Intro To CompSci Cheat Sheet (This was written in a *loose* tone to make it easier to read.)

By TAs: Moshe Rosenswieg and Aaron Shakibpanah

The following are areas that we have noticed many people could use further clarification. We will try to use brevity.
It's a useful review even if you think you've got all of these down pat.

[1] Declaring and using variables

Examine the following code excerpt:

[Version 1]
```
int a = 1;
      if(a < 2) {
            int a = 5; // do not redeclare "a"
      }
```
[Version 2]
```
int a = 1;
      if(a < 2) {
            double a = 5; // do not redeclare "a"
      }
```
What's the problem here? The compiler will tell you that a variable is already defined. When you want to use the variable "a" a 2nd time, you DO NOT and CANNOT redeclare it.
Instead your code should look like this:

```
int a = 1;
      if(a < 2) {
            a = 5;
      }
```

For all compiler errors you come across, look at this http://www.cs.williams.edu/~cs134/pages/errors.pdf

[2] The scope of variables

Examine the following code excerpt:

```
int[] tempAr = {1,2,3,4,5};
      for(int i = 0; i < tempAr.length; i++) {
            System.out.print(tempAr[i]);
      }
```

```
        return i; //this variable has not been declared
```

What's the problem here? The compiler will tell you that it "cannot find symbol" (if I recall correctly).
The problem here is that you only declared "i" inside your for loop, therefore, at the end of the "for" loop, the variable "i" will be erased. Consequently, you cannot return the value of "i" because it no longer exists and you have not declared it after the loop. Instead your code should look like this:

```
int[] tempAr = {1,2,3,4,5};
        for(int i = 0; i < tempAr.length; i++) {
                System.out.print(tempAr[i]);
        }
        int i = 0; //notice we redeclared I
        return i;
```

Or don't use a for loop and do this:

```
int[] tempAr = {1,2,3,4,5};
        int i = 0; // notice, we declared it before the loop
        while(int i < tempAr.length) {
                System.out.print(tempAr[i++]);
        }
        return i;
```

[3] Exceptions

What are exceptions and why do we use them? This is a big topic, there's a lot out there, and it's worth reading, but I'm going to give the very basics here.

[A] Two Categories of exceptions
There are two types of exceptions: (1) Unchecked (2) Checked.

(1) **Checked** - These are problems that you can anticipate at compile time and therefore, if you don't handle them properly the compiler will yell at you. An exception that is not dealt with will have a compiler message like this "`unreported exception FileNotFoundException; must be caught or declared to be thrown.`" For example:

FileNotFoundException - when you try to make a new file, if a problem comes up while trying to. make it, you'll get a FileNotFoundException. You (as a programmer) know that when you set try to open a new file, a problem could arise. So, you have to take that into account (see below), otherwise you'll get the compiler mad - and you don't want the compiler mad.

(2) **Unchecked** - These are problems you don't expect to happen - and they

generally cause the program to shut down. For example, if for some reason you try to divide by zero, or use a null value inappropriately, or you try to access an array entry that's too big —> these are problems that you generally cannot continue from. Consider the following code:

```
int[] tempAr = {1,2,3,4,5};
      int i = 0;
      while(int i < tempAr.length + 1) { // notice the "+1"
            System.out.print(tempAr[i++]);
      }
      return i;
```

This code will, at runtime try to access "tempAr[5]", then it wants to print it out. The problem is that you cannot print out what doesn't exist! The compiler won't catch this, because at compiler time, it doesn't make sure that all iterations of the loop will work out (that's what runtime is for). In fact, you know that the first set of loop iterations will work, just the last one won't. Meaning, it **will** print out "12345" on your screen before it throws the **ArrayOutOfBoundsException**. However, once the exception is thrown, your code goes home (i.e. it ends).

(To summarize, I quote from bluejay "*Slightly simplified, the difference is this: checked exceptions are intended for cases where the client should expect that an operation could fail (for example, if it tries to write to a disk, it should anticipate that the disk could be full). In such cases, the client will be forced to check whether the operation was successful. Unchecked exceptions are intended for cases that should never fail in normal operation—they usually indicate a program error. For instance, a programmer would never knowingly try to get an item from a position in a list that does not exist, so when they do, it elicits an unchecked exception.* ")

[B] Types of exceptions
There are may different types of exceptions. Above we mentioned a two - **FileNotFoundException, ArrayOutOfBoundsException**. Why? So that you can give more specific details about the problem by specifying the type of exception thrown. That already tells you the general idea of what went wrong. (See the next two sections, about how you can get even more specific with reporting problems with exceptions.)

[C] Throwing your own exception
You can throw your own exception. This is helpful for several reasons. Firstly, because it can allow you to more specifically identify the problem. Instead of a generic **FileNotFoundException**, you could give it a description of "When you tried to open the file in the beginning of the code, it broke because you gave me a bad file path." Then when you catch it (see below) you can distinguish this specific problem from others. How do you throw one? Use the "throw" clause. For example:

if(a < b) throw new Exception("a was less than b");

You can also throw an unchecked exception as follows:

```
if(a < b) throw new RuntimeException(“a was less than b”);
```

[D] Recovery / Dealing with exceptions
Ok, an exception was thrown, now what? Well, it depends. If it’s an unchecked exception, you don’t do anything. Like we said above - unchecked exceptions are problems that you cannot deal with and just let the program die. You don’t need to “try” or “catch” an unchecked exception. However, you may “catch” an unchecked exception if you think it’s a good idea.
When you’re dealing with code that will throw a checked exception, you need to “try” and “catch” it.

(1) **Try** - You write a try **block** like you would an “if” block or a “while” block, with curly brackets - like this “try{ }”. That means that any variable declared inside the “try” will be erased when you leave the try block, **regardless of whether or not an exception was thrown**. Try means that you’re *trying* to do something that might throw an exception - hence the name try. You **can** have one “try” for many things that could throw exceptions. Meaning, in a try{ I could try to open a file, and I could try to write to a file } // both things that could throw exceptions. When you write a “try”, write it as if the code will work properly, just have in mind that it could fail. Once an exception is throw, none of the rest of the code in the “try” will run. Instead, the program will jump to the “}” of the “try” and move on to the “catch”.
(2) **Catch** - You must have a “catch” block immediately after the “throw”. And yes, it is also a block, so any variable declared inside the “{}” will be erased when you leave. Anyways, a “catch”, catches exceptions. It can catch a specific exception ex: catch(FileNotFoundException e) or it can catch all types of exceptions: catch(Exception e). The first type of “catch” catches only FileNotFoundExceptions. This is helpful if you know you’re “try” might send several different types of exceptions. This allows to react differently for a FileNotFoundException than an ArrayOutOfBoundsException. If you don’t care about the specific type of exception being thrown, you can use generic catch(Exception e). If you care about some types of exceptions and not others, use both, for example:

```
try{
//something that shows an exception or two
}
catch(FileNotFoundException e){
//do something specific for FileNotFoundException’s
}
catch(ArrayOutOfBoundsException e){
/do something specific for ArrayOutOfBoundsException
}
catch(Exception e){
//do something for all the other types of exceptions that aren’t FileNotFoundExceptions
```

or ArrayOutOfBoundsExceptions.
}

(3) **finally** - For the sake of brevity, I will not cover finally here.