COMP250

JAVA

Syntax

Operators:

```
NOT !, AND &&, OR ||, +=, -=, *=, /=, x++, --x
```

Order of Operations:

```
Parenthesis, !, Typecasting, Arithmetic(+,-,etc.),
    ↪ Comparison(1.Relational: <,>,<=,>= 2.Equality:
    ↪ ==, !=), Boolean (&&, ||).
```

Primitive Data types:

```
Integers: byte (8), short (16), int (32), long (64)
Reals: float (32), double (64)
Other; boolean(1), char (16)
With n bits, you can represent 2 raised to n values.
Overflow is when the number you are trying to represent
    ↪  needs more bits than its data type has.
Java automatically converts one type to the other if
    ↪ need be and no info is lost.
```

Typecasting

```
Explicit typecasting is necessary only if going to a
    ↪ narrower type. O.W. done automatically.
(wide) db. , fl., lg., int, ch., sh., by. (narrow)
```

Strings:

```
String is a class and its instances are objects.
someString.equals(Obj obj) compares the string to the
    ↪ object, returns a boolean.
s.length(), s.charAt(i), s.toLowerCase().
Convert int to string: String s = "" + 4;
Convert String to int: int x = Integer.parseInt("54")
```

Arrays:

```
Contains a fixed number of values of the same type.
int[] arrayName = {1,2,3,4};
String[] arrayName = new String[7];
Default values when creating array with second method
    ↪ are num:0, Str/reference:null, char:0, bool:
    ↪ false.
a.length gives the length of the array a.
import java.util.Arrays;
Arrays.equals(x,y); Arrays.toString(x); Arrays.sort(x);
```

Random:

```
import java.util.random;
Random robj = new Random(seed); int diceRoll = robj.
    ↪ nextInt(6);
```

While Loop:

```
while (some condition) {// some code}
The block of code is repeated as long as the condition
    ↪ evaluates to true.
```

For Loop:

```
for (initializer; boolean expression; update) {//some
    ↪ code}
Any statement can be left out, semi colons must be
    ↪ there.
Initializer executes once before, boolean exp. and
    ↪ update executed before each loop.
```

For each loop:

```
for(int num: someArray){// do sum}
```

Error and Exceptions:

```
Compile time errors: something is wrong with the Syntax
Runtime errors: array out-of-bounds, null pointer,
    ↪ division by zero, etc. program would compile
    ↪ complain at runtime.
Throwing an exception;
   throw new [nameOfException]("message")
Try-catch block:
   try{\\code that may be problematic}
   catch(SomeException e){\\what to do in this case}
   finally{\\this code will always run no matter what}
To display info about exception e: e.printStackTrace();
```

Scope of a variable:

```
A variable only exists in the block where it is
    ↪ declared.
```

Overloading:

```
Having more than one method with the same name, but
    ↪ different outputs.
```

OOP

Keywords:

```
new: creates an object and allocates memory for it.
null: points to nowhere in the memory.
package: group of classes, you have access to all
    ↪ methods whithin the package. If method is from
    ↪ another package:
   animals.Dog myDog = new animals.Dog(); import
       ↪ animals.Dogs; import animals.*;
this: refers to the object on which the method has been
    ↪  called.
```

Modifiers:

```
CLASSES:
   public: accessible by any other class
   default (package-private): accessible only in the
       ↪ same package
   final: class cannot be inherited, i.e. no
       ↪ subclasses
   abstract: cannot be used to create objects.
ATTRIBUTES and METHODS:
   public: accessible from all other classes
   private: only accessible within the class
   default: only accessible within same package
   protected: accessible in same package and
       ↪ subclasses.
   final: Method cannot be overridden by subclasses,
       ↪ final attribute can only be initialized
       ↪ once. If the object the attribute refers to
       ↪  is mutable, we can change it freely, but
       ↪ it must stay the same obj. A final non-
       ↪ static field must be initialized in every
       ↪ constructor. A final static field must be
       ↪ initialized on the same line of the
       ↪ declaration.
   abstract: Can only be used in an abstract class,
       ↪ and can only be used on methods, does not
       ↪ have a body and must be overriden in
       ↪ subclass.
   static: Attributes and methods belongs to the class
       ↪ , rather than an object. Static methods are
       ↪  used to perform operations that are not
       ↪ dependent upon instance creation
         ClassName.methodName() or ClassName.varName
```

Constructor:

```
Name must be the same as the class name, no return type
    ↪  and non-static. The default constructor takes
    ↪ no argument and has an empty body, you lose
    ↪ access to it as soon as you declare your own
    ↪ constructor.
```

Getters and Setters:

```
Getters: in general, all fields should be declared
    ↪ private and be accessed with a getter:
   public <type> getField(){
      return this.field;}
Setters (Mutators): Allow you to modify the value of a
    ↪ non-static field.
   public void setField(<type> value){
      this.field = value;}
```

toString():

```
This is the method that will be called when we use
    ↪ println().
public String toString(){
   //returns a value of type String.}
```

Immutable:

If all fields in a class are private and there are no mutator methods, then the type is immutable.

Some guidelines:

Constructors that initialize mutable reference types should make a copy first.
Get/set methods that access/mutate a mutable reference type should make a copy first.

Inheritance:

Class can extend another if latter is visible from where former is, it inherits all public and protected methods and fields from superclass.
Syntax: `public class Dog extends Animal {}`
Hiding a field: A field with the same name as one in the superclass is hiding the inherited field.
Hiding a method: A STATIC method with the same name, signature and return type as the one from the superclass.
Overriding a method: A NON-STATIC method with the same name, signature and return type as the one from the superclass.
Constructor: The constructor is not inherited, can invoke a constructor from superclass using the keyword super. If no constructor is invoked, the no-argument constructor is called, if it does not exist, compile time error.
super keyword: Can be used in a similar way as this, refering to the instance of the superclass on which a non-static method was called. Useful if the method we want to access was overriden in the subclass.
super(parameters); can be placed in the subclass to invoke the superclass constructor.

Object Class:

The only class with no superclass, some methods:
hashCode(): Returns a 32 bit integer associated to the object. a.equals(b) is true implies a.hashCode() == b.hashCode(), the converse is false.
toString(): Returns a string representation of the object. It is of the form <classOfObj>@Obj.hashCode().
equals(): returns true IFF a == b is true.

Type Casting reference types:

Implicit upcasting: `Animal myPet = new Dog();` is allowed because Dog is a subclass of Animal.
Explicit downcasting: `Dog myDog = myPet;` will cause compile time error, must do `Dog myDog = (Dog) myPet;` will cause a runtime error if myPet turned out to not be a Dog.
IMPORTANT: Casting does not change the type of the object, only labels it differently.

instanceof operator:

Returns a boolean, if applied on a null value, returns false.
Useful to make sure that explicitely downcasting will not cause an error e.g. when overriding equals.

Abstract:

An abstract method is one that is declared without implementation:
`public abstract double getArea();`
Used when you want a class to have a certain method, but want the implementation to be specified by the subclass (e.g. area of triangle vs circle).
Abstract class: If a class contains an abstract method, it must also be declared abstract, converse is false. It can declare both abstract and concrete methods. A subclass derived from an abstract class must either implement all the abstract methods or be abstract itself. Can contain a constructor, cannot be instantiated.

Wrapper Classes:

Integer, Double, Character are wrapper classes, conversion is done automatically, these classes enable us to call methods on primitive types such as Integer.MAX_VALUE.
Autoboxing: converting to wrapper class
Unboxing: converting to primitive

Lists

Array Lists:

A list which keeps track of the number of elements it contains, creates a new array double its size and copies each element to it each time it reaches its limit. Cannot store primitive types, must use wrapper.
Double length k=log2(N) times for final length of N.
# of copy operations to add N elements to list: N-1.
`ArrayList<Integer> sum = new ArrayList<Integer>();`

Singly Linked lists:

```
public class SLinkedList{
    private SNode head; private SNode tail;
    private int size;
    private class SNode{
        Object element; Object next;}}
```

Nodes have one pointer to the next element and one to the object they contain.

Doubly Linked Lists:

Nodes have a pointer to the previous, next node, and to the object they contain.
Avoid edge cases with dummy nodes:

```
public class DLinkedList {
    private DNode dummyHead; private DNode dummyTail;
    private int size;
public DLinkedList(){
    dummyHead = new DNode(); dummyTail = new DNode();
    dummyHead.next = dummyTail;
    dummyTail.prev = dummyHead; size =0;}}
```

Speed up getNode(i): start from head if i<size/2, and from tail o.w.

Time complexity of arraylist, Slinked, Dlinked

add/remove first: O(N), O(1), O(1)
add Last: O(1), O(1), O(1)
remove Last: O(1), O(N), O(1)

Bubble sort:

Repeatedly iterate through the list and swap adjacent items if they are in the wrong order.
After one iteration, we know the largest item is at the end, so at each iteration we can stop comparing items one step earlier.
Stop the algorithm when no change is made in the list.

```
sorted = false; i = 0;
while (!sorted) { sorted = true;
    for j from 0 to list.length - i -2 {
        if(list[j] > list[j+1]) {
            swap(list[j], list[j+1]);
            sorted = false;}}i++;}
```

Selection sort:

Iterate through the list, find the minimum value, swap it with the first element of the unsorted list, start again, but one index further in the list

```
for(int i = 0; i<array.length-1; i++){ int min = i;
    for(int j = i +1; j<array.length; j++){
        if(array[min] > array[j]) { min = j}}
        int temp = array[i]; array[i] = array[min];
        array[min] = temp;}
```

Inner Loop iterates N(N-1)/2 times.

Insertion Sort:

Select first element x of unsorted part of the list.
Insert x into correct position in sorted part of list.
Change where you divide the array sorted/unsorted part.

```
for(int i=1; i<array.length;i++){int temp=array[i];
        int j = i-1; while(j>=0 && array[j]>temp) {
        array[j + 1]=array[j];j--;}array[j+1]=temp;}
```

Conclusion

All three sorting algorithms have an average sorting time O(pow(n,2))
Insertion and bubble best case is O(n), selection O(pow(n,2)).