

CMSC 131: Chapter 19 (Supplement)

Random Thoughts and Strings

Random Number Generation (Recap)

Recall: Random numbers generated in Java (and all other programming languages) are **not** truly random. They are "pseudo-random".

Seed Value: A sequence of pseudo-random numbers is uniquely determined by an initial **seed value**—a long integer.

Math.random(): Returns a pseudo-random **double** r in the range
 $0.0 \leq r < 1.0$

Invokes Java's more general **Random** number class, which resides in the **java.util** package.

Package: java.util (import java.util.*;)

Constructors: Each call to "**new Random**" creates a new pseudo-random sequence.

- **Random(long seed):** Creates a new sequence using the given seed. Given the same seed, the same sequence is generated every time.
- **Random():** (Default constructor) Creates a sequence using the **time of day** as the seed. The sequence is different every time you run it.

Getting a new Random Value:

- **nextBoolean():** Returns a random **boolean**
- **nextInt():** Returns a random **int** (over the entire range of int's)
- **nextInt(int n):** Returns a random **int** r over the range $0 \leq r \leq n-1$.
- **nextDouble():** Returns a random **double** r , where $0.0 \leq r < 1.0$.
- Also: **nextFloat()**, **nextLong()**, **setSeed(long seed)**.

Why seeds?

Example: Three random sequences

```
import java.util.*;

Random g1 = new Random( 1234567 );
System.out.print( " g1:" );
for ( int i = 0; i < 10; i++ )
    System.out.print( " " + g1.nextInt( 1000 ) );
```

Output: g1: 946 436 151 498 348 644 465 913 103 944

```
Random g2 = new Random( );
System.out.print( "\n g2:" );
for ( int i = 0; i < 10; i++ )
    System.out.print( " " + g2.nextInt( 1000 ) );
```

Output: g2: 697 475 636 594 451 979 380 336 810 862

```
Random g3 = new Random( 1234567 );
System.out.print( "\n g3:" );
for ( int i = 0; i < 10; i++ )
    System.out.print( " " + g3.nextInt( 1000 ) );
```

Output: g3: 946 436 151 498 348 644 465 913 103 944

Example: Dealing Cards

Problem: Generate a **random integer array**, `deck[52]`, containing the values 0 through 51, with each value appearing **exactly once**.

Sample Result: (suppose the deck size is 20 rather than 52)

12 17 3 8 16 9 7 2 6 10 11 1 5 0 14 15 13 18 19 4

We will present two solutions, both correct, but one more efficient than the other. We will also illustrate some common methods in array manipulation:

- Nested loops
- Array element swapping

First attempt: For each of the 52 positions, we generate a **random integer** in the range [0...51]. If that number **already appears** in the array, we **try again** until we succeed.

Dealing Cards: First Attempt

Pseudo-code first attempt:

```
for ( i running from 0 to 51 )
    · r = random integer over [0...51]
    · if ( r already appears among deck[0...i-1] ) go back here
    · deck[i] = r
```

Pseudo-code second attempt:

```
for ( i running from 0 to 51 )
    do {
        r = random integer over [0...51]
        foundIt = false
        for ( j running from 0 to i-1 )
            if ( deck[j] == r ) foundIt = true;
        } while ( foundIt is false )
    deck[i] = r
```

Dealing Cards: First Attempt

```
/* Returns a random permutation of [0..nCards-1] */
public static int[] deal1( int nCards ) {
    Random generator = new Random( );           // create random generator
    int[] deck = new int[nCards];               // create deck array
    for ( int i = 0; i < nCards; i++ ) {
        boolean foundIt;
        int r;
        do {
            r = generator.nextInt( nCards );
            foundIt = false;
            for ( int j = 0; j < i; j++ )
                if ( deck[j] == r ) foundIt = true;
            } while ( foundIt );
        deck[i] = r;
    }
    return deck;
}
```

Dealing Cards: Second Attempt

What is wrong with the previous algorithm?

As the array becomes full it becomes **harder** to find **unused** random values.

Imagine filling the **last entry** of the call `deal1( 10,000 )`. Since 9,999 entries have already been used, you only have a 1/10,000 chance of hitting the only unused random number.

Q: Can we do this with **exactly one** invocation of the random number generator for **each array entry**?

A: Yes. Our approach is to repeatedly select a **random** element from the **remaining values**, and store it in `deck[i]`.

Dealing Cards: Second Attempt

Array layout: After the *i*-th iteration:

- The current **random elements** are stored in `deck[0...i-1]`.
- The **unchosen elements** are stored in the back part of the array `deck[i...nCards-1]`.
- We generate a **random index** in `[i...nCards-1]` and **swap** this with `deck[i]`.

Example:

```
i=0 r=6 : 0 1 2 3 4 5 6 7 8 9
i=1 r=5 : 6 1 2 3 4 5 0 7 8 9
i=2 r=3 : 6 5 2 3 4 1 0 7 8 9
i=3 r=6 : 6 5 3 2 4 1 0 7 8 9
i=4 r=8 : 6 5 3 0 4 1 2 7 8 9
i=5 r=6 : 6 5 3 0 8 1 2 7 4 9
i=6 r=7 : 6 5 3 0 8 2 1 7 4 9
i=7 r=8 : 6 5 3 0 8 2 7 1 4 9
i=8 r=8 : 6 5 3 0 8 2 7 4 1 9
i=9 r=9 : 6 5 3 0 8 2 7 4 1 9
Final : 6 5 3 0 8 2 7 4 1 9
```

Dealing Cards: Second Attempt

How do we **swap** the values of two variables x and y?

First try:

```
x = y;  
y = x;
```

Correct Swap:

```
temp = x;  
x = y;  
y = temp;
```

Pseudo-code for Card Dealing:

```
for ( i running from 0 to nCards )  
    deck[i] = i  
for ( i running from 0 to nCards )  
    r = random integer over [i..nCards-1]  
    swap deck[i] with deck[r]
```

Dealing Cards: Second Attempt

```
/* Returns a random permutation of [0..nCards-1] */  
public static int[] deal2( int nCards ) {  
    int[] deck = new int[nCards];  
    for ( int i = 0; i < nCards; i++ ) deck[i] = i;  
  
    Random generator = new Random( );  
    for ( int i = 0; i < nCards; i++ ) {  
        int r = generator.nextInt( nCards - i ) + i;  
        int temp = deck[i];  
        deck[i] = deck[r];  
        deck[r] = temp;  
    }  
    return deck;  
}
```

More Efficient?

To see which method was more efficient, let's run the program for nCards = 100, 200, ..., 1,000 and count the number of calls to the random number generator for each method.

<u>nCards</u>	<u>deal1</u>	<u>deal2</u>
100	514	100
200	941	200
300	2296	300
400	2194	400
500	3629	500
600	4431	600
700	6031	700
800	5432	800
900	5562	900
1000	8125	1000

More about Strings: valueOf

We have discussed strings before. Here are a few other handy things to know about them.

Converting things into Strings: The static method `valueOf()` can be used to convert both primitive types and objects into Strings.

Examples:

```
String.valueOf( boolean x )
String.valueOf( char x )
String.valueOf( char[] x )
String.valueOf( char[] x, int start, int count )
String.valueOf( double x )
String.valueOf( float x )
String.valueOf( int x )
String.valueOf( long x )
String.valueOf( Object x )
```

More about Strings: Text Matching

Text Matching: Given a String, we want to know whether a given substring occurs within it. Here are some useful methods:

Returns the **index of the first occurrence** of the specified substring.

```
int indexOf( String str );
```

Example:

```
String s = "von Weinerschnitzel";
int x = s.indexOf( "Weiner" );    // x = 4
```

This returns the index of the first occurrence. Other variants:

```
int indexOf( String str, int from ) // start at from index
int lastIndexOf( String str )      // index of the last occurrence
```

Examples:

```
String t = "yabadabadoo";
int y = t.lastIndexOf( "bad" );    // y = 6
int z = t.indexOf( "ab", 2 );      // z = 5
```

More about Strings: Splitting

Splitting: It is often useful to break a string up into **words**.

Returns a String array by breaking a String at the given character.

```
String[ ] split( String pattern );
```

Example:

```
String u = "Dude, where's my car?";  
String[ ] a = u.split( " " );
```

Example: Can split around any substring.

```
String t = "yabadabadoo";  
String[ ] b = t.split( "ab" );
```

More about Strings: Regular Expressions

Regular Expressions: The splitting expression need not be just a simple substring. It can even be a general string **pattern**, called a **regular expression**.

Some regular expression elements:

x	A single character matches itself (the character 'x')
.	A period matches any character
[abc]	Matches any of the characters in the square brackets ('a' or 'b' or 'c')
[a-z]	Matches any of the characters in the range ('a' - 'z')
...	There are many more . (See the Java API documentation)

More about Strings : Regular Expressions

Examples of Regular Expressions:

[a-zA-Z]	Matches any alphabetic letter, lower or upper case.
c.t	Matches 'c', anything, 't', for example, "cat", "cot", "cut" "c4t", "c;t".
[-+][0-9]	Matches any string that starts with either '-' or '+' and is followed by a single digit.

```
String w = "it is+1with times+8-7okay?";  
String[ ] c = w.split( "[-+][0-9]" );
```

We will not discuss regular expression extensively, but know that they can be used for matching complex string patterns.