JPL :: Modular Programming - 2

TalentSprint

Licensed To Skill

Version 1.0.4

Learning Objectives

By the end of this presentation, you are able to:

- Write programs to problems by decomposing functionality into methods and using the methods
- Write computationally efficient programs
- Create meaningful functional decomposition systematically
- Develop and test your programs progressively

Modular Programming - 2

Write a method to check whether or not a number is a Power of two. Using that method, print all 4-digit powers of two.

Modular Programming - 2

Power of Two

```
public class PowerOfTwo {
    public static void main(String[] args) {
        int i ;
        for (i  = 1000; i <= 9999; i++) {
            if  (isPowerOfTwo (i))
                System.out.println(i + " is power of two");
        }
    }
    public static boolean isPowerOfTwo(int givenNumber) {
        int i ;
        for (i  = 2;  i  <= givenNumber; i = i * 2) {
            if  (i  == givenNumber)
                return true;
            return false;
        }
    }
}
```

Modular Programming - 2

Power of Two - More Configurable

```
public class PowerOfTwo {
    public static void main(String[] args) {
        int i;
        int n1 = Integer.parseInt(args[0]);
        int n2 = Integer.parseInt(args[0]);
        for (i = n1; i <= n2; i++) {
            if (isPowerOfTwo (i))
        System.out.println(i + " is power of two");
        }
    }
    public static boolean isPowerOfTwo (int givenNumber) {
        int i;
        for (i = 2; i <= givenNumber; i = i * 2) {
            if (i == givenNumber)
                return true;
            return false;
        }
    }
}
```

Modular Programming - 2

Power of Two - Different Logic

```
public static boolean isPowerOfTwo(int givenNumber) {
    while (givenNumber > 1) {
        if  (givenNumber % 2 == 1)
            return false;
        givenNumber  /= 2;
    }
    return true;
}
```

Modular Programming - 2

Why Methods?

Now, print all the 4-digit Powers of Two with new logic.

↓

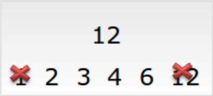
You will find that you do not need to change the mai method at all.

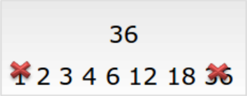
↓

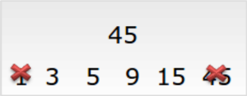
Which logic is better for PowerOfTwo?

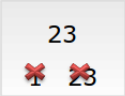
Modular Programming - 2

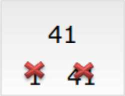
Write a method to check whether a number is prime or not.









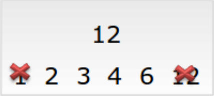


Modular Programming - 2

```
public class PrimeNumber {
    public static void main(String[] args) {
        int number = Integer.parseInt(args[0]);
        if  (isPrime (number))
            System.out.println(number + " is prime");
        else
            System.out.println(number + " is not prime");
    }
    public static boolean isPrime(int givenNumber) {
        int i ;
        for (i  = 2;  i  < givenNumber; i++)
            if  (givenNumber % i == 0)
                return false;
        return true;
    }
}
```

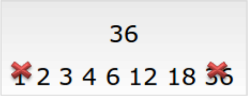
Modular Programming - 2

Prime Number - Different Logic



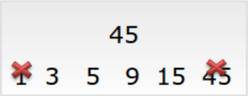
12

~~1~~ 2 3 4 6 ~~12~~



36

~~1~~ 2 3 4 6 12 18 ~~36~~



45

~~1~~ 3 5 9 15 ~~45~~

```
public static boolean isPrime(int givenNumber) {
    int i;
    for (i = 2; i <= givenNumber/2; i++) {
        if (givenNumber % i == 0)
            return false;
        return true;
    }
}
```

Modular Programming - 2

Write program that prints prime numbers between 2 and a given number n.

Modular Programming - 2

```
public class PrimeNumberRange {
    public static void main(String[] args) {
        int number = Integer.parseInt(args[0]);
        int i ;
        for(i = 2; i <= number; i++) {
            if (isPrime (i))
                System.out.println(number + " is prime");
        }
    }
}
```

Modular Programming - 2

Write a program to find the aliquot sum of a given number.

Hint:

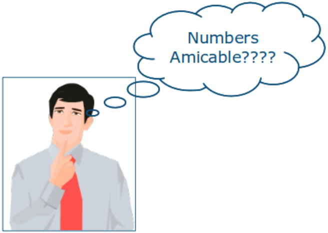
The aliquot divisors of a number are all of its divisors except the number itself. The aliquot sum is the sum of the aliquot divisors. For example, the aliquot divisors of 12 are 1, 2, 3, 4, and 6 and it's aliquot sum is 16.

Modular Programming - 2

Let us now explore the modularity a bit more and understand how we can decompose functionality into methods...

Problem

Write a program to check whether two given numbers are amicable or not.



Modular Programming - 2

Amicable Numbers

Aliquot Divisor Numbers with which a given number gets evenly divided

Modular Programming - 2

Aliquot Divisors of 284 are: 1, 2, 4, 71 and 142

↓

Sum of Aliquot Divisors of 284 → ASUM(284)
= 1 + 2 + 4 + 71 + 142 = 220

Aliquot Divisors of 220 are: 1, 2, 4, 5, 10, 11, 20, 22, 44, 55 and 110

↓

Similarly, the sum of Aliquot Divisors of 220 → ASUM(220)
= 1 + 2 + 4 + 5 + 10 + 11 + 20 + 22 + 44 + 55 + 110 = 284

ASUM(284) = 220 and ASUM(220) = 284

↑

Pair of any numbers in such relation (i.e. ASUM(A)=B and ASUM(B)=A) are called Amicable Pairs.

Modular Programming - 2

Let us write a program to print all 4-digit amicable pairs.
Let us decompose functionality into methods. First, we need a method to find aliquot sum of a given number

```
public static int  aliquotSum(int n) {
    int i;
    int sum = 1;
    for (i  = 2;  i  <= n/2; i++)
        if  ((n % i) == 0)
            sum += i;
    return sum;
}
```

Modular Programming - 2

Next, we probably need a method to Find Amicable Pairs - Using *aliquotSum()* Method:

```
public static boolean areAmicable(int n1,int n2) {
    return ((aliquotSum(n1) == n2) && (aliquotSum(n2) == n1)
     );
}
```

Modular Programming - 2

Decomposing functionality into methods

Program to Find Amicable Pairs in a Range:

```
public static void  amicablePairsInRange (int lower, int
     upper) {
    int i  = lower;
    while (i <= upper ) {
        int j  = aliquotSum(i);
        if (i  == aliquotSum(j) && j >= i)
            System.out.println("("+i+" , "+j+")");
        i++;
    }
}
```

Lesson to Learn:

Just because we wrote a method, we don’t have to use it.

Modular Programming - 2

Decomposing functionality into methods

Amicable pairs in a range - main:

```
class AliquotBad {
    public static void main(String[] args) {
        int n1 = Integer.parseInt(args[0]);
        int n2 = Integer.parseInt(args[1]);
        amicablePairsInRange (n1,n2);
    }
}
```

Modular Programming - 2

Writing Computationally Efficient Programs

Program to Find Aliquot Sum:

```
public static int  aliquotSum(int n) {
    int i  = 2;
    int sum = 1;
    while ( i * i  <= n) {
        if  ((n % i) == 0)
            sum += i + (n /i);
        i++;
    }
}
```

Modular Programming - 2

Writing Computationally Efficient Programs

Amicable pairs in a range - main:

```
class AliquotGood {
    public static void main(String[] args) {
        int n1 = Integer.parseInt(args[0]);
        int n2 = Integer.parseInt(args[1]);
        amicablePairsInRange (n1,n2);
    }
}
```

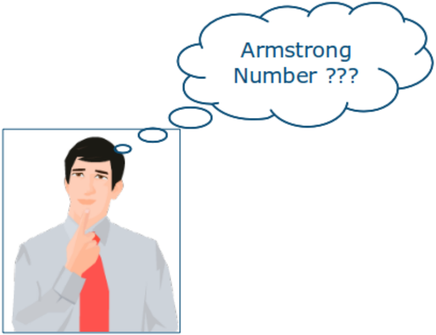
Modular Programming - 2

Writing Computationally Efficient Programs

Now, run AliquotBad and AliquotGood, to print all 5-digit amicable pairs, 6-digit amicable pairs and watch the difference.

Modular Programming - 2

Write a program that prints all Armstrong numbers between two given numbers.



Modular Programming - 2

Armstrong Number

- A Number of which, each digit is powered by the total digits in the number and summed up and...
- if the sum becomes equal to the original number then the number is called Armstrong Number

```
          153
      153 ==> 13 53 33
  153 13 53 33 13 + 53   + 33
153 13 53 33 13 + 53   + 33 = 153
```

Modular Programming - 2

Making Meaningful Functional Decomposition

Problem

Writing a program that prints all Armstrong numbers between two given numbers.

- Step 1 At the highest level, we need to loop through each number between the two given numbers
- Step 2 Check if it is Armstrong Number
- Step 3 If it is, print it.

Looping and printing can be done by **main()** method. Checking can be done by another method called *isArmstrong()*.

Modular Programming - 2

Making Meaningful Functional Decomposition

Problem

Writing *isArmstrong()* method

Step 1 It requires us to find Sum of Powers of Digits.

Step 2 First, we can have a method to find digits of the given number. Let's call it *getDigits()*.

Step 3 Let's have another method that finds sum of powers for the given digits. Let's call it *sumOfPowers()*.

So, we will have four methods namely: main, isArmstrong, getDigits, sumOfPowers.

Modular Programming - 2

Code for *getDigits()* Method:

```
public static int[] getDigits(int number) {
    int[]  digits  = new int[Integer.toString(number).length()];
    int index=0;
    while (number > 0) {
        digits[index++] = number % 10;
        number = number / 10;
    }
    return digits;
}
```

Modular Programming - 2

Let us test *getDigits()* first

```
public static void main(String args[]) {
    int digits [];
    int givenNum = Integer.parseInt(args[0]);
    digits  []  = getDigits(givenNum);
    for (int i  = 0;  i  < digits.length; i++)
        System.out.println(digits[i]);
}
```

Modular Programming - 2

Code for *sumOfPowers()* Method:

```
public static int sumOfPowers(int[] digits) {
    int sum = 0;
    int length = digits.length;
    for(int counter = 0; counter < length; counter++)
        sum += Math.pow(digits[counter], length);
    return sum;
}
```

Modular Programming - 2

Now, let us test *sumOfPowers*

```
public static void main(String args[]) {
    int numDigits[];
    int givenNum = Integer.parseInt(args[0]);
    numDigits [] = getDigits(givenNum);
    int sum = sumOfPowers(numDigits);
    System.out.println(sum);
}
```

Modular Programming - 2

Code for *isArmstrong()* Method:

```
public static boolean isArmstrong(int number) {
    int numDigits[];
    numDigits [] = getDigits(number);
    int sumPowers = sumOfPowers(numDigits);
    return (number == sumPowers);
}
```

Modular Programming - 2

Code for **main()** Method:

```
public static void main(String args[]) {
    int startNum = Integer.parseInt(args[0]);
    int endNum = Integer.parseInt(args[1]);
    for(int i  = startNum; i <= endNum; i++)
        if  (isArmstrong(i))
            System.out.println(i+ "   ");
}
```

Modular Programming - 2

For the following problems, use a separate method that takes line number (n) as a parameter and returns 'n'th line as a string. The main method calls that method iteratively to print one line at a time.

1. Write a program which accepts a number as argument and print the following output. The pyramid shape should be preserved for 2-digit numbers as well.

```
Input : 5
Output:
          1
        1 2
      1 2 3
    1 2 3 4
  1 2 3 4 5
```

Modular Programming - 2

2. Write a program which accepts a number (number of lines) and character as arguments and print the following output.

```
Input  : 5,  %
Output :

         %
       %   %
      %  %  %
     %  %   %  %
    %  %  %  %  %
```

Modular Programming - 2

Problem

Write a program that finds the least number that can be formed using the digits of the given number.

Step 1 Get the digits of the given number. (getDigits method)

Step 2 Sort the digits. (sort method)

Step 3 Form the number with the sorted digits. (getNumberFromDigits method)

Modular Programming - 2

Code for *getDigits()* Method:

```
public static int[] getDigits(int number) {
    int[]  numDigits = new int[Integer.toString(number).length
     ()];
    int index=0;
    while(number > 0) {
        numDigits[index++] = number % 10;
        number = number / 10;
    }
    return numDigits;
}
```

Modular Programming - 2

Arays.sort() method is available in Java which sorts an array. Now, we need another method for generating the number from sorted digits.

Code for generating number from digits

```
public static int getNumberFromDigits(int[] digits) {
    int theNumber = 0;
    for (int i = 0; i < digits.length; i++)
        theNumber = theNumber * 10 + digits[i];
    return theNumber;
}
```

Modular Programming - 2

Then, we will have the main method using all the methods.

```
public static void main(String args[]) {
    int givenNum = Integer.parseInt(args[0]);
    int[]  digits  = getDigits(givenNum);
    Arrays.sort(digits);
    int leastNum = getNumberFromDigits(digits);
    System.out.println ("The least no. is : " + leastNum);
}
```

Modular Programming - 2

Can we do without using sort?

1. We know that all digits are between 0 and 9.
2. So, parse the digits of the given number few times.
3. In the first pass, find and copy all 0’s.
4. In the second pass, find all 1’s, then all 2’s and so on until all 9’s.
5. Then, form the number from the copied digits.

Modular Programming - 2

Code for parsing the digits

```
public static int[] parseDigits(int[] origDigits) {
    int[]  parsedDigits = new int[origDigits.length];
    int i, j, index = 0;
    for (i = 0; i <= 9; i++)
        for (j = 0; j < origDigits.length; j++)
            if (origDigits[j] == i)
                parsedDigits[index++] = i;
    return parsedDigits;
}
```

Modular Programming - 2

Write the main method now using the *parseDigits()* method instead of **Arrays.sort()**.
Analyze which approach is better.

Modular Programming - 2