# 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. Introduction

Refactoring is a controlled technique for improving the design of an existing code base. Its essence is applying a series of small behavior-preserving transformations, each of which "too small to be worth doing". However the cumulative effect of each of these transformations is quite significant. By doing them in small steps we reduce the risk of introducing errors. We also avoid having the system broken while carrying out the restructuring - which allows us to gradually refactor a system over an extended period of time. [1]

In simple words, refactoring is a behavior preserving code transformation technique, for example- rename, move, extract etc. Refactoring rely on two important factors - precondition checks and code changes. In this paper, we brief about the rename refactoring and explain the preconditions for each type of rename refactorings.

## II. 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 Variables
4) Rename Local Variables
5) Rename Package Declarations

## III. 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"*.

```java
package p;

class A{
}

class B{
}

class C{
}
```

(a) Before

```java
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.

```java
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"*.

**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.

```
package p;

public class A{

  class N{
  }

  class N{
  }
}
```

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

```
package p;

public class M{

  class M{
  }

  class N{
  }
}
```

(a) After RcR on Outer Class A to M

```
package p;

public class A{

  class A{
  }

  class N
  }
}
```

(b) After RcR on Inner Class M to A

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

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 III-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{
}
```

(a) Before

```
package q;
import p.C;

class A{
}

class C{
}
```

(b) After

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

From the Fig. 6. Any class name and imported class name cannot be duplicate in the same java file. If they have the same class name, Java compiler produces an error *"C is already defined in this compilation unit"*.

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

```
//A.java

package q;
import p.C;

public class A{
}

class B extends A{
}

class D extends B{
}
```

(a) Before

```
//A.java

package q;
import p.C;

public class A{
}

class C extends A{
}

class D extends B{
}
```

(b) After

Fig. 6. **Rename Class B to C**

*C. The target class 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
```

(a) Before RcR

```
//C.java

public class C{
}

class B{
}


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

(b) After RcR from A to C

```
//A.java

public class A{
}

class C{
}


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

(c) After RcR from B to C

Fig. 7. **RcR to existing java file name**

For example, from Fig. 7(b), 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.

Also the same concept is applicable to other classes in the same package. When we apply RcR from *B to C* as shown in Fig. 7(c), the compiler produces same error *"Compilation unit 'C.java' already exists"*.

## IV. Preconditions of Rename Method Refactoring

Rename Method Refactoring (RmR) changes the name of the method and all references to that method to the new name without changing its functionality in the program.

There are three types of Rename Method Refactorings:

1) Rename Static Method Declarations.
2) Rename Non-static Method Declarations.
3) Rename Constructor Method Declarations.

There are certain preconditions required for RmR which are applicable to each type of RmR.

1) The target method cannot be a duplicate of an existing method after rename.
2) A duplicate method in a child class cannot have different return type.
3) A duplicate method in a child class cannot reduce visibility.

### *A. The target method cannot be duplicate of an existing method after rename.*

If two or more methods have same method signature with below conditions, those methods will be called as duplicate methods.

- Same method name
- Same parameter types in order
- Same number of parameters

From Fig. 8, when we apply RmR from m to n, the compiler produces an error "*method n() is already defined in class A*". This is because another duplicate method already exists in the class. Hence, we have to check the target method name can not be the same as the existing method.

```
class A {

  void m(int a) {
  }

  void n(int b) {
  }
}
```

(a) Before

```
class A {

  void n(int a) {
  }

  void n(int b) {
  }
}
```

(b) After

Fig. 8. **Rename from m to n**

To further define the duplicate methods, some of the use-cases are as follows:

*1) Duplicate Methods with convertible Input parameters:* From Fig. 9, method m1 has input parameter type as 'int' and m2 has input parameter type as 'double'. When we call method m2 with int data type as the parameter, int automatically and implicitly type-casted by java compiler into double and outputs 'bye' as shown in Fig. 9(a).

When we apply RmR from m2 to m1 and call method m1 by passing 'int' as input parameter then the output is 'hello' as shown in Fig. 9(b). Here, it is clearly evident that after RmR, the original output has changed even with pre-checking condition for duplicate method. This implies that the definition for duplicate method is not precise and we need to revise duplicate methods to prevent coercion.

Below are primitive type coercions (i.e., implicit type conversions). Java allows to convert primitive types without losing information about a numeric value.

**byte < short < int < float < double**

When the duplicate methods have the larger type size and smaller type is passed then it can be converted to larger type size. The revised definition is that the duplicate methods can have the different data type but the property of Widening or Automatic Type Conversion has to be taken care of.

```
class A {

  void m1(int i) {
    System.out.print("hi");
  }

  void m2(double j) {
     System.out.print("bye");
  }

  void main(String[] args){
     // Outputs bye
     this.m2(100);
  }
}
```

(a) Before

```
class A {

  void m1(int i) {
    System.out.print("hi");
  }

  void m1(double j) {
     System.out.print("bye");
  }

  void main(String[] args){
     // Outputs hi
     this.m1(100);
  }
}
```

(b) After

Fig. 9. **Rename from m2 to m1**

*2) Non-Duplicate methods Ambiguity:* From Fig. 10(a), methods m1 and m2 are non-duplicate methods and the output is 'hi'. However, when we apply RmR from m2 to m1 as shown in Fig. 10(b), the compiler produces an error *"reference to m1 is ambiguous as both method m1(String,float) in A and method m1(Object,int) in A match"*.

```
class A{

  void m1(String s, float d){
      // Outputs hello
      System.out.print(s);
   }

  void m2(Object o, int i){
      System.out.print(o);
   }

  void main(String[] args){
      this.m1("hi", 1);
   }
}
```

(a) Before

```
class A{

  void m1(String s, float d){
      // Error
      System.out.print(s);
   }

  void m1(Object o, int i){
      System.out.print(o);
   }

  void main(String[] args){
      this.m1("hi", 1);
   }
}
```

(b) After

Fig. 10. **Rename from m2 to m1**

Hence, it is essential to pre-check that the non-duplicate methods on RmR does not result in argument ambiguity.

### *B. A duplicate method in a child class cannot have different return type.*

If a duplicate method exists in both parent and child class, Java does not allow these duplicate methods to have different return types.

In Fig. 11 (b), when we apply RmR from B.m2() to B.m1(), compiler produces an error " *m1() in B cannot override m1() in A as return type int is not compatible with void*". This is because the duplicate method m1 have different return types which is not allowed.

```java
class A {

  void m1() {
    System.out.print("hi");
  }
}
class B extends A{

  int m2() {
    System.out.print("bye");
  }
}
```

(a) Before

```java
class A {

  void m1() {
    System.out.print ("hi");
  }
}
class B extends A{

  int m1() {
    System.out.print ("bye");
  }
}
```

(b) After

Fig. 11. **Rename from m2 to m1**

### *C. A duplicate method in a child class cannot reduce visibility.*

The use of access modifiers helps with visible duplicate methods, however there is a proper hierarchy for access modifiers to be followed for duplicate methods.

**public > protected > package-public > private**

As shown in 12(b), the package-public visibility gets reduced to private for duplicate methods m1() and compiler produces an error *"m1() in B cannot override m1() in A as attempting to assign weaker access privileges; was package"*.

```java
class A {

  void m1() {
    System.out.print ("hi");
  }
}
class B extends A{

  private void m2() {
    System.out.print ("bye");
  }
}
```

(a) Before

```java
class A {

  void m1() {
    System.out.print ("hi");
  }
}
class B extends A{

  private void m1() {
    System.out.print ("bye");
  }
}
```

(b) After

Fig. 12. **Rename from m2 to m1**

Therefore, it is essential to pre-check that new name of method does not result in visibility reduction.

## V. Preconditions of Rename Field Refactoring

Rename Field Refactoring (RfR) changes the declaration and references to the field variables 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 variable cannot be duplicate with any existing field within same class after rename.
2) The target name of field variable cannot be duplicate with any local variable within same method after rename.
3) The duplicate field variable in a child class cannot reduce visibility.

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

In order to apply RfR 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. 13(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 names for field variables. As shown in Fig. 13(c), when we apply RfR from *'s' to 'i'*, Java compiler produces an error *"variable i is already defined in class A"*.

```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 s = "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. 13. **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 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. 14(a), the output is '0'. However, after we apply RfR from *'x' to 'y'* as shown in Fig. 14(b), the output is '5'. Although the compiler have not produced any error but the behavior of code has changed since the output after RfR has 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*. 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.

```java
class A{

   int x = 0;

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

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

(a) Before

```java
class A{

   int y = 0;

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

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

(b) After

Fig. 14. **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. 15(a), the output is '0'. However, after we apply RfR on field variable of a child class from j to i as shown in Fig. 15(b), the output is '1'. This Variable Shadowing has changed the original behavior of the program after RfR.

```
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); // Outputs 0
   }
}
```

(a) Before

```
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); // Outputs 1
   }
}
```

(b) After

Fig. 15. **RfR on child class field variable j to i**

Therefore, it is essential to pre-check that the duplicate field variables in parent and child class does not result in variable shadowing.

### *C. The duplicate field variable in a child class cannot reduce visibility.*

The child class variables are able to reduce the visibility of the parent class variables. The use of access modifiers helps with visible duplicate field variables, however there is a proper hierarchy for access modifiers to be followed for duplicate variables.

$$\textbf{public} > \textbf{protected} > \textbf{package-public} > \textbf{private}$$

As shown in Fig. 16(a), the two field variables 'i' and 'j' are completely distinct and have completely separate visibilities in different classes. However, on applying RfR on child class variable from 'j' to 'i' as shown in Fig. 16(b), the public visibility of field variable 'i' gets reduced to private and Java compiler produces an error "field B.i is not visible".

Therefore, it is essential to pre-check that the new name of the field variable does not result in visibility reduction.

```
public class A {

   public int i = 0;

   public static void main(String[] args){

      B b = new B();
      System.out.print(b.i); // Outputs 0
   }
}

class B extends A {

   private int j = 1;
}
```

(a) Before RcR on child class field j

```
public class A {

   public int i = 0;

   public static void main(String[] args){

      B b = new B();
      System.out.print(b.i); // Error
   }
}

class B extends A {

   private int i = 1;
}
```

(b) After RcR on child class field j to i

Fig. 16. **Visibility reduction not allowed**

## VI. PRECONDITIONS OF RENAME LOCAL VARIABLE REFACTORING

A local variable is a variable declared inside a method and it is only accessible inside the method that declared it

Rename Local Variable Refactoring (RvR) 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 RvR.

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.

### *A. The renamed local variable cannot be duplicate with any of the existing local variable in a method.*

From Fig. 17 and Fig. 18, we see two examples of duplicate local variable. In Example 1 when we apply RvR 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 RvR on 'y' to 'num' or 'x' to 'y' . 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 any of the existing local variable in a method after RvR.

```java
public class A {

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

(a) Before

```java
public class B {

    void m1(int num) {
        int num = 1;
        int y = 2;
    }
}
```

(b) After

Fig. 17. **Rename from x to num**

```java
public class A {

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

(a) Before

```java
public class B {

    void m1(int num) {
        int y = 1;
        int y = 2;
    }
}
```

(b) After

Fig. 18. **Rename from x to y**

*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".

```java
public class B {

    public int i = 2;

    public void display() {
      int j = 1;
      System.out.println(j);
    }

    public static void main(String args[]) {

      B b = new B();
      b.display();  //Outputs 1
    }
}
```

(a) Before

```java
public class B {

    public int i = 2;

    public void display() {
      int i = 1;
      System.out.println(i);
    }

    public static void main(String args[]) {

      B b = new B();
      b.display();  //Outputs 1
    }
}
```

(b) After

Fig. 19. **Rename from j to i**

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. From Fig. 19, the class B has field variable 'i' with value 2 and a method (display()). In a method there is local variable 'j' with value 1. After RvR from 'j' to 'i' , when we access the variable in the method, the local variable value will be printed shadowing the field variable.

Similarly variable shadowing occurs when the same variables are defined in parent and child classes. For Example, from Fig. 20, the field variable in parent class B has variable 'i' with value 1 and child class C has a variable 'k' with value 3, local variable in method 'm1' of class B has the variable name as 'j' with value 2. Aft RvR from j to i and k to i ,when we try to access the variable 'i' in methods through Class B and C, the respective local variable will be printed shadowing the field variable of its parent class.

```java
public class B {

    int i = 1;

    public void m1() {
      int j = 2;
      System.out.println(j);
    }
}

class C extends B {

    int k = 3;
}

public class Main {

    public static void main(String args[]) {

       B b = new B();
       b.m1();     // outputs 2
       C c = new C();
       c.m1();     // outputs 2

    }
}
```

(b) Before

```java
public class B {

    int i = 1;

    public void m1() {
      int i = 2;
      System.out.println(i);
    }
}

class C extends B {

    int i = 3;
}

public class Main {

    public static void main(String args[]) {

       B b = new B();
       b.m1();   // outputs 2
       C c = new C();
       c.m1();   // outputs 2
    }
}
```

(b) After

Fig. 20. **Rename from j to i and k to i**

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

## VII. Preconditions of Rename Package Refactoring

Rename Package Refactoring (RpR) changes the name of the package and all references to that package to the new name without changing its behavior. There is one precondition required for RpR.

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 folders.

For example, we assume that there are two packages in one source folder like Fig. 21, 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. 21. **Example of RpR**

## References


[1] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2019.