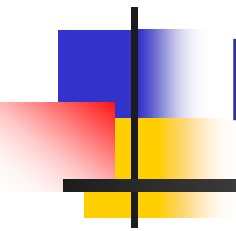
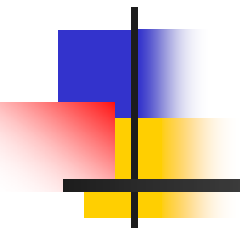


Block I – Unit 4

Inheritance, Composition, and Polymorphism



Section 1

Java Mechanisms for Software Reuse

Based on Budd chapter 10

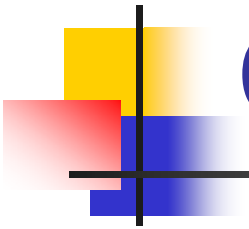
Java Mechanisms for Software Reuse

- Objectives:
  - Compare and Contrast the inheritance and composition mechanisms and be able to analyse the main advantages and disadvantages of each.
  - Use that knowledge as a guide to building classes using either inheritance, composition, or both.
  - Describe how the is-a and has-a relationships relate to inheritance, subtyping and composition.

The Abstract data type stack

- The abstract data type named stack has the following four operations:

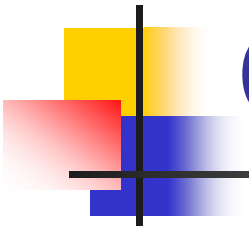
empty	Returns true if the given stack contains no elements, and returns false otherwise.
push	Returns the stack with a given element currently added to the top.
peek	Returns a copy of the element currently at the top of the given stack
pop	Removes the top element from the given stack



Composition vs Inheritance

- Composition means implementing the class *Stack* in such a way that each instance of stack contains a *Vector* object that holds the elements pushed on the stack.

- The stack operations are then implemented as methods that are already defined for a vector object.



Composition vs Inheritance

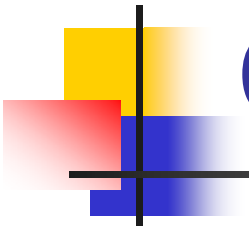
- Inheritance means extending the implementation of *Vector* to include methods implementing the operations that are required for a stack.

Implementation of class *Stack* using Composition.

```
public class Stack {
    private Vector theData;

    public Stack ()
        { theData = new Vector(); }

    public boolean empty ()
        { return theData.isEmpty(); }

    public Object push (Object item)
        { theData.addElement(item); return item; }

    public Object peek ()
        { return theData.lastElement(); }

    public Object pop () {
        Object result = theData.lastElement ();
        theData.removeElementAt(theData.size() – 1);
        return result;
        }
}
```

Implementation of class *Stack* using Inheritance.

```
public class Stack extends Vector{

    public Object push (Object item)
        { addElement(item); return item; }

    public Object peek ()
        { return elementAt(size() - 1); }

    public Object pop () {
        Object obj = peek ();
        removeElementAt(size() – 1);
        return obj;
        }
}
```



Composition vs Inheritance

- Which implementation is better for the *Stack;* Composition or Inheritance??
  - Hint: Think about the functionality of the stack (method size in the class *Vector*)

Composition and Inheritance in *InputStream*

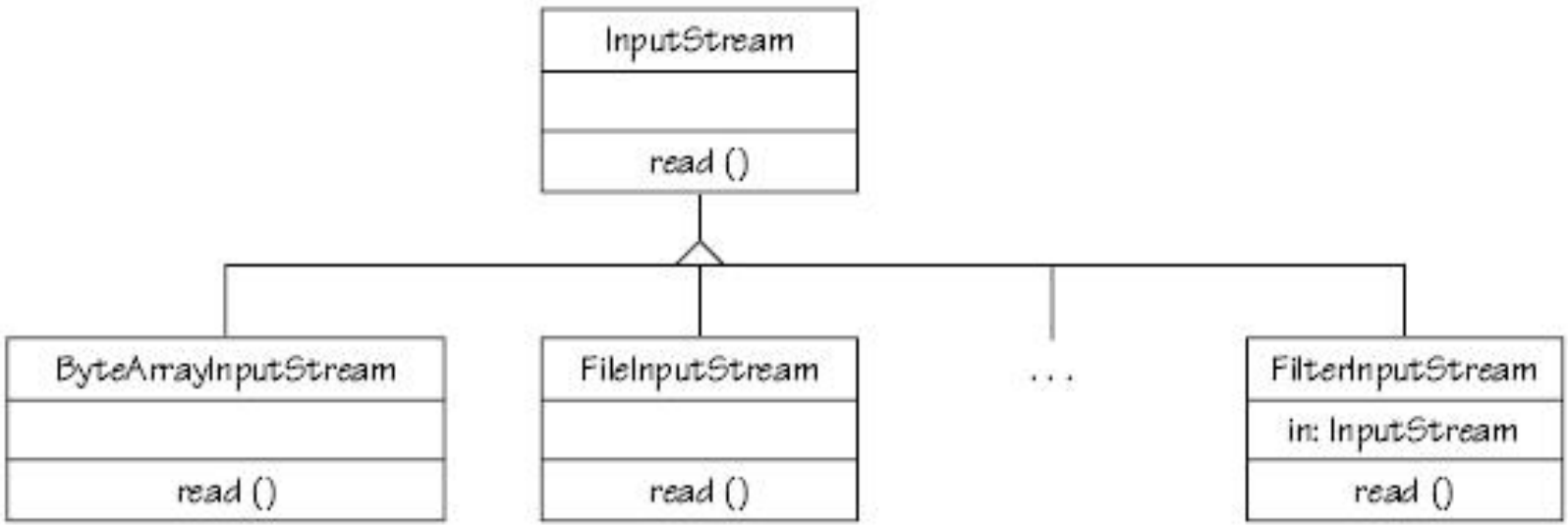


Figure 1.1 Inheritance and composition used by FilterInputStream class

- In the figure above the *FilterInputStream* class is inheriting from *InputStream* and as well as having a data field of type *InputStream*.

Composition and Inheritance in *InputStream*

- Also the figure shows that *InputStream* has a method named read which is intended to return a single byte. That method is overridden in all the subclasses so that in each case the byte is taken from a different source (ByteArray, File, etc..).

Composition and Inheritance in *InputStream*

- The new class *FilterInputStream* also overrides the *read* method but makes use of its data field *in* to provide a byte from whatever type of *InputStream* is referenced by *in*

```
public int read() {
  return in.read (); }
```

- For example, if *in* referenced a *ByteArrayInputStream*, the byte returned by the *read* method would be from that *ByteArrayInputStream*.

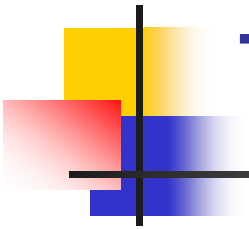
Composition and Inheritance in *InputStream*

- Suppose we want to add to *FilterInputStream* a new functionality to count the number of bytes read, via a method called *count* we can do the following:

```
private int number;
public int count() {
        return number; }


public int read() {
        number = number + 1;
        in.read(); }
```

The is-a relationship

- The relationship derives its name from a simple rule of thumb that tests the relationship to determine if concept X is a specialized instance of concept Y.
- For example:
  - A dog is-a mammal
  - A PinBall is-a Ball



The has-a relationship

- The has-a relationship is when the second concept is a component of the first, but the two are not in any sense the same thing.
- For example:
  - A car has-a engine

Composition and Inheritance

- Composition means incorporating data field(s) into the definition of a class. For example, composition was involved in building the *Ball* class since it contains an instance variable (data field) called location of type *Point* In other words, an object of type *Ball* is either partly composed or (*has-a*) *Point* object. The *has-a* relationship is often represented diagrammatically as a line drawn between classes as shown below.

```
class Ball {
        .
        .
        protected Point location;
        .
        .
}
```

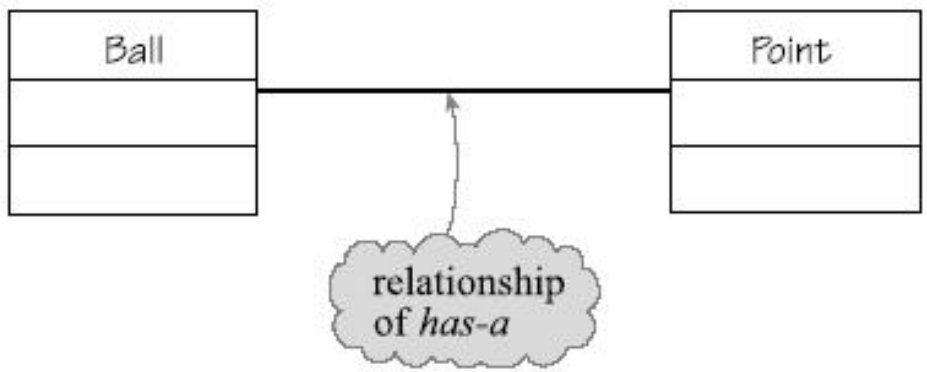
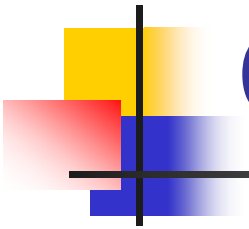


Figure 1.2 Class definition and diagram showing composition



Composition and Inheritance

- Inheritance can be a manifestation of the *is-a* rule, but only where the subclass is also a subtype. In the case of *inheritance for construction* , the *is-a* relationship does not exist (i.e. it is not manifested).

Composition Vs. Inheritance

Inheritance carries with it an implicit assumption of substitutability.	No such assumption of substitutability is associated with the use of composition.
To know what operations are legal for the new structure, the programmer must examine the declaration for the original.	In Composition, the advantage is that it more clearly indicates exactly what operations can be performed.
Using Inheritance is not necessary to write any code to access the functionality provided by the class on which the new structure is built.	Composition requires almost always longer source codes and provide less functionality.

Composition Vs. Inheritance

Inheritance does not prevent users from manipulating the new structure using methods from the parent class.	
A component constructed using Inheritance has access to fields and methods in the parent class that have been declared as protected.	A component constructed using Composition can only access the public portions of the included component.
A programmer faced with understanding the inheritance version needs to understand both classes in some cases.	Composition code, although longer, is the only code that another programmer must understand.

Relationship Diagram

- The figure below shows that *FilterInputStream* has a relationship of both Inheritance and Composition with *InputStream*.

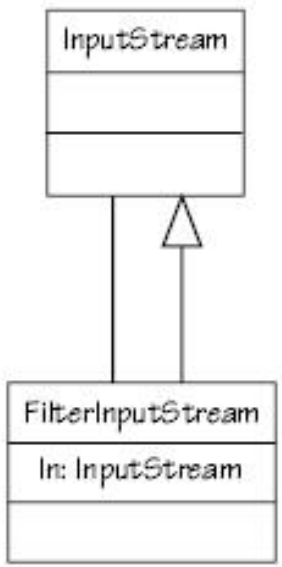
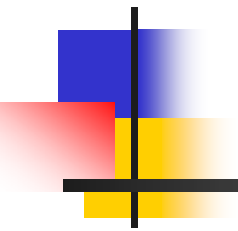


Figure 1.3 The relationships between FilterInputStream and InputStream

Key Terms and Concepts

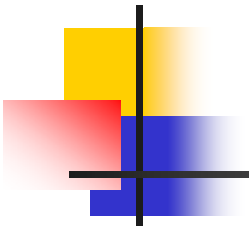
composition	has-a relationship	substitutability
data hiding	is-a relationship	subtype
encapsulation	reuse	



Section 2

Implications of Inheritance

Based on Budd chapter 11



Implications of Inheritance

- Objectives:
  - Describe what is meant by a polymorphic variable and relate it to the concept of inheritance;
  - Describe what is meant by, and exploit, the concepts of reference semantics and copy semantics as they affect assignment, tests for equality and copying objects.

The Polymorphic Variable

- A polymorphic variable is declared as maintaining a value of one type but in fact holds a value from another type. Just like the variable “form” in the example below.

```
class ShapeTest {
    public static void main (String [] args) {
        Shape form = new Circle (10, 10, 5);
        System.out.println (“form is “ + form.describe());
        form = new Square (15, 20, 10);
        System.out.println (“form is “ + form.describe());
    }
}
```

Polymorphic Variable

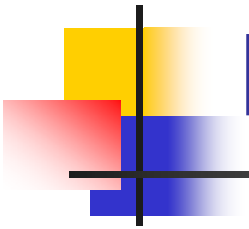
- Consider the following classes:

```
class Shape {

    protected int x;
    protected int y;

    public Shape (int ix, int iy) { x = ix; y = iy; }

    public String describe ( ) { return "unknown shape";}
}
```

Polymorphic Variable

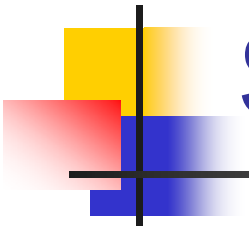
```
class Square extends Shape {
    protected int side;
    public Square (int ix, int iy, int is) {super(ix, iy); side = is; }
    public String describe ( ) { return "square with side "+ side; }
}
class Circle extends Shape {
    protected int radius;
    public Circle (int ix, int iy, int ir) {super(ix, iy); radius = ir; }
    public String describe ( ) { return "circle with radius "+ radius; }
}
```

- Which of following 2 statements are legitimate in Java:
  - Shape s = new Square(6,8,3);
  - Circle c = new Square(10,12,4);



Memory Layout

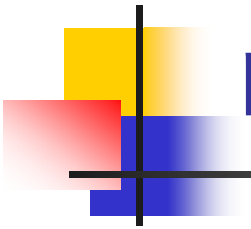
- From the point of view of memory manager, there are two major categories of memory values.
  - stack-based memory locations
  - heap-based memory values
- Let us see how variables are normally stored in most programming languages



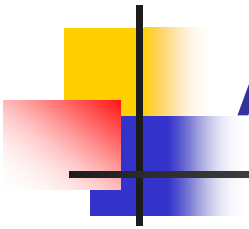
Stack-based memory locations

- Stack-based memory locations are tied to method entry and exit. When a method is started, space is allocated on a run-time stack for local variables. These values exist as long as the method is executing, and are erased, and the memory recovered, when the method exits.

Advantages of stack-based memory allocation

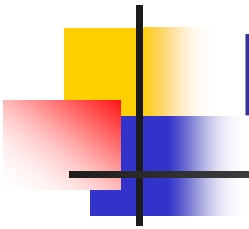


- All local variables can be allocated or deallocated as a block, instead of one by one. This block is called an activation record.
- Internally, variables can be described by their numeric offset within the activation record.



A serious disadvantage

- The numeric offsets associated with variables must be determined at compile time, not at run time. In order to do this, the compiler must know the amount of memory to assign to each variable (static address does not change at execution time).
  - Note: this is exactly the information we do not know for a polymorphic variable!



Example

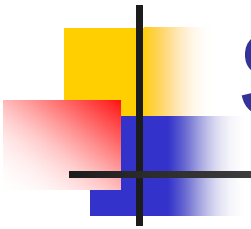
```
class FacTest {
   static public void main (String [] args) {
      int f = factorial (3);
      System.out.println ("Factorial of 3 is " + f);
   }

static public int factorial (int n) {
   int c = n-1;
   int r;
   If (c > 0)
      r = n * factorial(c);
   else
      r = 1;
   return r;
   }
}
```

Example

0	**n:1**	Third activation record
4	**r:1**	
8	**c:0**	
0	**n:2**	Second activation record
4	**r:?**	
8	**c:1**	
0	**n:3**	First activation record
4	**r:?**	
8	**c:2**	

Heap-based memory values as a solution

- In Java, a heap is an alternative memory-management system to the activation record stack, one that is **not** tied to method entry and exit.
- Memory is allocated on the heap when explicitly requested (using the *new* operator) and is freed, and recycled, when no longer needed (Dynamic address).



Semantics

- The meaning of the word semantics is meaning! Hence, the term reference semantics is used when we wish to give a meaning to something in terms of references.

  - Note: reference semantics is sometimes also called pointer semantics

Assignment (reference semantics)

Consider the following code:

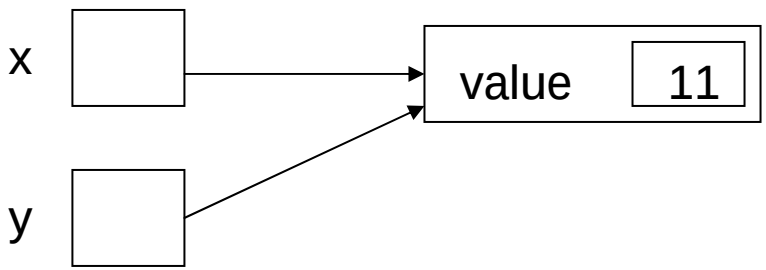
```
public class Box{
    private int value;

    public Box ()
        { value = 0; }

    public void setValue (int v)
        {value = v; }

    public int getValue ()
        { return value; }
}
```

Assignment (reference semantics)

```
public class BoxTest {
    public static void main (String [] args) {

        Box x = new Box();
        x.setValue (7);

        Box y = x;
        y.setValue (11);

        System.out.println("contents of x " + x.getValue());
        System.out.println("contents of y " + y.getValue());
    }
}
```

Assignment (reference semantics)

After BoxTest is executed, the following will be generated in memory:

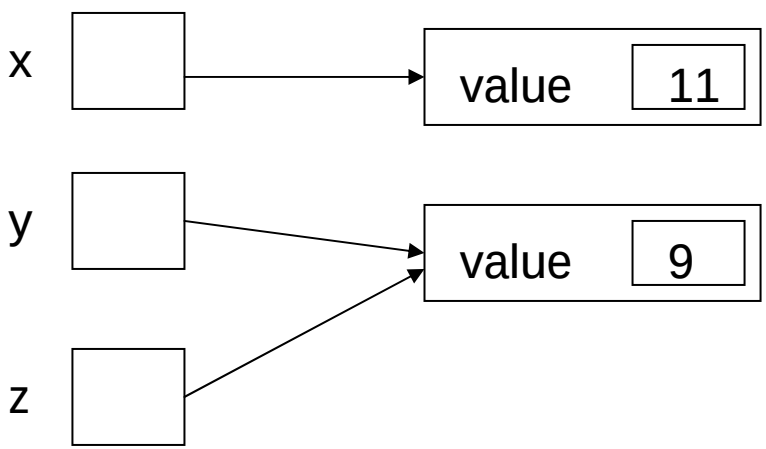


The key observation is that the two variables, although assigned separate locations on the activation record stack, nevertheless point to the same location on the heap.

Assignment (reference semantics)

Modify the main method of BoxTest to include the following after y.setValue(11)

```
Box z = new Box();
z.setValue (9);
y = z;
```

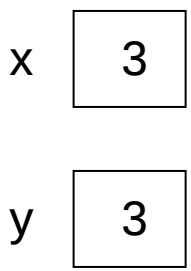


Assignment (copy semantics)

- Variables with primitive data types store their values rather than references to these values

For example:

```
int x = 3;
y = x;
```

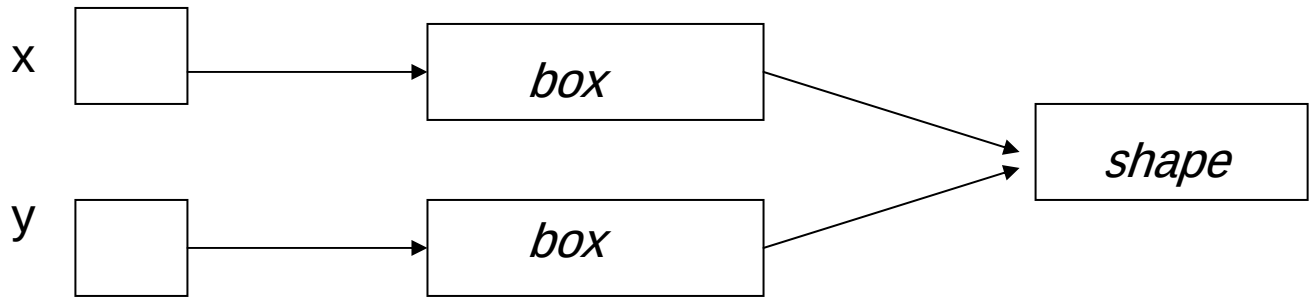


- The above example is known as assignment with copy semantics.

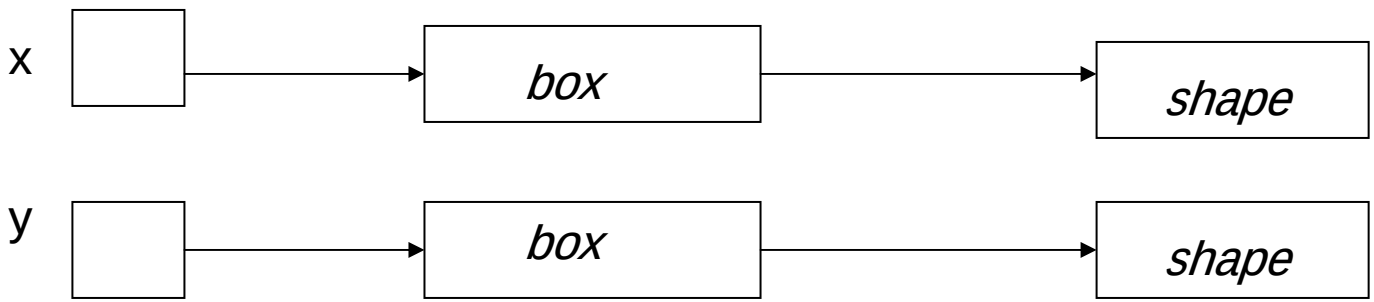
Assignment (copy semantics)

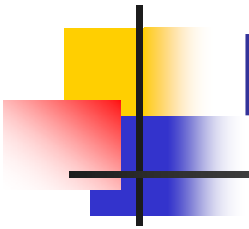
- Shallow copy vs. Deep copy

Shallow Copy:



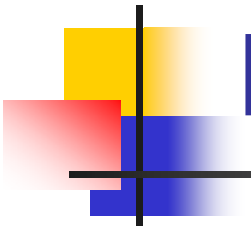
Deep Copy:





Equality Test

- Consider the following:
  - The ASCII of ‘a’ is 141
  - 3 + 4 = 7
  - 2.0 = 2
- In Java == is used to test for equality between 2 values and hence the following would return true:
  - ‘a’ == ‘\141’
  - (3 + 4) == 7
  - 2 == 2.0 (The integer value is converted to float and then compared).

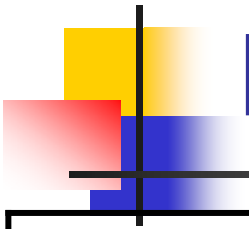


Equality Testing with Objects

```
Integer x = new Integer (7);
Integer y = new Integer (3 + 4)
```

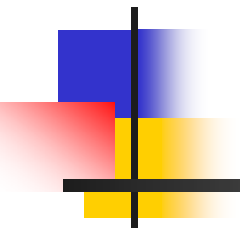
x == y would return false!
x=y would return true!

Remember that objects in Java are internally represented by pointers, the natural interpretation of assignment using reference semantics. Hence, using == with objects is often termed as "testing object identity"



Key Terms and Concepts

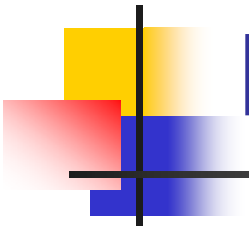
activation record	formal argument	semantics
actual argument	formal parameter	shallow copy
actual parameter	heap	stack
assignment	heap memory management	stack frame
coercion	object identity	stack memory management
copy semantics	pointer	value semantics
copying objects	polymorphic variable	
deep copy	reference semantics	



Section 3

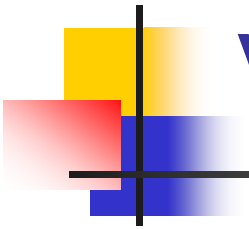
Polymorphism

Based on Budd chapter 12



Polymorphism

- Objectives:
  - Recognize pure polymorphism;
  - Recognize overloading of method names;
  - Describe the concept of coercion;
  - Distinguish between overloading of method names in classes unrelated by inheritance and overloading of method names within the same class definition;
  - Recognize parametric overloading;
  - Recognize overriding and distinguish between replacement and refinement semantics;
  - Understand the use of abstract methods;
  - Appreciate why polymorphism may not be an efficient mechanism but provides other characteristics that are more useful to the programmer such as ease of development and use, and readability of code.



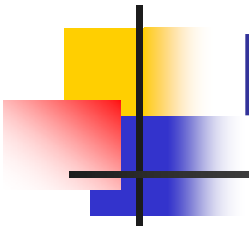
Varieties of Polymorphism

- Pure Polymorphism occurs when a single function can be applied to arguments of a variety of types.
- In pure polymorphism there is one function (the code body) and a number of interpretations (different meanings).
- The other extreme occurs when we have a number of different functions all denoted by the same name. This is known as overloading or ad hoc polymorphism.

Example of Pure Polymorphism

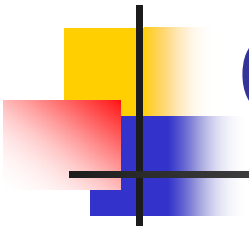
```
public class String {
   public static String valueOf (Object obj)
   {
      if (obj == null)
         return “null”;
      return obj.toString();
   }
}
```

- The method valueOf does not know the type of its arguments.



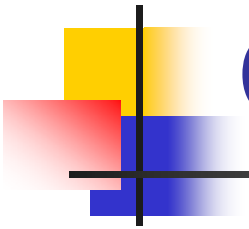
Polymorphic Variables

- In dynamically bound languages (such as smalltalk), all variables are potentially polymorphic.
- In statically typed languages, such as Java, the situation is slightly more complex. Polymorphism occurs in Java through the difference between the declared (static) class of a variable and the actual (dynamic) class of the value the variable contains. (target.hitBy, target is of type PinBallTarget)



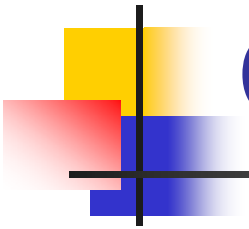
Overloading

- There are 3 forms of method overloading:
  - Where the method heading is the same as that in the parent class, but the body is redefined.
  - Where methods in the same class (or classes related by inheritance) have the same names but different arguments;
  - Where methods in two or more classes not linked by inheritance have the same name (whether or not they have the same arguments is not an issue).



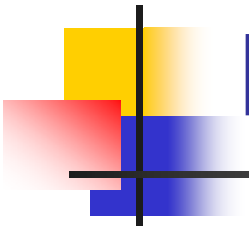
Overloading

- Coercion:
  - Coercion occurs when a value of one type is converted to a value of another type.
- Parametric Overloading:
  - Parametric Overloading occurs when there are two or more methods in the same class definition with the same name but that differ in either (or both) the number of arguments or the type of arguments.



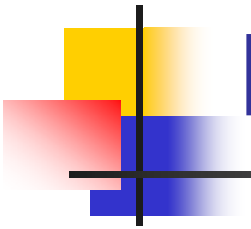
Overriding

- If in a subclass a method with the same name as that of the superclass is defined that hides access to the inherited method, we say the method of the subclass overrides the one in the superclass.



Replacement and Refinement

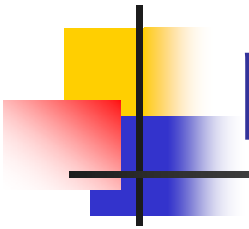
- Overriding by replacement (replacement semantics): occurs when the code of the method in the parent class is not executed.
- Overriding by refinement (refinement semantics): occurs when the code of the subclass invokes the code of the superclass.



Deferred Methods

- An abstract method (also known as a deferred method) is a method that is specified in the parent class, that is, it is without a body, but must be implemented in a descendant class.
- Interfaces contain abstract methods only.
- An abstract method is denoted by including the keyword *abstract* in the heading of the method (and there should be no body associated with the method).

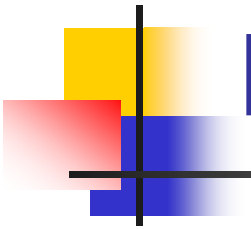
Advantages of polymorphic programming techniques



- Rapid program development
- Consistent application behaviour
- Code reuse

Key Terms and Concepts

abstract	dynamic class	polymorphism
abstract method	overloading	pure polymorphism
coercion	overriding	refinement semantics
default constructor	overriding by refinement	replacement semantics
deferred method	overriding by replacement	static binding (methods)
dynamic binding (methods)	parametric overloading	static class



Holiday