Intro to CompSci (This was written in a *loose* tone to make it easier to read.)
By - TAs Moshe Rosensweig and Aaron Shakibpanah
What are Objects - important terms and concepts

[] Introduction

This is part II of the Object Review. This will focus on what different parts of an object mean. (This is going to be a bit long - because it has to be.)

[1] What is an object?

An object is the way the computer stores information about objects in the real world. Meaning, if you want to tell the someone about a car that you own, how would you do that? First, you'd make sure that they understand the concept of a car. Then you'd tell the specifics. In this example, you'd first tell that person "a car is a vehicle that has 4 wheels, a color, a company that makes it, a gas mileage, number of seats, an owner, etc…" Then you'd tell them "my car is a blue, Honda, that can hold 8 people, and has a gas milage of 7, etc…" What you've just done, is explained what a car is and more specifically your car. The same is true for computers.

How do you explain what a car is to a computer? The answer is, by writing a class. There you explain to the computer that there's this concept called a car with all these details (the ones we mentioned above). When you want to tell the computer about your specific car - you do that by constructing a *new* car. When you tell your computer about something your car did - you do that by *invoking* one of the car's methods. (Disclaimer: objects do not need to reflect things in the real world, but often they do and it is a good way to think about them.)

Let's get a bit more detailed.

[2] Fields

When you explain to the computer what a car is, you tell it that it has all sorts of traits/characteristics. For (a simple) example "a car has a color and a maximum number of passengers." In this case, the computer will recognize anything which has (1) a color and (2) a number of passengers, as a car - because you told the computer that that's what a car is. If you told the computer that "a car has a color, a maximum number of passengers and was made by a specific manufacturer" then the computer would only recognize it as a car if you told it those 3 pieces of information.

As you would expect, those traits are *encapsulated* as *fields*. When you write *instance fields* (or *global variables)* as we mentioned in the previous document, you need to group your fields together. This is important because it allows you to look over the different traits - your object should have - at once.

[Static vs Instance fields]

First we need to understand the difference between *static* and *instance*. Instance refers to something limited to an object, while static refers to something related to the class, regardless of it's instances. For example, in the car class, the field "number of

wheels" does not change from car to car. Whether I'm talking about my car, your car, or X's car, I know that it will have 4 wheels. To explain this to the computer, I would write "static int numberOfWheels = 4;" What this also means, is that if the number of wheels changes at some point in time so now cars have 5 wheels, you would just change "numberOfWheels = 5" and all cars would have that change.

On the other hand, things that are specific to this car would be *instance fields.* My car's color is not the same and has nothing to do with you car's color. Therefore, I would write "Color color = BLUE;" without the word *static*. If I decided to paint my car red, then when I do "color = RED" that doesn't effect any other car (unlike "numberOfWheels, where a change, effected **all** cars).

Because of this difference, the way you initialize fields are different. A *static* field should be initialize right away. In our example, when you write the "Car" class, you would fill in the "numberOfWheels" field as equal to 4 in the code, because the number of wheels a car has, has nothing to do with any specific car. However, when you write the "color" field, you would have to leave it uninitialized as such "Color color;" so that the user can fill it in later by using a constructor (see constructors below). (In theory, if there's a default value, you could initialize the field in the code itself, but often that's not the case.)

[3] Constructors

The constructor is where you tell the computer about your specific car. "I have a red, 8-seater, Honda" would translate to a constructor like this:

```
/**
  * Set all the fields with parameters
  */
 public Car(int numOfSeats, Color carColor, String carCompany){
      numberOfSeats = numOfSeats;
      color = carColor;
      company = carCompany;
 }
```

And the client would write code like this "Car myHonda = new Car(8, RED, "Honda");" This means the computer will now create a new "Car" objected called 'myHonda" which has all of those properties. As you can see. There values "numberOfSeats," "color," and "company," are initialized in the constructor.

[this and *Parameters]*

Another way to write this constructor is as follows:

```
/**
  * Set all the fields with parameters
  */
 public Car(int numberOfSeats, Color color, String company){
      this.numberOfSeats = numberOfSeats;
      this.color = color;
      this.company = company;
```

```
}
```

Look carefully at this constructor. This time the parameter names are the same as the field names. On one hand this is helpful because we can know exactly what each parameter is supposed to align with. On the other hand, if I wrote "color = color" which color would I be referring to on the left or right? To clarify this (and to make it work) we use the keyword *this*. When you have two fields that have the same name, either because one's a parameter and one's a field or because you're using two objects (see later) you need a way to distinguish. The keyword *this* does that. For more on that see *methods*, but for now, know that the word "*this.name*" refers to the object's field (not it's parameter).

[4] Methods

Methods are the actions an object can do. In our example, a Car can drive, stop, park, etc… The way we tell the computer to have a car do these things is to *invoke* the object's method. You do that by writing "object.method()" so for our example it would be "myHonda.drive()". "Drive" is a method you wrote, and it has no parameters. A drive method with a parameter would look like this "public void drive(int speed)". When you invoke that method, you need to give it an int as a parameter, in this case, the parameter would be the speed you want the car to drive at.

[Return types and what they indicate]

How should I know what a method's return type should be? Well it depends, are you asking the object to give you information or not. If you're not asking your object to give you information back, then the return type would be *void*. This means that you do not "return" anything from this method and (in general) should not use the term *return* at all.

Well, if the object isn't returning anything, why am invoking it? The answer is that generally *void* methods change the state of a field. For example, the "paint(Color color)" would be used to change the "color" field of the car. There's nothing the car needs to tell me when this method runs.

On the other hand, a method with a return type means that the method is giving you information. This means that no matter how the code runs, it must end with a return statement. This means, if you have an if/else statement, both the "if" and the "else" need to have their own return statements - or you could just put the return statement after both the if/else.

[this]

When you invoke a method and pass another object as the parameter, it will become important to use the term *this*. For example if you do "myHonda.crash(yourFord);" (assuming cars have an "isCrashed" field) the code for the method would look something like this:

```
/**
 * Crash Method - turns both cars's "isCrashed" to true
 */
public void crash(Car thatCar)
{
        this.isCrashed = true; //this could also be written as:
```

```
this.setIsCrashed(true);
            thatCar.setIsCrashed(true);
        }
```

In this method you're dealing with 2 Car objects, "myHonda" and "yourFord". Since they're both Cars, they both have an "isCrashed" field. How are you supposed to know which one is which? Well, whichever one is being *invoked* is *this*. In this example, we *invoke* myHonda's "crash" method, so we're asking myHonda to do an action - in this case, to crash into another car. Therefore, the crash method is being run by myHonda and not by yourFord. Consequently, *this* refers to the object invoked, so in our example, since myHonda is invoked, any use of the term *this* will refer to myHonda as a whole. In our crash method, we wrote "this.isCrashed" which is equivalent to saying "myHonda.isCrashed".

In this case, you wouldn't have to use the term *this*, because without it, the default assumption is that you're referring to the invoked object, so when if you just write "isCrashed = true;" that would also work. However, using the term *this* here, adds clarity to the reader/programmer, so it's not a bad idea to write *this* anyways.

[static methods]

We saw a fair explanation on stack overflow:

"*One rule-of-thumb: ask yourself "does it make sense to call this method, even if no Obj has been constructed yet?" If so, it should definitely be static.*

So in a class `Car` *you might have a method* `double convertMpgToKpl(double g)` *which would be static, because one might want to know what 35mpg converts to, even if nobody has ever built a Car. But* `void setMileage(double mpg)` *(which sets the efficiency of one particular Car) can't be static since it's inconceivable to call the method before any Car has been constructed.*"

(Another example you're familiar with is the "Math" class. There you use the *class methods* without creating "Math" objects. For example, you probably have Witten "Math.abs(value);" In this case, you're *invoking* the "Math" class and using it's static method call "abs".)

[5] Privacy

In general your fields should be private. What this means is, if for example you have "private int speed;" - other code cannot do "int yourSpeed = myHonda.speed" If other objects or classes need access to the field, make getters and setters. Wait, doesn't it defeat the point of making it private? To answer this I defer again to stack overflow

"*A public member [field] can be accessed from outside the class, which for practical considerations means "potentially anywhere". If something goes wrong with a public field, the culprit can be anywhere, and so in order to track down the bug, you may have to look at quite a lot of code.*

A private member, by contrast, can only be accessed from inside the same class,

so if something goes wrong with that, there is usually only one source file to look at. If you have a million lines of code in your project, but your classes are kept small, this can reduce your bug tracking effort by a factor of 1000."

This is part of the basic argument. Even if you don't fully get what he's saying, accept for now that you should by default make your fields private and if you need to, use getters and setters. (If you make a field *final*, that means it cannot be changed ever! Then it's less bad to make it public.)

(In general there are 4 different types of access a field could have: (1) private (2) protected (3) public (4) undeclared. This is beyond the scope of this document, see here for more.)

(Privacy is good for methods too, but for brevity's sake, we won't discuss it here.)