**Object Oriented Programming  (continued)**

**Passing primitives vs. objects to methods**

Remember that passing primitives to a method can never alter the variable passed in because Java has no reference operator like C++.   
Specifically the following example of int swap does not work in Java:

|  |
| --- |
|  |
| // Futile attempt to swap ints. Can't work! Java has no pass by reference and no address of operator  public class Futile  {  public static void main()  {  int x,y;  x=5;  y=10;  swap( x, y );  System.out.println("after swap x: " + x + " y: " + y );  }  // this swap method must be static because main is static. You may not call a non static method  // from a static context.  public static void swap( int a, int b )  {  int temp = a;  a = b;  b = temp;  }  } //EOF |

Arrays and objects are non primitives. As such their reference variables may be passed to methods, allowing the method to modify the data that the reference points to. This mechanism is called pass by reference although Java really has no reference operator or address of operator.

To set up our reference passing  discussion we introduce another class. The Account class.

**Account.java**

|  |
| --- |
| // Account class: stripped down version, no setters  public class Account  {  private String name;  private double balance;  public Account(String name, double balance)  {  this.name = name;  this.balance = balance;  }  public String getName()  {  return name;  }  public double getBalance()  {  return balance;  }  public void deposit(double amount)  {  balance += amount;  }  public void withdrawal(double amount)  {  if (balance >= amount)  balance -= amount;  // else handle NSF condition  }  // transfers funds OUT of this account To another account  public void transferTo( Account other, double amount )  {  if (balance >= amount)  {  other.deposit( amount ); // deposit into other account  withdrawal( amount ); // withdrawal from this account  }  // else handle NSF condition  }  public String toString()  {  return name + " $" + balance ;  }  } |

And a test program to illustrate passing object to methods:

[**AccountTransfer.java**](http://www.cs.pitt.edu/~hoffmant/401/OOP-2/examples/AccountTransfer.java)

|  |
| --- |
| // Account Transfer passes one account object into a  // member method of another account object  public class AccountTransfer  {  public static void main(String args[])  {  // initilize a couple Accounts  Acccount cust1 = new Account("Michael",100.00);  Account cust2 = new Account("Joel",200.00);  // echo accounts after initialization  System.out.println(cust1);  System.out.println(cust2);  System.out.println("Xferring $50.00 from " + cust1.getName() + " to " + cust2.getName() );  cust1.transferTo(cust2, 50.00);    // echo accounts after transfer  System.out.println(cust1);  System.out.println(cust2);  } //END main  } |

**Composition**

Once new data types are defined,  objects of that type can be declared as data members of other classes just like primitives are used.

[**JointTest.java**](http://www.cs.pitt.edu/~hoffmant/401/OOP-2/examples/JointTest.java)

|  |
| --- |
| public class JointTest  {  public static void main(String args[])  {  Account cust1 = new Account("Mike",100.00);  Account cust2 = new Account("Sue",200.00);  Joint partners = new Joint(cust1,cust2);  partners.print();  System.out.println("'Joint' balance is: " + partners.getBalance());  }  }  // Notice we include our class defintion INSIDE the same filer as the App  // this is not-typical but it saves space.  class Joint  {  private Account first; // Composition - data member is itself a class  private Account second; // Composition - data member is itself a class  public Joint(Account f, Account s)  {  first = f;  second = s;  }  public double getBalance()  {  return first.getBalance() + second.getBalance();  }  public void print()  {  System.out.println(first);  System.out.println(second );  }  } |

**Inheritance**

Inheritance is a major OOP concept. It allow reuse of code by allowing one class to extend the definition of a previous class. Typically the existing class is called the **parent** class, and the derived or extended class is called the **child** class.

As usual we start with a simple example:  
  
[**SpecialAccountTest.java**](http://www.cs.pitt.edu/~hoffmant/401/OOP-2/examples/SpecialAccountTest.java)

|  |
| --- |
| public class SpecialAccountTest  {  public static void main(String args[])  {  SpecialAccount cust1 = new SpecialAccount("J",100.00,.10);  System.out.println(cust1.getName());  cust1.deposit(100.00);  System.out.println(cust1.getBalance());  System.out.println(cust1.getPercent());  }  }  // Again for space saving - we put our class def in soam file as main App  class SpecialAccount extends Account  {  double percent;  public SpecialAccount(String name,double balance,double percentc)  {  super(this.name,this.balance); // calling parent C'Tor  this.percent = percent;  }  public double getPercent()  {  return percent;  }  } |

It is the keyword **extends** that causes the new child class to inherit everything the parent has. The keyword **super** calls the parent's constructor. Just like the keyword call **this(..)** to call a sibling constructor, **super(..)** must be the first line in the child class' constructor if you want to call your parent's constructor.

If you want to prevent you class from being extended use the keyword final before the word class to disable anyone from using it as a parent class.

public final MyClass // no one can extend this class

{

// ....

}

**Public, Private, Protected, Package**

For a good explanation of the access specifier keywords we go directly to [Sun's Online Reference](http://java.sun.com/docs/books/tutorial/java/javaOO/accesscontrol.html)

**Polymorphism**

Inheritance has more advantages than just code re-use.  Inheritance allows binding of overwritten methods to be resolved dynamically at run-time.

[**PolyTest.java**](http://www.cs.pitt.edu/~hoffmant/401/OOP-2/examples/PolyTest.java)

|  |
| --- |
| /\* Polytest.java - demonstrates Polymorphism by deriving child and grandchild  classes, then overloading print method of child and grandchild.  \*/  public class PolyTest // Polytest is out main Appp  {  public static void main(String args[])  {  A a[] = { new A(), new B(), new C() };  for (int i = 0; i < 3; i++)  a[i].print();  }  } // END PolyTest app  //------------------------------------------------------------------------  // We combine our class definitions in same file as main App to save space  class A  {  A() { }  void print(){System.out.println("A"); }  }  class B extends A  {  B( ){ }  void print(){ System.out.println("B"); }  }  class C extends B  {  C(){}  void print(){ System.out.println("C");}  } |

You may be looking at the above example and wondering "Why do we need polymorphism? " After all you could have just declared 3 totally unrelated classes, given them all print method and then invoked the print of each class.  The convenience of Polymorphism is that all the classes share the is-a relationship to the base class. As such we can declare an array of these classes. Note, you can't declare an array of different types. Thus every element in the array is-a Type A object. It is not until it is referenced at runtime that the object is distinguished to be an A ,B or C object and the proper print method resolved.

The advantages are numerous. You can pass a reference to any of the types into a method expecting the base type.

**The Object Class**

The Object class is the ancestor of all classes in Java. Every class in Java or every class you write, it is as if you wrote **extends Object**after every class prototype.

Since all classes descend from Object, there are a few notable methods that all classes inherit from Object. The first two inherited methods need to be overwritten by your class definition if they are to have any functionality. The last one is implemented generically

* public boolean equals( Object o)
* public String toString()
* public final native Class getClass()

To overwrite equals and toString you simple write our own version of those those two methods in your class. definition.

**Abstract Classes**

Often a class is designed with the expectation that it only make sense to use this class to derive extended classes from but to instantiate classes directly from this type. Typically this base class characterizes behavior that derived classes will implement in their own special way.  A **Shape** class could describe behavior for various different kinds of shapes, and each derived class such as**Circle,** **Square**and **Rectangle** would provide specific different methods to implement the behaviors.

**An Abstract class provides a base class that is suitable for derivation but cannot be directly instantiated.**

|  |
| --- |
| abstract class Shape  {  public abstract double area();  public abstract double perimeter();  } |

Notice now that our abstract class also has abstract methods declared inside. This means that any class designer than derives from Shape must also override area() and perimeter(). Not every abstract class need have abstract methods, but if a class has abstract methods, those methods must be over written in the extended class.

|  |
| --- |
| Shape s = new Shape(); // ILLEGAL! Can't instantiate an abstract class |

[**TestShapes.java**](http://www.cs.pitt.edu/~hoffmant/401/OOP-2/examples/TestShapes.java)

|  |
| --- |
| /\* TestShapes.java defines an abstract Shape class then defines Circle and Square  classes based on Shape.  \*/  public class TestShapes  {  public static void main(String args[])  {  Circle c = new Circle( 3.0 );  System.out.println("Circle C has area: " + c.area() );  System.out.println("Circle C has perimeter: " + c.perimeter() );  Square s = new Square( 10.0 );  System.out.println("Square S has area: " + s.area() );  System.out.println("Square S has perimeter: " + s.perimeter() );  } // END main  } // END TestShapes  // ----------------------------------------------------------------------------------  // We define our Shape, Circle & Square classes in the same file as our App  // ----------------------------------------------------------------------------------  // ..................................................................................  abstract class Shape  {  public abstract double area();  public abstract double perimeter();  }  //...................................................................................  class Circle extends Shape // OK - we derive from Shape  {  double radius; // we added this data member to our abstract base defintion - OK  Circle( double radius )  {  this.radius = radius;  }  public double area() // REQURIED override of abstract parent's area  {  return( Math.PI \* radius\*radius );  }  public double perimeter() // REQUIRED override of abstract parent's perimeter  {  return( 2.0 \* Math.PI \* radius );  }  }  // ..................................................................................  class Square extends Shape // OK - we derive from Shape  {  private double side; // we added this data member to our abstract base defintion - OK  Square( double side )  {  this.side = side;  }  public double area() // REQURIED override of abstract parent's area  {  return( side \* side );  }  public double perimeter() // REQUIRED override of abstract parent's perimeter  {  return( 4.0 \* side );  }  } |

**Static members**

Often is is needful to define data in a class that serves the class at large rather than any particular object of the class.  The **static** keyword overrides the separate storage of data for each object and causes only one instance of the member to be created.  This static  member (data or method) exists at runtime even before any object  of the class has been instanced.

[**StaticTest.java**](http://www.cs.pitt.edu/~hoffmant/401/OOP-2/examples/StaticTest.java)

|  |
| --- |
| /\* StaticTest.java defines a class with a static data member which gets incremented  every time an object of the class is constructed.  \*Note: we get to the static variable in Foo class via the classname Foo  NOT via a ref var of the class  \*/  public class StaticTest  {  public static void main(String args[] )  {  System.out.println("numObjects before any created: " + Foo.numObjects );  Foo f1 = new Foo();  Foo f2 = new Foo();  Foo f3 = new Foo();  Foo f4 = new Foo();  System.out.println("numObjects after some objects created: " + Foo.numObjects );  } // END main  }  // we put our class defintion inside our app file again  class Foo  {  public static int numObjects;  Foo()  {  ++numObjects; // increment static object counter  System.out.println("Just constructed Foo object " + Foo.numObjects );  }  } |

**Interfaces**

In C++ a class definition can inherit from multiple parents. This is multiple inheritance and it is not directly supported in Java.  Java does however have the **interface** keyword which provides a way to inherite the interface from more than one parent.

An interface is like an abstract class except it cannot have any code bodies for methods. Instead we see what looks like C or C++ style function prototypes. The class that implements this interface must provide real code for those prototypes.

A class can implement multiple interfaces.

[**TestDrawable.java**](http://www.cs.pitt.edu/~hoffmant/401/OOP-2/examples/TestDrawable.java)

|  |
| --- |
| /\* TestDrawable.java  define an inteface and implement that interface in a class  \*/  interface Drawable  {  public final static int BLUE = 1, RED = 2;  void setColor(int c);  void setPosition(double x, double y);  void draw();  } //END interface Drawable  // by implementing Drawable we are required to define the code bodies for  // setColor, setPosition, and draw inside this class  class DrawableCircle extends Circle implements Drawable  {  private int color;  private double xpos, ypos;  public DrawableCircle(int radius)  {  super(radius); // just called our Parent (Circle's) C'Tor  System.out.println("Just constructed Drawable Circle");  }  public void setColor(int c)  {  color = c;  System.out.println("set Color");  }  public void setPosition(double x, double y)  {  xpos = x; ypos = y;  System.out.println("Set Position");  }  public void draw()  {  System.out.println("Draw");  }  } // END DrawableCircle class  // ================ App with Main method ==================  public class TestDrawable  {  public static void main(String args[])  {  DrawableCircle c = new DrawableCircle(5);  System.out.println(c.area());  System.out.println(c.perimeter());  c.setColor(Drawable.BLUE);  c.setPosition(1.5, 2.5);  c.draw();  }  } |

**Inner Classes**

Inner classes are nested classes that server the class they are enclosed in. They have access to all the members of the class they live in because they typically are created to server the enclosing class. Inner classes can be static or non-static. Static inner classes have access to only static members, while non static have access to all members of the enclosing class.

[**Top.java**](http://www.cs.pitt.edu/~hoffmant/401/OOP-2/examples/Top.java)

|  |
| --- |
| public class Top // this is the TOP level class  {  private int j = 5; // note this is non-static variable  public static void main(String[] args)  {  Top.Static sic = new Top.Static();  Top tl = new Top();  Top.NonStatic nsc = tl.new NonStatic();  }  public static class Static // static Inner class  {  static int x = 6;  Static()  {  System.out.println("Static Constructor just fired");  // Can't access non static 'j'  }  }  public class NonStatic // non static Inner class  {  NonStatic()  {  System.out.println("Value: " + j);  }  }  } |

**Summary**

**A typical class definition follows these rules:**

* contained in a file of same name as the class but with ".java" added
* has 1 or more private data fields
* has 1 or more public constructors to initialize objects of the class
* has a copy constructor to allow initialization to a value of another object of the type
* has a set and get method for each private field
* has a toString method for convenient print of the object state

**Once a class definition file has been written - the class name represents a new data type.**

* the class name becomes another data type name and objects (variables) of the class type can be declared just like with other data types
* the **new** keyword must be used to call the object's constructor
* more than 1 object of the class type may be declared - just like with other data types
* each object constructed gets its own copy of the fields and methods

Classes can inherit from classes via the keyword extend  
Classes can declare member in 1 or 4 access modes: public, private, protected, package  
Inheritance enables polymorphism which is runtime binding of reference to overridden methods  
Java supports interfaces to effect multiple inheritance