* Enums
  + A type of class that can only take on a small finite set of values that we only want to call by name, enums are perfect
  + Ex:

public enum Month{

JANUARY,

FEBRUARY,

MARCH,

APRIL,

MAY,

JUNE,

JULY,

AUGUST,

SEPTEMBER,

OCTOBER,

NOVERMBER,

DECEMBER

}

* Good practice to make all caps.
* Month.OCTOBER calls the october object within the enum
* Class Inheritance
  + Use keyword "extends"
  + Main reason we have this is so we don't repeat code
  + Each class can only inherit from one class
    - They can inherit from as many interfaces as possible, but only one class!!!
* Semantics - one word that can mean different in different contexts
  + Ex: An athlete can run, a refrigerator can run, and a nose can run
  + But if you use "Is-a" relationship, you can explicitly give a context to a class
    - Ex: public class Chevy extends Car{…}
* Inheritance casting
  + So if you have an object that has a superclass and want to call a method within the superclass, do this:

DNAStrand myStrand;

If(head instanceof SuperDNAStrand){

((Super DNAStrand myStrand).someMethod());

}

* Constructor Chaining In Inheritance - you can call constructors in supermethods

class A{

int x;

String a;

public A(int num1, String s){

this.x = num1;

this.a = s;

}

}

class B extends A{

public B(int num1, String s){

super(num1, s);

}

}

* Null
  + Any reference-typed variable can take null values
  + If you call a method on a null-valued variable, you will get a NullPointerException
  + Ex:

object = null;

object.method(); ---> this will result in a NullPointerException!

* String s1 = null;
* String s2 = "";

S1 is not the same as s2!

* Every Java class inherits from the Object class, even if you don't say so
  + Thus the Object type variable can hold any reference type
  + But you won't be able to call any methods you have defined as a programmer
  + You don't need it often, but know that it exists
* Access modifiers
  + The keywords public and private are called access modifiers
  + Tells java which over classes may access your fields or methods, which is critical for encapsulation

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Keyword | Same class? | Same package? | Subclasses? | Everybody? |
| Public | Yes | Yes | Yes | Yes |
| Protected | Yes | Yes | Yes | No |
| None/default | Yes | Yes | No | No |
| Private | Yes | No | No | no |

* Sometimes you would want to make it protected if it has to be accessed but you don't have a getter
* Four pillars of object orientation
  + Encapsulation
    - Access modifiers
  + Modularity
    - Taking stuff apart and putting back together to make new classes
  + Inheritance of classes
  + Polymorphism
    - Interfaces
    - Overriding
    - Overloading
* Late binding

class A{

int foo(){

return 5;

}

}

Class B extends A{

int foo() {

return 6;

}

}

//then whe you do this:

A a = new B();

a.foo(); ----> this will return 6, not 5, because of late binding. Java doesn't know until it tries what type of object will be in a when it runs, so it sees that it is a new B object and call B's foo rather than A's. Early binding: java knows before running which method to call, so in this case it would know it is of type a, so it returns 5.

* Every method in java is virtual, or does late binding!
  + This will be on a test, according to Brian the grad TA

* Know when you can override and when you can't
  + You can't override just by giving it a different type
  + To overload, you have to change the parameters