JAVA™ 2

SUN™ CERTIFIED PROGRAMMER & DEVELOPER

13

Clarity and Maintainability

CERTIFICATION OBJECTIVE

- Writing Clear and Maintainable Code

CERTIFICATION OBJECTIVE

Write Clear and Maintainable Code

Now that you've made your code *readable*, does your easy-to-read code actually make sense? Can it be easily maintained? These are huge issues for the exam, worth a very significant chunk of your assessment score. We'll look at everything from class design to error handling. *Remember that you're a Team Player.* Some key areas of code clarity are covered in more detail in the Documentation chapter, so we won't discuss them here. Those areas include the importance of meaningful comments and self-documenting identifiers. The issues raised in *this* chapter are

- General programming style considerations
- Following OO design principles
- Reinventing the wheel
- Error-handling

General Programming Considerations

The coding conventions covered in the previous chapter are a great starting point. But the exam is also looking for consistency and appropriateness in your programming *style*. The following section lists some key points you should keep in mind when writing your perfectly-formatted code. Some of these will be explained in subsequent sections; several of these points are related to OO design, for example, and we cover them in more detail in that section. Once again, this is no time to debate the actual *merits* of these principles. Again, imagine you've come into a project team and need to prove yourself as a, what? Yes! *Team Player*. The first thing the team is looking for is whether you can follow the conventions and standards so that everyone can work together without wanting to throw one another out the seventh floor window and onto the cement fountain below. (Unless you're a dot-com company and your office now looks over an abandoned gas station.) These points are in no particular order, so don't infer that the first ones are more important than the last. You can infer, however, that your exam assessor will probably be asking if you've done these things appropriately.

Keep Variable Scope as Small as Possible

Don't use an instance variable when a local variable will work! Not only does this impact memory use, but it reduces the risk that an object "slips out" to some place it shouldn't be used, either accidentally or on purpose. Wait to declare a variable until just before it's used. And you should *always* initialize a local variable at the time it is declared (which is just before use), with the exception of *try/catch* blocks. In that case, if the variable is declared and assigned in the *try/catch* block, the compiler won't let you use it *beyond* that block, so if you need the variable after a *try* or *catch* block, then you'll have to declare it first *outside* the *try/catch*.

Another way to reduce scope is to use a *for* loop rather than *while*. Remember from the Programmer's exam chapters that when you declare a variable as part of the *for* loop declaration (as opposed to merely initializing a variable declared prior to the loop), then the variable's scope ends with the loop. So you get scope granularity that's even smaller than a method.

Avoid Designing a Class That Has No Methods

Objects are meant to have both state and behavior; they're not simply glorified structs. If you need a data structure, use a Collection. There are exceptions to this, however, that might apply to your exam assignment. Sometimes you *do* need an object whose sole purpose is to carry data from one location to another—usually as a result of a database request. A row in a table, for example, should be represented as an object in your Java program, and it might not always need methods if its sole job is to be, say, displayed in a GUI table. This is known as the *ValueObject* pattern. Which brings us to the next issue.

Use Design Patterns

When you use familiar patterns, then you've got a kind of shorthand for discussing your design with other programmers (even if that discussion is between your code/comments and the other person. If you've done it right, *you* won't personally be there to talk about it, as is the case with the Developer exam). If you need a Singleton, make a Singleton—don't simply document that there is to be only one of these things. On the other hand, don't go forcing your design into a pattern just for the sake of using a pattern. Simplicity should be your first concern, but if it's a toss-up between your approach and an equally complex, well-known design pattern, go for the pattern.

Reduce the Visibility of Things As Much As Possible

In general, the more public stuff you expose to the world, the less free you are to make changes later without breaking someone else's code. The less you expose, the more flexibility you have for implementation changes later. And you *know* there are always changes. So, making variables, methods, and classes as restricted as you can while limiting what you expose to your "public interface," you'll be in good shape down the road. Obviously there are other subtle issues about inheritance (as in, what does a subclass get access to?), so there's more to consider here, but in general, be thinking about reducing your exposure (think of it as reducing your liability down the road). This is closely related to reducing the scope of variables.

Use Overloading Rather Than Logic

If you've got a method that needs to behave differently depending on the kind of thing it was actually handed, consider overloading it. Any time you see *if* or *switch* blocks testing the type of an argument, you should probably start thinking about overloading the method. And while you're at it…

Avoid Long Argument Lists

If you have a ton of arguments coming into a method, perhaps you need to encapsulate the stuff you need in that method into a class of its own type.

Don't Invoke Potentially Overridable Methods from a Constructor

You already know that you can't access any nonstatic things *prior* to your superconstructor running, but keep in mind that even *after* an object's superconstructor has completed, the object is still in an incomplete state until after *its* constructor has finished. Polymorphism still works in a constructor. So if B extends A, and A calls a method in its constructor that B has overridden, well, guess what happens when somebody makes an instance of B. You got it. The B constructor invokes its superconstructor (A's constructor). But inside the A constructor it invokes one of its own methods, but B has overridden that method. B's method runs! In other words, an object can have one of its methods invoked even *before* its constructor has completed! So while B isn't even a fully formed object, it can still be running code and even accessing its own instance variables. This is a problem because its instance variables have not yet been initialized to

anything other than default values, *even if they're given explicit values when they're declared.* Yikes! So don't do it. If it's a final or private instance method, then you're safe since you know it'll never be overridden.

Code to Interfaces

Polymorphism, polymorphism, polymorphism. Use polymorphic arguments, return types, and variables whenever possible (in other words, declare a variable, return type, or argument as an interface type rather than a specific class type). Using an interface as the type lets you expose only the definition of *what* your code can do, and leaves the implementation flexible and extensible. And maintainable. And all the other good OO things-that-end-with-ble. But if you can't…

Use Abstract Classes When You Need Functionality to Be Inherited

If you really must have implementation code and/or instance variables, then use an abstract class and use that class as the declared polymorphic variable, argument, and return type.

Make Objects You're Finished with Eligible for Garbage Collection

You already know how to do this. Either explicitly set the reference variable to null when you have no more use of the object, or reassign a different object to that reference variable (thus abandoning the object originally referenced by it). At the same time…

Don't Make More Objects Than You Need To

Just because there's a garbage collector doesn't mean you won't have "memory issues." If you keep too many objects around on the heap, ineligible for garbage collection (but you won't, having read the preceding point), then you can still run out of memory. More likely, though, is just the problem that your performance might be slightly degraded by the overhead of both making all those objects and then having the garbage collector reclaim them. Don't do *anything* to alter your design just to shave a few objects, but pay attention in your implementation code. In some cases, you might be able to simply reuse an existing object by resetting its state.

Avoid Deeply Nested and Complex Logic

Less is more when it comes to branching. In fact, your assessor may be applying the Cyclomatic Complexity measure to your code, which considers code to be complex *not* based on lines of code, but rather on how many branch points there are. (It's actually much more complex than that. Ironically, the test for code complexity is itself a rather complex formula.) The bottom line is, whenever you see a nested *if* or anything other than very simple logic flow in a method, you should seriously consider redesigning that method or splitting functionality into separate methods.

Use Getters and Setters That Follow the JavaBean Naming Convention

That means you should use `set`<yourPropertyName> for methods that can modify a property (normally a property maps directly to an instance variable, but not necessarily) and `get`<yourPropertyName> for methods that can read a property. For example, a String variable *name* would have the following getter/setter methods:

```
setName(String name)
String getName()
```

If the property is a boolean, then you have a choice (yes, you actually have a choice) of whether to call the read method get<property> or is<property>. For example, a boolean instance variable *motorOn* can have the following getter/setter methods:

```
setMotorOn(boolean state)
boolean getMotorOn()
boolean isMotorOn()
```

The beauty of adhering to the JavaBeans naming convention is that, hey, you have to name it *something* and if you stick with the convention, then most Java-related tools (and some technologies) can read your code and automatically detect that you have editable properties, for example. It's cool; you should do it.

Don't Be a Procedural Programmer in an OO World

The two dead giveaways that you haven't really made the transition to a complete object "being," are when you use the following:

- **Really Big Classes** that have methods for everything.

- **Lots of static methods.** In fact, *all* methods should be nonstatic unless you have a truly good reason to make them static. This is OO. We don't have global variables and functions. There's no "start here and then keep executing linearly except when you branch, of course…". This is OO, and that means *objects all the way down.*

Make Variables and Methods As Self-Explanatory As Possible

Don't use variable names like x and y. What the heck does this mean: int x = 27; 27 *what*? Unless you really think you can lock up job security by making sure *nobody* can understand your code (and assuming the homicidal maniac who tries won't find you), then you should make your identifiers as meaningful as possible. They don't have to be *paragraphs*. In fact, if it takes a paragraph to explain what a variable represents, perhaps you need to think about your design again. Or at the least, use a comment. But don't make them terse! Take a lesson from the core APIs. They could have called `ArInBException`, but instead they called it `ArrayIndexOutOfBoundsException`. Is there *any* question about what that exception represents? Of course, the big Sun *faux pas* was the infamous `NullPointerException`. But despite the use of the forbidden word *pointer*, everybody knows what it means when they get it. But there could be some confusion if it were called `NPTException` or even `NullException`.

Use the Core APIs!

Do not reinvent the wheel, and do *not*—or you'll automatically fail for certain—use any libraries other than code *you* developed and the core Java APIs. Resist any temptation to think that you can build something faster, cleaner, more efficient, etc. Even if that's true, it isn't worth giving up the benefit of using standard classes that others are familiar with, and that have been *extremely, heavily tested in the field.*

Make Your Own Exception Classes If You Can't Find One That Suits Your Needs

If there isn't a perfect checked `Exception` class for you in java.lang, then create your own. And make it specific enough to be meaningful to the catcher. In other words, don't make a `BadThingHappenedException` and throw it for every possible business error that occurs in your program.

Do Not Return Error Codes!

This is Java. This is OO. If you really need to indicate an exceptional condition, use an Exception! If you really want to annoy an assessor, use error codes as return values from some of your methods. Even *one* method might do the trick.

Make Your Exceptions with a String Constructor Argument

Doing so gives you a chance to say more about what happened to cause the exception. When you instantiate an Exception, call the constructor that takes a String (or the one that takes another lower-level exception if you're doing exception chaining). When you create your *own* Exception class, be sure to put in a constructor that takes a String.

Follow Basic OO Design Principles

In the preceding section, some of the key points touched on areas we'll dig a bit deeper into here. You don't have to be the World's Best OO Designer, but you do need to follow the basic principles on which the benefits of OO depend. Obviously we can't make this a "How to Be a Good OO Designer in 10 Easy Pages." You need a lot more study and practice, which we assume you've already done. This should be old news by now, but you can bet that your assessor will be looking at these issues, so a refresher won't hurt. We're hitting the highlights of areas where you might get points deducted from your assignment.

Hide Implementation Details

This applies in so many places, but coding with interfaces and using encapsulation is the best way to do it. If you think of your code as little self-contained, pluggable components, then you don't want anyone who uses one of your components to have to think about *how* it does what it does. It all comes down to inputs and outputs. A public interface describes *what* a method needs from you, and *what* it will return back to you. It says nothing about *how* that's accomplished. You get to change your implementation (even the *class* doing the implementing) without affecting calling code. Implementation details can also be propagated through exceptions, so be careful that you don't use an interface but then put implementation-specific exceptions in the throws clause! If a client does a "search," they shouldn't have to catch an SQLException, for example. If your implementation code happens to be doing database work that can generate SQLExceptions (like JDBC code would), the client

should not have to know that. It's your job to catch that implementation-specific exception and throw something more meaningful—a *business-specific* exception—back to client code.

Use Appropriate Class Granularity

A class should be of the right, you know, *granularity*. It shouldn't be too big or too tiny. Rarely is the problem a class that's too *small*; however, most not-quite-OO programmers make classes that are too *big*. A class is supposed to represent a *thing* that has state and behaviors. Keep asking yourself, as you write each method, if that behavior might not be better suited for some *other* thing. For example, suppose you have a Kitchen class that does all sorts of Kitchen things. Like Oven things and Refrigerator things, etc. So now you've got Kitchen things (Kitchen being a *room*) and Refrigerator things and Oven things all in the same class. That's three different things. Classes (and thus the objects instantiated from them) really should be *specialists.* They should do the kinds of behaviors that a *thing* of that type *should* do, and no more. So rather than having the Kitchen class include all the code for Refrigerator and Oven behaviors, have the Kitchen class *use* a Refrigerator and Oven in a HAS-A relationship.

This keeps all three classes simple, and reusable. And that solves your naming problem, so that you don't have to name your do-everything Kitchen class *KitchenFridgeOven*.

Another possible cause of a Big Class is that you've got too many inner classes defined. *Too many* meaning some of the inner classes should have been either top-level classes (for reuse) or simply methods of the enclosing class. Make sure your inner or nested classes really need to be included.

Limit Subclassing

If you need to make a new subclass to add important functionality, perhaps that functionality should really be in the parent class (thus eliminating the need for the subclass—you just need to *fix* the superclass). When you feel the need to extend a class, *always* look at whether the parent class should change, or whether you need *composition* (which means using HAS-A rather than IS-A relationships). Look in the core Java API for a clue about subclassing versus composition: the core API inheritance hierarchy is *really* wide but very shallow. With a few exceptions (like GUI components), most class hierarchies are no more than two to three levels deep.

Use Appropriate Method Granularity

Just as classes should be specialists, so too should methods. You'll almost certainly be docked points for your assignment if your methods are long (although in some cases, especially in your Swing GUI code, long methods aren't necessarily a reflection of bad design). In most cases, though, the longer the method the more complex, because often a long method is a reflection of a method *doing too much*. You're all programmers so we don't have to hammer the point about smaller modular functionality—*much* easier to debug, modify, reuse, etc. Always see if it makes sense to break a longer method up into smaller ones. But while in a deadline crunch you might get away with long methods in the *real* world (feeling guilty of course), it won't fly for your Developer assignment.

Use Encapsulation

Your assignment will be scrutinized for this most fundamental OO principle. Expect the assessor to look at the way in which you've controlled access to the state of your object. In other words, the way you've protected your instance variables with setters and getters. No need to discuss it here, just do it. Allow access to your data (except for constants, of course) *only* through more accessible methods. Be careful about your access modifiers. Having a nice set of accessor methods doesn't matter if you've left your variables wide-open for direct access. Again, make things as private and scope-limited as you can.

Isolate Code That Might Change from Code That Won't Have To

When you design your classes, be sure to separate out the functionality that might change into separate classes. That way, you restrict the places where you'll have to track down and make modifications as the program evolves.

Don't Reinvent the Wheel

Why would you want to? Well, most people end up doing it for one of two reasons:

- They believe they can do it *better*.
- They didn't know there already *was* a wheel.

You need to be certain that you

- Get it out of your head that you can do it better, regardless of whether you actually *can.* A better mousetrap (to completely mix metaphors here) isn't what's required. A solid, maintainable design *is.*
- Always look for an existing solution first!

Use Core APIs

Always always always check the core APIs, and know that occasionally you might find the class you're looking for in a package *other* than where you'd expect it. So be sure to really search through the APIs, even digging into packages and classes you might think are a little off the path. Sometimes a solution can be where you least expect it, so stay open to approaches that aren't necessarily the ones you would normally take. Flipping through a reference API book can help. A method might catch your eye and even if it turns out *not* to be your solution, it might spark an idea about a different solution.

In some cases, you might not find *exactly* what you're looking for, but you might find a class you can extend, thus inheriting a bunch of functionality that you now won't have to write and test (subject to the warnings about subclassing we mentioned previously).

Using core API's (besides being essential for the exam) lets you take advantage of a ton of expertise and testing, plus you're using code that hundreds of thousands of other Java developers are familiar with.

Use Standard Design Patterns

We can't tell you which ones you'll actually need for your assignment; that depends on both your assignment and your particular approach. But there are plenty of standard design patterns that let you take advantage of the collective experience of all those who've struggled with your issue before you (although usually at a fairly abstract level—that's usually where most patterns do their work). So while the core APIs let you take advantage of someone else's implementation code, design patterns let you take advantage of someone else's *approach to a problem.*

If you put a gun to our heads, though, we'd probably have to say that Singleton should be way up on your list of things to consider when developing your assignment. But you *might* also take a look at MVC (for your client GUI), Façade, Decorator, Observer, Command, Adapter, Proxy, and Callback, for starters. Pick up a book on design patterns (the classic reference is known as the "Gang of Four" (GOF) book,

Design Patterns: Elements of Reusable Object-Oriented Software, by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides) and take time to step back and look at where your program might be trying to do something well-solved by a design pattern. The patterns don't tell you how to construct your algorithms and implement your code line by line, but they can guide you into a sound and maintainable design. Perhaps most importantly, as design patterns are becoming more and more well-known, developers have a common vocabulary to discuss design trade-offs and decisions.

We believe that the use of design patterns has recently become more important in the exam assessment than it has been in the past, due in large part to their growth in popularity.

Handle Errors Appropriately

You'll be evaluated for appropriate and clear error-handling throughout your project. You might do really well with it in your GUI and then fall down in your server, but it matters everywhere in your program.

Don't Return Error Codes

This is Java. Using error codes as return values, rather than using exceptions, is a Really Bad Idea. We're pretty sure your exam assessor knows that.

Don't Send Out Excessive Command-Line Messages

Don't be too verbose with your command-line messages, and be sure not to leave debugging messages in! Your command-line messages should include only what's necessary to verify the startup of your programs and a *very* minimal amount of status messages that might be crucial if the program fails. But in general, if something goes wrong that you *know* could go wrong, you should be handling it with exceptions.

Whatever you do, don't use command-line messages to send alert messages to the user! Use a proper dialog box if appropriate.

Use Dialog Boxes Where Appropriate

On the other hand, don't use dialog boxes for every possible message the user might need to know about. If you need to display information to the user that isn't of an urgent nature (urgent being things like a record-locking problem or if you need to

offer a "Are you sure you want to Quit?" option). In many cases, a dialog box is what you'll use to alert the user when something in your program has caught an exception, and you need user input to deal with it appropriately. The use of dialog boxes from a usability perspective will be covered in more detail in Chapter 14.

Throw Checked Exceptions Appropriately

There's a correct time and place for throwing checked exceptions, and being *reluctant* to throw them can be just as bad as throwing them carelessly.

- Use runtime exceptions for programming errors.
- Use checked exceptions for things that your code might recover from (possibly with help from the user).
- Checked exceptions are *only* for truly exceptional conditions.
- Do not use exceptions for flow control! Well, not if you hope to do well both on the exam and in real life.

Remember, checked exceptions sure don't come for free at runtime; they've got overhead. Use them when, *but only when*, you need them.

Create and Throw Your Own Exceptions When Appropriate

Make use of standard exceptions when they make sense, but never hesitate to create your own if appropriate. If there's a reasonable chance that an exceptional condition can be recovered from, then use a checked exception and try to handle it. Normally, the exceptions that you create can be thought of as Business Exceptions—in other words, things like "RecordLockedException" or "InsufficientSearchCriteriaException". The more specific your exception, the more easily your code can handle it, and you get the benefit of providing specific catch blocks, thus keeping the granularity of your catch blocks useful. The opposite of that strategy would be to simply have everything in one big *try* block that catches `Exception` (or worse, `Throwable`!).

Catch Low-Level Implementation Exceptions and Throw a Higher-Level Business Exception

Say you catch an SQLException (not likely on the Developer exam). Do you throw this back to a client? Of course not. For a client, it falls into the category of "too

much information." The client should not know—or care—that the database server happens to be using SQL. Instead, throw back to the client a more meaningful custom business exception that he or she can deal with. That more meaningful *business* exception is defined in your public interface, so the client is expecting it as a possibility. But simply passing a low-level exception all the way to a client reflects a poor design, since it couples the client with implementation details of the server—that's *never* a good idea in an OO design.

Make Your Exception Classes with a String Constructor (As Well As a no-arg) for Providing Additional Meaning

Every Exception class you develop should have both a no-arg constructor and a constructor that takes a String. Exception inherits a `getMessage()` method from Throwable, and it returns the String of that message, so you can pass that message back to your super constructor and then the catcher can query it for more information. The message's main use, however, is to provide more information in the stack trace. So the more detailed your message (usually about the state of key parts of the system at the time the Exception occurs), the more helpful it will be in diagnosing the problem.

Never, Ever, Ever *Eat* an Exception

By *eat* we mean the following horrible practice:

```
try {
   doRiskyThing();
} catch(Exception e) {}
```

See what's missing? By catching the exception and then not handling it in any way, it goes completely unnoticed, as if it never occurred. You should at the *least* print the stack trace. Putting something like this in your exam project might be the death blow.

Announce ALL Your Exceptions (Not Their Superclasses) in Method Declarations

Your method should declare the exact, specific Exception types that it can throw, as opposed to declaring a supertype. The following code shows an example:

```
class MyException extends Exception { }
class FooException extends MyException { }
class BooException extends MyException { }
public class TestException {
   public void go() throws MyException {  // Usually BAD to do this
     boolean x = true;
     if(x) {
      throw new FooException();
     } else {
      throw new BooException();
   }
  }
}
```

In the preceding code, class `TestException` declares a method `go()` that declares a `MyException`. But in reality, it might throw a `BooException` or it might throw a `FooException`. This is perfectly legal, of course, since both exceptions are subclasses of the declared exception. But why bother throwing two different exceptions if you don't declare it? Surely you don't want to force the catcher to insert logic to figure out *what* kind of exception they got? This doesn't mean that `catch` code won't sometimes do this, but it should be up to the catcher, not the thrower, to make that choice.

Key Points Summary

That wraps up our look at clarity and maintenance issues, and here's a list of the key points. Cut it out and tape it to your wall next to all the other incredibly valuable pages you've ripped from this book and taped to your wall. We're thinking of just offering wallpaper so you can leave your book intact.

General Programming Considerations

- Avoid designing a class that has no methods.
- Use design patterns.
- Reduce the visibility of things as much as possible.
- Use overloading rather than logic.
- Avoid long argument lists.

- Don't invoke potentially overridable methods from a constructor.
- Code to interfaces.
- Use abstract classes when you need implementation functionality.
- Make objects you're finished with eligible for garbage collection.
- Don't make more objects than you need to.
- Avoid deeply nested and complex logic.
- Use getters and setters that follow the JavaBean naming convention.
- Don't be a procedural programmer in an OO world.
- Make variable and method names as self-explanatory as possible.
- Make your own Exception classes if you can't find one in the API to suit your needs.
- Don't return error codes.
- Make your exceptions with a String message.

Follow Basic OO Design Principles

- Hide implementation details.
- Use appropriate class granularity.
- Use appropriate method granularity.
- Use encapsulation.

Don't Reinvent the Wheel

- Use core APIs.
- Use standard design patterns.

Handle Errors Appropriately

- Don't return error codes.
- Don't send out excessive command-line messages.
- Use dialogs boxes where appropriate.

- Throw checked exceptions appropriately.
- Create and throw your own exceptions when appropriate.
- Catch low-level implementation exceptions and throw a high-level business exception instead.
- Make your own custom exception classes have a String constructor (to take a detail message).
- Never, ever, eat an exception.
- Announce all your exceptions, not just their supertypes.