List of Slides

Title

Java Just in Time

John Latham

October 16, 2018

Chapter 6

Control statements nested in loops

Chapter aims

- We can nest **statement**s within each other
  - thus have complex **algorithm** structures to solve significant problems.

Section 2

Example:
Film certificate age checking the whole queue

Aim

AIM: To introduce the ideas of nesting an **if statement** within a **for loop**, and declaring a **variable** inside a **compound statement**. We also introduce the **conditional operator**.

Statement: statements can be nested within each other

- Execution control statements, e.g. **loop**s, **if else statement**s, contain other **statement**s
  - those can be any statement, including execution control statements.
- Allows complex **algorithm**s – unlimited nesting.
- E.g. print out all non-negative multiples of x between y (≥ 0) and z inclusive (inefficient. . .).

```
for (int multipleOfX = 0; multipleOfX <= z; multipleOfX += x)
  if (multipleOfX >= y)
    System.out.println("A multiple of " + x + " between " + y
                       + "and " + z + " is " + multipleOfX);
```

Coffee time: Why is this code inefficient?

Variable: can be defined within a compound statement

- Can declare **variable**s inside body of a **method** (practically) anywhere.
  - can be used from that point within the method body.
    - **scope**.
- But scope of variables declared inside a **compound statement** is restricted to the compound statement
  - don't exist after the compound statement.
- Allows us to localize variables to exact point of their use
  - don't clutter other parts of the code
  - can't accidentally use variables where have no relevance.

Variable: can be defined within a compound statement

```
public static void main(String[] args)
{
  ...
  int x = ...
  ... x is available here.
  while (...)
  {
    ... x is available here.
    int y = ...
    ... x and y are available here.
  } // while
  ... x is available here, but not y,
  ... so we cannot accidentally refer to y instead of x.
} // main
```

Film certificate age checking the whole queue

```java
// Program to check the ages of a queue of film goers.
// First argument is the minimum age required.
// Remaining arguments is the ages of people in the queue.
public class FilmAgeCheck
{
  public static void main(String[] args)
  {
    // The minimum age required to watch the film.
    int minimumAge = Integer.parseInt(args[0]);

    // The number of underage people found so far, initially 0.
    int underAgeCountSoFar = 0;

```

Film certificate age checking the whole queue

```java
    // We loop through the queue, checking each age.
    for (int queuePosition = 1; queuePosition < args.length; queuePosition++)
    {
      int ageOfPersonAtQueuePosition = Integer.parseInt(args[queuePosition]);
      if (ageOfPersonAtQueuePosition < minimumAge)
      {
        System.out.println("The person at position " + queuePosition
                           + " is only " + ageOfPersonAtQueuePosition
                           + ", which is less than " + minimumAge);
        underAgeCountSoFar++;
      } // if
    } // for

```

Film certificate age checking the whole queue

```
    // Now report how many underage were found.
    if (underAgeCountSoFar == 1)
      System.out.println("There is 1 under age");
    else
      System.out.println("There are " + underAgeCountSoFar + " under age");
  } // main

} // class FilmAgeCheck
```

Coffee time: What if we tried to use `ageOfPersonAtQueuePosition` after the loop?

What is the scope of `queuePosition`?

- Notice the fuss for “is” or “are”.

Expression: conditional expression

- Have different sub-expressions **evaluate**d depending on a **condition**
  - **conditional operator** permits **conditional expression**s.
- General form:

```
c ? e1 : e2
```

  - c is condition, e1 and e2 are **expression**s of some **type**.
- c is evaluated
  - if c is **`true`** e1 is evaluated
  - if c is **`false`** e2 is evaluated instead.

Expression: conditional expression

- E.g.:

  ```
  int maxXY = x > y ? x : y;
  ```

  same effect as:

  ```
  int maxXY;
  if (x > y)
    maxXY = x;
  else
    maxXY = y;
  ```

Film certificate age checking the whole queue

Coffee time: Convince yourself that the last **if else statement** of the main method of `FilmAgeCheck` could be replaced with the following.

```
System.out.println("There "
                   + (underAgeCountSoFar == 1 ? "is" : "are")
                   + " " + underAgeCountSoFar + " under age");
```

Coffee time: The brackets around the **conditional expression** in the code above are necessary – what does that tell you about the **operator precedence** of the **conditional operator**?

Trying it

Console Input / Output

```
$ java FilmAgeCheck 18
There are 0 under age
$ _
```

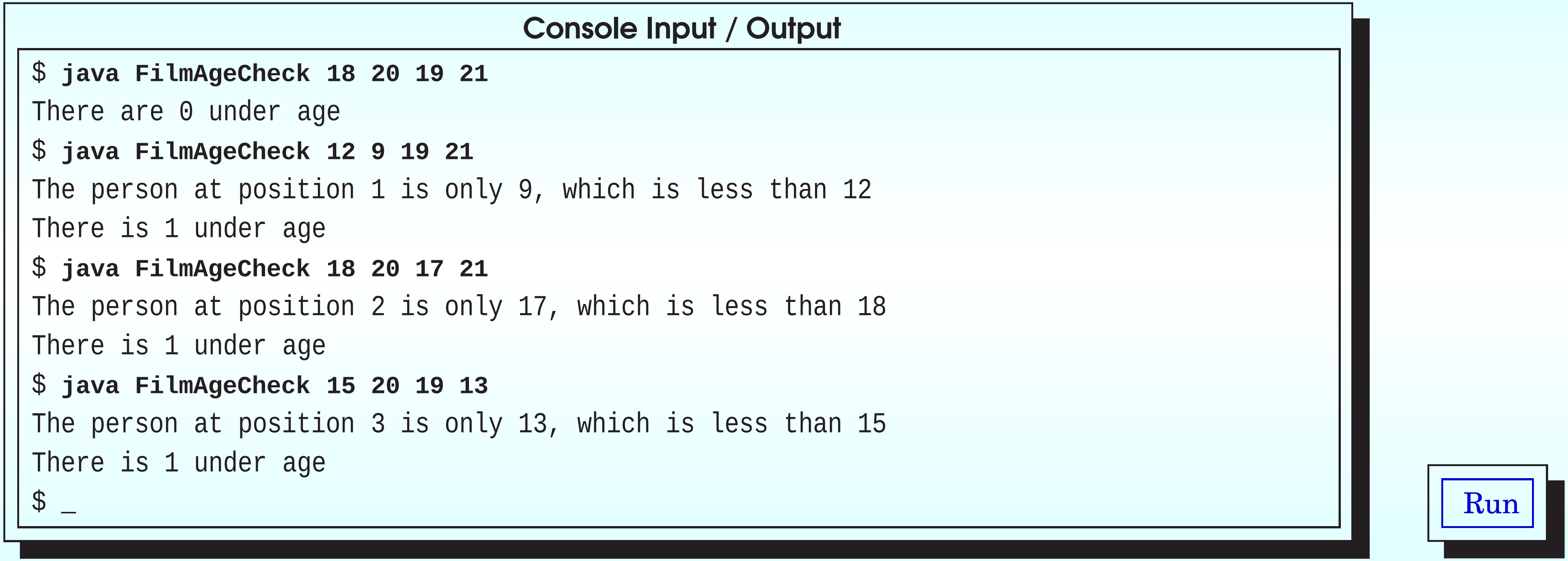
Run

Console Input / Output

```
$ java FilmAgeCheck 15 15
There are 0 under age
$ java FilmAgeCheck 12 10
The person at position 1 is only 10, which is less than 12
There is 1 under age
$ java FilmAgeCheck 18 19
There are 0 under age
$ _
```

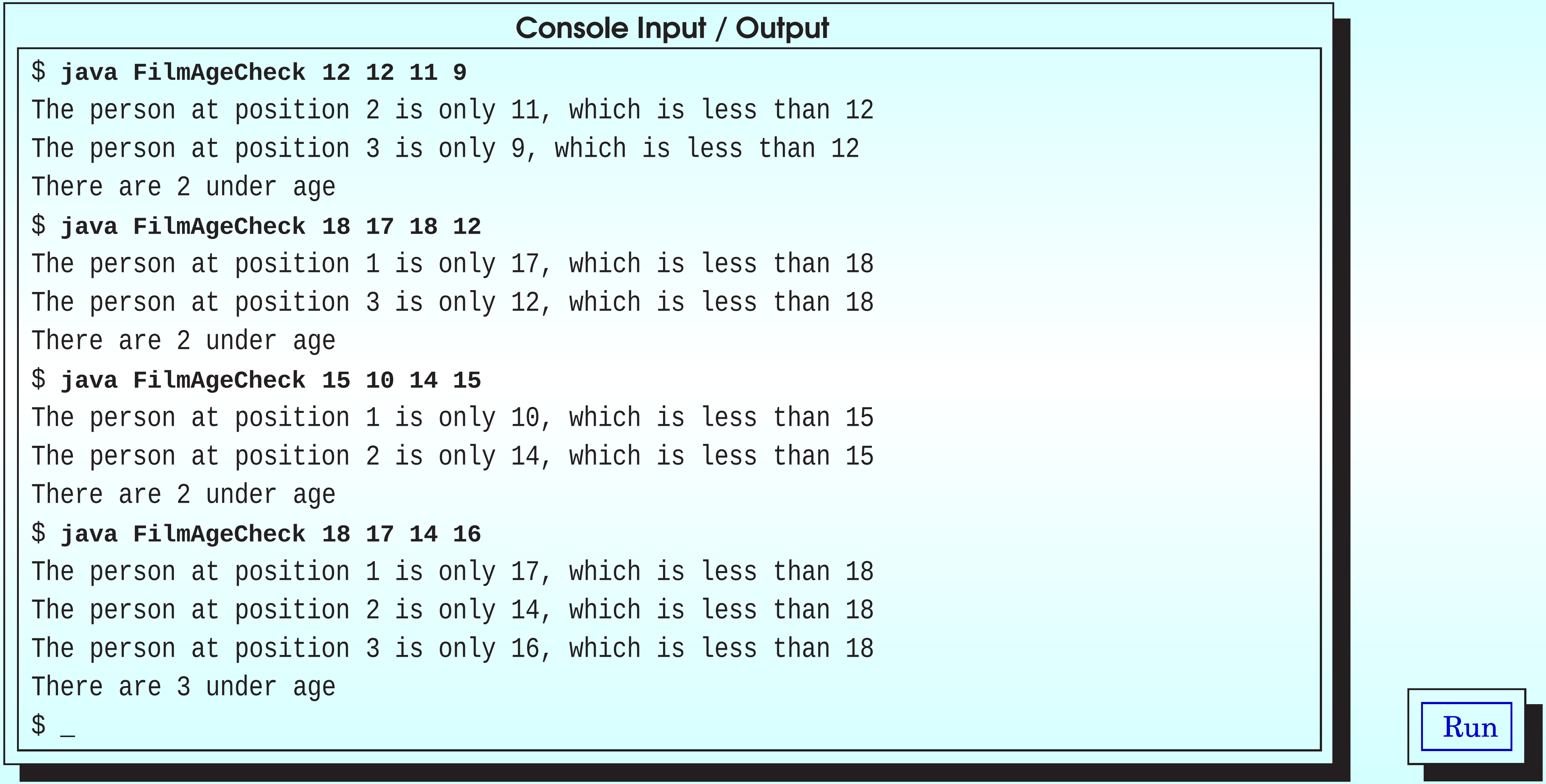
Run

Trying it



```
$ java FilmAgeCheck 18 20 19 21
There are 0 under age
$ java FilmAgeCheck 12 9 19 21
The person at position 1 is only 9, which is less than 12
There is 1 under age
$ java FilmAgeCheck 18 20 17 21
The person at position 2 is only 17, which is less than 18
There is 1 under age
$ java FilmAgeCheck 15 20 19 13
The person at position 3 is only 13, which is less than 15
There is 1 under age
$ _
```

Trying it



```
$ java FilmAgeCheck 12 12 11 9
The person at position 2 is only 11, which is less than 12
The person at position 3 is only 9, which is less than 12
There are 2 under age
$ java FilmAgeCheck 18 17 18 12
The person at position 1 is only 17, which is less than 18
The person at position 3 is only 12, which is less than 18
There are 2 under age
$ java FilmAgeCheck 15 10 14 15
The person at position 1 is only 10, which is less than 15
The person at position 2 is only 14, which is less than 15
There are 2 under age
$ java FilmAgeCheck 18 17 14 16
The person at position 1 is only 17, which is less than 18
The person at position 2 is only 14, which is less than 18
The person at position 3 is only 16, which is less than 18
There are 3 under age
$ _
```

Trying it

Coffee time: Was that a good set of tests? Are there significant combinations of input conditions that have been overlooked, or would you trust the program now?

Coursework: `MaxList`

(Summary only)

Write a program to find the maximum of a given **list** of numbers.

Section 3

Example:
Dividing a cake (GCD)

Aim

AIM: To introduce the idea of nesting an **if else statement** within a **while loop**.

Dividing a cake (GCD)

- Two sisters, same birthday, different ages.
- Squabbling over who has how much birthday cake.
- Mum says “divide cake into smallest number of equal sized pieces
  - so can each have a number of pieces proportional to her age.”
- Key to solution: find **greatest common divisor** of two ages.

Dividing a cake (GCD)

- Repeatedly subtract smallest from largest until they are **equal**.
  - Both ages are a multiple of their GCD,
    - and so is their difference.
- E.g.

$$\begin{aligned} & GCD(25,20) \\ &= GCD(25-20,20) && = GCD(5,20) \\ &= GCD(5,20-5) && = GCD(5,15) \\ &= GCD(5,15-5) && = GCD(5,10) \\ &= GCD(5,10-5) && = GCD(5,5) \\ &= 5 \end{aligned}$$

Dividing a cake (GCD)

```java
// Program to decide how to divide a cake in proportion to the age of two
// persons, using the minimum number of equal sized portions.
// The two arguments are the two positive integer ages.
public class DivideCake
{
  public static void main(String[] args)
  {
    // Both ages must be positive.
    // First person’s age.
    int age1 = Integer.parseInt(args[0]);
    // Second person’s age.
    int age2 = Integer.parseInt(args[1]);

```

Dividing a cake (GCD)

```java
// This is a multiple of the GCD, initially age1.
int multiple1OfGCD = age1;
// This is a multiple of the GCD, initially age2.
int multiple2OfGCD = age2;

// Compute the GCD of multiple1OfGCD and multiple2OfGCD.
// While the two multiples are not the same, the difference
// between them must also be a multiple of the GCD.

// E.g. X = x * d, Y = y * d, (X - Y) = (x - y) * d

```

Dividing a cake (GCD)

```java
    // So we keep subtracting the smallest from the largest
    // until they are equal.
    while (multiple1OfGCD != multiple2OfGCD)
      if (multiple1OfGCD > multiple2OfGCD)
        multiple1OfGCD -= multiple2OfGCD;
      else
        multiple2OfGCD -= multiple1OfGCD;

```

Dividing a cake (GCD)

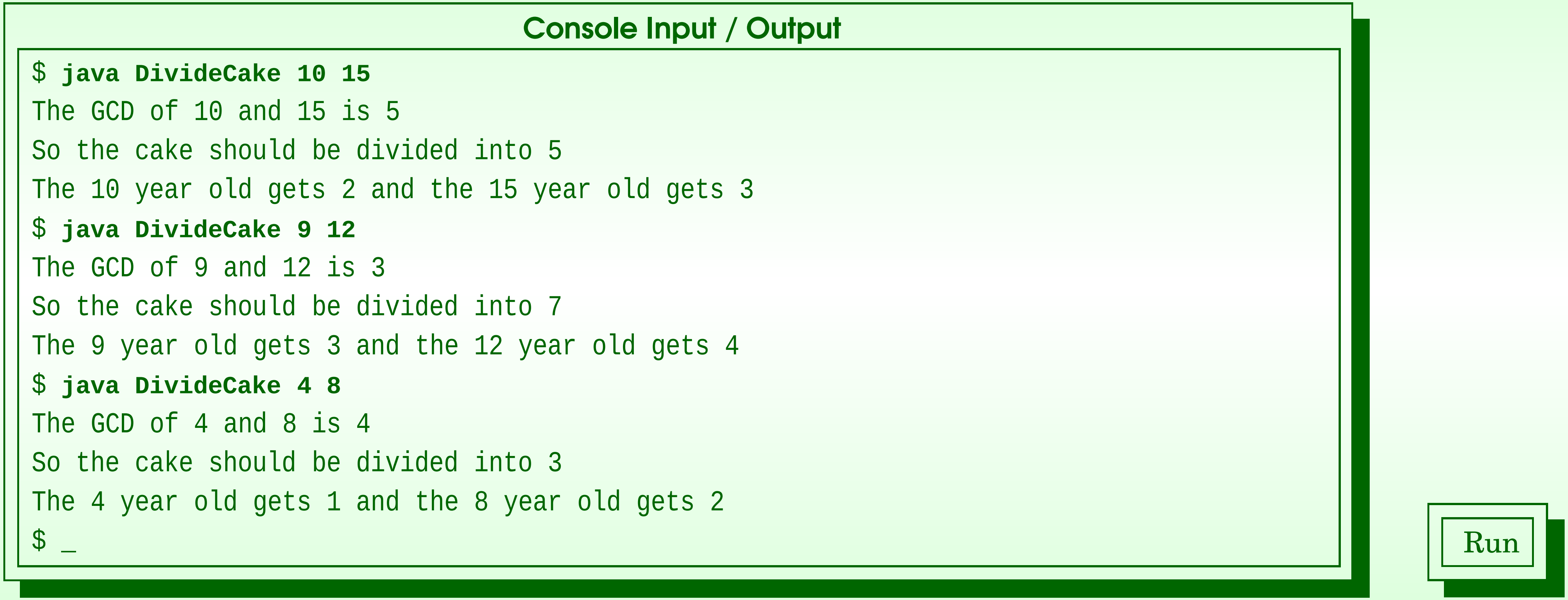
```
// Now multiple1OfGCD == multiple2OfGCD
// which is also the GCD of age1 and age2.
System.out.println("The GCD of " + age1 + " and " + age2
                   + " is " + multiple1OfGCD);

// Calculate the number of portions for each person.
int noOfPortions1 = age1 / multiple1OfGCD;
int noOfPortions2 = age2 / multiple1OfGCD;

// Report the total number of portions.
System.out.println("So the cake should be divided into "
                   + (noOfPortions1 + noOfPortions2));

```

Dividing a cake (GCD)

```
  // Report the number of portions for each person.
  System.out.println
   ("The " + age1 + " year old gets " + noOfPortions1
    + " and the " + age2 + " year old gets " + noOfPortions2);
 } // main

} // class DivideCake
```

Coffee time: Why did we need to put brackets around `noOfPortions1 + noOfPortions2`? (Hint: `+` has **left associativity** and has the same precedence when used as **concatenation** that it does when used as **addition**.)

Trying it



Console Input / Output

```
$ java DivideCake 10 15
The GCD of 10 and 15 is 5
So the cake should be divided into 5
The 10 year old gets 2 and the 15 year old gets 3
$ java DivideCake 9 12
The GCD of 9 and 12 is 3
So the cake should be divided into 7
The 9 year old gets 3 and the 12 year old gets 4
$ java DivideCake 4 8
The GCD of 4 and 8 is 4
So the cake should be divided into 3
The 4 year old gets 1 and the 8 year old gets 2
$ _
```

Coursework: `DivideCake3`

(Summary only)

Write a program to compute the **greatest common divisor** of three numbers.

Section 4

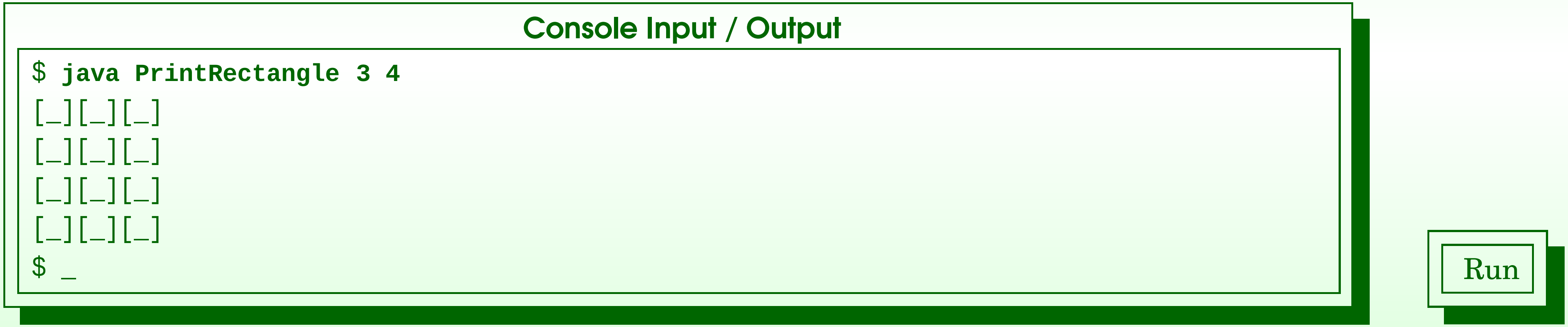
Example:
Printing a rectangle

Aim

AIM: To introduce the idea of nesting a **for loop** within a **for loop**. We also meet `System.out.print()` and revisit `System.out.println()`.

Printing a rectangle

- Print an `**ASCII** art' rectangle
  - made up of cells "[_]".



Console Input / Output

```
$ java PrintRectangle 3 4
[_][_][_]
[_][_][_]
[_][_][_]
[_][_][_]
$ _
```

Printing a rectangle

- Have a **loop nested** inside another loop
  - outer one deals with lines, **execute**s height times
    - inner one deals with cells, **execute**s width times.
- Some **pseudo code**:

```
get width and height
for row = 1 to height
  for column = 1 to width
    output a cell with no new line
  output a new line
end-for
```

- Inner code executed height * width times.

Standard API: `System`: `out.print()`

- Another **method** in `System`: `out.print()`
  - same as `out.println()` except no **new line**.
- E.g.

```
System.out.print("Hello");
System.out.print(" ");
System.out.println("world!");
```

  same effect as:

```
System.out.println("Hello world!");
```

- Most useful for output generated in a **loop**.

Standard API: System: out.println(): with no argument

- System also has another version of out.println() with no arguments.
  - prints only a **new line**.

```
System.out.println();
```

same effect as:

```
System.out.println("");
```

- E.g.

```
System.out.print("Hello world!");
System.out.println();
```

same effect as:

```
System.out.println("Hello world!");
```

- Useful for ending a line generated by a loop, or when want blank line.

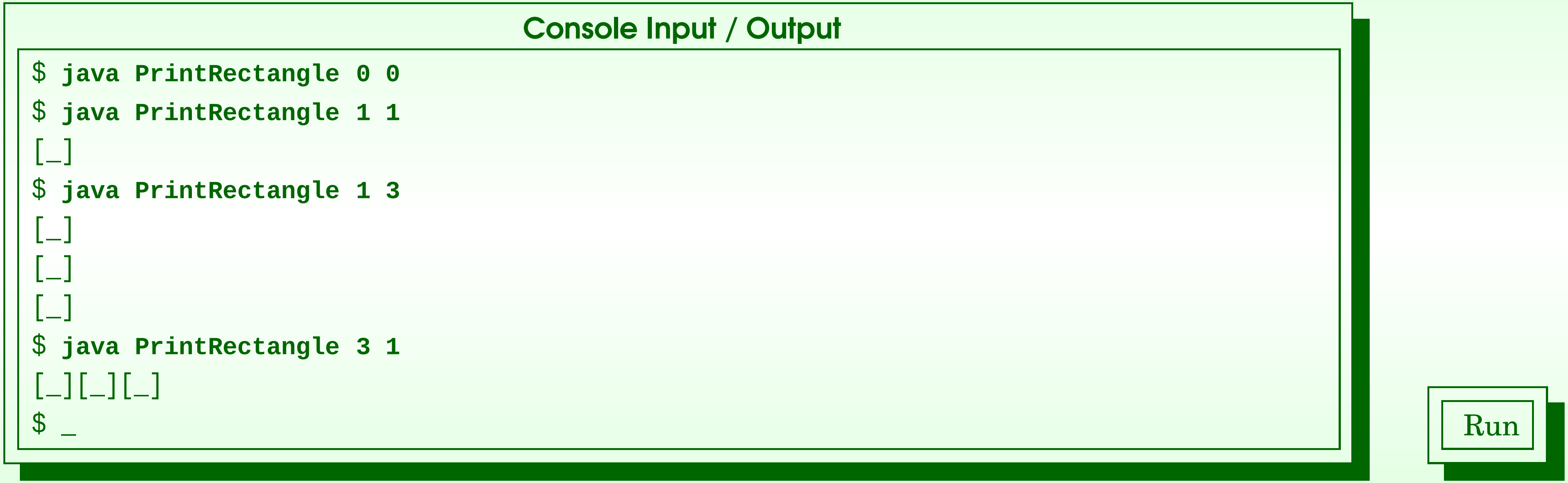
Printing a rectangle

```java
// Program to print out a rectangle.
// The width and then the height are given as arguments.
// We assume the arguments represent positive integers.
public class PrintRectangle
{
  public static void main(String[] args)
  {
    // The width of the rectangle, in cells.
    int width = Integer.parseInt(args[0]);
    // The height of the rectangle, in cells.
    int height = Integer.parseInt(args[1]);

```

Printing a rectangle

```
  // Print out height number of rows.
  for (int row = 1; row <= height; row++)
  {
    // Print out width number of cells, on the same line.
    for (int column = 1; column <= width; column++)
      System.out.print("[_]");
    // End the line.
    System.out.println();
  } // for
 } // main

} // class PrintRectangle
```

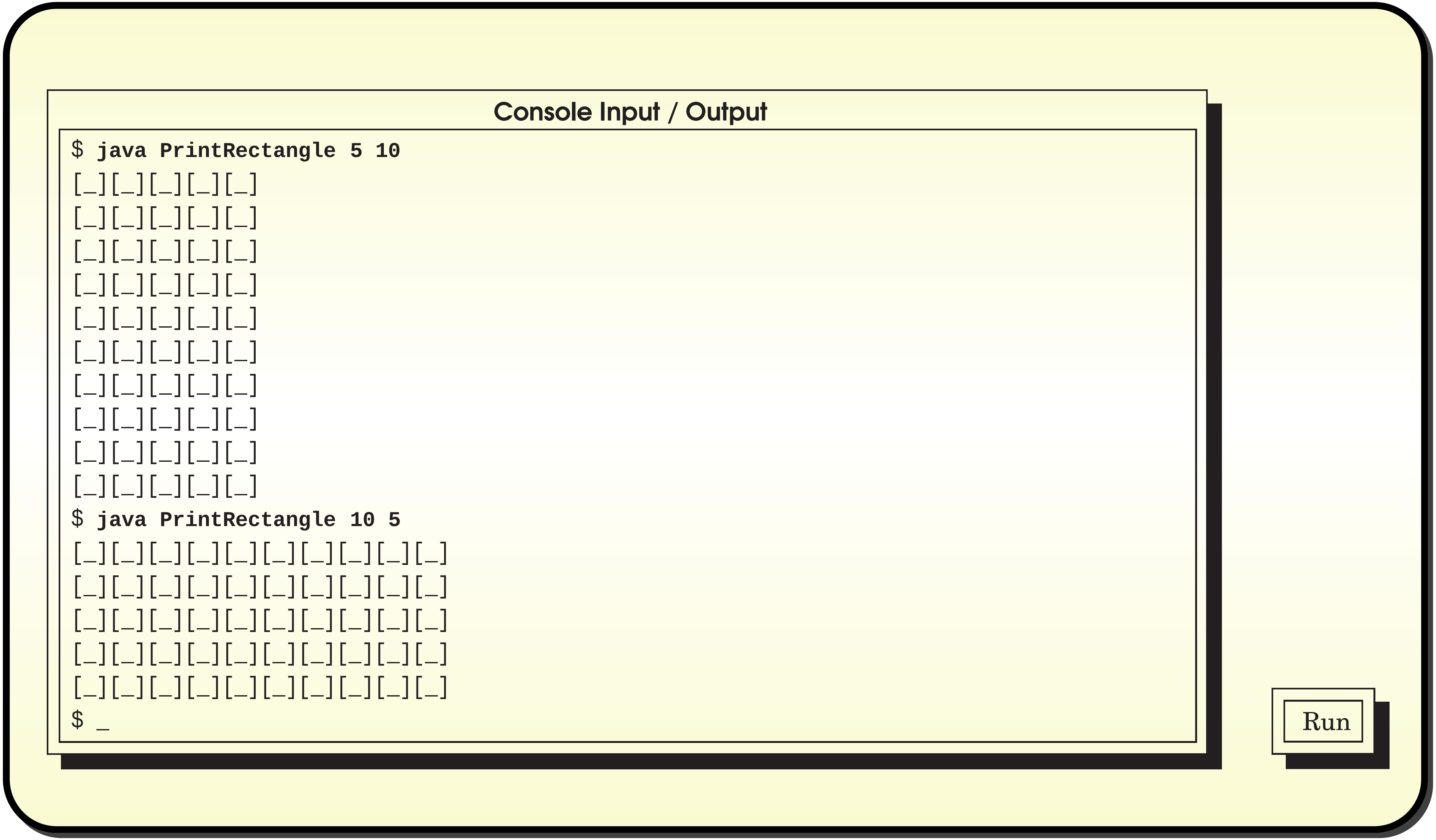
Trying it



Console Input / Output

```
$ java PrintRectangle 0 0
$ java PrintRectangle 1 1
[_]
$ java PrintRectangle 1 3
[_]
[_]
[_]
$ java PrintRectangle 3 1
[_][_][_]
$ _
```

Trying it



```
$ java PrintRectangle 5 10
[_][_][_][_][_]
[_][_][_][_][_]
[_][_][_][_][_]
[_][_][_][_][_]
[_][_][_][_][_]
[_][_][_][_][_]
[_][_][_][_][_]
[_][_][_][_][_]
[_][_][_][_][_]
[_][_][_][_][_]
$ java PrintRectangle 10 5
[_][_][_][_][_][_][_][_][_][_]
[_][_][_][_][_][_][_][_][_][_]
[_][_][_][_][_][_][_][_][_][_]
[_][_][_][_][_][_][_][_][_][_]
[_][_][_][_][_][_][_][_][_][_]
$ _
```

Coursework: `PrintHoledRectangle`

(Summary only)

Write a program to print out a rectangle with a hole in it.

Section 5

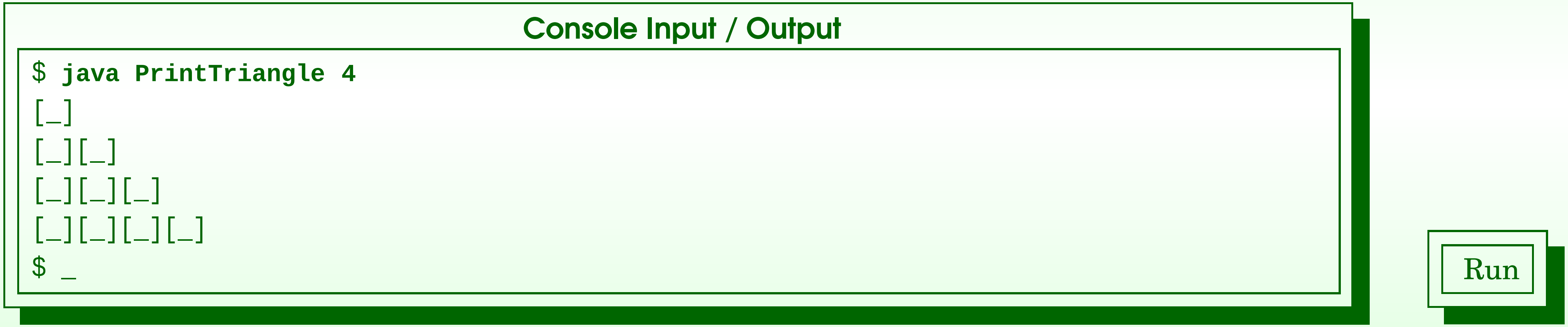
Example:
Printing a triangle

Aim

AIM: To reinforce the idea of nesting a **for loop** within a **for loop**.

Printing a triangle

- Similar to previous program.



Console Input / Output

```
$ java PrintTriangle 4
[_]
[_][_]
[_][_][_]
[_][_][_][_]
$ _
```

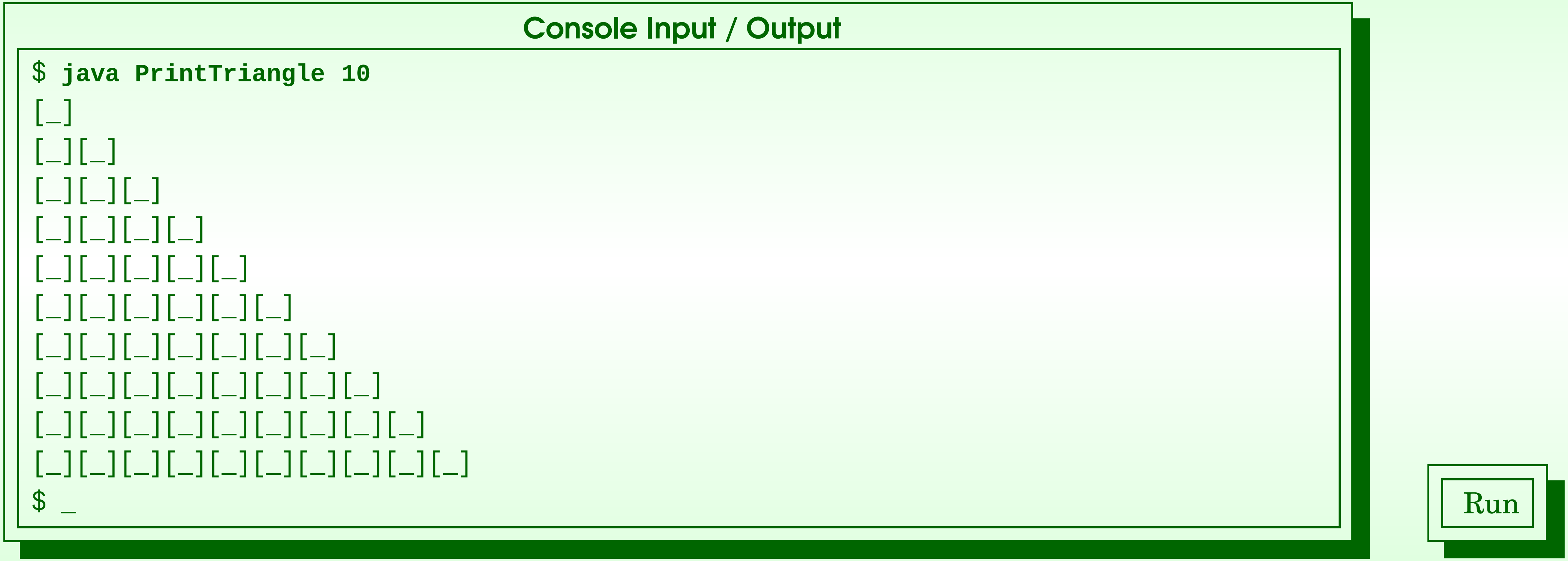
Printing a triangle

```java
// Program to print out an isosceles right angled triangle.
// The height (which is also the width) is given as an argument.
// We assume the argument represents a positive integer.
public class PrintTriangle
{
  public static void main(String[] args)
  {
    // The height (also the width) of the triangle.
    int height = Integer.parseInt(args[0]);

```

Printing a triangle

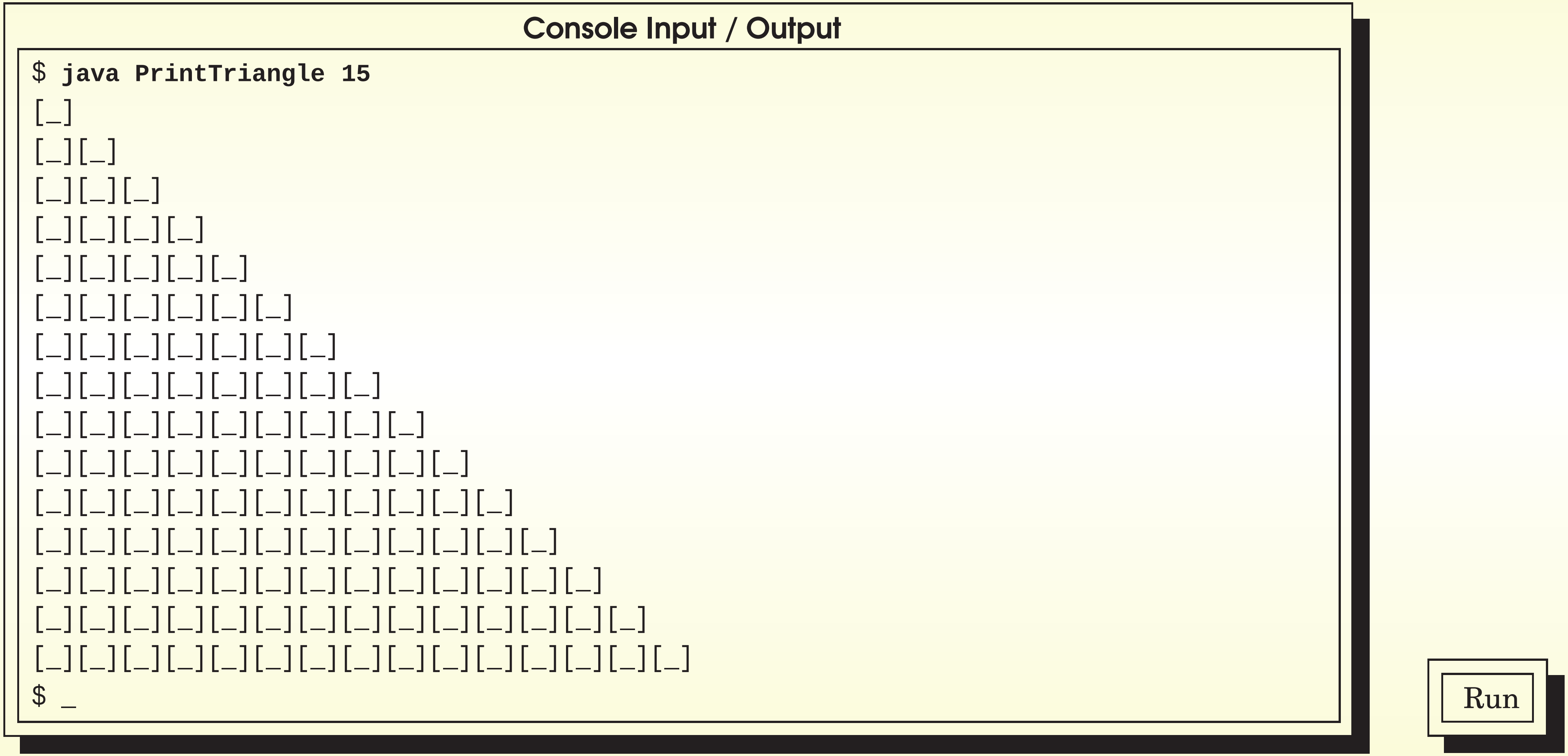
```
    // Print out height number of rows.
    for (int row = 1; row <= height; row++)
    {
      // Print out row number of cells, on the same line.
      for (int column = 1; column <= row; column++)
        System.out.print("[_]");
      // End the line.
      System.out.println();
    } // for
  } // main

} // class PrintTriangle
```

Trying it



```
$ java PrintTriangle 10
[_]
[_][_]
[_][_][_]
[_][_][_][_]
[_][_][_][_][_]
[_][_][_][_][_][_]
[_][_][_][_][_][_][_]
[_][_][_][_][_][_][_][_]
[_][_][_][_][_][_][_][_][_]
[_][_][_][_][_][_][_][_][_][_]
$ _
```

Trying it



```
$ java PrintTriangle 15
[_]
[_][_]
[_][_][_]
[_][_][_][_]
[_][_][_][_][_]
[_][_][_][_][_][_]
[_][_][_][_][_][_][_]
[_][_][_][_][_][_][_][_]
[_][_][_][_][_][_][_][_][_]
[_][_][_][_][_][_][_][_][_][_]
[_][_][_][_][_][_][_][_][_][_][_]
[_][_][_][_][_][_][_][_][_][_][_][_]
[_][_][_][_][_][_][_][_][_][_][_][_][_]
[_][_][_][_][_][_][_][_][_][_][_][_][_][_]
[_][_][_][_][_][_][_][_][_][_][_][_][_][_][_]
$ _
```

Trying it

Coffee time: What would happen if we changed the outer **for loop** to the following?

```
for (int row = 0; row < height; row++)
```

Coffee time: What would happen if we changed the inner **for loop** to the following?

```
for (int column = 1; column <= height - row + 1; column++)
```

Coursework: `PrintTriangleMirror`

(Summary only)

Write a program to print out an isosceles right angled triangle, with a straight right edge, and the longest side at the top.

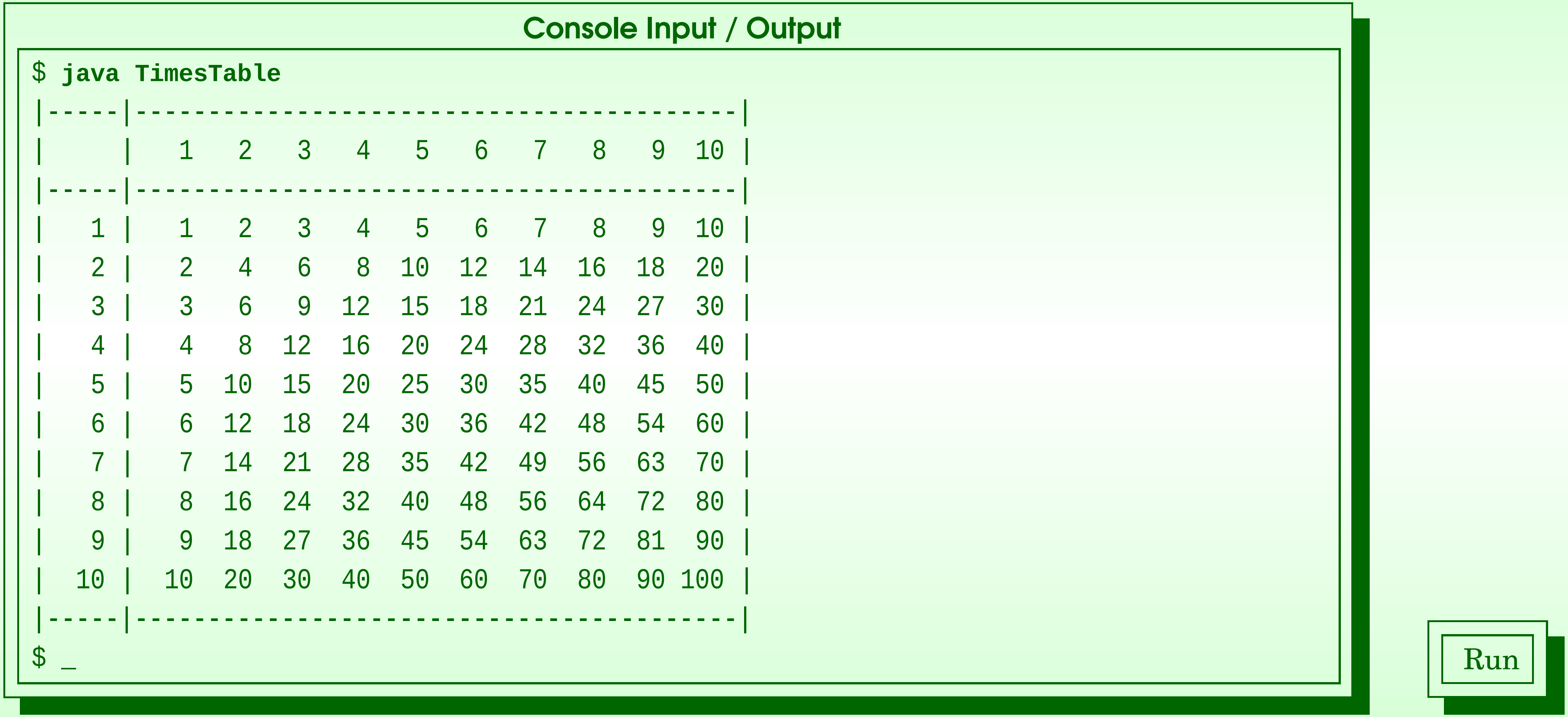
Section 6

Example: Multiple times table

Aim

AIM: To reinforce the idea of having **nested statements** within each other, and explore the idea of using multiple **loops** in sequence.

Multiple times table



Console Input / Output

```
$ java TimesTable
|-----|-----------------------------------------|
|     |   1   2   3   4   5   6   7   8   9  10 |
|-----|-----------------------------------------|
|   1 |   1   2   3   4   5   6   7   8   9  10 |
|   2 |   2   4   6   8  10  12  14  16  18  20 |
|   3 |   3   6   9  12  15  18  21  24  27  30 |
|   4 |   4   8  12  16  20  24  28  32  36  40 |
|   5 |   5  10  15  20  25  30  35  40  45  50 |
|   6 |   6  12  18  24  30  36  42  48  54  60 |
|   7 |   7  14  21  28  35  42  49  56  63  70 |
|   8 |   8  16  24  32  40  48  56  64  72  80 |
|   9 |   9  18  27  36  45  54  63  72  81  90 |
|  10 |  10  20  30  40  50  60  70  80  90 100 |
|-----|-----------------------------------------|
$ _
```

Multiple times table

- Initial **pseudo code**:

```
print the box top line
print column headings
print headings underline
for row = 1 to 10
  print a row
print the box bottom line
```

Multiple times table

- Second draft:

```
print the box top line
print column headings
print headings underline
for row = 1 to 10
  print box left side
  print row label
  print separator
  for column = 1 to 10
    print row * column
  print box right side and new line
end-for
print the box bottom line
```

Multiple times table

- More drafts, adding more detail. . . .

Multiple times table

```java
// Program to print out a neat 10 by 10 multiplication table.
public class TimesTable
{
  public static void main(String[] args)
  {
    // Top line.
    // Left side, 5 characters for row labels, separator.
    System.out.print("|-----|");
    // Above the column headings.
    for (int column = 1; column <= 10; column++)
      // 4 characters for each column.
      System.out.print("----");
    // The right side.
    System.out.println("-|");

```

Multiple times table

```
// Column headings.
System.out.print("|     |");
for (int column = 1; column <= 10; column++)
  // Need to make column number always occupy 4 characters.
  if (column < 10)
    System.out.print("   " + column);
  else
    System.out.print("  " + column);
System.out.println(" |");

// Underline headings -- same as Top line.
System.out.print("|-----|");
for (int column = 1; column <= 10; column++)
  System.out.print("----");
System.out.println("-|");
```

Multiple times table

```java

    // Now the rows.
    for (int row = 1; row <= 10; row++)
    {
      // Need to make row number always occupy 7 characters
      // including vertical lines.
      if (row < 10)
        System.out.print("|   " + row + " |");
      else
        System.out.print("|  " + row + " |");

```

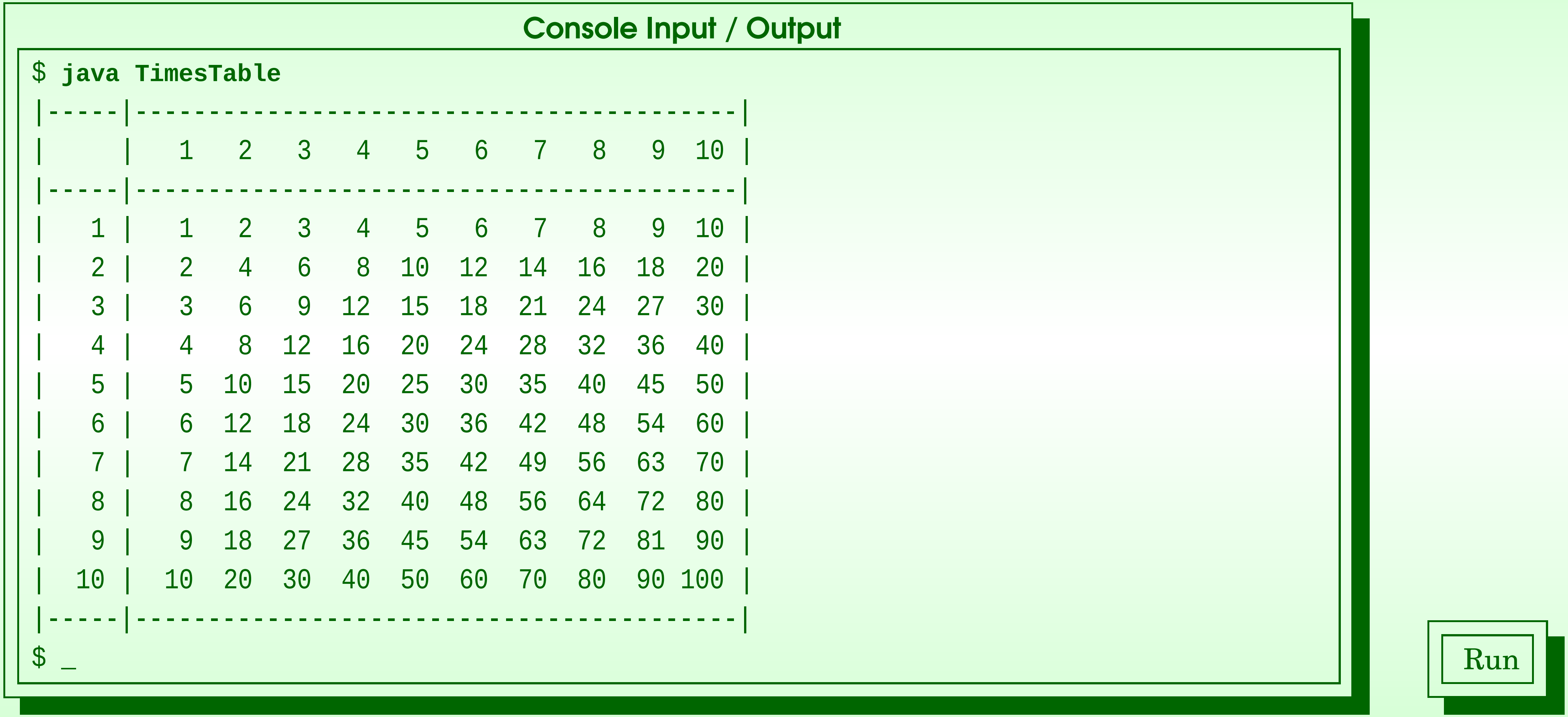
Multiple times table

```java
      // Now the columns on this row.
      for (int column = 1; column <= 10; column++)
      {
        int product = row * column;
        // Need to make product always occupy 4 characters.
        if (product < 10)
          System.out.print("   " + product);
        else if (product < 100)
          System.out.print("  " + product);
        else
          System.out.print(" " + product);
      } // for

```

Multiple times table

```
   // The right side.
   System.out.println(" |");
  } // for

  // Bottom line -- same as Top line.
  System.out.print("|-----|");
  for (int column = 1; column <= 10; column++)
   System.out.print("----");
  System.out.println("-|");
 } // main

} // class TimesTable
```

Trying it



```
$ java TimesTable
|-----|-----------------------------------------|
|     |   1   2   3   4   5   6   7   8   9  10 |
|-----|-----------------------------------------|
|   1 |   1   2   3   4   5   6   7   8   9  10 |
|   2 |   2   4   6   8  10  12  14  16  18  20 |
|   3 |   3   6   9  12  15  18  21  24  27  30 |
|   4 |   4   8  12  16  20  24  28  32  36  40 |
|   5 |   5  10  15  20  25  30  35  40  45  50 |
|   6 |   6  12  18  24  30  36  42  48  54  60 |
|   7 |   7  14  21  28  35  42  49  56  63  70 |
|   8 |   8  16  24  32  40  48  56  64  72  80 |
|   9 |   9  18  27  36  45  54  63  72  81  90 |
|  10 |  10  20  30  40  50  60  70  80  90 100 |
|-----|-----------------------------------------|
$ _
```

Trying it

Coffee time: What if when we show the output to the end user, perhaps a primary school teacher, she tells us she wanted a 12 by 12 table? What changes would we have to make to our program? What would happen if we just changed every `10` to `12`? Is there something we could have done to make the program more flexible in this respect?

Coursework: `CommonFactorsTable`

(Summary only)

Write a program to produce a table showing pairs of numbers which share common factors.

Section 7

Example:
Luck is in the air: dice combinations

Aim

AIM: To introduce the idea of using **nested loops** to generate combinations.

Luck is in the air: dice combinations

- Designing board game
  - players throw 3 dice per move.
- Want chart of all values obtainable from three dice
  - all combinations of 3 dice.

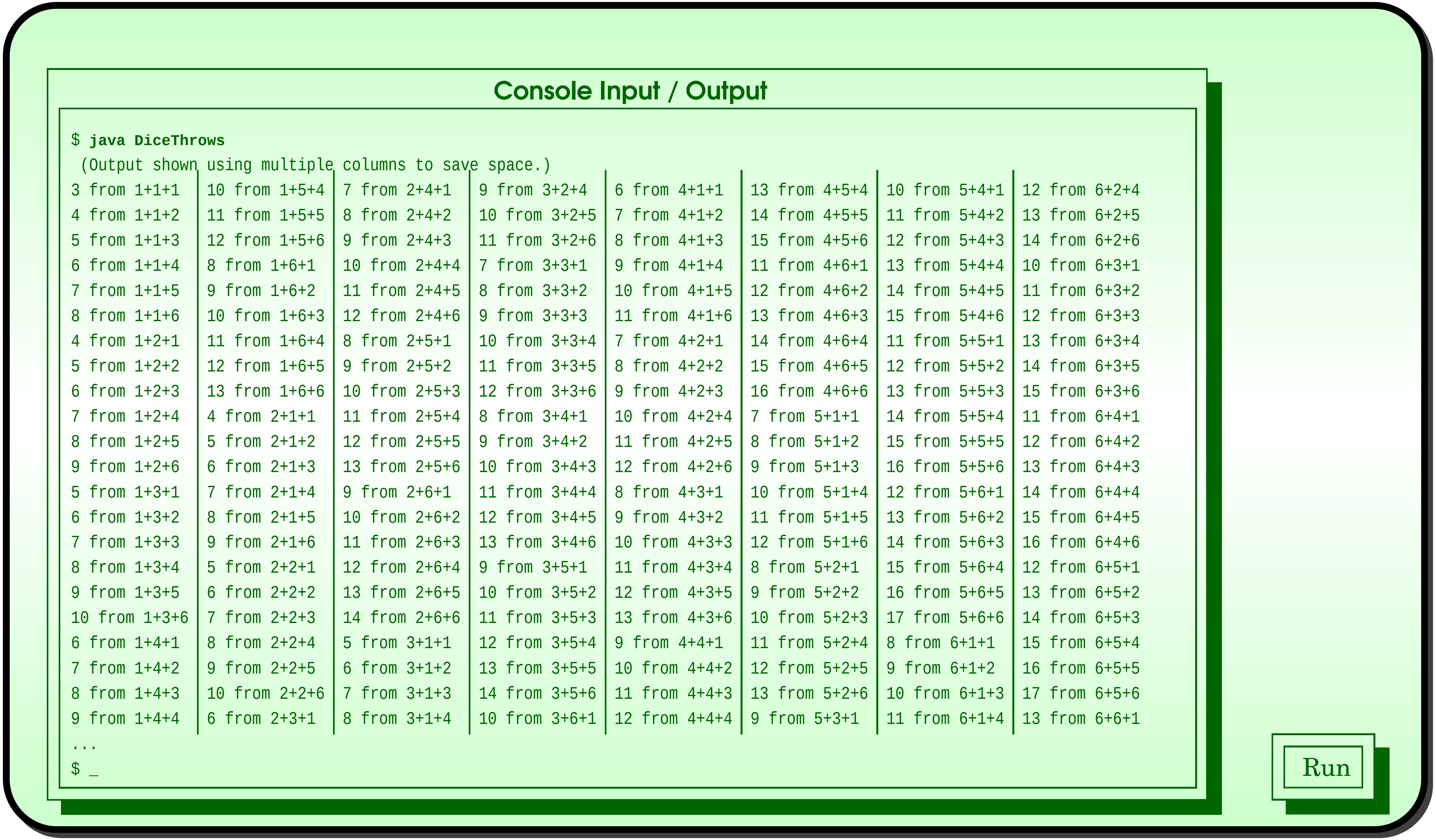
Luck is in the air: dice combinations

```
// Program to output all 216 combinations of three six sided dice.
public class DiceThrows
{
  public static void main(String[] args)
  {
    // Nested loops produce all 216 combinations of die1 to die3.
    for (int die1 = 1; die1 <= 6; die1++)
      for (int die2 = 1; die2 <= 6; die2++)
        for (int die3 = 1; die3 <= 6; die3++)
          System.out.println(die1 + die2 + die3 + " from "
                             + die1 + "+" + die2 + "+" + die3);
  } // main

} // class DiceThrows
```

Luck is in the air: dice combinations

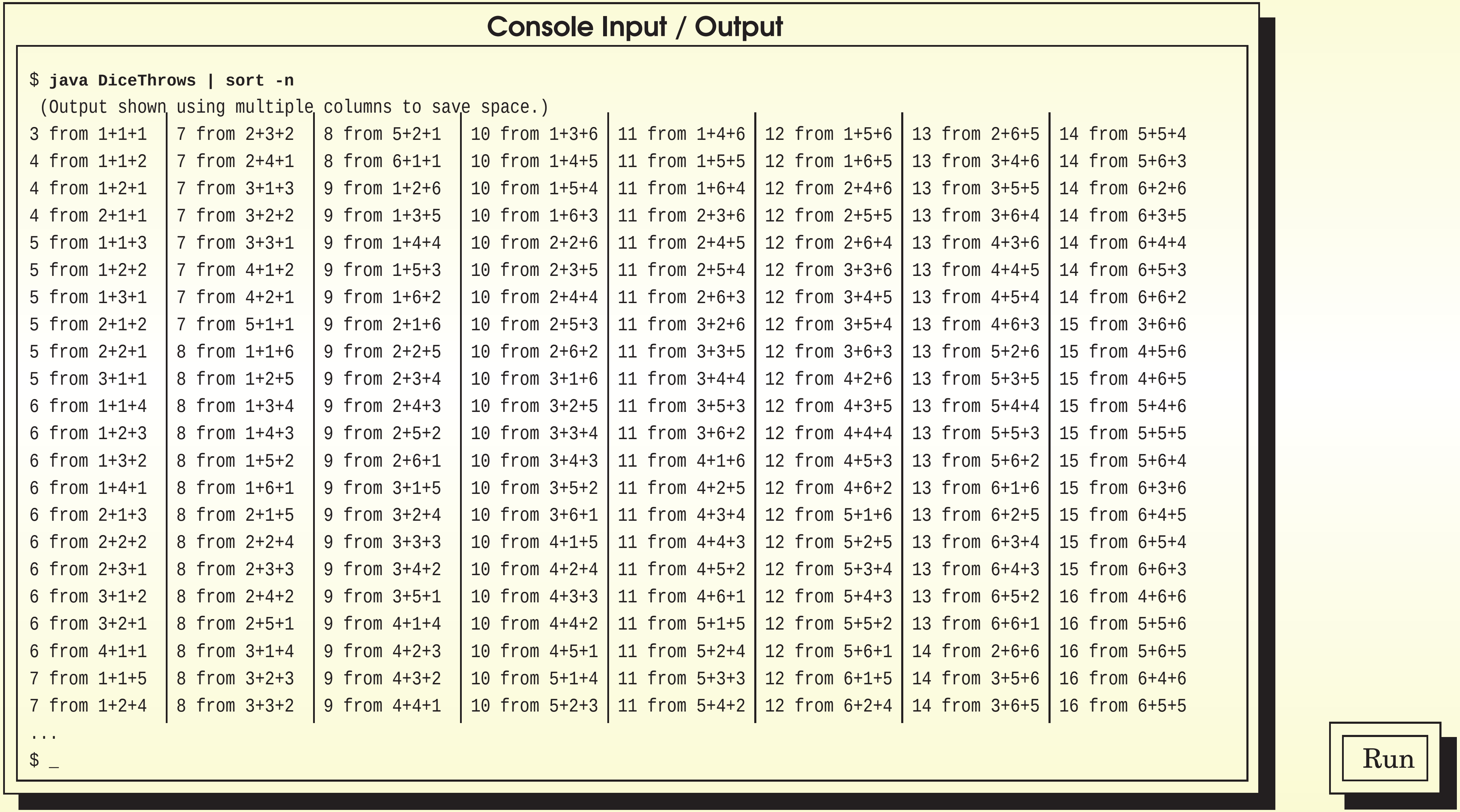
Coffee time: Why do we *not* need brackets around `die1 + die2 + die3` to avoid the `+` **operator** being interpreted as **concatenation**?

Trying it



```
$ java DiceThrows
 (Output shown using multiple columns to save space.)
3 from 1+1+1  | 10 from 1+5+4 | 7 from 2+4+1  | 9 from 3+2+4  | 6 from 4+1+1  | 13 from 4+5+4 | 10 from 5+4+1 | 12 from 6+2+4
4 from 1+1+2  | 11 from 1+5+5 | 8 from 2+4+2  | 10 from 3+2+5 | 7 from 4+1+2  | 14 from 4+5+5 | 11 from 5+4+2 | 13 from 6+2+5
5 from 1+1+3  | 12 from 1+5+6 | 9 from 2+4+3  | 11 from 3+2+6 | 8 from 4+1+3  | 15 from 4+5+6 | 12 from 5+4+3 | 14 from 6+2+6
6 from 1+1+4  | 8 from 1+6+1  | 10 from 2+4+4 | 7 from 3+3+1  | 9 from 4+1+4  | 11 from 4+6+1 | 13 from 5+4+4 | 10 from 6+3+1
7 from 1+1+5  | 9 from 1+6+2  | 11 from 2+4+5 | 8 from 3+3+2  | 10 from 4+1+5 | 12 from 4+6+2 | 14 from 5+4+5 | 11 from 6+3+2
8 from 1+1+6  | 10 from 1+6+3 | 12 from 2+4+6 | 9 from 3+3+3  | 11 from 4+1+6 | 13 from 4+6+3 | 15 from 5+4+6 | 12 from 6+3+3
4 from 1+2+1  | 11 from 1+6+4 | 8 from 2+5+1  | 10 from 3+3+4 | 7 from 4+2+1  | 14 from 4+6+4 | 11 from 5+5+1 | 13 from 6+3+4
5 from 1+2+2  | 12 from 1+6+5 | 9 from 2+5+2  | 11 from 3+3+5 | 8 from 4+2+2  | 15 from 4+6+5 | 12 from 5+5+2 | 14 from 6+3+5
6 from 1+2+3  | 13 from 1+6+6 | 10 from 2+5+3 | 12 from 3+3+6 | 9 from 4+2+3  | 16 from 4+6+6 | 13 from 5+5+3 | 15 from 6+3+6
7 from 1+2+4  | 4 from 2+1+1  | 11 from 2+5+4 | 8 from 3+4+1  | 10 from 4+2+4 | 7 from 5+1+1  | 14 from 5+5+4 | 11 from 6+4+1
8 from 1+2+5  | 5 from 2+1+2  | 12 from 2+5+5 | 9 from 3+4+2  | 11 from 4+2+5 | 8 from 5+1+2  | 15 from 5+5+5 | 12 from 6+4+2
9 from 1+2+6  | 6 from 2+1+3  | 13 from 2+5+6 | 10 from 3+4+3 | 12 from 4+2+6 | 9 from 5+1+3  | 16 from 5+5+6 | 13 from 6+4+3
5 from 1+3+1  | 7 from 2+1+4  | 9 from 2+6+1  | 11 from 3+4+4 | 8 from 4+3+1  | 10 from 5+1+4 | 12 from 5+6+1 | 14 from 6+4+4
6 from 1+3+2  | 8 from 2+1+5  | 10 from 2+6+2 | 12 from 3+4+5 | 9 from 4+3+2  | 11 from 5+1+5 | 13 from 5+6+2 | 15 from 6+4+5
7 from 1+3+3  | 9 from 2+1+6  | 11 from 2+6+3 | 13 from 3+4+6 | 10 from 4+3+3 | 12 from 5+1+6 | 14 from 5+6+3 | 16 from 6+4+6
8 from 1+3+4  | 5 from 2+2+1  | 12 from 2+6+4 | 9 from 3+5+1  | 11 from 4+3+4 | 8 from 5+2+1  | 15 from 5+6+4 | 12 from 6+5+1
9 from 1+3+5  | 6 from 2+2+2  | 13 from 2+6+5 | 10 from 3+5+2 | 12 from 4+3+5 | 9 from 5+2+2  | 16 from 5+6+5 | 13 from 6+5+2
10 from 1+3+6 | 7 from 2+2+3  | 14 from 2+6+6 | 11 from 3+5+3 | 13 from 4+3+6 | 10 from 5+2+3 | 17 from 5+6+6 | 14 from 6+5+3
6 from 1+4+1  | 8 from 2+2+4  | 5 from 3+1+1  | 12 from 3+5+4 | 9 from 4+4+1  | 11 from 5+2+4 | 8 from 6+1+1  | 15 from 6+5+4
7 from 1+4+2  | 9 from 2+2+5  | 6 from 3+1+2  | 13 from 3+5+5 | 10 from 4+4+2 | 12 from 5+2+5 | 9 from 6+1+2  | 16 from 6+5+5
8 from 1+4+3  | 10 from 2+2+6 | 7 from 3+1+3  | 14 from 3+5+6 | 11 from 4+4+3 | 13 from 5+2+6 | 10 from 6+1+3 | 17 from 6+5+6
9 from 1+4+4  | 6 from 2+3+1  | 8 from 3+1+4  | 10 from 3+6+1 | 12 from 4+4+4 | 9 from 5+3+1  | 11 from 6+1+4 | 13 from 6+6+1
...
$ _
```

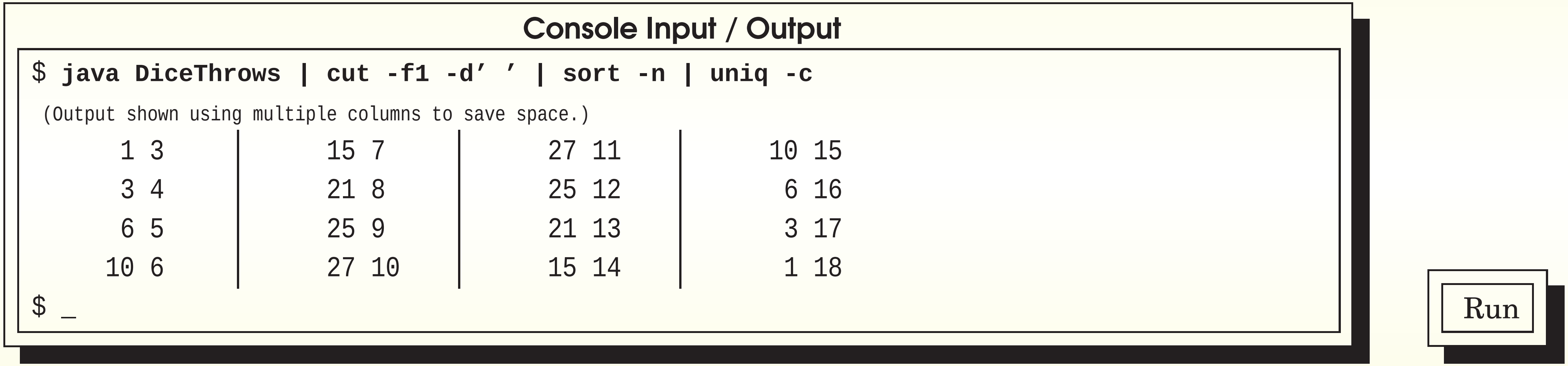
Trying it



Console Input / Output

```
$ java DiceThrows | sort -n
 (Output shown using multiple columns to save space.)
3 from 1+1+1 │ 7 from 2+3+2 │ 8 from 5+2+1 │ 10 from 1+3+6 │ 11 from 1+4+6 │ 12 from 1+5+6 │ 13 from 2+6+5 │ 14 from 5+5+4
4 from 1+1+2 │ 7 from 2+4+1 │ 8 from 6+1+1 │ 10 from 1+4+5 │ 11 from 1+5+5 │ 12 from 1+6+5 │ 13 from 3+4+6 │ 14 from 5+6+3
4 from 1+2+1 │ 7 from 3+1+3 │ 9 from 1+2+6 │ 10 from 1+5+4 │ 11 from 1+6+4 │ 12 from 2+4+6 │ 13 from 3+5+5 │ 14 from 6+2+6
4 from 2+1+1 │ 7 from 3+2+2 │ 9 from 1+3+5 │ 10 from 1+6+3 │ 11 from 2+3+6 │ 12 from 2+5+5 │ 13 from 3+6+4 │ 14 from 6+3+5
5 from 1+1+3 │ 7 from 3+3+1 │ 9 from 1+4+4 │ 10 from 2+2+6 │ 11 from 2+4+5 │ 12 from 2+6+4 │ 13 from 4+3+6 │ 14 from 6+4+4
5 from 1+2+2 │ 7 from 4+1+2 │ 9 from 1+5+3 │ 10 from 2+3+5 │ 11 from 2+5+4 │ 12 from 3+3+6 │ 13 from 4+4+5 │ 14 from 6+5+3
5 from 1+3+1 │ 7 from 4+2+1 │ 9 from 1+6+2 │ 10 from 2+4+4 │ 11 from 2+6+3 │ 12 from 3+4+5 │ 13 from 4+5+4 │ 14 from 6+6+2
5 from 2+1+2 │ 7 from 5+1+1 │ 9 from 2+1+6 │ 10 from 2+5+3 │ 11 from 3+2+6 │ 12 from 3+5+4 │ 13 from 4+6+3 │ 15 from 3+6+6
5 from 2+2+1 │ 8 from 1+1+6 │ 9 from 2+2+5 │ 10 from 2+6+2 │ 11 from 3+3+5 │ 12 from 3+6+3 │ 13 from 5+2+6 │ 15 from 4+5+6
5 from 3+1+1 │ 8 from 1+2+5 │ 9 from 2+3+4 │ 10 from 3+1+6 │ 11 from 3+4+4 │ 12 from 4+2+6 │ 13 from 5+3+5 │ 15 from 4+6+5
6 from 1+1+4 │ 8 from 1+3+4 │ 9 from 2+4+3 │ 10 from 3+2+5 │ 11 from 3+5+3 │ 12 from 4+3+5 │ 13 from 5+4+4 │ 15 from 5+4+6
6 from 1+2+3 │ 8 from 1+4+3 │ 9 from 2+5+2 │ 10 from 3+3+4 │ 11 from 3+6+2 │ 12 from 4+4+4 │ 13 from 5+5+3 │ 15 from 5+5+5
6 from 1+3+2 │ 8 from 1+5+2 │ 9 from 2+6+1 │ 10 from 3+4+3 │ 11 from 4+1+6 │ 12 from 4+5+3 │ 13 from 5+6+2 │ 15 from 5+6+4
6 from 1+4+1 │ 8 from 1+6+1 │ 9 from 3+1+5 │ 10 from 3+5+2 │ 11 from 4+2+5 │ 12 from 4+6+2 │ 13 from 6+1+6 │ 15 from 6+3+6
6 from 2+1+3 │ 8 from 2+1+5 │ 9 from 3+2+4 │ 10 from 3+6+1 │ 11 from 4+3+4 │ 12 from 5+1+6 │ 13 from 6+2+5 │ 15 from 6+4+5
6 from 2+2+2 │ 8 from 2+2+4 │ 9 from 3+3+3 │ 10 from 4+1+5 │ 11 from 4+4+3 │ 12 from 5+2+5 │ 13 from 6+3+4 │ 15 from 6+5+4
6 from 2+3+1 │ 8 from 2+3+3 │ 9 from 3+4+2 │ 10 from 4+2+4 │ 11 from 4+5+2 │ 12 from 5+3+4 │ 13 from 6+4+3 │ 15 from 6+6+3
6 from 3+1+2 │ 8 from 2+4+2 │ 9 from 3+5+1 │ 10 from 4+3+3 │ 11 from 4+6+1 │ 12 from 5+4+3 │ 13 from 6+5+2 │ 16 from 4+6+6
6 from 3+2+1 │ 8 from 2+5+1 │ 9 from 4+1+4 │ 10 from 4+4+2 │ 11 from 5+1+5 │ 12 from 5+5+2 │ 13 from 6+6+1 │ 16 from 5+5+6
6 from 4+1+1 │ 8 from 3+1+4 │ 9 from 4+2+3 │ 10 from 4+5+1 │ 11 from 5+2+4 │ 12 from 5+6+1 │ 14 from 2+6+6 │ 16 from 5+6+5
7 from 1+1+5 │ 8 from 3+2+3 │ 9 from 4+3+2 │ 10 from 5+1+4 │ 11 from 5+3+3 │ 12 from 6+1+5 │ 14 from 3+5+6 │ 16 from 6+4+6
7 from 1+2+4 │ 8 from 3+3+2 │ 9 from 4+4+1 │ 10 from 5+2+3 │ 11 from 5+4+2 │ 12 from 6+2+4 │ 14 from 3+6+5 │ 16 from 6+5+5
...
$ _
```

Trying it



Console Input / Output

```
$ java DiceThrows | cut -f1 -d' ' | sort -n | uniq -c
 (Output shown using multiple columns to save space.)
      1 3    |      15 7     |      27 11    |      10 15
      3 4    |      21 8     |      25 12    |       6 16
      6 5    |      25 9     |      21 13    |       3 17
     10 6    |      27 10    |      15 14    |       1 18
$ _
```

Coursework: `SumOfCubedDigits`

(Summary only)

Write a program that determines which 3 digit decimal whole numbers are **equal** to the sum of the cubes of their digits.

Concepts covered in this chapter

- Each book chapter ends with a list of concepts covered in it.
- Each concept has with it
  - a self-test question,
  - and a page reference to where it was covered.
- Please use these to check your understanding before we start the next chapter.