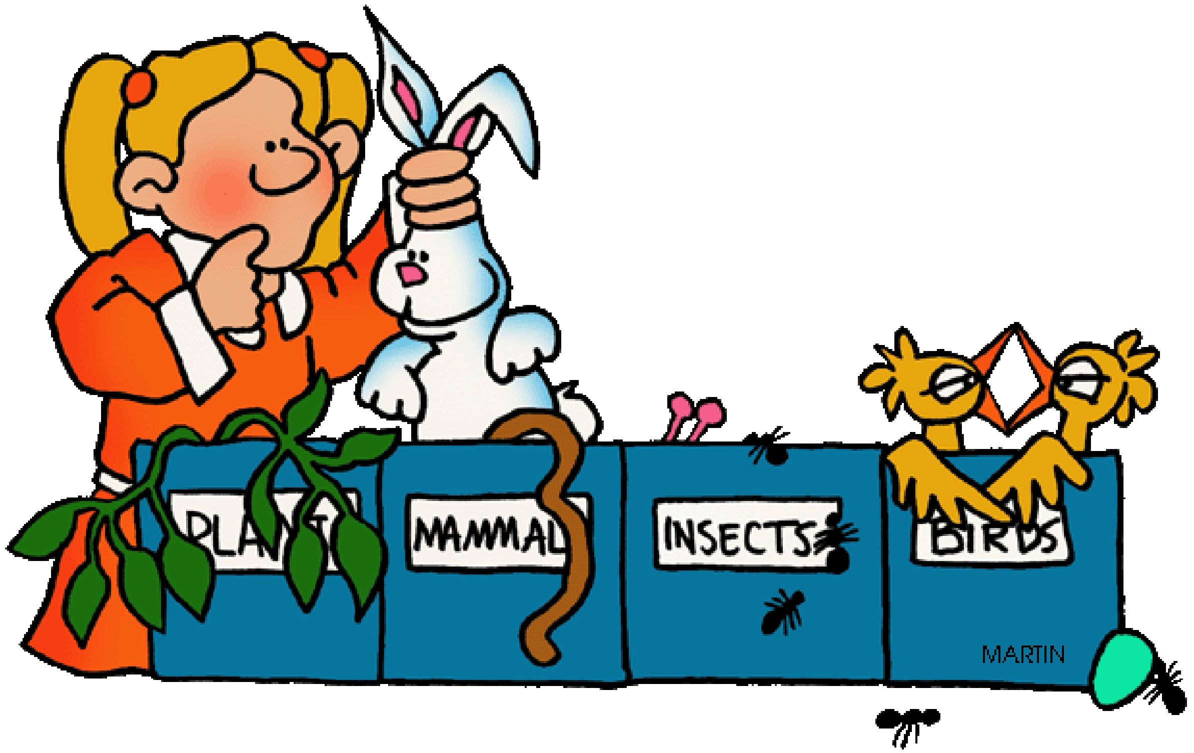
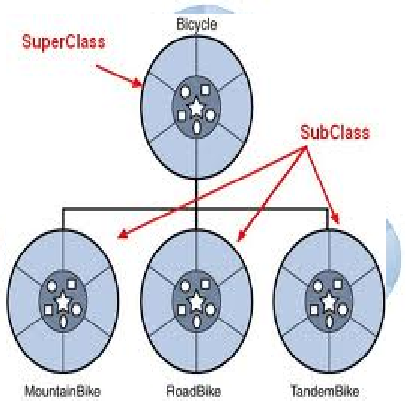
# Classification or Categorization of Things



*Why?*

# 3 Advanced Programming

## 3.1 Inheritance and Subclasses



Inheritance is when a new class is created from an existing class by absorbing (or *inheriting*) its fields and methods **and** adding to these, other capabilities (new fields and/or methods) that the new class requires to distinguish itself from its *ancestor*.

The old, previously defined class is called the **superclass** and the new class which inherits the capabilities of the superclass is called the **subclass**.

The **subclass** thus *has all the fields and methods of the superclass* **plus** any new ones it defines. As such the subclass is more specific and represents a smaller group of objects than the superclass.

Every object of a subclass is also an object of the superclass.

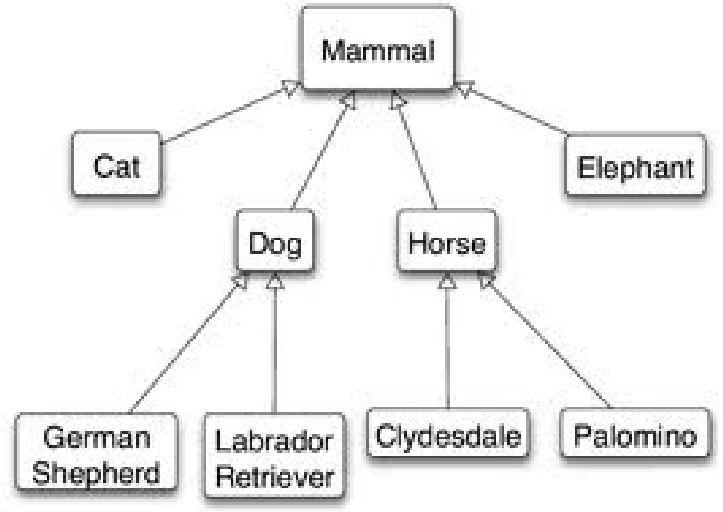
The opposite is NOT true.

## 3 Advanced Programming
### 3.1 Inheritance and Subclasses

Inheritance forms a tree-like hierarchical structure. For example:

*Shape is the superclass.*
*2D and 3D shape are two subclasses of the superclass shape.*

*Circle, Square and Triangle are subclasses of the 2D shape class.*



*A Triangle is a 2D shape and also a Shape. However a Shape is NOT a Triangle.*

If A is a B

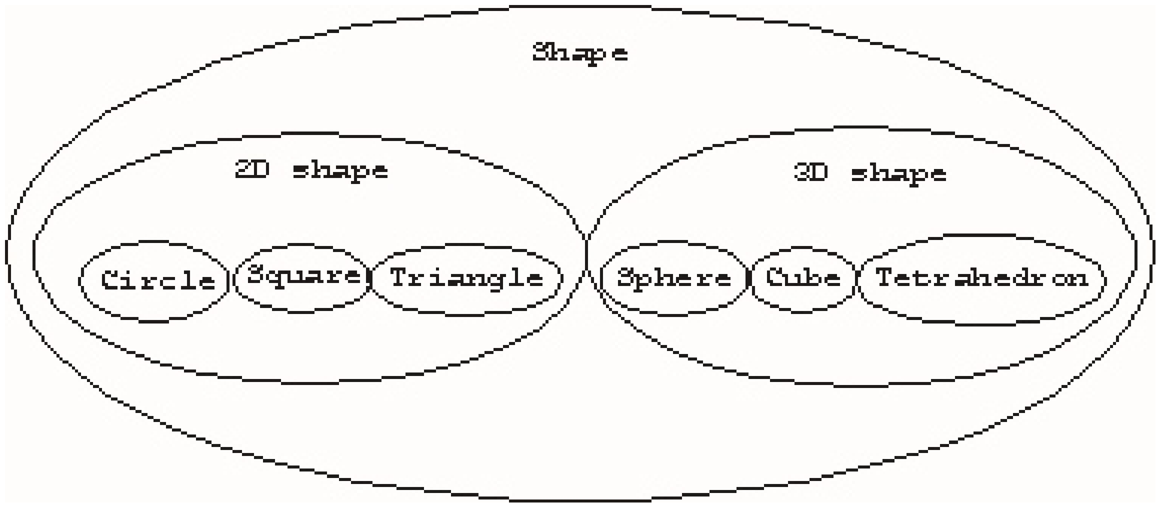
- then B is the super class and A is the sub class.
- all fields and methods of B are A's as well (except if B declared them private)
- it works one-way. B is not an A.

All classes in Java are subclasses of the one true super class Object.

Inheritance can be thought of as a tree structure .

## 3 Advanced Programming
### 3.1 Inheritance and Subclasses

### Shape Hierarchy Test Example

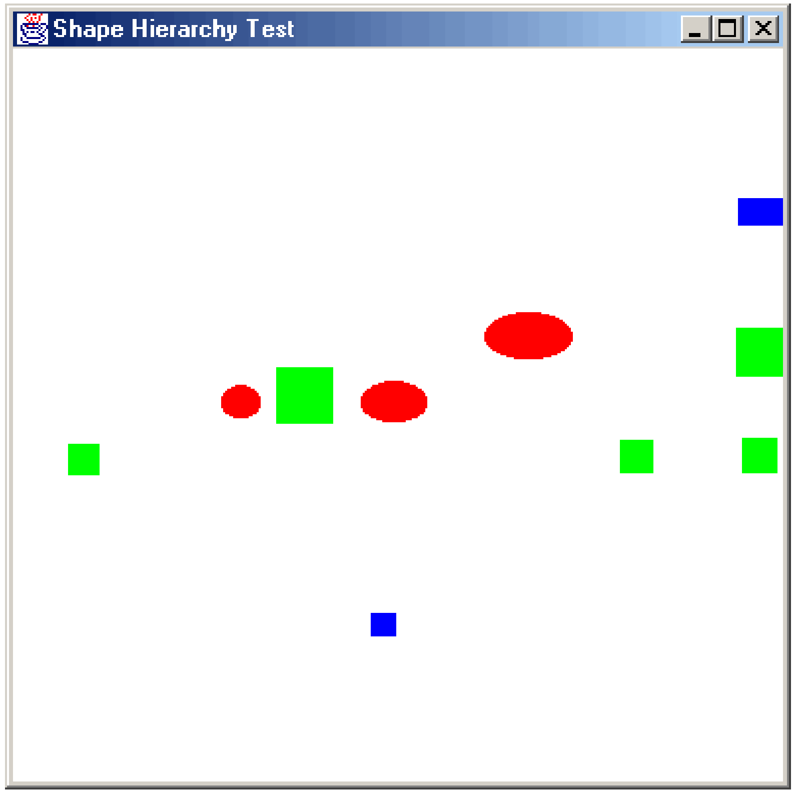


Notes:
- subclass **extends** superclass
- subclass constructor MUST call super constructor
- **OverRidding**(in contrast to OverLoading) means giving the method in the sub class the exact same name, return type and parameter list as in the super class (eg toString() and clone())
- **final** prevents overridding in a sub class.

## 3.1.1 A Geometric Example

In this example we are going to create a program which will display a variety of shapes (rectangles, squares and ovals) on the screen and move them about, bouncing them off the sides of the window. (In addition to this, the ovals change size as they move.)

It would look something like:



## 3.1.1 A Geometric Example

This will require 3 classes:
Rectangle,
Oval and
Square.

These classes however *have several things in common* . They all have a position, ie. an *x* , *y* for their top left corner. They all have a *colour*. They also have *dx* and *dy* values to indicate how much their position changes (so they can be moved). They all have to be *drawn*, *erased* and *moved*.

Because they have so much in common, it *may* make sense to put all the common stuff in a single class and make the Rectangle, Square and Oval class *inherit* these fields and methods from this common class.

Let’s call this class Shape.

*Think about this for a moment. A Square is a Rectangle. A Rectangle and an Oval are Shapes.*

## 3.1.1 A Geometric Example

*A Square is a Rectangle. A Rectangle and an Oval are Shapes.*

Shape is the superclass,
Rectangle is a subclass of Shape and
Square is a subclass of Rectangle.
Oval is a subclass of Shape.

Every Square object is also a Rectangle and a Shape.

Every Oval object is also a Shape object.

This means that whatever fields and methods Shape has, Rectangle and Oval have as well (in addition to its own fields and methods.)

And whatever fields and methods Rectangle has, so to does Square.

## 3.1.1 A Geometric Example

*A Square is a Rectangle. A Rectangle and an Oval are Shapes.*

```
abstract class Shape implements Colorable
{
          protected int x, y;
          protected int dx, dy;
          protected Color color;

          abstract public void draw (Graphics g);
          abstract public void erase (Graphics g);

          public Shape (int x, int y, int dx, int dy)
          {
                    this.x = x;
                    this.y = y;
                    this.dx = dx;
                    this.dy = dy;
                    color = Color.black;
          }


          public void slide (Graphics g)
          {
                    erase (g);
                    if (x <= 4 || x >= g.getClipBounds().width - 25)
                              dx = -dx;
                    if (y <= 25|| y >= g.getClipBounds().height- 25)
                              dy = -dy;
                    x += dx;
                    y += dy;
                    draw (g);
          }

          public void setColor (Color c)
          {
                    color = c;
          }

          public Color getColor ()
          {
                    return color;
          }

}
```

**Implements -** is similar to 'extends'; it is used to create a subclass - a subsclass of multiple classes; but these classes would be 'abstract' classes ***that contain no*** actual useable methods.

**Extends** - is a java keyword that declares that the defined class is a subclass of a named 'super' class.

## 3.1.1 A Geometric Example

*A Square is a Rectangle. A Rectangle and an Oval are Shapes.*

```
class Rectangle extends Shape
{
            protected int width, height;

            public Rectangle (int x, int y, int dx, int dy, int w, int h)
            {
                        super (x, y, dx, dy);
                        height = h;
                        width = w;
                        setColor (Color.blue);
            }

            public void draw (Graphics g)
            {
                        g.setColor (color);
                        g.fillRect (x, y, width, height);
            }


            public void erase (Graphics g)
            {
                        g.clearRect (x, y, width, height);
            }
}
```

**Extends** - is a java keyword that declares that the defined class is a subclass of a named 'super' class.

**A subclass must always call the superclass constructor.** This is done explicitly by calling `super()` or if this is not done the computer will automatically call it with no parameters. (This would cause a compile error if no such constructor exists!)

**An overridden method in a subclass** has the same name, return type and parameters as in the superclass.

```
class Square extends Rectangle
{
            public Square (int x, int y, int dx, int dy, int s)
            {
                        super (x, y, dx, dy, s, s);
                        setColor (Color.green);
            }
}
```

## 3.1.1 A Geometric Example

*A Square is a Rectangle. A Rectangle and an Oval are Shapes.*

```
class Oval extends Shape implements Sizeable
{
        protected int width, height;

        public Oval (int x, int y, int dx, int dy,int w, int h)
        {
                super (x, y, dx, dy);
                height = h;
                width = w;
                setColor (Color.red);
        }

        public void draw (Graphics g)
        {
                g.setColor (color);
                g.fillOval (x, y, width, height);
        }

        public void erase (Graphics g)
        {
                g.clearRect (x, y, width, height);
        }

        public void setWidth (Graphics g, int w)
        {
                erase (g);
                width = w;
                draw (g);
        }

        public int getWidth ()
        {
                return width;
        }

        public void setHeight (Graphics g, int h)
        {
                erase (g);
                height = h;
                draw (g);
        }

        public int getHeight ()
        {
                return height;
        }
}
```

**Implements -** is similar to 'extends'; it is used to create an subclass - a subsclass of multiple classes; but these would be 'abstract' classes that contain no actual useable methods.

**A subclass must always call the superclass constructor.** This is done explicitly by calling `super()` or if this is not done the computer will automatically call it with no parameters. (This would cause a compile error if no such constructor exists!)

**An overridden method in a subclass has the same name, return type and parameters as in the superclass.**

## 3.1.2 Overriding Methods and Dynamic Method Lookup

Look more closely at the draw method. There are four such methods: one for each class.

This is not *overloading*. Overloading means the parameter list is different: the computer knows which method to use by examiningthe parameter list.

Won't the computer get confused by all the duplicate method names? For example since a Rectangle object inherits all the methods from Shape won't it have two draw methods (one from shape and one from itself)? Yes that is correct. But no; the computer doesn't get confused.

Note class C has one method, but the "latest" or most current is the one it inherits from B.

## 3.1.2 Overriding Methods and Dynamic Method Lookup

Here is a very simple example to illustrate overriding and dynamic method lookup;

```
class A
{
        public void method (Console c)
        {
                c.println ("A method."); }
        }

class B extends A
        {
        public void method (Console c)
        {
                c.println ("B method."); }
        }

class C extends B
        {
        }
```

Here are 3 simple classes. Each inherits “method” from the previous. So a B object has two methods. BUT the method it inherited from A is overridden and cannot be accessed from outside the class.

Note class C has one method, but the “latest” or most current is the one it inherits from B.

## 3.1.2 Overriding Methods and DM-Lookup

Here is a very simple example to illustrate overriding and dynamic method lookup;

```
class A
{
        public void method (Console c)
        {
                c.println ("A method."); }
        }

class B extends A
        {
        public void method (Console c)
        {
                c.println ("B method."); }
        }

class C extends B
        {
        }
```

Here are 3 simple classes. Each inherits “method” from the previous. So a B object has two methods. BUT the method it inherited from A is overridden and cannot be accessed from outside the class.

```
public class DynamicMethodLookupTest
{
static Console k;
public static void main (String [] args)
{
k = new Console ();
A a = new A ();
B b = new B ();
C c = new C ();
a.method (k); // prints "A method"
b.method (k); // prints "B method"
c.method (k); // prints "B method"
A a1, a2, a3;
a1 = a;
a2 = b;
a3 = c;
a1.method (k); // prints "A method"
a2.method (k); // prints "B method"
a3.method (k); // prints "B method"
}
}
```

**Dynamic Method Lookup** is the situation in which the computer determines which method to use at run time.

Note class C has one method, but the “latest” or most current is the one it inherits from B.

## 3.1.2 Overriding Methods and Dynamic Method Lookup

**Dynamic Method Lookup** is the situation in which the computer determines which method to use at run time. Several things must be true in order for this to occur:

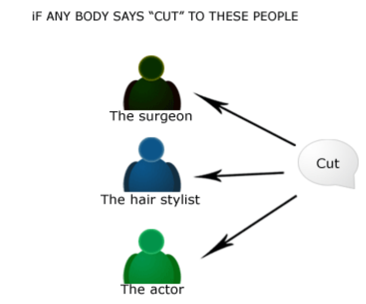
+ The reference must be a superclass type, but it must refer to a subclass.

+ the subclass must override the method in question.

If these two conditions are true, the computer chooses the subclass's method and not the superclass's (overridden) method.

In other languages such as C++ this concept is called **polymorphism**.

One final note: only instance methods can be overridden. Class methods can't be overridden.



The surgeon would begin to make an incision.

The hair stylist would begin to cut someone's hair.

The actor would abruptly stop acting out the current scene, awaiting directorial guidance.

## 3.1.3 Abstract Classes

An **abstract method** is a method which is not defined. That is, it has a name, return type and parameter list, but no body.

The method must be identified with the keyword abstract as in:

```
abstract public void draw ();
```

Any class which contains an abstract method must be declared as abstract itself; as in:

```
abstract class Shape
```

An **abstract** class is one which is declared as abstract. A non-abstract class is called **concrete.**

newable

**An abstract class cannot be instantiated .** That is, there can not be any object of an abstract class. For example you can't have:

```
Shape s = new Shape(...); // compile error
```

It is quite permissible to have references to an abstract class provided they refer to a concrete subclass of the abstract class.

```
Shape s = new Oval(...); // this is okay
```

Oval o = s;

## 3.1.3 Abstract Classes

Let’s review:

We could have done all this with 3 separate classes and no inheritance, but then each would need to have a lot of common fields and methods such as slide, setColor, etc.

*The complexity of your code would be less.... KISS... creating independent Classes for Rectangle, Square and Oval...*

*BUT the volume (or redundancy of your code) would be more...*

*KISS > Volume ??*
*perhaps....*

*BUT the capacity and capability of your code would also be less... using the KISS approach and independent Classes....*

## 3.1.3 Abstract Classes

Shape was created so that the common things were only done once and the 3 sub classes can all use those inherited methods.

In the main program we want to create a bunch of different shapes, but as far as possible we want to treat them all the same, thus we will not have separate reference types for each shape, rather we will use an array of Shape references to refer to the different types. Dynamic method lookup will ensure the correct methods are called.

Because we are going to use Shape references, the Shape class must include all the methods that we wish to access in the main program, especially draw. However we can't define a draw for a Shape. Therefore it must be abstract.

*But we can 'gain' the capability and capacity of now considering an Array of Shapes; whether they be Rectangles, Squares or Ovals... because these classes extend Shape; we have the ability to group 'Shapes' together because of their 'close' relationship.*

## 3.1.4 Interfaces

An interface is essentially an abstract class with only abstract methods.

(The words “abstract class” are replaced with “interface”.)

It cannot have any concrete methods.

Here are the two interfaces used in the above:

```
interface Colorable
{
        abstract public void setColor (Color color);
        abstract public Color getColor ();
}

interface Sizeable
{
        abstract public void setWidth(Graphics g, int w);
        abstract public int getWidth ();
        abstract public void setHeight(Graphics g, int h);
        abstract public int getHeight ();
}
```

***Why use an interface?***
Why not just include it as part of the abstract superclass?
It depends on each situation, but a subclass can be a subclass of only one superclass.

A class can inherit fields and methods from only one superclass, BUT it can implement one or more interfaces.

In this case, since lots of things might be Colorable or Sizeable, not just Shapes, it was decided to make these Interfaces and not simply part of the Shape base class. All the normal rules of inheritance apply to interfaces, therefore any class which implements an interface must override all the interface’s methods, otherwise it will become abstract itself.

You will notice that Shape, which implements Colorable and Oval which implements Sizeable override the methods of the interfaces.

## 3.1.5 AWT Class Hierarchy

As previously stated we aren't going to draw our shapes on a Console.

Instead we will use the Abstract Windowing Toolkit.

This is a complex group of interconnected classes, used to create a GUI (Graphical User Interface).

For now we will look at the Frame class and in the next chapter we will look at the Applet class. Here is a diagram of just some of the classes.

Object
(clone, toString)

Component
(getGraphics, action, getSize, setSize enable, disable)

Button Canvas Container TextComponent
(add, paint)

Panel Window TextField

Applet Dialog Frame
(init) (isResizeable) (setTitle)

You see that Frame is a sub, sub, sub, subclass of Object! Complex yes, but we can now make sense of it.

## 3.1.6 Composition of Classes

Inheritance is a means of grouping or combining classes. Another, perhaps simpler method of combining related classes is through composition. **Composition** of classes is when one class has a field which is an object of another class.

| | |
|---|---|
| Does A have a B? | composition: A has a field of type B |
| Is A a B? | inheritance: A has all of B's fields and methods. |

which?

- Line (two points)
- union employee
- cat (an animal)
- chess board (has pieces)
- ER doctor
- a Cube(a square with depth)

## 3.1.7 An Inheritance and Composition Example: A Frame (page 39)

In our example we are going to create a window (a Frame to be exact) and do our drawing on this, as opposed to using a Console.

Thus our main ShapeTest class **is a** Frame,
that is, it **inherits** from the Frame class.

But it also **has a** shape on it. To be more precise it has an array of n Shapes.

It also listens for window commands, so it **implements** the WindowListener **interface**.

## Dynamic Method Lookup

```
class One
{
        int A() {}
        void B (int, float) {}
        double C () {}
}
class Two extends One
{
        int A() {}
        void B (int, int) {}
        double C() {}
}
main pgm
        One x;
        Two y;
        One z, w;
        x = new One();
        y = new Two();
        z = x;
        w = y;
        x.A()
        x.B(...)
        x.C();
        y.A()
        y.B(...)
        y.C();
        z.A()
        z.B(...)
        z.C();
        w.A()
        w.B(...)
        w.C();
```

Which calls involve DML or Polymorphism?