LAB D
Translating Objects

Perform the following groups of tasks:

LabD1.s

1. Create a directory to hold the files for this lab.

2. Create and run the following two Java classes:

```
public class Fraction
{
     private int numerator;
     private int denominator;

     public Fraction(int numerator, int denominator)
     {
          this.numerator = numerator;
          this.denominator = denominator;
     }

     public int getNumerator()   { return this.numerator;  }
     public int getDenominator() { return this.denominator;}

     public void add(Fraction other)
     {
          this.numerator = this.numerator * other.denominator +
                           this.denominator * other.numerator;
          this.denominator = this.denominator * other.denominator;
     }
}
```

```
public class FractionClient
{
     public static void main(String[] args)
     {
          Fraction a = new Fraction(3,8);
          Fraction b = new Fraction(1,2);
          a.add(b);
          System.out.print(a.getNumerator());
          System.out.print('/');
          System.out.print(a.getDenominator());
     }
}
```

We seek to explore how such classes are translated.

3. Launch your favourite editor and create LabD1.s as follows:

```
        .globl   Fraction
        .text

Fraction:
        #-------------------------------------------
        add      $t0, $0, $a0      # t0 = numerator
        add      $t1, $0, $a1      # t1 = denominator

        # store the attributes somewhere in memory
        # and return a reference to where they are

        jr       $ra
```

   This is our first attempt at translating the constructor of `Fraction`. The top section specifies that the constructor is `public`. The constructor's body extracts the parameters passed to it (the numerator and denominator) and stores them in temporary registers. We now need to store these two values somewhere in memory so we can retrieve them in the other methods of the class.

4. Where would you store these integers? Based on what we learned in earlier labs, *two* possibilities should come to mind. The following tasks ask you to explore each.

5. The first possibility is to store the attributes in **`.data`**, i.e. create a data segment in our program and allocate space in it via **`.word`** as we did in the previous lab for the static attributes of a utility. Argue that this approach will *not* work. Your argument must explain why it is OK to use the data segment for storing static attributes but not so for non-static (instance-based) ones. To that end, note how clients of `Fraction` use it to create objects:

```
Fraction a = new Fraction(3,8);
Fraction b = new Fraction(1,2);
```

6. The second possibility is to store the attributes on the **`stack`**, i.e. push them. If we followed this approach, we would have to decrement **`$sp`** (by 8) in order to allocate space for two integers on the stack. Argue that this approach does *not* work. Recall how **`$sp`** behaves within a method and after it exits.

7. If static and automatic storage options do not work, only one storage option remains: the **`heap`**. This area of memory starts after **`.data`** and grows toward the stack, i.e. it grows toward larger addresses. Unlike the stack, there is no register associated with it because it is not intended to be a simple LIFO structure. Instead, a system call is provided to allocate room in it. Add these lines to the constructor:

```
addi    $v0, $0, 9
addi    $a0, $0, 8
syscall
```

   System call 9 expects the number of needed bytes in **`$a0`**. It allocates a block of that many bytes on the heap and returns its starting address in **`$v0`**.

8. Now that you have a space to hold the attributes, go ahead and store them:

```
sw      $t0, 0($v0)
sw      $t1, 4($v0)
```

9. This completes the development of the constructor. What should it return? Given that a client can create an arbitrary number of objects, we need to remember where the attributes of each are stored, and this is ideally delegated to the client. We therefore return the heap address to the client. Since this address is currently in **$v0** and this is where we normally place the return, no further action is needed other than:

```
jr      $ra
```

LabD1Client.s

10. Create the program LabD1Client.s to translate the following:

```
Fraction a = new Fraction(3,8);
```

As usual, the client must start by pushing the return address:

```
        .text
main:   sw      $ra, 0($sp)
        addi    $sp, $sp, -4
```

It should then create a `Fraction` object:

```
        addi    $a0, $0, 3
        addi    $a1, $0, 8
        jal     Fraction
        add     $s0, $0, $v0    # s0 holds a
```

The last line captures the returned address and stores it in **$s0**. This address is the *handle* (or *reference* or *pointer*) through which we can access our object.

11. Launch SPIM and open LabD1.s and then LabD1Client.s.

12. Run the program. Our client does not generate any output but note the content of the data pane in SPIM. You see the contents of the heap in between static data and the stack. Do you see the attributes of our fraction object?

13. Modify the client so it also translates the following statement:

```
Fraction b = new Fraction(1,2);
```

14. Have your client store the object reference **b** in register **$s1**.

15. Reload LabD1 and its modified client in SPIM and step through the execution while monitoring the heap and the stack in the data pane.

16. When the program ends, examine the contents of **$s0** and **$s1**. Justify your findings.

LabD2 and its Client

17. Save LabD1 and its client as LabD2 and its client.

18. We seek to modify LabD2.s by adding the `getNumerator` accessor:

```
getNumerator:
        #-----------------------------------------

        // retrieve the numerator of this
        // fraction and return it in $v0.

        jr      $ra
```

19. We know from Java that this method does not take any parameter. A client of ours invokes it on an object reference like this:

```
int n = a.getNumerator();
```

    But if no parameters are passed, how can our assembly method possibly know the address of the object in question? At any given time, many `Fraction` objects may be present in the heap and we have no way of knowing which one is involved in this particular invocation. The solution is to pass the object reference (the one before the dot operator) as an implicit parameter. As a rule, we will always pass it as the first parameter, i.e. in **`$a0`**.

20. Based on this, our accessor can expect the heap's address in **`$a0`**, and hence, it can implement its body in one instruction, as follows:

```
lw      $v0, 0($a0)
```

21. Create an accessor for the fraction's denominator:

```
getDenominator:
        #-----------------------------------------

        // retrieve the denominator of this
        // fraction and return it in $v0.

        jr      $ra
```

    Here too the body can be implemented in *one* statement.

22. To test our work, we modify our LabD2Client.s by invoking the accessors and outputting their returns. For example, here is how to invoke **`getNumerator`** on **`$s0`**:

```
add     $a0, $0, $s0
jal     getNumerator
```

    This determines the numerator of the fraction object pointed at by **`$s0`** and returns it in **`$v0`**. The client can sore the return in some **`s`** variable or print it.

23. Complete the development of LabD2Client so it prints both the numerator and denominator of the two fractions (in **`$s0`** and **`$s1`**).

24. For clarity, have your client output a slash character between the numerator and denominator. For example, the first output would look like this:

```
3/8
```

    To facilitate this, you may want to create a Util.s class containing two methods:

```
printFraction
println
```

    The first takes two integers and prints them delimited with a slash, the second prints an empty line to be inserted in between the two fraction outputs.

LabD3 and its Client

25. Save LabD2 and its client as LabD3 and its client.

26. We now seek to incorporate the `add` method in our LabD3.s class. First of all, since `add` is an instruction name, and these cannot clash with labels, let us switch to the name **`adding`**. Hence, the method would start like this:

```
adding:
        #-----------------------------------------

        // Given a fraction address in $a0,
        // and a second fraction address in $a1,
        // mutate the fraction at $a0 so it becomes
        // the sum of the two fractions

        jr      $ra
```

27. Note that your method can readily access the numerator and denominator of either fraction. It can do so directly (via the passed parameters and the load-word instruction) or indirectly by invoking the accessor. Note, however, that the indirect approach makes **`adding`** a non-leaf method, and, hence, necessitates pushing **`$ra`** in the prologue and popping it in the epilogue.

28. Given the two numerator and denominator pairs, we can compute the sum by a literal translation of the Java code given at the beginning of this lab. Recall, however, that the instruction:

```
mult    $t0, $t1
```

    computes the product as a 64-bit quantity and stores it in the LO and HI registers. We will assume (as the Java developer of `Fraction` did) that the product fits in 32 bits in all cases and, hence, ignore HI and look only at LO. Note that the appropriateness of this decision rests with the Java programmer, or the high-level designer, not the translator, which is the role we are assuming in these labs.

29. Complete the development of LabD3.s. Recall that, as a callee, you must preserve the contents of the **s** registers. Hence, any intermediate values should be stored in the **t** registers.

30. Modify your LabD3Client so it tests fraction addition.

31. Load your class and its client in SPIM and run. Does your code behave as expected? Do you get the same output as the one generated earlier by the Java code? If not, use the breakpoint/step features of SPIM to locate the problem and fix it.

LabD4 and its Client

32. Save LabD3 and its client as LabD4 and its client.

33. Modify LabD4 to incorporate a translation of the following mutators:

```
public void setNumerator(int numerator)
{
     this.numerator = numerator;
}

public void setDenominator(int denominator)
{
     this.denominator = denominator;
}
```

34. Modify LabD4Client to test the above changes and then run your class and client in SPIM. Does your code behave as expected?

LabD5 and its Client

35. Save LabD4 and its client as LabD5 and its client

36. Modify LabD5 in order to incorporate a translation of the following methods:

```
public void multiply(Fraction other)
{
     this.numerator = this.numerator * other.numerator;
     this.denominator = this.denominator * other.denominator;
}
public void subtract(Fraction other)
{
     Fraction temp = new Fraction(-other.numerator,
                                  other.denominator);
     this.adding(temp);
}
public void divide(Fraction other)
{
     Fraction temp = new Fraction(other.denominator,
                                  other.numerator);
     this.multiply(temp);
}
```

37. Modify LabD5Client to test the above changes and then run your class and client in SPIM. Does your code behave as expected?

LabD6 and its Client

38. We seek to create a class similar to Java's `StringBuffer`. Here is a sample client of that class:

```
public static void main(String[] args)
{
     StringBuffer sb = new StringBuffer(16);
     sb.append('A');
     sb.append('B');
     System.out.println(sb);
     sb.setCharAt(1, 'A');
     System.out.println(sb);
}
```

39. Start fresh and create LabD6 as follows:

```
          .globl   SB
          .globl   append
          .globl   setCharAt
          .text
SB:       #-------------------- # a0 = maximum size
          addi     $v0, $0, 9
          syscall
          sb       $0, 0($v0)    # null terminator
          jr       $ra
```

40. The constructor allocates room on the heap for the passed capacity. We assume that the string will never exceed that capacity and treat this assumption as a precondition. In order to determine the size of the string, we adopt the convention that the string is null-terminated, i.e. its last byte contains 8 zero bits.

41. The append method receives the string's address in **`$a0`** and a char in **`$a1`**. It must append the string with the passed character. This means it should first find the null terminator; replace it with the passed character; and then null terminate:

```
append:   #---------------------a0=reference, a1=char
          lb       $t0, 0($a0)

          ... (about 5 lines)

          jr       $ra
```

42. Complete the development of this method.

43. The last method in this class is **`setCharAt`**. This method allows the client to mutate the string by changing one of its characters. It expects the string's address in **`$a0`**, the position of the character to change in **`$a1`**, and the new character in **`$a2`**. The position is assumed to be between 0 and one less than the length of the string.

```
setCharAt:
        #-----------------a0=string, a1=position, a2=char

        ... (about two lines)

        jr      $ra
```

44. Create LabD6Client along the following lines:

```
        .text
main:   sw      $ra, 0($sp)
        addi    $sp, $sp, -4

        addi    $a0, $0, 16
        jal     SB              # call the constructor
        add     $s0, $0, $v0

        add     $a0, $0, $s0
        addi    $a1, $0, 'A'
        jal     append          # append 'A'

        add     $a0, $0, $s0
        addi    $a1, $0, 'B'
        jal     append          # append 'B'

        add     $a0, $0, $s0
        addi    $v0, $0, 4
        syscall                 # print the string at a0

        addi    $sp, $sp, 4
        lw      $ra, 0($sp)
        jr      $ra
```

The client uses system call #4 to print the returned string. This service expects the starting address of the string in **`$a0`**. It prints until it encounters the null terminator.

45. Load LabD6 and its client in SPIM and run. Does your code behave as expected?

46. Include a test of your **`setCharAt`** in the client.

LAB D

Notes

-