School of Computing Sciences

CMP-4008Y — Programming 1

LAB EXERCISES 3 — Using Objects and Problem Solving

Level of difficulty: (*) - easy; (**) - intermediate; (***) - decidedly tricky. Questions do not necessarily require the use of a computer. It is a good idea to read through the exercises before the laboratory session.

1. (*) Given that one kilometer is 0.6214 miles, one pint is 0.5683 litres, one gallon is comprised of 8 pints and that a litre of petrol costs £ 1.34, write a program to compute the cost of driving 100 kilometers in a car with an efficiency of 30 miles per gallon. The program should display:

   - the efficiency of the vehicle in kilometers per gallon to an accuracy of 2 decimal places;
   - the distance travelled per liter of petrol to the nearest meter;
   - the cost of driving 100 kilometers using a format appropriate to the current locale of the computer.

   Note that the difficulty of this question lies mostly in the problem solving part - *"working out what you want the computer to do"*. The coding part - *"working out how to tell the computer to do it"* (i.e. translating the design, step-by-step, into Java), should be comparatively straightforward. Don't try to do both steps at the same time!

2. (*) Problem solving (without a computer): In the lecture, we saw an algorithm for solving a simple maze, provided the mouse started facing the entrance to the maze. However, the original specification indicated that the mouse starts at an arbitrary location outside the maze. We therefore need an algorithm for locating the maze and for finding the entrance to the maze. Assume that the maze lies in a walled garden with no exits and that the maze is located somewhere within the garden, but not touching any of the walls. Write pseudo-code for algorithms for finding the maze, and for positioning the mouse facing the entrance. Simulate the algorithm by hand, using a simple testing scenario of your own devising. Is your algorithm efficient? Can you find a better solution?

3. (**) Strings are stored as a sequence of `char` values that can't be changed. They are *immutable*. You can find the length of a string, access a value or substring, but you can't change any part of the string. What output do you expect from the following code? Note you may need to investigate the API documents to find out what some of the methods are for.

```java
String text = "dinosaur";
System.out.println("String length: "
   + text.length());
System.out.println("character at index 3: "
   + text.charAt(3));
System.out.println("character value at index 3: "
   + text.codePointAt(3));
System.out.println("Starts with 'dino'? "
   + text.startsWith("dino"));
System.out.println("Substring: "
   + text.substring(2, 5));

```

4. (**) Characters are sometimes identified by their hexadecimal Unicode value. An `int` can be cast to a `char` ("code point" is a technical term for a numeric value used to represent a particular character):

```java
int codePoint = 0x00E9;

System.out.format("code point: %X ", codePoint);

char c = (char)codePoint;

System.out.println("character: " + c);

```

What output do you expect from the following?

```java
char c = 'n';
int codePoint = (int)c;

if (c == codePoint)
{
   System.out.format("%c == %d%n", c, codePoint);
}

if (c != codePoint)
{
   System.out.format("%c != %d%n", c, codePoint);
}

String text = "dinosaur";
c = text.charAt(3);
codePoint = text.codePointAt(3);

if (c == codePoint)
{
   System.out.format("%c == %d%n", c, codePoint);
}
else
{
   System.out.format("%c != %d%n", c, codePoint);
}

```

5. (**) What is the difference between this bit of code that uses a `String`:

```java
String string = "penguin";
String subString = "engu";

int index = string.indexOf(subString);

System.out.println("Substring starts at index "+index);
System.out.println("Substring length: "+subString.length());

System.out.println(string);
string = string.replace(subString, "uff");
System.out.println(string);

```

and this bit of code that uses a `StringBuilder`:

```java
StringBuilder builder = new StringBuilder("penguin");
String subString = "engu";

int index = builder.indexOf(subString);

System.out.println("Substring starts at index "+index);
System.out.println("Substring length: "+subString.length());

System.out.println(builder);
builder.replace(index, index+subString.length(), "uff");
System.out.println(builder);

```

6. (***) Create a `StringBuilder` object that initially stores the sequence of characters `wombat`, then:

   - use the appropriate method to insert the string `le` after the fourth character;
   - use the appropriate method to delete the last two characters in the sequence;
   - use the appropriate method to append the slightly smiling emoticon (0x1F642) followed by the dromedary camel character (0x1F42A);
   - Find the code points for your favourite emoticons or emojis and add those too, for good measure.

   Use `System.out.println` to display the result.

7. (***) You can specify a character by its hexadecimal value in a literal string using the notation `\u`*nnnn* . (Exactly four hexadecimal digits are required.) For example:

```java
String text = "p\u00E2t\u00E9";

```

Values larger than 0xFFFF need to be split into high and low surrogate pairs. What do you think the output from the following code will be?

```java
String text = "\uD83D\uDE42";
char c = text.charAt(0);
int codePoint = text.codePointAt(0);

System.out.println(text);

if (c == codePoint)
{
   System.out.println((int)c + " == " + codePoint);
}

if (c != codePoint)
{
   System.out.println((int)c + " != " + codePoint);
}

```

Hint: search the on-line help to find out what `charAt` and `codePointAt` do, and look up UTF-16 in Wikipedia.

Gavin Cawley
1st October 2024