1 Refactor | Introduce Variable

1.1 Purpose

This refactoring allows you to take an expression and replace it with a newly introduced local variable

1.2 Test Checks

1.2.1 Introduce a variable for a simple expression

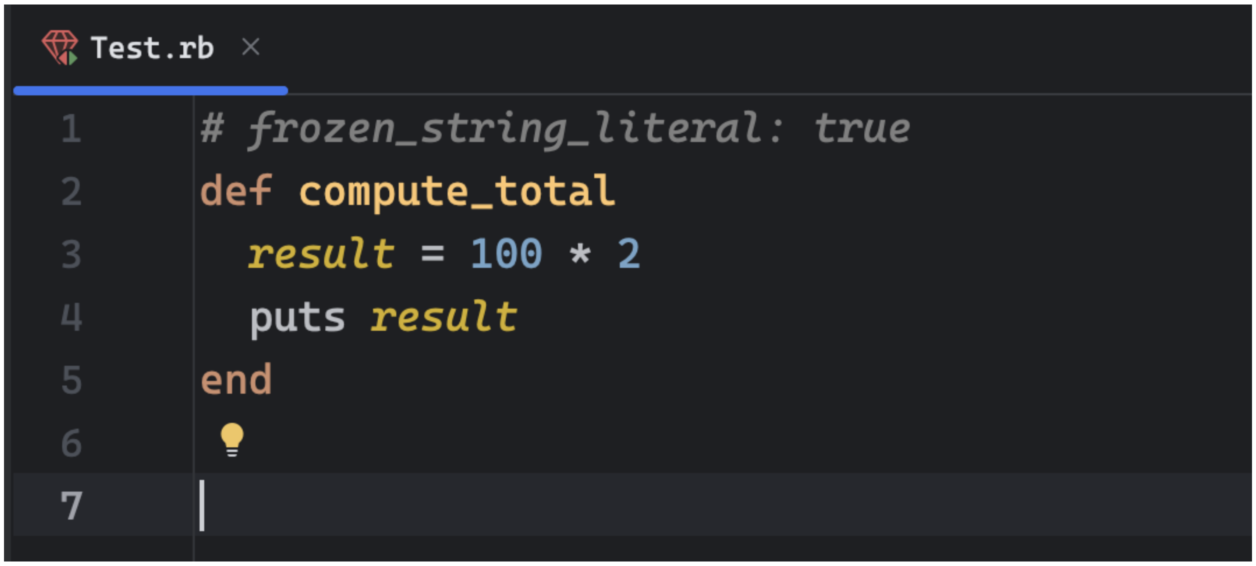
1.2.1.1 Steps

1. Open a Ruby file and write a method with a simple expression, for example:
   def compute_total
     puts 100 * 2
   end
2. Select an expression 100 * 2
3. Use Refactor -> Introduce Variable
4. Choose a name for a new variable
5. Confirm the refactoring

1.2.1.2 Expected Result

def compute_total
  result = 100 * 2
  puts *result*
end

1.2.1.3 Actual Result



Test.rb

```
# frozen_string_literal: true
def compute_total
  result = 100 * 2
  puts result
end

```

Figure 1 Actual Result for check #1

The actual result is the same as expected.

1.2.2 Introduce Variable for a complex expression

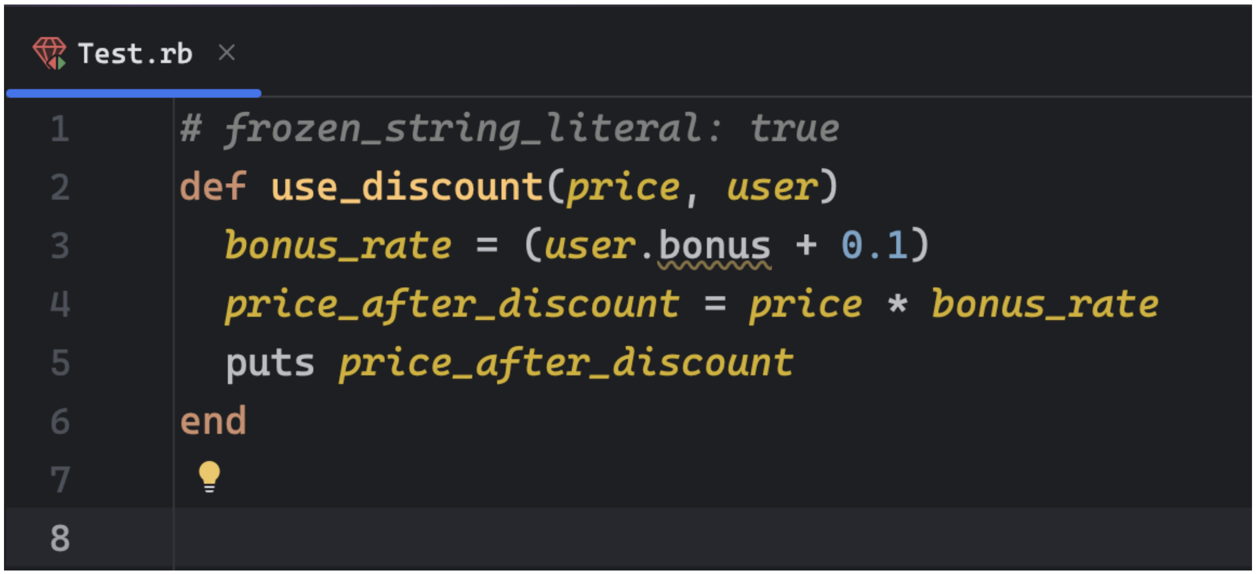
1.2.2.1 Steps

1. Write a method that includes more complex expression, for example:
   ```
   def use_discount(price, user)
     price_after_discount = price * (user.bonus + 0.1)
     puts price_after_discount
   end
   ```
2. Select (user.bonus + 0.1)
3. Use Refactor -> Introduce Variable
4. Choose a name for a new variable
5. Confirm the refactoring

1.2.2.2 Expected Result

```
def user_discount(price, user)
  bonus_rate = user.bonus + 0.1
  price_after_discount = price * bonus_rate
  puts price_after_discount
end
```

1.2.2.3 Actual Result



Test.rb

```
# frozen_string_literal: true
def use_discount(price, user)
  bonus_rate = (user.bonus + 0.1)
  price_after_discount = price * bonus_rate
  puts price_after_discount
end
```

Figure 2 Actual Result for check #2

The actual result is the same as expected.

1.2.3 Introduce Variable when expression is used multiple times

1.2.3.1 Steps

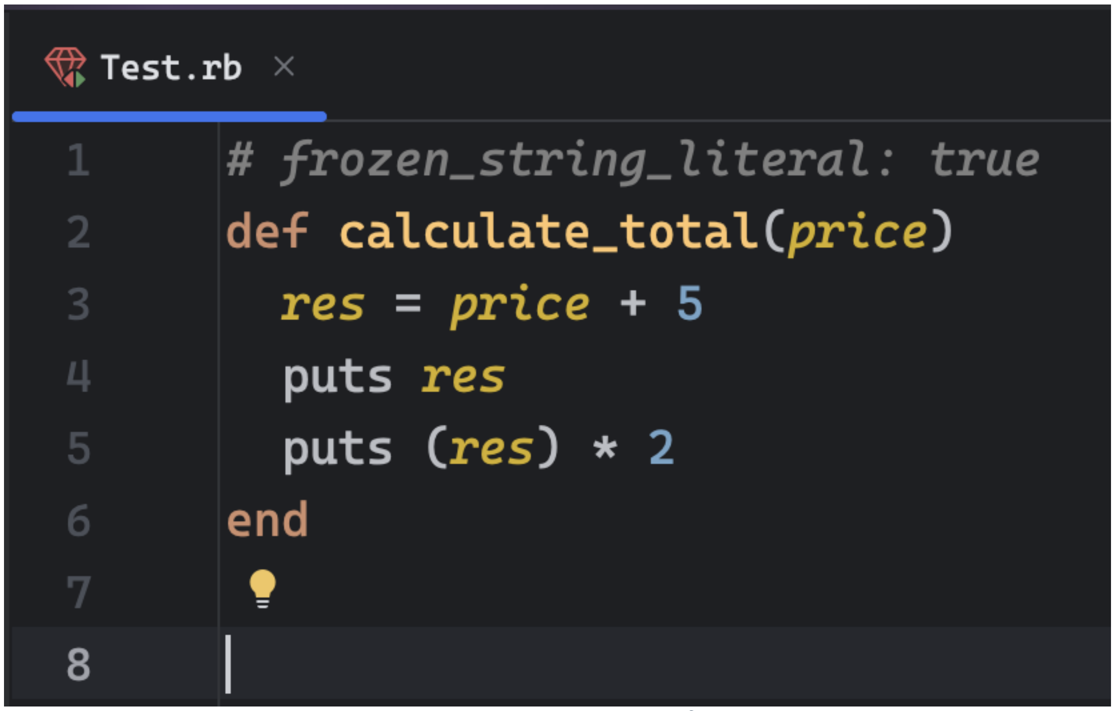
1. Write a method expression that is used multiple times, for example:
   ```
   def calculate_total(price)
     puts price + 5
     puts (price + 5) * 2
   end
   ```
2. Select repeating expression (price + 5)

3. Use Refactor -> Introduce Variable
4. Choose “Replace all x occurrences”
5. Choose a name for a new variable
6. Confirm the refactoring

1.2.3.2 Expected Result

```
def calculate_total(price)
  res = price + 5
  puts res
  puts (res) * 2
end
```

1.2.3.3 Actual Result


Test.rb

```ruby
# frozen_string_literal: true
def calculate_total(price)
  res = price + 5
  puts res
  puts (res) * 2
end

```

Figure 3 Actual Result for check #3

The actual result is the same as expected.

1.2.4 Check for name collisions

1.2.4.1 Steps

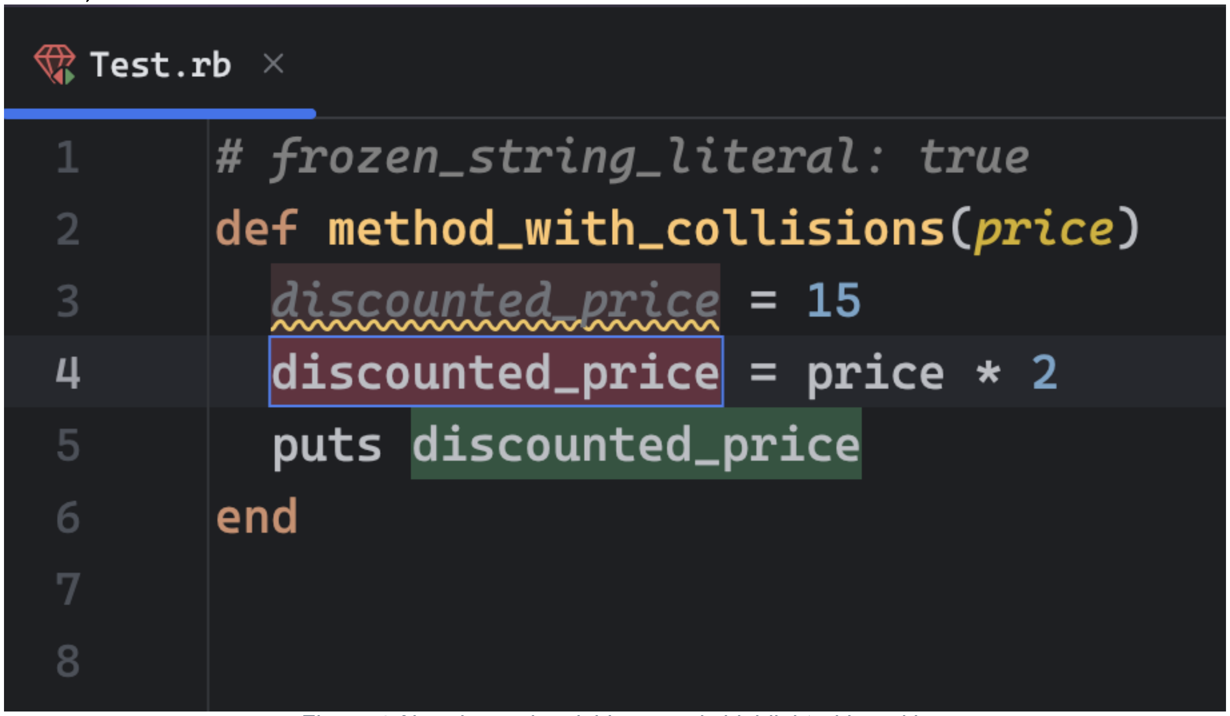
1. Write a method where name collision might occur, for example:
   ```
   def method_with_collisions(price)
     discounted_price = 15
     puts price * 2
   end
   ```
2. Select price * 2
3. Use Refactor -> Introduce Variable
4. Try to use the already existing variable name in this scope (discounted_price)
5. Observe refactoring behavior

1.2.4.2 Expected Result

RubyMine should warn you about name collision or/and automatically suggest a different variable name. The refactored code should not overwrite or break existing variable.

1.2.4.3 Actual Result

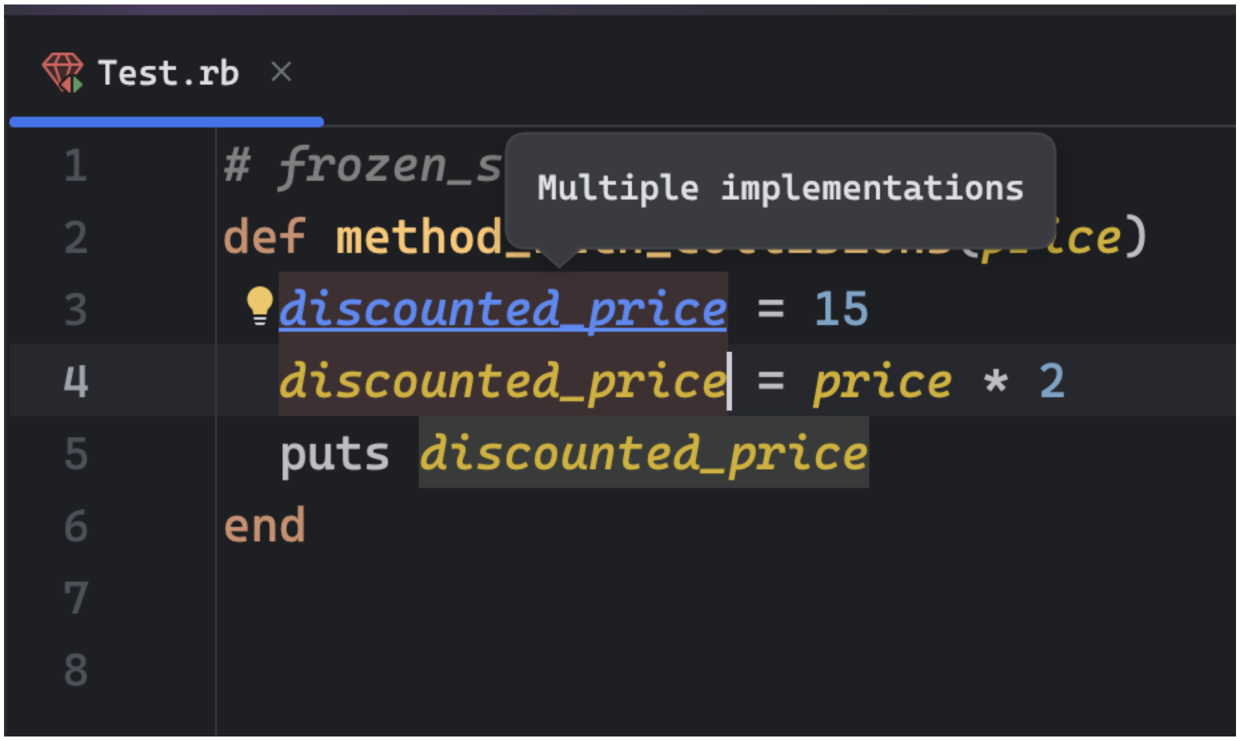
Actual Result differs from an expected. RubyMine warns user about variable name collision by highlighting the variable that new introduced one may share the same name, as shown at the screenshot below:



```ruby
# frozen_string_literal: true
def method_with_collisions(price)
  discounted_price = 15
  discounted_price = price * 2
  puts discounted_price
end
```

Figure 4 Already used variable name is highlighted in red here

However, IDE does not prevent user from using the same name, neither suggest one automatically. This may result in user missing name collision and introducing new variable with the same name.



```
# frozen_s
def method_                    ice)
  discounted_price = 15
  discounted_price = price * 2
  puts discounted_price
end
```

Multiple implementations

Figure 5 Ide allows user to introduce new variable with the same name

This warn should be more explicit, for example involve additional confirmation from user that attempts to introduce variable with an already existing name.

2 Refactor | Extract Method

2.1 Purpose

This refactoring allows you to take a piece of code and move it to a new method, replacing the original snippet with a call to that method

2.2 Test Checks

2.2.1 Extract a single line into method

2.2.1.1 Steps

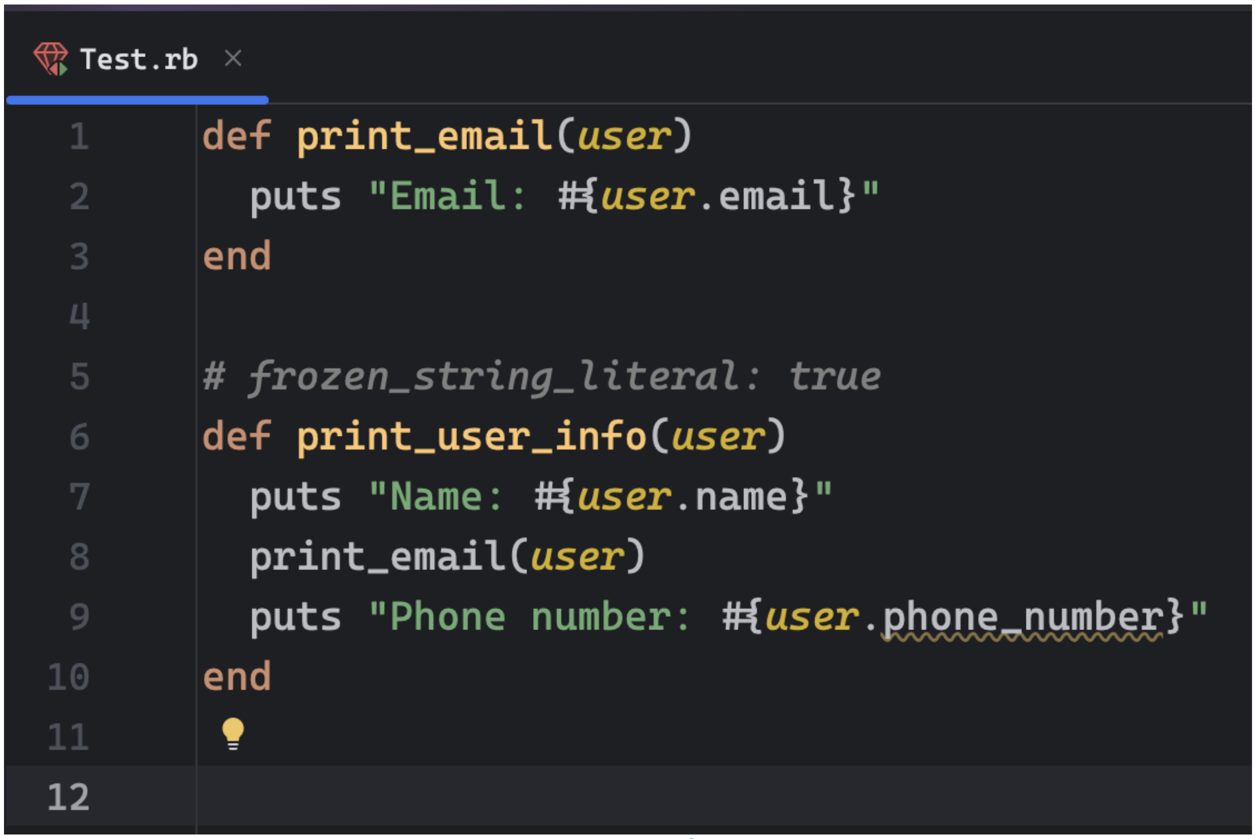
1. Write a method, which you will be extracting code from, for example:
```
def print_user_info(user)
  puts "Name: #{user.name}"
  puts "Email: #{user.email}"
  puts "Phone number: #{user.phone_number}"
end
```
2. Select a line from this method which will be extracted, for example: puts “Email: #{user.email}”
3. Use Refactor -> Extract Method
4. Provide a name for a method
5. Confirm the refactoring

2.2.1.2 Expected Result

```
def print_user_info(user)
  puts "Name: #{user.name}"
  print_email(user)
  puts "Phone number: #{user.phone_number}"
end

def print_email(user)
  puts "Email: #{user.email}"
end
```

2.2.1.3 Actual Result



Test.rb

```
def print_email(user)
  puts "Email: #{user.email}"
end

# frozen_string_literal: true
def print_user_info(user)
  puts "Name: #{user.name}"
  print_email(user)
  puts "Phone number: #{user.phone_number}"
end
```

Figure 6 Actual Result for check #1

2.2.2 Extract multiple lines into method

2.2.2.1 Steps

1. Write a method, which you will be extracting code from, for example:

```
def calculate_grand_total(price, tax_rate, discount_rate)
  discounted_price = price * (1 - discount_rate)
  taxed_price = discounted_price * (1 + tax_rate)
  final_price = taxed_price.round(2)
  puts "The grand total is #{final_price}"
  final_price
end
```

2. Select multiple lines from this method which will be extracted, for example I will extract first three lines of method that was provided earlier
3. Use Refactor -> Extract Method
4. Provide a name for a method
5. Confirm the refactoring

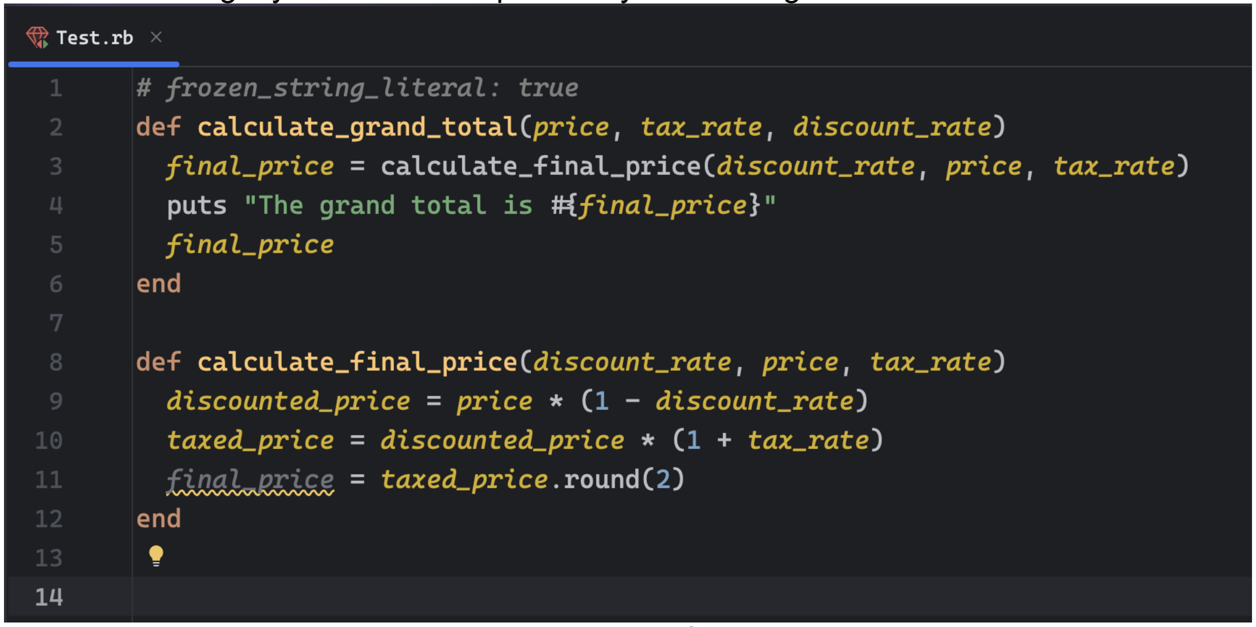
2.2.2.2 *Expected Result*

```
def calculate_grand_total(price, tax_rate, discount_rate)
  final_price = calculate_final_price(discount_rate, price, tax_rate)
  puts "The grand total is #{final_price}"
  final_price
end

def calculate_final_price(discount_rate, price, tax_rate)
  discounted_price = price * (1 - discount_rate)
  taxed_price = discounted_price * (1 + tax_rate)
  taxed_price.round(2)
end
```

2.2.2.3 *Actual Result*

Actual result slightly differs from expected by introducing useless variable:



```
# frozen_string_literal: true
def calculate_grand_total(price, tax_rate, discount_rate)
  final_price = calculate_final_price(discount_rate, price, tax_rate)
  puts "The grand total is #{final_price}"
  final_price
end

def calculate_final_price(discount_rate, price, tax_rate)
  discounted_price = price * (1 - discount_rate)
  taxed_price = discounted_price * (1 + tax_rate)
  final_price = taxed_price.round(2)
end

```

Figure 7 Actual Result for check #2

2.2.3 Extract code into a method when it depends on local variables

2.2.3.1 *Steps*

1. Write a method local variables are used, for example:

```
def purchase_items(items)
  total = 0
  items.each do |item|
    total += item.price
  end
  puts total
  total
end
```

2. Select a block of code that depends on local variable, for example

   ```
   items.each do |item|
     total += item.price
   end
   ```

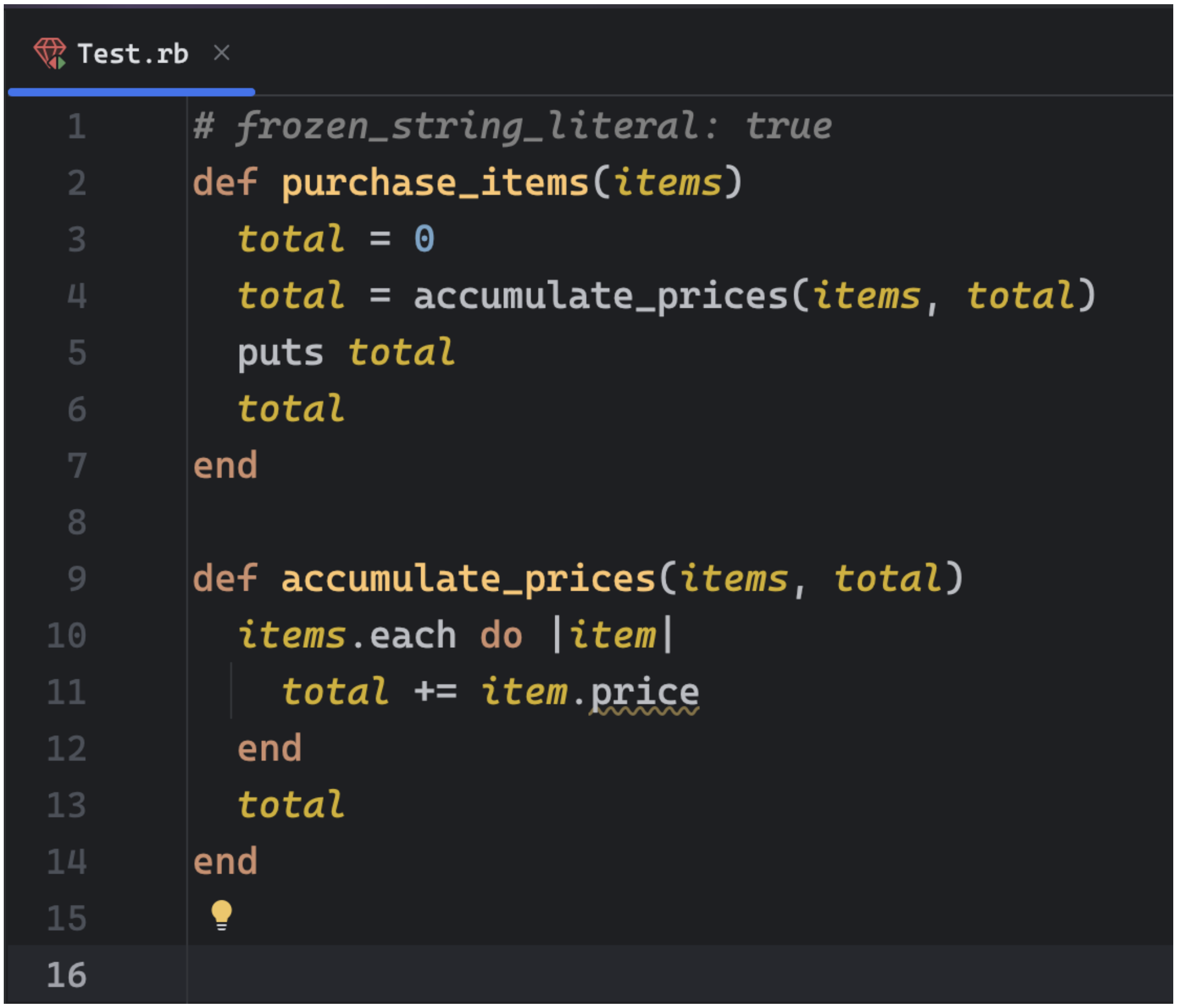
3. Use Refactor -> Extract Method
4. Provide a name for a method
5. Confirm the refactoring

2.2.3.2 Expected Result

```
def purchase_items(items)
  total = 0
  total = accumulate_prices(items, total)
  puts total
  total
end

def accumulate_prices(items, total)
  items.each do |item|
    total += item.price
  end
  total
end
```

2.2.3.3 Actual Result



Test.rb

```ruby
# frozen_string_literal: true
def purchase_items(items)
  total = 0
  total = accumulate_prices(items, total)
  puts total
  total
end

def accumulate_prices(items, total)
  items.each do |item|
    total += item.price
  end
  total
end

```

Figure 8 Actual Result for check #3