* OOP = create software objects that are associated with state/data, and define object as a collection of fields that is interpreted in a certain way
* major themes of the course
  + abstraction
  + encapsulation
  + inheritance
  + polymorphism
  + delegation
  + design patterns
  + asynchronous programming
* value types - integers, real numbers, booleans, characters
* reference types - string, array, objects typed by their class, classes themselves
* method signature
  + access modifier
    - public,private,protected
    - if unspecified, then "package" access
  + method type
    - static or default
    - static = class method, otherwise instance method
  + return type
  + method name
    - must start with a letter, $, or \_
    - can contain letters, numbers, $, or \_(no spaces or other punctuation)
  + parameter list
* java execution model
  + your program is always executing within the context of some method
  + starts off in the main class method defined in whatever class you told the program to start with
  + execution context includes:
    - local variable names that are in scope
    - parameter names(act just like local variables)
* Strings
  + reference type
  + any changes are really just creating new strings
* Arrays
  + fixed length/size
  + zero based indexing
  + empty arrays can be created with the new keyword
    - will be filled with null
  + literal array syntax can be used to create arrays from literal values:

int[] myArray = {1,2,3};

* Classes/objects
  + fundamental units of abstraction
  + physical analogy
    - classes are liek factories
    - contain blueprint for an object
    - factory itself may have some capabilities(class members and lass methods)
      * relate to the abstraction itself
  + object's design must reflect its purpose
  + collection of named fields that represent information about the object
    - current value of the fields reflect the "state" of the object
    - which fields to use will depend on what it is does and how it interacts
* encapsulation(interfaces, getters and setters that validate, keeping fields private)
  + don't expose internal state of object directly
  + protects object fields from being put into an inconsistent or erroneous state
  + avoids situation in which external code is dependent on this specific implementation
  + separate exposed behavior from internal behavior
    - exposed: procedures functions other objects/code expected to interact with
    - internal: procedures/functions defined only for use by methods that are part of the class\

* Interfaces in Java
  + like classes, should go in their own .java file
  + should have same name as file
  + list of method signatures
    - implementing classes must declare these methods as public
  + classes specify which interfaces they implement
* Advantages of encapsulation
  + provides different implementation of the same behavior
  + avoids bugs
* merits of immutability
  + can be shared as a pluarlity, you know it won't change

* polymorphism(constructor overloading, method overloading, class polymorphism, subinterfacing, subclassing)
  + principle of providing access to an abstraciton or method in many forms
  + idea is that different forms fit different contexts
  + goal of the underlying funcitonality is the same

* method overloading
  + regular methods can also be overloaded
  + parameter list must somehow be different(number, type)
  + provide access to constructor/method in a more context specific way
* Is-A casting
  + a subclass can be casted into its parent
    - Ex:

Song s = new Song();

Media m = (Media) s;

* BUT, a subclass cannot be casted into another subclass that shares the same parent class
* instance fields
  + subclass has direct access to public and protected fields of its parent class, but not private
* subclasses can overload methods of its superclasses
  + still, something must be different in the parameter list
* different from overriding, where the subclass defines the exact same method and overrides the parent's version of the method

* virtual methods
  + all methods are virtual, meaning the method defined at most specific class level is executed NO MATTER WHAT
  + drawback: you want to override the method, but you also want to call the method from the parent's point of view
    - to fix this, you call "super" keyword to refer to the version of the method at the superclass level
      * this essentially turns off the virtual keyword

* Composition And aggregation
  + composition: individual parts that make up the whole are "owned" solely by the whole
    - they don't otherwise have a reason for being
    - encapsulated objects created internally, usually within the constructor
    - no setters and often no getters
    - encapsulated objects do not make sense outside of the abstraction
    - not shared with other abstractions
  + aggregation
    - the individual parts may also exist on their own outside of the whole
    - encapsulated objects are provided externally as parameters to the constructor
    - getters and setters for these components often provided
    - encapsulated objects may be independently reference outside of the aggregating object
      * including possibly as part of another aggregation
* Delegation
  + claiming an "is-a" relationship with an interface but relying on an internal object to actually do the work
  + can occur in either aggregation or composition, although more common in composition

* Iterator Design Pattern
  + provide a separate object which represents a traversal of a collection
  + the collection object is iterable
    - in order to be classified as iterable, it must provide a method to construct an iterator object
* Exception Handling
  + unchecked vs. checked
    - checked = follows catch or specify rule = if exception could happen, you either have to deal with it or specify that the exception could occur in a certain method
  + two Parts:
    - throwing
      * signaling that an exception has occurred, also known as raising an exception
    - catching
      * handling the exceltion
  + benefits
    - promotes consistency, modularity, separation of concerns, extensibility
    - provides an abstraction hierarchy for error information
      * information about when and why the error occurred is encapsulated into an object
    - improves code organization
      * separates error handling code from normal code
  + general principle: be specific
    - use an existing exception type
    - crate your own exception types
      * subclass either RuntimeException or Exception
  + general principle: catch late
    - exceptions should rise to level where application has enough context to deal with them effectively
    - catching exception just because you can is not always the right thing to do
    - pass the buck unless response to this exception under all or nearly all circumstances is well-understood at this point
  + general principle: throw early
    - validate values as early as possible
    - rather than waiting for exception generated by invalid values sent to other code
    - helpful for nullpointerexceptions
* Factory design pattern
  + when direct construction of an object is co plicated or harmful
* Observer/observable
* Decorator
  + add behavior to an object but without using subclassing
  + you decorate interface by extending it, and then create a new class that inherits this interface
    - use delegation to delegate all old stuff
* Model/View and MVC
  + ways to think about software as a whole
* Java threads and deadlock and join stuff