CS 535 Object-Oriented Programming & Design
Fall Semester, 2000
Doc 10 this, super, Classes & Inheritance
Contents

References

The Java Programming Language, 2 ed., Arnold & Gosling, Chapter 2 & 10

The Java Language Specification, Gosling, Joy, Steele, 1996

Surviving Object-Oriented Projects, Alistair Cockburn, Addison Wesley, 1998

Some Observations

Retraining in object-think swamps all other costs.

Ideal object designer/programmer

- Thinks abstractly
- Deals well with uncertainty
- Communicates reasonably

These are from Surviving Object-Oriented Projects

Initializing Fields

There are three ways in Java to give a field an initial value:

Direct Assignment

Instance Initialization Block

Constructors

Direct Assignment

```
public class  BankAccount
{
   public  float  balance = 0.0F;

   public void deposit( float amount )
   {
      balance += amount ;
   }

   public String toString()
   {
      return "Account Balance: " + balance;
   }
}
```

Whenever a BankAccount object is created from the above class, the balance field will be set to 0.0F

Initializing Fields
Instance Initialization Blocks

Instance initialization blocks are indicated by blocks of code inside the class, but outside any method.

Whenever an object is created from a class the code in each instance initialization block is executed. If there is more than one instance initialization block they are executed in order, from top to bottom of the class. Use initialization blocks when the initialization cannot be done in a simple assignment and needs no extra input parameters. Direct assignment and constructors are used far more often than initialization blocks.

```
public class  TaxAccount
{
   public  double  balance;

   {  //Instance Initialization block
      float baseTaxRate = .33F;
      float companySize = 1.24F;

      balance = baseTaxRate * companySize -
            Math.sin( baseTaxRate ) + .013f;
   }

   public static void main( String[] args )      //Test method
   {
      TaxAccount x = new TaxAccount();
      System.out.println( x.balance);
   }
}
```

Output

```
0.09815697215174707
```

Initializing Fields - Constructors

Constructors have the same name as their class

Constructors have no return type

Constructors have zero or more arguments

Constructors are not methods! You can only call a constructor using "new ClassName"

Constructors are called when objects are created (using "new")

new BankAccount(arg1, arg2) will call the BankAccount constructor with two arguments

Like methods, the argument list used (actual parameters) must match the constructors definition (formal parameters)

Implicit Constructors

If a class has no constructor the compiler generates an implicit constructor with no arguments for the class. If a class has any constructor, the compiler will not generate the implicit constructor. This can cause some confusion. Start with a working class with no constructor. Now add a constructor to the class. It compiles with no problem. But code that uses the class will no longer compile since it used the implicit constructor, which is no longer provided. If you add a constructor to an existing class, you may need to also add a constructor with no arguments.

```
public class  ImplicitConstructorOnly  {
   int  size  =  5;
}

public class  OneConstructor  {
   OneConstructor(  String  message  )  {
      System.out.println(  message  );
   }
}

public class  TwoConstructors  {
   TwoConstructors  (  String  message  )  {
      System.out.println(  message  );
   }

   TwoConstructors  (  )  {
      System.out.println(  "No argument Constructor"  );
   }
}

public class  Constructors  {
   public static void main( String args[] ) {
   ImplicitConstructorOnly  ok  =  new  ImplicitConstructorOnly();

   TwoConstructors  alsoOk  =  new  TwoConstructors();
   OneConstructor  compileError  =  new  OneConstructor();
   }
}
```

Order of Initialization

When you create an object the direct assignment of fields and instance initialization blocks are done in order from top to bottom of class, then the constructor is executed

```
public class  OrderExample
{
   int aField = 1;
   {
      System.out.println( "First block: " + aField );
      aField++;
   }

   public OrderExample()
   {
      System.out.println( "Start Constructor: " + aField );
      aField++;
      System.out.println( "End Constructor: " + aField );
   }

   {
      System.out.println( "Second block: " + aField );
      aField++;
   }

   public  static  void  main( String  args[] )
      {
      OrderExample test = new OrderExample();
      }
}
```

Output

```
First block: 1
Second block: 2
Start Constructor: 3
End Constructor: 4
```

Order of Class Elements

Layout

The compiler will let you order the element of a class nearly any order!

Fields must be declared before they are used in an initialization block

Human readers require some consistency

Guidelines suggest placing all field declaration in one location

place all fields at the beginning of the class

or

place all fields at the end of the class

Place all static methods together

Place all instance methods together

Place all

public instance methods together
protected instance methods together
package instance methods together
private instance methods together

Overloading Methods

The signature of a method is its name with number, type and order of its parameters.

The return type is not part of the signature of a method.

Two methods in the same class can have the same name if their signatures are different.

```
public class OverLoad
{
   public void same() {
      System.out.println( "No arguments" );
   }

   public void same( int firstArgument ) {
      System.out.println( "One int arguments" );
   }

   public void same( char firstArgument ) {
      System.out.println( "One char arguments" );
   }

   public int same( int firstArgument ) {         // Compile Error
      System.out.println( "One char arguments" );
      return 5;
   }

   public void same( char firstArgument, int secondArgument) {
      System.out.println( "char + int arguments" );
   }

   public void same( int firstArgument, char secondArgument ) {
      System.out.println( "int + char arguments" );
   }
}
```

Overloading and Signature

Signatures of method calls are determined statically. The actual type of object passed to a method is not used in determining which method to call. The declared type of parameters are used in selecting which version of foo (below) is called.

```
import java.util.Vector;

public class Parent
   {
   public void foo( Object a)
      {
      System.out.println( "Parent object");
      }

   public void foo( Vector a)
      {
      System.out.println( "Parent vector");
      }

   public static void main(String args[])
      {
         Parent test = new Parent();
         Vector container = new Vector();
         Object trick = container;
         test.foo( container );
         test.foo( trick );
      }
}
```

Output

```
Parent vector
Parent object
```

this
(not that)

"this" refers to the current object. It is normally used to return self from method and pass the current object as a parameter from the current object. Java's "this" differs from C++'s "this". The difference occurs with inheritance.

In the example below there are two uses of "this". The first use of this "this.balance = initialBalance" is not needed. "this.balance" refers to the field named "balance". In the first constructor, replacing "this.balance" with "balance" would not change the behavior of the code. The second use of this "this.balance = balance" is required. Since the second constructor has a parameter named "balance", in the second constructor the name "balance" refers to the argument, not the field. So "balance = balance" in this constructor would not set the value of the field. Since "this.balance" refers to the field, the statement "this.balance = balance" does set the value of the field to the value of the parameter.

```
public class  BankAccount
{
   public  float  balance;

   public BankAccount( float initialBalance )
   {
      this.balance = initialBalance;
   }

   public BankAccount( int balance )
   {
      this.balance = balance;
   }

   public void deposit( float amount )
   {
      balance += amount ;
   }

   public String toString()
   {
      return "Account Balance: " + balance;
   }
}
```

Returning this

Since the deposit method returns "this" which is the current object, we can nest calls to deposit. richStudent.deposit(100F).deposit(200F) is read from left to right. First deposit(100F) is sent to the object referred to by richStudent. This method returns "this", the richStudent object with the increased balance. Then deposit(200F) is sent to the object returned by the first deposit method, i.e. the richStudent object. This would not work if deposit did not return "this".

```
public class  BankAccount
{
   public  float  balance;

   public BankAccount( float initialBalance )
   {
      this.balance = initialBalance;
   }

   public BankAccount deposit( float amount )
   {
      balance += amount ;
      return this;
   }
}

public class RunBank
{
   public static void main( String args[] )
   {
      BankAccount richStudent  =  new BankAccount( 10000F );

      richStudent.deposit( 100F ).deposit( 200F ).deposit( 300F );

      System.out.println( "Student: " + richStudent.balance );
   }
}
```

Output

```
Student: 10600.0
```

this as Parameter
A Convoluted Contrived Example

This example shows how a BankAccount object can sent itself as a parameter of a method to another object. In this case the BankAccount object adds itself to a list of customers with bad balances.

```
public class CustomerList {
   public BankAccount[] list = new BankAccount[ 100 ];
   public int nextFreeSlot = 0;

   public void add( BankAccount newItem ){
      list[ nextFreeSlot++ ] =  newItem;
   }
}

public class  BankAccount  {
   public  float  balance;

   public BankAccount( float initialBalance ) {
      this.balance = initialBalance;
   }

   public void badBalanceCheck( CustomerList badAccounts ) {
      if ( balance <= 0F ) badAccounts.add( this );
   }
}

public class RunBank {
   public static void main( String args[] ) {
      BankAccount richStudent  =  new BankAccount( 10000F );
      CustomerList customersToDrop = new CustomerList();
      richStudent.badBalanceCheck( customersToDrop );
   }
}
```

this and Chaining Constructors

"this" with an argument list as first line of a constructor will call another constructor of the same class with the given parameters. The call to "this" must be the first statement in the constructor. The example below has the first two constructors calling another constructor. The changing done here (the first constructor calling the second, which calls the third) is common. The last constructor does all the work. The other constructors just determine default values for some of the parameters. This keeps all the actual work in one place and avoids cutting-and-pasting code between the constructors. Note the name clash in the last constructor. There are two origins in the constructor. One is the name of a field, the other is the name of a parameter. The local name takes precedence. One has to use this.origin to access the field.

```
public class Rectangle
{
   Point origin;
   float width;
   float height;

   public Rectangle()
   {
      this( new Point( 0, 0) );
   }

   public Rectangle( Point origin)
   {
      this( origin, 0, 0 );
   }

   public Rectangle( Point origin, float width, float height)
   {
      this.origin = origin;
      this.width = width;
      this.height = height;
   }
}
```

Calling Methods before calling a "this" Constructor

You can call a static method before calling a constructor with "this". A bit restrictive at times, but it is the rule of the language.

```
public class MethodCallInConstructor
   {
   private int value;

   public MethodCallInConstructor( int first, int second, int third )
      {
      this( nonstaticAdd( first, second, third ) );  // compiler error
      }

   public MethodCallInConstructor( int first, int second )
      {
      this( add( first, second ) );       // OK, can call static method
      System.out.println( "2 arguments: Value = " + value );
      }

   public MethodCallInConstructor( int first )
      {
      value = first;
      System.out.println( "1 argument: Value = " + value );
      }

   public int nonstaticAdd( int a, int b, int c )
      {
      return a + b + c;
      }

   public static int add( int a, int b )
      {
      return a + b;
      }
   }
```

Inheritance
Class Object

All classes inherit directly or indirectly from java.lang.Object

```
class  Parent  { int  size; }
```

Is short hand for

```
class  Parent  extends  Object  { int  size; }
```

The child class below is a grandchild of Object

Having a common ancestor class allows Java to provide standards on all objects, like toString()

```
class  Parent  { int  size; }

class Child extends Parent { int age; }

class  TestObject  {

   public static void main( String args[] ) {

      Parent  watchThis  =  new Parent();

      int  myHash  =  watchThis.hashCode();

      System.out.println(  myHash  );

         // Where does hashCode come from?
   }
}
```

Object's Methods (minus thread related)

clone()
Creates a clone of the object.

equals(Object)
Compares two Objects for equality.
Uses "==" to test for equality

finalize()
Code to perform when this object is garbage collected.

getClass()
Returns the Class of this Object.

hashCode()
Returns a hashcode for this Object.

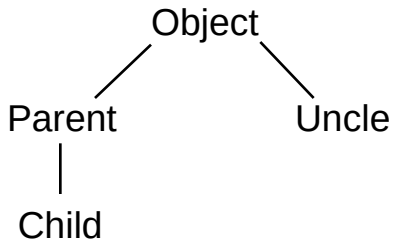
toString()
Returns a String that represents the value of this Object.

If a class needs an implementation of equals, which differs from the default "equals", the class should override both equals and hashCode

If two different objects satisfy the equals method, the hashCode should return the same value for both objects

Casting and Classes

An instance of a child class can be assigned to a variable (field) of the parent class. If a variable references an object of a subclass, the object can be cast down to its actual type with an explicit cast. The explicit cast is required. A runtime error occurs when explicitly casting an object to a type that it is not. In the code below the cast "(Uncle) object" is a runtime error because at that time object holds an instance of the Child class, which is not of type (or subclass) of Uncle. An object of type Parent cannot be cast to type Child.




```
class Parent { int data; }

class Child extends Parent { String name; }

class Uncle { String rich; }

class Casting {
   public static void main( String args[] ) {
      Object object;
      Parent parent;
      Child child = new Child();
      Uncle uncle;

      parent = child;
      object = child;
      parent = (Parent) object;     // explicit cast down
      child = (Child) object;       // explicit cast down

      uncle = (Uncle) object;       //Runtime exception
      }
}
```

Output

```
java.lang.ClassCastException: Child: cannot cast to Uncle
```

Casting Rules

Let "x" and "Y" be references (fields, variable, etc.).
Let "x" be of class X,
Let "y" be of class Y,
Let "y" reference object "a" of class A.

The statement "x = y;" will compile if and only if

class X and class Y are the same class or
class X is an ancestor (parent, grandparent, etc.) of class Y

The statement "x = (X) y;" will compile if and only if
class X and class Y are the same class or
class X is an ancestor of class Y or
class Y is an ancestor of class X

The statement "x = (X) y;" will execute at runtime if and only if
class X and class A are the same class
class X is an ancestor of class A

Inheritance and Name Clashes

What happens when the parent class has a method(field) with the same name as a method(field) of the child class?

Terms and Rules

Overloading

Providing more than one method with the same name, but with different signatures

Overriding

A class replacing an ancestor's implementation of a method with an implementation of it own

Signature and return type must be the same

Hiding Fields

Giving a field in a class the same name as a field in an ancestor hides the ancestor's field

The field exists, but can not be accessed by its short name

When invoking a method on an object, the ***actual type*** of the ***object*** is used to determine which method implementation to use. Method references are determined dynamically. You determine which method will be used at runtime by examining the actual object you are sending the method to.

When accessing a field in an object, the ***declared type*** of the ***reference*** is used to determine which field to use. Field references are determined statically. You determine which field will be used by looking at the source code.

Hiding Fields

A parent and child class can both have non-static fields with the same name. An instance of the child class will always contain both fields, even if access levels on the parent field do not allow the child to access the parent's field directly. Determine which "name" will be used statically.

```
class  Parent  {
   public String  name  =  "Parent";
   public  void  parentPrint() { System.out.println(  name  ); }
}

class  Child  extends  Parent  {
   String  name  =  "Child";
   public  void  print()  { System.out.println(  name  ); }
}

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

      Child  whoAmI  =  new Child();

      whoAmI.print();
      whoAmI.parentPrint();

      System.out.println( whoAmI.name );
      System.out.println( ( (Parent ) whoAmI).name );
   }
}
```

Output

```
Child
Parent
Child
Parent
```

Overriding Methods

Note that the statement "overRidden.print();" prints out different text depending on what type of object overRidden references.

```
class  Parent  {
   public void print() {
      System.out.println( "In Parent" );
   }
}

class  Child  extends  Parent  {
   public  void  print()  {
      System.out.println( "In Child" );
   }
}

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

      Child  whoAmI  =  new Child();
      whoAmI.print();

      Parent overRidden = whoAmI;
      overRidden.print();
      overRidden = new Parent();
      overRidden.print();
   }
}
```

Output

```
In Child
In Child
In Parent
```

Why not the same rule for Methods and Fields?

- Resolving overridden methods dynamically gives us polymorphism
- Resolving hidden fields can not be done dynamically

Resolving hidden fields dynamically would allow child classes to break parent method that refer to parent fields. Assume that hidden fields are found dynamically. Let a child class create a field of the same name, but different type as parent's field. Calling a parent's method that uses the field through a child's instance would break the method. The field used would now be the wrong type. If fields were resolved dynamically, then in the statement " trouble.test();" the test method would try to divide a string by 3, which is not allowed in Java.

```
class  Parent  {
   public  int  name  =  5;

   public  void  test() {
      System.out.println( "In Parent \t" + (name / 3) );
   }
}

class  Child  extends  Parent  {
   public String name  =  "Can't divide me";
}

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

      Child  trouble  =  new Child();
      trouble.test();
   }
}
```

Overriding and Signature

Signature of a method call is determined statically

To override a method use the same signature as parent method

```
public class Point
   {
   // Does not overload Object.equals method
   public boolean equals( Point p)
      {
      return distance(p ) == 0;
      }


   // Overloads Object.equals method
   public boolean equals( Object aPoint)
      {
      if  (aPoint instanceof Point)
         return distance((Point) aPoint ) == 0;
      else
         return false;
      }
   }
```

This Again

this

- references to fields via this are resolved statically
- references to methods via this are resolved dynamically

That is to say, "this" follows the normal rules.

```
class Parent {
   String name = "Parent";

   public void print() {System.out.println( "In Parent" + name ); }

   public void thisFunction() {
      System.out.println( this.name );
      this.print();
   }
}

class Child extends Parent {
   String name = "Child";

   public void print() { System.out.println( "In Child " + name ); }
}

class This {
   public static void main( String args[] ) {
      Parent parentThis = new Child();
      parentThis.thisFunction();

      Child childThis = new Child();
      childThis.thisFunction();
   }
}
```

Output

```
Parent
In Child Child
Parent
In Child Child
```

Overloading, Overriding and Return Types

When overriding a method, the child's method must have the same signature and return type as the parent's method. In C++ if the parent class overloads a method name (i.e. print) and the child class overrides one of these methods, then the child can not directly access any of the overloaded parent's methods.

```
class  Parent  {
   public void print() {
      System.out.println( "In Parent" );
   }

   public void print( String message ) {
      System.out.println( "In Parent" + '\t' + message );
   }

   public void print( int value ) {
      System.out.println( "In Parent" + '\t' + value );
   }
}

class  Child  extends  Parent  {
   public  int  print()  {                    // Compiler Error
      System.out.println( "In Child" );
      return 5;
   }

   public void print( String message ) {
      System.out.println( "In Child" + '\t' + message );
   }
}

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

      Child  whoAmI  =  new Child();
      whoAmI.print();
      whoAmI.print( "Hi Mom" );
      whoAmI.print( 10 );                    // Ok in Java,
                                                  // Compile error in C++
   }
}
```

Super

super

Refers to the superclass of class that implements the method containing the reference to super

All references to super are resolved statically

To find out what class super refers to go to the source code, find the code that contains "super", then go to that classes parent class.

```
class Parent {   String name = "Parent"; }

class Child extends Parent {
   String name = "Child";

   public void print() {
      System.out.println( name );
      System.out.println( super.name );
   }
}

class GrandChild extend Child {
   String name = "GrandChild";
}

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

      GrandChild whoAmI = new GrandChild();
      whoAmI.print();
   }
}
```

Output

```
Child
Parent
```

super.super

Java does not allow you to chain supers to refer to your grandparent class.

The need to reference a grandparent's class should be very rare

Constructors and Inheritance

Before the constructor in a Child class is called, its parent's constructor will be called

If the Child class constructor does not do this, the parent's constructor with no arguments will be implicitly called

super(arg list) as the first line in a constructor explicitly calls the parent's constructor with the given signature

If

parent class implements a constructor with arguments &

parent class does not implement a constructor with no arguments

Then the Child constructors must eventually explicitly call a parents constructor

Implicit Call to Parent's Constructor

```
class  Parent  {
   public  Parent()  {
      System.out.println(  "\t In Parent"  );
   }
}

class  Child  extends  Parent  {
   public  Child()  {
      System.out.println(  "\t In Child"  );
   }

   public Child( String notUsed ) {
      System.out.println(  "\t In Child with argument"  );
   }
}

class  Constructors  {

   public static void main( String args[] ) {
      System.out.println(  "Construct Parent"  );
      Parent  sleepy   =  new Parent();

      System.out.println(  "Construct Child"  );
      Child  care  =  new Child();
      System.out.println(  "Construct Second Child"  );
      care  =  new Child( "Try Me" );
   }
}
```

Output

```
Construct Parent
   In Parent
Construct Child
   In Parent
   In Child
Construct Second Child
   In Parent
   In Child with argument
```

Explicit Call to Parent's Constructors

```
class  Parent  {

   public  Parent( )  {
      System.out.println(  "In Parent, No Argument"  );
   }

   public  Parent(  String  message  )  {
      System.out.println(  "In Parent"  +  message  );
   }
}

class  Child  extends  Parent  {

   public  Child( String  message,  int  value  )  {
      this(  message  );                        // if occurs must be first
      System.out.println(  "In Child"  );
   }

   public  Child( String  message  )  {
      super(  message  );        // if occurs must be first
      System.out.println(  "In Child"  +  message  );
   }
}

class  Constructors  {
   public static void main( String args[] ) {
      System.out.println(  "Construct Child"  );
      Child  care  =  new Child( ">Start from Child<", 5 );
   }
}
```

Output

```
Construct Child
In Parent>Start from Child<
In Child>Start from Child<
In Child
```

No Default Parent Constructor

The compiler will not generate the default constructor for the class "Parent", since "Parent" has a constructor. The class "Child" makes an implicit call to the default constructor in "Parent". This causes a compile error.

```
class  Parent  {

   public  Parent(  String  message  )  {
      System.out.println(  "In Parent"  +  message  );
   }
}

class  Child  extends  Parent  {

   public  Child(   )  {                        // Compile Error
      System.out.println(  "In Child"  );
   }

   public  Child( String  message  )  {
      super(  message  );
      System.out.println(  "In Child"  +  message  );
   }
}
```

Nonstatic Methods in Constructors are Resolved Dynamically

Methods called in constructors follow the rules.

```
class Parent
   {
   public Parent( )
      {
      whichOne( );
      }

   public void whichOne( )
      {
      System.out.println( "In Parent" );
      }
   }

class Child extends Parent
   {
   public void whichOne( )
      {
      System.out.println( "In Child" );
      }
   }

class Test
   {
   public  static  void  main( String  args[] )
      {
      new Child( );
      }
   }
```

Output

```
In Child
```

Static Methods

Static methods references are resolved statically

```
class  Parent  {

   public  static  void  classFunction()  {
      System.out.println(  "In Parent"  );
   }

   public  static  void  inheritMe()  {
      System.out.println(  "Inherited"  );
   }
}

class  Child  extends  Parent  {

   public  static  void  classFunction()  {
      System.out.println(  "In Child"  );
   }
}

class  StaticTest  {

   public static void main( String args[] ) {

      Parent  exam  =  new Child();
      exam.classFunction();

      Child  question  =  new Child();
      question.classFunction();
      question.inheritMe();
   }
}
```

Output

```
In Parent
In Child
Inherited
```

Protected Access and Inheritance

protected

- Accessible in the package that contains the class
- Accessible in all subclasses

```
package Botany;

public class Protected {
   protected String name = "Protected";
}

package Botany;

public class NoRelation {
   public  void  SampleAccess( Protected accessOK )  {
      accessOK.name  =  "This is legal";
   }
}

package  Tree;

public class NoRelationEither {
   public  void  SampleAccess( Botany.Protected noAccess)  {
      noAccess.name  =  "This is a compile Error";
   }
}

package  Tree;

public class Child  extends  Botany.Protected {
   public  void  SampleAccess( Botany.Protected noAccess,
                               Child  accessOK)  {
      name = "This is legal, I am related";
      noAccess.name  =  "This is a compile Error";
      accessOK.name  =  "This is legal";
   }
}
```

Protected Access and Inheritance: The Rules[1]

Let C be a class with a protected member or constructor.

Let S be a subclass of C in a different package.

The declaration of the use of the protected member or constructor occurs in S.

If the access is of a protected member, let Id be its name

> If the access is by a field access expression of the form super.ID, then the access is permitted
>
> If the access is by a qualified name Q.Id, where Q is a type name, then the access is permitted if and only if Q is S or a subclass of S.
>
> If the access is by a qualified name Q.Id, where Q is an expression then the access is permitted if and only if the type of the expression Q is S or a subclass of S
>
> If the access is by a field access expression E.Id, then the access is permitted if and only if the type of E is S or a subclass of S

[1]From The Java Language Specification, Gosling, Joy, Steele, 1996, section 6.6.2, page 100

Package Access and Inheritance

Package Access

Accessible in the package that contains the class

Not accessible outside the package that contains the class

```
package Botany;

public class NoExplicit {
   String name = "No explicit access level given";
}

package Botany;

public class NoRelation {
   public  void  SampleAccess( NoExplicit accessOK )  {
      accessOK.name  =  "This is legal";
   }
}

package  Tree;

public class NoRelationEither {
   public  void  SampleAccess( Botany.NoExplicit noAccess)  {
      noAccess.name  =  "This is a compile Error";
   }
}

package  Tree;

public class Child  extends  Botany.NoExplicit {
   public  void  SampleAccess( Botany.NoExplicit noAccess,
                               Child  alsoNoAccess)  {
      name = "I am related, but this is NOT LEGAL";
      noAccess.name  =  "This is a compile Error";
      alsoNoAccess.name  =  "This is a compile Error";
   }
}
```

Inheritance of Access Levels

The access modifier of an overriding method must provide at least as much access as the overridden method

Note

A private method is not accessible to subclasses, so can not be overridden in the technical sense.

Thus, a subclass can declare a method with the same signature as a private method in one of its superclasses. There is no requirement that the return type or throws clause of the method bear any relationship to those of the private method in the superclass

```
class Parent
   {
   protected void foo()  {};
   public void bar() {};
   private void noSeeMe() {};
   }

class Child extends Parent
   {
   public void foo() {};                // OK
   protected void bar() {};             // Compile Error
   public int noSeeMe() { return 5;};     //OK

   }
```

Rules Are Different for Fields

Since fields are hidden, not overridden we can set the access permission on fields in the class "Child" independent of the access permissions of the fields in class "Parent"

```
class Parent
    {
    protected int foo;
    public int bar;
    }

class Child extends Parent
    {
    public int foo;          // OK
    protected int bar;       // OK
    }
```

Inheritance and Final

The final modifier provides:

- Security
- Performance optimizations

A class declared final, can not have any subclasses

```
final  class  EndOfTheLine  {

   int  noSubclassPossible;

   public  void  aFunction()  {
      System.out.println(  "Hi Mom"  );
   }
}

class  ThisWillNotWork  extends  EndOfTheLine  {
   int  ohNo;
}
```

Does not compile

Final Method

A final method can not be overwritten

```
class  Parent  {

   public  final  void  theEnd()  {
      System.out.println(  "This is it"  );
   }

   public  void  normal()  {
      System.out.println(  "In parent"  );
   }
}

class  Child  extends  Parent  {

   public  void  theEnd()       {                  // Compile Error
      System.out.println(  "Two Ends?"  );
   }

   public  void  normal()  {
      System.out.println(  "In Child"  );
   }
}
```

Abstract Classes

```
abstract class NoObjects {

   int noDirectObjectPossible = 5;

   public void aFunction() {
      System.out.println( "Hi Mom" );
   }

   public abstract void subClassMustImplement( int foo );
}

class Concrete extends NoObjects {

   public void subClassMustImplement( int foo ) {
      System.out.println( "In Concrete" );
   }
}

class AbstractClasses {

   public static void main( String args[] ) {
      NoObjects useful = new Concrete();
      Concrete thisWorks = new Concrete();

      useful.subClassMustImplement( 10 );

      System.out.println( thisWorks.noDirectObjectPossible );
   }
}
```

Output

```
In Concrete
5
```

Relationships between Classes

Is-kind-of or **is-a**
is-analogous-to
is-part-of or **has-a**

Is-kind-of, is-a, is-a-type-of

Let A and B be classes

To determine if A should be the parent class of B ask:

Is B an A (or is B a type of A, or is B a kind of A)

If the answer is no, then **do not** make A the parent of B

If the answer is yes, then **consider** making A the parent of B

Example 1 Bank Accounts

A JuniorSavingsAccount is a type of a SavingsAccount

So, consider making JuniorSavingsAccount a subclass of SavingsAccount

Is-kind-of, is-a, is-a-type-of
Example 2 Employees

Consider Employee, Manager, Engineer, and FileClerk

A Manager is a kind of an Employee

So should we make Employee a parent class of a Manager class?

Is Manager a class or a role that an Employee object will play?

> Does Manager contain data or operations that Employee does not?
>
> If no, then Manager is likely a role for an Employee object

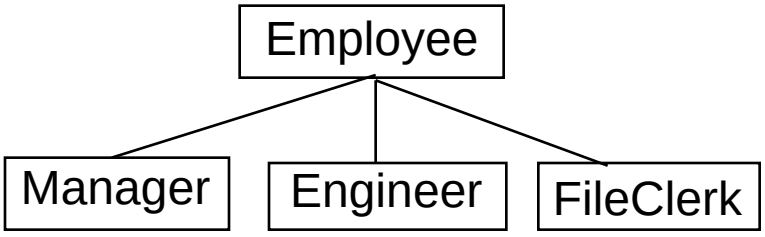
A very common mistake is to use inheritance to model roles that an object of one class will take on in a program. Just because an employee object will take on the role of an engineer, does not mean we need an engineer class. What happens when Smith changes positions from an engineer to a manager?

```
class Employee { //lots of stuff here not shown }

class Driver
   {
   public static void main( String[] args )
      {
      Employee manager = new Employee( "Sally" );
      Employee engineer = new Employee( "Smith" );

      }
   }
```

Example 2 Employees - Continued

If Manager, etc are classes then several options:

Option 1 Direct inheritance

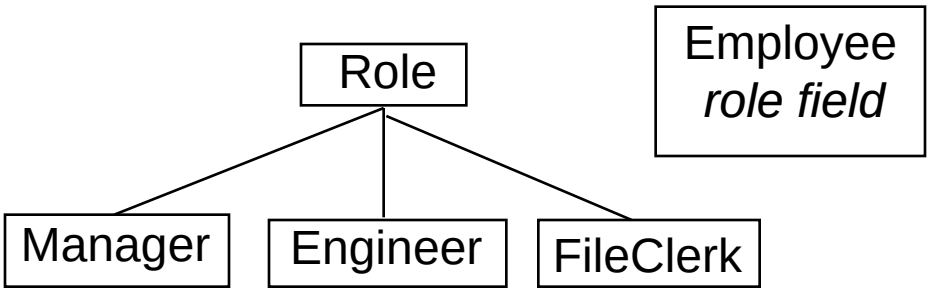



```
class Employee
   {
   // stuff not shown
   }

class Manager extends Employee
   {
   // stuff not shown
   }
```

etc.

This option is not very flexible. What happens when an employee changes from one type of position to another? See Option 2

Option 2 Separate Inheritance Structure

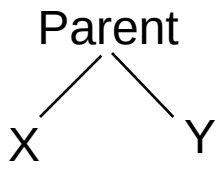



```
class Employee
    {
    Role position;

    // stuff not shown
    }

class Role
    {
    // stuff not shown
    }

class Manager extends Role
    {
    // stuff not shown
    }
```

This is a much more flexible approach. It is very easy for an employee object to change roles. Compare this to option 1.

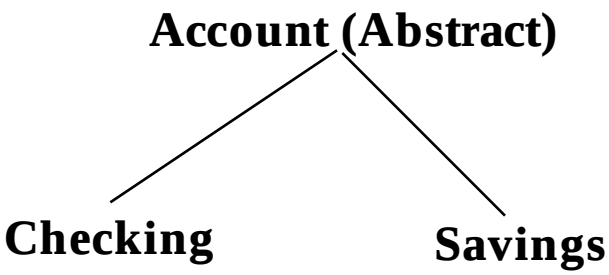
is-analogous-to

If class X is-analogous-to class Y then look for superclass



Example BankAccounts

A checking account is analogous to a saving account



is-part-of or **has-a**

If class A is-part-of class B then there is no inheritance

Some negotiation between A and B for responsibilities may be needed

Example: Linked-List class and a Stack

A stack uses a linked-list in its implementation

A stack has a linked-list

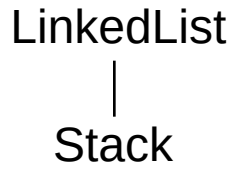
The Stack class and the LinkedList class are separate classes

Stack LinkedList

```
class LinkedList
   {
   public void addFront( Object item) { // not shown}
   public Object removeFront() { // not shown}
   public Object removeEnd() { // not shown}
   }

class Stack
   {
   LinkedList   stackElements = new LinkedList();
   public void push( Object item )
      {
      stackElements.addFront( item);
      }
   }
```

A Common Mistake

Using inheritance with a is-part-of (has-a) relation between classes




```
class LinkedList
   {
   // fields not shown
   public void addFront( Object item) { // not shown}
   public Object removeFront() { // not shown}
   public Object removeEnd() { // not shown}
   }
class Stack extends LinkedList
   {
   public void push( Object item )
      {
      addFront( item);
      }
   }
class Test
   {
   public static void main( String[] arguments)
      {
      Stack problem = new Stack();
      problem.push( "Hi");
      problem.push( "Mom" );
      problem.removeEnd();    // ????
      }
   }
```