== **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`

Composite Design Pattern

```
public abstract class SheepComponent {public abstract void sheer();
}

public class Sheep extends SheepComponent {
String name;
public Sheep(String name) {this.name = name; }
public String getSheepName() { return name; }
@Override
public void sheer() { System.out.println("Sheering " + getSheepName() + "...\n"); }
}

public class SheepGroup extends SheepComponent {
String groupName;
ArrayList<SheepComponent> sheepComponents;
public SheepGroup(String name) {sheepComponents = new ArrayList<SheepComponent>(); groupName = name; }
public String getGroupName() {return groupName; }
public void add(SheepComponent sheepComponent) { sheepComponents.add(sheepComponent); }
public void remove(SheepComponent sheepComponent) { sheepComponents.remove(sheepComponent); }
public SheepComponent getComponent(int index) { return sheepComponents.get(index); }
@Override
public void sheer() {
int numOfSheep = sheepComponents.size();
System.out.println("Group Name: " + groupName + "\n" + "---" + "\n");
for (int i = 0; i < numOfSheep; i++) { sheepComponents.get(i).sheer(); }
}}
```

```
public class SheepExample {
public static void main(String agrs[]) {
SheepGroup sg1 = new SheepGroup("Sheep Group 1");
Sheep s1 = new Sheep("Sheep 1");
Sheep s2 = new Sheep("Sheep 2");
Sheep s3 = new Sheep("Sheep 3");
sg1.add(s2);
sg1.add(s3);
s1.sheer();
sg1.sheer();
}
}
```

Iterator Design Pattern

```
public class Song {
String name;
String artist;
public Song(String name, String artist) {
this.name = name;
this.artist = artist;
}

public String getName() { return name; }
public String getArtist() { return artist; }
public String toString() { return ("Name: " + this.getName() +" Artist: " +
 this.getArtist());}
}

public class MySongs implements Iterable<Song> {
HashMap<Integer, Song> mySongs;
public MySongs() {
mySongs = new HashMap<Integer, Song>();
mySongs.put(0, new Song("Kingdom Hearts Theme Song", "Utada Hikaru"));
mySongs.put(1, new Song("Sephiroth's Theme Song", "Nobuo Uematsu"));
mySongs.put(2, new Song("Let it go", "Idina Menzel"));
}
@Override
public Iterator<Song> iterator() { return new MySongsIterator(mySongs);}
}

public class MySongsIterator implements Iterator<Song> {
private HashMap<Integer, Song> songs;
private int indexKey;
public MySongsIterator(HashMap<Integer, Song> s) { this.songs = s;
indexKey = 0; }
@Override
public boolean hasNext() {return this.indexKey < this.songs.size(); }
@Override
public Song next() {return this.songs.get(indexKey++); }
}

public class SongsMain {
public static void main(String[] args) {
YourSongs songs1 = new YourSongs();
MySongs songs2 = new MySongs();
for (Song s: songs1) {System.out.println(s); }
for (Song s: songs2) {System.out.println(s); }
// the above is the same as:
Iterator<Song> it = songs1.iterator();
while (it.hasNext()) {System.out.println(it.next());}
}}
```

Observer Design Pattern

```
public class Parcel extends Observable {
   private String trackingNumber;
   private String location;
   public Parcel(String trackingNumber, String location) {
      this.trackingNumber = trackingNumber;
      this.location = location; }
   @Override
   public String toString() {
      return "Parcel has" + trackingNumber + "."; }
   public void updateLocation(String newLocation) {
      location = newLocation;
      this.setChanged();
      this.notifyObservers("Updated location to " + location); }
}
public class Customer implements Observer {
   private String name;
   public Customer(String name) {
      this.name = name; }
   @Override
   public String toString() {return name; }

   @Override
   public void update(Observable o, Object arg) {
      System.out.println("Customer " + this.name + " observed a change in " + o);
      System.out.println("   The notification said: " + arg); }
}

***Company is the same thing as Customer cuz***

public class Main {
   public static void main(String[] args) {
      Customer sadia = new Customer("Sadia");
      Parcel order = new Parcel("ASDF", "Mississauga");
      order.addObserver(sadia);
      order.updateLocation("Toronto"); }
}
```

Builder Design Pattern Example:

```
public static void main(String[] args){
   Director director = new Director();
   Builder builder = Null;
   Scanner s = new Scanner(System.in);
   String ans = s.nextline();
   if(ans.equals("kid"){builder = new Kidsmealbuilder();}
   else{builder = new Studentmealbuilder();}
   Meal meal = director.createMeal(builder)}

public abstract class MealBuilder {
   protected Meal meal = new Meal();
   public abstract void buildDrink();
   public abstract void buildMain();
   public abstract Meal getMeal();}

public class director{
   #no constructor
   public Meal createMeal(Mealbuilder builder){ builder.buildDrink(); builder.buildFood();
   return builder.getMeal();}

public class KidsMealBuilder extends MealBuilder{
   public void buildDrink(){meal.setDrink("Kid drink: Kool-aid");}
   public void buildMain(){meal.setMain("Chicken nuggets");}
   public Meal getMeal(){return meal;}
```

Strategy Design Pattern:

```
public interface TravelStrategy {
   public void travel(Person p, String location);}
public class CarStrategy implements TravelStrategy {
   public void travel(Person p, String location) {
      p.setLocation(location);
      System.out.println(p.getName() + " has traveled to " + p.getLocation() + " by car.");}}
public class TravelContext {
   private TravelStrategy strategy;
   public void setTravelStrategy(TravelStrategy s){strategy =s;}
```

Factory Design Pattern:

```
public abstract class Fruit {
final String type;
public Fruit(String type) { this.type = type; }
public String getType() {return type; }
}
public class Apple extends Fruit {
public Apple() { super("Apple"); }
}
***Orange is the same thing as Apple***

public class FruitFactory {                    //this has no constructor
public Fruit makeFruit(String type) {
Fruit fruit = null;
if (type == "Apple") {fruit = new Apple(); }
else if (type == "Orange") { fruit = new Orange(); }
return fruit; }
}
```

Singleton Design Pattern

```
public class Client(){
      Singleton S1 = Singleton.getInstance();
      Singleton S2 = Singleton.getInstance(); #S1 and S2 are the same instance of the singleton object}
public class Singleton(){
   private static Singleton instance = new Singleton();
   public static Singleton getInstance() {
         return instance;}}
```

Command Design Pattern

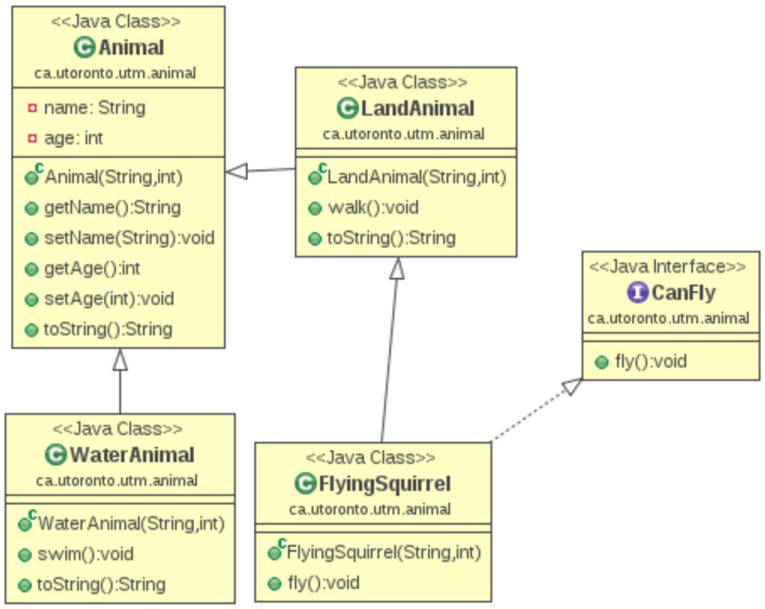
```
public interface Command {public void execute();
}
public class TurnOnCommand implements Command {
Light light;
public TurnOnCommand(Light light) { this.light = light; }

@Override
public void execute() {
this.light.switchOn(); }}

***TURN OFF IS THE SAME THING FAM***

public class RemoteControl {
private Command currentCmd;
public void setCommand(Command cmd) { currentCmd = cmd; }
public void pressButton() { this.currentCmd.execute(); }
}
public class Light {
private boolean on = false;
public void switchOn() {on = true; System.out.println("Turned on"); }
public void switchOff() {on = false;System.out.println("Turned off"); }
}
public class Client {
public static void main(String[] args) { Light light = new Light();
RemoteControl control = new RemoteControl();
Command lightsOn = new TurnOnCommand(light);
Command lightsOff = new TurnOffCommand(light);
// switch on
control.setCommand(lightsOn);
control.pressButton();
// switch off
control.setCommand(lightsOff);
control.pressButton();
}}
```

```
public class Main {
public static void main(String[] args) {
Fruit fruit;
FruitFactory fruitFactory = new FruitFactory();
fruit = fruitFactory.makeFruit("Apple");
System.out.println("The fruit is an " + fruit.getType());
fruit = fruitFactory.makeFruit("Orange");
System.out.println("The fruit is an " + fruit.getType()); }
}
```



Git Basics
git pull [location name]
git add [Þle 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 [Þle name]
git branch -D [branch] Force delete the speciÞed 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! ☺ =======