**Statics:**

* Static fields and static methods do not belong to a single instance of a class
* Common explanation is that statics belong to the entire class, not just one instance
* Main is static because the class needs to access that
  + Main is also global, and apparently globals are bad
* Static fields and methods are properties of a class
  + Static methods are good for utility i.e. it will be used from multiple classes, multiple instances
* Non-static fields and methods are properties of an instance
  + Non-static or instance is the same thing
  + Every time new is used, it’s a new instance of a method i.e. new int[4];
* Example, every math method is static because it makes much more sense to have a static method available to every class than to force a new instance of math class every single time there needs to be a math call. Math.sqrt(25.0); instead of Math myMath = new Math(); myMath.sqrt(25.0);
* You can access statics anywhere, and since global is bad, don’t have static unless its for utilitarian stuff
* Primitive static fields are initialized to 0 if no initialization is performed
* All instances of class shares the same static fields
  + I feel like that would explain some bad behavior in some of my code
* Ask yourself, does it make sense to call this method without constructing an Object?
* Static can’t access instance fields
  + Reason why I add static to everything but apparently that’s bad
* Reasons why statics are bad:
  + Static is global, and the memory exists the entire runtime of the program
  + No thread safety
  + Not good security
* Static methods can’t be overridden because method overriding only occurs in thecontext of runtime

**Objects:**

* Objects passed to methods as arguments
* Java passes arguments by value
* When an object is passed as an argument, the value of the reference is passed
* Basically any type that isn’t the primitive 8 types are just passed as a pointer to the object
* In the example, r is just a pointer to globalRect
  + public Rectangle globalRect;
  + public void displayRectangle(Rectangle r){ System.out.println(r.getLength()); }
* Methods can return any object type, not just primatives

**Strings:**

* Good programming practive for each class is to have a **toString()** method in each class, which will return the “state” of an object created in a class
  + In RobotTorso.java, RobotHead.java, etc, there is a toString function that returns how many arms or color of eye, etc.
    - If you try and System.out.println() a class object, it will look for **toString()**, else it will print the name and address of the object
* Strings are immutable
  + Changing String dog = “Rex”; 🡪 dog = “fido”; doesn’t actually change the string stored at location dog, it points dog to a brand new address that has a brand new address. STRINGS ARE IMMUTABLE. PERIOD
  + Brand new address every time a string is changed, pointed to a new location
* StringBuilder can change specific characters, insert, delete, etc, without creating new object and address
  + Create StringBuilder by StringBuilder(), StringBuilder(int length), StringBuilder(String str)
* StringTokenizer doesn’t exist anymore, **split** instead
  + split(String regex, int limit); //doesn’t need limit
    - regex = regular expression
  + returns an array of strings
  + throws PatternSyntaxException if the regular expression syntax isn’t valid
  + “geekss@for@geekss”.split(“@”,2) 🡪 “geekss” & “for@geekss”

**Regex:**

* . matches any character
* ^regex finds regex that matches at the beginning of the line
* Regex$ finds regex that matches at the end of the line
* [abc] can match the letter a or b or c
* [abc][vz] can match a or b or c followed by either v or z
* [^abc] negates the pattern. Patter matches any character except a or b or c
* \d is digit
* \D non digit
* \s is any whitespace
* \S non-whitespace char
* \w a word character
* \W a non-word
* Many many many more

**Inheritance:**

* Every single Java class inherits from Object, which provides toString() and equals()
* Every plane has wings, every plane has engine, every \_\_\_ has something, use inheritance!
* Get rid of duplication by having similar functions in the large class, and have smaller classes inherit from large class
* public class F18Hornet extends Airplane 🡪 F18Hornet inherits all fields
* classes that inherit from superclasses will not inherit private fields, but often classes can access superclass private fields through getters and setters
* Auto is superclass, SportsCar is subclass
* Subclasses don’t inherit every field in superclass. Kind of….
  + In the memory of the computer, subclasses have an address for every field in the superclass, but subclasses can’t access private fields without public getters or setters
  + Technically, subclasses only inherit the public fields, but all fields are initialized in memory so superclass can still use its own fields
* In order to create SportsCar, you need to create a new Auto class and have SportsCar inherit from Auto
* When subclass is instantiated, the superclass default constructor is execuded first
  + You have to have a parent before you can have a baby
* SportsCar mclaren = new SportsCar() 🡪 calls Auto() first then it will call SportsCar() constructor
* public SportsCar(){ super(); hasLeatherSeats = false; } = public SportsCar() { hasLeatherSeats = false;} because SportsCar extends Auto, so Auto superclass constructor will be called first
  + Calling super(); is just a way of calling the super constructor method
    - super() can take arguments if superclass constructor calls for it
      * If super() requires arguments, you need to actually call it in subclass constructor

**Overloading/Overriding:**

* Overloading is methods with the same name, different arguments
  + Works with inheritance
    - Animal has eat(), Dog has eat(String)
      * Can call eat() and eat(String) from Dog
* Overriding is a superclass and subclass with the same name
  + Auto has getGasMileage() and SportsCar has getGasMileage()
    - If Auto getGasMileage() is private, SportsCar getGasMileage() is ***NOT*** overriding, just creating a new class
  + Subclasses will look within themselves for method before going to check in superclass
    - Subclass overrides superclass method
  + Can call super method from subclass by super.getGasMileage()

**Protected:**

* Private is most restrictive and most hidden, public is most open, protected is somewhere between
* In UML diagram, # represents protected
* Protected methods can be accessed by children objects, but private to the rest of the world
* Protected works for fields and methods

**Copies:**

* The best way to copy is by making a copy constructor
  + Don’t just make a new object with the same arguments
* Make a method that returns type of class
  + For example,
    - public Car copy(){
      * Car copyCar = new Car(color, engine, name, doors, ….”);
      * return copyCar;
    - }

**Aggregation:**

* An object made up of other objects
  + House has door, window, room, etc
* In a UML diagram, an object that contains some other class will be represented by a diamond
  + Left box contains right box

**This:**

* Why would we ever need this?
  + You can live without ‘this’, but it makes some code clearer
* In constructor, when you set a class variable equal to a constructor argument, using this.myVar = myVar; will set the class var equal to the constructor argument myVar
* ‘this’ references itself inside its own class
  + Basically, MY variable = argument variable

**Primatives:**

* byte, short, int, long, float, double, boolean, char

**Enums:**

* enum typeName { (one or more enum constants}
  + enum Day {SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY}
    - Day dayOfWeek = Day.WEDNESDAY;

**Autoboxing (boxing), Unboxing:**

* Autoboxing (boxing)
  + Converting a primitive valye into an object of the corresponding wrapper class is called autoboxing. For example, converting int to Integer class
    - Passed as a parameter to a method that expects an object of the corresponding wrapper class
* Unboxing
  + Converting from wrapper class to primitive
* Types
  + char, Character
  + byte, Byte
  + short, Short
  + int, Integer
  + long, Long
  + float, Float
  + double, Double
  + boolean, Boolean
* Only reason to use class wrapper instead of primitive is to use a lot of prebuilt methods
  + Integer.parseInt(double a); is an example

**Polymorphism:**