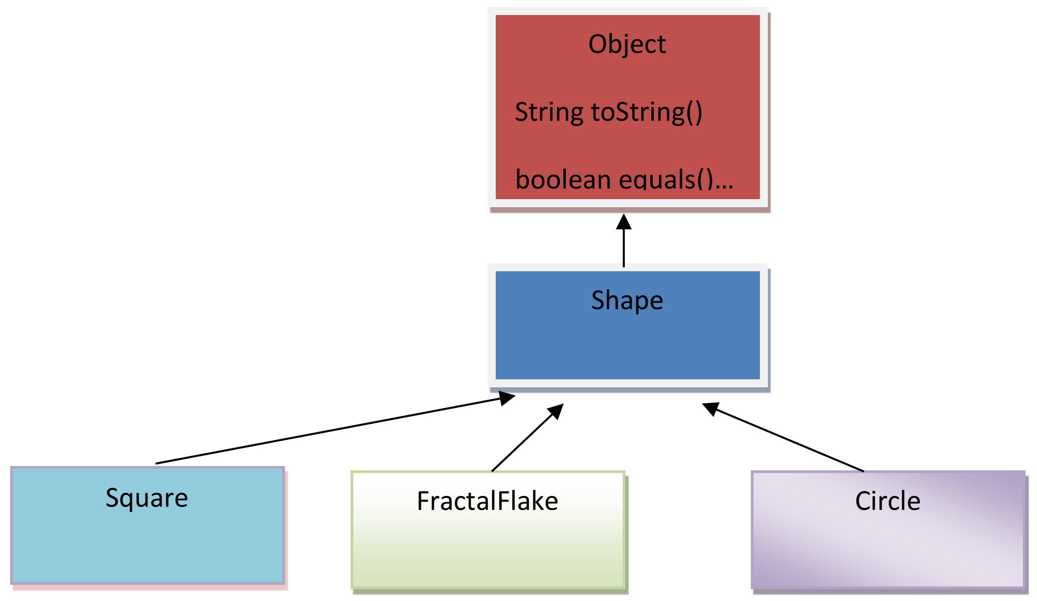
Lab5 part I: Inheritance

Inheritance, Overriding, & Subclasses*

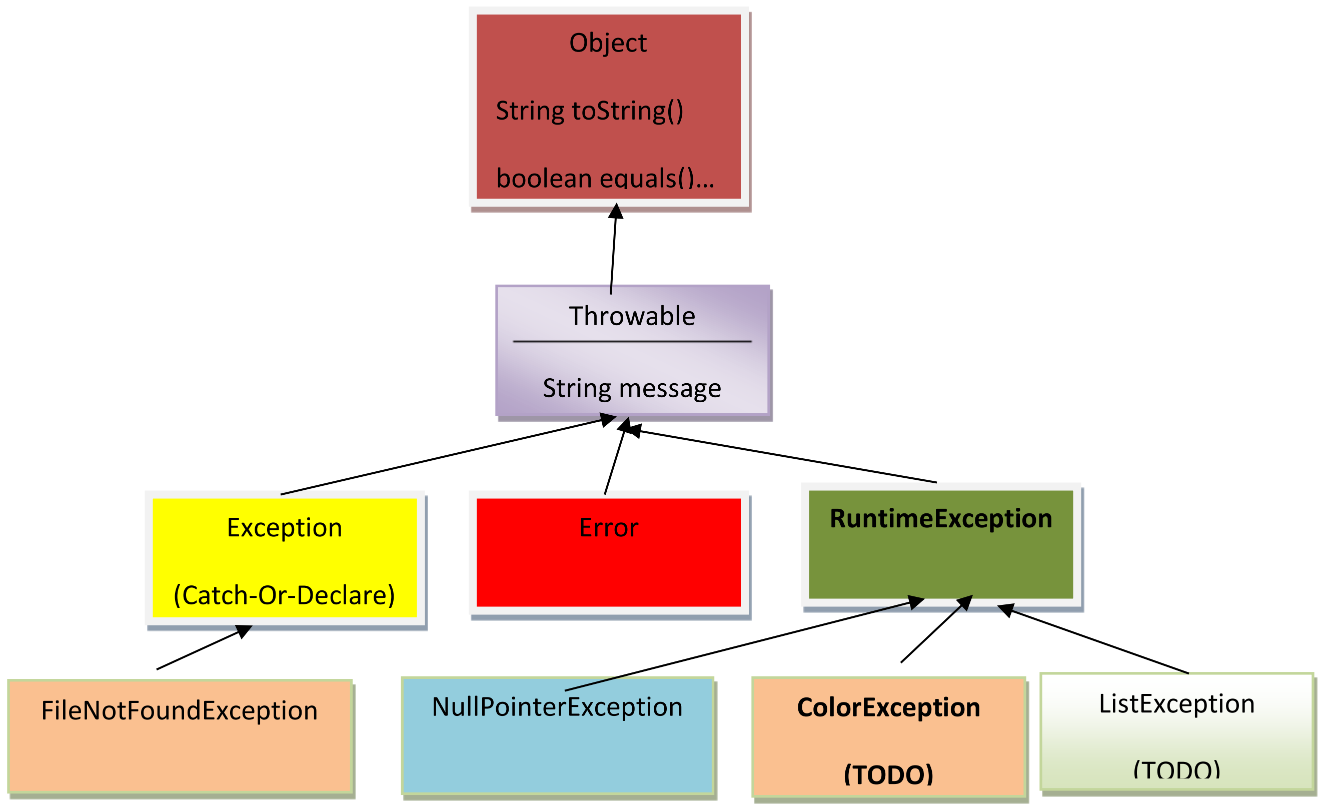
CSSSKL162: Programming Methodology Skills

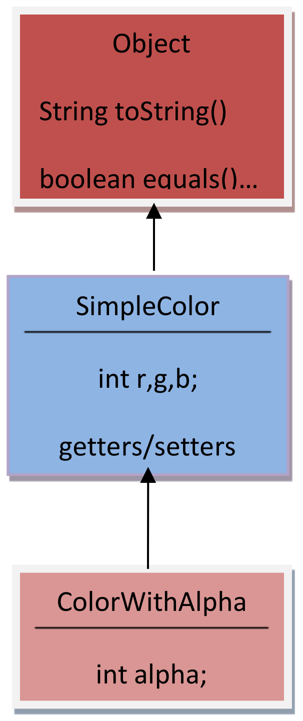
Summary

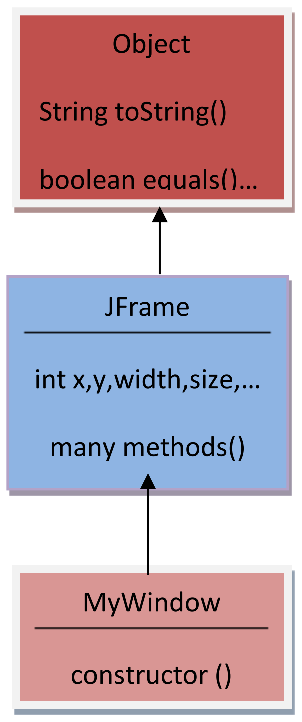
The purpose of this lab is to practice building new classes by inheriting methods and data from existing classes. We'll start with a toy example and then move on to consider our next homework assignment.



This lab is created by Rob Nash; modification and additions by A. Retik


Object
String toString()
boolean equals()…
Throwable
String message
Exception
(Catch-Or-Declare)
Error
RuntimeException
FileNotFoundException
NullPointerException
ColorException
(TODO)
ListException
(TODO)


Object
String toString()
boolean equals()…
SimpleColor
int r,g,b;
getters/setters
ColorWithAlpha
int alpha;


Object
String toString()
boolean equals()…
JFrame
int x,y,width,size,…
many methods()
MyWindow
constructor ()

Introduction to Inheritance (the "Is A" Relationship)

Would it be easier to reach great heights by standing on our own toes, or by standing on the shoulders of giants? Many would suggest we leverage (or resuse) the already-existing giants to provide a nice vantage point to aid us in our lofty quest. We can simply climb on-top of this existing structure, adding only a little height to the existing stature of the giant, but yet appear to be a huge monolith. Inheritance is a software engineering technique used to quickly produce new classes by building on top of existing classes. Using inheritance, we could avoid lots of the cutting and pasting we've done in this class, as this is effectively what inheritance reduces to: copy all public methods and data to the subclass. Inheritance borrows from Mendelian genetics; in his experiments, Gregor sought to reliably reproduce characteristics of a parent plant in its offspring. In the same way, a class that inherits from another gets a copy of all public methods and data defined in the parent class. (Note that private data would be inaccessible and so is not copied; similarly, no constructor is ever inherited). If a superclass Shape defines a getX() and getY() as public, then the subclass Square will also have a (free) copy of these methods as well. Hence, a Square ***is a*** Shape, and can do everything a Shape can do (like getX() or getY()) and then some. In fact, a Square will define additional methods not found in Shape, such as getSideLength(); similarly, a Circle will define a radius and getRadius() method, which will be absent from the Shape superclass (or Square sibling class). Similarities such as x,y, and Color will go in the superclass, whereas distinctions such as radius, getSideLength(), and getRadius() will go in specific subclasses. Lets start with a simple inheritance example using only six lines of code.

ColorException Extends RuntimeException

As programs execute, they may encounter exceptional situations in which they could either try to recover or simply terminate due to the situation at hand. We're going to make a specific type of exception for our SimpleColor (and ColorWithAlpha) class, in the event the user tries to set any channel to anything outside the range [0,255], inclusive. Note that this class will be only a handful of lines, and demonstrates how inheritance can quickly produce fully functioning classes in a short amount of time.

1. Create a new project for this lab
2. Define a new class called ColorException
    a. This class should include "extends RuntimeException"
3. Build a default, no-arg constructor that contains only one line
    a. super("An error occurred in Color");
4. Overload your constructor with a second constructor that takes a String "msg" as input. Then, the only line of code in this function will be:
    a. super(msg);
5. In your old SimpleColor class (and in your new ColorWithAlpha class below), in the setter functions (setRed, setGreen, setBlue, etc.), throw a new ColorException if the user tries to set a value outside of [0,255], inclusive.
6. To test your exception, make a main in ColorException and include the line:
    a. throw new ColorException("A test in main");

ColorWithAlpha Extends SimpleColor

Our classes shouldn't start from scratch when we design them for use in our applications, and we have an opportunity to realize the often-elusive goal of code reuse here in this lab. If you still have the SimpleColor class from a previous lab, you should locate that now. The purpose of this lab is to get you started with inheritance by building new classes from existing classes. We'll build a class that can represent an RGB value and an Alpha value, used in determining transparency. Define this new class using the following format: "public class ColorWithAlpha extends SimpleColor {", and make sure SimpleColor.java is in the same directory as your new ColorWithAlpha.java file. Even though the class is empty, it should compile just fine, and in fact, contains all the methods and data defined in the SimpleColor superclass.

(1) Download the ColorDriver.java class.
(2) Find your old SimpleColor class, or build a new SimpleColor superclass now.
(3) Build a new subclass called ColorWithAlpha, and extend SimpleColor.
(4) Add a new int called "alpha" for use in determining the color's alpha channel
    a. This too will be clamped from 0-255, representing 8 bits of storage.
(5) Define getters and setters for the new alpha channel.
(6) Build one constructor that takes only an alpha value
    a. This should initialize RGB to 0
(7) Build a second (overloaded) constructor that takes 4 values (RGBA) and calls the superclass constructor to initialize the RGB values.
    a. The first line of your constructor will be "super(r,g,b);"
        i. What does this line accomplish? What function are we calling?
(8) Build a third (overloaded) copy constructor that takes an existing ColorWithAlpha object as input
(9) Override the toString() function to print out your "(R,G,B,A)" values as we do below.
    a. What does the following call to super do?

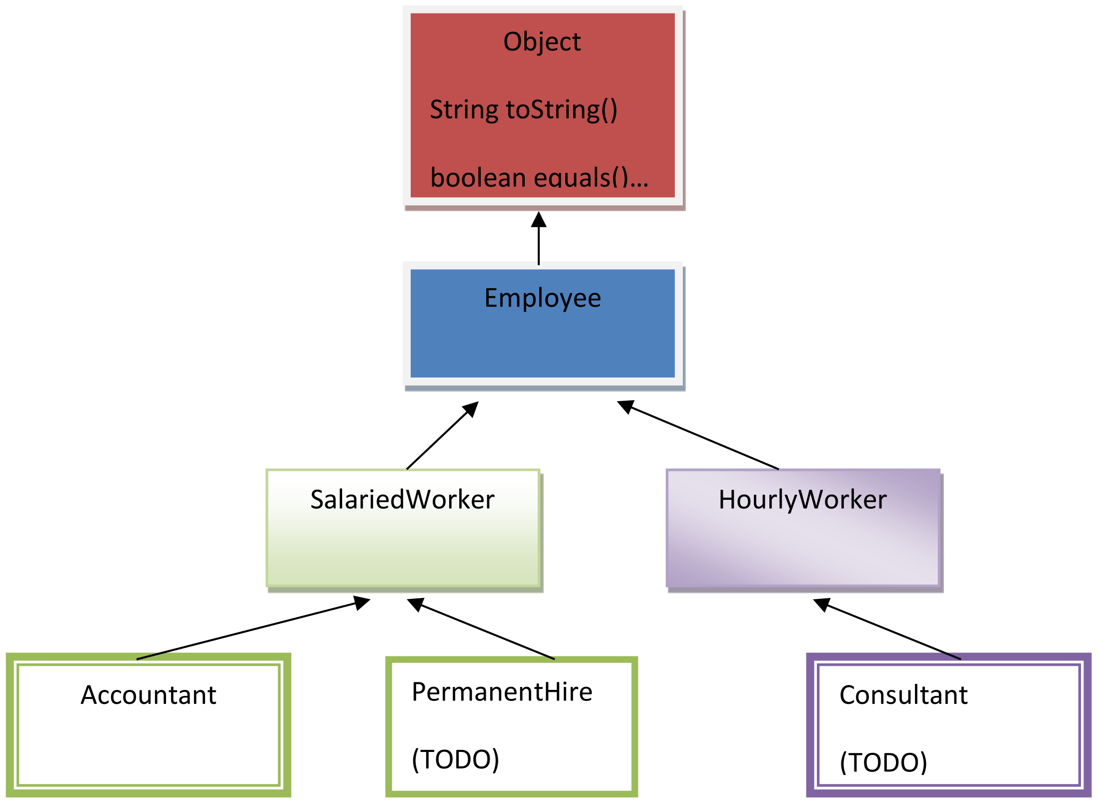
```
public String toString() {
            return super.toString() + ", alpha:"+ alpha;
}
```

(10)Override the equals(Object o) function to determine if two ColorsWithAlpha are equal
    a. Use "==" for primitives such as your alpha amount, and use ".equals()" with objects such as SimpleColors.
(11) Run the driver to test your new ColorWithAlpha subclass.

Window Extends JFrame

For this demonstration, view the Window.java file example to learn how to produce your own GUI by extending the existing JFrame class. This is defined pictorially above in the example inheritance hierarchies. We will use this feature in future labs.

The Employee Inheritance Hierarchy



Download the following classes for this lab: {Employee, HourlyWorker, SalariedWorker, Accountant, Driver}. These classes represent an inheritance hierarchy with the Employee class acting as the Parent class for both HourlyWorker and SalariedWorker. Read the code in each class, and run the corresponding driver to see the output from each employee. In this warm-up exercise, I have created an Accountant class, which inherits from SalariedWorker. SalariedWorker, in turn, inherits from Employee, which is the topmost class in our hierarchy. In fact, Employee is so generic that you cannot even create Employee objects (it's an abstract class), so we must define subclasses that override the calculateWeeklyPay() method in order to create objects we can use. The Accountant class is rather short, as most of the class's functionality is inherited from the SalariedWorker class. Your job is to **create two classes here**: one that inherits from SalariedWorker (see Accountant for such an example), and one that inherits from HourlyWorker. To start things off, be sure your class header indicates who you are inheriting from, such as: "public class Consultant extends HourlyWorker {".

PermanentHire Class Extends SalariedWorker

In this section, we'll experiment with an inheritance hierarchy that has more levels. We seek to build a Permanent Hire class that will model a full-time position as a specific kind of SalariedWorker. That is to say, a PermanentHire "is a" SalariedWorker, and a SalariedWorker "is an" Employee (and an Employee "is an" Object).

(1) Download the classes outlined in the first inheritance hierarchy on page one. (Employee, SalariedWorker, HourlyWorker, Accountant)

(2) Look at Accountant to see an example of inheritance already completed for you.

(3) Create a class called PermanentHire, and this should inherit from SalariedWorker.
   a. public class PermanentHire extends SalariedWorker

(4) Once you've built PermanentHire, then you can create constructors for your class
   a. See the Accountant class for examples of constructors

(5) Try overriding the calculateWeeklyPay() method in your subclass so it does something different than the original version.
   a. Consider making PermanentHire an employee that has a base salary plus a flat bonus every month, which would require a new (overridden) calculateWeeklyPay() method.

(6) Using the EmployeeDriver, build a few PermanentHires with different instance data and add them to the ArrayList of Employees.
   a. Be sure to see your new PermanentHires in the console output.

(7) Using the EmployeeDriver, build a ColorWithAlpha and try to add it to the ArrayList of Employees. What error do you encounter? What does this mean?

Consultant Class Extends HourlyWorker

In this section, we'll produce a class (based on existing classes) for use in modeling a part-time position as a specific kind of HourlyWorker. Your Consultant "is a" HourlyWorker, which "is an" Employee.

(8) Next, create a class called Consultant, and this should inherit from HourlyWorker
   a. public class Consultant extends HourlyWorker

(9) Create constructors for Consultant just like you did in PermanentHire, and as you can see in Accountant.

(10) Override the calculateWeeklyPay() method so that it calculates the hours worked (20 for part-time) by the hourly pay ($3.00/hr) to produce a result.

(11) Using the EmployeeDriver, build a few Consultants with different instance data and add them to the ArrayList of Employees.
   a. Be sure to see your new Consultants in the console output.

(12) Using the EmployeeDriver, build a ColorException and try to add it to the ArrayList of Employees. What error do you encounter? What if you *wanted* the ArrayList to store both Employees and ColorException – how would this change the way ArrayList is declared?