List of Slides

20 Java concepts
21 Expression: arithmetic: remainder operator
22 Java concepts
23 **Section 3:** Program design concepts
24 Aim
25 Program design concepts
26 Program design concepts
28 Designing the variables
29 Designing the variables
30 Designing the variables
31 Designing the variables
32 Designing the variables
33 Designing the variables
34 Designing the variables
35 Designing the variables
36 Designing the variables
37 Designing the variables
38 Designing the variables

Title

Java Just in Time

John Latham

November 5, 2018

Chapter 9

Consolidation of concepts so far

Chapter aims

- Consolidate concepts covered so far.
  - opportunity to decide what you should reread.
- Introduce a few more simple concepts.
- Look specifically at process of **design**ing programs
  - mostly so far done via osmosis.

Section 2

Java concepts

Aim

AIM: To consolidate and summarize the Java concepts introduced in the preceding chapters. We also introduce some more simple concepts which were not covered before, but which compliment those that were – such as the **`float` type**.

Java concepts

- Read chapter to review what already has been covered.
- Take time also to meet new concepts
  - some of them will be used in later chapters.
- Java source code lexical details. . .
- Classes, methods, types, variables and layout. . .

Type: `long`

- The **type** `int` uses four **byte**s, 32 **binary digit**s.
  - Gives range -2^{31} through to $2^{31}-1$.
  - $2^{31}-1$ is 2147483647.
- The type `long` represents **long integer**s using eight bytes.
  - Gives range -2^{63} through to $2^{63}-1$.
  - $2^{63}-1$ is 9223372036854775807.
- A **long literal** has `L` on the end
  - e.g. `-15L`, `2147483648L`.

Type: `short`

- The **type `short`** represents **short integer**s using two bytes.
  - Gives range -2^{15} through to $2^{15}-1$.
  - $2^{15}-1$ is 32767.
- Useful when have huge number of **integer**s lying in that range
  - uses less space.

Type: `byte`

- The **type** `byte` represents integers using one byte.
  - Gives range -2^7 through to $2^7 - 1$.
  - $2^7 - 1$ is 127.

Type: char

- Characters are represented by **type `char`**.
- A **char variable** can store a single **character**.

Type: char: literal

- A **character literal** is written using single quotes.
- E.g. `'J'`

Variable: char variable

- Can have **char variable**s, e.g.

```
char firstLetter = 'J';
```

Type: char: literal: escape sequences

- A **character literal** can have **escape sequence**s
  - same as for **string literal**s.
- E.g.

```
char backspace = '\b';          char tab = '\t';
char newline = '\n';            char formFeed = '\f';
char carriageReturn = '\r';     char doubleQuote = '\"';
char singleQuote = '\'';        char backslash = '\\';
```

Type: `float`

- The **type** `float` stores **real** numbers
  - uses **single precision floating point representation**
  - only four **byte**s per number
- The **type** `double`
  - uses **double precision** – far more accurate
  - needs eight bytes per number.
- A **float literal** ends in `f` or `F`
  - e.g. `0.0F`, `-129.934F`, `98.2375f`

Java concepts

- Typical single class program:

Example-class.java

```
001: // Comments here to say what the program does.
002: // This may take several lines.
003: public class ProgramName
004: {
005:   // Each class variable should have comments saying what it is used for.
006:   private static int someVariable;
007:
008:   // Variables can be initialized when they are declared.
009:   private static double someOtherVariable = someVariable * 100.0;
010:
011:
```

Example-class.java

```
012:   // Each method should have comments saying what it does.
013:   // If it does not return a result, we write the word void.
014:   public static void someMethod(int aParameter)
015:   {
016:     // Local variables should also have a comment.
017:     int aLocalVariable;
018:
019:     ... Method instruction statements go here.
020:     // Comments are used to help make tricky code readable.
021:     ...
022:   } // someMethod
023:
024:
```

Example-class.java

```
025:   // Methods which are only of use in this class/program should be private.
026:   // If it returns a result, we state the type in the heading.
027:   private static double someOtherMethod(boolean parameter1, int parameter2)
028:   {
029:     double aLocalVariable;
030:     ...
031:     return something;
032:   } // someOtherMethod
033:
034:
```

Example-class.java

```
035:   // The program always starts its execution at the main method.
036:   // The parameters are the command line arguments of the program.
037:   public static void main(String[] args)
038:   {
039:     ...
040:   } // main
041:
042: } // class ProgramName
```

Java concepts

- Statements. . .
- Expressions. . .

Expression: arithmetic: remainder operator

- Another **arithmetic operator** – **remainder**: %
  - also known as **modulo**.
- Given two **integer operand**s
  - yields remainder after dividing first by second.
- E.g.

```
public static boolean isEven(int number)
{
  return number % 2 == 0;
} // isEven
```

Java concepts

- Errors...
- Standard classes...

Section 3

Program design concepts

Aim

AIM: To look more formally at the process of **design**ing an **algorithm** and writing a program. In particular, we look closely at **designing variables**.

Program design concepts

- Seen lots of example programs, done coursework
  - by osmosis you have begun to learn skill of programming.
- Now we formalize this a little: how do we write a program / **method**?
  - Really: how do we write an **algorithm**?
- Guidelines – not a recipe:

Program design concepts

1. Understand the problem – obviously you cannot possibly get the computer to solve it otherwise. (It is amazing how many people overlook this!)
2. Ask yourself how you would solve the problem if you were not going to program a computer. If you cannot answer this then almost certainly you will fail to get the computer to do it.
3. Consider whether the way you would do it is the way the computer should do it. Often it is, because we humans are actually very good at being lazy, and finding the best way to do something when we put our minds to it. On the other hand, sometimes the way a computer would do it is different because of the nature of computers compared with us – they are very quick at doing mindless things.
4. Decide how the computer should do it, i.e. what is the **basic method** your algorithm will use.

Program design concepts

5. Design the **variable**s used in the algorithm, very carefully. We shall say more about this shortly.
6. Design the algorithm itself in **pseudo code** . If the previous step is done properly, this one will almost do itself.
7. Finally implement the algorithm in code – this should be the easiest part if the previous steps have been followed carefully.

Designing the variables

- A **variable** is just a named location
  - your code can stick a value there
    - (of appropriate **type**)
  - and change it from time to time.

Designing the variables

- No, No, NO, NOO, NOOOOO, NOOOOOOOOOOOOO!!!!!!!
  - Far too computer centric view to help us **design**.
  - This is *exactly* what is wrong with many programs.

Designing the variables

- Focus on *meaning* of **data** by **designing variables**
  - get this wrong and our programs will
    - not work
    - be badly written – hard to fix
    - essentially useless!
- Classic mistake:
  - recognise variable needed of certain type
  - *then* quickly choose its name
  - never *carefully* consider what it means with respect to problem
  - Result:
    - cryptic names
    - **bug**gy code.

Designing the variables

- Java invites this attitude: you write type of variable before name!
  - Resist!!!!
- We must really think about these the other way round.

Designing the variables

- Example code, written by computer centric programmer.
  - Assume `getCurrentHumidity()` and `PRECIPITATION_THRESHOLD`

```
boolean b;
if (getCurrentHumidity() > PRECIPITATION_THRESHOLD)
  b = true;
else
  b = false;
...
if (b == true)
  System.out.println("Take your umbrella!");
```

Designing the variables

- Is that code okay?
  - Well it does work. . . maybe.
- Recall how naughty code like `if (b == true)` is?
  - Terrible!

Designing the variables

- Let's choose better name for **boolean variable**, `b`
  - what does it really mean?

```
boolean rainIsLikely;
if (getCurrentHumidity() > PRECIPITATION_THRESHOLD)
  rainIsLikely = true;
else
  rainIsLikely = false;
...
if (rainIsLikely == true)
  System.out.println("Take your umbrella!");
```

Designing the variables

- How often do you say

"Do you think it will rain equals true?"

?

Designing the variables

- Do not need **if else statement**.

```
boolean rainIsLikely
  = getCurrentHumidity() > PRECIPITATION_THRESHOLD;
```

- Do not need == `true`.

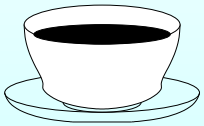
```
if (rainIsLikely)
  System.out.println("Take your umbrella!");
```

- Poor design led to poor *style*.

Designing the variables

- Example: where it leads to buggy code
  - we wish to count items – here is pseudo code.

```
int i = 1
while there are more items
  get an item
  i++
end-while
output "There were "  i " items"
```



Coffee time: What is wrong with this design and how should it be fixed? There are two obvious ways to fix it. Which is best?

Designing the variables

- Think carefully what `i` really is...
  - number of items we have had so far.

```
int itemCountSoFar = 1
while there are more items
  get an item
  itemCountSoFar++
end-while
output "There were " itemCountSoFar " items"
```

- Ah ha! First line is a *lie* – that is where error is.

Designing the variables

- Or, `i` is number of *next* item to be obtained.

```
int nextItemNumber = 1
while there are more items
  get an item
  nextItemNumber++
end-while
output "There were " nextItemNumber " items"
```

- Now last line is a lie – *that* is where error is!

Designing the variables

- `i` was an ambiguous name
  - because of lack of design about what it really means
  - was treated with different interpretation in different parts of design.
- Which interpretation is best? `itemCountSoFar` or `nextItemNumber`?
  - subjective
  - but, always a count of items so far, not always a next item ....

Designing the variables

- Treat variables with respect – design them carefully.
- They represent identifiable entities in the problem.
- If cannot fully and precisely describe meaning in one sentence
  - then probably is a bad variable
  - think again!
- That sentence will be your comment for the variable.
- Condense it to a few words to make variable name.
- *Then* you can identify its type!

Designing the variables

- If you design variables well then algorithm code almost writes itself
  - just ensure value of every variable reflects its meaning at all times
  - you cannot have a bug!
- Well, you might make a mistake
  - but if you do, it will be easy to spot.
- Plus, code is easier to read and maintain too!

Designing the algorithm

- Express algorithm at high level of **abstraction**.
- Add more detail – lower abstraction level.
- Until low enough that implementation in Java straightforward.
- E.g. our first example was `MinimumBitWidth`:

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

Designing the algorithm

- Then added more detail:

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

- Close enough to Java so easy to implement.

Designing the algorithm

- Next example was `PiEstimation`:

```
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
```

Designing the algorithm

- We lowered level of abstraction:

```
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++
end-while
s.o.p latestEstimate
s.o.p termCount
s.o.p the standard known value of Pi for comparison
```

Designing the algorithm

- And again:

```
double tolerance = args[0]
double previousEstimate = 0
double latestEstimate = 4
int termCount = 1
int nextDenominator = 3
int nextNumeratorSign = -1
while Math.abs(latestEstimate - previousEstimate) > tolerance
  previousEstimate = latestEstimate
  latestEstimate += nextNumeratorSign * 4 / nextDenominator
  termCount++
  nextNumeratorSign *=  -1
  nextDenominator += 2
end-while
```

Designing the algorithm

```
s.o.p latestEstimate
s.o.p termCount
s.o.p Math.PI
```

Designing the algorithm

- Another example was `TimesTable`:

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

- Later we implemented this using separate methods.
- Steps in early abstract designs often make good choice for methods:

TimesTable.java

```
001: // Program to print out a neat multiplication table.
002: public class TimesTable
003: {
004:   // The size of the table -- the number of rows and columns.
005:   private static int tableSize = 12;
006:
007:
008:   // The main method implements the top level structure of the table.
009:   public static void main(String[] args)
010:   {
011:     // Top line.
012:     printLine();
013:
014:     // Column headings.
015:     printColumnHeadings();
```

TimesTable.java

```
016:
017:     // Underline headings.
018:     printLine();
019:
020:     // Now the rows.
021:     for (int row = 1; row <= tableSize; row++)
022:       printRow(row);
023:
024:     // Bottom line.
025:     printLine();
026:   } // main
...
076: } // class TimesTable
```

Designing the algorithm

- Generalize algorithm design process:
  1. Identify main steps of algorithm.
  2. Express algorithm in terms of main steps.
  3. Separately expand on each step using same process
     i.e. identify its main steps...
- Sometimes called **top down stepwise refinement**.

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.