Java String Class

- **String is a class**
  - Do not need `new` to create String
    ```java
    String msg = "hello";
    ```

- **Can join strings (concatenate) with +**
  - ```java
    String mail = "John says " + msg;
    ```

- **Most common String methods:**
  - ```java
    int length(); // get number of chars in it
    String substring(int start, int stop); // substring gets part of string
    int indexOf(String key); // finds loc of key
    char charAt(int index); // get a single char
    ```
Why Inheritance?

- **Add new shapes easily without changing much code**
  - `Shape s1 = new Circle();`
  - `Shape s2 = new Square();`

- **Interface/abstract base class:**
  - `interface or abstraction`
  - `Function called at runtime`

- **concrete subclass**
  - `All abstract functions implemented`
  - `Later we'll override`

- **“is-a” view of inheritance**
  - `Substitutable for, usable in all cases as-a`

User’s eye view: think and program with *abstractions*, realize different, but conforming *implementations*,

don’t commit to something concrete until as late as possible
Benefits of Inheritance, Interfaces

- Consider new algorithm for determining unique word count
  ```java
  public static void test(UniqueCounter uc, String[] list){
    double start = System.currentTimeMillis();
    int count = uc.uniqueCount(list);
    double end = System.currentTimeMillis();
    System.out.println(count + " unique words");
    System.out.println((end-start)/1000 + " seconds");
  }
  ```

- Why can we pass different kinds of objects to `test`?
  - Why is this an advantage?
  - Inheritance and late/dynamic binding
Example of Inheritance

- What is behavior of a shape?

```java
void doShape(Shape s) {
    System.out.println(s.area());
    System.out.println(s.perimeter());
    s.expand(2.0);
    System.out.println(s.area());
    System.out.println(s.perimeter());
}

Shape s1 = new Circle(2);
Shape s2 = new Square(4);
Shape s3 = new Rectangle(2,5);
doShape(s1); doShape(s2); doShape(s3);
```
Inheritance (language independent)

- **First view: exploit common interfaces in programming**
  - Iterators in Java or C++
  - Implementation varies while interface stays the same

- **Second view: share code, factor code into parent class**
  - Code in parent class shared by subclasses
  - Subclasses can *override* inherited method
    - Subclasses can override and call

- **Polymorphism/late(runtime) binding (compare: static)**
  - Function actually called determined when program runs, *not* when program is compiled
What can an **Object** do (to itself)?

- [http://java.sun.com/j2se/1.5.0/docs/api/](http://java.sun.com/j2se/1.5.0/docs/api/)
  - Look at java.lang.Object

- **toString()**
  - Used to print (System.out.println) an object, overriding toString() can result in 'useful' information being printed, also used in String concatenation: String s = x + y;
  - Default is basically a pointer-value

- **equals()**
  - Determines if guts of two objects are the same, must override, e.g., for using a.indexOf(o) in ArrayList a
  - Default is ==, pointer equality

- **hashCode()**
  - Hashes object (guts) to value for efficient lookup
Objects and Values

- **Primitive variables are boxes**
  - think memory location with value
- **Object variables are labels that are put on boxes**

```java
String s = new String("genome");
String t = new String("genome");
if (s == t) {they label the same box}
if (s.equals(t)) {contents of boxes the same}
```

What's in the boxes? "genome" is in the boxes
Objects, Values, Classes

- **For primitive types: int, char, double, boolean**
  - Variables have names and are themselves boxes (metaphorically)
  - Two int variables assigned 17 are equal with ==

- **For object types: String, Sequence, others**
  - Variables have names and are labels for boxes
  - If no box assigned, created, then label applied to `null`
  - Can assign label to existing box (via another label)
  - Can create new box using `new`

- **Object types are references or pointers or labels to storage**
Java Arrays

- **Fixed size, once created**
  - Can hold primitive types
  - Can hold objects (references)

- **Example: Creating an array of doubles**
  
  ```java
  double[] times;
  times = new double[30]; // or could combine w prev
  ```

- **Example: Creating an array of DLicenses**
  
  ```java
  DLicense[] dls;
  dls = new DLicense[50]; // create array (or combine)
  for (int k; k < dls.length; k++) {
      dls[k] = new DLicense(); // create objects in dls
  }
  ```
Java Arrays

- Can also create arrays by specifying initial values
  - Avoids need for `new`
  - Avoids need to count the number of values

- Example: Creating an array of `ints`
  ```java
  int[] counts = { 3, 12, 0, 8, 10};
  ```
  - Use `counts.length` to get size of array

- Example: Creating an array of `Strings`
  ```java
  String[] aHotel = {"Hilton", "Swans", "Astoria"};
  String[] bHotel = {"Kwik8", "SleepyT", "TuckUIn"};
  String[] cHotel = {"DiveX", "RRXing", "Swampys"};
  ```

- Example: Creating an array of arrays (matrix)
  ```java
  String[][] hotelChoice = {aHotel, bHotel, cHotel};
  ```
For-Each Loop

- For Arrays (and Collections) May Use Special Loop
  - Syntax
    ```
    for (Type name : expression){
        body of loop
    }
    ```
  - *Type* is the type of object returned for use in loop
  - *name* is of variable that take on value for use in loop
  - *expression* is an array or collection

- Example: (dl is a DLicense object and dls an array of dl)
  ```
  for (DLicense dl : dls) {
      System.out.println(dl.getName() + " "
      + dl.getNum());
  }
  ```
  - But *cannot* change entries! (effectively dealing with copy)
Java ArrayList Class

- **Flexible Arrays**
  - Grows in size as needed!
  - Many different methods to improved array processing

- **Create with:**
  ```java
  ArrayList vect = new ArrayList();
  ```

- **Uses:** (assume dl, sl, are DLlicense objects)
  ```java
  vect.add(dl); // add to "end"
  vect.add(k, dl); // insert at position k (shifts!)
  sl = (DLlicense) vect.get(m); // retrieve from
  // position m - note cast to DLlicense
  ```

- **Note that [ ] brackets don’t work !!!**
  - Also see: remove(), indexOf(), toArray(), contains(), size(), ... Look them up!
Java ArrayList Class

- **Generic forms**
  - Previous example stored items as Objects
  - On retrieving, needed to cast back to original class

- **Create with:**
  
  ```java
  ArrayList<DLicense> vect = new ArrayList<Dlicense>();
  ```

- **Uses:** (assume dl, sl, are DLlicense objects)
  ```java
  vect.add(dl); // add to "end"
  vect.add(k, dl); // insert at position k (shifts!)
  sl = vect.get(m); // get at position m: no cast needed
  for (DLlicense cl : vect) {
    System.out.println("Number is " + cl.getNum());
  }
  ```