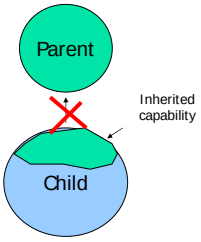
Final and Abstract Classes

Restricting Inheritance



Final Members: A way for Preventing Overriding of Members in Subclasses

- All methods and variables can be overridden by default in subclasses.
- This can be prevented by declaring them as final using the keyword "final" as a modifier. For example:
  - final int marks = 100;
  - final void display();
- This ensures that functionality defined in this method cannot be altered any. Similarly, the value of a final variable cannot be altered.

Final Classes: A way for Preventing Classes being extended

- We can prevent an inheritance of classes by other classes by declaring them as final classes.
- This is achieved in Java by using the keyword final as follows:

```
final class Marks
{ // members
}
final class Student extends Person
{ // members
}
```

- Any attempt to inherit these classes will cause an error.

Abstract Classes

- When we define a class to be "final", it cannot be extended. In certain situation, we want to properties of classes to be always extended and used. Such classes are called Abstract Classes.
- An *Abstract* class is a conceptual class.
- An Abstract class cannot be instantiated – objects cannot be created.
- Abstract classes provides a common root for a group of classes, nicely tied together in a package:

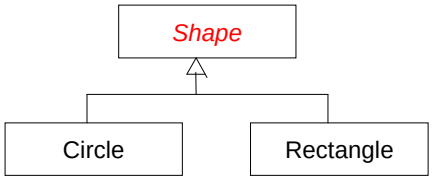
Abstract Class Syntax

```
abstract class ClassName
{
        ...
        ...
        abstract Type MethodName1();
        ...
        ...
        Type Method2()
        {
           // method body
        }
}
```

- When a class contains one or more abstract methods, it should be declared as abstract class.
- The abstract methods of an abstract class must be defined in its subclass.
- We cannot declare abstract constructors or abstract static methods.

Abstract Class -Example

- Shape is a abstract class.



The Shape Abstract Class

```
public abstract class Shape {
      public abstract double area();
      public void move() { // non-abstract method
          // implementation
      }
}
```

- Is the following statement valid?
  - Shape s = new Shape();
- No. It is illegal because the Shape class is an abstract class, which cannot be instantiated to create its objects.

Abstract Classes

```
public Circle extends Shape {
   protected double r;
   protected static final double PI =3.1415926535;
   public Circle() { r = 1.0; )
   public double area() { return PI * r * r; }
…
}
public Rectangle extends Shape {
   protected double w, h;
   public Rectangle() { w = 0.0; h=0.0; }
   public double area() { return w * h; }
}
```


Abstract Classes Properties

- A class with one or more abstract methods is automatically abstract and it cannot be instantiated.
- A class declared abstract, even with no abstract methods can not be instantiated.
- A subclass of an abstract class can be instantiated if it overrides all abstract methods by implementation them.
- A subclass that does not implement all of the superclass abstract methods is itself abstract; and it cannot be instantiated.

Summary

- If you do not want (properties of) your class to be extended or inherited by other classes, define it as a final class.
  - Java supports this is through the keyword “final”.
  - This is applied to classes.
- You can also apply the final to only methods if you do not want anyone to override them.
- If you want your class (properties/methods) to be extended by all those who want to use, then define it as an abstract class or define one or more of its methods as abstract methods.
  - Java supports this is through the keyword “abstract”.
  - This is applied to methods only.
  - Subclasses should implement abstract methods; otherwise, they cannot be instantiated.

Interfaces

Design Abstraction and a way for loosing realizing Multiple Inheritance

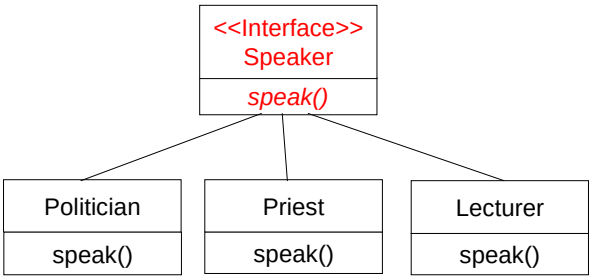
Interfaces

- *Interface* is a conceptual entity similar to a Abstract class.
- Can contain only constants (final variables) and abstract method (no implementation) - Different from Abstract classes.
- Use when a number of classes share a common interface.
- Each class should implement the interface.

Interfaces: An informal way of realising multiple inheritance

- An interface is basically a kind of class—it contains methods and variables, but they have to be only abstract classes and final fields/variables.
- Therefore, it is the responsibility of the class that implements an interface to supply the code for methods.
- A class can implement any number of interfaces, but cannot extend more than one class at a time.
- Therefore, interfaces are considered as an informal way of realising multiple inheritance in Java.

Interface - Example



Interfaces Definition

- Syntax (appears like abstract class):

```
interface InterfaceName {
    // Constant/Final Variable Declaration
    // Methods Declaration – only method body
}
```

- Example:

```
interface Speaker {
    public void speak( );
}
```


Implementing Interfaces

- Interfaces are used like super-classes who properties are inherited by classes. This is achieved by creating a class that implements the given interface as follows:

```
class ClassName implements InterfaceName [, InterfaceName2, …]
{
    // Body of Class
}
```


Implementing Interfaces Example

```
class  Politician implements Speaker {
        public void speak(){
            System.out.println("Talk politics");
        }
}
```

```
class  Priest implements Speaker {
        public void speak(){
            System.out.println("Religious Talks");
        }
}
```

```
class  Lecturer implements Speaker {
        public void speak(){
            System.out.println("Talks Object Oriented Design and Programming!");
        }
}
```

Extending Interfaces

- Like classes, interfaces can also be extended. The new sub-interface will inherit all the members of the superinterface in the manner similar to classes. This is achieved by using the keyword **extends** as follows:

```
interface InterfaceName2 extends InterfaceName1
{
    // Body of InterfaceName2
}
```


Inheritance and Interface Implementation

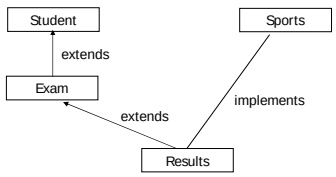
- A general form of interface implementation:

```
class ClassName extends SuperClass implements InterfaceName [,
InterfaceName2, …]
{
    // Body of Class
}
```

- This shows a class can extended another class while implementing one or more interfaces. It appears like a multiple inheritance (if we consider interfaces as special kind of classes with certain restrictions or special features).

Student Assessment Example

- Consider a university where students who participate in the national games or Olympics are given some grace marks. Therefore, the final marks awarded = Exam_Marks + Sports_Grace_Marks. A class diagram representing this scenario is as follow:



Software Implementation

```
class Student
{
    // student no and access methods
}
interface Sport
{
    // sports grace marks (say 5 marks) and abstract methods
}
class Exam extends Student
{
    // example marks (test1 and test 2 marks) and access methods
}
class Results extends Exam implements Sport
{
    // implementation of abstract methods of Sport interface
    // other methods to compute total marks = test1+test2+sports_grace_marks;
    // other display or final results access methods
}
```


Interfaces and Software Engineering

- *Interfaces*, like abstract classes and methods, provide templates of behaviour that other classes are expected to implement.
- Separates out a design hierarchy from implementation hierarchy. This allows software designers to enforce/pass common/standard syntax for programmers implementing different classes.
- Pass method descriptions, not implementation
- Java allows for inheritance from only a single superclass. *Interfaces* allow for *class mixing*.
- Classes *implement* interfaces.

A Summary of OOP and Java Concepts Learned So Far

Summary

- *Class* is a collection of data and methods that operate on that data
- An *object* is a particular instance of a *class*
- *Object members* accessed with the 'dot' (Class.v)
- *Instance variables* occur in each instance of a class
- *Class variables* associated with a class
- Objects created with the *new* keyword

Summary

- Objects are not explicitly 'freed' or destroyed. Java automatically reclaims unused objects.
- Java provides a default constructor if none defined.
- A class may inherit the non-private methods and variables of another class by *subclassing*, declaring that class in its *extends* clause.
- *java.lang.Object* is the default *superclass* for a class. It is the root of the Java *hierarchy*.

Summary

- *Method overloading* is the practice of defining multiple methods which have the same name, but different argument lists
- *Method overriding* occurs when a class redefines a method inherited from its superclass
- *static*, *private*, and *final* methods cannot be overridden
- From a *subclass*, you can explicitly invoke an overridden method of the *superclass* with the *super* keyword.

Summary

- Data and methods may be hidden or encapsulated within a class by specifying the *private* or *protected* visibility modifiers.
- An abstract method has no *method body*. An abstract class contains abstract methods.
- An *interface* is a collection of *abstract methods* and constants. A class *implements* an interface by declaring it in its *implements* clause, and providing a method body for each *abstract method.*