Exercise 1

Augment the classes `Rectangle` and `LocatedRect` in the files of those names as follows:

A. For both `Rectangle` and `LocatedRect` write two methods that rotate the rectangle by 90° counterclockwise around the lower left-hand corner:

   - A *destructive* method, `R.DestRotate()`. This modifies `R` itself to be the result of the rotation. It returns `R` itself.
     
     **Answer:** For `Rectangle` there is no lower left-hand corner; all you can do is switch the xSpan and ySpan.

     ```java
     public void DestRotate() {
         setSpans(ySpan, xSpan);
     }
     
     For `LocatedRect`, the new lower left-hand corner is located at `(xL - ySpan, yL)`.

     ```java
     public void DestRotate() {
         xL = xL-ySpan;
         setSpans(getYSpan(), getXSpan());
     }
     
     ```

   - A *non-destructive* method, `R.NonDestRotate()`. This returns a new rectangle which is the result of rotating `R`, but leaves `R` unchanged.

     **Answer:** The non-destructive methods must create a new object with the proper values of the fields.

     For `Rectangle`:

     ```java
     public Rectangle NonDestRotate() {
         return new Rectangle(ySpan, xSpan);
     }
     
     For `LocatedRect`:

     ```java
     public LocatedRect NonDestRotate() {
         return new LocatedRect(xL-getYSpan(), xL, yL, yL+getXSpan());
     }
     ```

B. Suppose you have written a program that contains calls to `R.DestRotate()`. You decide to rewrite every such call as `R = R.NonDestRotate()`. Have you changed what the program does? Either give an argument that the new code does the same thing as the old code, or give an example where the new code does something different.

**Answer:** These two code sequences do something different

<table>
<thead>
<tr>
<th>Sequence 1</th>
<th>Sequence 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Rectangle R = new Rectangle(1.0,2.0);</code></td>
<td><code>Rectangle R = new Rectangle(1.0,2.0);</code></td>
</tr>
<tr>
<td><code>Rectangle S = R;</code></td>
<td><code>Rectangle S = R;</code></td>
</tr>
<tr>
<td><code>R.DestRotate();</code></td>
<td><code>R = R.NonDestRotate();</code></td>
</tr>
</tbody>
</table>

In Sequence 1, the third statement rotates both `S` and `R`, since it changes the object that both `S` and `R` point to. In Sequence 2, `R` is reassigned to a new object, and the object that `S` references is unchanged.
C. Write a method `R.Intersect(Q)` that returns the intersection of located rectangles `R` and `Q`. If `R` and `Q` do not intersect, the method should return `null`.

Answer:

```java
public LocatedRect Intersect(LocatedRect Q) {
    double ILeft = max(left(), Q.left());
    double IRight = min(right(), Q.right());
    double IBot = max(bottom(), Q.bottom());
    double ITop = min(top(), Q.top());
    if (IBot <= ITop && ILeft <= IRight)
        return new LocatedRect(ILeft, IRight, IBot, ITop);
    else return null;
}
```

Exercise 2

Suppose you want to modify the definition of `Person` in `Person.java` to record the person’s spouse. Assume that a person has at most one spouse. Write the code that you would need to:

- Add a data field `spouse`.
- Add a method `getSpouse()`.
- Add a method `setSpouse(Person P)`. When you call `Q.getSpouse(P)`, that should both mark `P` as the spouse of `Q` and mark `Q` as the spouse of `P`.

```java
private Person spouse;

public Person getSpouse() { return spouse; }

public void setSpouse(Person P) {
    spouse = P;
    P.spouse = this;
}
```

Exercise 3

Consider the code for `Hwk1Ex3.java` on the attached handout.

A. What does this output?

B. Explain the output.

Answer:

The first call to `P.show(P)` uses the method with signature `show(A X)`, because the argument `P` is declared as an `A`. It uses the method of this signature defined in class `A`, because the owner `P` is in fact an `A`. It uses the method `getName()` for class `A` because `X=P` is in fact an `A`. Therefore, it prints out “PP”.

The first call to `P.show(Q)` uses the method with signature `show(B X)`, because the argument `Q` is declared as an `B`. It uses the method of this signature defined in class `A`, because the owner `P` is
in fact an A. It uses the method `getname()` for class B because X=Q is in fact an B. Therefore, it prints out “PBCQ”.

The call to Q.show(P) uses the method with signature show(A X), because the argument P is declared as an A. It uses the method of this signature defined in class B, because the owner Q is in fact a B. It uses the method `getname()` for class A because X=P is in fact an A. Therefore, it prints out “QDP”.

The call to Q.show(Q) uses the method with signature show(B X), because the argument Q is declared as an B. It uses the method of this signature defined in class B, because the owner Q is in fact an B. It uses the method `getname()` for class B because X=Q is in fact an B. Therefore, it prints out “QECQ”.

After the reassignment P=Q. So P now is a reference to the same object as Q; note that the name is ”Q” and that it is a B.

The second call to P.show(P) uses the method with signature show(A X), because the argument P is declared as an A. It uses the method of this signature defined in class B, because the owner P is in fact an B. It uses the method `getname()` for class B because X=P is in fact an B. Therefore, it prints out “QDCQ”.

The second call to P.show(Q) uses the method with signature show(B X), because the argument Q is declared as an B. It uses the method of this signature defined in class B, because the owner P is in fact an B. It uses the method `getname()` for class B because X=Q is in fact an B. Therefore, it prints out “QECQ”.

**Honors Exercise**

The idea behind the method `setParent(P)` is that it should maintain the integrity constraint that, if P is the mother or father of Q then Q is a child of P. However, as written, this code is breakable.

A. Describe how an external program can create a set of Person objects that do not satisfy the constraint. Your solution to this part should not involve any changes to the file Person.java.

**Answer:** The problem is that if you assign one person X to be the father of Q, and then assign another person Y to be the father of Q, it will change the father of Q from X to Y, and it will add Q to the children of Y, but it does not eliminate Q among the children of X. For example with the following sequence of calls:

```java
public static void main(String args[]) {
    Person John = new Person("John","Smith",Sexes.Male);
    Person Abe = new Person("Abe","Brown",Sexes.Male);
    Person Susan = new Person("Susan","Johnson",Sexes.Female);
    Susan.setParent(John);
    Susan.setParent(Abe);
    System.out.println(Susan.getFather().getFirstName());
    listChildren(John)
}
```

At the end, the father of Susan is recorded to be Abe, but Susan is still the first child of John.

B. Write an improved definition of `setParent` that fixes this problem.
**Answer:** One solution would be to require that the `father` and `mother` fields can only be set once. Add a statement to `setParent(P)` that if the relevant parent of `this` is non-null, then raise an exception. But that is rather Draconian.

A better solution is to change `setParent` so that, if the parent was a non-null value `P` and is now `Q`, then `this` is removed from the children of `P`. That takes a little work, since if you’ve implemented a set as an array, then deleting one element involves sliding everything after it down. Here’s the code:

```java
// removeChild(P) removes "this" from P's list of children.

private void removeChild(Person P) {
    int i = 0;
    while (i < P.numChildren && P.children[i] != this) i++;
    // Search for this among the children of P.
    if (i == P.numChildren)
        System.out.println("Help! " + this.firstName + " " +
                        this.lastName + " is not a child of " + P.firstName
                        + " " + P.lastName);
        // You should never reach this point in the code
        // but just in case.
    else { // i is the index of this among the children of P
        P.numChildren--; // 1 fewer child
        while (i < P.numChildren) { // shift all the children after this
            P.children[i] = P.children[i+1];
            i++;
        }
    } // end else
} // end removeChild

// setParent(P) sets P to be the parent (of the right type) of this,
// and adds this to the children of P.
public void setParent(Person P) {
    if (P.numChildren < MaxChildren) {
        // Note that if P has already has MaxChildren, then nothing happens
        P.children[P.numChildren] = this;
        P.numChildren = P.numChildren + 1;
        if (P.sex == Sexes.Male) {
            if (father != null) removeChild(father);
            father = P;
        } else {
            if (mother != null) removeChild(mother);
            mother = P;
        }
    } // end SetParent
} // end Person
```