Preconditions on Rename Refactorings

Soumya Mudiyappa
West Chester University
fs926226@wcupa.edu

Swetchha Shukla
West Chester University
ss928947@wcupa.edu

Yung-Chen Cheng
West Chester University
yc917559@wcupa.edu

I. Rename Refactorings

RENAME Refactoring changes the name of identifiers in a program without changing the program's behavior. There are five types of Rename Refactoring in Java:

1) Rename Class Declarations
2) Rename Method Declarations
3) Rename Field Declarations
4) Rename Local Variables
5) Rename Package Declarations

II. Preconditions of Rename Class Refactoring

Rename Class Refactoring (RcR) changes the name of the class and all references to that class to the new name without changing its behavior. There are certain preconditions required for RcR.

1) The target class cannot be duplicate with any existing class within same package after rename.
2) The target class cannot be duplicate with any imported class from different package after rename.
3) The target class file name cannot be duplicate with any existing java file name within same package after rename.

A. The target class cannot be duplicate with any existing class within same package after rename.

When we rename a class with any existing class name, the Java compiler produces error message about duplicate class name. The classes will be conflicted if we run RcR on the target class using the name of an existing class in the same package. So, we can not have duplicate class names in the same package.

For example, if we run RcR on the class name from *A* to *B* as in Fig. 1, then the compiler produces an error message: *"duplicate class: p.B"*.

```
package p;

class A{
}

class B{
}

class C{
}
```

(a) Before

```
package p;

class B{
}

class B{
}

class C{
}
```

(b) After

Fig. 1. **Example of RcR from A to B**

Furthermore, the same situation occurs in nested classes. The examples below show that for RcR we can not use the same name either as inner or as outer class for nested classes like Fig. 2.

```
package p;

public class A{

  class M{
  }

  class N{
  }
}
```

Fig. 2. **Nested Class before RcR**

Example 1: In order to apply RcR for inner class, we should pre-check that we do not use the same name as any of other inner class name. As shown in Fig. 3, when we run RcR on the inner class from M to N, the Java compiler produces an error *"class A.N is already defined in class A"*.

```
package p;

public class A{

  class N{
  }

  class N{
  }
}
```

Fig. 3. **Example 1 for Nested Class after RcR from M to N**

Example 2: In order to apply RcR for outer class, we should pre-check that we do not use the same name as any of the inner class name and vice-versa. As shown in Fig. 4, when we use same name for outer class and inner class, the Java compiler produces an error *"class M is already defined in package p"* for Fig. 4(a) and *"class A is already defined in package p"* for Fig. 4(b).

Therefore, checking whether a class with the same name already exists in a package should be the first precondition for RcR.

B. The target class cannot be duplicate with any imported class from different package after rename.

If a class is imported from different package, we have to pre-check that the new name of the target class is not duplicate with the imported class after rename refactoring.

```
package p;

public class M{

  class M{
  }

  class N{
  }
}
```

```
package p;

public class A{

  class A{
  }

  class N{
  }
}
```

(a) After RcR on Outer Class A to M (b) After RcR on Inner Class M to A

Fig. 4. **Example 2 of Nested Class after RcR**

In Fig. 5(a), we see that class B is not duplicate with class A and we can apply RcR on class B to any other name except 'A' as mentioned in section II-A. However, in Fig. 5(b), when we apply RcR from *B to C*, Java compiler produces an error *"C is already defined in this compilation unit"*. This is because the compiler cannot distinguish between the *imported class C* of package 'p' and the *existing class C* of package 'q'.

```
package q;
import p.C;

class A{
}

class B{
}
```

```
package q;
import p.C;

class A{
}

class C{
}
```

(a) Before (b) After

Fig. 5. **RcR from B to C**

This precondition also holds good for renaming a child class. Suppose a Java file has a class and its children classes as shown in Fig. 6. We see that if parent class A imports a class C from package 'p' and if we apply RcR on the child class from B to C or D to C, Java compiler produces an error *"C is already defined in this compilation unit"*. This precondition is applicable for all ancestor class and we have to trace back and check if any of the parent class is importing a class with same name before renaming the child class within the same java file.

```
//A.java

package q;
import p.C;

public class A{
}

class B extends A{
}

class D extends B{
}
```

```
//A.java

package q;
import p.C;

public class A{
}

class C extends A{
}

class D extends B{
}
```

(a) Before (b) After

Fig. 6. **RcR for child class B to C**

If a parent class imports a class from different package and if the child class is defined in a separate java file, then in that case we can apply RcR on child class to the imported class name.

Therefore, it is essential to pre-check that the target class should not have duplicate name with any of imported class after RcR.

C. The target class file name cannot be duplicate with any existing java file name within same package after rename.

If classes are defined in separate java files, we have to pre-check that the new name of the target class declared in its class file name is not duplicate with any existing Java file name within the same package.

```
//A.java

public class A{
}

class B{
}


//C.java
//empty file
```

```
//C.java

public class C{
}

class B{
}


//C.java
//empty file
```

(a) Before (b) After

Fig. 7. **RcR from A to C if A is declared in file A.java**

For example, from Fig. 7, when we apply RcR from *A to C* where C.java is an empty file, the Java compiler produces an error *"Compilation unit 'C.java' already exists"*. This is because after RcR the Java file also gets renamed from *A.java to C.java* and therefore creates a conflict for duplicate file name as *C.java* already exists in the same package.

However, we can apply RcR from *B to C* as class B is not declared within its class file name.

III. Preconditions of Rename Method Refactoring

IV. Preconditions of Rename Field Refactoring

Rename Field Refactoring(RfR) changes the declaration and usages of the field to the new name without changing its behavior. Fields are the variables of a class i.e. instance variables and static variables. There are certain preconditions required for RfR.

1) The target name of field cannot be duplicate with any existing field within same class after rename.
2) The target name of field cannot be duplicate with any local variable within same method after rename.

A. The target name of field cannot be duplicate with any existing field within same class after rename.

In order to apply RfR on field, we have to pre-check that the new name of the field is not duplicate with any existing field name of any data type within same class. As shown in Fig. 8(b), when we apply RfR from *'j' to 'i'*, Java compiler produces an error *"variable i is already defined in class A"*. This is because of the conflict for duplicate field declaration for same type variable. As shown in Fig. 8(c), when we apply

RfR from *‘s’ to ‘i’*, Java compiler produces an error *“variable i is already defined in class A”*. This is because the compiler will not be able to distinguish between different type variables *‘String i’* and *‘int i’*.

```java
class A {

   int i = 0;
   int j = 2;
   String s = "hi";
}
```

(a) Before RfR

```java
class A {

   int i = 0;
   int i = 2;
   String i = "hi";
}
```

(b) After RfR from j to i

```java
class A {

   int i = 0;
   int j = 2;
   String i = "hi";
}
```

(c) After RfR from s to i

Fig. 8. **RfR on field variables**

B. The target name of field cannot be duplicate with any local variable within same method after rename.

In order to apply RfR on field, we have to pre-check that the new name of field is not duplicate with any local variable name, if field is being used in the same block as local variable.

As shown in Fig. 9(a), on executing the output is ‘0’. However, after we apply RfR from *‘x’ to ‘y’* and on executing Fig. 9(b), the output is ‘5’. Although the compiler did not produce any error but the behavior of code has changed since the output on RfR got changed. This is because if a field variable and a local variable have the same name, then the local variable will be accessed. This process effectively shadows the field variable and therefore known as *Variable Shadowing*.

```java
class A{

   int x = 0;
   void m(int y){
      System.out.print(x);
   }

   void main(String[] args){
      A.m(5);
   }
}
```

(a) Before

```java
class A{

   int y = 0;
   void m(int y){
      System.out.print(y);
   }

   void main(String[] args){
      A.m(5);
   }
}
```

(b) After

Fig. 9. **RfR from x to y**

The same concept is applied to class hierarchies. A field variable declared within a parent class will be shadowed by any variable with the same name in a child class. As shown in Fig. 10, on executing the output it ‘0’. However, after we apply RfR on field variable of a child class from j to i and on executing the program in Fig. 11, the output is ‘1’.

Therefore, it is essential to pre-check that the new name of field is not duplicate with any existing local variable if both are used in same block or in parent and child classes.

```java
class A {

   int i = 0;
}

class B extends A {

   int j = 1;
}

public class C {
   public static void main(String[] args){

      B b = new B();
      System.out.println(b.i);
   }
}
```

Fig. 10. **Before RfR on child class field variable j**

```java
class A {

   int i = 0;
}

class B extends A {

   int i = 1;
}

public class C {
   public static void main(String[] args){

      B b = new B();
      System.out.println(b.i);
   }
}
```

Fig. 11. **After RfR on child class field variable j to i**

V. Preconditions of Rename Local Variable Refactoring

A variable defined within a block or method or constructor is called local variable. The scope of the local variables shown in Fig. 12,are as follows

1) These variable are created when the block in entered or the function is called and destroyed after exiting from the block or when the call returns from the function.
2) The scope of these variables exists only within the block in which the variable is declared. i.e. we can access these variable only within that block.
3) Initialization of Local Variable is Mandatory.

Rename Local Variable Refactoring (RlR) changes the name of the local variable and all references to that variable to the new name without changing its behavior. There are certain preconditions required for RlR.

1) The renamed local variable cannot be duplicate with any of the existing local variable in a method or block or constructor.
2) The renamed local variable must not result in variable shadowing.

```
public class B {
        int i = 1;
        //Constructor
        B () {
            int j = 10;
            System.out.println(j); // Outputs 10
        }

        public void printNum() {
                int x = 2;
                int y = 3;
// access to local and field variables is allowed
                System.out.println(x); // outputs 2
                System.out.println(y); // outputs 3
                System.out.println(i); // outputs 1
        }

        public static void main(String args[]) {
                B b = new B();
                b.printNum();
// access to local variables is not allowed
                System.out.println(x); // Error
                System.out.println(y); // Error
                System.out.println(j); // Error
        }
}
```

Fig. 12. **Scope of local variable**

A. The renamed local variable cannot be duplicate with any of the existing local variable in a method or block or constructor.

From Fig. 13, we see two examples of duplicate local variable. In Example 1 when we apply RlR for local variable ‘x’ to ‘num’ , the java compiler produces the error as “variable num is already defined in method m1(int)”. Similarly we cannot run RlR on ‘y’ to ‘num’ or ‘x’ to ‘y’ as shown in example 2. Renaming local variables to existing local variable in a method creates a conflict of duplicity in local variable.

Therefore, it is essential to pre-check that the target name of local variable should not have duplicate name with the any of the existing local variable in a method or block or constructor after RlR.

```
public class A {

   void m1(int num) {
        int x = 1; // rename to num
        int y = 2
    }
}
```

(a) Example 1

```
public class B {

   void m1(int num) {
        int x = 1; // rename to y
        int y = 2;
    }
}
```

(b) Example 2

Fig. 13. **Duplicate local variables**

B. The renamed local variable must not result in variable shadowing.

Shadowing refers to the concept of using two variables with the same name within scopes that overlap. When we do that, the variable with the higher-level scope is hidden because the variable with lower-level scope overrides it. This results in the higher-level variable being “shadowed”.

Suppose a local variable has the same name as one of the field variable(instance variable), the local variable shadows the field variable inside the method block. For Example, from Fig. 14, the class B has two field variables (name, age) and a method (display()). In the method there are two local variables same as the field variables (name and type). When we access the variables in the method, the local variable values will be printed shadowing the field variables.

```
public class B {
        String name = "John";
        int age = 12;

        public void display() {
                String name = "Bob";
                int age = 20;
                System.out.println(name); // outputs Bob
                System.out.println(age); // outputs 20
        }

        public static void main(String args[]) {
                B b = new B();
                b.display();

        }
}
```

Fig. 14. **Ex 1 : Variable Shadowing**

Similarly variable shadowing occurs when there are same variables are defined in different methods of parent and child classes. For Example, from Fig. 15, the field variable in parent class B and child class C , local variable in method ‘m1’ of class B and method ‘m2’ of class C has the same variable name as ‘id’. When we try to access the variable ‘id’ in methods, the respective local variable will be printed shadowing the field variable of its parent class.

```
public class B {
        int id = 1;

        public void m1() {
                int id = 2;
                System.out.println("id:␣" + id);
        }
}

class C extends B {
        int id = 3;

        public void m2() {
                int id = 4;
                System.out.println("id:␣" + id);
        }
}
public class Main {
        public static void main(String args[]) {
                B b = new B();
                b.m1();          // outputs 2
                C c = new C();
                c.m2();          // outputs 4

        }
}
```

Fig. 15. **Ex 2 : Variable Shadowing**

Therefore, it is essential to pre-check that local variable must not result in variable shadowing after RlR.

VI. Preconditions of Rename Package Refactoring

For Rename Package Refactoring (RpR), we can not use duplicate package name within the same source folder. However, we can run RpR with same package name from different Java Project folders.

For example, we have two packages in one source folder like Fig. 16, we can not run RpR on package p to q since package q is already exist in the same source folder.

```
package p;

class A{
}

class B{
}
```

(a) package p

```
package q;

class M{
}

class N{
}
```

(b) package q

Fig. 16. **Example of RpR**