CS106A Handout 22S
Winter 2013-2014 February 7, 2014

CS106A Midterm Exam Solutions

Problem One: Tower-Building Karel (20 Points)

Here are two possible solutions: one using **`beepersInBag`** and one without:

```
import stanford.karel.*;

public class TowerBuildingKarel extends
SuperKarel {
    public void run() {
        while (frontIsClear()) {
            buildOneTower();
            move();
        }
        buildOneTower();
    }

    private void buildOneTower() {
        turnLeft();
        pickUpAllBeepers();
        placeBeepers();
        returnHome();
        turnLeft();
    }

    private void pickUpAllBeepers() {
        while (beepersPresent()) {
            pickBeeper();
        }
    }

    private void placeBeepers() {
        while (beepersInBag()) {
            putBeeper();
            if (frontIsClear()) {
                move();
            }
        }
    }

    private void returnHome() {
        turnAround();
        while (frontIsClear()) {
            move();
        }
    }
}
```

```
import stanford.karel.*;

public class TowerBuildingKarel extends
SuperKarel {
    public void run() {
        while (frontIsClear()) {
            buildOneTower();
            move();
        }
        buildOneTower();
    }

    private void buildOneTower() {
        if (beepersPresent()) {
            while (beepersPresent()) {
                pickBeeper();
                if (beepersPresent()) {
                    addNextBeeper();
                    comeBackHome();
                }
            }
            putBeeper();
        }
    }

    private void addNextBeeper() {
        turnLeft();
        while (beepersPresent()) {
            move();
        }
        putBeeper();
    }

    private void comeBackHome() {
        turnAround();
        while (frontIsClear()) {
             move();
        }
        turnLeft();
    }
}
```

Common mistakes on this problem included missing the very first or last tower on 1st Street due to a fencepost error, accidentally crashing into the top wall when Karel tries to build a tower that just barely fits into the world, or looping infinitely when trying to assemble a tower.

Problem Two: Jumbled Java hiJinks **(20 Points Total)**

(i) Expression Tracing **(6 Points)**

`4 / 7 * (double)7 / 4`	`0.0`
`137 / 42 == 0 && 137 / 0 == 42`	`false`
`1 + 2 + "1 + 2" + 1 + 2`	`"31 + 212"`
`(char)('3' - '0' + 'A')`	`'D'`

The first expression evaluates to 0.0 because the integer division involved in computing 4 / 7 rounds down to 0. Multiplying this by the double value 7 yields 0.0, and dividing by the double value 4 then yields 0.0 again.

The second expression evaluates to **`false`** because short-circuit evaluation prevents Java from evaluating the expression that divides by zero in the second half.

To understand the result of the third expression, remember that Java always evaluates + from left-to-right. This means that the expression is evaluated as

```
((((1 + 2) + "1 + 2") + 1) + 2)
```

This means that the leftmost 1 + 2 evaluates to 3, then is concatenated with the string "1 + 2" to form the string "31 + 2". We then append a 1 to get "31 + 21", and finally append a 2 to get "31 + 212.".

The final expression evaluates to 'D' because subtracting '3' and '0' yields 3. Adding 3 to 'A' gives the numeric code for 'D', and casting the result to a **`char`** then produces the character 'D'.

(ii) Program Tracing **(14 Points)**

```
poorMan = 403
beggarMan = 500
soldier = 903
```

Problem Three: The Saint Petersburg Game (25 Points)

Here is one possible solution:

```
import acm.program.*;
import acm.util.*;

public class SaintPetersburgGame extends ConsoleProgram {
    private static final int TARGET_AMOUNT = 20;

    public void run() {
        int totalWinnings = 0;
        int numRounds = 0;

        while (totalWinnings < TARGET_AMOUNT) {
            totalWinnings += playGame();
            println("Your total is $" + totalWinnings);

            numRounds++;
        }
        println("It took " + numRounds + " games to earn $" + TARGET_AMOUNT);
    }

    private int playGame() {
        RandomGenerator rgen = RandomGenerator.getInstance();
        int winnings = 1;

        while (rgen.nextBoolean()) {
             winnings *= 2;
        }

        println("This game, you won $" + roundWinnings);
        return winnings;
    }
}
```

Common mistakes included generating a coin flip incorrectly (for example, as a real value in the range from 1.0 to 2.0), not flipping the coin inside the game loop, and forgetting to reset the per-game winnings to 1 after each game. Another common mistake was stopping at exactly $20, which might never happen (for example, if you win $32 on your first game).

Problem Four: Subdivision **(25 Points)**

Here is one solution:

```
import acm.program.*;
import acm.graphics.*;
import java.awt.*;
import java.awt.event.*;

public class Subdivision extends GraphicsProgram {
    public void run() {
        createCircle(0, 0, getWidth(), getHeight());
        addMouseListeners();
    }

    private void createCircle(double x, double y, double width, double height) {
        GOval circle = new GOval(x, y, width, height);
        circle.setFilled(true);
        add(circle);
    }

    public void mouseClicked(MouseEvent e) {
        GObject hit = getElementAt(e.getX(), e.getY());
        if (hit != null) {
             replaceObject(hit);
        }
    }

    private void replaceObject(GObject hit) {
        remove(hit);

        double size = hit.getWidth() / 2.0;
        double x = hit.getX();
        double y = hit.getY();

        createCircle(x, y, size, size);
        createCircle(x + size, y, size, size);
        createCircle(x, y + size, size, size);
        createCircle(x + size, y + size, size, size);
    }
}
```

A very common mistake in this program was not using **`getElementAt`** to detect what circle was clicked on. Another frequent error was forgetting to check the return value of **`getElementAt`** to ensure that it wasn't **`null`**.

Problem Five: Grade-School Arithmetic (30 Points)

One possibility:

```
private String addIntegerStrings(String firstNum, String secondNum) {
    String result = "";
    int carry = 0;

    for (int i = firstNum.length() - 1; i >= 0; i--) {
        int firstDigit  = firstNum.charAt(i) - '0';
        int secondDigit = secondNum.charAt(i) - '0';

        int sum = firstDigit + secondDigit + carry;

        result = (sum % 10) + result;
        carry = sum / 10;
    }

    if (carry != 0) {
        result = carry + result;
    }

    return result;
}
```

This program is fairly intricate and there were several common mistakes that were easy to make:

- Building up the result string in the wrong order – for example, returning 731 as the sum of 90 and 47. This is usually due to adding each new digit to the wrong side of the result string.
- Converting the digits in the string into their numeric values incorrectly. The character '0' does not have numeric value 0, so adding the digits of the numbers directly would lead to incorrect results.
- Forgetting to carry a final 1 at the end of the sum. If you add 9 and 4, you get a unit's digit of 3 and a carry of 1. Since the for loop ends at this point, you need to somehow include logic to add a 1 to the start of the result.