MAHARISHI UNIVERSITY of MANAGEMENT

Engaging the Managing Intelligence of Nature

Computer Science Department

CS390 Foundamental Programming Practices (FPP)

Professor Paul Corazza

Lecture 4 Appendix: Cloning

The clone() Method

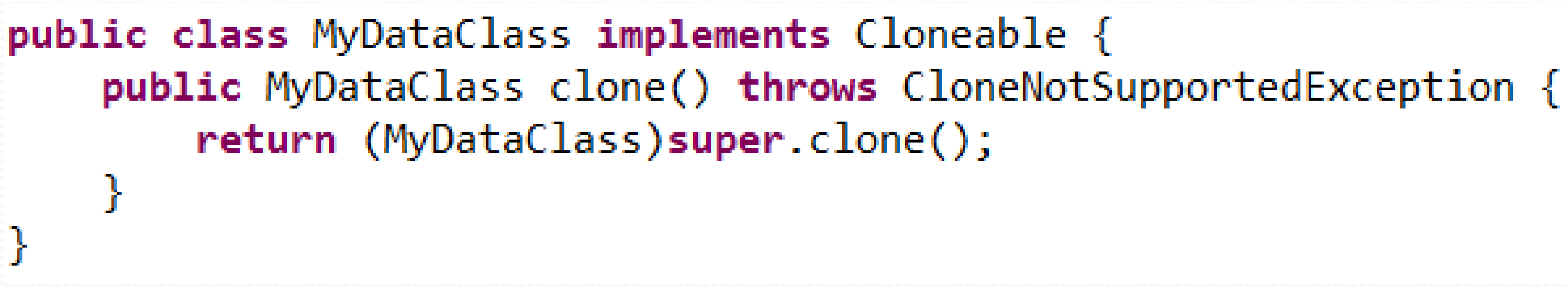
- The following is a method of Object:

```
protected Object clone() throws
      CloneNotSupportedException
```

The `CloneNotSupportedException` is thrown when an attempt is made by an object of type A to perform a cloning operation but A does not implement the `Cloneable` interface.

Access of protected Members from Within a Subclass

The rules governing `protected` allow a subclass to directly access protected members of its superclasses. Here is an example where the superclass is `Object` and `MyClass` is any other class (the subclass). This behavior matches the common understanding of the rules for `protected` members. See the demo lesson4.clonegood.



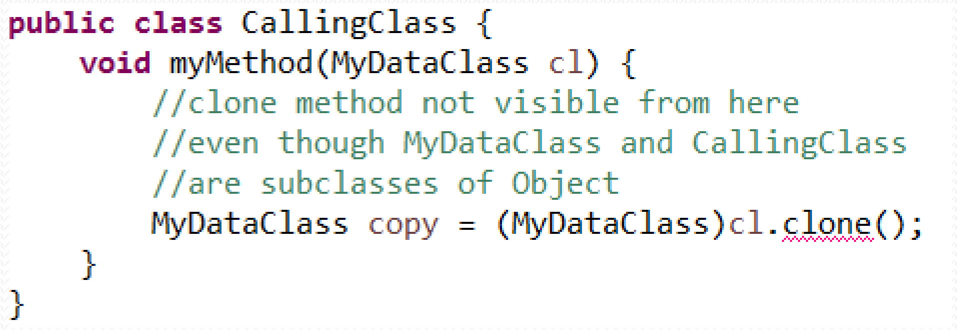
```
public class MyDataClass implements Cloneable {
    public MyDataClass clone() throws CloneNotSupportedException {
        return (MyDataClass)super.clone();
    }
}
```

Attempting to Access protected Members from the Outside

Demo: `lesson4.clonebad`

The following produces a compiler error. The `CallingClass` is attempting to access the `proteced` member of `Object` by using the object reference `cl` of type `MyClass`. Compiler error states that "`clone()` is not visible." Note that `CallingClass` and `MyClass` are both subclasses of `Object`.

```
public class MyDataClass implements Cloneable {
    String name = "harry";
}
```



```
public class CallingClass {
    void myMethod(MyDataClass cl) {
        //clone method not visible from here
        //even though MyDataClass and CallingClass
        //are subclasses of Object
        MyDataClass copy = (MyDataClass)cl.clone();
    }
}
```

Usual Approach to Gain Access from Outside: Override

Demo: See `lesson4.clonegood`

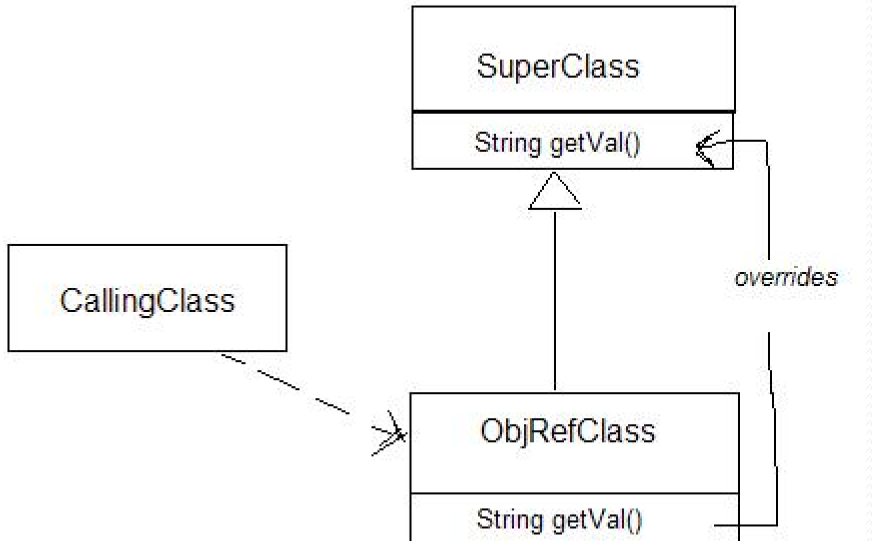
```
public class CallingClass {
    public MyClass tryToClone(MyClass cl) {
        try {
            //ok since clone() is now a public method in MyClass
            return (MyClass) cl.clone();
        } catch(CloneNotSupportedException e) {
            return null;
        }
    }

    public static void main(String[] args) {
        CallingClass cc = new CallingClass();
        MyClass cl = new MyClass();
        MyClass result = cc.tryToClone(cl);
    }
}
//Developer is now able to declare that class is available for cloning
public class MyClass implements Cloneable {
    String name = "harry";

    @Override
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}
```

Use of `protected` in General

Use of the `protected` keyword follows same access rules as we see for the `clone` method:

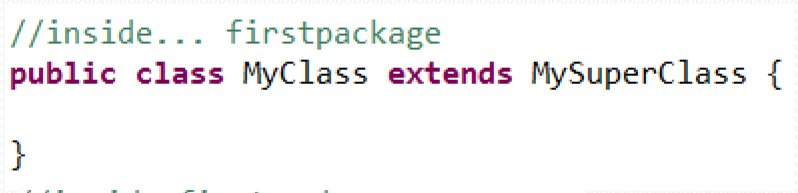
In order to access a `protected` method of superclass

by accessing a subclass, the subclass must explicitly provide access by overriding the superclass method

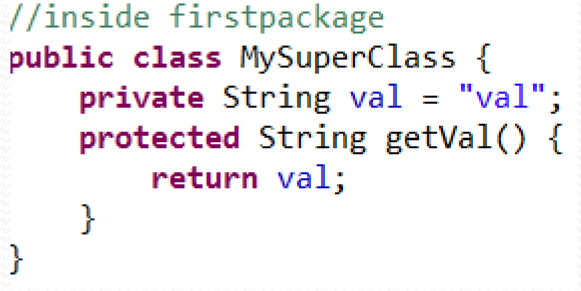


Exception: If the calling class happens to be in the same package as the super class, it can access the superclass method through the subclass even if subclass does not override. See demo lesson4.protectedex.try3.objrefpkg

Exercise 4.5: Working with the `protected` Qualifier

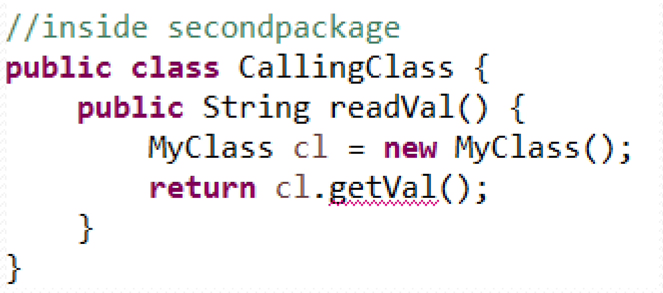


```java
//inside... firstpackage
public class MyClass extends MySuperClass {

}
```



```java
//inside firstpackage
public class MySuperClass {
    private String val = "val";
    protected String getVal() {
        return val;
    }
}
```

The code below generates a compiler error. How can this be fixed?



```java
//inside secondpackage
public class CallingClass {
    public String readVal() {
        MyClass cl = new MyClass();
        return cl.getVal();
    }
}
```

Shallow Copies

1. The default version of the clone() method creates a *shallow copy* of an object. A shallow copy of an object will have an exact copy of all the fields of the original object. If the original object has any references to other objects as fields, then only *references* of those objects are copied into the clone object; *copies* of those objects are not created. That means any changes made to those objects through the clone object will be reflected in original object, and vice-versa.
2. Demo: `lesson4.clone.shallowcopy`
3. A shallow copy is good for copying primitives and immutable objects, but other object references still point to the original objects; this behavior is not usually desirable.

```
public class Job implements Cloneable {
	int numhours;
	String typeOfJob;
	public Job(int n, String t) {
		numhours = n;
		typeOfJob = t;
	}
	@Override
	public Object clone() throws CloneNotSupportedException {
		//shallow copy is fine here - variables are primitive
		//or immutable
		return (Job)super.clone();
	}
	public String toString() {
		return typeOfJob + ": " + numhours;
	}
}
public class Person implements Cloneable {
	String name;
	Job job;
	public Person(String name, Job j) {
		this.name = name;
		job = j;
	}
	public String toString() {
		return "name: " + name + ", job: [" + job + "]";
	}
	@Override
	public Object clone() throws CloneNotSupportedException {
		//shallow copy not fine here: Job in copy is same as
		//Job in original
		return (Person) super.clone();
	}
}
```

```java
public class Main {
   public static void main(String[] args) {
      Job joesjob = new Job(40, "Carpenter");
      Person joe = new Person("Joe", joesjob);
      System.out.println(joe);
      try {
         Person joecopy = (Person)joe.clone();
         System.out.println(joecopy);
         joecopy.job.typeOfJob = "Painter";
         //modifies original object!
         System.out.println(joe);
      } catch(CloneNotSupportedException e) { }
   }
}
```

Producing Deep Copies

1. A deep copy of an object will have an exact copy of all the fields of the original object, just like a shallow copy. But in addition, if the original object has any references to other objects as fields, then a copy of each of those objects is also created when clone() is called. That means the clone object and the original object will be 100% disjoint and 100% independent of each other. None of the changes made to the clone object will be reflected in the original object; none of the changes made to original object will be reflected in the copy either.
2. Demo: `lesson4.clone.deepcopy`
3. A *deep copy* is produced by separately cloning all the object instance variables in the class to be cloned and inserting them into the clone.

```
public class Job implements Cloneable {
    int numhours;
    String typeOfJob;
    public Job(int n, String t) {
        numhours = n;
        typeOfJob = t;
    }
    @Override
    public Object clone() throws CloneNotSupportedException {
        //shallow copy is fine here - variables are primitive
    //or immutable
        return (Job)super.clone();
    }
    public String toString() {
        return typeOfJob + ": " + numhours;
    }
}
public class Person implements Cloneable {
    String name;
    Job job;
    public Person(String name, Job j) {
        this.name = name;
        job = j;
    }
    public String toString() {
        return "name: " + name + ", job: [" + job + "]";
    }
    @Override
    public Object clone() throws CloneNotSupportedException {
        //creates a deep copy
        Person pcopy = (Person) super.clone();
        pcopy.job = (Job) job.clone();
        return pcopy;
    }
}
```

```java
public class Main {
  public static void main(String[] args) {
     Job joesjob = new Job(40, "Carpenter");
     Person joe = new Person("Joe", joesjob);
     System.out.println(joe);
     try {
        Person joecopy = (Person)joe.clone();
        System.out.println(joecopy);
       joecopy.job.typeOfJob = "Painter";
        //does not modify orig object!
        System.out.println(joe);
     } catch(CloneNotSupportedException e) {  }
   }
}
```

Shallow Copy vs Deep Copy In Java

Shallow Copy	**Deep Copy**
Clone Object and original object are not 100% disjoint.	Clone Object and original object are 100% disjoint.
Any changes made to clone object will be reflected in original object and vice versa.	No changes made to clone object will be reflected in original object and vice versa.
Default version of clone method creates a shallow copy of an object.	To create a deep copy of an object, you have to override clone method.
Shallow copy is preferred if an object has only primitive and immutable fields.	Deep copy is preferred if an object has references to other mutable objects as fields.
Shallow copy is fast and also less expensive.	Deep copy is slower and more expensive.

Summary

1. Inheritance provides subclasses with access to data and methods that may be inaccessible to other classes.

2. Inheritance supports polymorphism, which makes it possible to perform operations on many different types by performing those operations on just one supertype.

3. Inheritance must be used wisely; the IS-A and LSP criteria provide guidelines for when subclassing can be used safely.

4. Java interfaces provide even more abstraction of classes, and also support polymorphism.

5. As of Java 8, static and default methods may be included in a Java interface.

6. Java's Reflection library makes it possible at runtime for objects to instantiate classes based only on their name and constructor argument types, and to examine the structure of objects at runtime. These tools can provide a powerful addition to OO programming techniques; they play a fundamental role in the design of modern frameworks, like Spring.

7. The Object class is the single root of the inheritance hierarchy that includes all Java classes. Consequently, every Java class, including user-defined classes, automatically inherits several methods defined in Object: equals, hashCode, toString, and clone. Whenever these methods are needed, their default implementation in Object typically needs to be overridden by in user-defined classes.

Connecting the Parts of Knowledge With the Wholeness of Knowledge

Using Reflection to create objects at the level of "name"

1. Ordinarily, an object of a certain type is created in Java by calling the constructor of the class that realizes this type. This is object construction on the level of *form*.
2. Java's Reflection API allows the developer to construct on object based on the knowledge of the name (and the number and types of arguments required by the constructor). This is object construction on the level of *name.*

---

3. **<u>Transcendental Consciousness:</u>** The fundamental impulses that structure both the name and form of an object have their basis in the silent field of pure consciousness.
4. **<u>Wholeness moving within itself</u>**: In Unity Consciousness, the finest structuring mechanics of creation are appreciated as modes of vibration of the Self.