

JN3025-Inheritance

Groovy enables one class to extend another, just as interfaces can, though classes extend at most one class. We can test for extended classes with the 'in' operator, just like with implemented interfaces:

```
class A{}  
class B extends A{}  
def b= new B()  
assert b in B && b in A  
  
class A1{}  
class A2{}  
//class C extends A1, A2{}  
//compile error when uncommented: a class can extend at most one class
```

Public instance fields, properties, and methods defined on an extended class are also available on the extending class:

```
class A{  
    public int prev //field  
    int signature //property  
    String sayPies(int n){ "There are ${prev= signature= n} pies!" } //method  
}  
class B extends A{  
    String sayBirds(int n){ "There are $n birds!" }  
}  
def b= new B()  
assert b.sayBirds(17) == 'There are 17 birds!'  
assert b.sayPies(11) == 'There are 11 pies!'  
    //method sayPies(int) from A acts as part of B  
assert b.prev == 11 //field 'prev' from A acts as part of B  
b.signature= 19  
assert b.signature == 19 //property 'signature' from A acts as part of B  
assert b.getSignature() == 19
```

We can use the 'private' and 'protected' modifiers to restrict the visibility of instance methods, properties, and fields:

```

class A{
    //private methods, properties, and fields are not visible outside the class,
    //even in inheriting classes...
    private int prevPies
    private String sayPies(int n){ "There are ${prevPies= n} pies!" }

    //protected methods, properties, and fields are visible in inheriting
    //classes (and within the same package)...
    protected int prevBeans
    protected String sayBeans(int n){ "There are ${prevBeans= n} beans!" }
}

class B extends A{
    def testAccesses(){
        assert sayPies(23) == 'There are 23 pies!'
            //Groovy bug: this private method shouldn't be visible here
        try{ prevPies; assert 0 }catch(e){ assert e in MissingPropertyException }
            //A's private field 'prevPies' not visible here

        assert sayBeans(29) == 'There are 29 beans!'
            //A's protected method visible here in an inheriting class
        assert prevBeans == 29
            //A's protected field visible here in an inheriting class
    }
}

def b= new B()
assert b.sayPies(11) == 'There are 11 pies!'
    //Groovy bug: this private method shouldn't be visible here
try{ b.prevPies; assert 0 }catch(e){ assert e in MissingPropertyException }
    //A's private field 'prevPies' not visible here

assert b.sayBeans(14) == 'There are 14 beans!'
    //this protected method is visible here in the same package it's defined in
assert b.prevBeans == 14
    //this protected field is visible here in the same package it's defined in

b.testAccesses()

```

Public static fields, properties, and methods are inherited by extending classes:

```

class A{
    static public int numBananas //field
    static signature //property
    static String sayBananas(int n){ //method
        "There are ${numBananas= signature= n} bananas!"
    }
}
class B extends A{}

assert A.sayBananas(23) == 'There are 23 bananas!' //method call
assert A.numBananas == 23 //field access
assert A.signature == 23 //property accesses
assert A.getSignature() == 23

assert B.sayBananas(23) == 'There are 23 bananas!' //method call

assert B.numBananas == 23 //field access
assert B.signature == 23 //property access
B.getSignature() == 23 //property access using method syntax

```

We can make static methods, properties, and fields private or protected:

```

class A{
    static private int numBananas= 0
    static private String sayBananas(int n){
        "There are ${numBananas= n} bananas!"
    }
    static protected int numApples= 0
    static protected String sayApples(int n){
        "There are ${numApples= n} apples!"
    }
}
class B extends A{
    static testAccesses(){
        assert sayBananas(37) == 'There are 37 bananas!'
        //numBananas //compile error when uncommented:
        //A's private field not visible here

        assert sayApples(29) == 'There are 29 apples!'
        numApples //A's protected field visible here in an inheriting class
    }
}
assert B.sayBananas(31) == 'There are 31 bananas!'
try{ B.numBananas; assert 0 }catch(e){ assert e in MissingPropertyException }
assert B.sayApples(23) == 'There are 23 apples!'
assert B.numApples == 23

B.testAccesses()

```

We can define what's called an "abstract class", a class with only some methods defined, the others being only declarations just like in interfaces. An abstract class and each method declaration in it must be modified with the keyword 'abstract':

```

interface X{
    def x()
}
interface Y{
    def y()
}
abstract class A{
    def a(){ println 1 } //method definition
    abstract b()          //declaration of method only
}
class B extends A implements X, Y{
    def x(){ println 2 }
    def y(){ println 3 }
    def b(){ println 4 } //declared method from abstract class A defined here
}

```

Whether a method is static or not is part of its definition, not its declaration. So interface and abstract methods may not be declared static.

```

interface X{
    def x()
    //static x1() //error when uncommented: interface methods can not be static
}
interface Y{
    def y()
}
abstract class A{
    static a(){ println 1 }
    abstract b()
    abstract c()
    //abstract static c1()
    //error when uncommented: abstract methods can not be static
}
class B extends A implements X, Y{
    static x(){ println 2 }
    def y(){ println 3 }
    static b(){ println 4 }
    def c(){ println 5 }
}

```

At the other end from abstract classes and methods are "final classes" and "final methods". A final class may not be extended; a final method may not be overridden:

```

class A{
    final a(){ 11 }
    def b(){ 12 }
}
final class B extends A{
    //def a(){ 15 } //compile error when uncommented: can not override final A.a()
    def b(){ 16 }
}
//class C extends B{} //compile error when uncommented: can not extend final C

```

Constructors

Just as a class's constructor can call another constructor at the beginning of its code, so also it can call a constructor on the superclass at the beginning of its code:

```

class A{
    def list= []
    A(){
        list<< "A constructed"
    }
    A(int i){
        this()
        list<< "A constructed with $i"
    }
}
class B extends A{
    B(){
        list<< "B constructed"
    }
    B(String s){
        super(5) //a constructor can call its superclass's constructor if it's
                //the first statement
        list<< "B constructed with '$s'"
    }
}

def b1= new B('kea')
assert b1.list.collect{it as String} == [
    "A constructed",
    "A constructed with 5",
    "B constructed with 'kea'",
]
def b2= new B()
assert b2.list == [
    "A constructed",
    //default parameterless constructor called if super() not called
    "B constructed",
]

```

Using Classes by Extending Them

Some classes supplied with Groovy are intended to be extended to be used. For example, `FilterInputStream`, `FilterOutputStream`, `FilterReader`,

and FilterWriter:

```
//When not extended, FilterOutputStream simply passes its method calls to the
//wrapped stream...
try{
    def fos= new FilterOutputStream(new FileOutputStream('abc.txt'))
    fos.write(33i)
    fos.write([34,35,36] as byte[])
    fos.write([34,35,36,37,38,39,40] as byte[], 3, 2)
    fos.close()
    def fis= new FilterInputStream(new FileInputStream('abc.txt'))
    def ba= new byte[6]
    fis.read(ba)
    assert ba.toList() == [33,34,35,36,37,38]
}

//We can extend FilterOutputStream to provide the logic for the filter...
class EvenNumberOutputStream extends FilterOutputStream{
    EvenNumberOutputStream(OutputStream out){
        super(out)
    }
    def write(int i){
        if(i%2 == 0) super.write(i) //call method of same name in the super-class
    }
    def write(byte[] ba){
        super.write( ba.toList().findAll{ it%2 == 0 } as byte[] )
    }
    def write(byte[] ba, int start, int size){
        this.write( ba[start..<(start+size)] as byte[] )
        //another way to call method of same name in same class definition
    }
}
try{ //...then call the methods...
    def fos= new EvenNumberOutputStream(new FileOutputStream('abc.txt'))
    fos.write(33i)
    fos.write([34,35,36] as byte[])
    fos.write([34,35,36,37,38,39,40] as byte[], 3, 2)
    fos.close()
    def fis= new FilterInputStream(new FileInputStream('abc.txt'))
    def ba= new byte[6]
    fis.read(ba)
    assert ba.toList() == [34,36,38,0,0,0]
}
```

We can similarly extend FilterInputStream, FilterReader, and FilterWriter.

The Object Hierarchy

All classes are arranged in a hierarchy with `java.lang.Object` as the root. Here are those we've met so far; those labelled as such are abstract and final classes:

```
java.lang.Object
    java.lang.Boolean (final)
    java.lang.Character (final)
    java.lang.Number (abstract)
```

```
java.lang.Integer (final)
java.lang.Long (final)
java.math.BigInteger
java.math.BigDecimal
java.lang.Short (final)
java.lang.Byte (final)
java.lang.Float (final)
java.lang.Double (final)
java.math.MathContext (final)
java.util.Random
java.util.Date
java.util.TimeZone (abstract)
    java.util.SimpleTimeZone
java.util.Calendar (abstract)
    java.util.GregorianCalendar
groovy.time.BaseDuration (abstract)
    groovy.time.Duration
    groovy.time.TimeDuration
    groovy.time.DatumDependentDuration
        groovy.time.TimeDatumDependentDuration
java.util.AbstractCollection (abstract)
    java.util.AbstractList (abstract)
        java.util.ArrayList
        groovy.lang.Sequence
        groovy.lang.IntRange
        groovy.lang.ObjectRange
    java.util.AbstractSet (abstract)
        java.util.HashSet
        java.util.TreeSet
java.util.AbstractMap (abstract)
    java.HashMap
    java.util.LinkedHashMap
    groovy.lang.SpreadMap
    java TreeMap
java.util.Collections
java.lang.String (final)
java.lang.StringBuffer (final)
java.util.regex.Pattern (final)
java.util.regex.Matcher (final)
groovy.lang.GroovyObjectSupport (abstract)
    groovy.lang.Binding
    groovy.lang.Closure (abstract)
    groovy.lang.GString (abstract)
    groovy.util.Expando
java.text.Format (abstract)
    java.text.NumberFormat (abstract)
        java.text.DecimalFormat
    java.text.DateFormat (abstract)
        java.text.SimpleDateFormat
java.text.DecimalFormatSymbols
java.text.DateFormatSymbols
java.io.File
java.io.InputStream (abstract)
    java.io.ByteArrayInputStream
    java.io.FileInputStream
    java.io.FilterInputStream
        java.io.BufferedInputStream
        java.io.DataInputStream
        java.io.LineNumberInputStream
```

```
    java.io.PushbackInputStream  
    java.io.SequenceInputStream  
    java.io.StringBufferInputStream  
java.io.OutputStream (abstract)  
    java.io.ByteArrayOutputStream  
    java.io.FileOutputStream  
    java.io.FilterOutputStream  
        java.io.BufferedOutputStream  
        java.io.DataOutputStream  
        java.io.PrintStream  
java.io.Reader (abstract)  
    java.io.BufferedReader  
        java.io.LineNumberReader  
    java.ioCharArrayReader  
    java.io.FilterReader (abstract)  
        java.io.PushbackReader  
    java.io.InputStreamReader  
        java.io.FileReader  
        java.io.StringReader  
java.io.Writer (abstract)  
    java.io.BufferedWriter  
    java.io.CharArrayWriter  
    java.io.FilterWriter (abstract)  
    java.io.OutputStreamWriter  
        java.io.FileWriter
```

```
java.io.PrintWriter  
java.io.StringWriter
```