Chapter 22
Security

OCP EXAM OBJECTIVES COVERED IN THIS CHAPTER:

- Secure Coding in Java SE Application
  - Prevent Denial of Service in Java applications
  - Secure confidential information in Java application
  - Implement Data integrity guidelines- injections and inclusion and input validation
  - Prevent external attack of the code by limiting Accessibility and Extensibility, properly handling input validation, and mutability
  - Securely constructing sensitive objects
  - Secure Serialization and Deserialization

It's hard to read the news without hearing about a data breach. As developers, it is our job to write secure code that can stand up to attack. In this chapter, you will learn the basics of writing secure code in stand-alone Java applications.

We will learn how Hacker Harry tries to do bad things and Security Sienna protects her application. By the end of the chapter, you should be able to protect an application from Harry just as well as Sienna.

The exam only covers Java SE (Standard Edition) applications. It does not cover web applications or any other advanced Java.

Designing a Secure Object

Java provides us with many tools to protect the objects that we create. In this section, we will look at access control, extensibility, validation, and creating immutable objects. All of these techniques can protect your objects from Hacker Harry.

Limiting Accessibility

Hacker Harry heard that the zoo uses combination locks for the animals' enclosures. He would very much like to get all the combinations.

Let's start with a terrible implementation.

```
package animals.security;
public class ComboLocks {
   public Map<String, String> combos;
}
```

This is terrible because the `combos` object has `public` access. This is also poor encapsulation. A key security principle is to limit access as much as possible. Think of it as "need to know" for objects. This is called the *principle of least privilege*.

In Chapter 7, "Methods and Encapsulation," you learned about the four levels of access control. It would be better to make the `combos` object `private` and write a method to provide the necessary functionality.

```
package animals.security;
public class ComboLocks {
   private Map<String, String> combos;

   public boolean isComboValid(String animal, String combo) {
      var correctCombo = combos.get(animal);
      return combo.equals(correctCombo);
```

```
    }
}
```

This is far better; we don't expose the combinations map to any classes outside the `ComboLocks` class. For example, package-private is better than `public`, and `private` is better than package-private.

Remember, one good practice to thwart Hacker Harry and his cronies is to limit accessibility by making instance variables `private` or package-private, whenever possible. If your application is using modules, you can do even better by only exporting the security packages to the specific modules that should have access. Here's an example:

```
exports animals.security to zoo.staff;
```

In this example, only the `zoo.staff` module can access the `public` classes in the `animals.security` package.

Restricting Extensibility

Suppose you are working on a class that uses `ComboLocks`.

```
public class GrasshopperCage {
   public static void openLock(ComboLocks comboLocks, String combo) {
      if (comboLocks.isComboValid("grasshopper", combo))
         System.out.println("Open!");
   }
}
```

Ideally, the first variable passed to this method is an instance of the `ComboLocks` class. However, Hacker Harry is hard at work and has created this subclass of `ComboLocks`.

```
public class EvilComboLocks extends ComboLocks {
   public boolean isComboValid(String animal, String combo) {
```

```
        var valid = super.isComboValid(animal, combo);
        if (valid) {
           // email the password to Hacker Harry
        }
        return valid;
     }
  }
```

This is great. Hacker Harry can check whether the password is valid and email himself all the valid passwords. Mayhem ensues! Luckily, there is an easy way to prevent this problem. Marking a sensitive class as `final` prevents any subclasses.

```
  public final class ComboLocks {
     private Map<String, String> combos;

     // instantiate combos object

     public boolean isComboValid(String animal, String combo) {
        var correctCombo = combos.get(animal);
        return combo.equals(correctCombo);
     }
  }
```

Hacker Harry can't create his evil class, and users of the `GrasshopperCage` have the assurance that only the expected `ComboLocks` class can make an appearance.

Creating Immutable Objects

As you might remember from Chapter 12, "Java Fundamentals," an immutable object is one that cannot change state after it is created. Immutable objects are helpful when writing secure code because you don't have to worry about the values changing. They also simplify code when dealing with concurrency.

We worked with some immutable objects in the book. The `String` class used throughout the book is immutable. In Chapter 14, "Generics and Collections," you used `List.of()`, `Set.of()`, and `Map.of()`. All three of these methods return immutable types.

Although there are a variety of techniques for writing an immutable class, you should be familiar with a common strategy for making a class immutable.

1. Mark the class as `final`.
2. Mark all the instance variables `private`.
3. Don't define any setter methods and make fields final.
4. Don't allow referenced mutable objects to be modified.
5. Use a constructor to set all properties of the object, making a copy if needed.

The first rule prevents anyone from creating a mutable subclass. You might notice this is the same technique we used to restrict extensibility. The second rule provides good encapsulation. The third rule ensures that callers and the class itself don't make changes to the instance variables.

The fourth rule is subtler. Basically, it means you shouldn't expose a getter method for a mutable object. For example, can you see why the following is not an immutable object?

```
1:  import java.util.*;
2:
3:  public final class Animal {
4:     private final ArrayList<String> favoriteFoods;
5:
6:     public Animal() {
7:        this.favoriteFoods = new ArrayList<String>();
8:        this.favoriteFoods.add("Apples");
9:     }
10:    public List<String> getFavoriteFoods() {
11:       return favoriteFoods;
```

```
12:    }
13: }
```

We carefully followed the first three rules, but unfortunately, Hacker Harry can modify our data by calling `getFavoriteFoods().clear()` or add a food to the list that our animal doesn't like. It's not an immutable object if we can change it contents! If we don't have a getter for the `favoriteFoods` object, how do callers access it? Simple, by using delegate methods to read the data, as shown in the following:

```
1:  import java.util.*;
2:
3:  public final class Animal {
4:     private final ArrayList<String> favoriteFoods;
5:
6:     public Animal() {
7:        this.favoriteFoods = new ArrayList<String>();
8:        this.favoriteFoods.add("Apples");
9:     }
10:    public int getFavoriteFoodsCount() {
11:       return favoriteFoods.size();
12:    }
13:    public String getFavoriteFoodsElement(int index) {
14:       return favoriteFoods.get(index);
15:    }
16: }
```

In this improved version, the data is still available. However, it is a true immutable object because the mutable variable cannot be modified by the caller. Another option is to create a copy of the `favoriteFoods` object and return the copy anytime it is requested, so the original remains safe.

```
10:    public ArrayList<String> getFavoriteFoods() {
11:       return new ArrayList<String>(this.favoriteFoods);
12:    }
```

Of course, changes in the copy won't be reflected in the original, but at least the original is protected from external changes. In the next section, we'll see there is another way to copy an object if the class implements a certain interface.

So, what's this about the last rule for creating immutable objects? Let's say we want to allow the user to provide the `favoriteFoods` data, so we implement the following:

```
1:  import java.util.*;
2:
3:  public final class Animal {
4:     private final ArrayList<String> favoriteFoods;
5:
6:     public Animal(ArrayList<String> favoriteFoods) {
7:        if(favoriteFoods == null)
8:           throw new RuntimeException("favoriteFoods is required");
9:        this.favoriteFoods = favoriteFoods;
10:    }
11:    public int getFavoriteFoodsCount() {
12:       return favoriteFoods.size();
13:    }
14:    public String getFavoriteFoodsElement(int index) {
15:       return favoriteFoods.get(index);
16:    }
17: }
```

To ensure that `favoriteFoods` is not `null`, we validate it in the constructor and throw an exception if it is not provided. Hacker Harry is tricky, though. He decides to send us a `favoriteFood` object but keep his own secret reference to it, which he can modify directly.

```
void modifyNotSoImmutableObject() {
      var favorites = new ArrayList<String>();
      favorites.add("Apples");
      var animal = new Animal(favorites);
      System.out.print(animal.getFavoriteFoodsCount());
```

```
        favorites.clear();
        System.out.print(animal.getFavoriteFoodsCount());
    }
```

This method prints `1`, followed by `0`. Whoops! It seems like `Animal` is not immutable anymore, since its contents can change after it is created. The solution is to use a *copy constructor* to make a copy of the list object containing the same elements.

```
6:     public Animal(List<String> favoriteFoods) {
7:        if(favoriteFoods == null)
8:           throw new RuntimeException("favoriteFoods is required");
9:        this.favoriteFoods = new ArrayList<String>(favoriteFoods);
10:    }
```

The copy operation is called a *defensive copy* because the copy is being made in case other code does something unexpected. It's the same idea as defensive driving. Security Sienna has to be safe because she can't control what others do. With this approach, Hacker Harry is defeated. He can modify the original `favoriteFoods` all he wants, but it doesn't change the `Animal` object's contents.

Cloning Objects

Java has a `Cloneable` interface that you can implement if you want classes to be able to call the `clone()` method on your object. This helps with making defensive copies.

The `ArrayList` class does just that, which means there's another way to write the statement on line 9.

```
9: this.favoriteFoods = (ArrayList) favoriteFoods.clone();
```

The `clone()` method makes a copy of an object. Let's give it a try by changing line 3 of the previous example to the following:

```
public final class Animal implements Cloneable {
```

Now we can write a method within the Animal class:"

```
public static void main(String… args) throws Exception {
   ArrayList<String> food = new ArrayList<>();
   food.add("grass");
   Animal sheep = new Animal(food);
   Animal clone = (Animal) sheep.clone();
   System.out.println(sheep == clone);
   System.out.println(sheep.favoriteFoods == clone.favoriteFoods);
}
```

This code outputs the following:

```
false
true
```

By default, the `clone()` method makes a *shallow copy* of the data, which means only the top-level object references and primitives are copied. No new objects from within the cloned object are created. For example, if the object contains a reference to an `ArrayList`, a shallow copy contains a reference to that same `ArrayList`. Changes to the `ArrayList` in one object will be visible in the other since it is the same object.

By contrast, you can write an implementation that does a *deep copy* and clones the objects inside. A deep copy does make a new `ArrayList` object. Changes to the cloned object do not affect the original.

```
public Animal clone() {
   ArrayList<String> listClone = (ArrayList) favoriteFoods.clone();
   return new Animal(listClone);
}
```

Now the `main()` method prints `false` twice because the `ArrayList` is also cloned.

You might have noticed that the `clone()` method is declared in the `Object` class. The default implementation throws an exception that tells you the `Object` didn't implement `Cloneable`. If the class implements `Cloneable`, you can call `clone()`. Classes that implement `Cloneable` can also provide a custom implementation of `clone()`, which is useful when the class wants to make a deep copy. Figure 22.1 reviews how Java decides what to do when `clone()` is called.

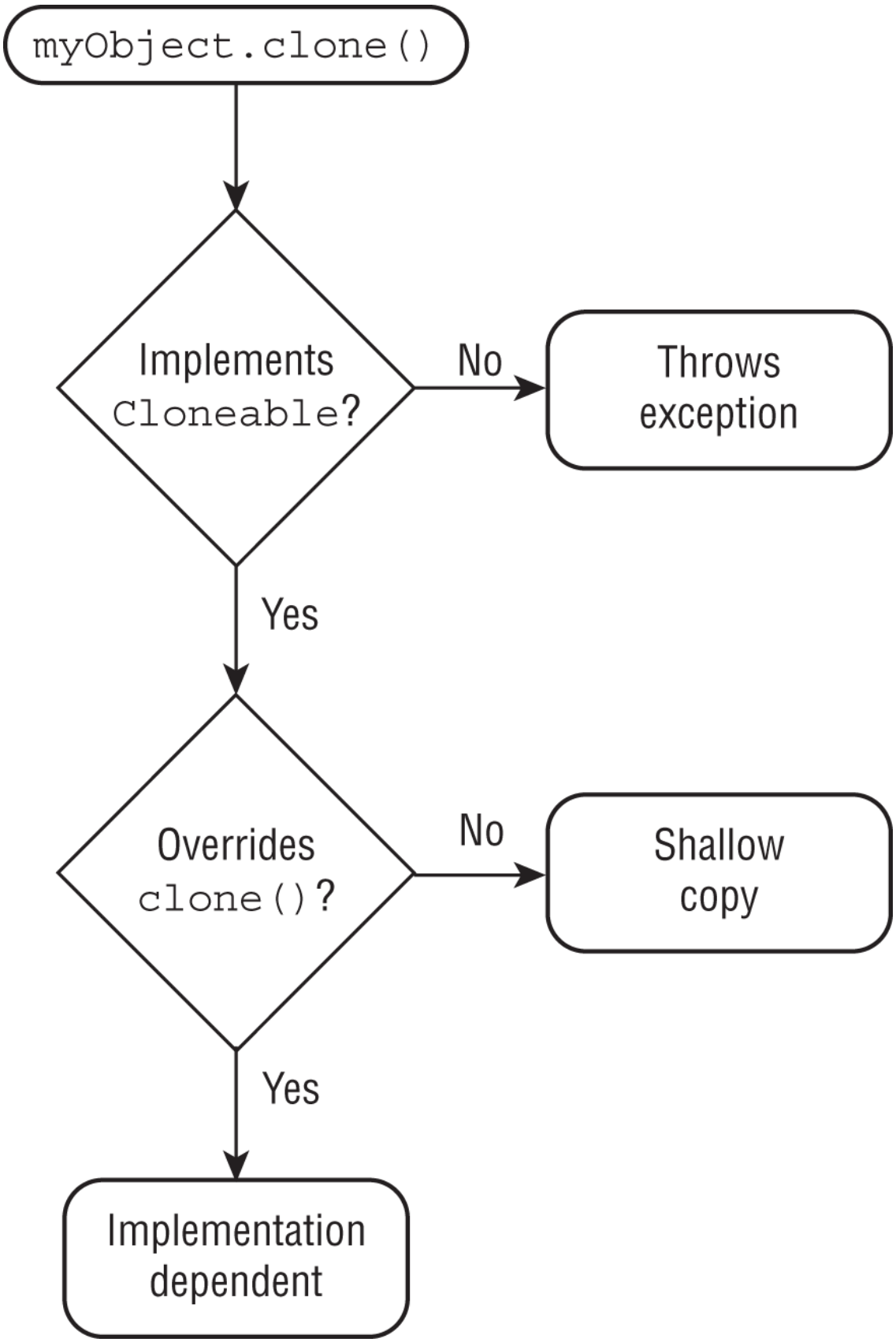


FIGURE 22.1 `Cloneable` logic

In the last block, implementation-dependent means you should probably check the Javadoc of the overridden `clone()` method before using it. It

may provide a shallow copy, a deep copy, or something else entirely. For example, it may be a shallow copy limited to three levels.

Introducing Injection and Input Validation

Injection is an attack where dangerous input runs in a program as part of a command. For example, user input is often used in database queries or I/O. In this section, we will look at how to protect your code against injection using a `PreparedStatement` and input validation.

An *exploit* is an attack that takes advantage of weak security. Hacker Harry is ready to try to exploit any code he can find. He especially likes untrusted data.

There are many sources of untrusted data. For the exam, you need to be aware of user input, reading from files, and retrieving data from a database. In the real world, any data that did not originate from your program should be considered suspect.

Preventing Injection with a *PreparedStatement*

Our zoo application has a table named `hours` that keeps track of when the zoo is open to the public. Figure 22.2 shows the columns in this table.

hours

day *varchar(20)*	opens *integer*	closes *integer*
sunday	9	6
monday	10	4
tuesday	10	4
wednesday	10	5
thursday	10	4
friday	10	6
saturday	9	6

FIGURE 22.2 Hours table

In the following sections, we will look at two examples that are insecure followed by the proper fix.

Using *Statement*

We wrote a method that uses a `Statement`. In Chapter 21, "JDBC," we didn't use `Statement` because it is often unsafe.

```
public int getOpening(Connection conn, String day)
      throws SQLException {
   String sql = "SELECT opens FROM hours WHERE day = '" + day +"'";

   try (var stmt = conn.createStatement();
      var rs = stmt.executeQuery(sql)) {
      if (rs.next())
         return rs.getInt("opens");
```

```
    }
    return -1;
  }
```

Then, we call the code with one of the days in the table.

```
int opening = attack.getOpening(conn, "monday");  // 10
```

This code does what we want. It queries the database and returns the opening time on the requested day. So far, so good. Then Hacker Harry comes along to call the method. He writes this:

```
int evil = attack.getOpening(conn,
   "monday' OR day IS NOT NULL OR day = 'sunday");  // 9
```

This does not return the expected value. It returned 9 when we ran it. Let's take a look at what Hacker Harry tricked our database into doing.

Hacker Harry's parameter results in the following SQL, which we've formatted for readability:

```
SELECT opens FROM hours
   WHERE day = 'monday'
      OR day IS NOT NULL
      OR day = 'sunday'
```

It says to return any rows where `day` is `sunday`, `monday`, or any value that isn't `null`. Since none of the values in Figure 22.2 is `null`, this means all the rows are returned. Luckily, the database is kind enough to return the rows in the order they were inserted; our code reads the first row.

Using *PreparedStatement*

Obviously, we have a problem with using `Statement`, and we call Security Sienna. She reminds us that `Statement` is insecure because it is vulnerable to SQL injection. As Hacker Harry just showed us an attack, we have to agree.

We switch our code to use `PreparedStatement`.

```
public int getOpening(Connection conn, String day)
      throws SQLException {
   String sql = "SELECT opens FROM hours WHERE day = '" + day +"'";
   try (var ps = conn.prepareStatement(sql);
      var rs = ps.executeQuery()) {
      if (rs.next())
         return rs.getInt("opens");
   }
   return -1;
}
```

Hacker Harry runs his code, and the behavior hasn't changed. We haven't fixed the problem! A `PreparedStatement` isn't magic. It gives you the capability to be safe, but only if you use it properly.

Security Sienna shows us that we need to rewrite the SQL statement using bind variables like we did in Chapter 21.

```
public int getOpening(Connection conn, String day)
      throws SQLException {
   String sql = "SELECT opens FROM hours WHERE day = ?";
   try (var ps = conn.prepareStatement(sql)) {
      ps.setString(1, day);
      try (var rs = ps.executeQuery()) {
         if (rs.next())
            return rs.getInt("opens");
      }
   }
   return -1;
}
```

This time, Hacker Harry's code does behave differently.

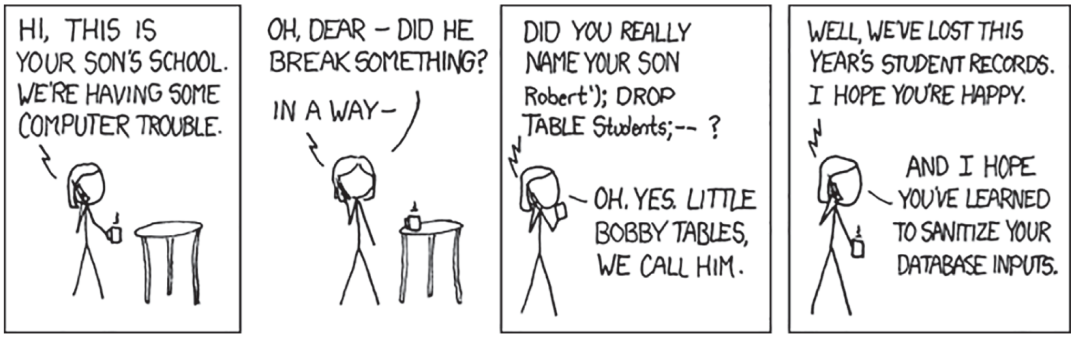
```
int evil = attack.getOpening(conn,
   "monday' or day is not null or day = 'sunday");  // -1
```

The entire string is matched against the `day` column. Since there is no match, no rows are returned. This is far better!

If you remember only two things about SQL and security, remember to use a `PreparedStatement` and bind variables.

LITTLE BOBBY TABLES

SQL injection is often caused by a lack of properly sanitized user input. The author of the popular xkcd.com web comic once asked the question, what would happen if someone's name contained a SQL statement?



"Exploits of a Mom" reproduced with permission from xkcd.com/327 /

Oops! Guess the school should have used a `PreparedStatement` and bound each student's name to a variable. If they had, the entire `String` would have been properly escaped and stored in the database.

Some databases, like Derby, prevent such an attack. However, it is important to use a `PreparedStatement` properly to avoid even the possibility of such an attack.

Invalidating Invalid Input with Validation

SQL injection isn't the only type of injection. *Command injection* is another type that uses operating system commands to do something unexpected.

In our example, we will use the `Console` class from Chapter 19, "I/O," and the `Files` class from Chapter 20, "NIO.2." Figure 22.3 shows the directory structure we will be using in the example.

The following code attempts to read the name of a subdirectory of `diets` and print out the names of all the `.txt` files in that directory:

```
Console console = System.console();
String dirName = console.readLine();
Path path = Paths.get("c:/data/diets/" + dirName);
try (Stream<Path> stream = Files.walk(path)) {
   stream.filter(p -> p.toString().endsWith(".txt"))
      .forEach(System.out::println);
}
```

We tested it by typing in `mammals` and got the expected output.

```
c:/data/diets/mammals/Platypus.txt
```

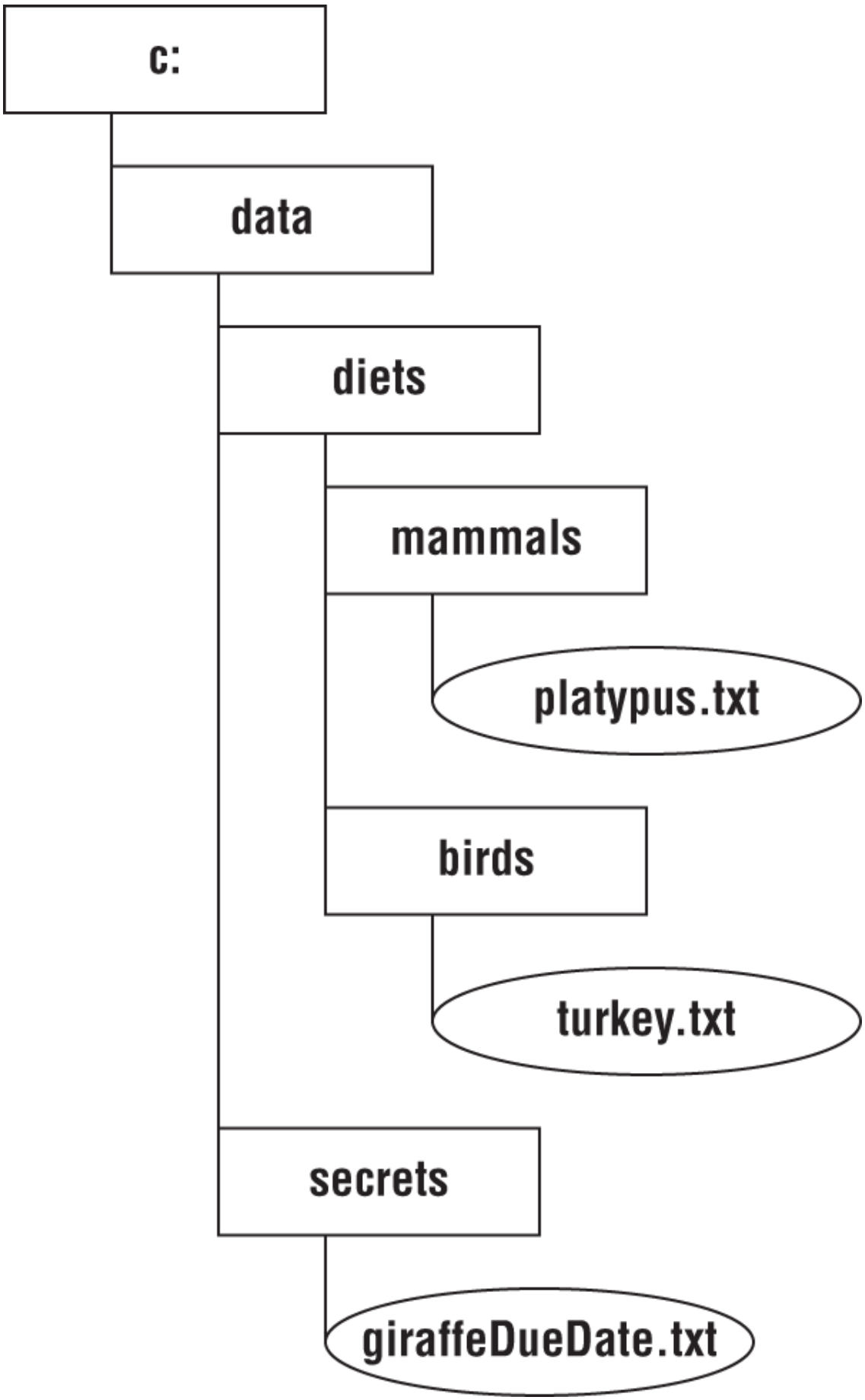


FIGURE 22.3 Directory structure

Then Hacker Harry came along and typed `..` as the directory name.

```
c:/data/diets/../secrets/giraffeDueDate.txt
c:/data/diets/../diets/mammals/Platypus.txt
c:/data/diets/../diets/birds/turkey.txt
```

Oh, no! Hacker Harry knows we are expecting a baby giraffe just from the filenames. We were not intending for him to see the `secrets` directory.

We decide to chat with Security Sienna about this problem. She suggests we validate the input. We will use a *whitelist* that allows us to specify which values are allowed.

```
Console console = System.console();
String dirName = console.readLine();
if (dirName.equals("mammal") || dirName.equals("birds")) {
   Path path = Paths.get("c:/data/diets/" + dirName);
   try (Stream<Path> stream = Files.walk(path)) {
      stream.filter(p -> p.toString().endsWith(".txt"))
         .forEach(System.out::println);
   }
}
```

This time when Hacker Harry strikes, he doesn't see any output at all. His input did not match the whitelist. When validation fails, you can throw an exception, log a message, or take any other action of your choosing.

WHITELIST VS. BLACKLIST

A *blacklist* is a list of things that aren't allowed. In the previous example, we could have put the dot (`.`) character on a blacklist. The problem with a blacklist is that you have to be cleverer than the bad guys. There are a lot of ways to cause harm. For example, you can encode characters.

By contrast, the whitelist is specifying what is allowed. You can supply a list of valid characters. Whitelisting is preferable to blacklisting for security because a whitelist doesn't need to foresee every possible problem.

That said, the whitelist solution could require more frequent updates. In the previous example, we would have to update the code any time we added a new animal type. Security decisions are often about trading convenience for lower risk.

Working with Confidential Information

When working on a project, you will often encounter confidential or sensitive data. Sometimes there are even laws that mandate proper handling of data like the Health Insurance Portability and Accountability Act (HIPAA) in the United States. Table 22.1 lists some examples of confidential information.

TABLE 22.1 Types of confidential data

Category	Examples
Login information	• Usernames • Passwords • Hashes of passwords
Banking	• Credit card numbers • Account balances • Credit score
PII (Personal identifiable information)	• Social Security number (or other government ID) • Mother's maiden name • Security questions/answers

In the following sections, we will look at how to secure confidential data in written form and in log files. We will also show you how to limit access.

Guarding Sensitive Data from Output

Security Sienna makes sure confidential information doesn't leak. The first step she takes is to avoid putting confidential information in a `toString()` method. That's just inviting the information to wind up logged somewhere you didn't intend.

She is careful what methods she calls in these sensitive contexts to ensure confidential information doesn't escape. Such sensitive contexts include

the following:

- Writing to a log file
- Printing an exception or stack trace
- `System.out` and `System.err` messages
- Writing to data files

Hacker Harry is on the lookout for confidential information in all of these places. Sometimes you have to process sensitive information. It is important to make sure it is being shared only per the requirements.

Protecting Data in Memory

Security Sienna needs to be careful about what is in memory. If her application crashes, it may generate a dump file. That contains values of everything in memory.

When calling the `readPassword()` on `Console`, it returns a `char[]` instead of a `String`. This is safer for two reasons.

- It is not stored as a `String`, so Java won't place it in the `String` pool, where it could exist in memory long after the code that used it is run.
- You can `null` out the value of the array element rather than waiting for the garbage collector to do it.

For example, this code overlays the password characters with the letter `x`:

```
Console console = System.console();
char[] password = console.readPassword();
Arrays.fill(password, 'x');
```

When the sensitive data cannot be overwritten, it is good practice to set confidential data to `null` when you're done using it. If the data can be

garbage collected, you don't have to worry about it being exposed later. Here's an example:

```
LocalDate dateOfBirth = getDateOfBirth();
// use date of birth
dateOfBirth = null;
```

The idea is to have confidential data in memory for as short a time as possible. This gives Hacker Harry less time to make his move.

Limiting File Access

We saw earlier how to prevent command injection by validating requests. Another way is to use a security policy to control what the program can access.

DEFENSE IN DEPTH

It is good to apply multiple techniques to protect your application. This approach is called *defense in depth*. If Hacker Harry gets through one of your defenses, he still doesn't get the valuable information inside. Instead, he is met with another defense.

Validation and using a security policy are good techniques to use together to apply defense in depth.

For the exam, you don't need to know how to write or run a policy. You do need to be able to read one to understand security implications. Luckily, they are fairly self-explanatory. Here's an example of a policy:

```
grant {
   permission java.io.FilePermission
      "C:\\water\\fish.txt",
```

```
        "read";
};
```

This policy gives the programmer permission to read, but not update, the `fish.txt` file. If the program is allowed to read and write the file, we specify the following:

```
grant {
   permission java.io.FilePermission
      "C:\\water\\fish.txt",
      "read, write";
};
```

When looking at a policy, pay attention to whether the policy grants access to more than is needed to run the program. If our application needs to read a file, it should only have `read` permissions. This is the principle of least privilege we showed you earlier.

Serializing and Deserializing Objects

Imagine we are storing data in an `Employee` record. We want to write this data to a file and read this data back into memory, but we want to do so without writing any potentially sensitive data to disk. From Chapter 19, you should already know how to do this with serialization.

Recall from Chapter 19 that Java skips calling the constructor when deserializing an object. This means it is important not to rely on the constructor for custom validation logic.

Let's define our `Employee` class used throughout this section. Remember, it's important to mark it `Serializable`.

```
import java.io.*;

public class Employee implements Serializable {
```

```
    private String name;
    private int age;

    // Constructors/getters/setters
}
```

In the following sections, we will look at how to make serialization safer by specifying which fields get serialized and the process for controlling serialization itself.

Specifying Which Fields to Serialize

Our zoo has decided that employee age information is sensitive and shouldn't be written to disk. From Chapter 19, you should already know how to do this. Security Sienna reminds us that marking a field as `transient` prevents it from being serialized.

```
    private transient int age;
```

Alternatively, you can specify fields to be serialized in an array.

```
private static final ObjectStreamField[] serialPersistentFields =
    { new ObjectStreamField("name", String.class) };
```

You can think of `serialPersistentFields` as the opposite of `transient`. The former is a whitelist of fields that should be serialized, while the latter is a blacklist of fields that should not.

If you go with the array approach, make sure you remember to use the `private`, `static`, and `final` modifiers. Otherwise, the field will be ignored.

Customizing the Serialization Process

Security may demand custom serialization. In our case, we got a new requirement to add the Social Security number to our object. (For our readers outside the United States, a Social Security number is used for reporting your earnings to the government, among other things.) Unlike age, we do need to serialize this information. However, we don't want to store the Social Security number in plain text, so we need to write some custom code.

Take a look at the following implementation that uses `writeObject()` and `readObject()` for serialization, which you learned about in Chapter 19. For brevity, we'll use `ssn` to stand for Social Security number.

```
import java.io.*;

public class Employee implements Serializable {
   private String name;
   private String ssn;
   private int age;

   // Constructors/getters/setters

   private static final ObjectStreamField[] serialPersistentFields =
      { new ObjectStreamField("name", String.class),
      new ObjectStreamField("ssn", String.class) };

   private static String encrypt(String input) {
      // Implementation omitted
   }
   private static String decrypt(String input) {
      // Implementation omitted
   }

   private void writeObject(ObjectOutputStream s) throws Exception {
      ObjectOutputStream.PutField fields = s.putFields();
      fields.put("name", name);
      fields.put("ssn", encrypt(ssn));
```

```
        s.writeFields();
    }
    private void readObject(ObjectInputStream s) throws Exception {
        ObjectInputStream.GetField fields = s.readFields();
        this.name = (String)fields.get("name", null);
        this.ssn = decrypt((String)fields.get("ssn", null));
    }
}
```

This version skips the `age` variable as before, although this time without using the `transient` modifier. It also uses custom read and write methods to securely encrypt/decrypt the Social Security number. Notice the `PutField` and `GetField` classes are used in order to write and read the fields easily.

Suppose we were to update our `writeObject()` method with the `age` variable.

```
        fields.put("age", age);
```

When using serialization, the code would result in an exception.

```
java.lang.IllegalArgumentException: no such field age with type int
```

This shows the `serialPersistentFields` variable is really being used. Java is preventing us from referencing fields that were not declared to be serializable.

Real World Scenario

WORKING WITH PASSWORDS

In this example, we encrypted and then decrypted the Social Security number to show how to perform custom serialization for security reasons. Some fields are too sensitive even for that. In particular, you should never be able to decrypt a password.

When a password is set for a user, it should be converted to a `String` value using a salt (initial random value) and one-way hashing algorithm. Then, when a user logs in, convert the value they type in using the same algorithm and compare it with the stored value. This allows you to authenticate a user without having to expose their password.

Databases of stored passwords can (and very often do) get stolen. Having them properly encrypted means the attacker can't do much with them, like decrypt them and use them to log in to the system. They also can't use them to log in to other systems in which the user used the same password more than once.

Pre/Post-Serialization Processing

Suppose our zoo employee application is having a problem with duplicate records being created for each employee. They decide that they want to maintain a list of all employees in memory and only create users as needed. Furthermore, each employee's name is guaranteed to be unique. Unlikely in practice we know, but this is a special zoo!

From what you learned about concurrent collections in Chapter 18, "Concurrency," and factory methods, we can accomplish this with a `private` constructor and factory method.

```
import java.io.*;
import java.util.Map;
```

```
import java.util.concurrent.ConcurrentHashMap;

public class Employee implements Serializable {
   …
   private Employee() {}
   private static Map<String,Employee> pool =
      new ConcurrentHashMap<>();

   public synchronized static Employee getEmployee(String name) {
      if(pool.get(name)==null) {
         var e = new Employee();
         e.name = name;          pool.put(name, e);
      }
      return pool.get(name);
   }
}
```

This method creates a new `Employee` if one does not exist. Otherwise, it returns the one stored in the memory pool.

Applying *readResolve()*

Now we want to start reading/writing the employee data to disk, but we have a problem. When someone reads the data from the disk, it deserializes it into a new object, not the one in memory pool. This could result in two users holding different versions of the `Employee` in memory!

Enter the `readResolve()` method. When this method is present, it is run *after* the `readObject()` method and is capable of replacing the reference of the object returned by deserialization.

```
import java.io.*;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

public class Employee implements Serializable {
   …
   public synchronized Object readResolve()
```

```
            throws ObjectStreamException {
        var existingEmployee = pool.get(name);
        if(pool.get(name) == null) {
            // New employee not in memory
            pool.put(name, this);
            return this;
        } else {
            // Existing user already in memory
            existingEmployee.name = this.name;
            existingEmployee.ssn = this.ssn;
            return existingEmployee;
        }
    }
}
```

If the object is not in memory, it is added to the pool and returned. Otherwise, the version in memory is updated, and its reference is returned.

Notice that we added the `synchronized` modifier to this method. Java allows any method modifiers (except `static`) for the `readResolve()` method including any access modifier. This rule applies to `writeReplace()`, which is up next.

Applying *writeReplace()*

Now, what if we want to write an `Employee` record to disk but we don't completely trust the instance we are holding? For example, we want to always write the version of the object in the pool rather than the `this` instance. By construction, there should be only one version of this object in memory, but for this example let's pretend we're not 100 percent confident of that.

The `writeReplace()` method is run *before* `writeObject()` and allows us to replace the object that gets serialized.

```
import java.io.*;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

public class Employee implements Serializable {
   …
   public Object writeReplace() throws ObjectStreamException {
      var e = pool.get(name);
      return e != null ? e : this;
   }
}
```

This implementation checks whether the object is found in the pool. If it is found in the pool, that version is sent for serialization; otherwise, the current instance is used. We could also update this example to add it to the pool if it is somehow missing.

If these last few examples seemed a bit contrived, it's because they are. While the exam is likely to test you on these methods, implementing these advanced serialization methods in detail is way beyond the scope of the exam. Besides, `transient` will probably meet your needs for customizing what gets serialized.

Reviewing Serialization Methods

You've encountered a lot of methods in this chapter. Table 22.2 summarizes the important features of each that you should know for the exam.

TABLE 22.2 Methods for serialization and deserialization

Return type	Method	Parameters	Description
`Object`	`writeReplace()`	None	Allows replacement of object *before* serialization
`void`	`writeObject()`	`ObjectInputStream`	Serializes optionally using `PutField`
`void`	`readObject()`	`ObjectOutputStream`	Deserializes optionally using `GetField`
`Object`	`readResolve()`	None	Allows replacement of object *after* deserialization

We also provide a visualization of the process of writing and reading a record in Figure 22.4.

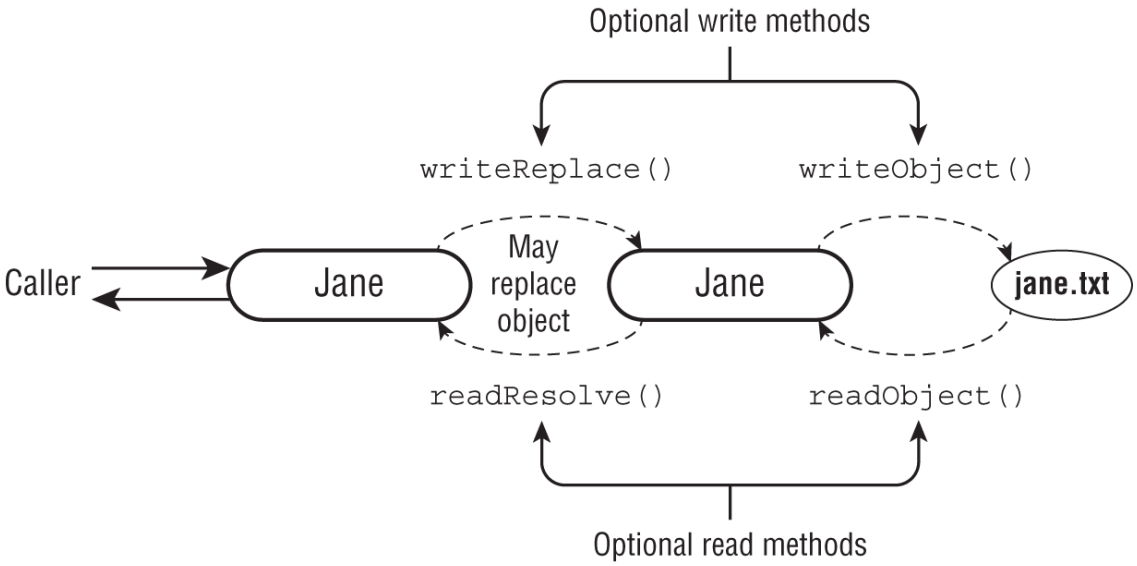


FIGURE 22.4 Writing and reading an employee

In Figure 22.4, we show how an employee record for Jane is serialized, written to disk, then read from disk, and returned to the caller. We also show that `writeReplace()` happens before `writeObject()`, while `readResolve()` happens after `readObject()`. Remember that all four of these methods are optional and must be declared in the `Serializable` object to be used.

Constructing Sensitive Objects

When constructing sensitive objects, you need to ensure that subclasses can't change the behavior. Suppose we have a `FoodOrder` class.

```
public class FoodOrder {
   private String item;
   private int count;

   public FoodOrder(String item, int count) {
      setItem(item);
      setCount(count);
   }
   public String getItem() { return item; }
   public void setItem(String item) { this.item = item; }
   public int getCount() { return count; }
```

```
    public void setCount(int count) { this.count = count; }
}
```

This seems simple enough. It is a Java object with two instance variables and corresponding getters/setters. We can even write a method that counts how many items are in our order.

```
public static int total(List<FoodOrder> orders) {
    return orders.stream()
        .mapToInt(FoodOrder::getCount)
        .sum();
}
```

This method signature pleases Hacker Harry because he can pass in his malicious subclass of `FoodOrder`. He overrides the `getCount()` and `setCount()` methods so that `count` is always zero.

```
public class HarryFoodOrder extends FoodOrder {
    public HarryFoodOrder(String item, int count) {
        super(item, count);
    }
    public int getCount() { return 0; }
    public void setCount(int count) { super.setCount(0); }
}
```

Well, that's not good. Now we can't order any food. Luckily, Security Sienna has three techniques to foil Hacker Harry. Let's take a look at each one. If you need to review the `final` modifier, we covered this in detail in Chapter 12.

Making Methods *final*

Security Sienna points out that we are letting Hacker Harry override sensitive methods. If we make the methods `final`, the subclass can't change the behavior on us.

```
public class FoodOrder {
   private String item;
   private int count;

   public FoodOrder(String item, int count) {
      setItem(item);
      setCount(count);
   }
   public final String getItem() { return item; }
   public final void setItem(String item) { this.item = item; }
   public final int getCount() { return count; }
   public final void setCount(int count) { this.count = count; }
}
```

Now the subclass can't provide different behavior for the get and set methods. In general, you should avoid allowing your constructors to call any methods that a subclass can provide its own implementation for.

Making Classes *final*

Remembering to make methods `final` is extra work. Security Sienna points out that we don't need to allow subclasses at all since everything we need is in `FoodOrder`.

```
public final class FoodOrder {
   private String item;
   private int count;

   public FoodOrder(String item, int count) {
      setItem(item);
      setCount(count);
   }
   public String getItem() { return item; }
   public void setItem(String item) { this.item = item; }
   public int getCount() { return count; }
   public void setCount(int count) { this.count = count; }
}
```

Now Hacker Harry can't create his malicious subclass to begin with!

Making the Constructor *private*

Security Sienna notes that another way of preventing or controlling sub-classes is to make the constructor `private`. This technique requires `static` factory methods to obtain the object.

```
public class FoodOrder {
   private String item;
   private int count;

   private FoodOrder(String item, int count) {
      setItem(item);
      setCount(count);
   }
   public FoodOrder getOrder(String item, int count) {
      return new FoodOrder(item, count);
   }
   public String getItem() { return item; }
   public void setItem(String item) { this.item = item; }
   public int getCount() { return count; }
   public void setCount(int count) { this.count = count; }
}
```

The factory method technique gives you more control over the process of object creation.

Since this chapter is about Java SE applications, the person running your program will have access to the code. More specifically, they will have the bytecode (`.class`) files, typically bundled in a JAR file. With the bytecode, Hacker Harry can decompile your code and get source code. It's not as well written as the code you wrote, but it has equivalent information.

Some people compile their projects with obfuscation tools to try to hide implementation details. *Obfuscation* is the automated process of rewriting source code that purposely makes it more difficult to read. For example, if you try to view JavaScript on a website, entire methods or classes may be on a single line with variable names like `aaa` , `bbb` , `ccc` , and so on. It's harder to know what a method does if it's named `gpiomjrqw()` .

While using an obfuscator makes the decompiled bytecode harder to read and therefore harder to reverse engineer, it doesn't actually provide any security. Remember that security by obscurity will slow down Hacker Harry, but it won't stop him!

Preventing Denial of Service Attacks

A *denial of service* (DoS) attack is when a hacker makes one or more requests with the intent of disrupting legitimate requests. Most denial of service attacks require multiple requests to bring down their targets. Some attacks send a very large request that can even bring down the application in one shot. In this book, we will focus on denial of service attacks.

Unless otherwise specified, a denial of service attack comes from one machine. It may make many requests, but they have the same origin. By con-

trast, a *distributed denial of service* (DDoS) attack is a denial of service attack that comes from many sources at once. For example, many machines may attack the target. In this section, we will look at some common sources of denial of service issues.

Leaking Resources

One way that Hacker Harry can mount a denial of service attack is to take advantage of poorly written code. This simple method counts the number of lines in a file using NIO.2 methods we saw in Chapter 20:

```
public long countLines(Path path) throws IOException  {
   return Files.lines(path).count();
}
```

Hacker Harry likes this method. He can call it in a loop. Since the method opens a file system resource and never closes it, there is a *resource leak*. After Hacker Harry calls the method enough times, the program crashes because there are no more file handles available.

Luckily, the fix for a resource leak is simple, and it's one you've already seen in Chapter 20. Security Sienna fixes the code by using the try-with-resources statement we saw in Chapter 16, “Exceptions, Assertions, and Localization.” Here's an example:

```
public long countLines(Path path) throws IOException  {
   try (var stream = Files.lines(path)) {
      return stream.count();
   }
}
```

Reading Very Large Resources

Another source of a denial of service attacks is very large resources. Suppose we have a simple method that reads a file into memory, does

some transformations on it, and writes it to a new file.

```
public void transform(Path in, Path out) throws IOException  {
   var list = Files.readAllLines(in);
   list.removeIf(s -> s.trim().isBlank());
   Files.write(out, list);
}
```

On a small file, this works just fine. However, on an extremely large file, your program could run out of memory and crash. Hacker Harry strikes again! To prevent this problem, you can check the size of the file before reading it.

Including Potentially Large Resources

An *inclusion* attack is when multiple files or components are embedded within a single file. Any file that you didn't create is suspect. Some types can appear smaller than they really are. For example, some types of images can have a “zip bomb” where the file is heavily compressed on disk. When you try to read it in, the file uses much more space than you thought.

Extensible Markup Language (XML) files can have the same problem. One attack is called the “billion laughs attack” where the file gets expanded exponentially.

The reason these files can become unexpectedly large is that they can include other entities. This means something that is 1 KB can become exponentially larger if it is included enough times.

While handling large files is beyond the scope of the exam, you should understand how and when these issues can come up.

Inclusion attacks are often known for when they include potentially hosted content. For example, imagine you have a web page that includes a script on another website. You don't control the script, but Hacker Harry does. Including scripts from other websites is dangerous regardless of how big they are.

Overflowing Numbers

When checking file size, be careful with an `int` type and loops. Since an `int` has a maximum size, exceeding that size results in integer overflow. Incrementing an `int` at the maximum value results in a negative number, so validation might not work as expected. In this example, we have a requirement to make sure that we can add a line to a file and have the size stay under a million.

```
public static void main(String[] args) {
   System.out.println(enoughRoomToAddLine(100));
   System.out.println(enoughRoomToAddLine(2_000_000));
   System.out.println(enoughRoomToAddLine(Integer.MAX_VALUE));
}

public static boolean enoughRoomToAddLine(int requestedSize) {
   int maxLength = 1_000_000;
   String newLine = "END OF FILE";

   int newLineSize = newLine.length();
   return requestedSize + newLineSize < maxLength;
}
```

The output of this program is as follows:

```
true
false
true
```

The first `true` should make sense. We start with a small file and add a short line to it. This is definitely under a million. The second value is `false` because two million is already over a million even after adding our short line.

Then we get to the final output of `true`. We start with a giant number that is over a million. Adding a small number to it exceeds the capacity of an `int`. Java overflows the number into a very negative number. Since all negative numbers are under a million, the validation doesn't do what we want it to.

When accepting numeric input, you need to verify it isn't too large or too small. In this example, the input value `requestedSize` should have been checked before adding it to `newLineSize`.

Wasting Data Structures

One advantage of using a `HashMap` is that you can look up an element quickly by key. Even if the map is extremely large, a lookup is fast as long as there is a good distribution of hashed keys.

Hacker Harry likes assumptions. He creates a class where `hashCode()` always returns 42 and puts a million of them in your map. Not so fast anymore.

This one is harder to prevent. However, beware of untrusted classes. Code review can help detect the Hacker Harry in your office.

Similarly, beware of code that attempts to create a very large array or other data structure. For example, if you write a method that lets you set the size of an array, Hacker Harry can repeatedly pick a really large array

size and quickly exhaust the program's memory. Input validation is your friend. You could limit the size of an array parameter or, better yet, don't allow the size to be set at all.

Real World Scenario

LEARNING MORE

This exam covers security as it applies to stand-alone applications. On a real project, you are likely to be using other technologies. Luckily, there are lists of things to watch out for.

Open Web Application Security Project (OWASP) publishes a top 10 list of security issues. Some will sound familiar from this chapter, like injection. Others, like cross-site scripting (XSS), are specific to web applications. XSS involves malicious JavaScript.

If you are deploying to a cloud provider, like Oracle Cloud or AWS, there is even more to be aware of. The Cloud Security Alliance (CSA) also publishes a security list. Theirs is called the Egregious Eleven. This list covers additional worries such as account hijacking.

We've included links to the OWASP Top 10 and Egregious Eleven on our book page.

http://www.selikoff.net/ocp11-2

This chapter is just a taste of security. To learn more about security beyond the scope of the exam, please read *Iron-Clad Java*, Jim Manico and August Detlefsen (Oracle Press, 2014).

Summary

When designing a class, think about what it will be used for. This will allow you to choose the most restrictive access modifiers that meet your re-

quirements. It will also help you determine whether subclasses are needed or whether the class should be `final`. If instances of the class are going to be passed around, it may make sense to make the class immutable so the state is guaranteed not to change.

Injection is an attack where dangerous input can run. SQL injection is prevented using a `PreparedStatement` with bind variables. Command injection is prevented with input validation and security policies. Whitelisting and the principle of least privilege provide the safest combination.

Confidential information must be handled carefully. It should be carefully dealt with in log files, output, and exception stack traces. Confidential information must also be protected in memory through the proper data structures and object lifecycle.

Object serialization and deserialization needs to be designed with security in mind as well. The `transient` modifier flags an instance variable as not being eligible for serialization. More granular control can be provided with the `serialPersistentFields` constant. It is used to constrain the `writeObject()` method with `PutField` and the `readObject()` method with `GetField`. Finally, the `readResolve()` and `writeReplace()` methods allow you to return a different object or class.

Regardless of whether you are using serialization, objects must take care that the constructor cannot call methods that subclasses can override. Methods that are called from the constructor should be `final`. Making the constructor `private` or the class `final` also meets this requirement.

Finally, applications must protect against denial of service attacks. The most fundamental technique is always using try-with-resources to close resources. Applications should also validate file sizes and input data to ensure data structures are used properly.

Exam Essentials

- **Identify ways of preventing a denial of service attack.** Using a try-with-resources statement for all I/O and JDBC operations prevents resource leaks. Checking the file size when reading a file prevents it from using an unexpected amount of memory. Confirming large data structures are being used effectively can prevent a performance problem.
- **Protect confidential information in memory.** Picking a data structure that minimizes exposure is important. The most common one is using `char[]` for passwords. Additionally, allowing confidential information to be garbage collected as soon as possible reduces the window of exposure.
- **Compare injection, inclusion, and input validation.** SQL injection and command injection allow an attacker to run expected commands. Inclusion is when one file includes another. Input validation checks for valid or invalid characters from users.
- **Design secure objects.** Secure objects limit the accessibility of instance variables and methods. They are deliberate about when subclasses are allowed. Often secure objects are immutable and validate any input parameters.
- **Write serialization and deserializaton code securely.** The `transient` modifier signifies that an instance variable should not be serialized. Alternatively, `serialPersistenceFields` specifies what should be. The `readObject()`, `writeObject()`, `readResolve()`, and `writeReplace()` methods are optional methods that provide further control of the process.

Review Questions

The answers to the chapter review questions can be found in the Appendix.

1. How many requests does it take to have a DDoS attack?
   A. None
   B. One

C. Two

D. Many

2. Which of the following is the code an example of? (Choose all that apply.)

```
public final class Worm {
   private int length;

   public Worm(int length) {
      this.length = length;
   }
   public int getLength() {
      return length;
   }
}
```

A. Immutability

B. Input validation

C. Limiting accessibility

D. Restricting extensibility

E. None of the above

3. Which can fill in the blank to make this code compile?

```
import java.io.*;

public class AnimalCheckup {
   private String name;
   private int age;

   private static final ObjectStreamField[]
      serialPersistentFields =
      { new ObjectStreamField("name", String.class)};

   private void writeObject(ObjectOutputStream stream)
      throws Exception {
```

```
            ObjectOutputStream. fields = stream.putFields();
            fields.put("name", name);
            stream.writeFields();
        }
        // readObject method omitted
    }
```

A. `PutField`
B. `PutItem`
C. `PutObject`
D. `UpdateField`
E. `UpdateItem`
F. `UpdateObject`

4. Which of the following can fill in the blank to make a defensive copy of `weights`? (Choose all that apply.)

```
public class Turkey {
    private ArrayList<Double> weights;
    public Turkey(ArrayList<Double> weights) {
        this.weights = ;
    }
}
```

A. `weights`
B. `new ArrayList<>(weights)`
C. `weights.clone()`
D. `(ArrayList) weights.clone()`
E. `weights.copy()`
F. `(ArrayList) weights.copy()`

5. An object has validation code in the constructor. When deserializing an object, the constructor is called with which of the following?
A. `readObject()`
B. `readResolve()`
C. Both
D. Neither

6. Which statements are true about the `clone()` method? (Choose all that apply.)
   A. Calling `clone()` on any object will compile.
   B. Calling `clone()` will compile only if the class implements `Cloneable`.
   C. If `clone()` runs without exception, it will always create a deep copy.
   D. If `clone()` runs without exception, it will always create a shallow copy.
   E. If `clone()` is not overridden and runs without exception, it will create a deep copy.
   F. If `clone()` is not overridden and runs without exception, it will create a shallow copy.
7. Which attack could exploit this code?

```
public boolean isValid(String hashedPassword)
   throws SQLException {
   var sql = "SELECT * FROM users WHERE password = ?";
   try (var stmt = conn.prepareStatement(sql)) {
      stmt.setString(1, hashedPassword);
      try (var rs = stmt.executeQuery(sql)) {
         return rs.next();
      }
   }
}
```

   A. Command injection
   B. Confidential data exposure
   C. Denial of service
   D. SQL injection
   E. SQL leak
   F. None of the above
8. You go to the library and want to read a book. Which is true?

```
grant {
   permission java.io.FilePermission
      "/usr/local/library/book.txt",
      "read,write";
};
```

A. The policy is correct.
B. The policy is incorrect because file permissions cannot be granted this way.
C. The policy is incorrect because `read` should not be included.
D. The policy is incorrect because the permissions should be separated with semicolons.
E. The policy is incorrect because `write` should not be included.

9. Which are true about securing confidential information? (Choose all that apply.)
   A. It is OK to access it in your program.
   B. It is OK to have it in an exception message.
   C. It is OK to put it in a `char[]`.
   D. It is OK to share it with other users.
   E. None of the above
10. Which types of resources do you need to close to help avoid a denial of service? (Choose all that apply.)
    A. Annotations
    B. Exceptions
    C. I/O
    D. JDBC
    E. String
11. Which of the following are considered inclusion attacks? (Choose all that apply.)
    A. Billion laughs attack
    B. Command injection
    C. CSRF
    D. SQL injection
    E. XSS
    F. Zip bomb

12. What is this code an example of?

```
public void validate(String amount) {
   for (var ch : amount.toCharArray())
      if (ch < '0' || ch> '9')
         throw new IllegalArgumentException("invalid");
}
```

    A. Blacklist
    B. Graylist
    C. Orangelist
    D. Whitelist
    E. None of the above

13. Which of the following are true statements about a class `Camel` with a single instance variable `List<String> species`? (Choose all that apply.)
    A. If `Camel` is well encapsulated, then it must have restricted extensibility.
    B. If `Camel` is well encapsulated, then it must be immutable.
    C. If `Camel` has restricted extensibility, then it must have good encapsulation.
    D. If `Camel` has restricted extensibility, then it must be immutable.
    E. If `Camel` is immutable, then it must have good encapsulation.
    F. If `Camel` is immutable, then it must restrict extensibility.

14. Which locations require you to be careful when working with sensitive data to ensure it doesn't leak? (Choose all that apply.)
    A. Comments
    B. Exception stack traces
    C. Log files
    D. `System.out`
    E. Variable names
    F. None of the above

15. What modifiers must be used with the `serialPersistentFields` field in a class? (Choose all that apply.)
    A. `final`

B. `private`
C. `protected`
D. `public`
E. `transient`
F. `static`

16. What should your code do when input validation fails? (Choose all that apply.)
    A. Call `System.exit()` immediately.
    B. Continue execution.
    C. Log a message.
    D. Throw an exception.
    E. None of the above
17. Which techniques can prevent an attacker from creating a top-level subclass that overrides a method called from the constructor? (Choose all that apply.)
    A. Adding `final` to the class
    B. Adding `final` to the method
    C. Adding `transient` to the class
    D. Adding `transient` to the method
    E. Making the constructor `private`
    F. None of the above
18. Which of these attacks is a program trying to prevent when it checks the size of a file?
    A. Denial of service
    B. Inclusion
    C. Injection
    D. None of the above
19. Fill in the blank with the proper method to deserialize an object.

```
public Object  ____________ throws ObjectStreamException {
   // return an object
}
```

    A. `readObject()`

B. `readReplace()`

C. `readResolve()`

D. `writeObject()`

E. `writeReplace()`

F. `writeResolve()`

20. The following code prints `true`. What is true about the `Wombats` class implementation of the `clone()` method?

```
Wombats original = new Wombats();
original.names = new ArrayList<>();
Wombats cloned = (Wombats) original.clone();
System.out.println(original.getNames() == cloned.getNames());
```

A. It creates a deep copy.

B. It creates a narrow copy.

C. It creates a shallow copy.

D. It creates a wide copy.