The following notes are meant to be a quick cheat sheet for Java. It is not meant to be a means on its own to learn Java or this course. For that you should look at your textbook and the sample code from class. Furthermore, this is not an exhaustive listing of all of Java’s capabilities, as we still have many a fundamental concept to cover in this course, and in other courses down the road. These notes will be updated as we learn more syntax. Enjoy!

1 Comments & Whitespace

- blanks and tabs ignored by Java compiler
- 2 types of comments:
  a) // This is a comment
  b) /* This is also a comment */
     /* /* I can be nested */ */
     /* And I can span
     multiple lines */

2 Tokens

2.1 Keywords

These are words reserved in Java for special use, so you cannot use them as your own identifiers (variable, method, & class names). They are listed on p17 of your text, but a sampling is:

boolean, char, class, const, double, else, final, float, for, if, import, int, long, new, public, return, static, throws, void, while

2.2 Identifiers

- is a name for a variable/class/method, i.e. myVar, booYA, I Hate Artichokes And Olives, ...
- must begin with a letter, underscore (_), or currency symbol ($)
- may contain any number of digits, letters, underscores, or currency symbols after the first character, i.e. _83yy$z
- Java is case-sensitive, i.e. I Hate Artichokes is different from i hate artichokes
- must not use the keywords as they already have special meaning
- convention: class names should begin with an uppercase letter, as in MyClass. Method names should begin with a lowercase letter, as in main.
2.3 Primitive Data Types

- there are eight primitive types: boolean, char, byte, short, int, long, float, double
- we typically use boolean, int, double, char
  - boolean: true, false (no 0s and 1s allowed)
  - char: 'a', 'b', 'c', ... (any 16-bit unicode character)
    * can convert chars to ints, and vice-versa
    ex. char whatAmI = 97; // This is the character's 'a'! Ooooo
    * use actual 'character', i.e. char = 'X';
    * or, use escape characters: \t (tab), \n (newline), " ("), \' ('), \ ( \)
  - int: 32 bits. Ranges from -2147483648 to 2147483647
  - double 64 bits.
    * use decimal point i.e., .10, 1.0, 1.0

2.4 Operators

- increment / decrement
  ex. x = x+1 could be written as x++ or ++x
  ex. artichoke = 1; baloney = artichoke++;
    baloney gets the current value of artichoke (i.e. 1), then artichoke increments to 2
- modulus (%) (the remainder operator)
  ex. 24 % 5 gives 4
- before of = (assign) vs == (equals)!
  ex. int a = 5;
    while( a = 5 ) {a = 4; } // This will loop forever
- object creation (new)
- object access (.)
  ex. Foobar fb = new Foobar(); // Creates new object
    fb.fixMe(); // Accesses a method from the object
- array element access( [] )
  ex. int[] myArray = {5, 4, 3, 2, 1};
    int temp = myArray[1]; // temp will have the value 4
2.5 Boolean Expressions

• true, false are the simplest boolean expressions
• can use relational operators to construct boolean expressions
  ex. i < 5, s.charAt(0) == s.charAt(6), s.equals( "trogdor" )
• can use boolean operators to combine boolean expressions
  ex. (n==1) || (n==2)

2.6 Punctuation

• use ( ) for expressions and methods
• use ; for ending statements
• use {} for blocks of statements

3 Statements

• block: any collection of statements inside {} 
• expression: assignments, increment / decrement, method calls, object creation 
• declaration: must tell Java about a variable before using it 
  ex. int a; // Declaring the variable a to be an int 
    a = 5;  // Using the variable a 
• assignment: to store a value in a variable 
• method call 
• object creation 
• selection: if-else, if-else if 
  – relations: <,>,<=,>=,==,!= 
  – logic: && (and), || (or), !(not) 
  – values: true, false 
    ex. if( b ) // Or, for example if( x < 5 ) 
      System.out.println( "b is true!" ); 
    ex. if( b ) 
      System.out.println( "b is still true!" ); 
      else 
        System.out.println( "b is false. Figures." ); 
• repetition: while, for
4 Methods

- syntax: `public static <return-type> <method-name>( arguments ) { <statements> }`

  ex. `public int squareMe( int i ) { return (i * i); }`

- return type can be `void` (returns no value), or any type.

- arguments are of the form `<type> <varname>, <type> <varname>, ..., or no arguments at all`.

- before the body of the method, may possibly throw exceptions

  ex. `public static readFile( String fname ) throws FileNotFoundException {...}

- if the method parameters are primitive types then the value of the actual parameter is copied into the formal parameter, that is, the method cannot change the value of an actual parameter

- if the method parameters are object/complex/reference types, then the pointer to the actual parameter is copied into the formal parameter, and you can change the value of an actual parameter

- overloading: writing multiple methods with the same name but different signature (i.e. different order of arguments, types of arguments, number of arguments, or any combination of the three)

5 Creating Objects & References

- to create an object (which is an instance of a class), use a constructor call:

  ex. `Movie m = new Movie( "Revenge of the Nerds" );`

- `null` is the value that represents no object

- Java does not allow you to "store" the actual object in a variable; instead, you need to use a variable of the object’s type that stores the address of this object. This kind of variable is called a `reference variable`.

  ex. `Aardvark a = new Aardvark();`

  - `a` is a variable of type `Aardvark`
– a stores the address of a newly created Aardvark
– if you print a, you’ll see the address value (Aardvark@....), not anything actually *useful*, God forbid.

- changing the contents of a reference variable means storing the address of a different object:
  
  ```java
  ex. Alien bossAlien;
  Alien a1 = new Alien();
  Alien a2 = new Alien();
  bossAlien = a1; // boss now contains the address of a1
  ```

- passing an object to a method really means passing a reference to that object

- returning an object from a method really means returning a reference to that object

6 Arrays

- arrays are objects — must use new!

- all elements of array must have same type

- indexed from 0. Indices must be integers, or be expressions that evaluate to integers

- syntax to declare: `<type>[] <varname>;` or `<type> <varname>[][];

  ```java
  ex. Aliens[] starsInMenInBlackII; // An array of Aliens
  ```

- syntax to assign: `<varname> = new <type>[size];`

  ```java
  ex. starsInMenInBlackII = new Aliens[ 3 ];
  ```

- can find length easily using `<varname>.length`

  ```java
  ex. starsInMenInBlackII.length is 3.
  ```

- arrays are 0-indexed:

  ```java
  ex. someArray[ 0 ] is the first element of someArray.
  ```

  ```java
  ex. someArray[ someArray.length -1 ] is the last element of someArray.
  ```

- can make arrays of objects (i.e. arrays of references), and multidimensional arrays
Strings and Characters

- strings are reference types (use \texttt{String} class)
- string literal: "yodelayheehoo" — is an instance of class \texttt{String}
- empty string: ""
- concatenation is easy:
  - ex. "you make me complete" + "ly miserable" → "you make me completely miserable"
- even concatenating non-strings is easy:
  - ex. "mumbo number " + 5 → "mumbo number 5"
- must put \texttt{String} on \texttt{one} line. One!
- multiple \texttt{String} constructors:
  - \texttt{String s0 = "yo!";} \hspace{1cm} // creates a string literal
  - \texttt{String s1 = new String();} \hspace{1cm} // creates an empty string
  - \texttt{String s2 = new String( "whassaaaap" );} \hspace{1cm} // creates string of "whassaaaap"
- Strings are immutable – once created, they cannot change! But you can copy them...
- strings indexed from 0
  - ex. \texttt{String s = new String( "Ra ra Rhasputin" );}
  - \texttt{char c = s.charAt( 1 );} \hspace{1cm} // c will hold the character 'a'
  - \texttt{String t = s.substring( 0, 5 );} \hspace{1cm} // t holds "Ra ra"
- Do not use == to compare strings, use \texttt{s1.equals(s2)}. Yes, sometimes == will work, but not all the time, so just stick with the \texttt{equals}, okay!!

Classes

- blueprint / mold for creating objects
- syntax: \texttt{<modifiers> class <classname> \{ <fields>; <constructors>; <methods>; \}}
- modifiers same as for methods: public or private
- fields and methods are called members
- fields represent properties of a class, methods are for accessing and modifying these properties
- fields get default values of their respective types
- every method can see any other method in the same class in any order
constructors return a reference to the newly created object – they do not have a return type

- syntax: \(<\text{modifiers}\>\ <\text{classname}\>(\ <\text{arguments}\>\ )\ \{\ <\text{body}\>\ }
- every class has at least one constructor, even if you don’t write one. The default is \text{classname}\()\ {}\), a.k.a. the empty constructor.

ex. class Movie {
    private String name;
    public Movie( String s ) { name = s; } \// This is the constructor
    public toString() { return name; }
    public addStudioToName( String studio ) { name = studio + \\
"'s " + name }
}

9 Useful Classes

9.1 The Math Class
- need to add \text{import java.lang.*; at top of file.}
- contains functions like abs, sqrt, pow, ...

9.2 The Random Class
- need to add \text{import java.util.*; at top of file}
- methods functions like \text{nextInt}(n)

9.3 The Scanner Class
- need to add \text{import java.util.*; at top off file}
- can construct for the console or from a File, or even from a String
  ex. Scanner input = new Scanner( new File( "input.txt" ) );
  Scanner console = new Scanner(System.in);
  Scanner console = new Scanner( "it’s just string" );
- method \text{nextLine()}, \text{nextInt}()

9.4 The File Class
- need to add \text{import java.io.*; at top of file}