



# Object Oriented

Classes, Objects, Inheritance,  
and Typing

By: Nicholas Merizzi  
January 2005

# Outline

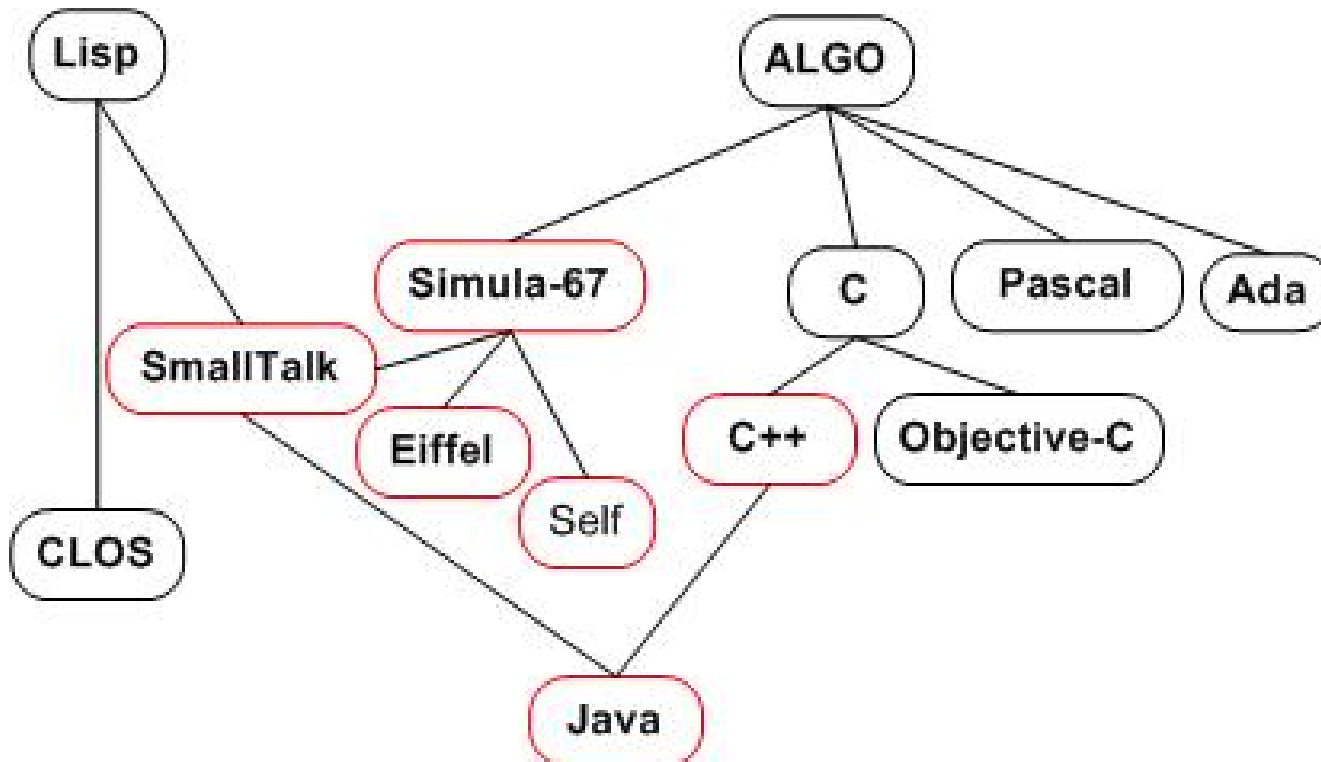
- **General Introduction**
  - Why the O.O. paradigm?
  - Classes & Objects
    - Properties, differences, Code Examples
- **Design Goals**
  - RDD, Coupling & Cohesion, Parnas's Principles
- **Inheritance**
  - Single/Multiple/Interfaces/Traits/Mixins
  - Types of Inheritance
- **Typing**
  - Subtyping, Supertyping
- **Prototype based O.O.**
  - Introduction
  - Differences
  - Examples
- **Pros & Cons of OO programming**
- **Conclusion & Discussion**



# General Introduction

# O.O Paradigm

- “The most important aspect of OOP is the creation of a universe of largely autonomous interacting agents”
- Object-Oriented Programming (OOP) in Simula I (1962)



# Classes & Objects

- **Classes:**
  - Is a template definition of the methods and variables in a particular kind of object.
  - The static attributes/features of the object (un-instantiated)
  - Generic form of an object
- **Objects:**
  - The dynamic attributes of a class
  - All Objects are instances of a class
  - Objects are what actually runs the software

# Classes & Objects (cont.)

- You create a class by typing out the syntax (code)
- You create an Object at runtime by instantiating it.
- An individual representation of a class is known as an instance
- Important to understand that two objects from the same class can have different contents.
- A class has two conflicting roles (will come back to later):
  - *generator of instances (must be complete)*
  - *unit of reuse (...yet must be small)*

# Properties

- **State-**

- The state of a component represents all the information held within it.
- The state is NOT static and can change over time.
- Some objects do have state.
- State is associated with an object.

- **Behavior-**

- The behavior of a component is the set of actions that it can perform (what it can do)
- For example on our OO polynomial class we had behaviors to divide, multiply, and print the object on the screen.
- Behavior is associated with the class, not with an object.

# Example

```
class polynomial {  
  
    private Vector coeff;  
    private Vector degree;  
  
    public polynomial() {}  
    public polynomial modPoly(polynomial Bx) {}  
    public boolean isZeroPoly() {}  
    public polynomial multPoly(polynomial px) {}  
    private void simplifyPoly() {}  
    public polynomial subPoly(polynomial px) {}  
  
}
```



# Properties of an Object

- The big 3:
  - Encapsulation
  - Inheritance
  - Polymorphism

# Encapsulation

- By definition it is the packaging of data with methods that operate on that data.
- **Encapsulation != Information Hiding**
  - Encapsulation facilitates, but does not guarantee, information hiding.
  - i.e. In Java you can have encapsulated data that is not hidden at all.

# Encapsulation (cont.)

- Encapsulation is a language facility, whereas information hiding is a design principle.
- Encapsulation can occur in non-OO environments.
- David Parnas first introduced the concept of information hiding in 1972.
  - Parnas stressed that the following should be hidden "difficult design decisions or design decisions which are likely to change."

# Encapsulation (cont.) - *Visibility*

- In order to have proper Information Hiding one should take advantage of packaging.
  - **Private** (Ruby, C++, Java)
    - Private variables are only visible from within the same class as they are created in
  - **Protected**(C++, Java, Ruby)
    - Is visible within a class, and in sub classes, the same package but not elsewhere
  - **Public** (C++, Java, Ruby)
    - Has global scope, and an instance can be created from anywhere within or outside of a program

# Packaging Example

```
class Base
  def aMethod
    puts "Got here"
  end
  private :aMethod
end
```

```
class Derived1 < Base
  public :aMethod
end
```

```
class Derived2 < Base end
```

- So how does Ruby have one method with two different visibilities?

# Inheritance

Provides the means to share state and functionality between classes.

- **Subclass (child class)-**

A class that inherits all the properties of the parent class, and adds other properties as well.

- **Superclass (parent class)-**

Parent class: a class one level higher in a class hierarchy

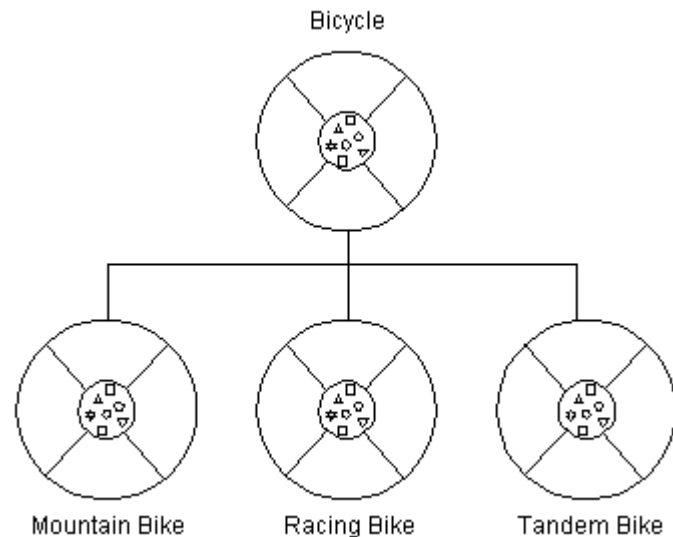
# Inheritance

A child class can be thought of as:

1. An **extension** of a parent class  
(larger set of properties)
2. A **contraction** of a parent class  
(More specialized (restricted) objects)

- **Different Types Exists:**

- Single Inheritance
- Interface Inheritance
- Multiple Inheritance
- Traits
- Mixins



# Benefits of Inheritance

- Software reusability
- Code Sharing
- Consistency of Interface
- Software components
- Rapid prototyping
- Information Hiding



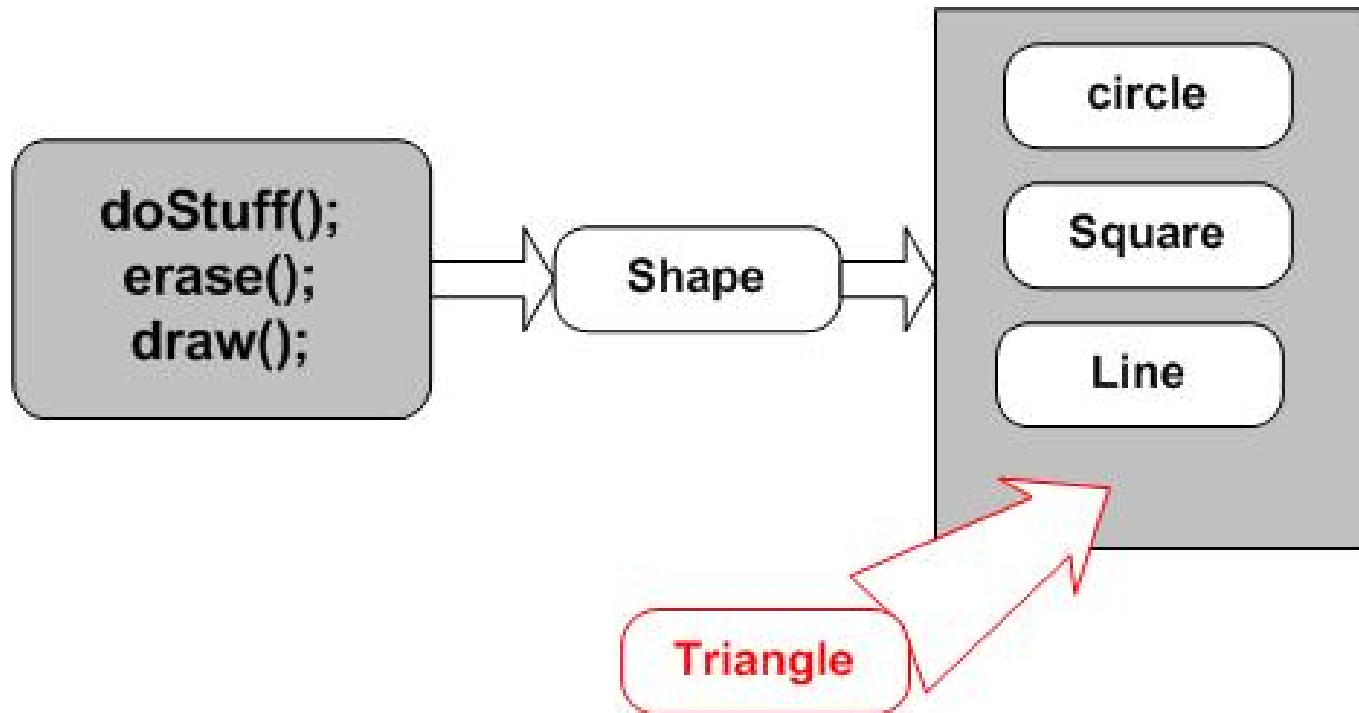
# Polymorphism

- based on Greek roots that mean "many shapes."
- Dynamic Binding allows variables to take different types dependent on the context at a particular time. This is called **polymorphism**.

*"One interface, many methods"*

- Two forms:
  - Compile-Time (function overloading-Covered Thursday)
  - Run-time

# Polymorphism- Example



## **Provides Extensibility**

Can use a subclass object wherever a Base class object is expected.

# Polymorphism- Dynamic (Late) Binding

- **Binding-**

Attaching a function call to a particular definition

- **Dynamic Binding-**

The compiler doesn't make the decision of which class method to use.

```
Shape c1 = new Circle();    c1.doStuff(c);  
Shape t1 = new Triangle();  t1.doStuff(t);  
Shape l1 = new Line();      l1.doStuff(l);
```

**“You’re a shape, I know you can doStuff( ) yourself, do it and take care of the details correctly.”**

- Java functions are automatically dynamically bounded
- C++ you use the keyword ‘virtual’

# Early vs. Late Binding Graphically

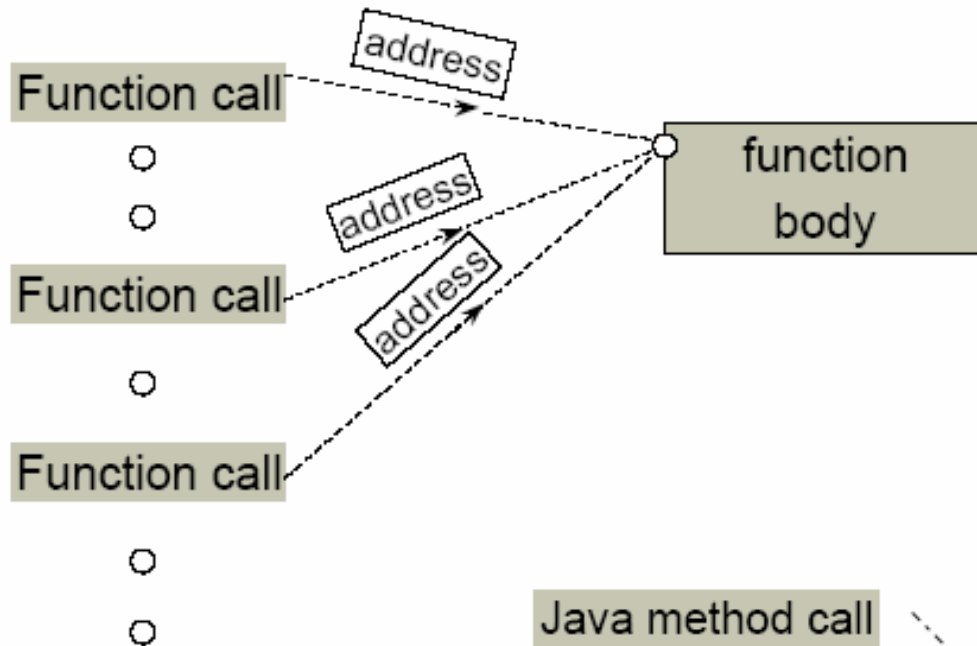


Figure 1. Early Binding

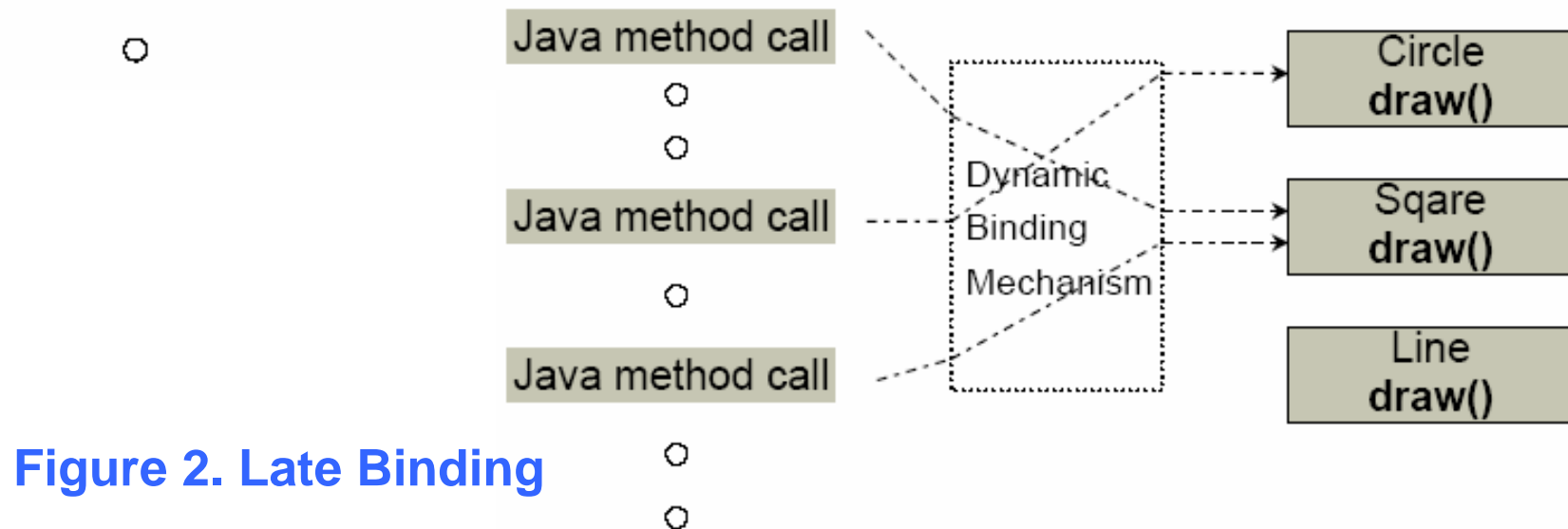
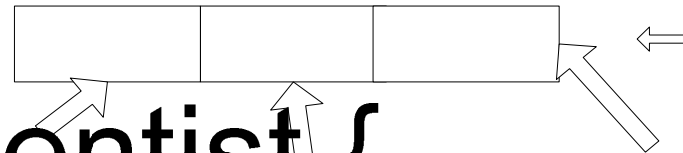
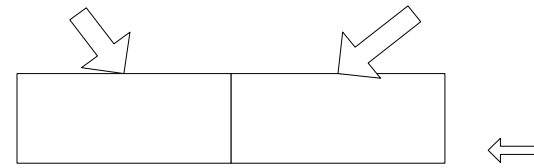
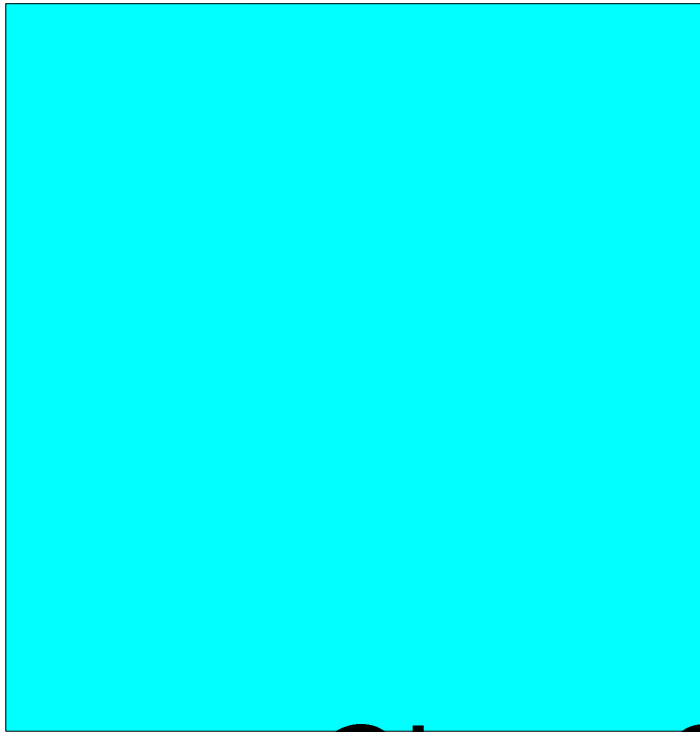


Figure 2. Late Binding

# Polymorphism- Dynamic (Late) Binding



Class Scientist {

....

Char name [40];



vptr

Physicist nick("Nicholas", "nuclear structure");

Scientist \*psc = &nick;

Psc ->show\_all();

public:



# Design Goals

# O.O Design Goals #1

- Responsibility Driven Design (RDD)
  - The most important factor when doing OO programming is a proper design technique that strives on allocating responsibilities.

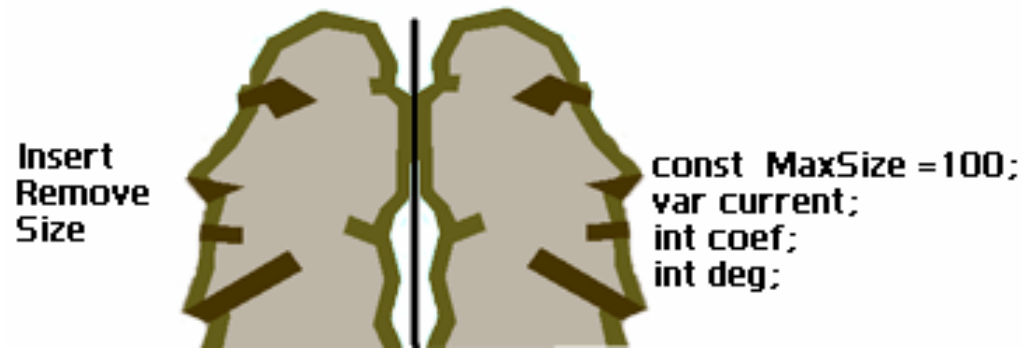
# O.O Design Goals #2

- **Cohesion**
  - The degree to which the responsibilities of a single component form a meaningful unit.
- **Coupling**
  - Describes the relationship between software components.
- We would like to have a **weak coupling** between modules and a **strong cohesion** within modules.



# O.O Design Goals #3

- Each Object has two faces



- Know how to use a component but not how it works.
- “Black-box programming”

# O.O Design Goals #3 (cont.)

## **Parnas's Principles:**

1. The developer of a software component must provide the intended user with all the information needed to make effective use of the services provided by the component, and should provide no other information
2. The developer of a software component must be provided with all the information necessary to carry out the given responsibilities assigned to the component, and should be provided with no other information.



# Inheritance

# Single Inheritance

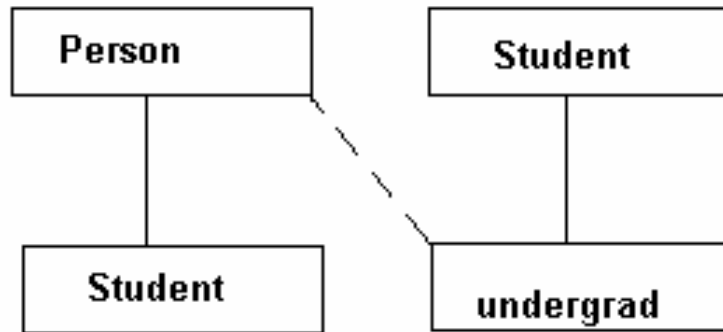
## Single Inheritance-

Data and behavior from a **single** superclass is available to a child class.

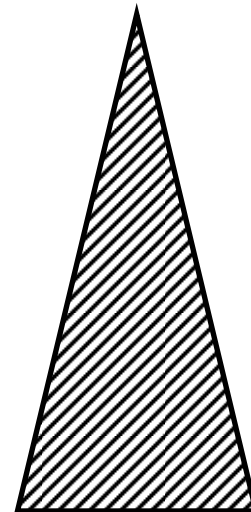
- The most common type of inheritance used in (i.e. Smalltalk, Java, and C#)
- **Advantage:**
  - Very simple to include in ones design
  - Code re-use is very simple to obtain
- **Disadvantage:**
  - Some models cannot be accurately modeled using S.I.
  - Leads to duplicate code, or inserting code in incorrect locations

# Single Inheritance

-Inheritance is always transitive



More  
Specific



More  
General



# Single Inheritance Example for Java

```
class C {  
    int a = 3;  
    void m () {a = a + 1;}  
}
```

```
class D extends C {  
    int b = 4;  
    void n () {m (); b = b + a;}  
}
```

```
D d = new D ();  
C c = new C ();  
d.m (); System.out.println (d.a);  
d.n (); System.out.println (d.b);  
c.n ();
```

# Disadvantages of SI

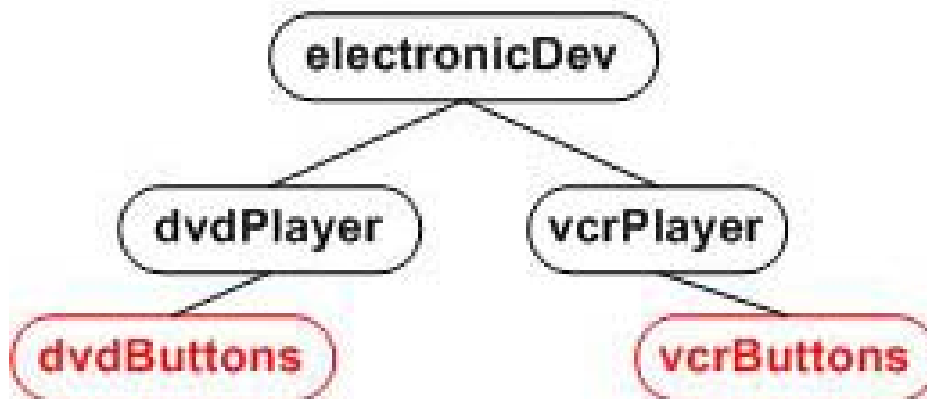
```
class electronicDev { ... }
```

```
class dvdPlayer extends electronicDev {...}
```

```
class vcrPlayer extends electronicDev {...}
```

```
class dvdButtons extends dvdPlayer {...}
```

```
class vcrButtons extends vcrPlayer {... }
```



# Multiple-Inheritance

- In multiple-Inheritance a class can inherit from anywhere in a a class hierarchy
- Allowed to have more than one parent
- M.I. is available in some OO languages such as C++, and Eiffel
- More realistic framework
  - i.e. A child has two parents

*“Multiple inheritance is good, but there is no good way to do it.”*

-A. Snyder 1987

- **Advantages:**
  - Greater Flexibility for design
- **Disadvantages:**
  - “Diamond Problem”
  - No single base class
  - Complexity (Linearization Problem)



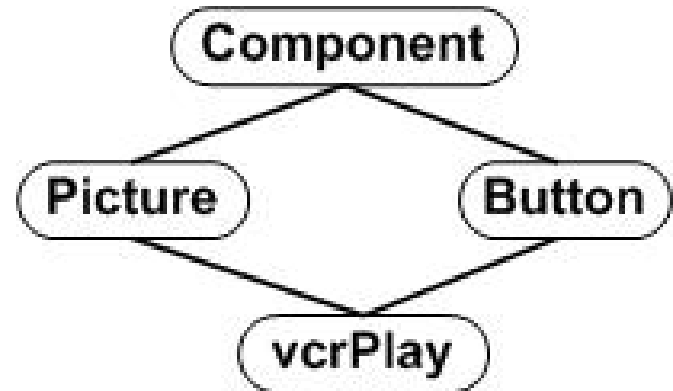
# “Diamond Problem”

```
abstract class Component {  
    abstract void printLabel();  
}  
class Picture extends Component {  
    void printLabel() {  
        System.out.println("I am a Picture");  
    }  
}  
class Button extends Component {  
    void printLabel() {  
        System.out.println("I am a Button");  
    }  
}
```

```
class vcrPlay extends Button, Picture{ }
```

**//Problem:**

```
Component vcrPlay1 = new vcrPlay();  
vcrPlay1.printLabel();
```



# Linearization

- **Options:** manual resolution by the programmer. For example:
  - **C++:** Explicit Delegation - the `scope` resolution operator
  - **Eiffel:** feature renaming.

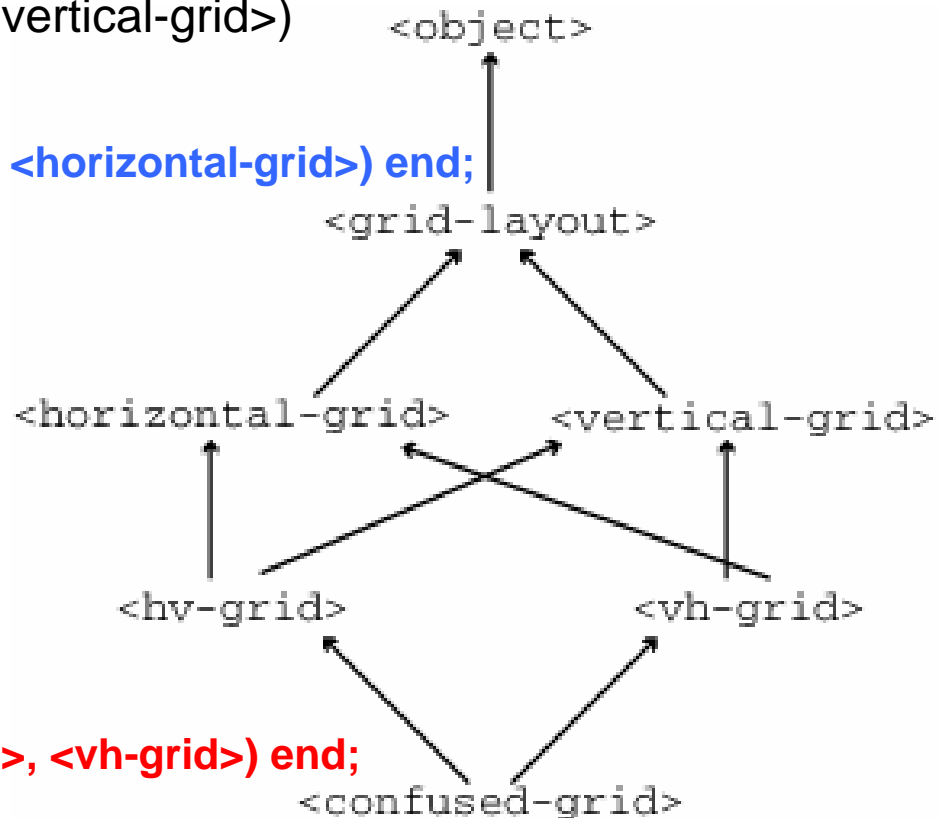
## Alternative....

- **Linearization-**
  - Automatic Resolution of ambiguity
  - Is computed by merging a set of constraints or, equivalently, topologically sorting a relation on a graph.
  - The merge of several sequences is a sequence that contains each of the elements of the input sequences.
  - class precedence list (CPL)-An ordering of the most specific to the least specific

# Linearization- Dylan example

```
define class <grid-layout> (<object>) end class <grid-layout>;  
define class <horizontal-grid> (<grid-layout>) end class <horizontal-grid>;  
define class <vertical-grid> (<grid-layout>) end class <vertical-grid>;  
define class <hv-grid> (<horizontal-grid>, <vertical-grid>) end class <hv-grid>;  
define method starting-edge (grid :: <horizontal-grid>)  
#"left" end method starting-edge;  
define method starting-edge (grid :: <vertical-grid>)  
#"top" end method starting-edge;  
define class <vh-grid> (<vertical-grid>, <horizontal-grid>) end;
```

**What is the starting-edge  
For a vh-grid? (or hv-grid)**



```
define class <confused-grid> (<hv-grid>, <vh-grid>) end;
```

# Interfaces

- Allows one to specify what a class must do, but not tell it how to do it.
- Java's solution to the limitations of S.I. and the problems of M.I.
  - (Also used in C#)
- **Interface != Abstract class**
  - In abstract classes you can define behavior, but only inherit one abstract class.
  - In Interfaces you cannot attach any behavior, but can implement multiple interfaces.
- **Advantages:**
  - Allows more flexibility than simple S.I.
  - You can extend interfaces (to get new interfaces)
- **Disadvantages:**
  - Does not allow for code re-use
  - If you implement an interface you must define all of its methods

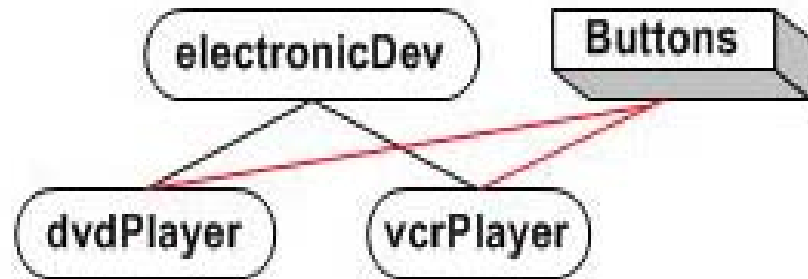
# Interface Example

```
Interface Buttons {  
    void play();  
    void stop();  
    void fwd();  
    void rwd();  
}
```

Class `electronicDev` { ... }

Class `dvdPlayer` extends  
`electronicDev` implements  
`Buttons` { ... }

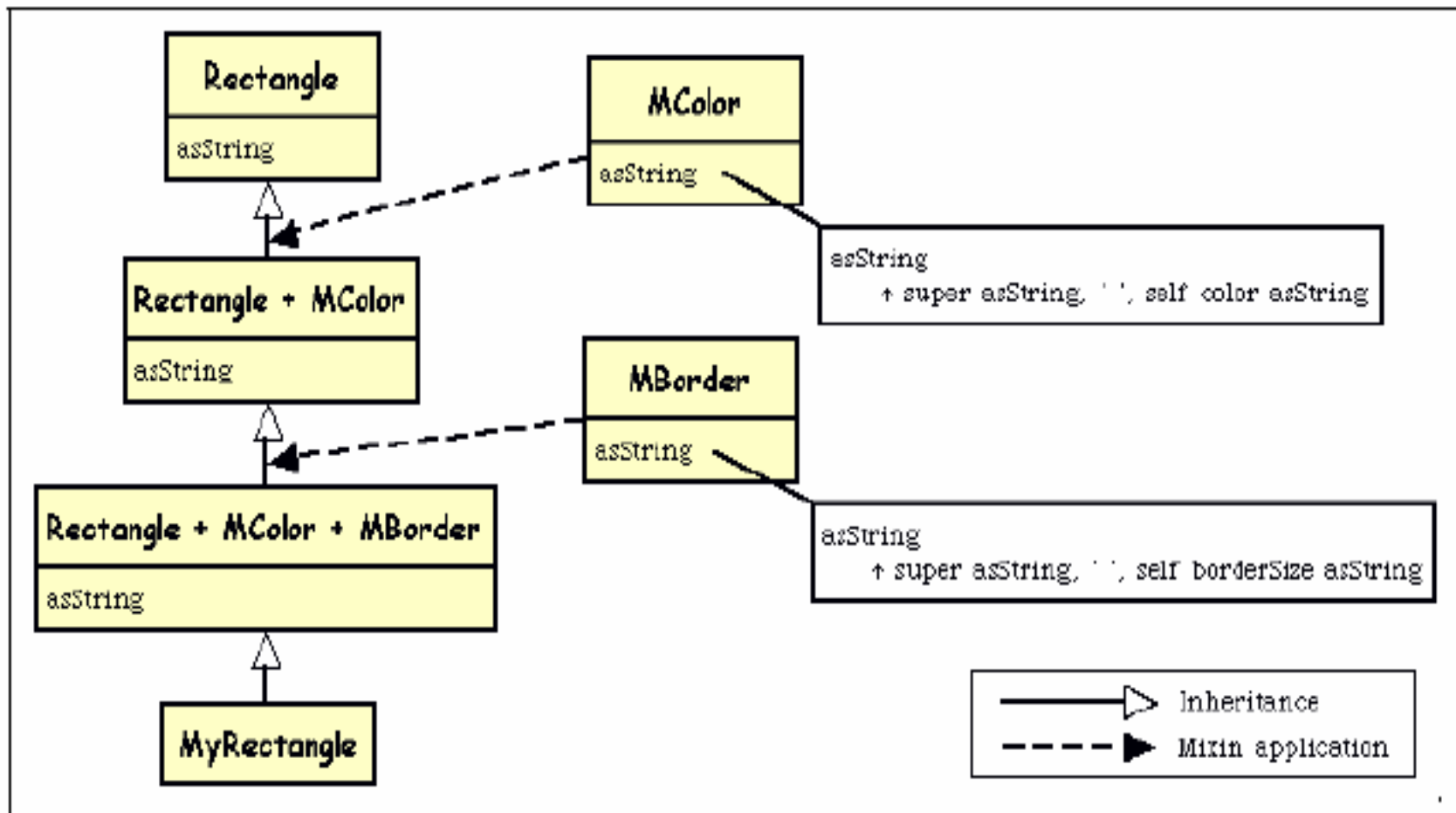
Class `vcrPlayer` extends  
`electronicDev` implements  
`Buttons` { ... }



# Mixins

- It is an abstract subclass specification that may be applied to various parent classes to extend them with the same set of features.
- Available in Languages such as Lisp, Scala, Ruby, Smallscript and CLOS
  - Note has been ported to Java as well
- **Advantages:**
  - Good code reuse
- **Disadvantages:**
  - Total Ordering
  - Delicate Hierarchies

# Mixins Example #1

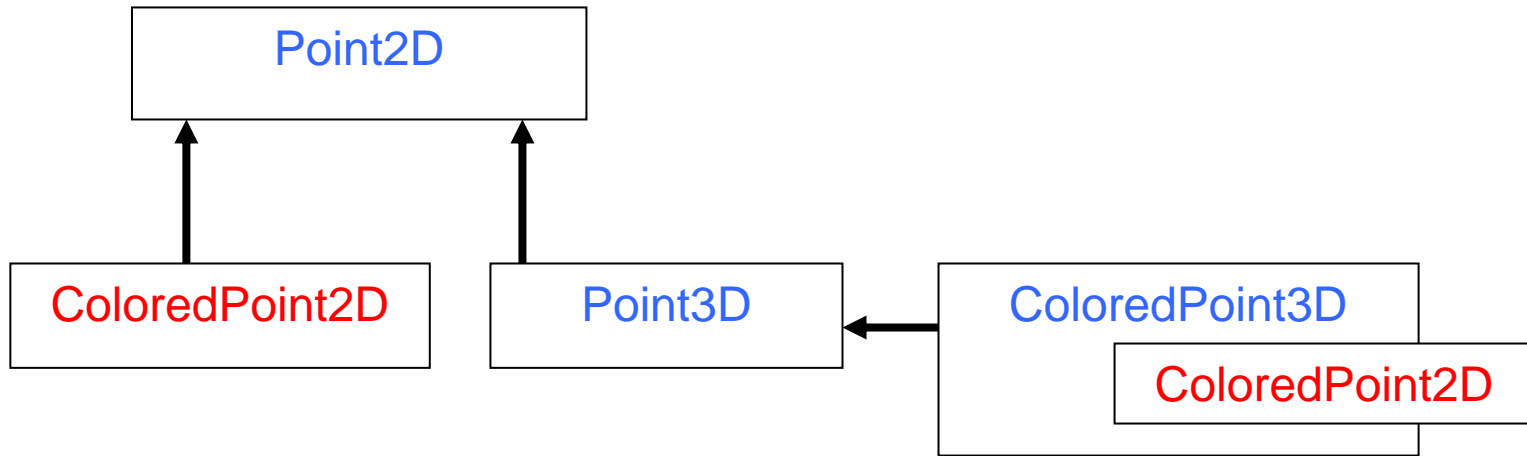


# Mixins Example #2

```
class Point2D(xc: Int, yc: Int) {  
  val x = xc;  
  val y = yc;  
  // methods for manipulating Point2Ds  
}  
class ColoredPoint2D(u: Int, v: Int, c: String)  
  extends Point2D(u, v) {  
  var color = c;  
  def setColor(newCol: String): Unit = color = newCol;  
}  
class Point3D(xc: Int, yc: Int, zc: Int)  
  extends Point2D(xc, yc) {  
  val z = zc;  
  // code for manipulating Point3Ds  
}  
class ColoredPoint3D(xc: Int, yc: Int, zc: Int, col: String)  
  extends Point3D(xc, yc, zc)  
  with ColoredPoint2D(xc, yc, col);
```



# Mixins Example #2 (cont.)



# Traits

- Essentially a group of methods that serve as building blocks for classes and are primitive units of code reuse.
- Similar to interfaces, **but** allows for the method to be partially implemented.
- Similar to Classes, **but** not allowed to possess state. They do not utilize specific state variables, and their methods never directly access state variables.
- A trait provides a set of methods that implement behavior.
- Used in languages such as Scala, and smalltalk
- **Advantages:**
  - No conflicting state
  - Complements S.I. extremely well
  - Less fragile as compared to Mixins
  - Reduces code reuse significantly

# Traits (cont.)

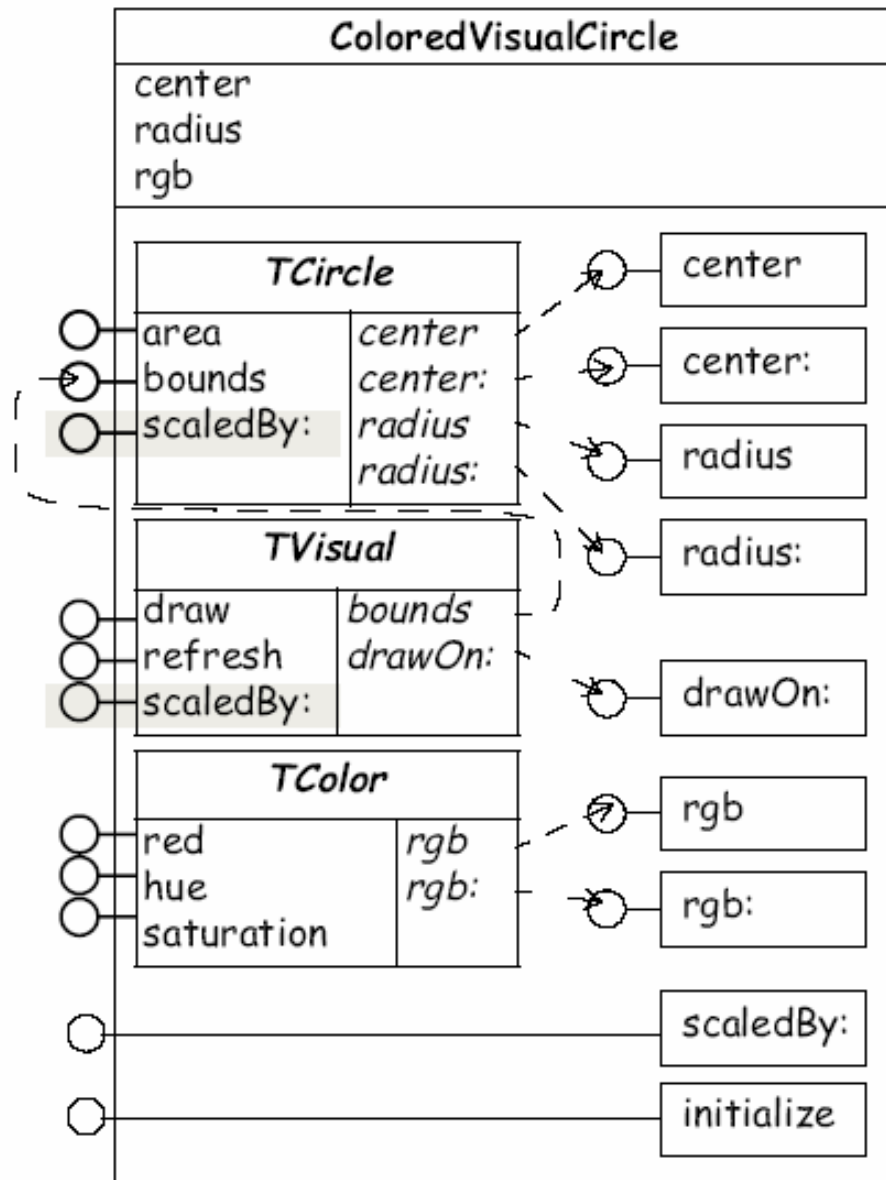
- Traits are used to decompose classes into reusable building blocks
- Class inheritance is to derive a class from another, whereas traits are to achieve greater structure and reusability within a class definition
  - **Class = State + Traits + Glue**

# Traits (cont.)

## Example 1.

```
trait Similarity {  
  def isSimilar(x: Any): Boolean;  
  def isNotSimilar(x: Any): Boolean = !isSimilar(x);  
}
```

# Traits (cont.)



# Principle of Substitutability

If we have two classes, say A and B, such that class B is a subclass of class A, it should be possible to substitute instances of class B for instances of class A in any situation with no observable effect.

# Trees Versus Forest

- Borrowing from Graph Theory:
  - A **tree** is a connected undirected graph with no simple circuits. (S.I.)– also called single-rooted hierarchy trees
- In M.I. You will get multi-rooted class hierarchies also known as a **forest**

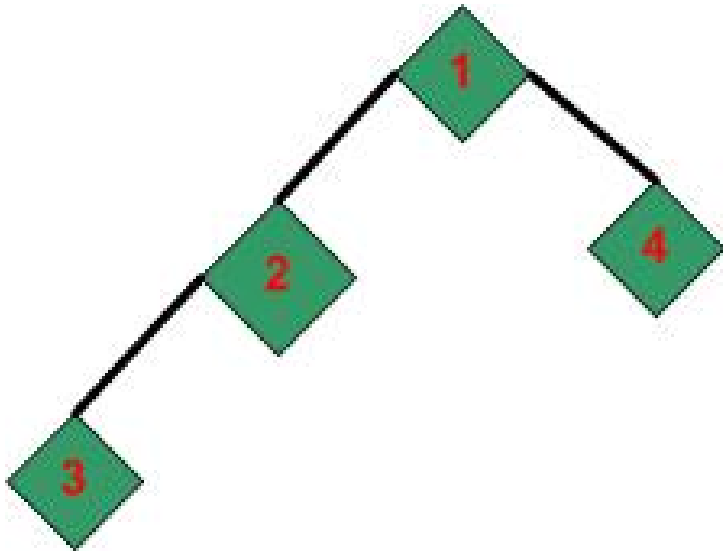


Fig. 1 - Inheritance Graph

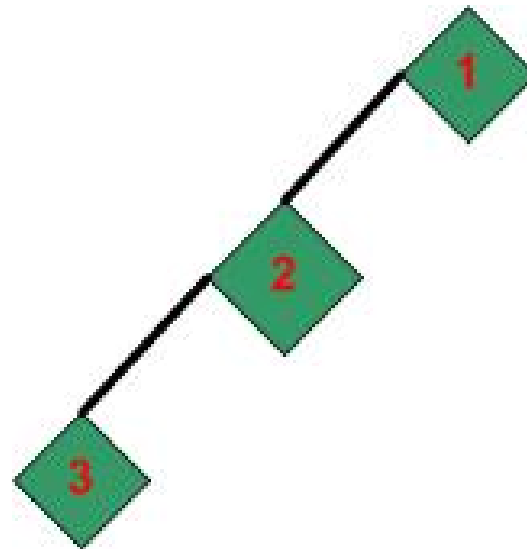


Fig. 2 – Inheritance Chain

# Trees Versus Forest

## Tree

- Pros:
  - Single class hierarchy
  - Standardization: Root's functionality inherited by all objects – all have basic functionality
- Cons:
  - Tight coupling
  - larger libraries
- Examples
  - Java's classes
  - Smalltalk
  - Objective C

## Forest

- Pros:
  - Many smaller hierarchies.
  - Smaller libraries of classes for application, less coupling possible
- Cons:
  - no shared functionality among all objects
- Examples
  - Java's interfaces
  - C++





# Typing

# Typing

*“A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute”*

- Pierce

- A **class** is a property of objects, a **type** is a property of variables and expressions.
- **Subtype-**
  - A class that satisfies the principle of substitutability.
- **Subclass-**
  - This is constructed through inheritance and does not have to satisfy the principle of substitutability.

# Subtyping vs. inheritance

- **What is the difference between subtyping and inheritance?**
  - The two concepts are not the same
  - Not all subclasses are subtypes
  - Subtypes can be constructed without being subclasses.
  - Inheritance extends a class to yield a new family of types.
- **What links subtyping and inheritance together?**
  - A subclass generates objects whose type is a subtype of the objects generated by a superclass.

# Prototype Based

# Prototype Based languages

- Do not think that because you are coding in OO that you must use classes.
- Classes are:
  - Static
  - Used to share methods
  - Too rigid
- Prototype-based language do not distinguish between classes and instances.
- They have only objects, which are similar to instances.

# Prototype Based languages

- i.e. JavaScript, Self, Kevo, NewtonScript, Mica, Obliq....
  - In Ruby you do “String.new” and not bother instantiating an object with “new String()”
- In the above languages you only create concrete objects (which are called prototypes).
- “prototype-based” == “object-based” == “programming by example”

# Prototype Based languages

- **Analogy:**
  - Building an object in a class-based language is like building a house from a plan, while building an object in a prototype-based language is like building a house like the neighbors.
- **Advantage: greater flexibility**
- **Create new objects not by instantiation but by cloning**

# Prototype vs. Class based

- Prototype vs. Abstraction
- Humans can relate to prototyping better
- It's hard to abstract or generalize all behavior ahead of time when using the classical OO paradigm
- You cannot change the number or the type of properties of a class after you define the class



# Java

```
public class Employee {
    public String name;
    public String dept;
    public Employee () {
        this.name = "";
        this.dept = "general";
    }
}

public class Manager extends Employee {
    public Employee[] reports;
    public Manager () {
        this.reports = new Employee[0];
    }
}

public class WorkerBee extends Employee {
    public String[] projects;
    public WorkerBee () {
        this.projects = new String[0];
    }
}
```

# JavaScript

```
function Employee () {  
  this.name = "";  
  this.dept = "general";  
}
```

```
function Manager () {  
  this.reports = [];  
}
```

```
Manager.prototype = new Employee;
```

```
function WorkerBee () {  
  this.projects = [];  
}
```

```
WorkerBee.prototype = new Employee;
```

```
//create new instance  
nick = new WorkerBee;
```

```
//modify properties  
nick.name = "John";  
nick.dept = "admin";
```

```
//add new properties  
nick.bonus = 3000;
```

```
//add new property to the prototype  
Employee.prototype.specialty = "none";
```

# Java vs. JavaScript

<b>Class-based</b>	<b>Prototype-based</b>
Class and instance are distinct entities.	All objects are instances.
Define a class with a class definition; instantiate a class with constructor methods.	Define and create a set of objects with constructor functions.
Create a single object with the new operator.	Same.
Construct an object hierarchy by using class definitions to define subclasses of existing classes.	Construct an object hierarchy by assigning an object as the prototype associated with a constructor function.
Inherit properties by following the class chain.	Inherit properties by following the prototype chain.
Class definition specifies <i>all</i> properties of all instances of a class. No way to add properties dynamically at runtime.	Constructor function or prototype specifies an <i>initial set</i> of properties. Can add or remove properties dynamically to individual objects or to the entire set of objects.



In Conclusion...

# O.O. Advantages

- Arguably a more intuitive programming paradigm
- Ease of Maintenance (users benefit from strong encapsulation capabilities)
- Integration is simplified (again due to encapsulation)
- Increase of code reuse

# OO Disadvantages

- A new paradigm to learn.
- Confusing and mis-used terminology
- Many OO environments can be inefficient for certain tasks.
  - Cross-cutting concerns

# Discussion & Questions

Slides Available at: <http://www.cas.mcmaster.ca/~merizzn>

# Exercise#1

Which is valid in C++ and why?

**Figure 1**

```
int main() {
    A objecta;
    objecta.i = 1;
    B *objectb;

    objectb = &objecta;
    return 0;
}
```

**Figure 2**

```
class base {
public:
    int i;
};

class derived : public base{
public:
    int j;
};

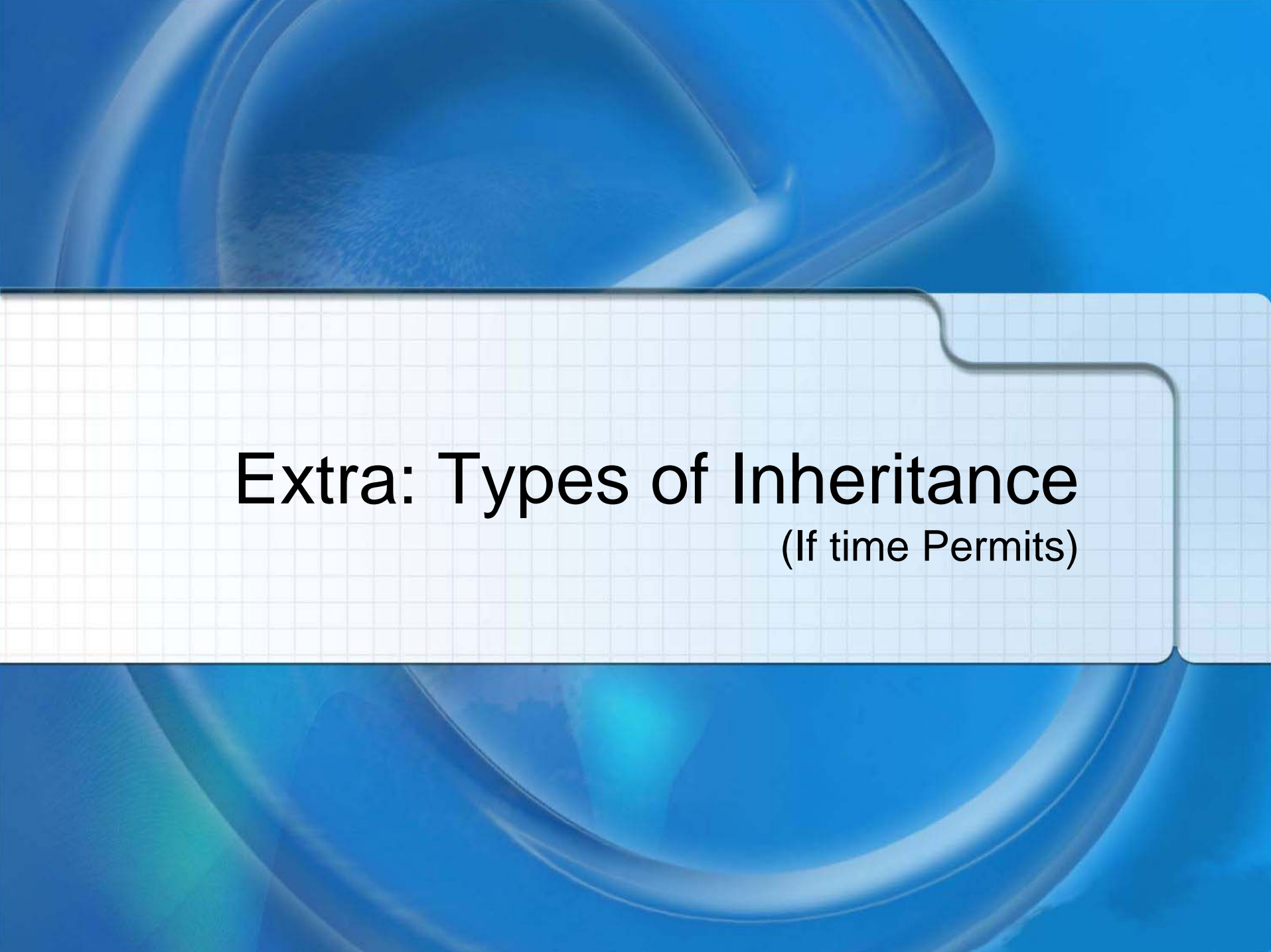
int main() {
    derived ob;
    base *p;
    p = &ob;
    return 0;
}
```



# Exercise#2

```
abstract class Base{
    abstract public void myfunc();
    public void another(){
        System.out.println("Another method");
    }
}
public class Abs extends Base{
    public static void main(String argv[]){
        Abs a = new Abs();
        a.amethod();
    }
    public void myfunc(){
        System.out.println("My func");
    }
    public void amethod(){
        myfunc();
    }
}
```

- 1) The code will compile and run, printing out the words "My Func"
- 2) The compiler will complain that the Base class has non abstract methods
- 3) The code will compile but complain at run time that the Base class has non abstract methods
- 4) The compiler will complain that the method myfunc in the base class has no body, nobody at all to love it



# Extra: Types of Inheritance

(If time Permits)

# Types of Inheritance

## Inheritance used in a variety of ways:

(List is not complete)

- Subclassing for Specialization
- Subclassing for Specification
- Subclassing for Construction
- Subclassing for Generalization
- Subclassing for Extension

# Subclassing for Specialization

- The new class is a specialized form of the parent class but satisfies the specifications of the parent in all relevant respects. It's a subtype of the parent.
- Most common form of inheritance
- Preserves substitutability
- **Example:**
  - Professor is a specialized form of Employee

# Subclassing for Specification

- This can be seen when the parent class does not implement actual behavior but simply defines the behavior that will be implemented in child classes.
- Preserves substitutability
- Goal is to define a common interface for a group of related classes
- **Example:**
  - class StackInArray gives implementations for method signatures defined in abstract class Stack Java:  
StackInArray extends Stack

# Subclassing for Construction

- Parent class is used only for its behavior. In this case the child class is not a subtype.
- No is-a relationship to the parent.
- Frowned upon
- **Example:**
  - Extending List class to develop Set BUT without “hiding” unneeded method

# Subclassing for Generalization

- The opposite of subclassing for specialization.
- Here the subclass extends the behavior of the parent class in order to create a more general kind of object.
- Does not change any of the inherited behavior.
- Try to avoid (solution: invert the class hierarchy)
- **Example:**
  - graphics display system which has a black&white based window class called Window. You can extend Window to ColoredWindow.

# Subclassing for Extension

- In subclassing for generalization we modify or expand on something already there.
- Whereas in subclassing for extension we add completely new functionality.
- We aren't changing inherited behavior we are simply adding new ones.
- **Example:**
  - StringSet extends Set, adding string-related methods (I.E. search by name)



# References

- Prata, Stephen “C++ Primer Plus” SE, Waite Group, 1995.
- Liskov Barbara, Gutttag Barbara “Program Development in Java”, Addison Wesley, 2000.
- Blaschek, Gunther, Object-Oriented Programming with Prototypes.
- Budd, Timothy “Understanding Object-Oriented Programming”, Second Edition.
- Scharli, Ducasse, Nierstrasz, Black, Traits: Composable Units of Behavior, Technical Report # CSE 02-012, November 2002
- Scharli, Ducasse, Nierstrasz, “Classes = Traits + States + Glue-Beyond mixins and multiple inheritance”, The Inheritance Workshop at ECOOP 2002.
- Kim Barrett, Bob Cassels, Paul Haahr, David A. Moon, Keith Playford, and P. Tucker Withington, “A *Monotonic Superclass Linearization for Dylan.*” presented on 12 October 1996 at the 1996 Object-Oriented Programming Systems, Languages and Applications (OOPSLA '96) conference in San Jose, California