Introduction to Computer Science I
Spring 2010
Sample mid-term exam — Answer key

1. [Question:] (15 points) Consider the code fragment below. **Mark each location** where an *automatic cast* will occur. Also find each location where an *explicit cast* must be inserted for the code to compile successfully, and **correct the line** with that explicit cast included. [Note: Be aware that some lines may require more than one cast!]

```
byte b = 13;
int i = b;
short s = i;
int i2 = s + i;
short s2 = s + 3;
boolean b2 = (i < s);
```

[Answers and discussion:] People frequently were confused about which casts were automatic and which had to be forced/explicit. Many people indicated that a particular line involved an automatic cast, but they did not make clear *to which portion* of the expression that cast would apply, thus losing some credit. We will address one line at a time:

- `byte b = 13;`
  No cast is performed or needed here. The compiler identifies `13` as a valid `byte` constant, and so this is a case of assigning a `byte` value into a `byte` space. If you believed that the `13` would be considered an `int`, then you would have to insert a *forced cast* to receive partial credit. Claiming an *automatic cast* here is not sensible.
- `int i = b;`
  An **automatic cast** on `b` occurs here, converting it from a `byte` to an `int`.
- `short s = (short)i;`
  A **forced cast** must be inserted on `i` to make that `int` value fit into a `short` space.
- `int i2 = s + i;`
  In order to add a `short` to an `int`, the compiler will perform an **automatic cast** on `s`, converting it to an `int` so that addition is really performed on two `int` values.
- `short s2 = (short)(s + 3);`
  Two casts must occur here. First, addition is only performed on `int` values. So, `s` is **automatically cast** to an `int`. Second, to assign the result of the `int` addition into a `short` space, a **forced cast** must be performed on the addition expression.
- `boolean b2 = (i < s);`
  Comparisons must be performed on like types. Therefore, `c` is **automatically cast** to an `int` so that it may be compared to `i`. As an alternate answer, one could insert a **forced case** on `i`, making it into a `short`, thus comparing two `short` values.

2. [Question] (15 points) Consider the Java code below and answer the questions that follow.

```
System.out.print("Enter a value for a: ");
int a = keyboard.nextInt();
System.out.print("Enter a value for b: ");
int b = keyboard.nextInt();

if (a < 0) {
    a = -a;
}

int i = 0;
while (i < a) {

  int k = 0;
  int j = 100;
  while (j >= b) {
    k = j * 2;
    System.out.print(k);
    System.out.print('-');
    System.out.println(j);
    j = j - 1;
  }

  k = k + 1;
  i = i + 1;

}
```

(a) What is the output of this code if the user enters 2 for `a` and 98 for `b`?

(b) What is the output of this code if the user enters -3 for `a` and 100 for `b`?

[Answer] For each case:

(a)
```
200-100
198-99
196-98
200-100
198-99
196-98
```

(b)
```
200-100
200-100
200-100
```

3. [Question] (10 points) The following method contains an error that will prevent it from compiling. **Find and correct it**.

```
public static int quux () {

    int x = 0;
    while (x <= 0) {

        System.out.print("Enter a value: ");
        x = keyboard.nextInt();
        if (x < 0) {
            int y = x;
        } else {
            y = -x;
        }

    }

    return y;

}
```

[Answer] The scope of y is too narrow. It ceases to exist at the closing brace before the `else` keyword, and thus the use of it in the *else branch* and in the `return` statement outside of the *while* loop is invalid. To fix it, `x` must be declared before the *while* loop, and its previous declaration must be turned into a mere assignments:

```
public static int quux () {

    int x = 0;
    int y = 0;
    while (x < 100) {

        System.out.print("Enter a value: ");
        x = keyboard.nextInt();
        if (x > 100) {
          y = x;
        } else {
          y = 2 * x;
        }

    };

    return y;

}
```

4. [Question] (30 points) **Write a method** named `printTable` that accepts a *height* and a *width* as parameters and then prints a table of those dimensions, where the table follows the following pattern:

```
1 2 3 4 5
2 4 6 8 10
3 6 9 12 15
4 8 12 16 20
```

The above table is the result of calling `printTable(4, 5)`.

[Answer] Although there are many ways to write a method that perform this task, here is one of the simplest:

```
public static void printTable (int height, int width) {

  for (int row = 1; row <= height; row++) {
    for (int column = 1; column <= width; column++) {
      System.out.print((row * column) + " ");
    }
    System.out.println();
  }

} // end printTable
```

5. (30 points) We want to write a program that allows the user to enter a list of positive numbers and then prints out the *mean* number from that list. That is, if the user enters the values 7, 4, 3.5, and 11.5, then the mean is $\frac{7+4+3.5+11.5}{4} = \frac{26}{4} = 6.5$.
The `main` method of this program is:

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

    double x = getMeanFromUser();
    System.out.println("Max = " + x);

}
```

Write the method `getMeanFromUser`. The user should be able to enter an arbitrary number of positive values. As soon as the user enters a non-positive value, the program should accept that value as an indication that the user has no more values to enter. After the user has entered all of her numbers, the method should return the mean value entered. Do not worry about a user that enters non-numeric values.

[ANSWER] Once again, there is more than one way to skin a cat, but here is one straightforward solution:

```
public static double getMeanFromUser () {

  double userEntry = 0.0;
  double sum = 0.0;
  int numberEntries = 0;
  while (userEntry > 0.0) {

    System.out.print("Enter next value (or a negative value if finished): ");
    userEntry = keyboard.nextDouble();
    if (userEntry > 0.0) {
      sum = sum + userEntry;
      numberEntries++;
    }

  }

  return sum / numberEntries;

} // end getMeanFromUser
```