**CS201**
**Arrays Part II**

**Random Trivia Game**

Let's refine our trivia game, and say that we would like to randomly select four questions out of all of the questions that we loaded, ask each to the player, output if the player is correct or not, and then output the total score. This doesn't make a lot of sense with just 5 questions, but if we had a lot more questions it would be make the game somewhat different every time we played.

To keep track of this new wrinkle we need some new variables at the Class level:

```
private boolean[] questionsUsed; // Track if question was asked
private int randQuestionNum;     // Which question we are asking
```

The `questionsUsed` array will be used to hold a boolean indicating if we have asked the question before or not. This is so we won't randomly pick the same question to ask twice. We should initialize this in the constructor where we set the size of the other arrays:

```
// Allocate arrays
questions = new String[numQuestions];
answers = new String[numQuestions];
values = new int[numQuestions];
questionsUsed = new boolean[numQuestions];
```

Let's write a method to pick a random question out of the array of questions. The `pickRandomQuestion` method is private because it is used internally in the class from `askNextQuestion` method. `pickRandomQuestion` picks a random unused question by looping until we find one that has not been asked, marks it as being used, and sets the variable `randQuestionNum` to the selected number:

```
private void pickRandomQuestion()
{
      do
            randQuestionNum = (int) (Math.random() * questions.length);
      while (questionsUsed[randQuestionNum] == true);
      questionsUsed[randQuestionNum] = true; // Mark question as used
}
```

Finally we can integrate this into the `askNextQuestion` method:

```
public boolean askNextQuestion()
{
	if (questionNum >= 4) // Hardcoded to end after 4 questions
	{
		return false;
	}
	// Show the current question
	System.out.println();
	System.out.println("Question " + questionNum);
	// Pick random question
	pickRandomQuestion();
	System.out.println(questions[randQuestionNum]);
	Scanner kbd = new Scanner(System.in);
	String guess = kbd.nextLine();

	// Check if the answer is correct or not
	guess = guess.toLowerCase();
	if (guess.equals(answers[randQuestionNum].toLowerCase()))
	{
		System.out.println("That is correct!");
		score += values[randQuestionNum];
	}
	else
	{
		System.out.println("Wrong.  The correct answer is "
			+ answers[randQuestionNum]);
	}

	// Go to next question
	questionNum++;
	return true;
}
```

In this code we have kept the use of the `questionNum` variable. This variable keeps track how many questions we've asked, while `randQuestionNum` keeps track of the randomly selected question. With more questions, this might actually make for an interesting game!

**Object-Based Trivia Game**

As an additional example, let's use objects to create the trivia game. An obvious place to introduce an object is to encapsulate a piece of trivia. Instead of the following four arrays:

```
private String[] questions;
private String[] answers;
private int[] values;
private boolean[] questionsUsed;
```

We will instead make a single array of a TriviaQuestion object that contains a question ,answer, value, and whether or not the question has been asked before.

First here is the TriviaQuestion class:

```
public class TriviaQuestion
{
	private String question;
	private String answer;
	private int value;
	private boolean used;

	public TriviaQuestion()
	{
		question = "";
		answer = "";
		value = 0;
		used = false;
	}

	public TriviaQuestion(String q, String a, int v, boolean u)
	{
		question = q;
		answer = a;
		value = v;
		used = u;
	}

	// Accessors
	public String getQuestion()
	{
		return question;
	}

	public String getAnswer()
	{
		return answer;
	}

	public int getValue()
	{
		return value;
	}

	public boolean isUsed()
	{
		return used;
	}

	// Mutator
	public void setUsed()
	{
		used = true;
	}
}
```

This class just encapsulates a question, answer, value, and a boolean indicating whether or not the question has been used.

Next let's declare our class level variables. We only need to store one array of TriviaQuestion objects:

```
private TriviaQuestion[] questions;
```

We can now delete these instance variables:

```
~~private String[] questions;~~
~~private String[] answers;~~
~~private int[] values;~~
~~private boolean[] questionsUsed;~~
```

Now we can fix the constructor where we load in all the questions. We used to have:

```
// Allocate arrays
questions = new String[numQuestions];
answers = new String[numQuestions];
values = new int[numQuestions];
questionsUsed = new boolean[numQuestions];

for (int i=0; i < numQuestions; i++)
{
      questions[i] = inputStream.nextLine();
      answers[i] = inputStream.nextLine();
      values[i] = inputStream.nextInt();
      inputStream.nextLine(); // skip newline
}
```

This gets changed to create a new TriviaQuestion object each iteration of the loop and adding it to the array:

```
// Allocate array
questions = new TriviaQuestion[numQuestions];

for (int i=0; i < numQuestions; i++)
{
  String question = inputStream.nextLine();
  String answer = inputStream.nextLine();
  int value = inputStream.nextInt();
  inputStream.nextLine(); // skip newline
  questions[i] = new TriviaQuestion(question,answer,value,false);
}
```

**It is important to remember that allocating the array doesn't construct individual TriviaQuestion objects**! This just constructs an array full of NULL's that don't contain objects we can use. We must still create a new TriviaQuestion object for each array entry.

Getting the rest of our methods to work with the new object requires accessing the questions[index] array first, followed by a dot and the method that corresponds to the property we want to access:

```
private void pickRandomQuestion()
{
	do
		randQuestionNum = (int) (Math.random() * questions.length);
	while (questions[randQuestionNum].isUsed() == true);
	questions[randQuestionNum].setUsed(); // Mark question as used
}

public boolean askNextQuestion()
{
	if (questionNum >= 4)
	{
		return false;
	}
	// Show the current question
	System.out.println();
	System.out.println("Question " + questionNum);

	// Pick random question
	pickRandomQuestion();
	System.out.println(questions[randQuestionNum].getQuestion());
	Scanner kbd = new Scanner(System.in);
	String guess = kbd.nextLine();

	// Check if the answer is correct or not
	guess = guess.toLowerCase();
	if (guess.equals(
		questions[randQuestionNum].getAnswer().toLowerCase()))
	{
		System.out.println("That is correct!");
		score += questions[randQuestionNum].getValue();
	}
	else
	{
		System.out.println("Wrong.  The correct answer is "
			+ questions[randQuestionNum].getAnswer());
	}

	// Go to next question
	questionNum++;
	return true;
}
```

The program behaves the same as before, but it now better encapsulates trivia information into a single object instead of having to tie together multiple arrays.

**Exercise - Set**

Here is the skeleton for a class that implements a set using an array. As an in-class exercise let's complete it:

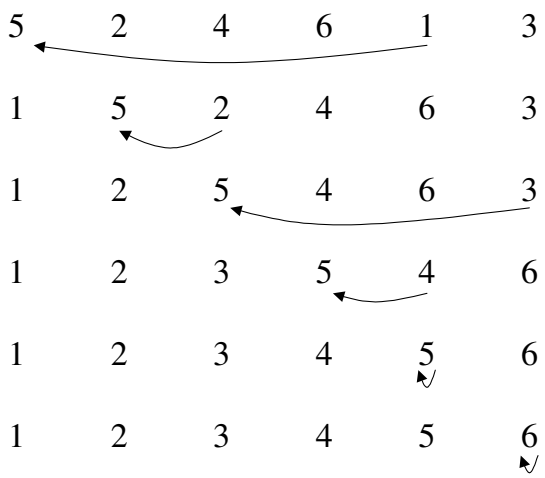
```
public class IntSet
{
	public static final int MAXITEMS = 100;
	private int[] data;
	private int numItems; // How many numbers are in the set

	public IntSet()
	{
		data = new int[MAXITEMS]; // Make array of integers
		numItems = 0;
	}

	// Add a new number to the set.
	// Duplicates should not be allowed.
	public void add(int numToAdd)
	{
		// Fill in code
	}

	// Returns true if num is in the set, false otherwise
	public boolean contains(int num)
	{
		// Fill in code
	}

	// Output everything in the set
	public void output()
	{
		// Fill in code
	}
	// Sample main program
	public static void main(String[] args)
	{
		IntSet myset = new IntSet();

		myset.add(4);
		myset.add(3);
		myset.add(3);
		myset.add(10);
		myset.output();  // Should output 4,3,10

		System.out.println(myset.contains(4));  // true
		System.out.println(myset.contains(33)); // false
	}
}
```

**Sorting – Selection Sort**

If we start with an array of unsorted data, how can we sort it? There are hundreds of techniques that have been devised to sort an array of data. Let's look at selection sort.

The idea behind selection sort is somewhat how you might sort a hand of cards. First, scan through all of the cards until the lowest card is found. Move it all the way to the left. Then scan through all of the remaining cards until the lowest card is found. Move it immediately to the right of the leftmost card. This means the first two cards are now sorted. Repeat the process for all cards:

Example:




Here is pseudocode for selection sort:

```
SelectionSort(array A)                    // A ranges from 0 .. LastIndex
        For j = 0 to LastIndex-1
                // Find min between j, LastIndex
                minIndex=FindIndexOfMin(j, LastIndex)
                Swap (minIndex,j)
```

Now we can flesh out some Java code:

```
public static void SelectionSort(int[] a)
{
      int i,j,minIndex,minValue,temp;
      int lastindex = a.length -1;

      for (j=0; j<lastindex; j++)
      {
            // Find minimum from j to lastindex
            for (i=j, minIndex=j, minValue=a[i];  i<=lastindex; i++)
            {
                  if (a[i]<minValue)
                  {
                        minIndex = i;
                        minValue = a[i];
                  }
            }
            // Swap minimum with j
            temp = a[j];
            a[j] = a[minIndex];
            a[minIndex] = temp;
      }
      return;
}
```

Here is a sample main:

```
public static void main(String[] args)
{
       int[] a = {5, 4, 3, 15, 12};
       int i;

       SelectionSort(a);
       for (i=0; i<=4; i++) System.out.println(a[i]);
}
```

The output is:

3
4
5
12
15

A couple of issues to note regarding this sorting routine: first, there are two nested loops, each running approximately N times, where N is the length of the array. As a result, this algorithm runs in time proportional to $N^2$. Consequently, if we had to sort an array of 1000 values, we would be processing 1,000,000 numbers. This is inefficient for large arrays. There are other techniques that can sort in $N*\log_2 N$ time, which is considerably more efficient. One bonus for this sorting technique though is the amount of space that is uses. Aside from a few temporary variables, we don't need to allocate any additional memory – this is called an **in-place** sorting algorithm.