Data Structures

Augustin Cosse.

Spring 2021

February 2, 2021

Strings

- In Java, the **char** type stores a value that represents a single character encoded through single quotes, such as `'G'`
- For sequences of characters, Java equivalently provides the `String` class
- A `string` encodes a sequence of zero or more characters
- Java uses double quotes to designate string literals.
- String are declared and initialized as

```
String s = "CSCI-UA 9102 Data
                Structures and Algorithms"
```

Strings

- Each character c in a string can be referenced using an index corresponding to the number of characters that come before c (that is if c is at position n in s, it can be accessed as $s[n-1]$)
- Two strings can be combined through concatenation as shown below

```
String sentence = "word1" + " " + "word2"
```

- An important aspect of the string class in Java is that instances are immutable (once an instance is created and initialize it cannot be changed)
- If you want to modify a string, you must therefore create a new string with the character replaced

Strings

- Strings can however be re-assigned

```
greeting = greeting + '!'
// becomes "greeting!"
```

- In order to support efficient editing of strings, Java provides a **StringBuilder**

Strings

- The **StringBuilder** class supports the following methods
  - **setCharAt(k, c):** changes the character at index k with character c
  - **insert(k,s)**: Insert a copy of string s starting at index k of the sequence, shifting existing characters further back to make room
  - **append(s)**: Append string s to the end of the sequence
  - **reverse()**: Reverse the current sequence
  - **toString()**: Return a traditional **String** instance based on the current character sequence.

Wrappers

- There are several data structures and algorithms in Java that only work with objects (not primitive types)
- To get around this obstacle, Java provides a wrapper class for each base type

Base type	Class Name	Creation example	Access Example
boolean	Boolean	obj = new Boolean(true);	obj.booleanValue()
char	Character	obj = new Character('Z');	obj.charValue()
byte	Byte	obj = new Byte((byte) 34);	obj.byteValue()
short	Short	obj = new Short((short) 100)	obj.shortValue()
int	Integer	obj = newInteger(1045);	obj.intValue()
long	Long	obj = new Long(10849L);	obj.longValue()
float	Float	obj = new Float(3.934F);	obj.floatValue()
double	Double	obj = new Double(3.934);	obj.doubleValue()

Wrappers

- In some contexts, Java provides implicit conversion (boxing and unboxing) between instances of the Wrapper class and primitive types
- The wrapper types also provide parse parseInt, parseDouble, parseFloat,.. that return the primitive value associated to a string

```
Integer a = new Integer(12); //
int k = a // implicit call to a.intValue()
int m = j+a // automatic unboxing before addition
// example of conversion from string
Integer b = new Integer("123")
int n = Integer.parseInt("2021")
```

Arrays

- A common task in programming is to keep track of a sequence of values
- This can be done using **arrays** which store a sequenced collection of variables
- Each cell in the array has an index which uniquely refers to the value stored in that cell
- The length of the array is sometimes known as its capacity and be accessed as a.length
- The cell with index k is accessed as a[k]

Arrays

- To avoid 'out of bounds' references (which can lead to buffer overflow attacks), Java always check array indices to determine whether or not they are out of boud. If they are, the runtime Java environment signals an error

- Arrays in Java are neither a base time nor an instance of a particular class. To declare a variable to have an array type, Java uses the (square) bracket notation followed by the type of the variable

```
int[] primes;
```

- To initialize the array, we use the round brackets

```
int[] primes = {2,3,5, 7,11,13,17,19,23,29};
```

Arrays

- Arrays can also be initialized with the new operator as shown below. In this case all the elements are automatically assigned to the default value of the indicated type (false if boolean and null if element type)

```
double[] measurements = new double[1000];
```

- The size of the array is fixed once it has been created. To append elements at the back of the array, it is however possible to create a larger array and they copy all the elements from one array to the next

```
Arrays.copyOf(originalArray, newSize)
```

Enum Type

- On top of the previously discussed types, Java also provides an elegant way to represent choices from a finite set by defining what is known as an enumerated type (enum for short)

```
public enum Day {MON, TUE, WED, THU, FRI,
                          SAT, SUN}
```

- Once defined, **Day** becomes an official type and it becomes possible to declare variables or parameters with the type **day**

- The declaration and assignment of a value in the case of an enum type can then be done as follows

```
Day today;
today = Day.TUE
```

Expressions, Literals, operators

- Variables, literals and operators can be combined to form **expressions** which define new values
- Java allows the following types of literals
  - The **null** object reference
  - Boolean **true** or **false**
  - Integer (the default for an integer is to be of type **int** which means 32 bits integer, A long integer should end with an "L" which indicates a 64bits integer)
  - Floating point. The default for floating point values is that they are **double**. To specify that a literal is a **float**, it must end with an "F" or "f". Floating point literals in exponential notation are also allowed such as 3.14E or .19e10
  - In Java, character constants are assumed to be taken from the unicode alphabet. For example 'a' and '?'

Expressions, Literals, operators

- In addition to the Unicode alphabet, Java also defines the following special character constants

'\n'	(newline)	'\t'	(tab)
'\b'	(backspace (terminal))	'\r'	(return)
'\f'	(form feed)	'\\'	(backslash)
'\' '	(single quote)	'\t'	(double quote)

- Finally, the last type of literals are String literals (sequence of characters enclosed in double quotes) such as ”dogs cannot climb trees”

Expressions, Literals, operators

- Java expressions involve combining literals with variables and operators
- Java defines the following arithmetic operators: $+$ (addition), - (subtraction), * (multiplication), $/$ (division), % (modulo)
- The modulo operator is defined as $n \mathrm{mod} m = r$ if $n = mq + r$ for integers q and $0 \leq r < m$ (remainder)
- Parentheses and the unary minus can be used as in regular arithmetic

Expressions, Literals, operators

- When used on strings, the $(+)$ operator performs concatenation so that the code

```
String rug =  "carpet"
String dog = "spot"
String mess = rug + dog
```

will produce the string "carpetspot"

Expressions, Literals, operators

- Like in C/C++ Java provides increment (++) and decrement (–) operators
- If those operators are placed in front of a variable, then 1 is added to this variable and the resulting value is read into the expression
- If it is used after the variable, then the value is first read and then the variable is incremented or decremented

```
int i = 8;
int j = i ++; // j becomes 8 and then i becomes 9
int k =++i; // i becomes 10 and then k becomes 10
int n = 9+ --i;
```

Expressions, Literals, operators

- Java also supports standard logical operators (whose result s a boolean value.)

$<$	less than
$<=$	less than or equal
$==$	equal to
$!=$	nnot equal to
$>=$	greater than or equal to
$>$	greater than

- The logical operators can also be applied to **char** in which case the result of the comparison is determined according to the underlyinng character codes

- Finally, note that a == b is true if **a** and **b** refer to the same object. Most object types also support an **equal** method **a.equals(b)** which returns true if **a** and **b** refer to similar instances of a class (not necessarily the same)

Expressions, Literals, operators

- For boolean values, Java also defines the usual operators ! (not), && (conditional and) as well as || (conditional or)
- For those operators, Java implements a form of short circuiting. I.e. the second operand in the expression will only be evaluated if needed (if the first operand in && is false or if the first operand in || is true, the second will not be evaluated)
- Correspondingly Java provides bitwise operators (working on integers and booleans and returning 0/1)

~	bitwise complement
&	bitwise and
|	bitwise or
^	bitwise exclusive or
<<	shifts bits left, filling in with zeros
>>	shift bits right, filling in with sign bits
>>>	shift bits right, filling in with zeros

Expressions, Literals, operators

- The standard assignment operator in java is the = operator. Note that

```
j = k = 25
```

  is totally valid because assignments are evaluated from right to left.

- Java also provides a couple of compound assignment operator of the form **variable op= expression** which are equivalent to the expression **variable = variable op expression**.

```
a[5] = 10;
j = 5;
a[j++]+=2; // leaves a[6] = 12 and j = 6
/* note that this is different from
a[j++] = a[j++]+2 */
```

Operator precedence

When parentheses are not used, precedence is determined according to the following rules

	Type	Symbol
1	array index	[]
	Method call	()
	dot operator	.
2	postfix ops	exp ++, exp –
	prefix ops	++exp, –exp, +exp, -exp, ∼ exp !exp
	cast	(type) exp
3	multi./div.	* / %
4	add./ subt.	+ -
5	shift	<< >> >>>

Operator precedence

	Type	Symbol
6	comparison	< <= > >= **instance of**
7	equality	== =!
8	bitwise and	&
9	bitwise xor	^
10	bitwise or	|
11	and	&&
12	or	||
13	conditional	booleanExpression ? valueIfTrue : valueIfFalse
13	Assignment	= += -= *= /= %= <<= >>= >>>= &= ^= |=

Type conversion

- Type conversion in Java is done through (implicit or explicit) **Casting**.
- Casting from double to int is known as narrowing while casting from int to double is known as widening

```
double d1 = 3.2;
double d2 = 3.9999;
int i1 = (int) d1;
double d3 = (double) i1;
```

Type conversion

- Explicit casting cannot convert a primitive type into a reference type or vice versa but as we have already seen with the wrappers (conversion between primative to object), there are other methods that can be used (for example to convert between string and the primitive type)

```
String s1 = "2014";
int i1 = Integer.parseInt(s1);
int i2 = -35;
String s2 = Integer.toString(i2);
```

Type conversion

- In some frameworks, Java can perform implicit casting
- This works for example for widening casts. However, it will fail in the narrowing framework

```
int i1 = 42;
double d1 = i1; // works
i1= d1; // fails
```

- Implicit casting works also in the context of some arithmetic operations (addition and subtraction)
- For division however the cast has to be explicit

```
3+5.7 // returns a double
(double) 7/4 // equivalent to (double 7)/4
// returns 1.75
7/4 // returns 1.
```

Type conversion

- There is one situation in which implicit casting is the only option allowed. It is in string concatenation
- Any time a string is concatenated with a base type or an object, the base type or object is automatically transformed into a string
- To perform an explicit conversion to a string, we must use the toString method. An laternative is to perform an implicit cast through a concatenation operation

```
String s = Integer.ToString(22)
String t = "" + 4.5
```

Control Flow

- Control Flow in Java is siilar to other languages, including **if** statements, **switch** statements , **break** and **continue** statements
- The structure of the **if** and **if else** statements are as follows

```
if (booleanExpression)
        trueBody
else
        falseBody
```

```
if (firstbooleanExpression)
        firstBody
else if(secondBooleanExpression)
        secondBody
else
        thirdBody
```

Control Flow

- With the former syntax the bodies are limited to single statements. For block statements, the curcly brackets must be used
- The value tested in the **if** statements must be a boolean in Java
- If the curly brackets are not used, only the first line is considered as being part of the **if** statement. In general it is better to use the brackets.

Control Flow

- Just as the other main programming languages, Java provides multiple value control flow through the **switch** statement

```
switch(d) {
        case MON:
          system.out.println("This is tough");
          break;
        case TUE:
          system.out.println("This is better");
          break;
        case WED:
          system.out.println("Half way there");
          break;
        default:
          system.out.println("Day off!")
}
```

Loops

- Java provides three main types of loops.
    - While loops

```
while (booleanExpression){
        loop Body}
```

    - Do-while loops

```
do {  loopBody }
while(booleanExpression);
```

    - For loops

```
for (init.; booleanCondition; increment){
        loopBody }
```

Loops

- For all of those loops, if you omit the braces, only the first line following the statement will be executed. As a result it is always a good thing to use the braces.
- The initialization, boolean condition and increment in the for loops are of the form

```
for (int j=0; j<n; j++){
        // body of loop
}
```

Loops

- On top of the main loops, since looping over a collection of elements is a common process, Java also provides a shorthand notation for such loops known as for-each loop. The syntax is the following

```
for(elementType name: container){

loopBody

}
```

Explicit Control Flow Statements

- Java provides a number of commands that explicitly change the flow of control of a program:
- The first such statement is **return**. If a method is declared with a return type of **void**, the flow of control returns when it reaches the last line of the body of the method or when it encounters a return statement with no argument
- If the method is defined with a return type, the method must exit by returning an appropriate value

Explicit Control Flow Statements

- The **break** statement was already used to exit from the body of the switch statement. More generally it can be used to exit from any **switch**, **for**, **while** or **do while** statement body.
- When executed, the **break** statement causes the flow to jump to the next line after the loop.

Explicit Control Flow Statements

- A last control statement which can be used inside a loop is the **continue** statement.
- The **continue** statement causes the the execution to skip over the remaining steps of the current iteration of the loop body but then, unlike the break statement, the flow returns to the top of the loop, assuming that the condition remains satisfied

Input and Output in Java

- Java provides a number of classes that enable the programmer to develop graphical user interfaces (including pop up windows and pull down menus), as well as methods for the display and input of text and numbers.
- Simple input output in Java either occurs through a special pop up windows, or through the terminal
- The default output in Java is to the terminal.
- Java provides a built in static object: System.out that performs output to the terminal

Input and Output in Java

- The **System.out** object is an instance of the **java.io.PrintStream** class. This class defines methods for a buffered output stream, meaning that characters are put in a temporary location, called **buffer** which is then emptied when the console window is ready to print characters

- The **java.io.Printscreen** class provides the following methods to perform simple output

print(String s)	Print the string s
print(Object o)	Print the object o using the 'toString' method
print(baseType b)	Print the base type value b
println(String s)	Print s followed by the new line char.
println(Object o)	Print(o), followed by new line char.
println(baseType b)	Print(b), followed by newline char.

Input and Output in Java

- Consider the following code fragment

```java
System.out.print("Java values: ");
System.out.print("3.1416 ");
System.out.print(' , ');
System.out.print(15);
System.out.println(" (double, char, int). ");
```

When executed,this fragment will print Java values: 3.1416, 15 (double,char,int).

Input and Output in Java

- Just as there is a special object for performing output to the Java console, there is also a special object called System.in for performing input from the Java console
- Technically, the input is coming from the ”standard input” device, whichby default is the computer keuboard echoing its characters in the Java console
- A simple way of reasing input with this object is to use it to create a Scanner object using the expression

```
new Scanner(System.in)
```

Input and Output in Java

```java
import java.util.Scanner;

public class InputExample {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.print("Enter your age in years: ");
    double age = input.nextDouble();
    System.out.print("Enter your maximum heart rate: ");
    double rate = input.nextDouble();
    double fb = (rate - age )*0.65;
    System.out.println("Your ideal fat-burning
                heart rate is " + fb); }}
```

Input and Output in Java

- The Scanner class reads the input stream and divides it into tokens which are strings of characters separated by delimiters
- A delimiter is a special kind of string and the default delimiter is the white space
- Tokens are separated by strings of space, tabs and new line by default
- Tokens can either be read immediately as strings, or the Scanner class can convert a token to a base type, if the token has the right form

Input and Output in Java

- The scanner class includes the following methods for dealing with tokens

hasNext()	returns true if there is another token in the input stream
next()	returns next token string in input stream
hasNexType()	returns true if there is another token in the stream and it can be interpreted as a base Type (boolean, ... double)
nextType	returns the next token in the input stream returned as the base Type corresponding to Type ex. nextInt()

Input and Output in Java

- The Scanner class can also process input line by line, ignoring delimiters and even look for patterns within lines while doinng so. The methods for processing lines in this way include the following

hasNextLine()	returns true if the input stream has another line
nextLine()	Advances the input past the current line ending and returns the input that was skipped
findInLine(String s)	Attempt to find a String matching the pattern s

The Math class

- For some of the exercises of today, you might need the math class
- The Math class provides a large number of basic mathematical functions that are often helpful in making calculations.
- All of the methods from the math class are static methods, which means they can be invoked through the name of the class without having to instantiate an object of the class first
- As an example, the line below computes the absolute value of the number stored in total adds it to the value of count raised to the fourth power and stores the result in the variable value.

```
value = Math.abs(total) + Math.pow(count, 4);
```

The Math class

static int abs(int num)	absolute value
static double acos(double num)	
static double asin(double num)	
static double atan(double num)	
static double cos(double angle)	
static double sin(double angle)	
static double tan(double angle)	
static double ceil(double num)	
static double exp(double power)	e^{power}
static double floor(double num)	
static double pow(double num, double power)	
static double random()	rand. num in [0,1[
static double sqrt(double num)	