== **Wrapper classes and autoboxing** ==

Every primitive type has a wrapper class version. It can be used to represent a primitive value when an Object is needed. Java can automatically "box" a primitive into an instance of its wrapper class.

Integer x = 6; // automatically does Integer i = new Integer(6)

And it can automatically "unbox" a wrapper object to get the primitive value.

int y = x + 4; // automatically does int y = x.intValue() + 4;

METHODS -- start at MOST SPECIFIC class

ATTRIBUTES -- check TYPE of the class you're using, start there

Top t2 = new Middle();

// making a Middle, casting it as a Top because a child class inherits everything from its parent any place where we could have used the parent, we can instead use the child It does not work the other way around!----- Middle m2 = new Top();

Object[] myArray = new Object[3]; ((String)myArray[0]).trim();

-When a child has a method of the same name as its parent - in overriding

-When a child has a variable of the same name as its parent - shadowing

-Default (package private), don’t type default, cant instantiate an abstract class!

MVC

**CONTROLLER**

public class DiceController implements ActionListener

private DiceModel dice;

public DiceController(DiceModel dice) this.dice=dice;

public void actionPerformed(ActionEvent e) this.dice.roll();

**VIEW**

public class DiceView implements Observer {

public DiceView(ActionListener controller) { //create gui

public void update(Observable o, Object arg) { DiceModel dice = (DiceModel)o; tfDie1.setText(""+dice.getDiceValue1()); tfDie2.setText(""+dice.getDiceValue2());

**APP**  //Create the model, view and controller, and hook them up

DiceModel model=new DiceModel(); DiceController controller = new DiceController(model);

DiceView view=new DiceView(controller); model.addObserver(view);

**Design Patterns bruhh**

Singleton - Includes a private constructor, Has a static variable that stores the instance, static since we never instantiate the class, Has a getInstance static method that returns the instance

Observer - When we need a class to update depending on another class. Needs a class that extends Observable and one that implements Observer, which has an update method update(Observable o, Object arg)

Iterator - When you need to go through a collection of elements, Needs a class that implements Iterable<Object> and has an iterator() method which returns an Iterator, and a class that implements Iterator<Object> that overrides hasNext() and next().

Composite – Used when you need to manipulate a hierarchial collection of primitive(simple) and composite(complex) objects. Used when we have composites that contain components each of which could be a composite. When you have an object A that supports some operations and an object B that is an aggregation of A and you want to treat them both with the same interface, ignoring their differences. Create an abstract class (component) that includes shared operations and both the individual (leaf) and the aggregate (composite) object will extend it. Client creates a command and sets its receiver.

Command - Different goals, different means. When you need to issue requests to objects, keep the object that invokes the operation from the one that knows how to perform it. Keep the object that invokes the operation separate the object that knows how to perform the operation. Command interface has an execute method. Concrete Commands implements command interface, execute method contains how to perform the operation. Invoker asks the command to carry out requests. Receiver knows how to carry out requests when a command is executed.

Strategy - When you need an app’s behaviour to be set at runtime, involves a family of algorithms, interchangeable. Different approaches to the same problem. Allows client to choose which strategy to use. Interface Strategy: for all algorithms in this family. Concrete Strategy: implement the strategy interface and modify the approach. Context: sets the strategy and uses it.

Builder - To construct a complex object from simple objects using step by step approach. Used to separate the construction of a complex object from its representation so that the same construction process can create different representations. Product: the complex object we’re creating. Builder: an interface for creating the parts that make a product. Concrete Builder: implements the builder interface and builds a certain representation of product. Director: Constructs the object through the builder’s interface.

Factory – Create objects without exposing the creation logic to the client. Advantages: allows subclasses to choose the type of objects to create, promotes loose-coupling, which means the code interacts solely with the resultant interface or abstract class so that it will work with any classes that implement/extend it. Use when a class doesn’t know what subclasses will be required to create, when a class wants that its subclasses specify the objects to be created, when the parent classes choose the creation of objects to its subclasses.

* You need a Product interface, concrete products and a creator, factory that creates product and returns it

**Files -** BufferedReader/PrintWriter–reads one line/time -FileReader/FileWriter – reads/ writes byte by byte

Scanner -A simple text scanner which can parse primitive types and strings using regular expressions.

Pattern p = Pattern.compile(“regex”); this creates a pattern

Matcher m = p.matcher(“string”); creates a new matcher

m.matches(); checks if the matcher matches its pattern or you can do Pattern.matches(regex, input);

**Regex** - For capturing groups, they start at 0, 0 being the whole string

- Escaping to match the actual symbols: \^, \$, \[ - ^Anchoring$

- \d any digit, \D non digit - \s whitespace, \S non white space - \w a word char, \W a non word char

- [a-d[m-p]] union, a-d or m-p \*\*\* [a-z&&[def]] intersection, d e or f \*\*\* [az&&[^bc]] subtraction a to z except b or c

- Backreference

Pattern p = Pattern.compile("(\d\d\d)ABC\1"); #the \1 captures the first group

Matcher m = p.matcher("123ABC123");

System.out.println(m.matches()); #true System.out.println(m.group(1)); #123

**Floating Point Numbers** Rounding to even:

~24th bit is 0 – do nothing

~24th bit is 1 followed by 10,01,11 round up (add 1 to the 0 least significant digit)

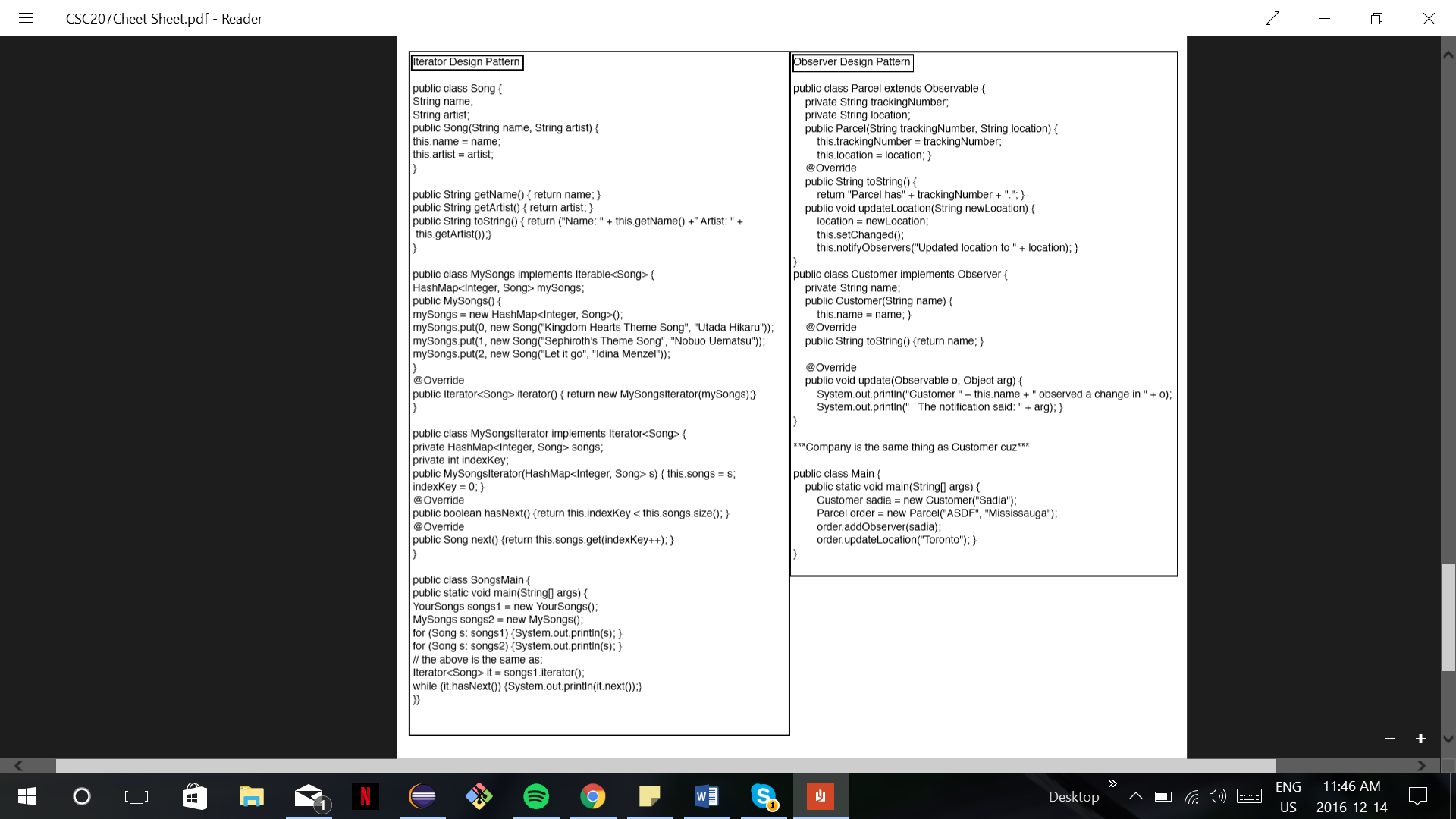
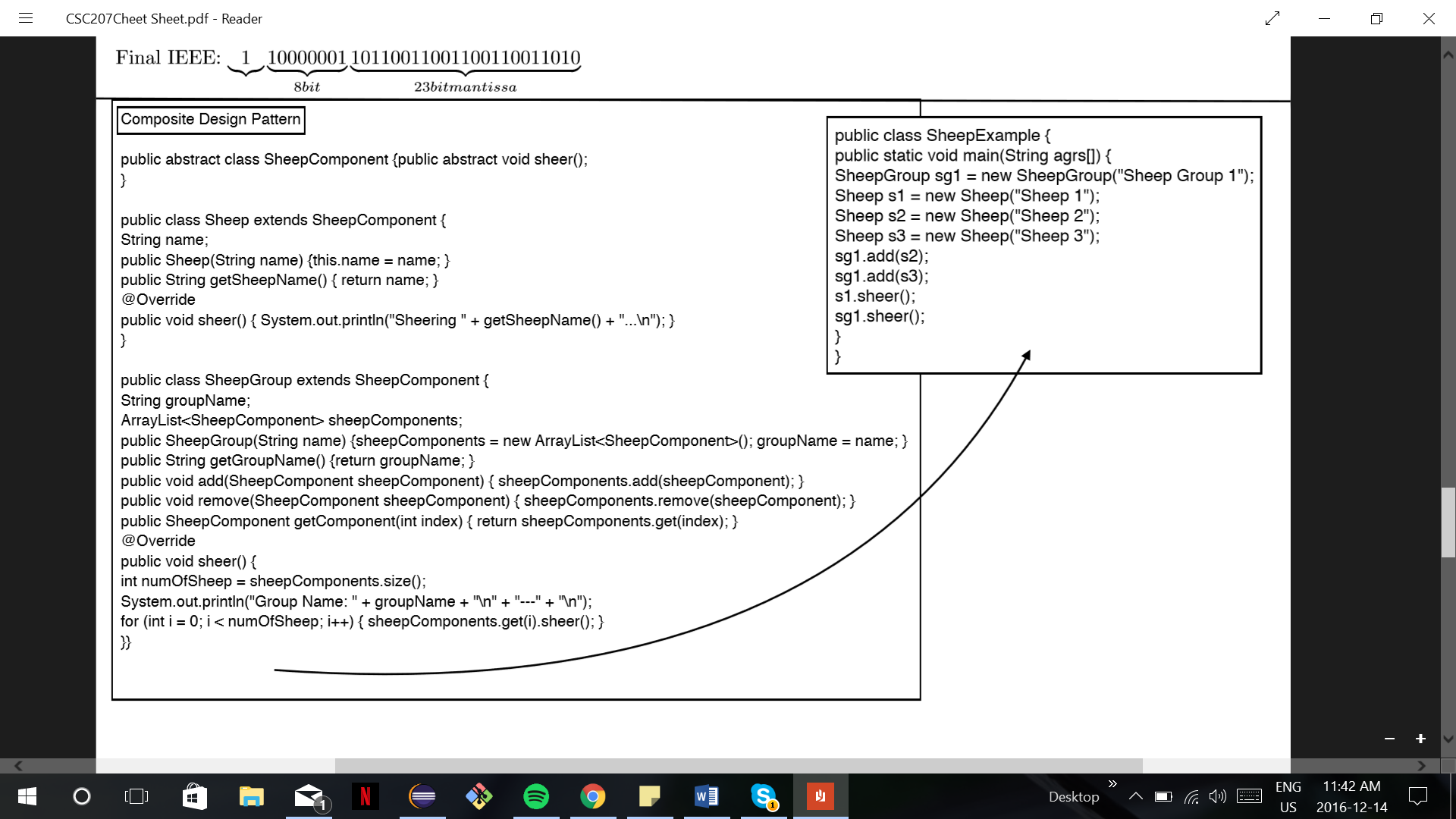
~Next 3 digits are 100, it’s a tie: if 23rd bit is 0 do nothing, if 1 round up.

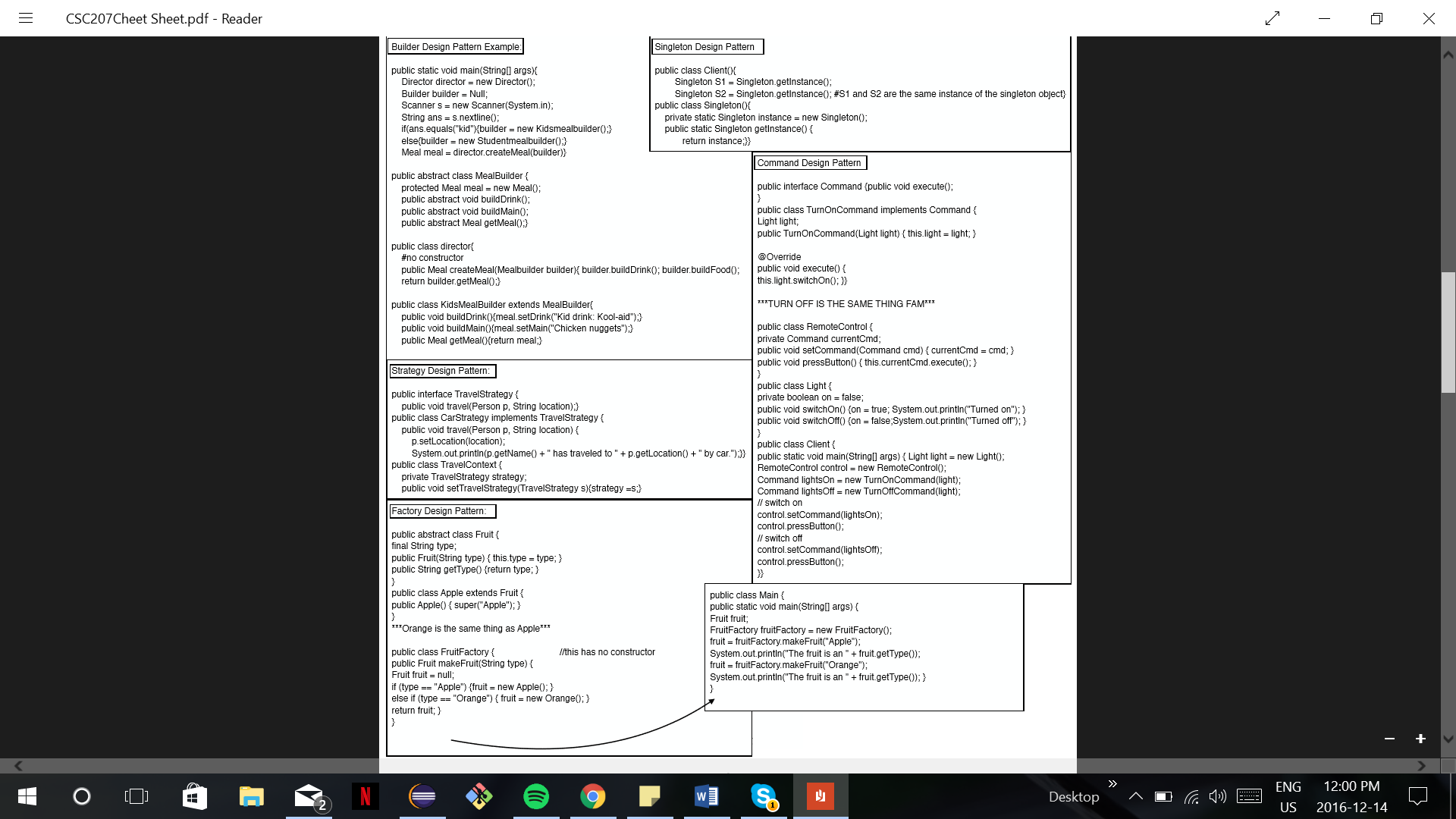
**Rounding v2: using GRS:**

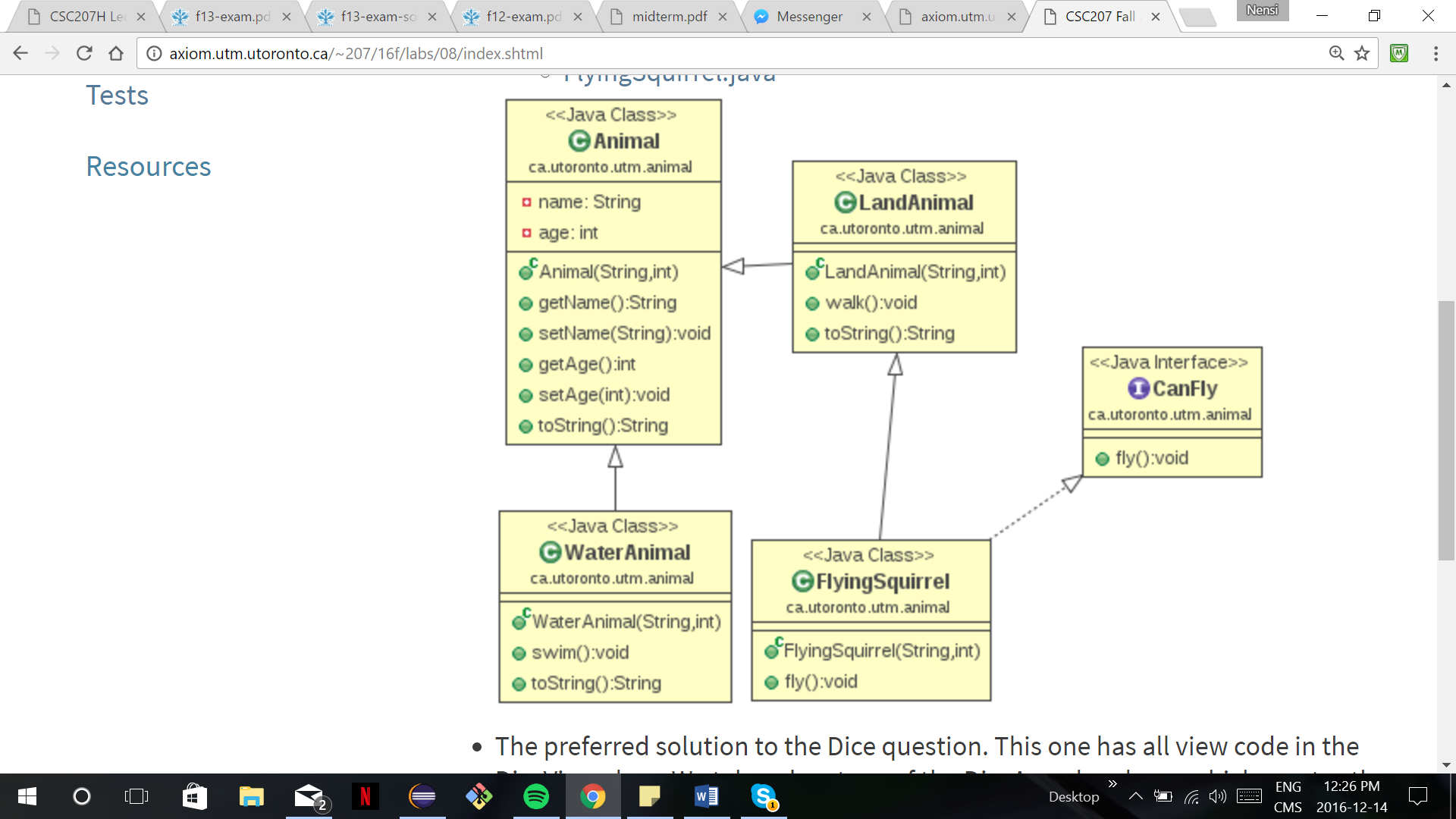
0xx – round down/ do nothing.

100 - round up if mantissa's bit just before G is 1, else round down/do nothing.

101/110/111 - round up

****

****

**Git Basics**

git pull [location name]

git add [ﬁle name] or . (adds all)

git commit -m (message)

git push [file name]

git checkout [branch name] – switches branches

git branch [branch name] – creates new branch or checks which branches exist.

git log – shows log of past push history

git clone [location] git merge [ﬁle name]

git branch -D [branch] Force delete the speciﬁed branch

git branch -d [branch] Delete the specified branch. prevents you from deleting the branch if it has unmerged changes.

git branch -m [branchName] rename branch to branchName

List 3 advantages of using a version control system like git.

1: Access to source on multiple systems

1: Ability to revert to previous versions of code

1: Ability to collaborate on development with others

1: Ability to work on many versions (branches) of code at the same time.

1: The repository is a 'backup' of the source.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Modifier** | **Class** | **Package** | **Subclass** | **World** |
| **Public** | **Yes** | **Yes** | **Yes** | **Yes** |
| **Protected** | **Yes** | **Yes** | **Yes** | **No** |
| **Default** | **Yes** | **Yes** | **No** | **No** |
| **Private** | **Yes** | **No** | **No** | **No** |

**Super**

In a subclass: -use super.attribute to refer to a variable or method in parent class -use super(attribute) to call a constructor defined in parent class

public class fileParser(){

private Pattern pname = Pattern.compile("regex");

public boolean parse(BufferReader input){

try{int state 0; Matcher m; String L;

while((L = input.readLine()) != null) {

switch(state){

case 0: m = pname.matcher(L);

if(m.matches()){int p1 = Integer.parseInt(m.group(1));

state = 1; break;}

error("message"; return false; }#state }#while }#try catch(Exception e){} return true; }#method }#class

**Scrum**

Scrum vs Waterfall: Iterative / frequent feedback / embraces change

Product Owner: Responsible for product backlog, represents users, expresses backlog items and orders them by value Development Team: Responsible for delivering potentially shippable increment of working software

Scrum Master: Removes obstacles, facilitates scrum events and communication

Product Backlog: Source of requirements for any changes to be made to the product. Ordered by value, risk, priority and necessity, estimated by team

Sprint planning meeting: Team selects items from backlog and defines a sprint goal, and the items are converted into tasks and estimated

Daily Scrum Meeting: Short meeting for the team to discuss what has been accomplished since last meeting, what will be done before the next meeting, and what obstacles are in the way

Sprint Review: Product Owner identifies what has been done, team discusses development process and demos current increment of software, product owner discusses current state of backlog, team decides what to do next.

NOTE: Interfaces can extend other interfaces; classes can implement multiple interfaces but extend a single class. B b = new A(); is wrong because of a type mismatch.

=======YOU GOT THIS! NO MATTER WHAT HAPPENS LIFE GOES ON! ☺ =======