Task: Refactoring

www.hyperiondev.com

Introduction

Welcome to the Refactoring Task!

Refactoring is changing the structure of a program without changing its functionality. This tasks 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.

CONNECT WITH YOUR MENTOR

You don't have to take our courses alone! This course has been designed to be taken with an online mentor that marks your submitted code by hand and supports you to achieve your career goals on a daily basis.

To access this mentor support simply navigate to www.hyperiondev.com/support.

Introduction to Refactoring

Refactoring is the name given to a collection of small, independently learnable techniques for improving code that already exists. This means that this code is already written and working and we are not interested in changing what it does.

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 behavior"* (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.

Reasons to Refactor

Refactoring is required as oftentimes code degenerates under maintenance, and sometimes the code just wasn't very good in the first place. Either way, there are warning signs called "smells" that indicate where refactorings are needed. Below is a list of these smells:

- **Code is duplicated**
  When 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.

- **A method is too long**
  Methods which are longer than a screen are rarely needed. To improve a system you should aim to increase its modularity. This is achieved by increasing the number of well-defined, well-named methods that do one thing and do it well. If a method would be cleaner if part of it were made into a separate method, create that separate method.

- **A loop is too long or too deeply nested**
  The code inside of a loop can be being converted into methods. This helps to better factor the code and reduce the loop's complexity.

- **A class has poor cohesion**
  If a class takes ownership for an assortment of unrelated responsibilities, it should

be broken up into multiple classes, each of which has responsibility for a cohesive set of responsibilities.

- **A class interface does not provide a consistent level of abstraction**
  Class interfaces tend to change over time as a result of modifications that are made in the heat of the moment. These modifications oftentimes favour convenience rather than interface integrity. Eventually the class interface can become a maintenance nightmare that does little to improve the intellectual manageability of the program.

- **A parameter list has too many parameters**
  Well-factored programs have many small, well-defined 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.

- **Changes within a class tend to be compartmentalised**
  A class can sometimes have two or more distinct responsibilities. When that happens you find yourself changing either one part of the class or another part of the class. Very few changes actually affect both parts of the class. This is a sign that the class should be divided into multiple classes along the lines of the separate responsibilities.

- **Changes require parallel modifications to multiple classes**
  When you find yourself constantly making changes to the same set of classes, this suggests the code in those classes could be rearranged so that changes affect only one class.

- **Inheritance hierarchies have to be modified in parallel**
  If you find yourself making a subclass of one class every time you make a subclass of another class, this is a special kind of parallel modification and should be addressed.

- **Case statements have to be modified in parallel**
  If you find yourself making parallel modifications to similar case statements in multiple parts of the program, inheritance might be a better approach.

- **Related data items that are used together are not organised into classes**
  If you find yourself repeatedly manipulating the same set of data items, those manipulations might be able to be combined into a class of their own.

- **A method uses more features of another class than of its own class**
  This suggests that the method should be moved into the other class and then

invoked by its old class.

- **A primitive data type is overloaded**
  If your program uses a primitive data type like an integer to represent a common entity such as money, consider creating a simple Money class so that the compiler can perform type checking on Money variables, so that you can add safety checks on the values assigned to money.

- **A class doesn't do very much**
  If a class doesn't seem to be carrying its weight, you could assign all of that class's responsibilities to other classes and eliminate the class altogether.

- **A chain of methods passes tramp data**
  Passing data to one method just so that method can pass it to another method is called "tramp data". You should ask yourself whether passing the specific data in question is consistent with the abstraction presented by each of the method interfaces. If the abstraction for each method is OK, passing the data is OK. If not, find some way to make each method's interface more consistent.

- **A middleman object isn't doing anything**
  If most of the code in a class is just passing off calls to methods in other classes, consider whether you should eliminate the middleman and call those other classes directly.

- **One class is overly intimate with another**
  Anytime you see one class that knows more about another class than it should, this includes derived classes knowing too much about their parents, err on the side of stronger encapsulation rather than weaker.

- **A method has a poor name**
  Change the name of the method where it's defined, change the name in all places it's called, and then recompile.

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

- **A subclass uses only a small percentage of its parents' methods**
  This typically indicates that that subclass has been created because a parent class happened to contain the methods it needed, and not because the subclass is logically a descendent of the superclass. You can achieve better encapsulation by

changing the relationship from an is-a relationship to a has-a relationship. Convert the superclass to member data of the former subclass, and expose only the methods in the former subclass that are really needed.

- **Comments are used to explain difficult code**
  Comments are important but, they should not be used as a crutch to explain bad code.

- **Global variables are used**
  Take time to reexamine global variables when you revisit a section of code that uses them. Since you're less familiar with the code than when you first wrote it, you might now find the global variables confusing and you might be able to develop a cleaner approach. You might also have a better sense of how to isolate global variables in access methods and a better sense of the pain caused by not doing so.

- **A method uses setup code before a method call or takedown code after a method call**

  The code like the example below should be a warning:

```
//This setuo code is a warning
WithdrawalTransaction withdrawal = new  WithdrawalTransaction();
withdrawal.SetCustomerId( customerId );
withdrawal.SetBalance( balance );
withdrawal.SetWithdrawalAmount( withdrawalAmount );
withdrawal.SetWithdrawalDate( withdrawalDate );

ProcessWithdrawal( withdrawal );

//This takedown code is also a warning
customerId = withdrawal.GetCustomerId();
balance = withdrawal.GetBalance();
withdrawalAmount = withdrawal.GetWithdrawalAmount();
withdrawalDate = withdrawal.GetWithdrawalDate();
```

  Anytime you see code that sets up for a call to a method or takes down after a call to a method, ask whether the method interface is presenting the right abstraction. In this case, perhaps the parameter list of ProcessWithdrawal should be modified to support code like this:

```
//Example of a method That Doesn't Require Setup or Takedown Code
```

```
ProcessWithdrawal( customerId, balance, withdrawalAmount, withdrawalDate
);
```

The converse of this example presents a similar problem. If you find yourself usually having a WithdrawalTransaction object but needing to pass several of its values to a method, you should also consider refactoring the ProcessWithdrawal interface so that it requires the WithdrawalTransaction object rather than its individual fields.

- **A program contains code that seems like it might be needed someday**
  "Designing ahead" can cause numerous problems. The best way to prepare for future requirements is not to write speculative code; it's to make the currently required code as clear and straightforward as possible so that future programmers will know what it does and does not do and will make their changes accordingly

Specific Refactorings

Below is a List of refactorings. This list is not exhaustive, but it includes some of the most useful refactorings.

Data-Level Refactorings

These refactorings improve the use of variables and other kinds of data.

- **Replace a magic number with a named constant** Replace numeric or string literals with a named constant. For example replace the literal 3.14 with PI.

- **Rename a variable with a clearer or more informative name** If a variable's name isn't clear, give it a better name. The same applies to renaming constants, classes, and methods.

- **Move an expression inline** Replace an intermediate variable that was assigned the result of an expression with the expression itself.

- **Replace an expression with a method** This is usually so that the expression isn't duplicated in the code.

- **Introduce an intermediate variable** Assign an expression to an intermediate variable whose name summarises the purpose of the expression.

- **Convert a multi-use variable to multiple single-use variables** If a variable is used for more than one purpose, create separate variables
for each usage, each of which has a more specific name.

- **Use a local variable for local purposes rather than a parameter** If an input-only method parameter is being used as a local variable, create a local variable and use that instead.

- **Convert a data primitive to a class** If a data primitive needs additional behavior or additional data, convert the data to an object and add the behavior you need.

- **Convert a set of type codes to a class or an enumeration** In older programs, it's common to see associations like the following:

```
const int SCREEN = 0;
const int PRINTER = 1;
const int FILE = 2;
```

  Rather than defining standalone constants, create a class so that you can receive the benefits of stricter type checking and set yourself up to provide richer semantics for OutputType if you ever need to. Creating an enumeration is sometimes a good alternative to creating a class.

- **Convert a set of type codes to a class with subclasses** If the different elements associated with different types have different behavior, consider creating a base class for the type with subclasses for each type code. For the OutputType base class, you might create subclasses like Screen, Printer, and File.

- **Change an array to an object** If you're using an array in which different elements are different types, create an object that has a field for each former element of the array.

- **Encapsulate a collection** If a class returns a collection, having multiple instances of the collection floating around can create synchronization difficulties. Consider having the class return a read-only collection, and provide methods to add and remove elements from the collection.

- **Replace a traditional record with a data class** Create a class that contains the members of the record. Creating a class allows you to centralize error checking, persistence, and other operations that concern the record.

Statement-Level Refactorings

These refactorings improve the use of individual statements.

- **Decompose a boolean expression** Simplify a boolean expression by introducing well named intermediate variables that help document the meaning of the expression.

- **Move a complex boolean expression into a well-named boolean function** If the expression is complicated enough, this refactoring can improve readability. If the expression is used more than once, it eliminates the need for parallel modifications and reduces the chance of error in using the expression.

- **Consolidate fragments that are duplicated within different parts of a conditional** If you have the same lines of code repeated at the end of an else block that you have at the end of the if block, move those lines of code so that they occur after the entire if-then-else block.

- **Use break or return instead of a loop control variable** If you have a variable within a loop like done that's used to control the loop, 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 of setting a return value and then unwinding your way through a lot of logic can be harder to follow.

- **Replace conditionals (especially repeated case statements) with polymorphism** Much of the logic that used to be contained in case statements in structured programs can instead be baked into the inheritance hierarchy and accomplished through polymorphic method calls.

- **Create and use null objects instead of testing for null values** Sometimes a null object will have generic behavior 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.

Method-Level Refactorings

These refactorings improve code at the individual-method level.

- **Extract method/extract method** Remove inline code from one method, and turn it into its own method.

- **Move a method's code inline** Take code from a method whose body is simple and self-explanatory, and move that method's code inline where it is used.

- **Convert a long method to a class** If a method is too long, sometimes turning it into a class and then further factoring the former method into multiple methods will improve readability.

- **Substitute a simple algorithm for a complex algorithm** Replace a complicated algorithm with a simpler algorithm.

- **Add a parameter** If a method needs more information from its caller, add a parameter so that that information can be provided.

- **Remove a parameter** If a method no longer uses a parameter, remove it.

- **Separate query operations from modification operations** Normally, query operations don't change an object's state. If an operation like GetTotals() changes an object's state, separate the query functionality from the state-changing functionality and provide two separate methods.

- **Combine similar methods by parameterising them** Two similar methods might differ only with respect to a constant value that's used within the method. Combine the methods into one method, and pass in the value to be used as a parameter.

- **Separate methods whose behavior depends on parameters passed in** If a method executes different code depending on the value of an input parameter, consider breaking the method into separate methods that can be called separately, without passing in that particular input parameter.

- **Pass a whole object rather than specific fields** If you find yourself passing several values from the same object into a method, consider changing the method's interface so that it takes the whole object instead.

- **Pass specific fields rather than a whole object** If you find yourself creating an object just so that you can pass it to a method, consider modifying the method so that it takes specific fields rather than a whole object.

- **Encapsulate downcasting** If a method returns an object, it normally should return the most specific type of object it knows about. This is particularly applicable to methods that return iterators, collections, elements of collections, and so on.

Class Implementation Refactorings

These refactorings improve code at the class level.

- **Move a method to another class** Create a new method in the target class, and move the body of the method from the source class into the target class. You can then call the new method from the old method.

- **Convert one class to two** If a class has two or more distinct areas of responsibility, break the class into multiple classes, each of which has a clearly defined responsibility.

- **Eliminate a class** If a class isn't doing much, move its code into other classes that are more cohesive and eliminate the class.

- **Hide a delegate** Sometimes Class A calls Class B and Class C, when really Class A should call only Class B and Class B should call Class C. Ask yourself what the right abstraction is for A's interaction with B. If B should be responsible for calling C, have B call C.

- **Remove a middleman** If Class A calls Class B and Class B calls Class C, sometimes it works better to have Class A call Class C directly. The question of whether you should delegate to Class B depends on what will best maintain the integrity of Class B's interface.

- **Replace inheritance with delegation** If a class needs to use another class but wants more control over its interface, make the superclass a field of the former subclass and then expose a set of methods that will provide a cohesive abstraction.

- **Replace delegation with inheritance** If a class exposes every public method of a delegate class (member class), inherit from the delegate class instead of just using the class.

- **Introduce a foreign method** If a class needs an additional method and you can't modify the class to provide it, you can create a new method within the client class that provides that functionality.

- **Introduce an extension class** If a class needs several additional methods and you can't modify the class, you can create a new class that combines the unmodifiable class's functionality with the additional functionality. You can do that either by subclassing the original class and adding new methods or by wrapping the class and exposing the methods you need.

- **Encapsulate an exposed member variable** If member data is public, change the member data to private and expose the member data's value through a method instead.

- **Remove Set() methods for fields that cannot be changed** If a field is supposed to be set at object creation time and not changed afterward, initialize 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** If the class interface would be more coherent without a method, hide the method.

- **Encapsulate unused methods** If you find yourself routinely using only a portion of a class's interface, create a new interface to the class that exposes only those necessary methods. Be sure that the new interface provides a coherent abstraction.

- **Collapse a superclass and subclass if their implementations are very similar** If the subclass doesn't provide much specialization, combine it into its superclass.

System-Level Refactorings

These refactorings improve code at the whole-system level.

- **Create a definitive reference source for data you can't control** Sometimes you have data maintained by the system that you can't conveniently or consistently access from other objects that need to know about that data. A common example is data maintained in a GUI control. In such a case, you can create a class that mirrors the data in the GUI control, and then have both the GUI control and the other code treat that class as the definitive source of that data.

- **Change unidirectional class association to bidirectional class association** If you have two classes that need to use each other's features but only one class can know about the other class, change the classes so that they both know about each other.

- **Change bidirectional class association to unidirectional class association** If you have two classes that know about each other's features but only one class that really needs to know about the other, change the classes so that one knows about the other but not vice versa.

- **Provide a factory method rather than a simple constructor** Use a factory method when you need to create objects based on a type code or when you want to work with reference objects rather than value objects.

- **Replace error codes with exceptions or vice versa** Depending on your error-handling strategy, make sure the code is using the standard approach.

Refactoring in Eclipse

Eclipse provides a variety of tools that allows you to refactor your code quickly and easily. Eclipse's refactoring tools can be grouped into three broad categories:

1. Changing the name and physical organization of code, including renaming fields, variables, classes, and interfaces, and moving packages and classes.

2. Changing the logical organisation of code at the class level. This includes turning anonymous classes into nested classes, turning nested classes into top-level classes, creating interfaces from concrete classes, and moving methods or fields from a class to a subclass or superclass.

3. Changing the code within a class. This includes turning local variables into class fields, turning selected code in a method into a separate method, and generating getter and setter methods for fields.

Physical Reorganization and Renaming

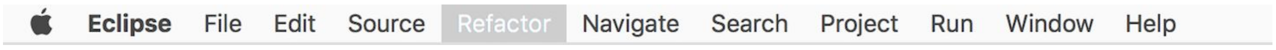
You do not need a special tool to rename or move files around in the file system, however, doing so with Java source files may require that you edit many files to update import or package statements. You can also rename classes, methods and variables using your text editor's search and replace functionality, however you need to be extremely careful when doing this. This is as different classes may have methods or variables with similar names and it can be tedious to go through all the files in a project making the sure that every instance is correctly identified and changed.

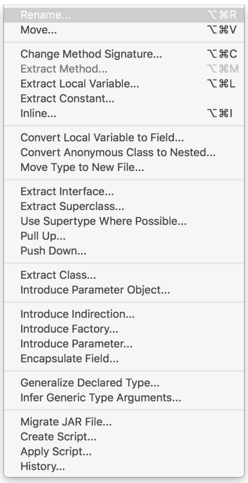
Eclipse's Rename and Move allow you to make these changes easily throughout the entire project without any intervention by the user. This is because Eclipse understands the code semantically and is able to identify references to a specific method, variable, or class names.

To rename an element:

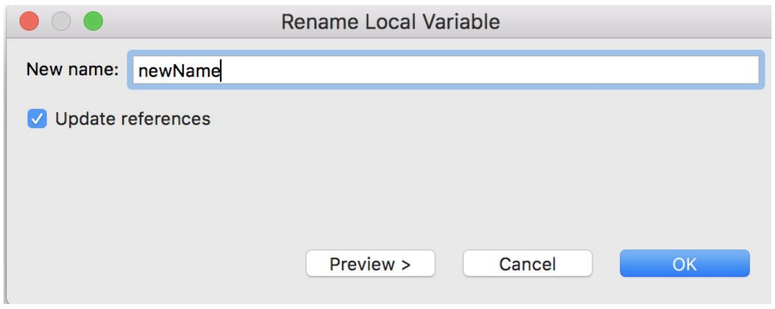
1. Click on the Java element in the Package Explorer view or select it in a Java source file

2. Now, select Refactor -> Rename from the menu





3. In the dialog box that should pop up, select the 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. The image below shows the fields displayed in the dialog box when renaming a local variable.


Rename Local Variable

New name: newName

☑ Update references

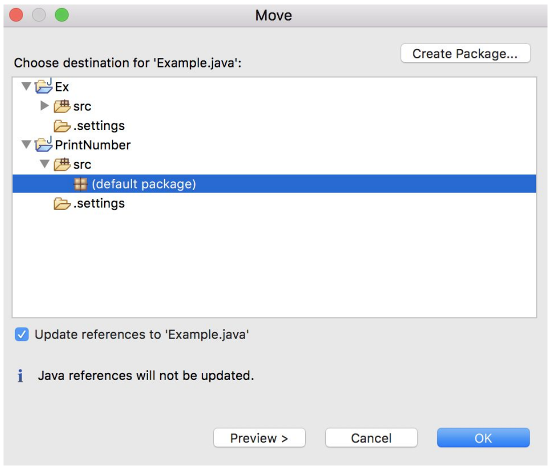
4. You can press Preview to see the changes that Eclipse proposes to make after you have specified everything necessary to perform the refactoring. This lets you reject or approve each change in each affected file, individually.

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

To move an element:

1. Move works very much like Rename. Click on the Java element (usually a class), and select select Refactor -> Move from the menu.

2. Now, specify its new location, and specify whether references should also be updated.


Move

Choose destination for 'Example.java':

```
▼ Ex
    ▶ src
      .settings
▼ PrintNumber
    ▼ src
        (default package)
      .settings
```

☑ Update references to 'Example.java'

Java references will not be updated.

3. Then choose Preview to examine the changes or OK to carry out the refactoring immediately.

Changing Code Within a Class

A large number of refactorings reorganise code within a class. These allow you to introduce (or remove) intermediate variables, create a new method from a portion of an old one, and create getter and setter methods for a field, among other things.

Extracting and Inlining

Many refactorings begin with the word Extract such as, Extract Method, Extract Local Variable, and Extract Constants.

The Extract Method creates a new method from code you've selected. For example, take the main() method in the code below that evaluates command-line options. If it finds any command-line options that begin with -D, it stores them as name-value pairs in a Properties object.

```
import java.util.Properties;
import java.util.StringTokenizer;
public class StartApp
{
   public static void main(String[] args)
   {
      Properties props = new Properties();
      for (int i= 0; i < args.length; i++)
      {
         if(args[i].startsWith("-D"))
```

```
            {
              String s = args[i].substring(2);
              StringTokenizer st = new StringTokenizer(s, "=");
               if(st.countTokens() == 2)
               {
                 props.setProperty(st.nextToken(), st.nextToken());
               }
            }
        }
        //continue...
    }
}
```

There are two main cases where you might want to take some code out of a method and put it in another method. The first case is if the method is too long and does two or more logically distinct operations. The second case is if there is a logically distinct section of code that can be reused by other methods. For example, if you sometimes find yourself repeating several lines of code in several different methods.

Assume that there is another place where you need to parse name-value pairs and add them to a Properties object, you could extract the section of code that includes the StringTokenizer declaration and following if statement. To do this:

1. Highlight the section of code.

2. Select Refactor -> Extract Method from the menu.

3. You'll be prompted for a method name; enter addProperty, and then verify that the method has two parameters, Properties prop and Strings. The code below shows the class after Eclipse extracts the method addProp().

```
import java.util.Properties;
import java.util.StringTokenizer;
public class Extract
{
   public static void main(String[] args)
   {
      Properties props = new Properties();
      for (int i = 0; i < args.length; i++)
      {
         if (args[i].startsWith("-D"))
         {
            String s = args[i].substring(2);
            addProp(props, s);
            }
        }
    }
    private static void addProp(Properties props, String s)
    {
        StringTokenizer st = new StringTokenizer(s, "=");
        if (st.countTokens() == 2)
        {
            props.setProperty(st.nextToken(), st.nextToken());
        }
    }
}
```

The Extract Local Variable refactoring takes an expression that is being used directly and assigns it to a local variable first. This variable is then used where the expression used to be. For example:

1. In the addProp() method above, highlight the first call to st.nextToken()
2. Then select Refactor -> Extract Local Variable from the menu.
3. You will be prompted to provide a variable. Call the new local variable key. There is an option to replace all occurrences of the selected expression with references to the new variable. This is often appropriate, but not in this case of the nextToken() method, which returns a different value each time it is called. Therefore, make sure this option is not selected.
4. Now, repeat this refactoring for the second call to st.nextToken(), this time calling the new local variable value. The code after these two refactorings is shown below.

```
private static void addProp(Properties props, String s)
    {
       StringTokenizer st = new StringTokenizer(s, "=");
        if(st.countTokens() == 2)
        {
            String key = st.nextToken();
            String value = st.nextToken();
          props.setProperty(key, value);
        }
    }
```

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 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. For example, in the code above we used -D" for the command line option defining a name-value pair.

1. Highlight -D" in the code.
2. Select Refactor -> Extract Constant from the menu.
3. Enter DEFINE as the name of the constant. This refactoring will change the code as shown below.

```
public class Extract
{
   private static final String DEFINE = "-D";
   public static void main(String[] args)
   {
      Properties props = new Properties();
      for (int i = 0; i < args.length; i++)
      {
         if (args[i].startsWith(DEFINE))
         {
            String s = args[i].substring(2);
            addProp(props, s);
         }
      }
   }
   // ...
```

For each Extract refactoring, there is a corresponding Inline refactoring that performs the reverse operation. For example:

1. Highlight the variable s in the code above.
2. Select Refactor -> Inline... from the menu.
3. Now, press OK. Eclipse uses the expression args[i].substring(2) directly in the call to addProp() as follows:

```
if(args[i].startsWith(DEFINE))
 {
    addProp(props,args[i].substring(2));
 }
```

This is slightly more efficient than using a temporary variable and making the code shorter, makes it either easier to read or more cryptic, depending on your point of view.

You can also highlight a method name or a static final constant in the same way that you can replace a variable with an inline expression. Select Refactor -> Inline... from the menu, and Eclipse will replace method calls with the method code, or references to the constant with the constants value, respectively.

Encapsulating Fields

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.

The first way to generate these methods is to use Source -> Generate Getter and Setter. This will display a dialog 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 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:

1. create a new Automobile class, as shown below:

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

2. Now, create a class that instantiates Automobile and accesses the make field directly

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

3. Encapsulate the make field by highlighting the field name and selecting Refactor -> Encapsulate Field.

4. In the dialog, 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.

5. 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";
      // ...
   }
}
```

Change Method Signature

The Change Method Signature refactoring is one of the most difficult to use. This refactoring changes the parameters, visibility, and return type of a method. However, the effect these changes have on the method or on the code that calls the method is not obvious. If the changes cause problems in the method being refactored the refactoring operations will flag these. You have the option to accept the refactoring anyway and correct the problems afterwards, or cancel the refactoring. If the refactoring causes problems in other methods, these are ignored and you must fix them yourself after the refactoring.

For example, consider the following class and method:

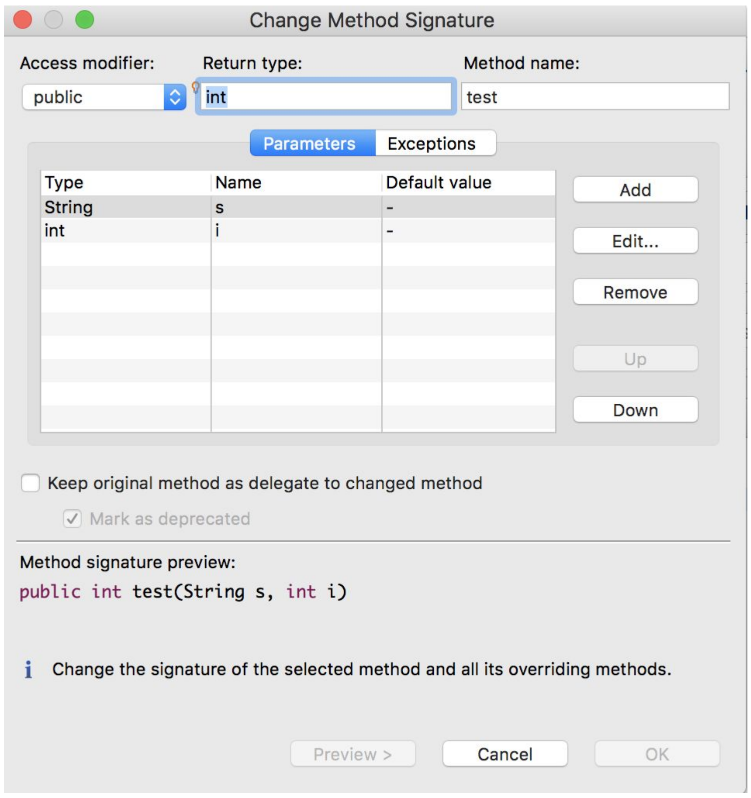
```
public class MethodSigExample
{
   public int test(String s, int i)
   {
      int x = i + s.length();
      return x;
   }
}
```

The method test() in the class above is called by a method in another class, as shown below:

```
public void callTest()
   {
     MethodSigExample eg = new MethodSigExample();
     int r = eg.test("hello", 10);
   }
```

1. Highlight test in the first class and select Refactor -> Change Method Signature.

2. The following dialog box will appear



Access modifier: public

Return type: int

Method name: test

Type	Name	Default value
String	s	-
int	i	-

☐ Keep original method as delegate to changed method

☑ Mark as deprecated

Method signature preview:

```
public int test(String s, int i)
```

i Change the signature of the selected method and all its overriding methods.

The first option is to change the method's visibility. In this example, changing it to protected or private would prevent the callTest() method in the second class from accessing. Eclipse will not flag this error while performing the refactoring; it's up to you to select an appropriate value.

The next option is to change the return type. For example, changing the return type to float isn't flagged as an error because an int in the test() method's return statement is automatically promoted to float. This will, however, result in a problem in the callTest() in the second class, because a float cannot be converted to int. You will need to either cast the return value returned by test() to int or change the type of r in callTest() to float.

Similar considerations apply if we change the type of the first parameter from String to int. This will be flagged during refactoring because it causes a problem in the method being refactored: int does not have a length() method.

In cases where this refactoring results in an error, whether flagged or not, you can continue by simply correcting the errors on a case-by-case basis. Another approach is to preempt errors. If you want to remove the parameter i, because it's not needed, you could start by removing references to it in the method being refactored. Removing the parameter will then go more smoothly.

Default Value option is only used when a parameter is being added to the method signature. It is used to provide a value when the parameter is added to callers. For example, if we add a parameter of type String, with a name n, and a default value of world, the call to test() in the callTest() method will be changed as follows:

```
public void callTest()
{
   MethodSigExample eg = new MethodSigExample();
   int r = eg.test("hello", 10, "world");
}
```

Compulsory Task

Follow these steps:

- For this task you are required to refactor the badly written program RPN.java. This program is a Reverse-Polish Notation calculator which uses a stack.
- You should also fix the indentation of the code.

SHARE YOUR THOUGHTS WITH US

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