Fidelity INVESTMENTS

PROGRAMMING WITH JAVA

EXERCISE MANUAL

ROITRAINING
MAXIMIZE YOUR TRAINING INVESTMENT

INNOVATION STARTS HERE. Leap

This page intentionally left blank.

Table of Contents

General Instructions

Exercises are done by an individual student or in pairs. Workshops are done by assigned workgroups on flip charts or whiteboards provided.

Programming exercises will use Eclipse as the development environment. Some exercises will require a new Eclipse project to be created, others will use an existing Eclipse project as a starting point.

This page intentionally left blank.

Chapter 1: Introduction to Java

Exercise 1.1: Using String Methods (Optional)

Time: 20 minutes

Format: Individual programming exercise

Consider the problem of generating an email address based on someone's name. Given a string representing someone's first and last name, separated by a space, create an email address of the form `lastname.firstname@fidelity.com` in all lowercase. Note the given string is guaranteed to contain only two names, but it may have any number of spaces surrounding those names.

1. Create an Eclipse project called `MyJavaExercises` (or something similar) to hold your solutions to the exercises: **File | New | Project | Java Project**
2. If asked whether to change to the Java perspective, click **No**. We will use the JavaEE perspective which supports, besides Java, many web technologies as well.
3. Right-click the newly created project and select **New | Class**. Name your class `EmailGenerator`. Check the box to create a `main` (`public static void main(String[] args)`). Put your class in an appropriate package (e.g., `com.fidelity.email`).
4. Create a method in this class named `makeEmailFromName()` that takes a `String` and returns a `String`. For now, simply have it return the empty string so the code compiles:

```
public String makeEmailFromName(String name) {
    return "";
}
```

5. In your `main` method, call `makeEmailFromName()` passing in different strings and print the results. If you didn't tick the box to create a `main`, you can just type in the definition. E.g.:

```
public static void main(String[] args) {
    EmailGenerator g = new EmailGenerator();
    // In all cases, expected output is doe.jane@fidelity.com
    System.out.println(g.makeEmailFromName("Jane Doe"));
    System.out.println(g.makeEmailFromName("Jane   Doe"));
    System.out.println(g.makeEmailFromName("  Jane Doe"));
    System.out.println(g.makeEmailFromName("Jane Doe   "));
}
```

6. Run this test code by right-clicking your class or project in the Project or Package Explorer and choosing **Run As | Java Application**. It should print a series of empty strings.
7. Now implement the `makeEmailFromName()` method for each of your strings in turn. Start with the simplest case and make that work, then continue until all the test cases work.
    a. You can find out what String methods are available by using Eclipse's built-in documentation feature. Type the name you chose for the `String` parameter (or any `String` variable), then a period, `.`, and scroll through the list of methods that appear. Those are what Eclipse has determined are available for you to call on this variable. As you move through each method by clicking or using the arrow keys, it will display the documentation as well as the method's signature.
8. Each time you make a change to your `makeEmailFromName()` method, re-run the application to check the output. Remember to make sure that all the previous test cases still work.
9. Compare your tests to others to see if you can come up with tests that break their code.

Notes:

1. We do not normally create a `main` method. In fact, this may be the only time you ever do that in these courses. Although `main` is the Java entry point, we usually start our code in other ways such as running it in a servlet container or using a test framework. We will see this in the next chapter.
2. We do not normally write code that prints to the console. If we need to see some information for debugging or monitoring, we usually use a logging framework.
3. We do not normally write tests that must be checked by eye. This is time-consuming and error-prone. That is what test frameworks are for.

Chapter 2: Test-Driven Development

Exercise 2.1: Practicing TDD

Time: 40 minutes

Format: Instructor-led introduction and then individual programming exercise

Consider the problem of determining how much money is earned in one year by a part-time employee on an hourly rate.

There are two use cases:

1. A long-term part-timer who works a full 52-week year and works a set number of hours each day at a defined pay rate per hour. They take a fixed number of unpaid vacation days.
2. A less established part-timer who works a different number of days each year. Again, they have a fixed hourly pay rate and a fixed number of hours per day.

Steps:

1. Open the `IncomeCalculator` project. This has been set up with the simplest JUnit 5 Maven configuration.
2. Right-click the project and select **New | Class**. Name your class `IncomeCalculator` and put it in a sensible package.
3. Now, right-click the class and select **New | Other | Java | JUnit | JUnit Test Case**. Accept the default class name (`IncomeCalculatorTest`) but change the source folder to `/src/test/java`. If you don't do this, you can still move the code into the other folder manually.
4. Discuss as a class what the first test should be and choose an interface for the `IncomeCalculator`. We suggest a single method called `getAnnualPay()` that returns a `double` and accepts three parameters:
    a. Hours worked per day (`int`)
    b. Hourly pay (`double`)
    c. Number of unpaid holidays (`int`)
5. Implement the first test. Then let Eclipse generate the method for you.
6. Run the first test in Eclipse and see that it fails.
7. Implement the minimum code to make the test pass.

8. Re-run the test and confirm that it does pass.
9. Now, write another test. It is usually better to pursue a particular part of the requirements fully before moving on. So, for example, leave out any handling of number of vacation days until the basic calculation is complete.
10. Run your new test and see that it fails. If it passes, then the test is not testing new functionality.
11. Implement the minimum code to make all the existing tests pass.
12. Continue implementing tests until you are confident that your solution to the first use case is complete. Run the tests in Maven by right-clicking the project and selecting **Run As | Maven test**.
13. Create tests and implement the second use case. We suggest the same method name, but a different set of parameters. Watch out for making the parameters the same, which would stop you overloading the methods. These would work:
    a. Number of days worked (`int`)
    b. Hours worked per day (`int`)
    c. Hourly pay (`double`)
14. Add some negative tests as discussed in the course. Until we have better means, return 0.0 when calculating pay for any illegal values. The following are illegal:
    a. Zero or negative value for any of the following: Hours worked per day and Hourly pay
    b. Negative value for Number of unpaid holidays and Number of days worked
15. Once you have implemented both use cases, look for opportunities to re-factor.

Compare your tests to others to see if you can come up with tests that break their code.

Note how confusing it is to have two methods with the same name, with very similar parameters in a different order.

Exercise 2.2: Refactoring Test Fixtures

Time: 20 minutes

Format: Individual programming exercise

Refactor your solution to the previous exercise to use Test Fixtures where appropriate.

Exercise 2.3: Debugging Derby

Time: 20 minutes

Format: Individual programming exercise

Consider the problem of validating a credit card number based on a *checksum algorithm*. The algorithm works as follows: accumulate a sum based on adding a value obtained from each digit of the credit card moving from left to right, where each k^{th} digit is examined starting with k=0 for the left-most digit.

1. If `k` is even, add the k^{th} digit to the sum.
2. If `k` is odd, multiply the k^{th} digit by two. If the result is greater than or equal to 10, subtract 9. Add this computed value to the sum.

If the resulting sum is divisible by 10, the credit card number passes the checksum test, otherwise, the card number is not valid.

Examine the code in the `DebuggingTDD` project that implements this algorithm.

1. Run its tests – they should all pass even though there is a bug in the code.
2. Use Eclipse's debugger to figure out the problem in the code and understand how it works. Before debugging the code, add a Breakpoint by double-clicking in the left margin beside the line of code you want to stop at. Then when you run the tests, use **Debug As** instead of **Run As** to change into the Debug Perspective. Step through the code to see the values of the variables change without having to add any print statements. Even if you think you know what the bug is, take some time to explore the tools provided by the Debugger.
3. Write at least one test to verify a bug exists within the `isValid()` method.
4. Fix the bug so that all the tests pass (both the old ones and new test so you see *all* green).
5. Add any additional tests you feel are needed to verify the code is robust for a variety of inputs.

Chapter 3: Implementing Classes

Exercise 3.1: Implementing the Model

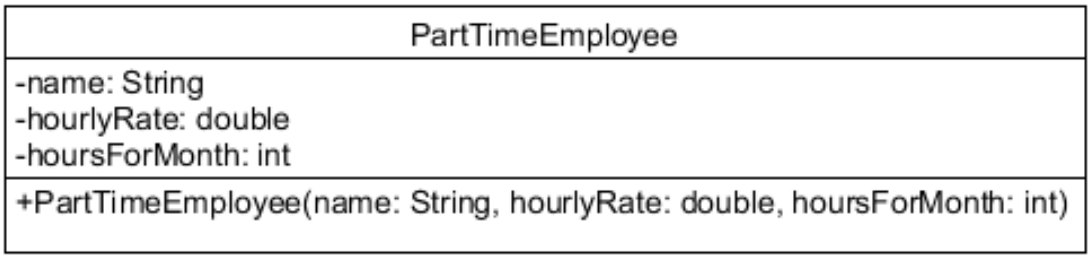
Time: 15 minutes

Format: Individual programming exercise

Create a new Eclipse project, named `MyPayrollExercises`, in which to work on the exercises relating to employees during the course.

Implement this UML class diagram as a Java class with the described constructor. Put your class in a sensible package.

We do not yet have any requirements for getters and setters.



Note: use Eclipse's **Source** menu to generate the constructor for you based on the declared fields to reduce the amount of code you actually write.

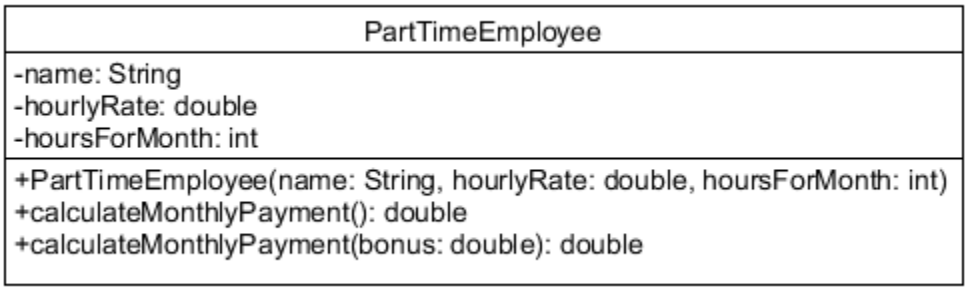
Exercise 3.2: Adding Behavior

Time: 20 minutes

Format: Individual programming exercise

Using TDD, implement the method `calculateMonthlyPayment()` of the `PartTimeEmployee` class you created in the previous exercise. This method takes no parameters and simply calculates the pay due for the entire month based on the object's fields.

Write a second overloaded version of the method that takes a parameter representing a bonus for the month. The given value is simply added to the regular calculated monthly pay.



Note: These methods are similar to the `IncomeCalculator` you wrote in the earlier TDD exercise. Try to build on any lessons you learned.

Chapter 4: Implementing Polymorphism

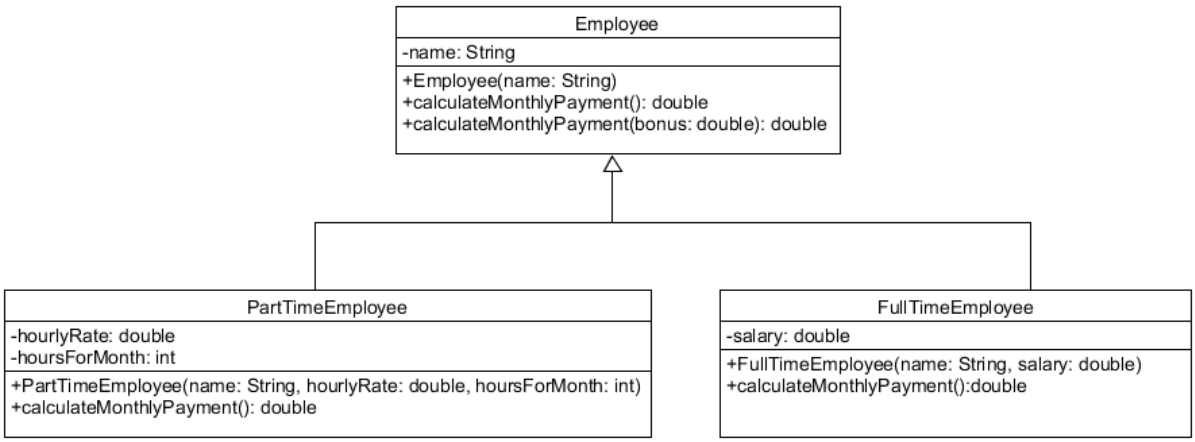
Exercise 4.1: Creating an Inheritance Hierarchy

Time: 30 minutes

Format: Individual programming exercise

Update your `PartTimeEmployee` class from the previous exercises and add new classes to implement this UML class diagram as a Java class hierarchy.

1. The method `calculateMonthlyPayment()` of the `FullTimeEmployee` class returns the yearly salary divided by 12, the number of months in a year.
2. What should the method `calculateMonthlyPayment()` of the generic `Employee` class return? Why did you choose that?



Note the following:

1. Even though you are testing individual subclasses, you should work with the superclass as much as possible. Make all your variables of the superclass type.
2. There is no need to implement the bonus-based monthly calculation in each sub-class if you structure your methods correctly. Can you see how? Initially, you could write the method in each subclass and then examine the code for re-factoring opportunities.

3. In your JUnit tests, note that you can write a helper method (not annotated with `@Test`) to reduce duplicated code that takes the `Employee` superclass and an expected value. This method should call `calculateMonthlyPayment()` and compare its result to the expected value. You may not feel the need to do this, but note that you *can* have helper methods in test classes: they are just not annotated with `@Test`.

Chapter 5: Effective Programming

Exercise 5.1: Using Eclipse to Generate Methods

Time: 15 minutes

Format: Pair programming exercise

Add the following tests to your `MyPayrollExercises` project:

```
@Test
void testFTEmployeeEquality() {
   Employee emp = new FullTimeEmployee("John Doe", 1000);
   assertEquals(new FullTimeEmployee("John Doe", 1000), emp);
   assertNotEquals(new FullTimeEmployee("John Doe", 2000),emp);
}

@Test
void testPTEmployeeEquality() {
   Employee emp = new PartTimeEmployee("John Doe", 10.0, 100);
   assertEquals(new PartTimeEmployee("John Doe", 10.0, 100), emp);
   assertNotEquals(new PartTimeEmployee("John Doe", 10.0, 200), emp);
}
```

When you run them, why do they fail?

Use Eclipse's **Source** menu to generate the three standard Object methods, (`equals()`, `hashCode()`, and `toString()`) for each of the three classes in your `Employee` inheritance hierarchy.

When you run the tests now, why do they pass?

Now, examine the generated methods and discuss the resulting code with your partner to make sure you both understand it. Then discuss how each method supports the effective concepts:

- Do they support equals symmetry?
- Is the same hash code generated for objects with the same values?
- Is the string representation of the object useful and readable?

Discuss with your partner how to create test cases for these concepts. Add JUnit tests for each of these classes based on your discussion.

Why is it necessary to have both `assertEquals()` and `assertNotEquals()` tests? What happens if you only create `equals()` for `Employee`, but not the subclasses?

This page intentionally left blank.

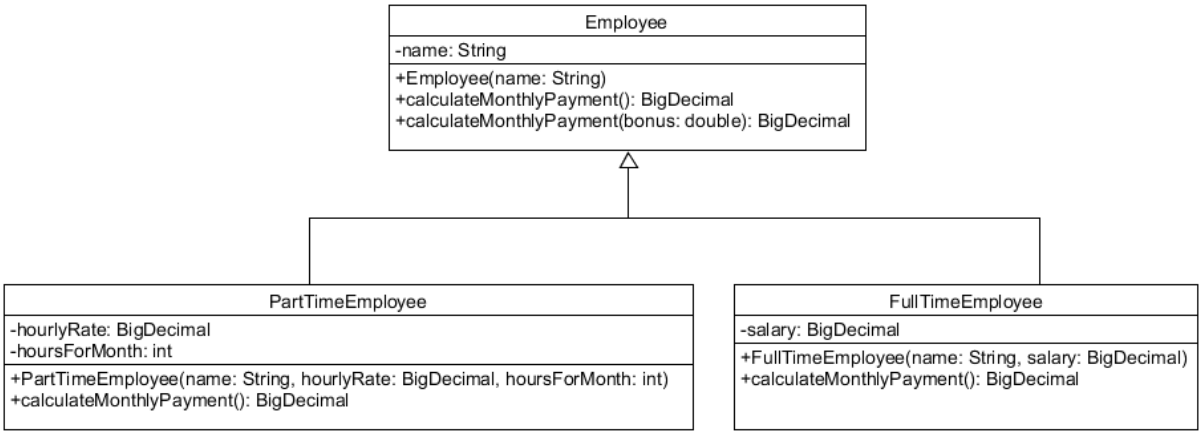
Chapter 6: Useful Classes

Exercise 6.1: `BigDecimal` Practice

Time: 20 minutes

Format: Individual programming exercise

Update your Employee class hierarchy so that all pay amounts are in `BigDecimal`, and work to two decimal places.



Make sure all your tests still pass.

This is a significant re-factoring. You need to decide on an approach that allows you to work on the code piece by piece.

- You could change the method signatures and have the methods do internal conversions.
- You could change the superclass signatures and comment out the subclass implementations until you are ready to work on them, then un-comment them one by one.
- You could implement parallel functionality rather than changing the existing methods.

Remember to re-generate the `Object` methods.

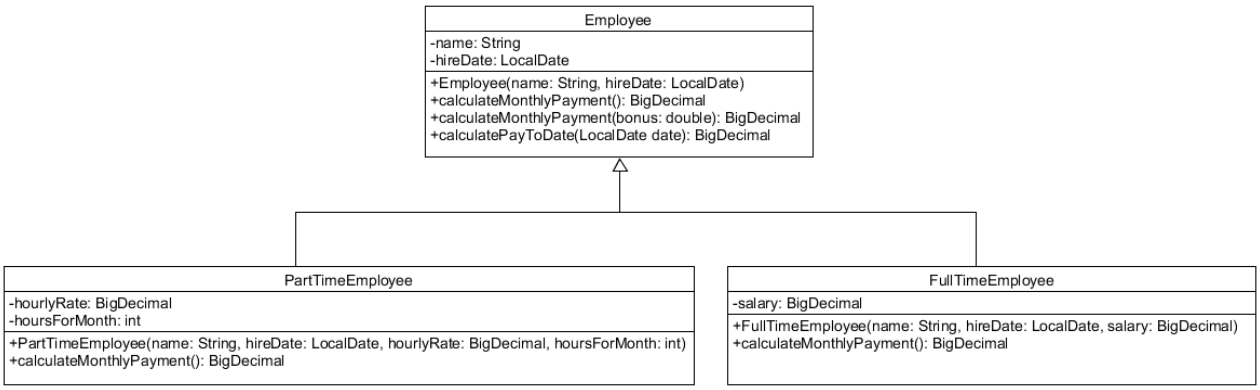
Exercise 6.2: Using Date/Time Methods

Time: 20 minutes

Format: Individual programming exercise

The objective of this exercise is to create a method in Employee that, given any date, will return the total pay received by the employee from the beginning of their employment. It should work in whole months only and not consider any fractional parts.

You will need to add a hire date to the Employee class and its subclasses.



1. Add the `hireDate` first and update existing tests. Remember to re-generate the standard `Object` methods.
2. There are several ways to implement the date calculations. Examine the Javadoc for the java.time package.
3. Don't forget to add tests, and in particular, negative tests. If the date passed to the `calculatePayToDate` method is null or if it is earlier than the start date, the method should return 0.0.

Hint: take a look at this https://docs.oracle.com/javase/tutorial/datetime/iso/period.html.

Chapter 7: Constants and Enumerations

Exercise 7.1: `enum` Practice

Time: 15 minutes

Format: Individual programming exercise

Consider the problem of representing the ESRB ratings for games[1] within a program. This is a discrete number of values that could be represented using integer or String constants, but each rating contains a description in addition to a category abbreviation. Create an enumerated type to represent the 6 ratings categories, including an “unrated” category for those games still awaiting approval.

Your enumerated type should have public methods that return both its full and abbreviated name and a method that, given an age, returns true if a game with this rating is appropriate and false otherwise (for example, it would return `true` for any value if the rating was Everyone).

When writing your tests, remember that multiple ratings may return `true` for the same age.

Create this enumerated type class and associated test class in a suitable package in your `MyJavaExercises` project.

Concentrate on getting the enum structure and one or two ratings implemented, don’t worry about having a full implementation of the standard. For example, you could consider the Everyone and Everyone 10+ ratings.

[1] https://www.esrb.org/ratings/

This page intentionally left blank.

Chapter 8: Abstract Classes and Interfaces

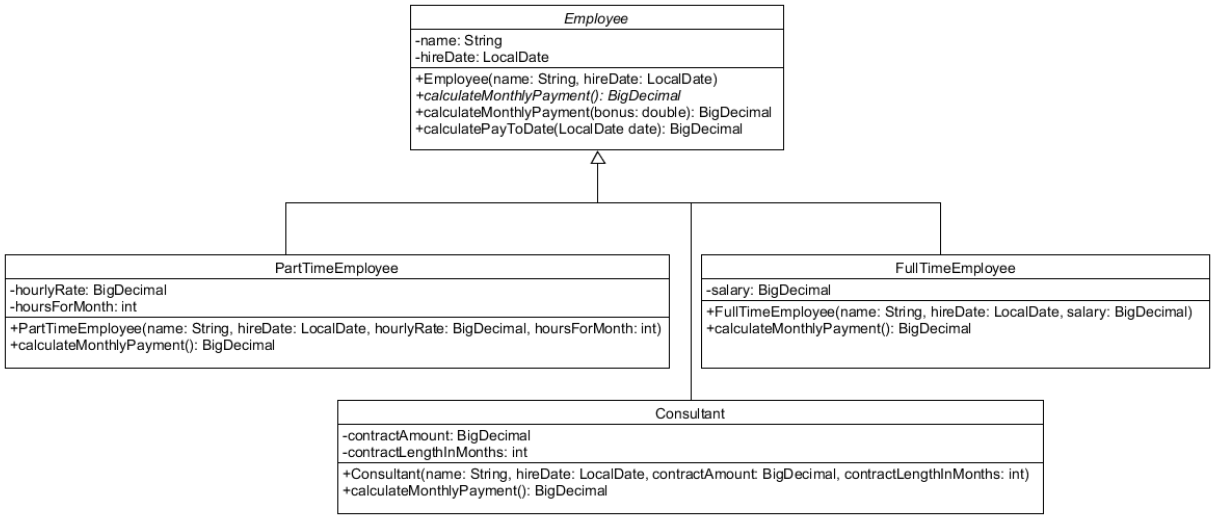
Exercise 8.1: Making `abstract` Superclass

Time: 10 minutes

Format: Individual programming exercise

Make the `Employee` superclass `abstract`. Does this affect any of your existing tests? If so, how would you update them, so they are still effective?

If you have time, implement the Consultant class according to this UML class diagram.



1. The method `calculateMonthlyPayment()` of the `Consultant` class returns the contracted amount divided by the term of the contract.
2. Notice that the bonus-based version of `calculateMonthlyPayment()` is still implemented in the superclass, even though it is abstract. It should just add the given bonus to the standard monthly pay.

When creating the new subclass in Eclipse, make sure the option **Inherit abstract methods** is checked. Examine the source that Eclipse generates for you.

This page intentionally left blank.

Chapter 9: Java Collections Framework

Exercise 9.1: Study the Collection Classes (Optional)

Time: 20 minutes

Format: Workgroup brainstorming

In groups of two or three, study and discuss the Collection classes and share the results of your discussion with the class. If there is something you like or dislike, or something you learned about design during the discussion, make sure to back it up with a specific example. Feel free to search the Internet for other resources beyond the Java documentation and tutorials.

Some things for your group to consider:

- How many interfaces do specific concrete collection classes implement (such as `LinkedList`)? What do you think is the purpose of each interface?
- How many different implementations are there for a specific collection class (such as `Set`)? Do you think the number justifies it being an interface or not?
- How many levels of superclasses do specific concrete collection classes have? What do you think is the purpose of each inheritance level?
- Why does it make sense to have the utility classes instead of adding that functionality to the collection types themselves? Are there any overlapping methods (ones that are in both a specific collection and a utility class)? If so, is there any guidance on which one you should use?

Exercise 9.2: Looping Over `ArrayList`

Time: 15 minutes

Format: Individual programming exercise

Create a utility class that accepts a list of `Employee`s and returns the total monthly pay.

Work TDD, so start by creating a test list and calling the method, then implement the method. Your method should accept `List<Employee>` rather than an implementation like `ArrayList`, but you can use `ArrayList` in your tests to create the list. Your method should not care what subclass of `Employee` is actually in the list.

Further, do not neglect negative tests. Introduce a test for the method being called with `null`, in which case it should return 0.0.

Create additional tests to confirm that your method works for all subtypes of `Employee` and combinations of them

This is polymorphism at work: the method works on `Employee`s, but the calculation differs based on what type of `Employee` is actually involved.

If you have extra time, create a second method that accepts a `LocalDate` and does the total income to date calculation.

Exercise 9.3: Primitive List Practice (Optional)

Time: 15 minutes

Format: Individual programming exercise

Consider the problem of maintaining a list of only non-negative values. Given a `List<Integer>` that contains both positive and negative values, remove any negative ones from the list, leaving only values that are non-negative. If all the numbers in the given `List` are non-negative, none should be removed.

You will *not* be able to use the simple `for-each` loop to solve this problem because it does not allow you to remove items from a list during iteration (try it – what error do you get from Java?). Consider what type of loop makes the most sense and what order you should access elements when using that type of loop.

When designing the signature of your method that will be called from your tests, consider whether or not you need to return the given list. Why or why not?

When writing your tests, consider what are good example lists to test the robustness of your method. When you think you have implemented a sufficient number of tests (remember to give them useful names that indicate the test's purpose), compare your tests with those around you to see what choices others made.

As usual, create a new class and associated test class in an appropriate package in your `MyJavaExercises` project.

Exercise 9.4: Practice Implementing `Comparable`

Time: 20 minutes

Format: Individual programming exercise

Change the `Employee` superclass header to declare that it implements the `Comparable<Employee>` interface. What compilation error do you get immediately after doing this? Use Eclipse to automatically fix this error. Then complete the generated `compareTo()` method to compare employee's names alphabetically.

Amend the utility class you created in Exercise 2.4.2, where it calculated aggregate pay, and add a static method that sorts a list of Employees by the default sort order.

When writing your tests, consider testing your `compareTo()` method separately from your utility sort method. Once you have confidence that comparing two employees works, how much extra work do you need to do to test that your utility sort method works?

Exercise 9.5: Practice Implementing `Comparator`

Time: 15 minutes

Format: Individual programming exercise

Make a separate class that implements the `Comparator<Employee>` interface and write the `compare()` method to compare employees primarily based on their monthly pay from largest to smallest (i.e., reverse of the default sorting order). If those values are equal, then compare their names alphabetically.

Add another sort method to your utility class. This one should accept a comparator as well as the list. Overload the method you added in the last exercise.

When writing your tests, consider testing your `compare()` method separately from your utility sort method.

Exercise 9.6: Anonymous Inner Class

Time: 10 minutes

Format: Individual programming exercise

Change your test code to create an anonymous inner class that implements the `Comparator<Employee>` interface instead of using the named class from the previous exercise. To focus on getting the syntax correct, you can just copy the implementation from your named class into this inline declaration of the `compare()` method.

Try defining the anonymous inner class both directly in the parameter list and saved as a field of your JUnit test class. Feel free to come up with a variety of ways to compare employees but make sure to test each different combination.

Chapter 10: Refactoring

Exercise 10.1: Refactoring Code

Time: 30 minutes

Format: Pair exercise

Spend a few minutes looking over the code given in the `Refactoring` project *alone* to form your *own* opinions about its design (both good and bad). Focus on reading the code critically, understanding its purpose, and identifying ways to improve its design through refactoring.

In addition to SOLID design principles and Code Smells, you should start to build a sense of the reader's expectations about code and what makes one piece of code better than another given a specific set of design goals. (An important goal of the code you write is to communicate your ideas to your teammates rather than just passing functionality tests.)

Then discuss your opinions with your partner using these questions to spark, but not limit your discussion:

- Where might comments be helpful within the code?
- Are there places where the code violates design principles we have discussed?
- How would you test this code?
- What additional methods or classes might be helpful?

Once you feel like you have reached some conclusions about the code, refactor it to improve its design. Create any new methods or classes you want but remember to focus on small incremental changes and developing a refactoring rhythm (test, change, test, change). Explore Eclipse's **Refactoring Menu** to see what it can automate for you (*Note:* this menu is context sensitive so different options appear depending on what code you have highlighted).

For each change, justify it in comments by explaining specifically how your group thinks it improves the code. Justifications should refer specifically to principles we have discussed rather than terms like "clearly/obviously", "good/sucks", or "like/hate".

This page intentionally left blank.

Chapter 11: Exceptions

Exercise 11.1: Catching Exceptions

Time: 20 minutes

Format: Individual programming exercise

Revisit the problem of maintaining a list of all non-negative values from Exercise 2.4.3. This was an optional exercise; if you did not complete the exercise, start with the solution project `PrimitiveListSolution`.

In this case, add a method that takes a `List<String>` as a parameter. It should be converted to a list of numbers and then passed to your original method to update the resulting `List<Integer>`. Such a list may have come from a database, a data file, or input from the user. However, such data cannot be trusted since there is no guarantee the strings all correctly represent numbers!

If an exception occurs, consider what the result should be. Discuss your decision with those around you. It may also be appropriate to throw an exception yourself and we will examine that possibility next.

Write a test before writing the try-catch block around your solution and see the error that is printed when the program abruptly terminates due to the unhandled exception. Read the error message printed and study the stack trace. Click each level of the stack to see where Eclipse takes you.

Use Eclipse's Debugger to step through the code to see exactly where it goes for each test to make sure you understand Java's execution flow with regard to exceptions.

Exercise 11.2: Throwing Exceptions

Time: 20 minutes

Format: Individual programming exercise

Working TDD, amend the code in your constructors and setters in all the classes in your `Employee` inheritance hierarchy to validate the values of the variables passed in. Throw an `IllegalArgumentException` if any input is an empty string, or a negative value. Throw a `NullPointerException` if any input is `null`.

Remember to write a test with an invalid input value that expects an exception, run it and see that it fails, then write the code to make it pass. Continue to run all your tests to check that your updates did not break any existing tests. Make sure to test both constructing an object and any public setters your classes have.

Also change any other methods that check input values and return special values. In some cases, these should now throw exceptions. In others they should return an appropriate value. For example, `calculatePayToDate` should throw an exception if called with a null date, but should return 0.0 if the date is earlier than the employee's start date.

Until you build a front end for your application (i.e., until there is a context for where the application will be run), these tests will likely be the only code that actually catches any thrown exceptions.

Exercise 11.3: Creating a Custom Exception

Time: 20 minutes

Format: Individual programming exercise

Modify one of the exceptional situations in your `Employee` class hierarchy to throw a custom exception that `extends RuntimeException` when an invalid input is provided instead of Java's standard `IllegalArgumentException`. The custom exception should have a descriptive error message that identifies which input type was not valid and why.

Update your JUnit tests to verify the custom exception is thrown as expected. If you specifically want to test the correct error message was used, do you have to modify your test code?

After the updated tests for your refactored code pass, experiment with making your custom exception extend `Exception` instead. What compilation error messages does this change create? Fix each one that appears automatically using Eclipse and study the resulting code it creates to make sure you understand why that change is being made. The needed changes should help explain why most coders prefer unchecked over checked exceptions!

(The solution to this exercise changes the exceptions in `Consultant`, but you are free to change any exception you like.)

This page intentionally left blank.

Chapter 12: API Design

Exercise 12.1: Creating a Service Class

Time: 30 minutes

Format: Individual programming exercise

Create a Service class in your `MyPayrollExercises` project. What package does it belong in? What about its tests?

Your class should have three methods:

- Sort a given `List<Employee>` using the `Employee`'s default comparison method
- Sort a given `List<Employee>` using a `Comparable<Employee>` object that is also passed in
- Return the total sum of the monthly pay of a given `List<Employee>`
- It may also have a method to calculate total pay to date of a `List<Employee>`

In fact, all of these methods should already exist in JUnit test code you implemented in previous exercises or in your utility class. You may even be able to use the utility class as the basis for the service. The point of Service classes is to have a clearly designed place in our project to put often-used but nonspecific functionality. It improves your design to have this kind of code in your business objects rather than in test code.

Before completing these refactoring steps, use your tests to help you think about how it makes sense to use this class within the rest of your project:

- Does it make sense to make instances of this class?
- Should these methods be instance or `static`?
- Why pass in a list to each of these methods instead of storing it as an instance variable?

These questions have significant design implications in terms of the usability, flexibility, and reliability of your code – especially in a multithreaded environment, such as the back end of a web service. After forming your own opinion, discuss your design ideas with other associates around you. You may not come to a consensus because each of you may have different design goals or assumptions about how the code will be used. That is okay as long you are able to articulate and justify your decisions.

Implement your Service class and refactor your test code.

This page intentionally left blank.

Chapter 13: Implementing Design Patterns

Exercise 13.1: Refactoring to a Pattern

Time: 30 minutes

Format: Individual programming exercise

1. Open the `DesignPatterns` project and examine the `FileEncrypter` class.
   a Notice that the constructor selects one of a pair of ciphers.
   b The overall class encrypts or decrypts a file according to that cipher.
   c The `Driver` class shows this in operation.
   d The unit tests cover the same functionality as `Driver`, but in a form that automatically checks the results for you. Run and review the tests.

2. Note the role of the classes in the `com.fidelity.cipher` package. These are the available ciphers for encrypting.
   a You do not need to understand how the ciphers work: you do not need to change any of them.
   b You should review the `CipherAlgorithm` interface to understand how it is used. All the ciphers implement this interface. What design pattern is at use here? The answer is on the next page: don't look ahead until you have thought about it for a while.

3. You have been assigned responsibility for extending the `FileEncrypter` so it can support all the cipher algorithms and any that are developed in the future.
   a Bearing the Open Close Principle in mind, you decide to use the Factory Method pattern.

4. Working TDD, convert `FileEncrypter` to use the Factory Method pattern to create the cipher algorithm.

5. Again, working TDD, add one of the additional cipher algorithms to your project to prove that `FileEncrypter` does not need to change.
   a If you needed to change `FileEncrypter`, try again until you can add an algorithm without making any changes to `FileEncrypter`.

Answer to question in 2b is on the next page.

Answer to question in 2b:

The `CipherAlgorithm` interface and the implementation classes are an example of the Strategy pattern. By using an interface, the code that uses the cipher is completely independent of whichever algorithm is chosen.