, the `static` field declaration at (1) has a *forward reference* to the `static` field `numOfWeeks` which has not been declared yet. The simple name of the `static` field `numOfWeeks` cannot be used in the initializer expression at (1) before its declaration.

Click here to view code image

```
  public class ScheduleV1 {
  //static int numOfDays  = 7 * numOfWeeks; // (1) Compile-time error! Simple name.
    static int numOfWeeks = 52;             // (2)
  }
```

The code will compile if we change the order of the declarations, and the `static` fields `numOfWeeks` and `numOfDays` will be initialized correctly with the values `52` and `364`, respectively.

Click here to view code image

```
  public class ScheduleV2 {
    static int numOfWeeks = 52;                         // (2) 52
    static int numOfDays  = 7 * numOfWeeks;                // (1) 364
  }
```

The code will also compile if the *class name* is used to access the field, but the `static` field `numOfDays` will *not* be initialized correctly.

Click here to view code image

```
  public class ScheduleV3 {
    static int numOfDays  = 7 * ScheduleV3.numOfWeeks;  // (1) 0
    static int numOfWeeks = 52;                           // (2) 52
  }
```

The code above is actually executed as follows, with the default value `0` of the `static` field `numOfWeeks` being used in the initializer expression at (1):

Click here to view code image

```
  public class ScheduleV3 {
    static int numOfDays;                                 // Default value: 0
    static int numOfWeeks;                                // Default value: 0
    static {
      numOfDays  = 7 * numOfWeeks;                        // (1) 0
      numOfWeeks = 52;                                    // (2) 52
    }
  }
```

However, the `static` field `numOfDays` will be initialized correctly if the `static` field `numOfWeeks` is declared `final`.

Click here to view code image

```
  public class ScheduleV4 {
    static int numOfDays  = 7 * ScheduleV4.numOfWeeks;  // (1) 364
    static final int numOfWeeks = 52;                     // (2) final: 52
  }
```

The initializer expression `52` for the `static` field `numOfWeeks` at (2) is a *constant expression* of type `int`, which is an `int` literal. The compiler is able to compute the value of a constant expression. A `final` variable of either a primitive type or type `String` that is initialized with a constant expression is called a *constant variable*—that is, once initialized, the value of a `final` variable cannot be changed (§5.5, p. 230). At (2), the `final static` field `numOfWeeks` is a constant variable. During class initialization, such `final static` fields are always initialized first, before any other initializers are executed. Such a constant field never gets initialized with the default value of its type at runtime. Rearrangement of code done by the compiler in this case is equivalent to the following code:

Click here to view code image

```
public class ScheduleV4 {
  static final int numOfWeeks = 52;                   // Constant variable: 52
  static int numOfDays;                               // Default value: 0
  static {
    numOfDays  = 7 * numOfWeeks;                      // (1) 364
  }
}
```

Static fields should be accessed statically—that is, using the class name—which is the best policy, but care should be exercised in the order of their declaration and initialization.

When making forward references using *simple names*, code in a `static` initializer is subject to the declaration-before-reading rule. Note that this rule applies only when the use of the field is by its simple name. Using the class name to access a `static` field is never a problem.

Example 10.3 illustrates forward references and the order of execution for `static` initializer expressions in field declarations and code in `static` initializer blocks. An illegal forward reference occurs at (4), where an attempt is made to read the value of the field `sf1` before its declaration. At (11) the read operation is after the declaration, and therefore, allowed. Forward reference made on the left-hand side of the assignment is always allowed, as shown at (2), (5), and (7). The initializers are executed in their declaration order. A `static` field has the value it was last assigned in an initializer. If there is no explicit assignment, the `static` field has the default value of its type. Referring to a `static` field using the class name is always allowed.

Declaring local variables using the reserved word `var` in static initializer blocks can be found at (5) and (12) in Example 10.3.

Example 10.3 *Static Initializers and Forward References*

Click here to view code image

```
package refs1;
```

```
public class ForwardRefs {

  static {                    // (1) Static initializer block
    System.out.printf("Enter static block 1: sf1=%s, sf2=%s%n",
        ForwardRefs.sf1, ForwardRefs.sf2); // Enter static block 1: sf1=0, sf2=0

    sf1 = 10;                 // (2) OK. Assignment to sf1 allowed
//  sf1 = if1;           // (3) Not OK. Non-static field access in static context
//  int a = 2 * sf1;     // (4) Not OK. Read operation before declaration
    var b = sf1 = 20;         // (5) OK. Assignment to sf1 allowed
    int c = ForwardRefs.sf1;  // (6) OK. Not accessed by simple name

    System.out.printf("Exit static block 1:  sf1=%s, sf2=%s%n",
        ForwardRefs.sf1, ForwardRefs.sf2); // Exit static block 1:  sf1=20, sf2=0
  }

  // Field declarations:
  static int sf1 = sf2 = 30;  // (7) Static field. Assignment to sf2 allowed

  static int sf2;             // (8) Static field
  int if1 = 5;                // (9) Non-static field

  static {                    // (10) Static initializer block
    System.out.printf("Enter static block 2: sf1=%s, sf2=%s%n",
        ForwardRefs.sf1, ForwardRefs.sf2); // Enter static block 2: sf1=30, sf2=30

    int d = 2 * sf1;          // (11) OK. Read operation after declaration
    var e = sf1 = 50;         // (12) OK. Assignment to sf1 allowed

    System.out.printf("Exit static block 2:  sf1=%s, sf2=%s%n",
        ForwardRefs.sf1, ForwardRefs.sf2); // Exit static block 2:  sf1=50, sf2=30
  }

  public static void main(String[] args) {
  }
}
```

Output from the program:

Click here to view code image

```
Enter static block 1: sf1=0, sf2=0
Exit static block 1:  sf1=20, sf2=0
Enter static block 2: sf1=30, sf2=30
Exit static block 2:  sf1=50, sf2=30
```

Example 10.4 gives an idea of the code rearrangement that the compiler does to facilitate the initialization for the class in Example 10.3. Field declarations from Example 10.3 are arranged first at (1) and (2) in Example 10.4, followed by a single `static` initializer block at (3). The `static` initializer block in Example 10.4 contains the code for the first `static` initializer block, followed by the initializer expression from the first `static` field declaration statement, and lastly the code from the second `static` initializer block in the order these initializers are declared in Example 10.3.

During class initialization, first the declarations of the `static` fields `sf1` and `sf2` at (1) and (2), respectively, result in them being created and initialized to their default value `0`. Not surprisingly, execution of the code in the `static` initializer block at (3) gives the same result as in Example 10.3.

Example 10.4 *Static Initializers and Order of Execution*

Click here to view code image

```
package refs2;

public class ForwardRefsSimulated {

  // Declaration of static fields:
  static int sf1;              // (1) Initialized to default value: 0
  static int sf2;              // (2) Initialized to default value: 0

  static {                     // (3) Static initializer block
    // Code from static block 1:

    System.out.printf("Enter static block 1: sf1=%s, sf2=%s%n",
        sf1, sf2);             // Enter static block 1: sf1=0, sf2=0

    sf1 = 10;                  // (4) sf1 gets the value 10
    var b = sf1 = 20;          // (5) b and sf1 get the value 20
    int c = sf1;               // (6) c gets the value 20

    System.out.printf("Exit static block 1:  sf1=%s, sf2=%s%n",
        sf1, sf2);             // Exit static block 1:  sf1=20, sf2=0

    // Initializer expressions for field declaration:
    sf1 = sf2 = 30;            // (7) sf1 and sf2 get the value 30

    // Code from static block 2:
    System.out.printf("Enter static block 2: sf1=%s, sf2=%s%n",
        sf1, sf2);             // Enter static block 2: sf1=30, sf2=30

    int d = 2 * sf1;           // (8) d gets the value 60
    var e = sf1 = 50;         // (9) e and sf1 get the value 50

    System.out.printf("Exit static block 2:  sf1=%s, sf2=%s%n",
        sf1, sf2);            // Exit static block 2:  sf1=50, sf2=30
  }

  public static void main(String[] args) {
  }
}
```

Output from the program:

Click here to view code image

```
Enter static block 1: sf1=0, sf2=0
Exit static block 1:  sf1=20, sf2=0
Enter static block 2: sf1=30, sf2=30
Exit static block 2:  sf1=50, sf2=30
```

Exception Handling in Static Initializer Blocks

Exception handling in static initializer blocks is no different from that in static initializer expressions: Uncaught checked exceptions cannot be thrown. Code in initializers cannot throw *checked* exceptions. A `static` initializer block cannot be called directly. Therefore, any checked exceptions must be caught and handled in the body of the `static` initializer block; otherwise, the compiler will issue an error. Example 10.5 shows a `static` initializer block at (3) that catches and handles a checked exception in the `try-catch` block at (4).

Example 10.5 *Static Initializer Blocks and Exceptions*

Click here to view code image

```
// File: ExceptionInStaticInitBlocks.java
package version1;

class TooManyCellsException extends Exception {          // (1) Checked Exception
  TooManyCellsException(String number) { super(number); }
}

//_____________________________________________________________________________
class Prison {
  // Static Members
  private static int   noOfCells = 365;
  private static int[] cells;                    // (2) No initializer expression

  static {                                       // (3) Static block
    try {                                        // (4) Handles checked exception
      if (noOfCells > 300)
        throw new TooManyCellsException(String.valueOf(noOfCells));
    } catch (TooManyCellsException e) {
      System.out.println("Exception handled: " + e);
      noOfCells = 300;
      System.out.println("No. of cells adjusted to " + noOfCells);
    }
    cells = new int[noOfCells];
  }
}
//_____________________________________________________________________________
public class ExceptionInStaticInitBlocks {
  public static void main(String[] args) {
    new Prison();
  }
}
```

Output from the program:

Click here to view code image

```
Exception handled: version1.TooManyCellsException: 365
No. of cells adjusted to 300
```

Static initializer blocks do not exactly aid code readability, and should be used sparingly, if at all. The code in the `static` initializer block at (3) in Example 10.5 can easily be refactored to instantiate the `static` array field `cells` at (2) using the `private static` method at (3) that handles the checked exception `TooManyCellsException`:

Click here to view code image

```
class Prison {
  // Static Members
  private static int   noOfCells = 365;
  private static int[] cells = initPrison();  // (2) Initializer expression

  //
  private static int[] initPrison() {          // (3) Private static method
    try {                                        // (4) Handles checked exception
      if (noOfCells > 300)
        throw new TooManyCellsException(String.valueOf(noOfCells));
    } catch (TooManyCellsException e) {
      System.out.println("Exception handled: " + e);
      noOfCells = 300;
      System.out.println("No. of cells adjusted to " + noOfCells);
    }
    return new int[noOfCells];
  }
}
```

10.8 Instance Initializer Blocks

Just as static initializer blocks can be used to initialize `static` fields in a named class, Java provides the ability to initialize fields during object creation using instance initializer blocks. In this respect, such blocks serve the same purpose as constructors during object creation. The syntax of an instance initializer block is the same as that of a local block, as shown at (2) in the following code. The code in the local block is executed every time an instance of the class is created.

Click here to view code image

```
class InstanceInitializers {
  long[] squares = new long[10];    // (1)
  // ...
  {                                 // (2) Instance Initializer
    for (int i = 0; i < squares.length; i++)
      squares[i] = i*i;
  }
  // ...
}
```

The array `squares` of specified length is first created at (1); its creation is followed by the execution of the instance initializer block at (2) every time an instance of the class `InstanceInitializers` is created. Note that the instance initializer block is not contained in any method. A class can have more than one instance initializer block, and these (and any instance initializer expressions in instance field declarations) are executed in the order they are specified in the class.

Declaration Order of Instance Initializers

Analogous to the other initializers discussed earlier, an instance initializer block cannot make a forward reference to a field by its simple name in a read operation as that would violate the declaration-before-reading rule. However, using the `this` keyword to access a field is not a problem.

The class below has an instance initializer block at (1) with forward references to the fields `i`, `j`, and `k` that are declared at (7), (8), and (9), respectively. These fields are accessed using the `this` reference in *read* operations at (3), (4), (5), and (6). Using the simple name of these fields at (3), (4), (5), and (6) to access their values will violate the declare-before-use rule, resulting in compile-time errors—regardless of whether the fields are declared with initializer expressions or not, or whether they are final or not. The fields `i` and `j` are accessed at (2) in *write* operations, which are permitted using the simple name. However, care must be exercised to ensure that the fields are initialized correctly. At (3), (4), and (5), the fields `i` and `j` have the value `10`. However, when the initializer expressions are evaluated in the instance field declarations, the value of `j` will be set to `100`.

Click here to view code image

```
public class InstanceInitializersII {

  { //Instance initializer with forward references.   (1)
    i = j = 10;                                     // (2) Permitted.
    int result = this.i * this.j;                   // (3) i is 10, j is 10.
    System.out.println(this.i);                     // (4) 10
    System.out.println(this.j);                     // (5) 10
    System.out.println(this.k);                     // (6) 50
  }
  // Instance field declarations.
  int i;                  // (7) Field declaration without initializer expression.
  int j = 100;            // (8) Field declaration with initializer expression.
  final int k = 50;       // (9) Final instance field with constant expression.

}
```

Example 10.6 illustrates some additional subtle points regarding instance initializer blocks. In Example 10.6, an illegal forward reference occurs in the code at (4), which attempts to read the value of the field `nsf1` before it is declared. The read operation at (11) occurs after the declaration, and therefore, is allowed. Forward reference made on the left-hand side of the assignment is always allowed, as shown at (2), (3), (5), and (7).

Declaring local variables using the reserved word `var` in instance initializer blocks is shown at (5) and (12) in Example 10.6.

Example 10.6 *Instance Initializers and Forward References*

Click here to view code image

```
public class NonStaticForwardReferences {

  {                           // (1) Instance initializer block.
    nsf1 = 10;                // (2) OK. Assignment to nsf1 allowed.
    nsf1 = sf1;               // (3) OK. Static field access in non-static context.
    //  int a = 2 * nsf1;     // (4) Not OK. Read operation before declaration.
    var b = nsf1 = 20;        // (5) OK. Assignment to nsf1 allowed.
    int c = this.nsf1;        // (6) OK. Not accessed by simple name.
  }

  int nsf1 = nsf2 = 30;       // (7) Non-static field. Assignment to nsf2 allowed.
  int nsf2;                   // (8) Non-static field.
  static int sf1 = 5;         // (9) Static field.

  {                           // (10) Instance initializer block.
    int d = 2 * nsf1;         // (11) OK. Read operation after declaration.
    var e = nsf1 = 50;        // (12) OK. Assignment to nsf1 allowed.
  }

  public static void main(String[] args) {
    NonStaticForwardReferences objRef = new NonStaticForwardReferences();
    System.out.println("nsf1: " + objRef.nsf1);
    System.out.println("nsf2: " + objRef.nsf2);
  }
}
```

Output from the program:

```
nsf1: 50
nsf2: 30
```

As in an instance initializer expression, the keywords `this` and `super` can be used to refer to the current object in an instance initializer block.

As in a static initializer block, the `return` statement is also not allowed in instance initializer blocks.

An instance initializer block can be used to factor out common initialization code that will be executed regardless of which constructor is invoked. A typical usage of an instance initializer block is in anonymous classes (§9.6, p.521), which cannot declare constructors, but can instead use instance initializer blocks to initialize fields. In **Example 10.7**, the anonymous class defined at (1) uses an instance initializer block at (2) to initialize its fields.

Example 10.7 *Instance Initializer Block in Anonymous Class*

Click here to view code image

```
// File: InstanceInitBlock.java
class Base {
  protected int a;
  protected int b;
  void print() { System.out.println("a: " + a); }
}
//________________________________________________________________________________
class AnonymousClassMaker {
  Base createAnonymous() {
    return new Base() {           // (1) Anonymous class
      {                           // (2) Instance initializer
        a = 5; b = 10;
      }

      @Override
      void print() {
        super.print();
        System.out.println("b: " + b);
      }
    };  // end anonymous class
  }
}
//________________________________________________________________________________
public class InstanceInitBlock {
  public static void main(String[] args) {
    new AnonymousClassMaker().createAnonymous().print();
  }
}
```

Output from the program:

```
a: 5
b: 10
```

Exception Handling and Instance Initializer Blocks

Exception handling in instance initializer blocks is similar to that in static initializer blocks. Example 10.8 shows an instance initializer block at (3) that catches and handles a checked exception in the `try-catch` block at (4). Another instance initializer block at (5) throws an unchecked exception at (6). The runtime system handles the runtime exception, printing the stack trace and terminating the program.

Exception handling in instance initializer blocks differs from that in static initializer blocks in the following aspect: The execution of an instance initializer block can result in an uncaught checked exception, provided the exception is declared in the `throws` clause of *every* constructor in the class. Static initializer blocks cannot allow this, since no constructors are involved in class initialization. Instance initializer blocks in anonymous classes have even greater freedom: They can throw any exception.

Example 10.8 *Exception Handling in Instance Initializer Blocks*

Click here to view code image

```
// File: ExceptionsInInstBlocks.java
class RoomOccupancyTooHighException
      extends Exception {}                       // (1) Checked exception
class BankruptcyException
      extends RuntimeException {}                // (2) Unchecked exception
//_______________________________________________________________________
class Hotel {
  // Instance Members
  private boolean bankrupt         = true;
  private int     noOfRooms        = 215;
  private int     occupancyPerRoom = 5;
  private int     maxNoOfGuests;

  {                                              // (3) Instance initializer block
    try {                                        // (4) Handles checked exception
      if (occupancyPerRoom > 4)
        throw new RoomOccupancyTooHighException();
    } catch (RoomOccupancyTooHighException exception) {
      System.out.println("ROOM OCCUPANCY TOO HIGH: " + occupancyPerRoom);
      occupancyPerRoom = 4;
    }
    maxNoOfGuests = noOfRooms * occupancyPerRoom;
  }

  {                                              // (5) Instance initializer block
    if (bankrupt)
      throw new BankruptcyException();            // (6) Throws unchecked exception
  }    // ...
}
//_______________________________________________________________________
public class ExceptionsInInstBlocks {
  public static void main(String[] args) {
    new Hotel();
  }
}
```

Output from the program:

Click here to view code image

```
ROOM OCCUPANCY TOO HIGH: 5
Exception in thread "main" BankruptcyException
        at Hotel.<init>(ExceptionsInInstBlocks.java:27)
        at ExceptionsInInstBlocks.main(ExceptionsInInstBlocks.java:33)
```

10.9 Constructing Initial Object State

Object initialization involves constructing the initial state of an object when it is created by the `new` operator. First the fields are initialized to their default values (§3.4, p.103)—whether they are subsequently given non-default initial values or not—and then the constructor is invoked. This can lead to *local* chaining of constructors. The invocation of the constructor at the end of the local chain of constructor invocations results in the following actions, before the constructor's execution resumes:

- Implicit or explicit invocation of the superclass constructor. Constructor chaining ensures that the state from the object's superclasses is constructed first (§5.3, p.209).
- Initialization of the instance fields by executing their instance initializer expressions and any instance initializer blocks, in the order they are specified in the class declaration.

Example 10.9 illustrates object initialization. The `new` operator is used at (8) to create an object of `SubclassB`. The no-argument constructor `SubclassB()` at (2) uses the `this()` construct to locally chain to the non-zero argument constructor at (3). This constructor then leads to an implicit call of the superclass constructor. As can be seen from the program output, the execution of the superclass's constructor at (1) reaches completion first. This is followed by the execution of the instance initializer block at (4) and the instance initializer expression at (6). Then the execution of the body of the non-zero argument constructor at (3) resumes. Finally, the no-argument constructor completes its execution, thereby completing the construction of the object state.

Note that the instance initializers are executed in the order they are specified in the class declaration. The forward reference to the field `value` at (5) is legal because the usage of the field `value` is on the left-hand side of the assignment (it does not violate the declaration-before-reading rule). The default value of the field `value` is overwritten by the instance initializer block at (5). The field `value` is again overwritten by the instance initializer expression at (6), and finally by the non-zero argument constructor at (3).

Example 10.9 *Object State Construction*

Click here to view code image

```
// File: ObjectConstruction.java
class SuperclassA {
  public SuperclassA() {                          // (1) Superclass constructor
    System.out.println("Constructor in SuperclassA");
  }
}
//______________________________________________________________________________
class SubclassB extends SuperclassA {

  SubclassB() {                                   // (2) No-argument constructor
    this(3);
    System.out.println("No-argument constructor in SubclassB");
  }

  SubclassB(int i) {                              // (3) Non-zero argument constructor
    System.out.println("Non-zero argument constructor in SubclassB");
    value = i;
  }

  {                                               // (4) Instance initializer block
    System.out.println("Instance initializer block in SubclassB");
    value = 2;                                    // (5)
  }

  int value = initializerExpression();            // (6) Instance field declaration

  private int initializerExpression() {           // (7)
    System.out.println("Instance initializer expression in SubclassB");
    return 1;
  }
}
//______________________________________________________________________________
public class ObjectConstruction {
  public static void main(String[] args) {
    SubclassB objRef = new SubclassB();           // (8)
    System.out.println("value: " + objRef.value);
  }
}
```

Output from the program:

Click here to view code image

```
Constructor in SuperclassA
Instance initializer block in SubclassB
Instance initializer expression in SubclassB
Non-zero argument constructor in SubclassB
No-argument constructor in SubclassB
value: 3
```

Some care should be exercised when writing constructors for non-`final` classes, since the object that is constructed might be a subclass instance. **Example 10.10** shows a situation where use of overridden methods in *superclass* initializers and constructors can give unexpected results. The example intentionally uses the `this` reference to underline that the instance methods and constructors are invoked on the current object, and that the constructor call results in the initialization of the object state, as expected.

The program output from **Example 10.10** shows that the field `superValue` at (1) in `SuperclassA` never gets initialized explicitly when an object of `SubclassB` is created at (8). The `SuperclassA` constructor at (2) does have a call to a method that has the name `doValue` at (3). A method with such a name is defined in `SuperclassA` at (4), but is also overridden in `SubclassB` at (7). The program output indicates that the method `doValue()` from `SubclassB` is called at (3) in the `SuperclassA` constructor. The implementation of the method `doValue()` at (4) never gets executed when an object of `SubclassB` is created. Method invocation always determines the implementation of the method to be executed, based on the *actual* type of the object. Keeping in mind that it is an object of `SubclassB` that is being initialized, the call to the method named `doValue` at (3) results in the method from `SubclassB` being executed. This can lead to unintended results. The overriding method `doValue()` at (7) in `SubclassB` can access the field `value` declared at (5) before its initializer expression has been executed; thus the method invoked can access the state of the object *before* this has been completely initialized. The value `0` is then printed, as the field `value` has not yet been initialized with the value `800` when the superclass constructor is executed.

Class initialization takes place before any instance of the class can be created or a static method of the class can be invoked. A superclass is initialized before its subclasses are initialized. Initializing a class involves initialization of the static fields by executing their static initializer expressions and any static initializer blocks.

Initialization of an interface involves execution of any static initializer expressions for the `public static final` fields declared in the interface. An interface cannot specify instance initializer expressions because it has no instance fields, nor can it specify any initializer blocks because it cannot be instantiated.

Example 10.10 *Initialization Anomaly under Object State Construction*

Click here to view code image

```
// File: ObjectInitialization.java
class SuperclassA {
  protected int superValue;                                 // (1)
  SuperclassA() {                                           // (2)
    System.out.println("Constructor in SuperclassA");
    this.doValue();                                         // (3)
  }
  void doValue() {                                          // (4)
    this.superValue = 911;
    System.out.println("superValue (from SuperclassA): " + this.superValue);
  }
}
//_______________________________________________________________________________
class SubclassB extends SuperclassA {
  private int value = 800;                                  // (5)
  SubclassB() {                                             // (6)
    System.out.println("Constructor in SubclassB");
    this.doValue();
    System.out.println("superValue (from SuperclassA): " + this.superValue);
  }
  @Override
  void doValue() {                                          // (7)
    System.out.println("value (from SubclassB): " + this.value);
  }
}
//_______________________________________________________________________________
public class ObjectInitialization {
  public static void main(String[] args) {

    System.out.println("Creating an object of SubclassB.");
    new SubclassB();                                        // (8)
  }
}
```

Output from the program:

Click here to view code image

```
Creating an object of SubclassB.
Constructor in SuperclassA
value (from SubclassB): 0
Constructor in SubclassB
```

```
value (from SubclassB): 800
superValue (from SuperclassA): 0
```

Review Questions

10.6 Given the following class, which of these static initializer blocks can be independently inserted at (1)?

Click here to view code image

```
public class MyClass {
  private static int count = 5;
  static final int STEP = 10;
  boolean alive;

  // (1) INSERT STATIC INITIALIZER BLOCK HERE
}
```

Select the three correct answers.

a. `static { alive = true; count = 0; }`

b. `static { STEP = count; }`

c. `static { count += STEP; }`

d. `static ;`

e. `static { }`

f. `static { count = 1; }`

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

Click here to view code image

```
public class MyClass {
  public static void main(String[] args) {
    MyClass obj = new MyClass(n);
  }


  static int i = 5;
```

```
    static int n;
    int j = 7;
    int k;

    public MyClass(int m) {
      System.out.println(i + ", " + j + ", " + k + ", " + n + ", " + m);
    }
    { j = 70; n = 20; } // Instance initializer block
    static { i = 50; }  // Static initializer block
  }
```

Select the one correct answer.

a. The code will fail to compile because of the instance initializer block.

b. The code will fail to compile because of the static initializer block.

c. The code will compile and print `50, 70, 0, 20, 0` at runtime.

d. The code will compile and print `50, 70, 0, 20, 20` at runtime.

e. The code will compile and print `5, 70, 0, 20, 0` at runtime.

f. The code will compile and print `5, 70, 0, 20, 20` at runtime.

g. The code will compile and print `5, 7, 0, 20, 0` at runtime.

h. The code will compile and print `5, 7, 0, 20, 20` at runtime.

10.8 Given the following class, which instance initializer block inserted independently at (1) will allow the class to be compiled?

Click here to view code image

```
  public class FirstClass {
    static int gap = 10;
    double length;
    final boolean active;

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

Select the one correct answer.

a. `instance { active = true; }`

b. `FirstClass { gap += 5; }`

c. `{ gap = 5; length = (active ? 100 : 200) + gap; }`

d. `{ }`

e. `{ length = 4.2; }`

f. `{ active = (gap > 5); length = 5.5 + gap;}`

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

Click here to view code image

```
public class Initialization {
  private static String msg(String msg) {
    System.out.println(msg);
    return msg;
  }

  public Initialization() { m = msg("1"); }

  { m = msg("2"); }

  String m = msg("3");

  public static void main(String[] args) {
    Object obj = new Initialization();
  }
}
```

Select the one correct answer.

a. The program will fail to compile.

b. The program will compile and print `1`, `2`, and `3` at runtime.

c. The program will compile and print `2`, `3`, and `1` at runtime.

d. The program will compile and print `3`, `1`, and `2` at runtime.

e. The program will compile and print `1`, `3`, and `2` at runtime.

10.10 What is the result of executing the following program?

Click here to view code image

```
public class Music {
  static {
    System.out.print("-C-");
  }
  {
    System.out.print("-D-");
  }
  public Music(){
    System.out.print("-E-");
  }
}
```

Click here to view code image

```
public class Song extends Music {
  static {
    System.out.print("-F-");
  }
  {
    System.out.print("-G-");
  }
  public Song(){
    System.out.print("-A-");
  }
}
```

Click here to view code image

```
public class Test {
  public static void main(String[] args) {
    Music x1 = new Song();
    Song x2 = new Song();
  }
}
```

Select the one correct answer.

a. `-C--D--E--F--G--A--G--A--G--A-`

b. `-C--D--E--F--G--A--D--E--G--A-`

c. `-C--F--D--E--G--A--D--E--G--A-`

d. `-C--F--D--G--E--A--D--G--E--A-`

e. `-C--F--E--D--A--G--E--D--A--G-`

10.11 Which labeled lines in the following code can be independently *uncommented* by removing the `//` characters at the beginning of a line, such that the code will still compile?

Click here to view code image

```
class GeomInit {
//int width = 14;             // (1)
  {
//  area = width * height;    // (2)
  }

  int width = 37;
  {
//  height = 11;              // (3)
  }

  int height, area;
//area = width * height;      // (4)
  {
//  int width = 15;           // (5)
    area = 100;

  }
}
```

Select the two correct answers.

a. Line (1)

b. Line (2)

c. Line (3)

d. Line (4)

e. Line (5)