Programming – Lecture 9

Strings and Characters (Chapter 8)

- Enumeration
- ASCII / Unicode, special characters
- **`Character`** class
- Character arithmetic
- Strings vs. characters
- **`String`** methods
- Splitting a string into tokens
- Aside: regular expressions
- Top-Down design

Five-Minute Review

1. Which three areas do we distinguish in memory? What is stored where?
2. What is stored in an *object variable*?
3. What is *garbage collection*?
4. What is *recursion*?
5. What is a *linked list*?

Enumerated Types in Java

Strategy 1: Named constants

```
public static final int NORTH = 0;
public static final int EAST = 1;
public static final int SOUTH = 2;
public static final int WEST = 3;
int dir = NORTH;
if (dir == EAST) ...
switch (dir) { case SOUTH: ...
```

Strategy 2: ***enum*** *Types*

```
public enum Direction {
  NORTH, EAST, SOUTH, WEST }
Direction dir = Direction.NORTH;
if (dir == Direction.EAST) ...
switch (dir) { case SOUTH: ...
```

Enum Types

Are special kind of class, can also contain methods etc.:

```
public enum Planet {
    MERCURY (3.303e+23, 2.4397e6),
    VENUS   (4.869e+24, 6.0518e6),
    EARTH   (5.976e+24, 6.37814e6),
    MARS    (6.421e+23, 3.3972e6),
    JUPITER (1.9e+27,   7.1492e7),
    SATURN  (5.688e+26, 6.0268e7),
    URANUS  (8.686e+25, 2.5559e7),
    NEPTUNE (1.024e+26, 2.4746e7);

    private final double mass;   // in kilograms
    private final double radius; // in meters
    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
    }
...
```

Enum Types

`values()` returns array of enum values, in declared order

```
for (Planet p : Planet.values()) {
    println("Planet " + p +
            "has mass" + p.getMass());
}
```

ASCII Subset of Unicode

	0	1	2	3	4	5	6	7
00*x*	\000	\001	\002	\003	\004	\005	\006	\007
01*x*	\b	\t	\n	\013	\f	\r	\016	\017
02*x*	\020	\021	\022	\023	\024	\025	\026	\027
03*x*	\030	\031	\032	\033	\034	\035	\036	\037
04*x*	*space*	!	"	#	$	%	&	'
05*x*	(	)	*	+	,	-	.	/
06*x*	0	1	2	3	4	5	6	7
07*x*	8	9	:	;	<	=	>	?
10*x*	@	A	B	C	D	E	F	G
11*x*	H	I	J	K	L	M	N	O
12*x*	P	Q	R	S	T	U	V	W
13*x*	X	Y	Z	[	\	]	^	_
14*x*	`	a	b	c	d	e	f	g
15*x*	h	i	j	k	l	m	n	o
16*x*	p	q	r	s	t	u	v	w
17*x*	x	y	z	{	l	}	~	\177

Special Characters

Characters that are not *printing characters*

Escape sequence: backslash + character/digits

`\b`	Backspace
`\f`	Form feed (starts a new page)
`\n`	Newline (moves to the next line)
`\r`	Return (moves to beginning of line without advancing)
`\t`	Tab (moves horizontally to the next tab stop)
`\\`	The backslash character itself
`\'`	The character ' (required only in character constants)
`\"`	The character " (required only in string constants)
`\`*ddd*	The character whose Unicode value is octal number *ddd*

Methods in `Character` Class

`static boolean isDigit(char ch)` Determines if the specified character is a digit.
`static boolean isLetter(char ch)` Determines if the specified character is a letter.
`static boolean isLetterOrDigit(char ch)` Determines if the specified character is a letter or a digit.
`static boolean isLowerCase(char ch)` Determines if the specified character is a lowercase letter.
`static boolean isUpperCase(char ch)` Determines if the specified character is an uppercase letter.
`static boolean isWhitespace(char ch)` Determines if the specified character is **whitespace** (spaces and tabs).
`static char toLowerCase(char ch)` Converts **`ch`** to its lowercase equivalent, if any. If not, **`ch`** is returned unchanged.
`static char toUpperCase(char ch)` Converts **`ch`** to its uppercase equivalent, if any. If not, **`ch`** is returned unchanged.

Character Arithmetic

```
char letterA = 'a';
char letterB = letterA++;
```

`letterB == 'B'`	*False*
`letterB == 'b'`	*False*
`letterB == 'a'`	*True*
`letterA == 'b'`	*True*
`letterA == 'B'`	*False*

```
public char randomLetter() {
   return (char) rgen.nextInt('A', 'Z');
}

public boolean isDigit(char ch) {
   return (ch >= '0' && ch <= '9');
}
```

Exercise: Character Arithmetic

```
// If 0 <= n <= 15, return hex digit
// Otherwise, return '?'
public char toHexDigit(int n) {
    if (n >= 0 && n <= 9) {
       return (char) ('0' + n);
    } else if (n >= 10 && n <= 15) {
       return (char) ('A' + n - 10);
    } else {
       return '?';
    }
}
```

Please visit http://pingo.upb.de/643250

https://xkcd.com/329

Strings *vs.* Characters

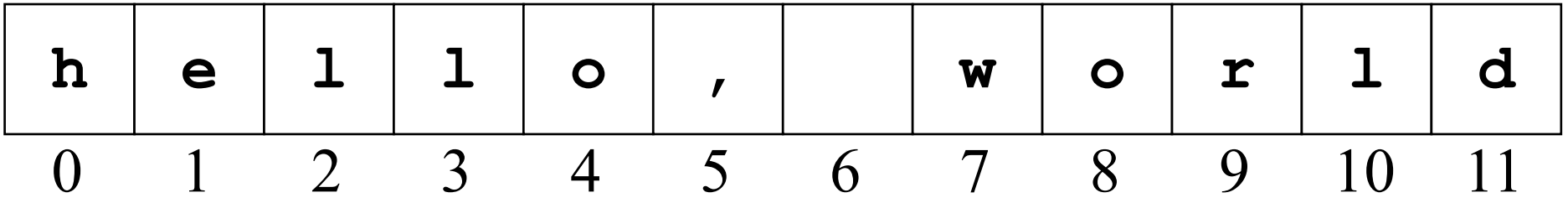
```
ch = Character.toUpperCase(ch);

str = str.toUpperCase();
```

Q: Why not simply:
```
str.toUpperCase();
```

A: Because Strings are immutable!

Selecting Characters from a String

```
String str = "hello, world";
```




```
int twelve = str.length();

char h = str.charAt(0);
```

Concatenation

```
println("hi".concat(" there"));
println("hi" + " there");
println(0 + 1);
println(0 + "1");
println(false + true);
println("" + false + true);
println(false + true + "");
println(false + "" + true);
```

hi there
hi there
1
01
Error!
falsetrue
Error!
falsetrue

Extracting Substrings

string`.substring(`*index-first*`,` *index-after-last*`);`

```
String prog = "infprogoo".substring(3, 7);
```

Checking for Equality

```
String s1 = new String("hello");
String s2 = new String("hello");
```

```
s1 == s2
```
False

```
s1.equals(s2)
```
True

```
String s3 = "hello";
String s4 = "hello";
String s5 = "hel" + "lo";
```

```
(s3 == s4) && (s4 == s5)
```
True

```
s1.intern() == s2.intern()
```
True

JVM uses *string literal pool*

Coding advice: use `equals()` to compare strings

Comparing Characters and Strings

```
char c1 = 'a', c2 = 'c';
```

```
c1 < c2
```
True

```
String s1 = "a", s2 = "c";
```

```
s1.compareTo(s2)
```
-2

Searching in a String

```
String str = "informatik";

str.indexOf('i')          0
str.indexOf("n")          1
str.indexOf("form")       2
str.indexOf('x')          -1

str.indexOf('i',1)        8
```

Other Methods in `String` Class

`int lastIndexOf(char ch)` *or* **`lastIndexOf(String str)`** Returns the index of the last match of the argument, or −1 if none exists.
`boolean equalsIgnoreCase(String str)` Returns **`true`** if this string and **`str`** are the same, ignoring differences in case.
`boolean startsWith(String str)` Returns **`true`** if this string starts with **`str`**.
`boolean endsWith(String str)` Returns **`true`** if this string ends with **`str`**.
`String replace(char c1, char c2)` Returns a copy of this string with all instances of **`c1`** replaced by **`c2`**.
`String trim()` Returns a copy of this string with leading and trailing whitespace removed.
`String toLowerCase()` Returns a copy of this string with all uppercase characters changed to lowercase.
`String toUpperCase()` Returns a copy of this string with all lowercase characters changed to uppercase.

Simple String Idioms

Iterating through characters in a string:

```
for (int i = 0; i < str.length(); i++) {
    char ch = str.charAt(i);
    . . . code to process each character in turn . . .
}
```

Growing new string character by character:

```
String result = "";
for (whatever limits are appropriate to application) {
    . . . code to determine next character to be added . . .
    result += ch;
}
```

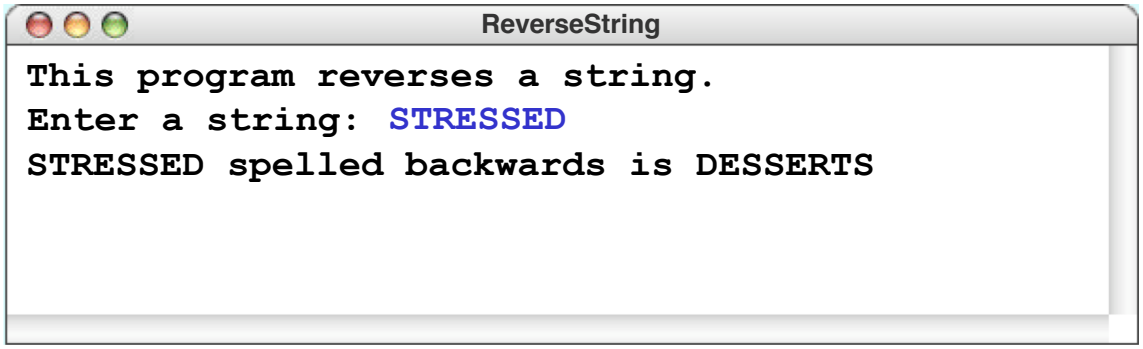
Exercises: String Processing

```
public String toUpperCase(String str) {
   String result = "";
   for (int i = 0; i < str.length(); i++) {
      char ch = str.charAt(i);
      result += Character.toUpperCase(ch);
   }
   return result;
}

public int indexOf(char ch) {
   for (int i = 0; i < length(); i++) {
      if (ch == charAt(i)) return i;
   }
   return -1;
}
```

reverseString

```
public void run() {
   println("This program reverses a string.");
   String str = readLine("Enter a string: ");
   String rev = reverseString(str);
   println(str + " spelled backwards is " + rev);
}
```

rev	str
DESSERTS	STRESSED



skip simulation

Splitting a String into Tokens

Method in **`String`**: **`split()`**

Example:

```
String line = "One short line";
String[] tokens = line.split("\\s");
for (String token : tokens) {
  println(token);
}
```

produces

```
One
short
line
```

Aside: Regular Expressions

`String[] split(String regex)`

`regex` must be *regular expression* (RE)

- RE induces regular language (RL)
- RL is also context-free language (CFL)
- However, CFL is not necessarily an RL; consider $L_2 = \{ a^n b^n : n \in \mathbb{N} \}$

In string literals, must use double backslashes (e.g. **`\\s`**) to encode backslash (**`\s`**)

https://www.quora.com/What-is-the-difference-between-regular-language-and-context-free-language

https://docs.oracle.com/javase/9/docs/api/java/util/regex/Pattern.html#sum

Aside: Regular Expressions

x	The character x
\\	The backslash character
\0n	The character with octal value 0n (0 <= n <= 7)
\t	The tab character ('\u0009')
[abc]	a, b, or c (simple class)
[^abc]	Any character except a, b, or c (negation)
.	Any character (may or may not match line terminators)
\d	A digit: [0-9]
\D	A non-digit: [^0-9]
\s	A whitespace character: [\t\n\x0B\f\r]
\S	A non-whitespace character: [^\s]
^	The beginning of a line
$	The end of a line
X?	X, once or not at all
X*	X, zero or more times
X+	X, one or more times
X{n}	X, exactly n times
XY	X followed by Y
X|Y	Either X or Y
(X)	X, as a capturing group
\n	Whatever the *n*th capturing group matched

Pig Latin

1. If word begins with consonant, move initial consonant string to end and add suffix *ay*:

scram → *amscray*

2. If word begins with vowel, add suffix *way*:

apple → *appleway*

Top-Down Design

```
public void run() {
```
Tell the user what the program does.
Ask the user for a line of text.
Translate the line into Pig Latin and print it on the console.
```
}
```

```
public void run() {
   print("This program translates ");
   println("a line into Pig Latin.");
   String line = readLine("Enter a line: ");
   println(translateLine(line));
}
```

```
private String translateLine(String line) {
  String result = "";
  String[] tokens = line.split("\\s");
  for (String token : tokens) {
    if (isWord(token)) {
      token = translateWord(token);
    }
    result += token + " ";
  }
  return result;
}
```

```
private boolean isWord(String token) {
  for (int i = 0; i < token.length(); i++) {
    char ch = token.charAt(i);
    if (!Character.isLetter(ch))
      return false;
  }
  return true;
}
```

```
private String translateWord(String word) {
   int vp = findFirstVowel(word);
   if (vp == -1) {
      return word;
   } else if (vp == 0) {
      return word + "way";
   } else {
      String head = word.substring(0, vp);
      String tail = word.substring(vp);
      return tail + head + "ay";
   }
}
```

`StringBuilder`

- **Recall:** Strings are immutable
- Whenever we operate on strings, e.g. with **`append()`**, must create new **`String`** objects
- **`StringBuilder`** is a more efficient, but usually less convenient, alternative to **`String`**

Programming advice:

- If performance *really* is an issue and you *heavily* operate on strings, use **`StringBuilder`**
- Otherwise, for better readability, use **`String`**

Example with `String`

```
String str = "";
for (int i = 0; i < n; i++) {
  str += str.length() + " ";
}
println(str);
```

produces for `n` = 10:

```
0 2 4 6 8 10 13 16 19 22
```

Example with `StringBuilder`

```
StringBuilder str = new StringBuilder();
for (int i = 0; i < n; i++) {
  str.append(str.length() + " ");
}
println(str);
```

produces for `n` = 10:

```
0 2 4 6 8 10 13 16 19 22
```

Starts to outperform `String` at n ≈ 100

Both typically well below 1 sec at n = 10000

Aside: Measuring Execution Time

- Outcome of program is (usually) deterministic
- However, run time is not
- Issues influencing timing:
  - Memory hierarchy: instruction cache, data cache
  - Scheduling, process interference
  - Just-in-time compilation
  - I/O delays
  - ...
- Therefore, do *multiple* measurement runs

```
private static final int NUM_TRIALS = 10;
long minTime, maxTime, startTime, stopTime, elapsedTime;

minTime = maxTime = 0;
for (int trial = 0; trial < NUM_TRIALS; trial++) {
  startTime = System.nanoTime();
  String str = "";
  for (int i = 0; i < n; i++) {
    str += str.length() + " ";
  }
  println(str);
  stopTime = System.nanoTime();
  elapsedTime = stopTime - startTime;
  if (minTime == 0 || elapsedTime < minTime) {
    minTime = elapsedTime;
  }
  if (elapsedTime > maxTime) {
    maxTime = elapsedTime;
  }
}
println("One trial took " + minTime / 1e6 + " to "
   + maxTime / 1e6 + " msec.");
```

Summary I

- Principle of *enumeration*: map non-numeric properties/data (e.g. characters) to numbers
- Java provides **`enum`** types
- **`char`**/**`Character`** maps characters to *Unicode*
- Distinguish *printing characters* and *special characters*
- Express special characters with *escape sequences*

Summary II

- Conceptually, strings are ordered collections of characters
- Strings are *immutable*
- Strings should be compared with **`equals`** or **`compareTo`**, not with **`==`**
- Only if (!) performance is an issue and you very heavily operate on strings, use **`StringBuilder`** instead of **`String`**
- Care should be taken when measuring execution times

https://www.youtube.com/watch?v=Hsx5R94YFAA