List of Slides

Title

Java Just in Time

John Latham

October 10, 2018

Chapter 5

Repeated execution

Chapter aims

- Most programs need parts of code to be **execute**d more than once
  - **repeated execution** or **iteration**.
- We meet the **while loop** and **for loop statement**s.
- Plus some more general concepts.

Section 2

Example: Minimum tank size

Aim

AIM: To introduce the idea of **repeated execution**, implemented by the **while loop**. We also meet the notion of a **variable update**.

Minimum tank size

- You make central heating oil storage tanks
  - always cubic
  - six pieces of sheet metal from $1M^2$ upwards, always whole metres.
- Want program to compute size of smallest tank to hold given volume.
  - start with smallest size
  - keep making bigger by 1 until big enough.

Execution: repeated execution

- Obeying instructions just once is not sufficient to solve many problems
  - some instructions need to be **execute**d, zero, one or many times.
- Known as **repeated execution**, **iteration**, or **loop**ing.
- Number of times depends on some **condition** involving **variable**s.

Statement: assignment statement: updating a variable

- Values of **variable**s can change.
- E.g. work out maximum of three numbers.

```
int x;
int y;
int z;
... Code here that gives values to x, y and z.

int maximumOfXYandZSoFar = x;
if (maximumOfXYandZSoFar < y)
  maximumOfXYandZSoFar = y;
if (maximumOfXYandZSoFar < z)
  maximumOfXYandZSoFar = z;
```

- `maximumOfXYandZSoFar` gets given a value, then maybe it is changed.

Statement: assignment statement: updating a variable

- Commonly wish the program to perform a **variable update**
  - often inside a **loop**.
- E.g. add one to value of `countSoFar`:

```
countSoFar = countSoFar + 1;
```

- Reminder – **assignment statement**s are not definitions of **equality**.

Statement: while loop

- One way of looping is the **while loop**.
- Two parts
  - **condition** – evaluated each time
  - **statement** – **execute**d while condition is `true`.
- Syntax:
  - **reserved word** `while`
  - condition in brackets
  - statement to be repeated

Statement: while loop

- E.g. – inefficient way to give x the value `21`:

```
int x = 1;
while (x < 20)
  x = x + 2;
```

- x starts with value 1
  - repeatedly has 2 added to it
  - stops when `x < 20` is false.
  - So ends with value `21`.
- Notice brackets, semi-colon and lay out.

Statement: while loop

- Observe similarity between while loop and **if statement**
  - *only* difference is first word!
- Similarity in meaning:
  - while loop executes body zero or *more* times
  - if statement executes body zero or *one* time.
- Avoid common novice phrase “if loop”…

Minimum tank size

```
public class MinimumTankSize
{
  public static void main(String[] args)
  {
    double requiredVolume = Double.parseDouble(args[0]);
    int sideLength = 1;
    while (sideLength * sideLength * sideLength < requiredVolume)
      sideLength = sideLength + 1;
    System.out.println("You need a tank of " + sideLength
                       + " metres per side to hold the volume "
                       + requiredVolume + " cubic metres");
  }
}
```

Trying it

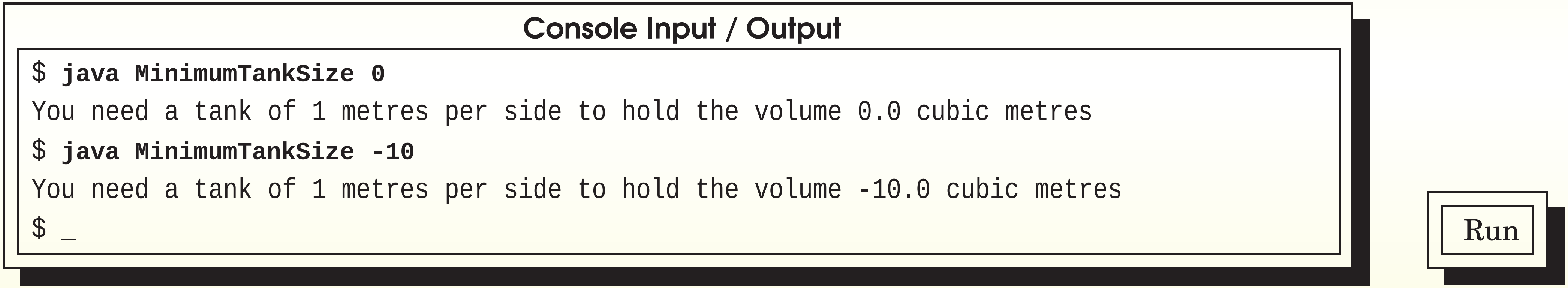
Console Input / Output

```
$ java MinimumTankSize 1
You need a tank of 1 metres per side to hold the volume 1.0 cubic metres
$ java MinimumTankSize 1.001
You need a tank of 2 metres per side to hold the volume 1.001 cubic metres
$ java MinimumTankSize 8
You need a tank of 2 metres per side to hold the volume 8.0 cubic metres
$ java MinimumTankSize 8.001
You need a tank of 3 metres per side to hold the volume 8.001 cubic metres
$ java MinimumTankSize 100
You need a tank of 5 metres per side to hold the volume 100.0 cubic metres
$ java MinimumTankSize 57.3
You need a tank of 4 metres per side to hold the volume 57.3 cubic metres
$ _
```

Run

Trying it

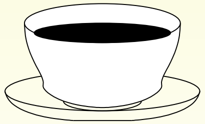
- What about some inappropriate values?



Console Input / Output

```
$ java MinimumTankSize 0
You need a tank of 1 metres per side to hold the volume 0.0 cubic metres
$ java MinimumTankSize -10
You need a tank of 1 metres per side to hold the volume -10.0 cubic metres
$ _
```

Trying it



Coffee time: A common error made by novice programmers is to place a semi-colon (;) at the end of lines which shouldn't have one. What do you think would happen if the **while loop** of our program was as follows?

```
while (sideLength * sideLength * sideLength < requiredVolume);
  sideLength = sideLength + 1;
```

(Hint: remember the **empty statement**).

Coursework: `MinimumTankSize` in half measures

(Summary only)

Write a program which calculates the minimum size of cubic tanks to hold given required volumes, where the possible sizes are in steps of 0.5 metre.

Section 3

Example:
Minimum bit width

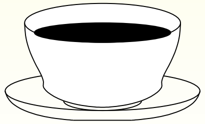
Aim

AIM: To introduce the idea of using **pseudo code** to help us **design** programs. We also meet `Math.pow()`.

Minimum bit width

- Numbers are represented in **binary** – base 2 representation
  - each is sequence of **binary digit**s (**bit**s)
  - each bit either 0 or 1.
- Want to calculate how many bits needed to represent given number of different values.
  - E.g.
  - one bit gives two values: `0`, `1`
  - two bits gives four values: `00`, `01`, `10`, `11`
  - three bits gives eight values: `000`, `001`, `010`, `011`, `100`, `101`, `110`, `111`
  - etc.

Minimum bit width



Coffee time: Convince yourself that the number of values representable using N bits is 2^N.

- We will use a **variable** `noOfBits`
  - start off with value 0
  - keep adding 1 while too small.

Design: pseudo code

- Complex programs are hard to write straight into the **text editor**
  - *so don't try to!*
- Need to **design** them *before* we implement them.
- Design does not start at first word and end at last one.
  - start wherever it suits us
  - typically at the trickiest bit.

Design: pseudo code

- We don´t express designs in Java
  - forces our mind to be cluttered with trivia
    - ∗ e.g. semi-colons, brackets, . . .
    - ∗ too distracting.
- We express **algorithm** designs in **pseudo code**
  - kind of informal programming language
  - no unnecessary trivia
  - might not bother writing **class** nor **method** headings.
- Can also vary level of **abstraction** to suit us
  - not constrained to use only features of Java at every stage.

Minimum bit width

- Pseudo code for minimum bit width:

```
get numberOfValues from command line
noOfBits = 0
while noOfBits is too small
  increment noOfBits
output noOfBits
```

Minimum bit width

- How know whether `noOfBits` is too small?
  - big enough when $2^{noOfBits} \geq numberOfValues$.
- So increment while $2^{noOfBits} < numberOfValues$.
- Rewrite pseudo code, closer to Java: less abstract.

```
numberOfValues = args[0]
noOfBits = 0
while $2^{noOfBits}$ < numberOfValues
  noOfBits = noOfBits + 1
s.o.p noOfBits
```

- Notice `s.o.p`, no semi-colons, no brackets
  - would be waste of time to write proper Java during **design**.

Standard API: `Math`: `pow()`

- No power **operator** in Java.
- But standard **class** `Math` has **method** `pow()`
  - takes two numbers, gives value of first raised to power of second.
- E.g. `Math.pow(2, 10)` produces 2^{10} i.e. `1024`.
- `Math` has many other useful maths functions.

Minimum bit width

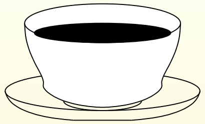
```java
public class MinimumBitWidth
{
  public static void main(String[] args)
  {
    int numberOfValues = Integer.parseInt(args[0]);
    int noOfBits = 0;
    while (Math.pow(2, noOfBits) < numberOfValues)
      noOfBits = noOfBits + 1;
    System.out.println("You need " + noOfBits + " bits to represent "
                       + numberOfValues + " values");
  }
}
```

Trying it

Console Input / Output

```
$ java MinimumBitWidth 0
You need 0 bits to represent 0 values
$ java MinimumBitWidth 1
You need 0 bits to represent 1 values
$ _
```

Run



Coffee time: What do you think of the last result above – that you can represent one value using no bits? For example, how much memory would be needed to store the gender of each member of a club that only allows women to join?

Trying it

Console Input / Output

```
$ java MinimumBitWidth 2
You need 1 bits to represent 2 values
$ java MinimumBitWidth 3
You need 2 bits to represent 3 values
$ java MinimumBitWidth 4
You need 2 bits to represent 4 values
$ java MinimumBitWidth 5
You need 3 bits to represent 5 values
$ _
```

Run

Trying it

Console Input / Output

```
$ java MinimumBitWidth 255
You need 8 bits to represent 255 values
$ java MinimumBitWidth 256
You need 8 bits to represent 256 values
$ java MinimumBitWidth 257
You need 9 bits to represent 257 values
$ _
```

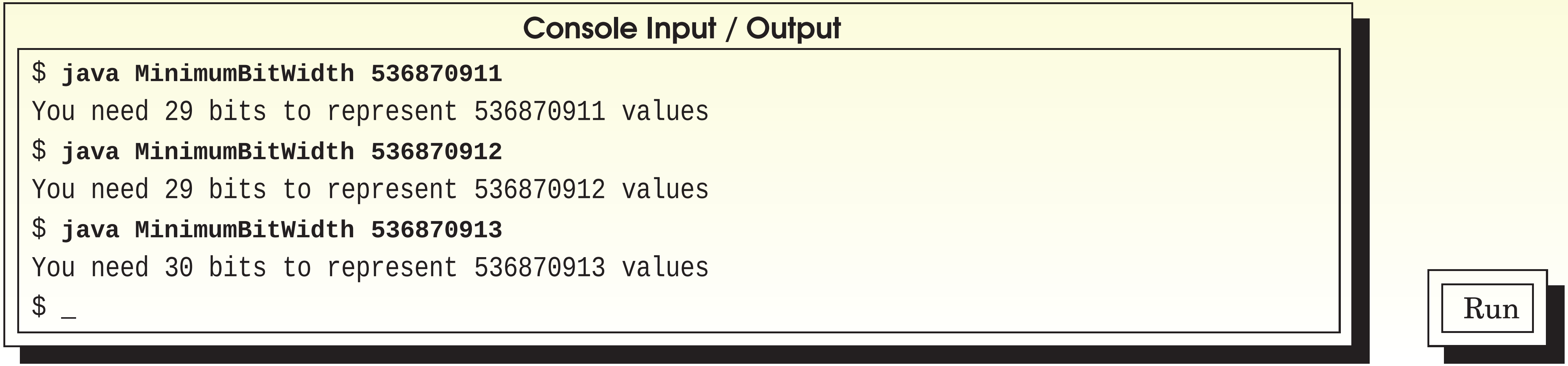
Run

Console Input / Output

```
$ java MinimumBitWidth 65535
You need 16 bits to represent 65535 values
$ java MinimumBitWidth 65536
You need 16 bits to represent 65536 values
$ java MinimumBitWidth 65537
You need 17 bits to represent 65537 values
$ _
```

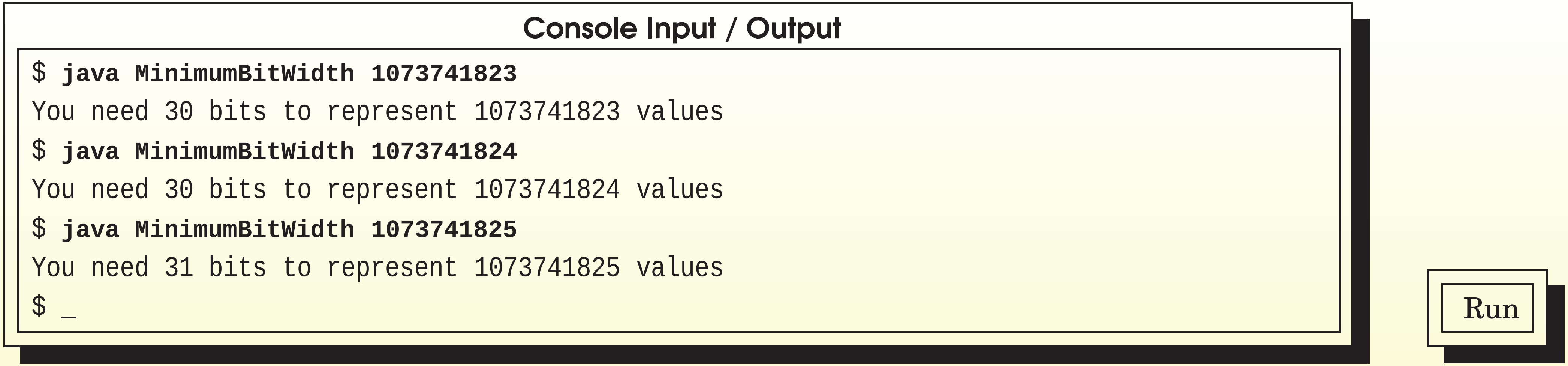
Run

Trying it



Console Input / Output

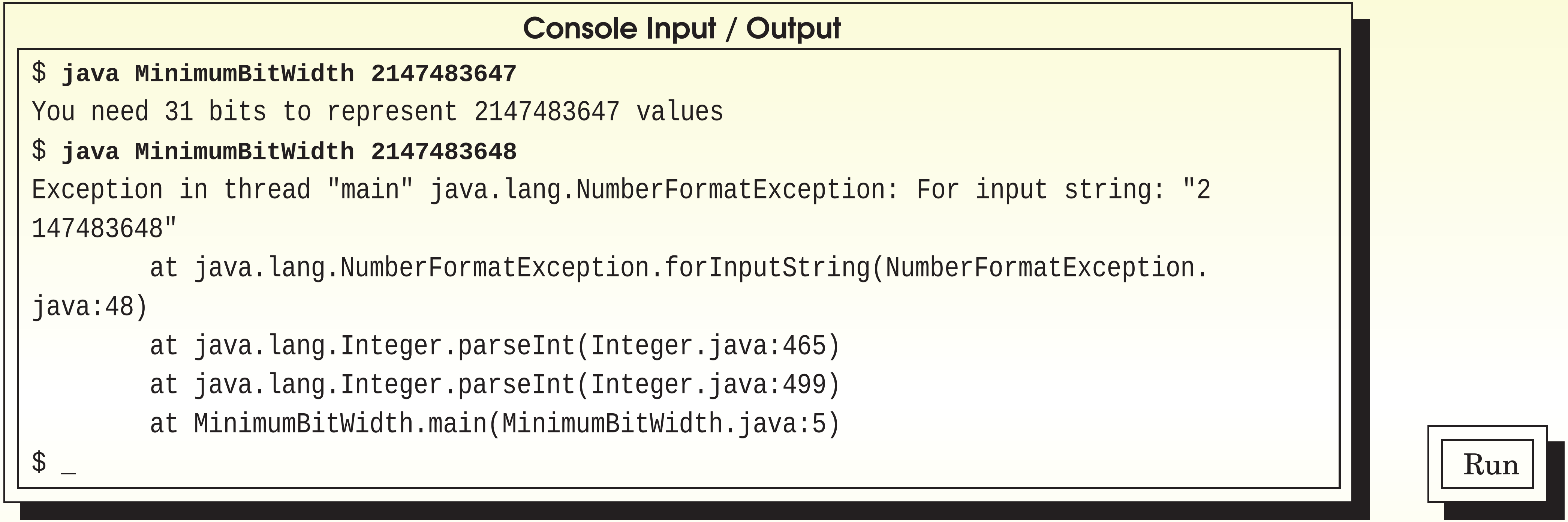
```
$ java MinimumBitWidth 536870911
You need 29 bits to represent 536870911 values
$ java MinimumBitWidth 536870912
You need 29 bits to represent 536870912 values
$ java MinimumBitWidth 536870913
You need 30 bits to represent 536870913 values
$ _
```



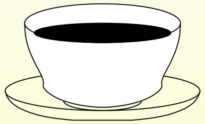
Console Input / Output

```
$ java MinimumBitWidth 1073741823
You need 30 bits to represent 1073741823 values
$ java MinimumBitWidth 1073741824
You need 30 bits to represent 1073741824 values
$ java MinimumBitWidth 1073741825
You need 31 bits to represent 1073741825 values
$ _
```

Trying it



```
$ java MinimumBitWidth 2147483647
You need 31 bits to represent 2147483647 values
$ java MinimumBitWidth 2147483648
Exception in thread "main" java.lang.NumberFormatException: For input string: "2
147483648"
        at java.lang.NumberFormatException.forInputString(NumberFormatException.
java:48)
        at java.lang.Integer.parseInt(Integer.java:465)
        at java.lang.Integer.parseInt(Integer.java:499)
        at MinimumBitWidth.main(MinimumBitWidth.java:5)
$ _
```



Coffee time: Can you guess what has caused the **exception** in the last test? (Hint: `int` uses 32 **bits** to represent numbers, and needs to store negative as well as non-negative values.)

Coursework: `LargestSquare`

(Summary only)

Write a program to find the largest square number which is **less than or equal** to a given number.

Section 4

Special note about design

Aim

AIM: To make sure the process of **design** does not get forgotten!

Special note about design

- Not enough time in lectures or room in book to show **pseudo code** for every example.
- So show only for a few.
- But don't get wrong impression:
  - all programs require some **design** work
    - depends on complexity and previous programmer experience.
- If new to programming: presume all examples from now on would require pseudo code.
- Common mistake: go to **text editor** and try to type code from start to end.
  - Makes it harder – don't do it!
    - Would you write essays that way?
    - Programs have more complex structure than essays.

Section 5

Example: Compound interest: known target

Aim

AIM: To reinforce the **while loop** and the **compound statement**.

Compound interest: known target

- Invest sum of money at given interest rate
  - How many years before reach required target balance?
- Use **while loop**
  - accumulate balance while less than target
  - count the years.
- Need **compound statement** because two **statement**s within **loop body**.

Compound interest: known target

```java
public class CompoundInterestKnownTarget
{
  public static void main(String[] args)
  {
    double initialInvestment = Double.parseDouble(args[0]);
    double interestRate = Double.parseDouble(args[1]);
    double targetBalance = Double.parseDouble(args[2]);
    int noOfYearsInvestedSoFar = 0;
    double currentBalance = initialInvestment;

```

Compound interest: known target

```
    while (currentBalance < targetBalance)
    {
      noOfYearsInvestedSoFar = noOfYearsInvestedSoFar + 1;
      currentBalance = currentBalance + currentBalance * interestRate / 100;
    }

    System.out.println(initialInvestment + " invested at interest rate "
                       + interestRate + "%");
    System.out.println("After " + noOfYearsInvestedSoFar + " years,"
                       + " the balance will be " + currentBalance);
  }
}
```

Trying it

Console Input / Output

```
$ java CompoundInterestKnownTarget 100.0 12.5 1000.0
100.0 invested at interest rate 12.5%
After 20 years, the balance will be 1054.50938424492
$ java CompoundInterestKnownTarget 100.0 4.5 1000.0
100.0 invested at interest rate 4.5%
After 53 years, the balance will be 1030.7738533669428
$ _
```

Run

Coursework: `MinimumBitWidth` by doubling

(Summary only)

Write a program to find the minimum **bit** width needed to support a given number of values, by doubling.

Section 6

Example:
Compound interest: known years

Aim

AIM: To introduce the **for loop**.

Compound interest: known years

- Invest sum of money at given interest rate for fixed number of years
  - what is balance at the end?
- Could use a **while loop**
  - **for loop** is more appropriate.

Statement: for loop

- The **for loop** best suited when number of **iteration**s is known at start.
- E.g.:

```
for (int count = 1; count <= 10; count = count + 1)
  System.out.println("Counting " + count);
```

- Syntax:
  - **reserved word `for`**
  - three items in brackets, separated by semi-colons.
  - then loop body – a **statement**
    - often a **compound statement**

Statement: for loop

- First of the three items is **for initialization**
  - performed once when loop starts
  - often delcares a **variable** and gives initial value to it.
  - E.g. `int count = 1`
- Second is **condition** for continuing
  - E.g. `count <= 10`
- Third is **for update**
  - a statement executed at *end* of each iteration
  - typically updates value of variable declared in first item.
  - E.g. `count = count + 1`

Statement: for loop

- Overall effect of example:

```
for (int count = 1; count <= 10; count = count + 1)
  System.out.println("Counting " + count);
```

  - declare `count`, set to 1
  - check if **less than** `10`
  - print `Counting 1`
  - add one to `count`
  - check again, print `Counting 2`, add one to `count`, check again,. . .
  - until condition is `false`
    - `count` has reached 11

Statement: for loop

- Don't really need for loop – **while loop** is sufficient.
- E.g.:

```
int count = 1;
while (count <= 10)
{
  System.out.println("Counting " + count);
  count = count + 1;
}
```

- However for loop places all loop control code together
  - easier to read
  - shorter
  - appropriate for known number of iterations.

Statement: for loop

- One subtle difference about **scope** of `count`
  - variables declared in for initialization can only be used in the for loop
    - do not exist elsewhere.
- Added benefit of for loop compared with equivalent while loop
  - cannot accidentally use control variable in rest of the code.

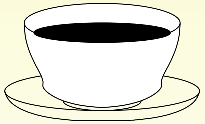
Compound interest: known years

```java
public class CompoundInterestKnownYears
{
  public static void main(String[] args)
  {
    double initialInvestment = Double.parseDouble(args[0]);
    double interestRate = Double.parseDouble(args[1]);
    int noOfYearsInvested = Integer.parseInt(args[2]);
    double currentBalance = initialInvestment;

```

Compound interest: known years

```
    for (int year = 1; year <= noOfYearsInvested; year = year + 1)
      currentBalance = currentBalance + currentBalance * interestRate / 100;

    System.out.println(initialInvestment + " invested at interest rate "
                       + interestRate + "%");
    System.out.println("After " + noOfYearsInvested + " years,"
                       + " the balance will be " + currentBalance);
  }
}
```

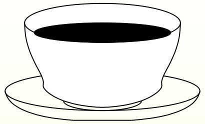
Compound interest: known years



Coffee time: Could we have written the first line of the for loop as follows?

```
for (int year = 0; year < noOfYearsInvested; year = year + 1)
```

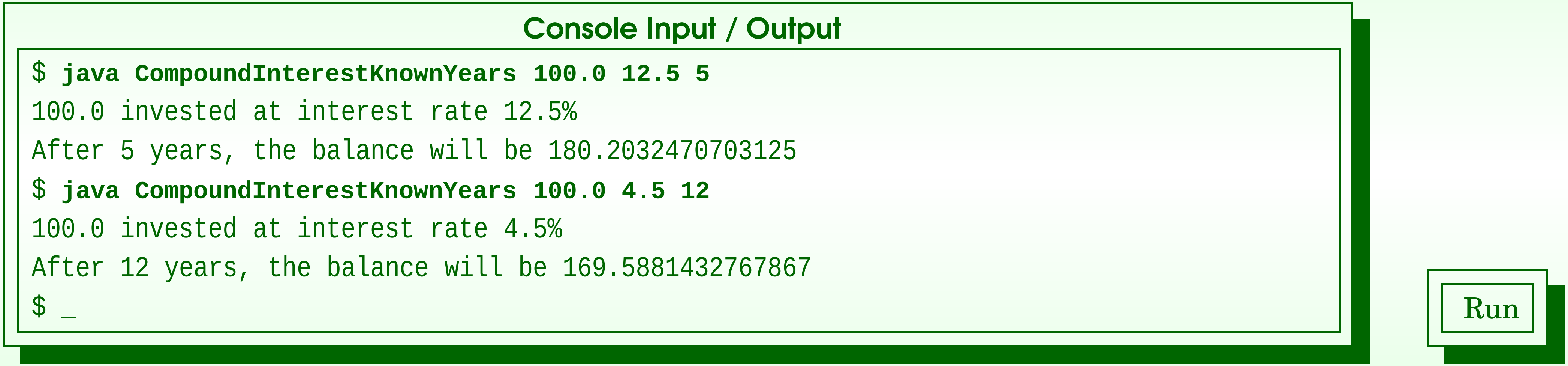
If so, which is better? What if we wanted to use the value of year inside the **loop** – would that affect your choice of which is best?



Coffee time: Could we have written it as this?!!

```
for (int year = 0; year < 2 * noOfYearsInvested; year = year + 2)
```

Trying it



```
$ java CompoundInterestKnownYears 100.0 12.5 5
100.0 invested at interest rate 12.5%
After 5 years, the balance will be 180.2032470703125
$ java CompoundInterestKnownYears 100.0 4.5 12
100.0 invested at interest rate 4.5%
After 12 years, the balance will be 169.5881432767867
$ _
```

Coursework: Power

(Summary only)

Write a program to raise a given number to the power of a second given number, without using `Math.pow()`.

Section 7

Example:
Average of a list of numbers

Aim

AIM: To show how to get the length of a **list**, note that an **index** can be a **variable**, and introduce **type cast**ing.

Average of a list of numbers

- Given **list** of **integer command line arguments**
  - reports their mean average.
- Compute sum in a **for loop**.
- Divide by number of numbers.

Command line arguments: length of the list

- The **command line argument**s are a **list** of strings.
- The length of a list is: name of list, dot, `length`.
- E.g. `args.length` is number of items in list `args`.

Command line arguments: list index can be a variable

- A **list** item **index** can be an **int variable** or **arithmetic expression**.
- E.g. sum **integer command line argument**s:

```
int sumOfArgs = 0;
for (int argIndex = 0; argIndex < args.length; argIndex = argIndex + 1)
  sumOfArgs = sumOfArgs + Integer.parseInt(args[argIndex]);
System.out.println("The sum is " + sumOfArgs);
```

- Can use same code to access different items, by, e.g., changing variable value in a loop.

Average of a list of numbers

- Sum of numbers is integer, number of numbers is integer.
- What happens when divide integer by integer?. . .

Type: casting an `int` to a `double`

- We can turn an **`int`** into a **`double`** by **cast**ing.
- E.g. `(double)5` is `5.0`.
- More likely to cast value of an **int variable** than **integer literal**!

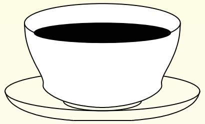
Average of a list of numbers

- No sense asking for average of no numbers.
  - So assume at least one.
- Sum is just first number to start with.
- Then add remaining numbers via for loop.

Average of a list of numbers

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

    for (int argIndex = 1; argIndex < args.length; argIndex = argIndex + 1)
      sumSoFar = sumSoFar + Integer.parseInt(args[argIndex]);

    System.out.println("The mean average is "
                       + sumSoFar / (double) args.length);
  }
}
```

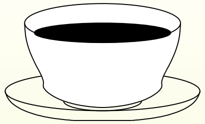
Average of a list of numbers



Coffee time: Recall that list indices start at 0. Convince yourself that we are correctly accessing the numbers in the list: should the for loop index start from 0? Why not? Would the following code for the for loop work? Is it better code?

```
    for (int argIndex = 1; argIndex <= args.length - 1; argIndex =
argIndex + 1)
```

Average of a list of numbers



Coffee time: What would happen if there were no numbers given on the command line? What sort of **exception** would be reported? What if we had started the value of `sumSoFar` at `0` and dealt with the first number inside the **loop**, instead of separately before the loop. What sort of exception would we *expect* to get now, if there were no command line arguments? Try it and see!

Trying it

Console Input / Output

```
$ java MeanAverage 100
The mean average is 100.0
$ java MeanAverage 100 500
The mean average is 300.0
$ java MeanAverage 34 67 12 904 -5 8375 -1249
The mean average is 1162.5714285714287
$ java MeanAverage 60 -100 40
The mean average is 0.0
$ _
```

Run

And no arguments?

Console Input / Output

```
$ java MeanAverage
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
        at MeanAverage.main(MeanAverage.java:5)
$ _
```

Run

Coursework: `Variance`

(Summary only)

Write a program to produce the variance of some given numbers.

Section 8

Example:
Single times table

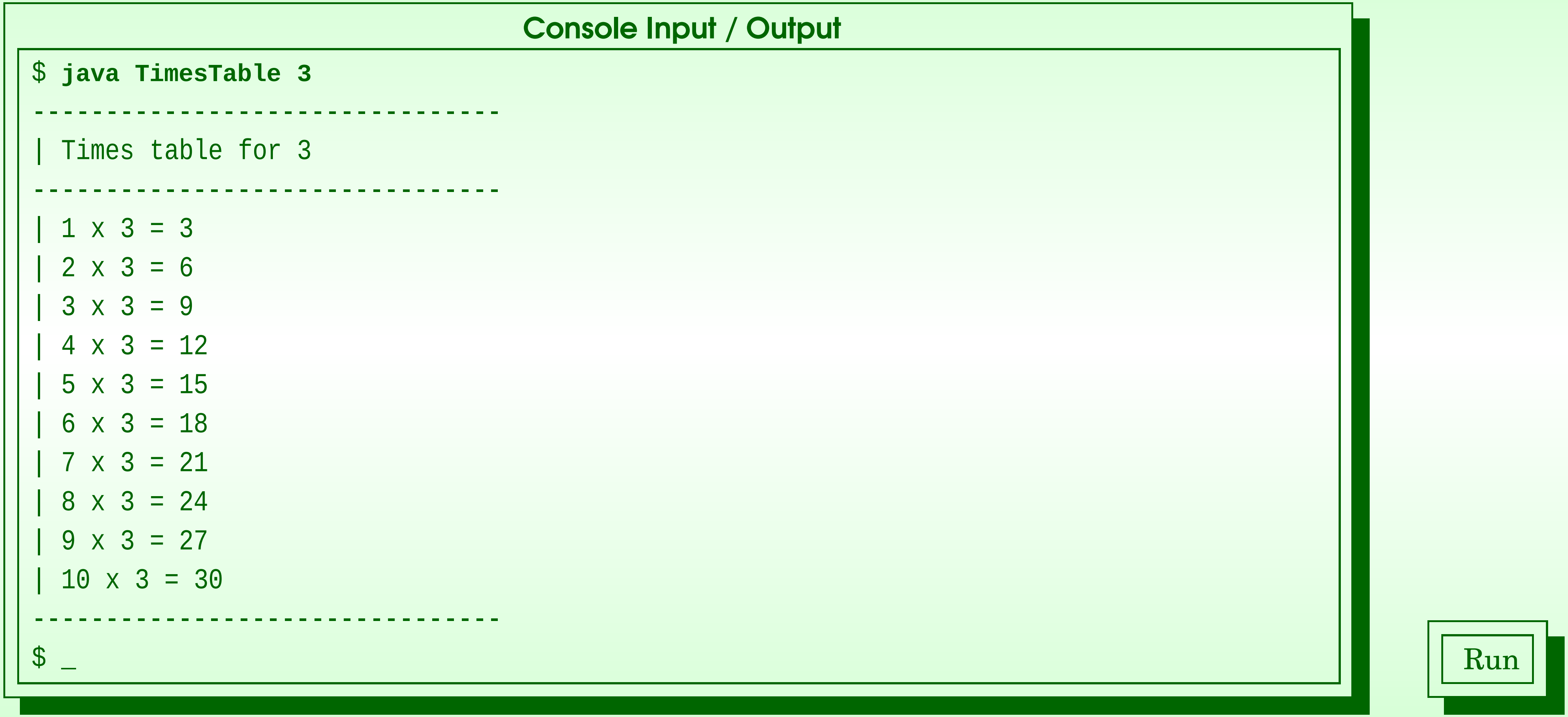
Aim

AIM: To reinforce the **for loop**.

Single times table

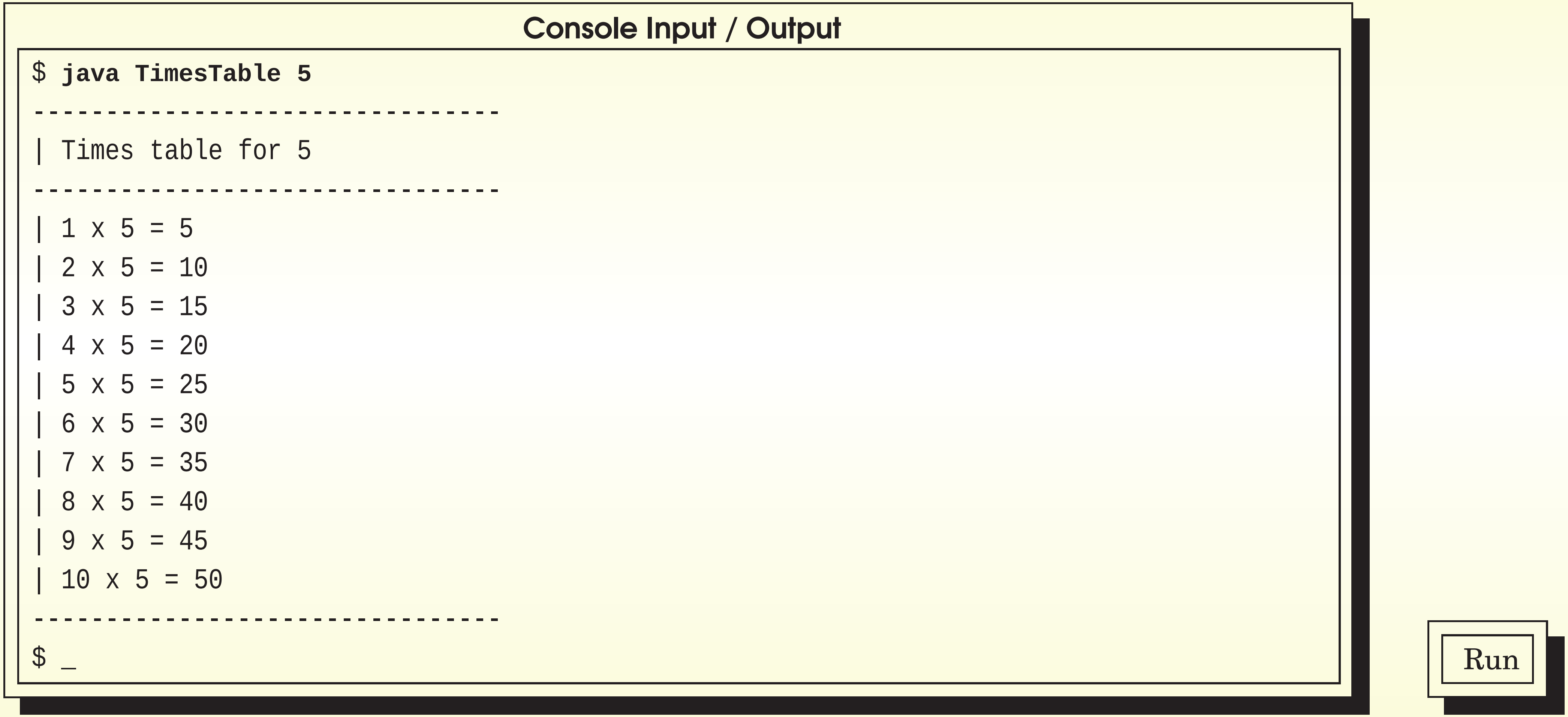
```java
public class TimesTable
{
  public static void main(String[] args)
  {
    int multiplier = Integer.parseInt(args[0]);

    System.out.println("--------------------------------");
    System.out.println("| Times table for " + multiplier);
    System.out.println("--------------------------------");
    for (int thisNumber = 1; thisNumber <= 10; thisNumber = thisNumber + 1)
      System.out.println("| " + thisNumber + " x " + multiplier
                         + " = " + thisNumber * multiplier);
    System.out.println("--------------------------------");
  }
}
```

Trying it



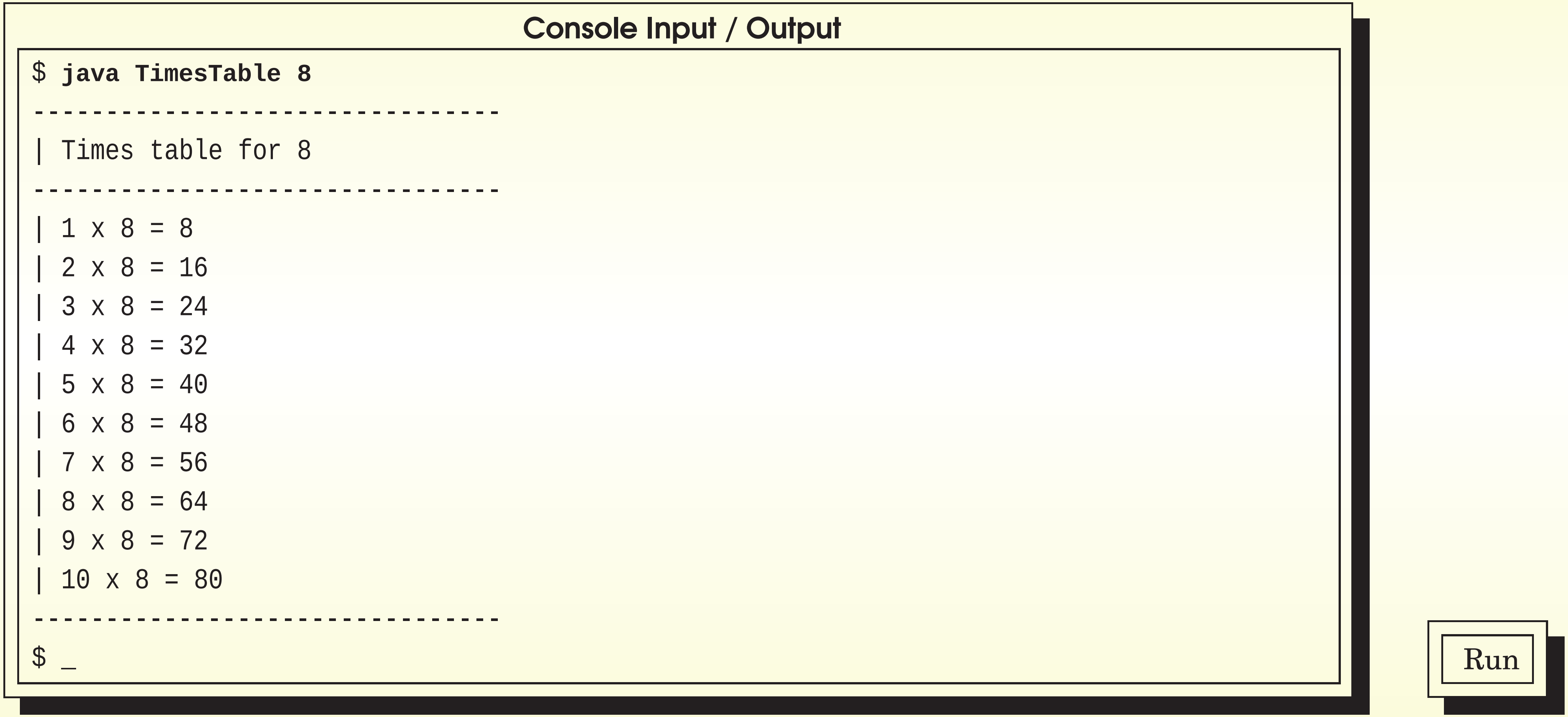
```
$ java TimesTable 3
--------------------------------
| Times table for 3
--------------------------------
| 1 x 3 = 3
| 2 x 3 = 6
| 3 x 3 = 9
| 4 x 3 = 12
| 5 x 3 = 15
| 6 x 3 = 18
| 7 x 3 = 21
| 8 x 3 = 24
| 9 x 3 = 27
| 10 x 3 = 30
--------------------------------
$ _
```

Trying it



```
$ java TimesTable 5
--------------------------------
| Times table for 5
--------------------------------
| 1 x 5 = 5
| 2 x 5 = 10
| 3 x 5 = 15
| 4 x 5 = 20
| 5 x 5 = 25
| 6 x 5 = 30
| 7 x 5 = 35
| 8 x 5 = 40
| 9 x 5 = 45
| 10 x 5 = 50
--------------------------------
$ _
```

Trying it



```
$ java TimesTable 8
--------------------------------
| Times table for 8
--------------------------------
| 1 x 8 = 8
| 2 x 8 = 16
| 3 x 8 = 24
| 4 x 8 = 32
| 5 x 8 = 40
| 6 x 8 = 48
| 7 x 8 = 56
| 8 x 8 = 64
| 9 x 8 = 72
| 10 x 8 = 80
--------------------------------
$ _
```

Coursework: `SinTable`

(Summary only)

Write a program to produce a sin table.

Section 9

Example:
Age history

Aim

AIM: To introduce the idea of documenting programs using **comment**s.

Code clarity: comments

- Layout and **indentation** enhance readability.
- So do **comments**
  - pieces of text ignored by the **compiler**.
- E.g.
  - at start of program: what it does, how it is used.
  - at each **variable** declaration: what it is used for.
  - within code: what next statements do.
- One form: // followed by comment text.
- E.g.

```
// This is a comment, ignored by the compiler.
```

Code clarity: comments: marking ends of code constructs

- Good idea to mark end of code constructs.
- Especially if long and doesn't all fit on screen. . .
- E.g.

```
public class SomeClass
{
  public static void main(String[] args)
  {
    ...
    while (...)
    {
      ...
      ...
```

Code clarity: comments: marking ends of code constructs

```
        ...
      } // while
      ...
    } // main

  } // class SomeClass
```

Age history

```java
// Program to print out the history of a person's age.
// First argument is an integer for the present year.
// Second argument is the birth year, which must be less than the present year.
public class AgeHistory
{
  public static void main(String[] args)
  {
    // The year of the present day.
    int presentYear = Integer.parseInt(args[0]);

    // The year of birth: this must be less than the present year.
    int birthYear = Integer.parseInt(args[1]);

```

Age history

```java
// Start by printing the event of birth.
System.out.println("You were born in " + birthYear);

// Now we will go through the years between birth and last year.

// We need to keep track of the year we are considering
// starting with the year after the birth year.
int someYear = birthYear + 1;

// We keep track of the age, starting with 1.
int ageInSomeYear = 1;

```

Age history

```
    // We deal with each year while it has not reached the present year.
    while (someYear != presentYear)
    {
      // Print out the age in that year.
      System.out.println("You were " + ageInSomeYear + " in " + someYear);

      // Add one to the year and to the age.
      someYear = someYear + 1;
      ageInSomeYear = ageInSomeYear + 1;
    } // while

```

Age history

```
  // At this point someYear will equal presentYear.
  // So ageInSomeYear must be the age in the present year.
  System.out.println("You are " + ageInSomeYear + " this year");
 } // main

} // class AgeHistory
```

Coffee time: What would happen if we ran the program with a birth year which is not **less than** the present year?

Trying it

Console Input / Output

```
$ java AgeHistory 2019 2018
You were born in 2018
You are 1 this year
$ java AgeHistory 2019 2000
 (Output shown using multiple columns to save space.)
You were born in 2000 | You were 7 in 2007  | You were 14 in 2014
You were 1 in 2001    | You were 8 in 2008  | You were 15 in 2015
You were 2 in 2002    | You were 9 in 2009  | You were 16 in 2016
You were 3 in 2003    | You were 10 in 2010 | You were 17 in 2017
You were 4 in 2004    | You were 11 in 2011 | You were 18 in 2018
You were 5 in 2005    | You were 12 in 2012 | You are 19 this year
You were 6 in 2006    | You were 13 in 2013
$ _
```

Run

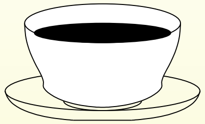
Trying it

- Try birth year **greater than** present year.
  - Run, kill after 1 second, show last 3 lines of output.

Console Input / Output

```
$ (java AgeHistory 2019 2020 & PID=${!}; sleep 1; kill $PID) | tail -3
You were 35653 in 37673
You were 35654 in 37674
You were 35655 in 37675
$ _
```

Run



Coffee time: Can you explain the program behaviour?

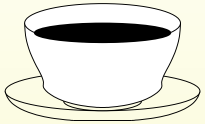
Trying it

- Repeat:

Console Input / Output

```
$ (java AgeHistory 2019 2020 & PID=${!}; sleep 1; kill $PID) | tail -3
You were 38244 in 40264
You were 38245 in 40265
You were 38246 in 40266
$ _
```

Run



Coffee time: Why is the result different? What would happen if we let the program run indefinitely? (Hint: is there a maximum value for `someYear`?)

Coursework: `WorkFuture`

(Summary only)

Write a program to print out all the years from the present day until the user retires.

Section 10

Example:
Home cooked Pi

Aim

AIM: To introduce various **shorthand operator**s for **variable update**s, have another example where we reveal the **pseudo code design**, and meet `Math.abs()` and `Math.PI`.

Home cooked Pi

- 15th century Indian mathematician Madhava of Sangamagrama discovered following sequence
  - rediscovered in 1673 by Gottfried Leibniz(**?**).

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \ldots$$

- More accurate with more terms, but never exact
  - each term jumps result either side of π, getting ever closer.
- Doesn't matter if don't know why it works – just implement correctly.

Home cooked Pi

- Not fastest **algorithm** for π, but interesting.
- Start with value 4.
- Subtract $\frac{4}{3}$.
- Add $\frac{4}{5}$.
- Etc.: each denominator is previous + 2, sign keeps swapping.
- Stop when difference between successive sums is **less than or equal** to given tolerance.

Home cooked Pi

- Some **pseudo code**:

```
obtain tolerance from command line
set up previousEstimate as value from no terms
set up latestEstimate as value from one term
while previousEstimate is not within tolerance of latestEstimate
  previousEstimate = latestEstimate
  add next term to latestEstimate
end-while
print out latestEstimate
print out the number of terms used
print out the standard known value of Pi for comparison
```

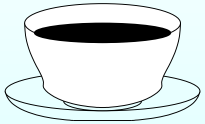
Home cooked Pi

- Make more concrete and add a **variable** to count terms:

```
double tolerance = args[0]
double previousEstimate = 0
double latestEstimate = 4
int termCount = 1
while previousEstimate is not within tolerance of latestEstimate
  previousEstimate = latestEstimate
  add next term to latestEstimate
  termCount = termCount + 1
end-while
s.o.p latestEstimate
s.o.p termCount
s.o.p the standard known value of Pi for comparison
```

Home cooked Pi

- To find next term, have two variables:
  - denominator
    - increase by two each time
  - sign of numerator.
    - alternate between 1 and -1.



Coffee time: What simple operation can we do to a variable to make it change the sign of its value?

Home cooked Pi

```
double tolerance = args[0]
double previousEstimate = 0
double latestEstimate = 4
int termCount = 1
int nextDenominator = 3
int nextNumeratorSign = -1
while previousEstimate is not within tolerance of latestEstimate
  previousEstimate = latestEstimate
  latestEstimate = latestEstimate + nextNumeratorSign * 4 / nextDenominator
  termCount = termCount + 1
  nextNumeratorSign = nextNumeratorSign * -1
  nextDenominator = nextDenominator + 2
end-while
```

Home cooked Pi

```
s.o.p latestEstimate
s.o.p termCount
s.o.p the standard known value of Pi for comparison
```

Home cooked Pi

- Only two bits to make more concrete
  - loop **condition**
  - standard known value of π.

Standard API: `Math`: `abs()`

- No Java **operator** to give **absolute value** of a number
  - i.e. ignore its sign.
- Instead `Math` contains `abs()`
  - takes a number and gives its absolute value.
- E.g.
  - `Math.abs(-2.7)` produces `2.7`
  - as does `Math.abs(3.4 - 0.7)`.

Home cooked Pi

- Our loop condition:

```
Math.abs(latestEstimate - previousEstimate) > tolerance
```

Standard API: `Math`: `PI`

- `Math` contains a constant called `PI`
  - most accurate value of π possible as a **`double`**.
- `Math.PI` is how we access it.
- E.g.:

```
double circleArea = Math.PI * circleRadius * circleRadius;
```

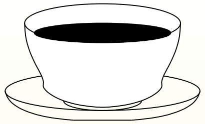
Statement: assignment statement: updating a variable: shorthand operators

- Java has **shorthand operators** for certain types of update.

Op.	Name	E.g.	Long meaning
`++`	postfix increment	`x++`	`x = x + 1`
`--`	postfix decrement	`x--`	`x = x - 1`
`+=`	compound assignment: add to	`x += y`	`x = x + y`
`-=`	compound assignment: subtract from	`x -= y`	`x = x - y`
`*=`	compound assignment: multiply by	`x *= y`	`x = x * y`
`/=`	compound assignment: divide by	`x /= y`	`x = x / y`

- Save a bit of typing – so what!
- Moreover: make program easier to read.
- (Historical efficient code motivation.)

Home cooked Pi



Coffee time: How many of these shorthand operators can be used in this program? Where? If we had known about them before this point, do you think we would have used them in our pseudo code?

Home cooked Pi

```
// A program to estimate Pi using Leibniz’s formula.
// Argument is desired tolerance between successive terms.
// Reports the estimate, the number of terms
// and the library constant for comparison.
public class PiEstimation
{
  public static void main(String[] args)
  {
    // The tolerance is the minimum difference between successive
    // terms before we stop estimating.
    double tolerance = Double.parseDouble(args[0]);

    // The result from our previous estimate, initially 0 for 0 terms.
    double previousEstimate = 0;

```

Home cooked Pi

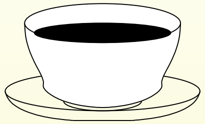
```java
    // The result from our latest estimate, eventually the final result.
    double latestEstimate = 4;

    // We count the terms, initially 1 for the 4.
    int termCountSoFar = 1;

    // The value of the next term denominator, initially 3.
    int nextDenominator = 3;

    // The sign of the next term, initially -ve.
    int nextNumeratorSign = -1;

```

Home cooked Pi

```java
    // Keep adding terms until change is within tolerance.
    while (Math.abs(latestEstimate - previousEstimate) > tolerance)
    {
      previousEstimate = latestEstimate;
      latestEstimate += nextNumeratorSign * 4.0 / nextDenominator;
      termCountSoFar++;
      nextNumeratorSign *= -1;
      nextDenominator += 2;
    } // while

```

Home cooked Pi

```java
    System.out.println("The estimated value of Pi to tolerance " + tolerance
                       + " is " + latestEstimate);
    System.out.println("The estimate used " + termCountSoFar + " terms");
    System.out.println("The library value of Pi is " + Math.PI);
  } // main

} // class PiEstimation
```



Coffee time: What would happen if we wrote 4 instead of `4.0` when computing the next term to add to the result? Without trying it, can you say what the output would be?

Trying it

Console Input / Output

```
$ java PiEstimation 0.1
The estimated value of Pi to tolerance 0.1 is 3.189184782277596
The estimate used 21 terms
The library value of Pi is 3.141592653589793
$ java PiEstimation 0.01
The estimated value of Pi to tolerance 0.01 is 3.1465677471829556
The estimate used 201 terms
The library value of Pi is 3.141592653589793
$ java PiEstimation 0.001
The estimated value of Pi to tolerance 0.0010 is 3.1420924036835256
The estimate used 2001 terms
The library value of Pi is 3.141592653589793
$ _
```

Run

- Number of terms grows rapidly with more accuracy – not fastest **algorithm**

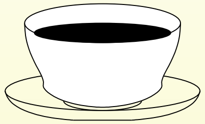
Trying it

- Note **scientific notation**.

Console Input / Output

```
$ java PiEstimation 0.00001
The estimated value of Pi to tolerance 1.0E-5 is 3.141597653564762
The estimate used 200001 terms
The library value of Pi is 3.141592653589793
$ java PiEstimation 0.000001
The estimated value of Pi to tolerance 1.0E-6 is 3.1415931535894743
The estimate used 2000001 terms
The library value of Pi is 3.141592653589793
$ _
```

Run



Coffee time: How many decimal places accuracy would you expect to get from the tolerance **command line argument** given in that last test? Does this tally with the results?

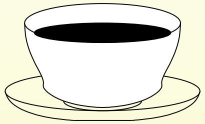
Trying it

- More decimal places:

Console Input / Output

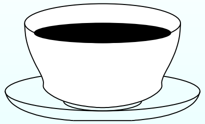
```
$ java PiEstimation 0.0000001
The estimated value of Pi to tolerance 1.0E-7 is 3.1415927035898146
The estimate used 20000001 terms
The library value of Pi is 3.141592653589793
$ java PiEstimation 0.00000001
The estimated value of Pi to tolerance 1.0E-8 is 3.1415926485894077
The estimate used 199999998 terms
The library value of Pi is 3.141592653589793
$ _
```

Run



Coffee time: Did you notice that the number of terms from the last test has broken the pattern from the previous ones? Might this suggest something about accuracy?

Trying it



Coffee time:	As we ask for more accuracy, the program takes longer to **run**: about 10 times more terms for each extra decimal place! What is the specific danger if we ask for too much accuracy? (Hint: is there a maximum value for `nextDenominator`? Also, remember that **`double`**s are only approximations of **real** numbers.)

Coursework: Shorthand operators

(Summary only)

Go through all the previous programs in this chapter to see where **shorthand operators** could have been used.

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.