List of Slides

Title

Java Just in Time

John Latham

October 2, 2018

Chapter 3

Types, variables and expressions

Chapter aims

- Introduce some more Java concepts:
  - There are different kinds of values – **type**s.
  - Values can be stored – **variable**s.
  - How? – **assignment statement**s.
  - We meet **arithmetic expression**s and **arithmetic operator**s.
    - With **operator precedence** and **operator associativity**.

Section 2

Example: Age next year

Aim

AIM: To introduce the concepts of **type**, `int`, **variable**, **expression** and **assignment statement**. We also find out how to convert a number to a string, and discover what it means for **data** to be **hard coded**.

Design: hard coding

- Input **data** might be
  - **command line argument**s
  - obtained via **user interface** – maybe a **GUI**
  - from **file**s.
- Sometimes input data built into the program – **hard coded**.
  - e.g. haven't written code that obtains the data yet
  - e.g. such data only rarely/never changes.

Age next year

```java
public class AgeNextYear
{
  public static void main(String[] args)
  {
```

Type

- Different kinds of **data**
  - numbers
  - text data
  - images
  - etc..
- The kind of a data item is its **type**.

Type: `int`

- The **type** `int` is **integers**
  - e.g. `0`
  - `-129934`
  - `982375`
  - etc..

Variable

- A **variable** – entity that can hold **data**.
- Has name, value and **type**.
- Similar to variables in algebra – not quite the same thing.
- Name:
  - carefully chosen by programmer
  - reflects meaning of thing it represents in relation to problem
  - does not change during program run.
- Value:
  - can be set and changed at **run time** – variable.
  - Java **compiler** maps variable names to **computer memory** locations.
- Type:
  - what kind of data is allowed.

Variable: `int` variable

- All **variable**s declared in a **variable declaration** before use.
- Programmer states **type** and name, e.g.:

  ```
  int noOfPeopleLivingInMyStreet;
  ```

  `noOfPeopleLivingInMyStreet` is an **int variable**.
  - Note semi-colon.
- At **run time** `noOfPeopleLivingInMyStreet` can hold an **integer**
  - can be changed, but always an `int`.
- Name reflects intended meaning
  - programmer writes code to ensure value reflects meaning.
- Convention:
  - variable names start with a lower case letter
  - first letter of subsequent words capitalized.

Age next year

```
    int myAgeNow;
    int myAgeNextYear;
```

Statement: assignment statement

- An **assignment statement** is a **statement**.
- Gives value to a **variable**
  - or change existing value.
- New value and variable must have matching **type**s.

Statement: assignment statement: assigning a literal value

- An **assignment statement** can assign a **literal value** (constant) to a **variable**, e.g:

```
noOfPeopleLivingInMyStreet = 47;
```

- Note use of single **equal sign**.
- 47 is an **`int`**
  - so okay if `noOfPeopleLivingInMyStreet` is an **int variable**.

Age next year

```
    myAgeNow = 18;
```

Expression: arithmetic

- Can have **arithmetic expressions** as in maths:
  - **literal values**
    - e.g. **integer literals** `1`, `18`
  - **variables**
    - must be already declared
  - **operators**, e.g. **arithmetic operators**
    - **binary infix operators**: `+`, `-`, `*`, `/`
    - **unary prefix operators**: `+`, `-`
- When **evaluated** each variable replaced with current value.
  - E.g. if `noOfPeopleLivingInMyStreet` contains `47` then `noOfPeopleLivingInMyStreet + 4` evaluates to `51`.

Statement: assignment statement: assigning an expression value

- More generally: **assignment statement** can have **expression**.
- E.g., assume

  ```
  int noOfPeopleToInviteToTheStreetParty;
  ```

  then

  ```
  noOfPeopleToInviteToTheStreetParty = noOfPeopleLivingInMyStreet + 4;
  ```

  when **execute**d

  - **evaluate**s `noOfPeopleLivingInMyStreet + 4`
  - puts result in `noOfPeopleToInviteToTheStreetParty`.

Age next year

```
    myAgeNextYear = myAgeNow + 1;
```

Type: `String`: conversion: from `int`

- The **operator** + used for both **addition** and **concatenation**
  - an **overloaded operator**.
- If at least one **operand** is a **text data string** then concatenation, else addition.
- If only one is a string, other is converted to string before concatenation.
- Note difference between an **integer** and string of decimal digits.
  - E.g. **integer literal** 123 is an **`int`**
  - "123" is a text data string – 3 separate **character**s.

Type: `String`: conversion: from `int`

- E.g. assume `noOfPeopleToInviteToTheStreetParty` has value 51

  ```
  System.out.println("Please invite " + noOfPeopleToInviteToTheStreetParty);
  ```

  produces:

  ```
  Please invite 51
  ```

  - 51 converted to `"51"`
  - `"Please invite "` concatenated with `"51"`
  - result passed to `System.out.println()`.

- For convenience a separate version of `System.out.println()` takes a single **`int`**, e.g.

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

- Same effect as:

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

Age next year

```
    System.out.println("My age now is " + myAgeNow);
    System.out.println("My age next year will be " + myAgeNextYear);
  }
}
```

The full AgeNextYear code

```
public class AgeNextYear
{
  public static void main(String[] args)
  {
    int myAgeNow;
    int myAgeNextYear;
    myAgeNow = 18;
    myAgeNextYear = myAgeNow + 1;
    System.out.println("My age now is " + myAgeNow);
    System.out.println("My age next year will be " + myAgeNextYear);
  }
}
```

Trying it

Console Input / Output

```
$ javac AgeNextYear.java
$ java AgeNextYear
My age now is 18
My age next year will be 19
$ _
```

Run

Coursework: Hard coded `YearsBeforeRetirement`

(Summary only)

Write a program to determine how many years *you* have before you retire!

Section 3

Example:
Age next year – a common misconception

Aim

AIM: To clarify the relationship between **variable**s and **assignment statement**s.

Age next year – a common misconception

- Common misconception: **assignment statements** are equations.
  - Not helped by use of single **equal sign**!
- If they are, then order doesn't matter!

```
public class AgeNextYear
{
  public static void main(String[] args)
  {
    int myAgeNow;
    int myAgeNextYear;
    myAgeNextYear = myAgeNow + 1;
    myAgeNow = 18;
    System.out.println("My age now is " + myAgeNow);
    System.out.println("My age next year will be " + myAgeNextYear);
  }
}
```

Trying it

Console Input / Output

```
$ javac AgeNextYear.java
AgeNextYear.java:7: variable myAgeNow might not have been initialized
    myAgeNextYear = myAgeNow + 1;
                    ^
1 error
$ _
```

Run

- Compiler checks **variable** has been given value before use.

Changing my age

- Can change the value of a variable.

```java
public class AgeNextYear
{
  public static void main(String[] args)
  {
    int myAgeNow;
    int myAgeNextYear;
    myAgeNow = 18;
    myAgeNextYear = myAgeNow + 1;
    myAgeNow = 60;
    System.out.println("My age now is " + myAgeNow);
    System.out.println("My age next year will be " + myAgeNextYear);
  }
}
```

Coffee time: What would be the result?

Section 4

Example:
Age next year with a command line argument

Aim

AIM: To introduce the idea of converting a **command line argument** into an `int` and using the value in a program.

Standard API: `Integer: parseInt()`

- Often want to turn a **text data string** representation of an **integer** into that number.
  - E.g. turn `"123"` into `123`.
- A simple way:

```
Integer.parseInt("123");
```

- `Integer` is a **class** in the **API**: has **method** called `parseInt`.
- E.g.

```
int firstArgument;
firstArgument = Integer.parseInt(args[0]);
```

  - takes first **command line argument**
  - computes number it represents (if it does – run time error otherwise)
  - stores that in `firstArgument`.

Age next year with a command line argument

```java
public class AgeNextYear
{
  public static void main(String[] args)
  {
    int ageNow;
    int ageNextYear;

    ageNow = Integer.parseInt(args[0]);
    ageNextYear = ageNow + 1;

    System.out.println("Your age now is " + ageNow);
    System.out.println("Your age next year will be " + ageNextYear);
  }
}
```

Trying it

Console Input / Output

```
$ javac AgeNextYear.java
$ java AgeNextYear 60
Your age now is 60
Your age next year will be 61
$ java AgeNextYear 18
Your age now is 18
Your age next year will be 19
$ java AgeNextYear John
Exception in thread "main" java.lang.NumberFormatException: For input string: "J
ohn"
        at java.lang.NumberFormatException.forInputString(NumberFormatException.
java:48)
        at java.lang.Integer.parseInt(Integer.java:449)
        at java.lang.Integer.parseInt(Integer.java:499)
        at AgeNextYear.main(AgeNextYear.java:8)
$ _
```

Run

Coursework: Command line `YearsBeforeRetirement`

(Summary only)

Write a program to determine how many years the user has before he or she retires.

Section 5

Example: Finding the volume of a fish tank

Aim

AIM: To reinforce the use of **command line argument**s and **expression**s, and introduce the idea of splitting up lines of code which are too long, whilst maintaining their readability. We also see that a **variable** can be given a value when it is declared.

Variable: a value can be assigned when a variable is declared

- We can declare a **variable** and give it a value at the same time.
- E.g.

```
int noOfHousesInMyStreet = 26;
```

Coffee time: Could we have already used that idea in this Chapter?

Finding the volume of a fish tank

```java
public class FishTankVolume
{
  public static void main(String[] args)
  {
    int width = Integer.parseInt(args[0]);
    int depth = Integer.parseInt(args[1]);
    int height = Integer.parseInt(args[2]);
    int volume = width * depth * height;
```

Code clarity: layout: splitting long lines

- Long **source code** lines are a bad idea:
  - more horizontal eye movement to scan the code
  - use horizontal scroll bar, or have wide/fullscreen window
  - when printed will truncate or at least line wrap
- Keep source code lines shorter than 80 **character**s.
- Long **statement**s split into separate lines.
  - Carefully chosen places
    - ∗ Human readers scan down the left hand side of the code.
    - ∗ If line continues previous, make obvious at start.
    - ∗ Use **indentation**.
    - ∗ Split line before symbol not normally used to start a statement.
- Code read many more times than written. . . .

Finding the volume of a fish tank

- Split at carefully chosen places.
- Use of indentation.

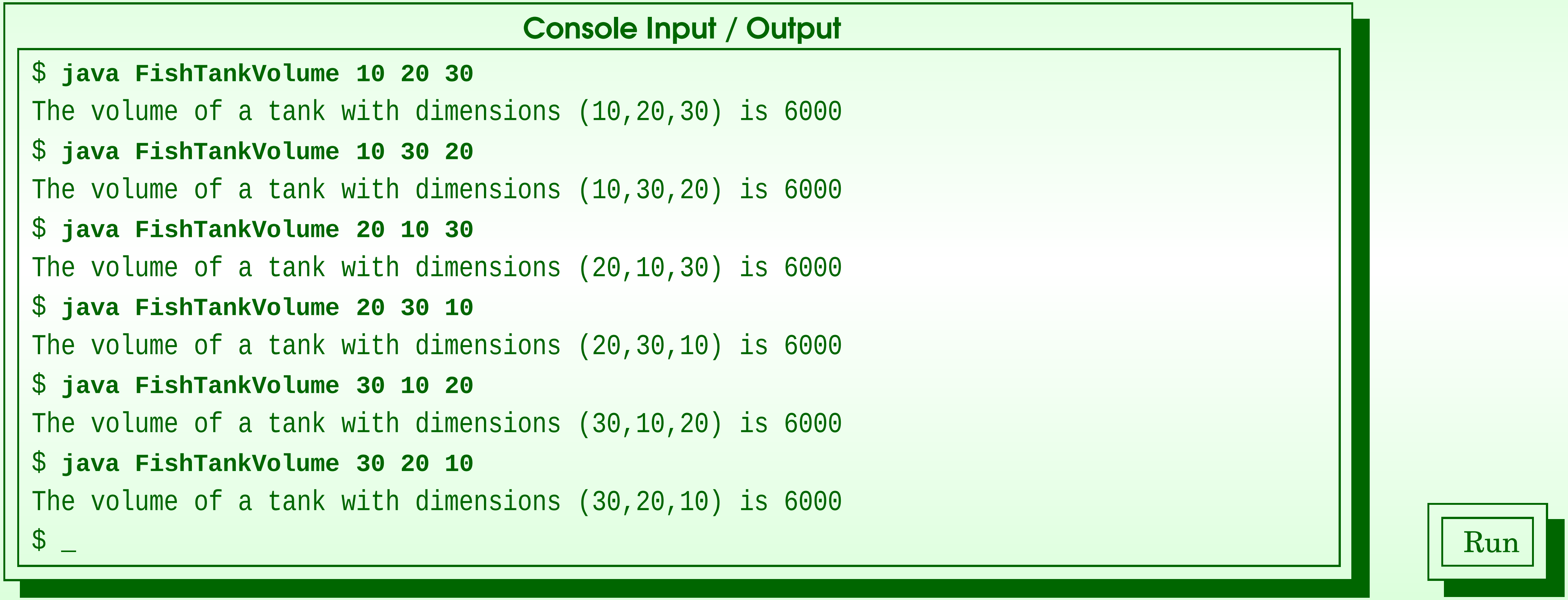
```
    System.out.println("The volume of a tank with dimensions "
                       + "(" + width + "," + depth + "," + height + ") "
                       + "is " + volume);
  }
}
```

The full FishTankVolume code

```java
public class FishTankVolume
{
  public static void main(String[] args)
  {
    int width = Integer.parseInt(args[0]);
    int depth = Integer.parseInt(args[1]);
    int height = Integer.parseInt(args[2]);
    int volume = width * depth * height;
    System.out.println("The volume of a tank with dimensions "
                       + "(" + width + "," + depth + "," + height + ") "
                       + "is " + volume);
  }
}
```

Trying it

Tests that show **command line argument** order not important.

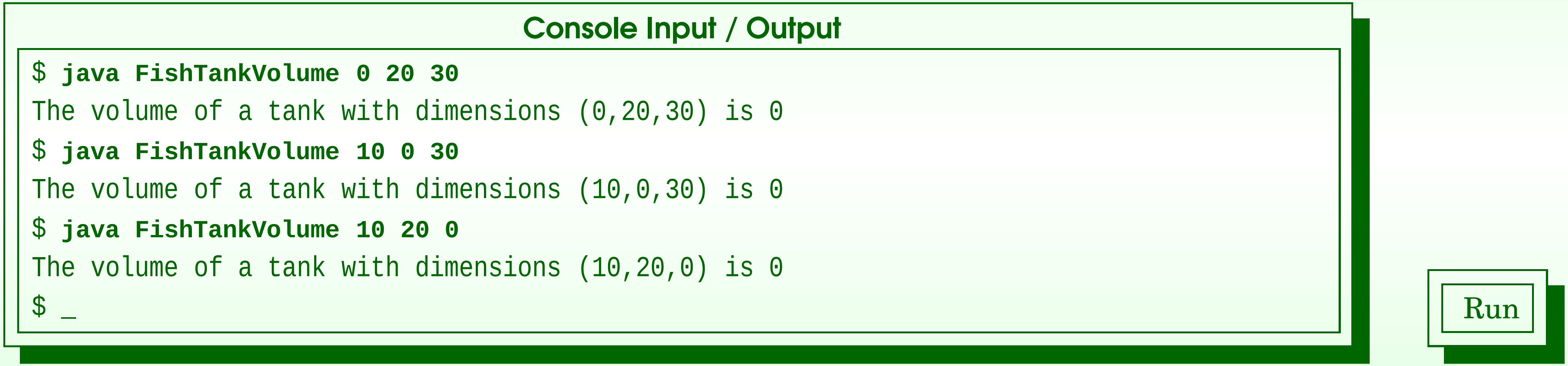


Console Input / Output

```
$ java FishTankVolume 10 20 30
The volume of a tank with dimensions (10,20,30) is 6000
$ java FishTankVolume 10 30 20
The volume of a tank with dimensions (10,30,20) is 6000
$ java FishTankVolume 20 10 30
The volume of a tank with dimensions (20,10,30) is 6000
$ java FishTankVolume 20 30 10
The volume of a tank with dimensions (20,30,10) is 6000
$ java FishTankVolume 30 10 20
The volume of a tank with dimensions (30,10,20) is 6000
$ java FishTankVolume 30 20 10
The volume of a tank with dimensions (30,20,10) is 6000
$ _
```

Trying it

Show effect of one dimension being zero.



Console Input / Output

```
$ java FishTankVolume 0 20 30
The volume of a tank with dimensions (0,20,30) is 0
$ java FishTankVolume 10 0 30
The volume of a tank with dimensions (10,0,30) is 0
$ java FishTankVolume 10 20 0
The volume of a tank with dimensions (10,20,0) is 0
$ _
```

Trying it

Coffee time: How about this next test? Is the result correct? Is it meaningful?

Console Input / Output

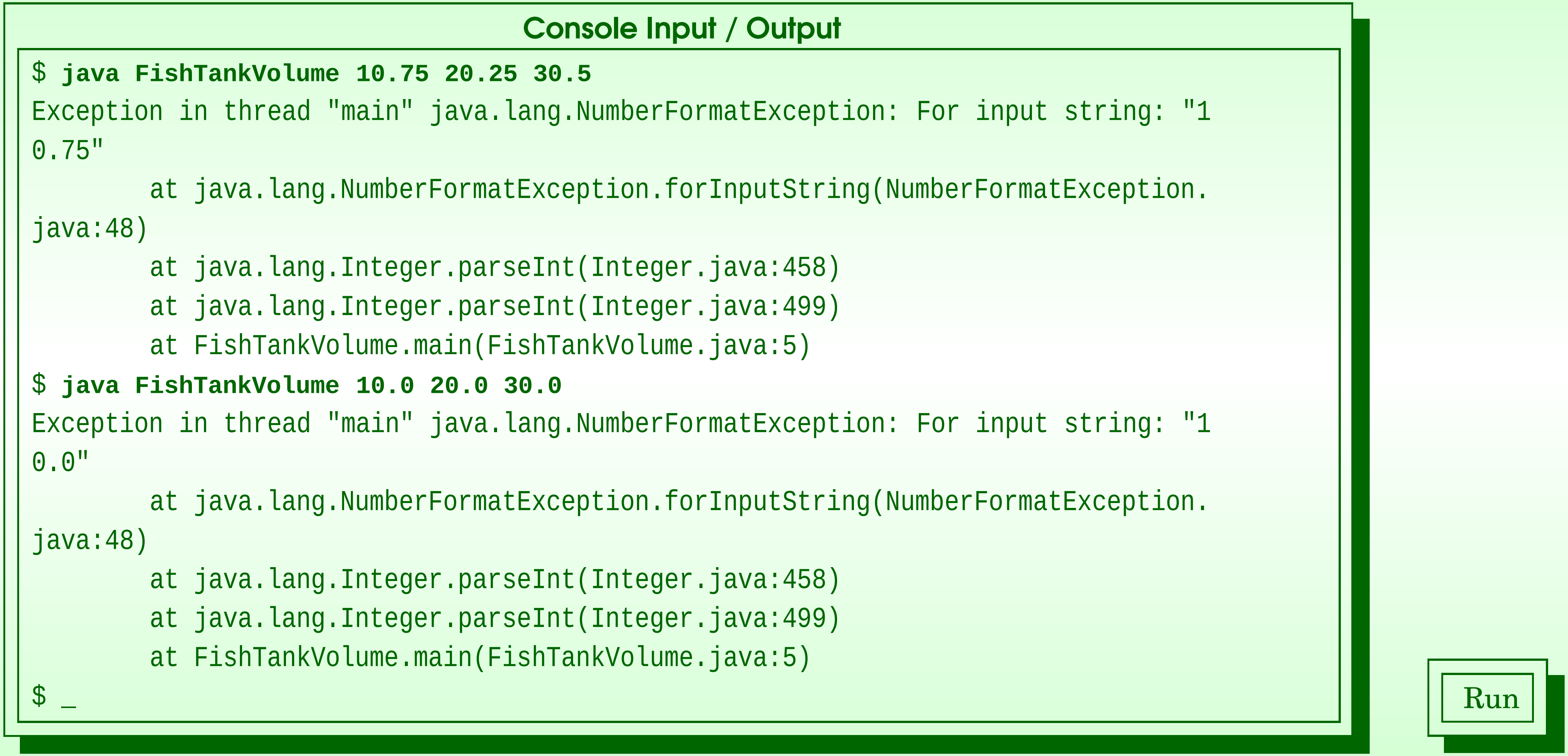
```
$ java FishTankVolume 10 -20 -30
The volume of a tank with dimensions (10,-20,-30) is 6000
$ _
```

Run

Trying it

Coffee time: If we are taking program testing seriously, then the whole point of it is to try and find situations that break the program, rather than 'prove' that it works. In what sense are the next two tests successful?

Trying it



```
$ java FishTankVolume 10.75 20.25 30.5
Exception in thread "main" java.lang.NumberFormatException: For input string: "1
0.75"
        at java.lang.NumberFormatException.forInputString(NumberFormatException.
java:48)
        at java.lang.Integer.parseInt(Integer.java:458)
        at java.lang.Integer.parseInt(Integer.java:499)
        at FishTankVolume.main(FishTankVolume.java:5)
$ java FishTankVolume 10.0 20.0 30.0
Exception in thread "main" java.lang.NumberFormatException: For input string: "1
0.0"
        at java.lang.NumberFormatException.forInputString(NumberFormatException.
java:48)
        at java.lang.Integer.parseInt(Integer.java:458)
        at java.lang.Integer.parseInt(Integer.java:499)
        at FishTankVolume.main(FishTankVolume.java:5)
$ _
```

Coursework: `FieldPerimeter`

(Summary only)

Write a program to determine how much fence is needed to surround a rectangular field.

Section 6

Example: Sum the first N numbers – incorrectly

Aim

AIM: To introduce the principle of **operator precedence**, and have a program containing a **bug**.

Sum the first N numbers – incorrectly

```
public class SumFirstN
{
  public static void main(String[] args)
  {
    int n = Integer.parseInt(args[0]);
```

Sum the first N numbers – incorrectly

- Formula:
  - find average of numbers 1 to n
  - multiply by n
  - i.e.: $\frac{1+n}{2}n$

Expression: brackets and precedence

- Java **expression**s can have round brackets.
  - Define structure.
- E.g.

```
(2 + 4) * 8
2 + (4 * 8)
```

different structures, different values: 48 and 34.

Expression: brackets and precedence

- Show structure as **expression trees**.

```
(2 + 4) * 8                  2 + (4 * 8)
        *                      +
    ___/ \                    / \___
   +      8                  2      *
  / \                                / \
 2   4                            4   8
```

Expression: brackets and precedence

- No brackets?

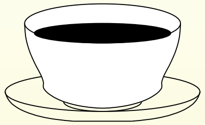
  ```
  2 + 4 * 8
  ```

- Rules to fill in missing brackets.
- 4 above is being 'pulled' by + and *.
  - Which one wins?
- Varying levels of **operator precedence**
  - * and / have higher precedence than + and -
- `2 + 4 * 8` evaluates to 34.

Sum the first N numbers – incorrectly

- If **operator**s **evaluate**d left to right would write: `1 + n / 2 * n`
- But **division** and **multiplication** higher precedence than **addition**.

```
    int sumOfFirstN = (1 + n) / 2 * n;
    System.out.println("The sum of the first " + n + " numbers is "
                       + sumOfFirstN);
  }
}
```

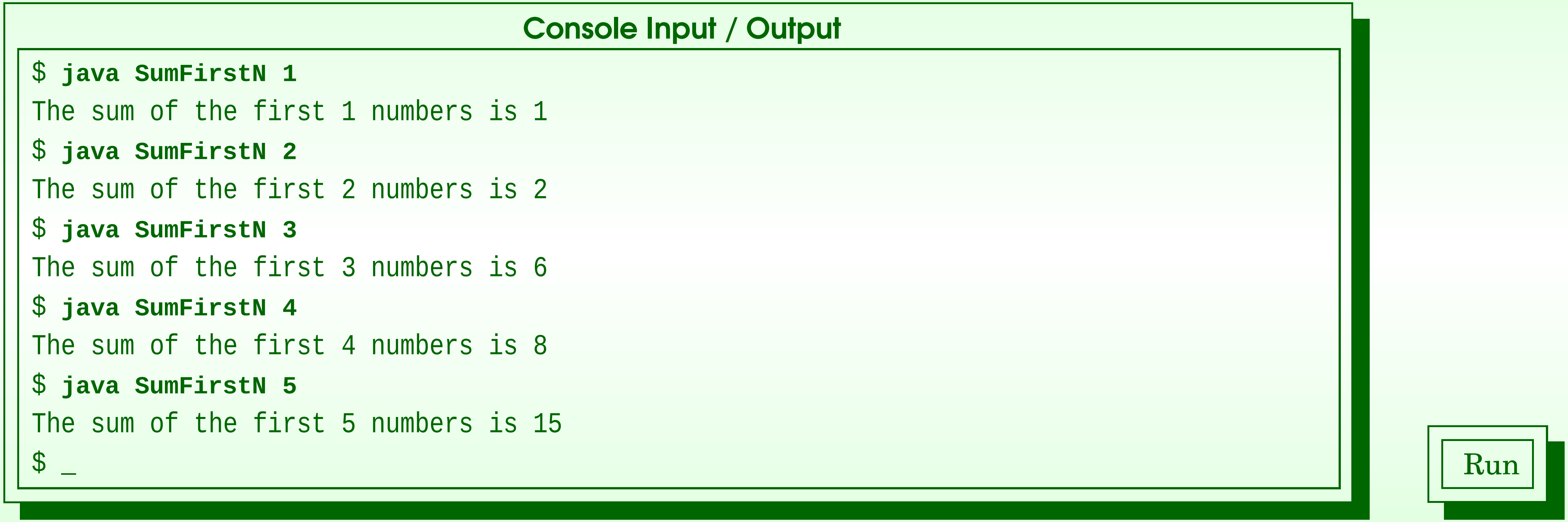


Coffee time: When computing the value of `sumOfFirstN`, do you think the division is done before the multiplication, or vice versa? Does it matter?

The full SumFirstN code

```java
public class SumFirstN
{
  public static void main(String[] args)
  {
    int n = Integer.parseInt(args[0]);
    int sumOfFirstN = (1 + n) / 2 * n;
    System.out.println("The sum of the first " + n + " numbers is "
                       + sumOfFirstN);
  }
}
```

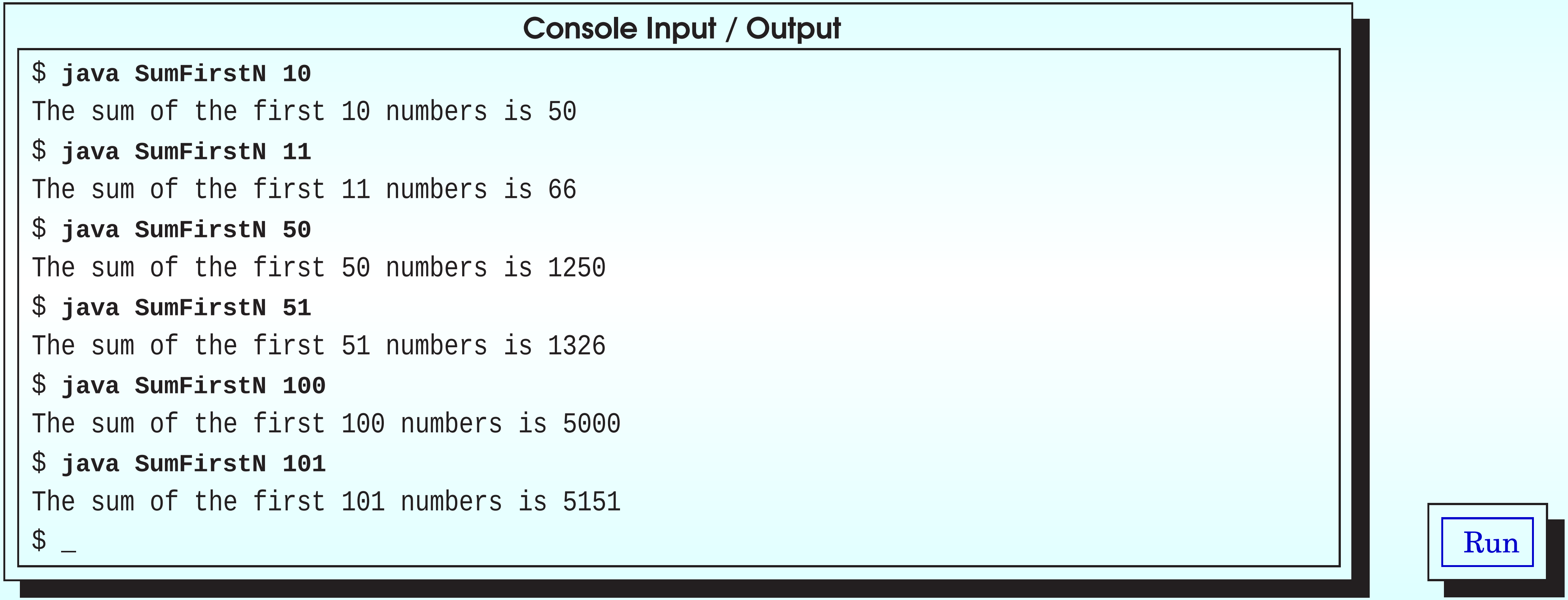
Trying it



Console Input / Output

```
$ java SumFirstN 1
The sum of the first 1 numbers is 1
$ java SumFirstN 2
The sum of the first 2 numbers is 2
$ java SumFirstN 3
The sum of the first 3 numbers is 6
$ java SumFirstN 4
The sum of the first 4 numbers is 8
$ java SumFirstN 5
The sum of the first 5 numbers is 15
$ _
```

Trying it



Console Input / Output

```
$ java SumFirstN 10
The sum of the first 10 numbers is 50
$ java SumFirstN 11
The sum of the first 11 numbers is 66
$ java SumFirstN 50
The sum of the first 50 numbers is 1250
$ java SumFirstN 51
The sum of the first 51 numbers is 1326
$ java SumFirstN 100
The sum of the first 100 numbers is 5000
$ java SumFirstN 101
The sum of the first 101 numbers is 5151
$ _
```

Trying it

Some of these results are wrong!

Coffee time: Figure out which ones are right and which are wrong, and see if you can spot a pattern, leading you to suggest what the problem might be. We know the formula is right, so you can still use it to work out what the answers should have been. The error lies somewhere in our implementation of the formula – maybe something there doesn′t behave as you might expect it to?

Coursework: `FishTankMaterials`

(Summary only)

Take a program with **bug**s in it, and fix them.

Section 7

Example: Disposable income

Aim

AIM: To introduce **operator associativity**. We also take a look at the **string literal escape sequence**s.

Disposable income

```java
public class DisposableIncome
{
  public static void main(String[] args)
  {
    int salary   = Integer.parseInt(args[0]);
    int mortgage = Integer.parseInt(args[1]);
    int bills    = Integer.parseInt(args[2]);
```

Expression: associativity

- Some **expressions** cannot be disambiguated just by **operator precedence**.
- E.g.

```
10 + 7 + 3
10 + 7 - 3
10 - 7 + 3
10 - 7 - 3
```

  - The 7 is being fought over by two **operators** with *same* precedence.

Expression: associativity

- Two possible structures:

```
10 OP1 (7 OP2 3)                (10 OP1 7) OP2 3

   OP1                                  ___OP2
  /   \___                             /      \
10        OP2                       OP1        3
         /   \                     /   \
        7     3                  10     7
```

Expression: associativity

- Does it make a difference?

Expression	**Value**
`(10 + 7) + 3`	20
`10 + (7 + 3)`	20
`(10 + 7) - 3`	14
`10 + (7 - 3)`	14
`(10 - 7) + 3`	6
`10 - (7 + 3)`	0
`(10 - 7) - 3`	0
`10 - (7 - 3)`	6

- Yes – when first operator is `-`.

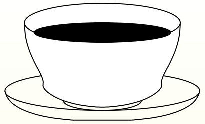
Expression: associativity

- Java operators also have **operator associativity**.
- +, -, * and / have **left associativity**
  - when two equal precedence operators fight over an **operand**?
    - The *left* one wins.

Expression	Implicit brackets	Value
`10 + 7 + 3`	`(10 + 7) + 3`	20
`10 + 7 - 3`	`(10 + 7) - 3`	14
`10 - 7 + 3`	`(10 - 7) + 3`	6
`10 - 7 - 3`	`(10 - 7) - 3`	0

- * and / also have equal precedence (higher than + and -).

Disposable income



Coffee time: Figure out why `"I earn " + 1 + 2 + 3 + 4 + 5 + 6` evaluates to `"I earn 123456"`, whereas `"I am " + (1 + 2 + 3 + 4 + 5 + 6)` becomes `"I am 21"`.

Disposable income

```
int disposableIncome = salary - (mortgage + bills);
```

Coffee time: Alternatively, we could have written our **expression** as `salary - mortgage - bills`. Convince yourself that this would produce the same result, whereas the expression `salary - mortgage + bills` would be wrong.

Type: `String`: literal: escape sequences

- Use **escape sequence** `\n` to have **new line character** in **string literal**.
- E.g.

  ```
  System.out.println("This text\nspans three\nlines.");
  ```

  produces:

  ```
  This text
  spans three
  lines.
  ```

- Note: `System.out.println()` always produces **line separator**
  - **carriage return character** followed by **new line character** on Windows.

Type: `String`: literal: escape sequences

Sequence	Name	Effect
`\b`	Backspace	Moves the cursor back one place, so the next **character** will over-print the previous.
`\t`	Tab (horizontal tab)	Moves the cursor to the next ‘tab stop’.
`\n`	New line (line feed)	Moves the cursor to the next line.
`\f`	Form feed	Moves to a new page on many (text) printers.
`\r`	Carriage return	Moves the cursor to the start of the current line, so characters will over-print those already printed.
`\"`	Double quote	Without the backslash escape, this would mark the end of the string literal.
`\'`	Single quote	This is just for consistency – we don’t need to escape a single quote in a string literal.
`\\`	Backslash	Well, sometimes you want the backslash character itself.

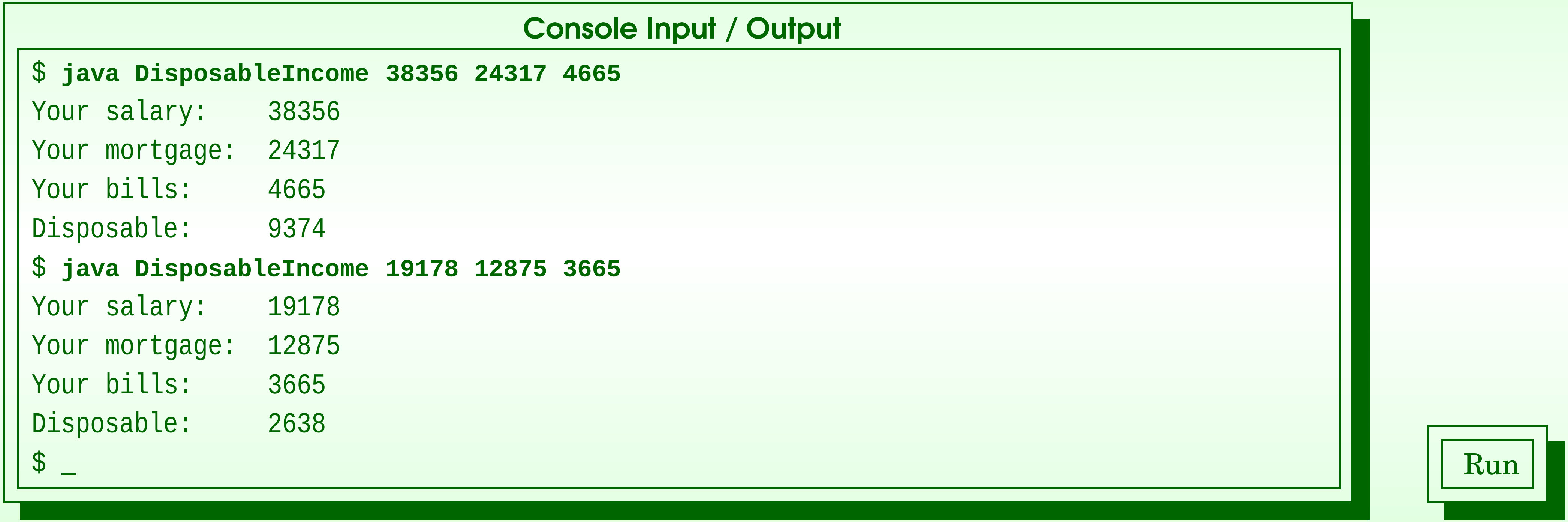
Disposable income

```
    System.out.println("Your salary:\t" + salary
                       + "\nYour mortgage:\t" + mortgage
                       + "\nYour bills:\t" + bills
                       + "\nDisposable:\t" + disposableIncome);
  }
}
```

The full DisposableIncome code

```java
public class DisposableIncome
{
  public static void main(String[] args)
  {
    int salary   = Integer.parseInt(args[0]);
    int mortgage = Integer.parseInt(args[1]);
    int bills    = Integer.parseInt(args[2]);
    int disposableIncome = salary - (mortgage + bills);
    System.out.println("Your salary:\t" + salary
                       + "\nYour mortgage:\t" + mortgage
                       + "\nYour bills:\t" + bills
                       + "\nDisposable:\t" + disposableIncome);
  }
}
```

Trying it



```
$ java DisposableIncome 38356 24317 4665
Your salary:    38356
Your mortgage:  24317
Your bills:     4665
Disposable:     9374
$ java DisposableIncome 19178 12875 3665
Your salary:    19178
Your mortgage:  12875
Your bills:     3665
Disposable:     2638
$ _
```

Coursework: `ThreeWeights`

(Summary only)

Write a program to show what weights can be weighed using a balance scale and three given weights.

Section 8

Example:
Sum the first N numbers – correctly

Aim

AIM: To introduce the fact that **integer division** produces a truncated result. We then look at the interaction between that and **operator associativity**.

Expression: arithmetic: `int` division truncates result

- The **division operator** uses **integer division** when given two **integer**s.
  - throws away any remainder.
- E.g.
  - `8 / 2` is `4`
  - `9 / 2` is `4`
- Always rounds towards zero:
  - `15 / 4` is `3`, not `3.75` nor `4`.

Sum the first N numbers – correctly

- Previous implementation: `(1 + n) / 2 * n`
  - only works if `n` is odd.
- Ensure **multiplication** done before **division**
  - must work: sum of the first `n` whole numbers is a whole number!

Expression: arithmetic: associativity and `int` division

- * and / have equal **operator precedence** and **left associativity**.
- But / with **integer**s truncates.
- E.g.

Expression	**Implicit brackets**	**Value**
`9 * 4 / 2`	`(9 * 4) / 2`	18
`9 / 2 * 4`	`(9 / 2) * 4`	16

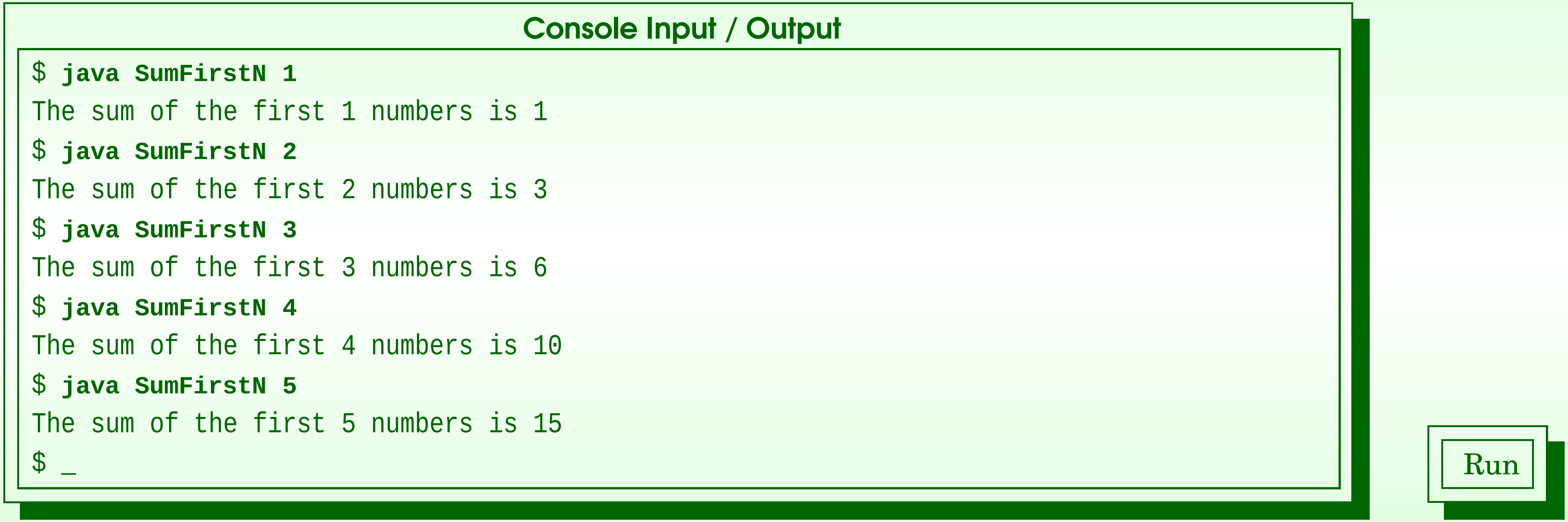
Sum the first N numbers – correctly

Simplest **bug** fix: swap order of divide and multiply.

Coffee time: Convince yourself that this will always avoid the problem for this program.

```
public class SumFirstN
{
  public static void main(String[] args)
  {
    int n = Integer.parseInt(args[0]);
    int sumOfFirstN = (1 + n) * n / 2;
    System.out.println("The sum of the first " + n + " numbers is "
                       + sumOfFirstN);
  }
}
```

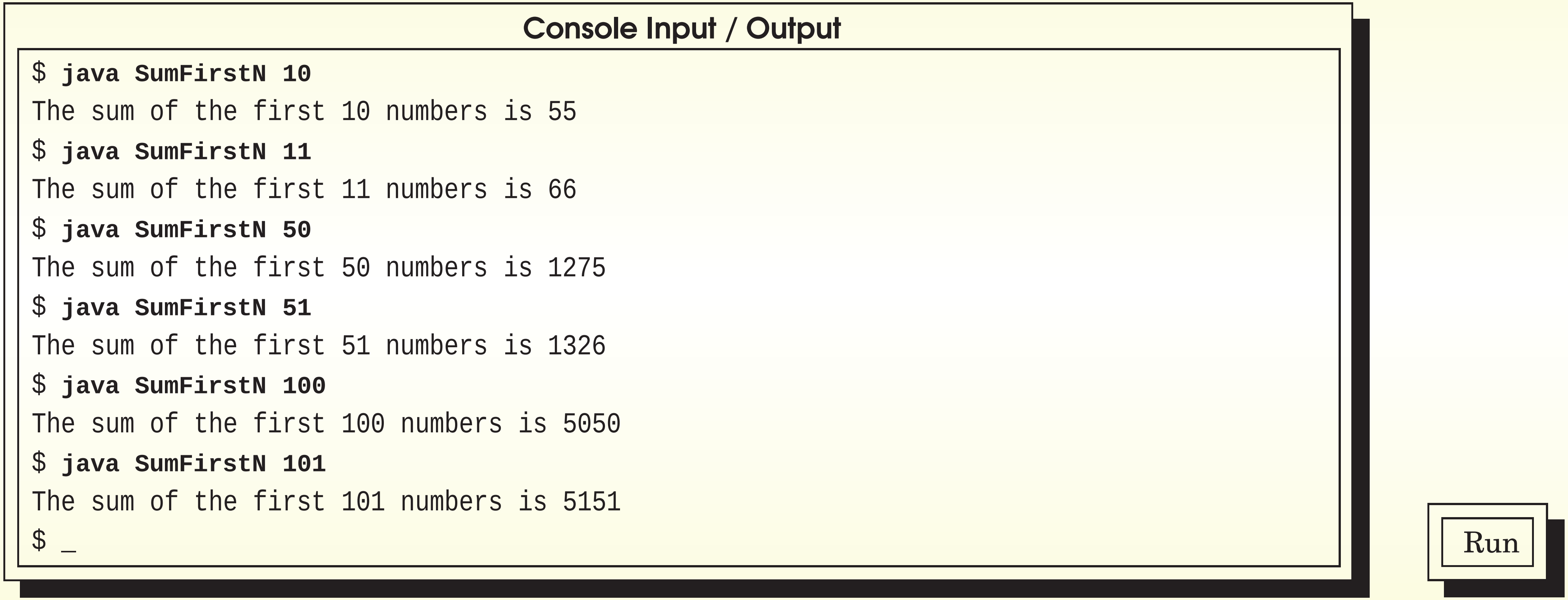
Trying it



Console Input / Output

```
$ java SumFirstN 1
The sum of the first 1 numbers is 1
$ java SumFirstN 2
The sum of the first 2 numbers is 3
$ java SumFirstN 3
The sum of the first 3 numbers is 6
$ java SumFirstN 4
The sum of the first 4 numbers is 10
$ java SumFirstN 5
The sum of the first 5 numbers is 15
$ _
```

Trying it



```
$ java SumFirstN 10
The sum of the first 10 numbers is 55
$ java SumFirstN 11
The sum of the first 11 numbers is 66
$ java SumFirstN 50
The sum of the first 50 numbers is 1275
$ java SumFirstN 51
The sum of the first 51 numbers is 1326
$ java SumFirstN 100
The sum of the first 100 numbers is 5050
$ java SumFirstN 101
The sum of the first 101 numbers is 5151
$ _
```

Coursework: `RoundPennies`

(Summary only)

Write a program to help a child determine whether she has enough pennies to go shopping!

Section 9

Example: Temperature conversion

Aim

AIM: To introduce the **`double`** **type** and some associated concepts, including converting to and from strings, and **double division**.

Type: `double`

- The **type** **`double`** is **real**s.
- E.g. `0.0`, `-129.934`, `98.2375`.
- Uses **double precision** storage technique.
  - real numbers only approximated: stored in finite memory space.
  - double precision uses twice as much memory per number
    - than older **single precision** technique.
  - much more precise than single precision.

Variable: `double` variable

- Can declare **double variable**s – **variable**s of **type** **`double`**.
- E.g.

```
    double meanAgeOfPeopleLivingInMyHouse;
```

- Value of `meanAgeOfPeopleLivingInMyHouse` can change
  - but must always be a **`double`**.
  - E.g. `33.0`

Standard API: `Double`: `parseDouble()`

- Often want to turn a **text data string** representation of a **real** into that number.
  - E.g. turn `"123.456"` into `123.456`.
- A simple way:

```
Double.parseDouble("123.456");
```

- `Double` is a **class** in the **API**: has **method** called `parseDouble`.
- E.g.

```
double firstArgument = Double.parseDouble(args[0]);
```

  - takes first **command line argument**
  - computes number it represents (if it does – run time error otherwise)
  - stores that in `firstArgument`.

Temperature conversion

```java
public class CelsiusToFahrenheit
{
  public static void main(String[] args)
  {
    double celsiusValue = Double.parseDouble(args[0]);
```

Expression: arithmetic: `double` division

- / uses **double division** with **`double`** result if at least one **operand** is **`double`**.
- E.g.

Expression	Result	Type of Result
`8 / 2`	`4`	`int`
`8 / 2.0`	`4.0`	`double`
`9 / 2`	`4`	`int`
`9 / 2.0`	`4.5`	`double`
`9.0 / 2`	`4.5`	`double`
`9.0 / 2.0`	`4.5`	`double`

Temperature conversion

- Celsius / Fahrenheit relationship: $F = \frac{9}{5}C + 32$

```
    double fahrenheitValue = celsiusValue * 9 / 5 + 32;
```

Type: `String`: conversion: from `double`

- Java **concatenation operator** also converts **`double`** to a string.
- E.g. `"" + 123.4` has the value `"123.4"`.

Temperature conversion

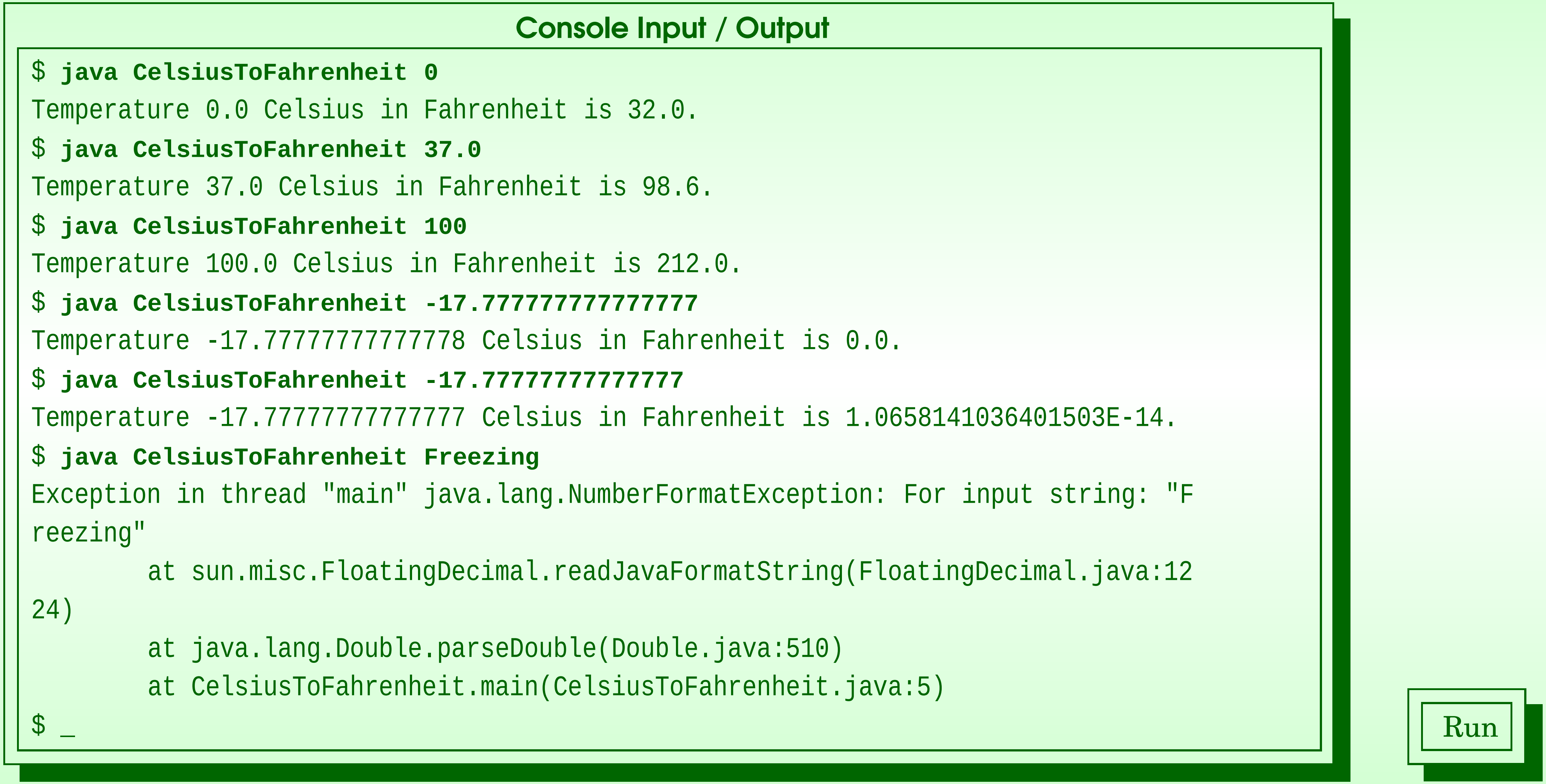
```
    System.out.println("Temperature " + celsiusValue + " Celsius"
                       + " in Fahrenheit is " + fahrenheitValue + ".");
  }
}
```

Coffee time: What do you think would happen if we declared the **variable** `fahrenheitValue` as an **`int`** instead of a **`double`**?

The full CelsiusToFahrenheit code

```java
public class CelsiusToFahrenheit
{
  public static void main(String[] args)
  {
    double celsiusValue = Double.parseDouble(args[0]);
    double fahrenheitValue = celsiusValue * 9 / 5 + 32;
    System.out.println("Temperature " + celsiusValue + " Celsius"
                       + " in Fahrenheit is " + fahrenheitValue + ".");
  }
}
```

Trying it



Console Input / Output

```
$ java CelsiusToFahrenheit 0
Temperature 0.0 Celsius in Fahrenheit is 32.0.
$ java CelsiusToFahrenheit 37.0
Temperature 37.0 Celsius in Fahrenheit is 98.6.
$ java CelsiusToFahrenheit 100
Temperature 100.0 Celsius in Fahrenheit is 212.0.
$ java CelsiusToFahrenheit -17.777777777777777
Temperature -17.77777777777778 Celsius in Fahrenheit is 0.0.
$ java CelsiusToFahrenheit -17.77777777777777
Temperature -17.77777777777777 Celsius in Fahrenheit is 1.0658141036401503E-14.
$ java CelsiusToFahrenheit Freezing
Exception in thread "main" java.lang.NumberFormatException: For input string: "F
reezing"
        at sun.misc.FloatingDecimal.readJavaFormatString(FloatingDecimal.java:12
24)
        at java.lang.Double.parseDouble(Double.java:510)
        at CelsiusToFahrenheit.main(CelsiusToFahrenheit.java:5)
$ _
```

Coursework: `FahrenheitToCelsius`

(Summary only)

Write a program to convert a temperature from Fahrenheit to Celsius.

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.