TASK

Refactoring

Visit our website

Copyright © 2021 HyperionDev. All rights reserved.

Introduction

WELCOME TO THE REFACTORING TASK!

Refactoring is important to ensure quality code. Refactoring is changing the structure of a program without changing its functionality. When refactoring your code, you take code that is working but make changes to ensure that your code adheres more to best-practice guidelines. This task introduces you to the concept of refactoring as well as the variety of tools provided by Eclipse that allows you to refactor your code quickly and easily.

INTRODUCTION TO REFACTORING

To improve the quality of your code, it is important to refactor it. Refactoring doesn't change the behaviour of the code and it doesn't remove errors. Rather, it has to do with restructuring code to make it more readable, maintainable and testable. Refactoring takes code that works and improves it.

Martin Fowler defines refactoring as "*a change made to the internal structure of the software to make it easier to understand and cheaper to modify without changing its observable behaviour*" (Fowler 1999). The word "refactoring" in modern programming grew out of Larry Constantine's original use of the word "factoring" in structured programming, which referred to decomposing a program into its constituent parts as much as possible.

Before we start refactoring code, take note of this important point: use version control. When you are changing code, you always run the risk of 'breaking' it. Therefore, always make sure that you use version control so that if a change doesn't have the desired effect, you can go back to a previously working version of your code. If you need a refresher on using Git for version control, refer back to the tasks that cover this topic.

REFACTORING BASICS

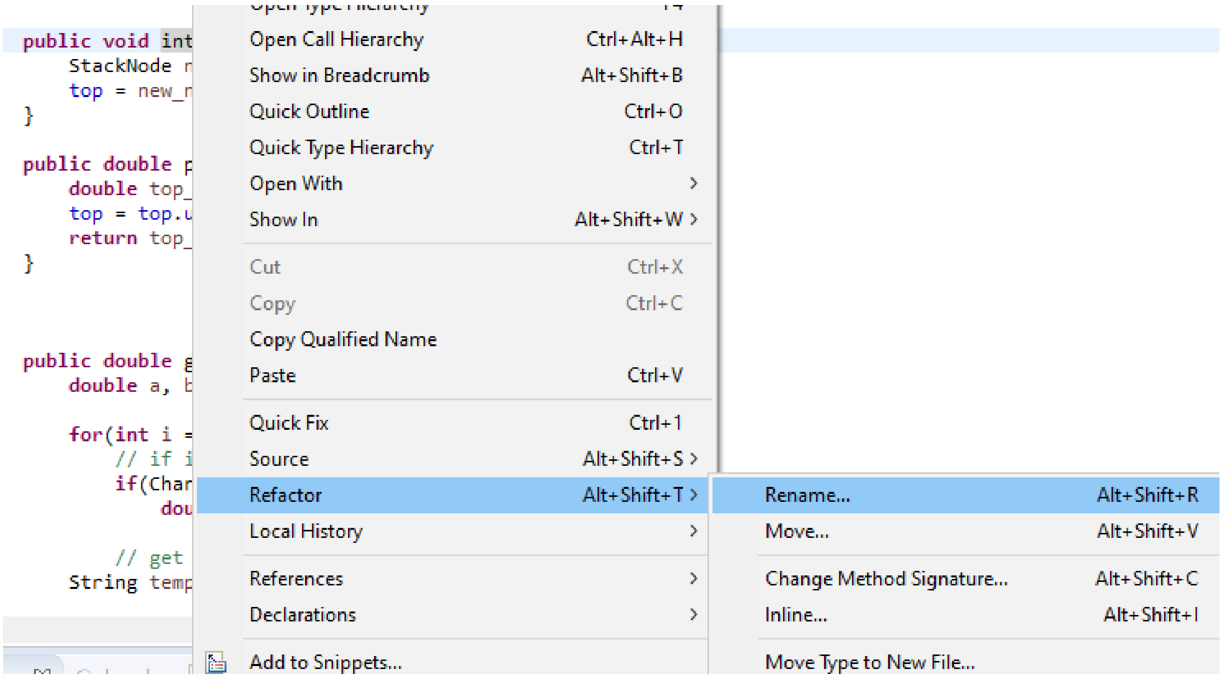
Refactoring includes doing some of the following:

1. **Use meaningful names**. Throughout your code, make sure that all variables, functions, modules, classes, etc. are given meaningful, descriptive names. To rename an element in Eclipse:

Copyright © 2021 HyperionDev. All rights reserved.

a. Click on the Java element in the Package Explorer view or select it in a Java source file.

b. Now, right-click and select *Refactor → Rename* from the menu.



```
public void int
    StackNode r
    top = new_r
}

public double p
    double top_
    top = top.u
    return top_
}


public double g
    double a, b

    for(int i =
        // if i
        if(Char
            dou

        // get
    String temp
```

c. In the dialogue box that should pop up, select a new name and choose whether Eclipse should also change references to the name. The exact fields that are displayed depend on the type of element that you select. For example, if you select a field that has getter and setter methods, you can also update the names of these methods to reflect the new field.

d. You can press *Preview* to see the changes that Eclipse proposes to make after you have specified everything necessary to perform the refactoring. This lets you reject or approve each change in each affected file, individually.

e. Otherwise, you can just press OK if you have confidence in Eclipse's ability to perform the change correctly.

2. Make sure **that your code is broken down into modules** (functions, classes, methods, etc). When refactoring it is important to change the structure of the code so that chunks of code are extracted to appropriate units such as functions or modules. There are many benefits to using modular code. Having modular code helps avoid duplicating code. When

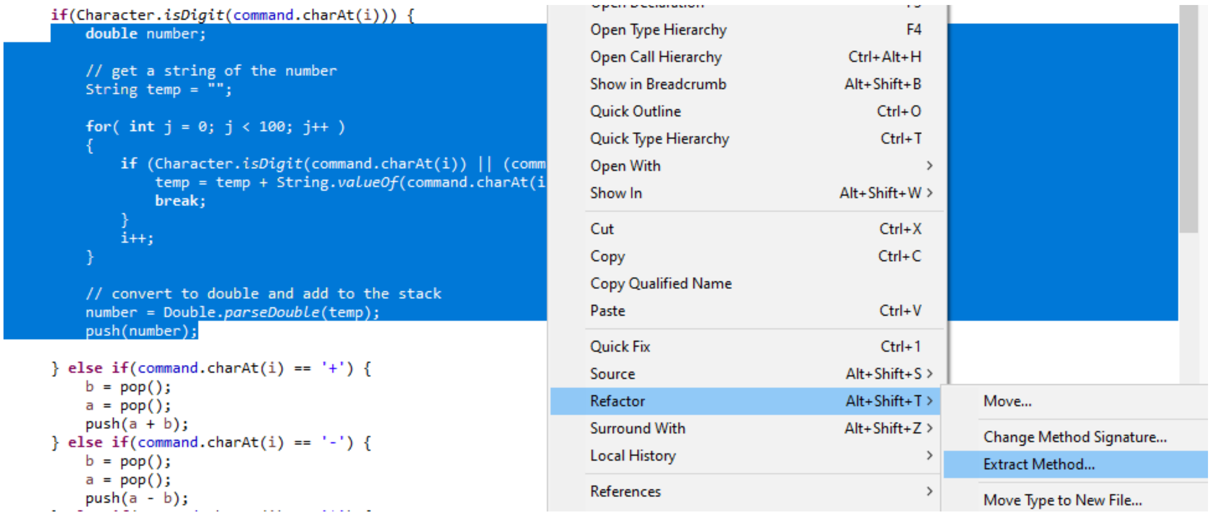
HyperionDev Copyright © 2021 HyperionDev. All rights reserved.

you have duplicated code, whenever you make changes in one place, you have to make parallel changes in another place. This sets you up to make parallel modifications. It also violates the “DRY principle”: Don’t Repeat Yourself.

Using modular code can also make your code easier to read, test and debug.

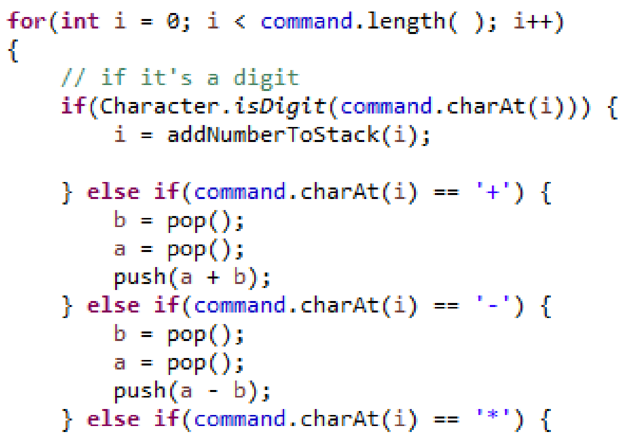
In Eclipse, it is easy to extract a unit of code into a method by:

- Highlighting the section of code.
- Right-clicking and selecting Refactor → Extract Method from the menu.



```
if(Character.isDigit(command.charAt(i))) {
    double number;

    // get a string of the number
    String temp = "";

    for( int j = 0; j < 100; j++ )
    {
        if (Character.isDigit(command.charAt(i)) || (comm
            temp = temp + String.valueOf(command.charAt(i
            break;
        }
        i++;
    }

    // convert to double and add to the stack
    number = Double.parseDouble(temp);
    push(number);

} else if(command.charAt(i) == '+') {
    b = pop();
    a = pop();
    push(a + b);
} else if(command.charAt(i) == '-') {
    b = pop();
    a = pop();
    push(a - b);
```

- You'll be prompted for a method name. A new method that contains the code you extracted will then be created. The call to the method will also be created in the appropriate place.



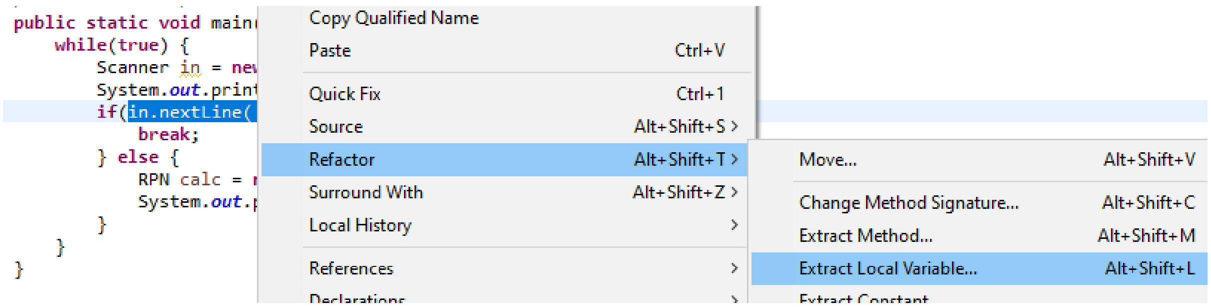
```
for(int i = 0; i < command.length( ); i++)
{
    // if it's a digit
    if(Character.isDigit(command.charAt(i))) {
        i = addNumberToStack(i);

    } else if(command.charAt(i) == '+') {
        b = pop();
        a = pop();
        push(a + b);
    } else if(command.charAt(i) == '-') {
        b = pop();
        a = pop();
        push(a - b);
    } else if(command.charAt(i) == '*') {
```

You can use a similar method to extract local variables or constants. For example, consider the code below:

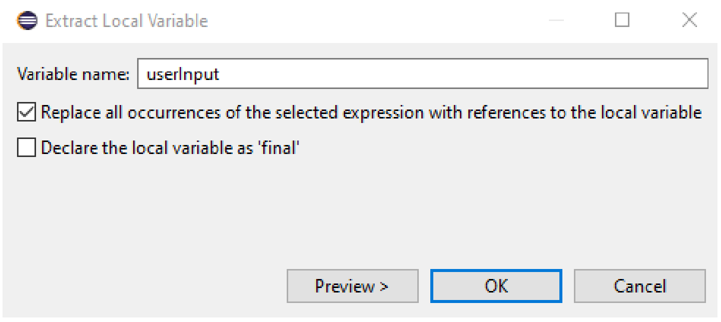
Copyright © 2021 HyperionDev. All rights reserved.

```
public static void main(String args[]) {
    while(true) {
        Scanner in = new Scanner(System.in);
        System.out.println("Enter RPN expression or \"quit\".");
        if(in.nextLine( ).equals("quit")) {
            break;
        } else {
            RPN calc = new RPN(in.nextLine( ));
            System.out.printf("Answer is %f\n", calc.calculateAnswer( ));
        }
    }
```

If you select **`in.nextLine()`**, right-click and then select *Refactor → Extract local variable…*



```
public static void main(
    while(true) {
        Scanner in = new
        System.out.print
        if(in.nextLine(
            break;
        } else {
            RPN calc =
            System.out.
        }
    }
}
```



Variable name: userInput

[✓] Replace all occurrences of the selected expression with references to the local variable

[ ] Declare the local variable as 'final'

The code is updated as shown below:

```
public static void main(String args[]) {
    while(true) {
        Scanner in = new Scanner(System.in);
        System.out.println("Enter RPN expression or \"quit\".");
        String userInput = in.nextLine( );
        if(userInput.equals("quit")) {
            break;
        } else {
            RPN calc = new RPN(userInput);
            System.out.printf("Answer is %f\n", calc.calculateAnswer( ));
        }
    }
}
```

There are several benefits to introducing variables this way. Firstly, it makes explicit what the code is doing by providing meaningful names to the expressions. Secondly, it makes it easier to debug the code, because we can

Copyright © 2021 HyperionDev. All rights reserved.

easily inspect the values that the expressions return. Finally, it can be more efficient in cases where multiple instances of an expression can be replaced with a single variable.

The Extract Constant refactoring is similar to Extract Local Variable, but you must select a static, constant expression, which the refactoring will convert to a static final constant. This is useful for removing hard-coded numbers and strings from your code.

3. **Be as brief as possible**. If you can do something with fewer lines of code without detracting too much from the readability of your code, do so.

4. **Make sure inputs to functions and output from functions are clear**. Make sure that your functions return real, meaningful output. Make input into your functions as clear as possible.

5. **Remove unnecessary code**. As we code we often create a lot of code that we can, and should, remove from the final version of our app.
   Delete:
   - Variables that are declared but never used.
   - Functions that are never called.
   - Code that you commented out because you thought you might need but you don't.
   - Statements used for debugging. Your code is likely to include things like `System.out` statements that you used to debug your system. For the final version of your app, get rid of these types of statements.

So far, we have considered the general principles for refactoring code. When creating object-oriented programs, there are some more specific guidelines for refactoring to consider. These are discussed next.

REFACTORING CLASSES

- **Ensure classes have high levels of cohesion**. Cohesion means that all the methods and properties in a class are closely related to each other and only perform one function. If a class takes ownership for an assortment of unrelated responsibilities, it should be broken up into multiple classes. Each class should only have one responsibility. All methods and data in that class should be strongly related to that responsibility.

HyperionDev Copyright © 2021 HyperionDev. All rights reserved.

- **Ensure programs with weak coupling**. Coupling describes how dependent classes are on each other. Weak coupling means that objects are not dependent on each other so that you can change one class without having to make changes to different classes. To enable weak coupling, we make sure that any variables/properties that should belong to an object are declared locally in that class. These properties can also be made private. Then other classes can't get to that data directly. If another class wants to change the properties of a class, they have to use the class's accessor or mutator methods to do this.

- **Remove `Set()` methods for fields that cannot be changed.** If a field is supposed to be set at object creation time and not changed afterwards, initialise that field in the object's constructor rather than providing a misleading `Set()` method.

- **Hide methods that are not intended to be used outside the class.** Do this by making these helper methods private.

- **Hide data members.** Public data members are always a bad idea as they blur the line between interface and implementation. Additionally, they inherently violate encapsulation and limit future flexibility. Consider hiding public data members behind access methods.

OTHER REFACTORING

- **Make sure that a loop isn't too long or too deeply nested.** The code inside a loop can be converted into methods. This helps to factor the code better and reduce the loop's complexity.

- **Use `break` or `return` instead of a loop control variable**. If you have a variable within a loop (like a boolean variable called 'done') that's used to control the loop exit, use `break` or `return` to exit the loop instead.

- **Return as soon as you know the answer instead of assigning a return value** within nested if-then-else statements. Code is often easiest to read and least error-prone if you exit a method as soon as you know the return value. The alternative to setting a return value and then unwinding your way through a lot of logic can be harder to follow.

- **Make sure that a method's parameter list doesn't have too many parameters.** Well-factored programs have many small, well-defined

Copyright © 2021 HyperionDev. All rights reserved.

methods that don't need large parameter lists. A long parameter list is a warning that the abstraction of the method interface has not been well thought out.

- Comments are important but **do not use comments as a crutch to explain bad code**.

- **Don't use global variables unnecessarily.** Rather encapsulate variables where it makes sense to do so. It's generally not considered good practice to expose the internal structure of your objects. That's why classes and subclasses have either private or protected fields, and public setter and getter methods to provide access. These methods can be generated automatically in two different ways in Eclipse.

  The first way to generate these methods is to use *Source → Generate Getter and Setter*. This will display a dialogue box with the proposed getter and setter methods for each field that does not already have one.

  However, this is not a refactoring because it does not update references to the fields to use the new methods and you'll need to do that yourself if necessary. This is a great time-saver, but it's best used when creating a class initially, or when adding new fields to a class because no other code references these fields yet, and there's, therefore, no other code to change.

  The second way to generate getter and setter methods is to select the field then select *Refactor → Encapsulate Field* from the menu.

  This method only generates getters and setters for a single field at a time, but in contrast to *Generate Getter and Setter*, it also changes references to the field into calls to the new methods. For example:

- Create a new **`Automobile`** class, as shown below:

```
public class Automobile extends Vehicle {
   public String make;
   public String model;
}
```

- Now, create a class that instantiates **`Automobile`** and accesses the make field directly.

HyperionDev Copyright © 2021 HyperionDev. All rights reserved.

```
public class AutomobileTest {
   public void race() {
      Automobilecar1 = new Automobile();
      car1.make= "Austin Healy";
      car1.model= "Sprite";
      // ...
   }
}
```

- Encapsulate the **`make`** field by highlighting the field name and selecting *Refactor → Encapsulate Field*.

- In the dialogue, enter names for the getter and setter methods; these are **`getMake()`** and **`setMake()`** by default. You can also choose whether methods that are in the same class as the field will continue to access the field directly or whether these references will be changed to use the access methods like all other classes.

- Press OK. The make field in the Automobile class will be private and will have **`getMake()`** and **`setMake()`** methods as shown below:

```
public class Automobile extends Vehicle {
   private String make;
   public String model;

   public void setMake(String make) {
      this.make = make;
   }

   public String getMake() {
      return make;
   }
}
```

- The **`AutomobileTest`** class will also be updated to use the new access methods.

```
public class AutomobileTest {
   public void race() {
      Automobilecar1 = new Automobile();
      car1.setMake("Austin Healy");
      car1.model= "Sprite";
      // ...
```

HyperionDev Copyright © 2021 HyperionDev. All rights reserved.

```
    }
}
```

- Replace numeric or string literals with a named constant. For example, replace the literal 3.14 with `PI`.

- Create and use null objects instead of testing for null values. Sometimes a null object will have generic behaviour or data associated with it, such as referring to a resident whose name is not known as "occupant." In this case, consider moving the responsibility for handling null values out of the client code and into the class — that is, have the Customer class define the unknown resident as "occupant" instead of having Customer's client code repeatedly test for whether the customer's name is known and substitute "occupant" if not.

BEST PRACTICE

There are many ways of getting a program to work — sort of, some of the time, fingers crossed. Often a key factor that separates 'cowboy coders' from professional software developers is the application of best practice when coding. A good resource when it comes to Java best practice is found **here**. One of the key goals of refactoring code is to bring your code more in line with best-practice guidelines.

Copyright © 2021 HyperionDev. All rights reserved.

Compulsory Task 1

Follow these steps:

- For this task, you are required to refactor the badly written program in the **Factory** folder.
- The factory pattern is a method that returns one of several possible classes that share a common superclass. It is particularly useful when you don't know beforehand what class you need and the decision can only be made at run time. The superclass that the Factory pattern uses is usually implemented as an interface. An interface is a special type of class that acts as a blueprint of classes to come. It is an abstract type that specifies the behaviour of future classes. An interface will contain method headings but the methods in an interface don't contain any logic. The logic for the methods specified in the interface is written in the classes that implement the interface. In essence, an interface shows a class what to do, and not how to do it.
- It is not vital for you to be familiar with design patterns for this task, but you can read more about them **here**.
- For this task be sure to:
  - Troubleshoot and debug the code first so that it runs correctly.
  - Fix the indentation and formatting of the code so that it adheres to the guidelines provided **here**.
  - Make sure that all the names of variables, classes, methods, etc. adhere to the guidelines provided **here**.
  - Refactor the code to improve the quality and readability of the code in other ways highlighted in this task.

Compulsory Task 2

Follow these steps:

- For this task, you are required to refactor the badly written program in the **Decorator** folder.
- The decorator pattern adds something extra to an object that already exists. It wraps an object in order to add functionality or features to the object. The purpose of the pattern is to make it easy to add or remove

HyperionDev Copyright © 2021 HyperionDev. All rights reserved.

functions if needed. This is done by wrapping a single object (not class) to add new behaviour. It has an attribute for that object and has all the same methods as the wrapped object.

- It is not vital for you to be familiar with design patterns for this task, but you can read more about them **here**.
- For this task be sure to:
    - Troubleshoot and debug the code first so that it runs correctly.
    - Fix the indentation and formatting of the code so that it adheres to the guidelines provided **here**.
    - Make sure that all the names of variables, classes, methods, etc. adhere to the guidelines provided **here**.
    - Refactor the code to improve the quality and readability of the code in other ways highlighted in this task.

Rate us

Share your thoughts

HyperionDev strives to provide internationally-excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved or think we've done a good job?

Click here to share your thoughts anonymously.

References

McConnell, S. C. (2004). Code Complete (2nd ed.). Redmond, Washington: Microsoft Press.

Copyright © 2021 HyperionDev. All rights reserved.