Advanced Programming

Object Orientation and Polymorphism in Java

Overview of Lecture

- Objects and classes
- Method signatures
- Overloading methods
- Overriding super class methods
- Polymorphism
- Dynamic binding

Object Oriented Programming Class

- Object oriented programming uses the class as a blueprint for building objects, the focus is not functional

- Class are used for abstraction

Classes versus objects

- A class is a blueprint for objects

- An object is an instance of a class

Class relationships

- Classes can have relationships:
- Use – one class uses another, e.g. one uses the other as a parameter to a method
- Containment – one class contains objects of another, Team and Player
- Inheritance – specialization, one class is a specialization of another more general class

Building your own class

- First you need some abstraction to model
- Suppose we decided to model a student, then we create a class called Student
- Then we can add attributes like name, age, year

Abstraction of Student

```
public class Student {

   private String name;
   private int age, year;

   public Student(String name, int age, int year) {
         this.name = name;
         this.age = age;
         this.year = year;
   }
}
```

Note: No main method…why?… because this class is not meant to run as an application

Constructor

- Constructors allow objects to be created:

```
public Student(String name, int age, int year) {
     this.name = name;
     this.age = age;
     this.year = year;
}
```

Overloading Constructors

- It’s possible to have more than one constructor in the same class, e.g. :

```
public Student(String name, int year) {
        this.name = name;
        this.year = year;
}

public Student(String name, int age, int year) {
        this.name = name;
        this.age = age;
        this.year = year;
}
```

This would provide <u>TWO</u> ways to create students

Using the Student class

This uses student and is NOT an application with a main method!

```
import java.util.Vector;

public class ClassList {

   private Vector classList = new Vector();

   public ClassList() {
        classList = new Vector();
   }

   public void addToClass(Student student) {
        classList.add(student);
   }

   public Vector getClassList() {
        return classList;
   }
}
```

Vectors

- Note the use of a Vector in the previous slide
- A vector is very like an array in the way that it holds lists of things
- A vector however can hold any object and can be expanded dynamically [arrays need to be set to a particular type and size]

Vectors

- To add to a vector use the *addElement (Object)* method
- To get something from the vector use the *elementAt(int)* method
- To get the size of a Vector uses the *size()* method

Using our ClassList class

An application that uses both Student and ClassList

```
public class SampleStudentGroup {

  public static void main(String[] args) {
       ClassList bn002Started2006 = new ClassList();
       Student student1 = new Student("Carter",22,1);
       Student student2 = new Student("Kelly",21,1);

       bn002Started2006.addToClass(student1);
       bn002Started2006.addToClass(student2);
       Vector sampleClassList = bn002Started2006.getClassList();

       for(int i = 0; i < sampleClassList.size(); i++) {
         Student student = (Student)sampleClassList.elementAt(i);
         System.out.println(student.getName());
       }
  }
}
```

Object Oriented Programming using Java Classes

- The student list example used in the previous slides is an example that applies the object oriented approach to programming using Java classes

- The classes Student and ClassList are abstractions and contain no main method, the SampleStudentGroup simply uses the other two class and is designed to create objects, it is a specific example of an application that uses the classes Student and ClassList

Java Classes

- These simple examples of the HelloWorld and student list have shown how a Java class can be used as a simple application or as a blueprint for objects

- The Student class was a blueprint for student objects (the objects were created using the **new** keyword)

- The HelloWorld and SampleStudentGroup classes were simply applications designed to be executed and produce some behaviour…

Method Signatures

- The signature of method is the combination of the method name combined with the parameter order and parameter types

- Two methods have the same signature if they have the same name and the same parameter types and order

Method Signature

- It does not matter what return type is specified in the method, this is <u>NOT</u> part of the method signature
- The exceptions that are thrown are not included in the method signature either

Method Signature – Example 1

```java
public class SameSignature {

 public int signatureMethod(double x)  {
  //Some code
 }

 public double signatureMethod(double x){
  //Some code
 }
}
```

This generates compile error, two methods with same signature at same level!!!

Method Signature – Example 1

- In the previous slide there are two methods with the same signature (recall: ignore the return type!!!)

- This class cannot compile as the two methods are indistinguishable from each other to the compiler!!!

Method Signature – Example 2

```
public class DifferentSignature {

 public int signatureMethod(int p, double x) {
  return 0;
 }

 public double signatureMethod(double x, int p) {
  return 0;
 }
}
```

The compiler is happy since the order of the parameters is different

Method Signature – Example 2

- The previous example was acceptable to the compiler because the method signatures are different

- The methods have the same name and the same parameter types BUT they have a different parameter order, this makes the methods distinguishable even though they have the same name [this is called **method overloading**]

Method Signature – Example 3

```java
public class ClassOne {

 public int signatureMethod(int x) {
  return 0;
 }
}


class ClassTwo {

 public int signatureMethod(int x) {
  return 0;
 }
}
```

The compiler is happy, signatures are same but they are in different classes

Method Signature – Example 3

- The previous example compiles because even though the two methods have the same signature but they are distinguishable because they belong to two different classes

- The compiler effectively sees two different methods, one called

  ClassOne.signatureMethod(int) and
  ClassTwo.signatureMethod(int)

Overriding superclass methods

- We have shown that it is possible to have two methods in two different classes with the same signature

- It is also possible for a subclass of a superclass to have a method with the same signature, however, this is called <u>overriding</u> the base/super class method since it REPLACES the method in the base/superclass with a more specific version

Overriding superclass method - Example

```
public class SuperClass {
 public int signatureMethod(int p, double x) {
  return 0;
 }
}
public class SubClass extends SuperClass {
 public int signatureMethod(int p, double x) {
  return 0;
 }
}
```

This method has same signature as parent and so has overridden the parent

Overriding superclass method - Example

- In the previous example the subclass has a method with the same signature as the parent

- The effect of this is that the child has rejected the parents implementation of the method and has decided to do it’s own thing…thus overriding the parent

- Overriding **only** happens if the signatures match

Class hierarchies (overriding superclass features)

- Any inherited part of the base/super class can be rejected by the child by inclusion of an overridden version in the child/sub class as long as the signatures match (or in the case of attributes the name and type match)

- If the child does not override the superclass feature then the superclass method/attribute is used for the child/subclass (if the parent method is not abstract of course)

Class Inheritance

- One class can inherit from another directly so that one becomes a child of the other
- The child uses the keywords *extends* and can directly access attributes that are *protected* or *public* or *package*, but cannot access the attributes that are *private*
- The child can also inherit directly the methods that are either *protected*, *public* or *package*, but will not inherit the *private* methods

Inheritance

- So what does inheritance actually mean?

- It means that the child class actually becomes the type of the parent, but that the child becomes a specialized version of the parent class and therefore all methods/attributes of suitable access actually belong to the child as well as the parent…but…it does not work both ways!!!, i.e., any attributes in the child class are not available to the parent!!!

- Think of this phrase "A cat is an animal and therefore behaves like an animal, but not all animals can say meow!!!"

Inheritance example

- Suppose we have a VisitingStudent class in our college system
- What class could this inherit from?…
- What attributes and behaviours do VisitingStudents get through inheritance?
- What specific attributes have VisitingStudents that regular students do not???

Inheritance example: VisitingStudent

```
public class VisitingStudent extends Student {
 private String collegeOfOrigin;

 public VisitingStudent(String name,int age,int year,String
   collegeOfOrigin) {
   super(name,age,year);
   this.collegeOfOrigin=collegeOfOrigin;
 }
}
```

Inheritance Example

- A VisitingStudent “is-a” Student as the VisitingStudent class extends Student class

- A VisitingStudent have all the attributes of a Student (name,age,year), but has a special feature, i.e., collegeOfOrigin

- To create a VisitingStudent we only need one additional piece of information over regular students, i.e., where did they come from

The VisitingStudent constructor

```
public class VisitingStudent extends Student {
  private String collegeOfOrigin;

   public VisitingStudent(String name, int age,
   int       year, String collegeOfOrigin) {
    super(name,age,year); //call Student
   constructor
    this.collegeOfOrigin=collegeOfOrigin;
   }
}
```

Re-use the Student constructor, only set the special attribute

Inheritance Example

- Can we use a VisitingStudent object in our ClassList program from earlier???
- Answer: Absolutely, a VisitingStudent “is a” Student after all!!!
- The next slide shows this!!!

```java
public class SampleStudentGroup {

 public static void main(String[] args) {
 ClassList bn002Started2006 = new ClassList();
 Student student1 = new Student("Carter",22,1);
 Student student2 = new Student("Kelly",21,1);
 VisitingStudent student3 = new VisitingStudent("Craemer",34,1,"UCLA");

 bn002Started2006.addToClass(student1);
 bn002Started2006.addToClass(student2);
 bn002Started2006.addToClass(student3);

 Vector sampleClassList = bn002Started2006.getClassList();

 for(int i = 0; i < sampleClassList.size(); i++) {
  Student student = (Student)sampleClassList.elementAt(i);
  System.out.println(student.getName());
 }
 if(!CollegeHelper.classSizeWithinLimit(sampleClassList.size())) {
  System.out.println("Class not within limits");
 }
 System.out.println("Pass grade is " + CollegeHelper.PASSGRADE);
 }

}
```

No problem!!!

Terminology

- The Student class is the **superclass** (it is also a base class as inherits from no-one)
- The VisitingStudent is a **sub-class** of Student
- The keyword **super** can be used to access features in a superclass from a sub-class

Overidding superclass behaviour

- Sub-classes can override behaviours found in the superclass

- For instance suppose the VisitingStudent class wanted a method called *getName()* [**Exactly the same as the super class **] that also prints the college of origin: “John from NUIM”

Overridden *getName()*

```java
public class VisitingStudent extends Student {
 private String collegeOfOrigin;

 public VisitingStudent(String name, int age, int year,
   String collegeOfOrigin) {
   super(name,age,year);
   this.collegeOfOrigin=collegeOfOrigin;
 }

 public String getName() {
   return super.getName() + " from " + collegeOfOrigin;
 }
}
```

Specialized getName() method!!!, re-uses superclass getName()… didn't have access to the name attribute as it is private

More on keyword *final*

- A method declared as **final** cannot be overridden by inheriting classes!!!
- A class that is declared **final** cannot be inherited!!!
- Both can be useful for preventing inheritance and overriding of methods/ classes that have security implications

Concrete class

- The classes we have seen so far have been classes with full implementation of all methods within the class…classes that implement all methods are called **concrete** classes [since they have concrete implementations]

- However, not all classes contain full implementations, there are some situations where classes can be <u>abstract</u> rather than concrete

Abstract classes

- Abstract classes are defined with the keyword *abstract* in Java

- Abstract classes are classes that NEVER become objects because they are not designed to become objects…they are used most commonly as generalizations

Abstract classes and generalization

- Consider modeling the Animal Kingdom

- What does an animal look like?…this question is so general that we can’t answer it!!!…therefore the concept of an Animal is an abstraction used for human classification

- What we can say is that animals MOVE!!!…but we can’t answer the question ‘how do animals move?’ because it’s too general a question!!!

Abstract classes and generalization

- We cannot create an object of type Animal [new Animal] and neither would we want to!!!

- Now here’s another question: What do Animals look like? …again this is too general…but we might decide that all animals have a name…we can add this as a general attribute of all animals as shown in the next slide.

- Also, it would be useful to ALL animals to be able to report their name using a generic accessor method getName()

Abstract class Animal

```
public abstract class Animal {

   //All animals have a name
   protected String animalName;

   //All animals must move, but how is up to the Animal
   public abstract void move();

   //A useful generic method for retrieving animal name
   public String getName() {
       return animalName;
   }
}
```

Everything in here is general to all animals

One abstract method one implementation

Using the Animal class

- We CANNOT create an Animal object:

  Animal shep = new Animal(); //NONSENSE!!!!
  //Shep’s a dog, i.e., a specific type of Animal!!!

- So how do we use the Animal class?…answer: We specialize the Animal class through inheritance [in Java use keyword *extends*]

Creating a specialized class

- A dog is a Mammal…answer the same questions for Mammal as we did Animal to find generalizations:
  - What do Mammals look like? [dunno??]
  - What do all Mammals do? [lactate..give milk!!]
- So there's an argument here for creating another abstract class called Mammal that extends the Animal class, because we still have no concrete implementation

Mammal class

```
public abstract class Mammal extends Animal {

  public abstract void move();

  public void lactate() {
  //if female and has young then...
  //produce milk
  }
}
```

- But we cannot say how all Mammals move, and we cannot create a Mammal object, however, we can implement the *lactate()* method general to all Mammals

Concrete classes of Animal

- At this point we have a specific enough class hierarchy to describe a Dog!!!

- Now we can create a concrete Dog class that extends Mammal which in turn inherits all the generic attributes and behaviours of Animal

- However, we can now implement to *move()* method for Dog class, since they all walk!!

Dog class

```
public class Dog extends Mammal {

   private String dogName;
   //Constructor for Dogs that are pets
   public Dog(String animalName,String dogName) {
        this.animalName = animalName;
        this.dogName = dogName;
   }

   //For dogs that are not pets
   public Dog(String animalName) {
        this.animalName = animalName;
   }

   public void move() {
        //Go walkies!!
   }
}
```

Where did the attribute called animalName come from?

This is implemented here and was abstract in the super class

Animals

- Of course we could keep this going forever and have lots of different types of animals each with a specialized way of moving, e.g., bounce, hop, slither etc.
- And we could even *extend* the dog class if needed!!!
- It is possible to have a generic constructor in the base class if it's appropriate [arguable for the Animal class hierarchy]

Creating my dogs

```
public class MyDogs {

  public static void main(String[] args) {
      Dog dog1 = new Dog("Jack Russell","Junior");
      Dog dog2 = new Dog("Labrador","Jake");
      Dog dog3 = new Dog("Labrador","Jeff");
      Dog dog4 = new Dog("Labrador","MJ");

      //This getName method comes from the base
  class     //Animal, also assuming I coded
  getDogName!!!
      System.out.println(dog1.getName());
      System.out.println(dog1.getDogName());
  }
}
```

Polymorphism

- Given that it is possible to override a superclass method with a subclass method of the same signature, it’s pretty clear that there can be many forms of the same method in an inheritance hierarchy
- This is given a the rather salubrious title in OOP, i.e., “Polymorphism” which literally just means “many forms”
- “Polymorph” has also been used in many sci-fi’s for example in Star Trek, Terminator and Red Dwarf to label aliens that can change form, i.e., “many forms”

Polymorphism Example

```
public abstract class Gun {

   protected String serialNumber;

   public abstract String getReloadInstructions();

   public String getSerialNumber(){
        return serialNumber;
   }

}
```

Polymorphism Example

- The example in the previous slide is a generic description of a Gun, there are many different types of gun in the world: “many forms”

- Each specific type of gun is loaded differently so the description of how to load a gun is very specific to each gun type

Polymorphism Example

```
public class Revolver extends Gun {

  public Revolver(String serialNumber) {
   this.serialNumber = serialNumber;
  }

  public String getReloadInstructions() {

   String desc = "Reload for " + this.getClass().getName();
   String step1 =  "Pop open the revolving chamber and load
   bullets";
   String step2 = "Pop the revolving chamber back in place";

   return desc + "\n" + step1 + "\n" + step2;
  }
}
```

Revolver class

- The revolver class overrides the Gun abstract method *getReloadInstructions*
- The Revolver class, therefore, implements polymorphism as there are now two forms of the *getReloadInstructions* method
- The Revolver class DOES NOT OVERRIDE the *getSerialNumber* method because the generic method is acceptable to the subclass

SemiAutomatic Class

```java
public class SemiAutomatic extends Gun {

   public SemiAutomatic(String serialNumber) {
    this.serialNumber = serialNumber;
   }

   public String getReloadInstructions() {
    String desc = "Reload for " + this.getClass().getName();
    String step1 = "Remove magazine";
    String step2 = "Pop bullets into magazine";
    String step3 = "Replace magazine";

    return desc + "\n" + step1 + "\n" + step2 + "\n" + step3;
   }
}
```

Polymorphism in Gun hierarchy

- There are many forms of Gun in this hierarchy
- There are many different forms of the *getReloadInstructions* method
- This is polymorphism…it is essential that the method signatures match, otherwise they are not different forms of the same method, they're just different methods

Creating Gun objects

- To create Gun objects we can do the following:

```
Gun peaceMaker1 = new Revolver("B12345678");
Gun peaceMaker2 = new SemiAutomatic("A12345678");
```

- Note that the Gun reference can refer to objects of the subclass since a Revolver “is-a” Gun

Getting the loading instructions

- Using the object references on the previous page, it’s possible to get the specific loading instructions as follows:

```
peaceMaker1.getReloadInstructions();
peaceMaker2.getReloadInstructions();
```

- Note that the instructions for peaceMaker1 will be for a Revolver and peaceMaker2 will be for a SemiAutomatic since the overridden method is called

Dynamic binding

- The call to the *getReloadInstructions* method on the previous slide was mapped to the subtype when the call was made, i.e., if we can’t see the object created the Java environment looks after the method call and ensures the correct version of the method is called for the correct subtype

- This process is called *dynamic binding* because it binds the class type to the method call at <u>RUNTIME</u>!

Multiple Objects

- The best way to store multiple objects is to use an object container, such as a Vector

- A Vector stores any object type but is not able to store the particular subclass instance of the objects stored, it simply “upcasts” everything to type Object

- Removing objects from the Vector requires a “downcast”…recall the phrase “all cats are animals but not all animals are cats”

Dynamic Binding

- Using Polymorphism and applying dynamic binding combined, it is possible to store multiple types in a Vector and call some common method to all subclasses without checking for the subtype of each object

```
public class Arsenal {

 public static void main(String[] args) {
  Vector munitionsList = new Vector();
  Gun peaceMaker1 = new Revolver("B12345678");
  Gun peaceMaker2 = new SemiAutomatic("A12345679");
  //Add to the list of munitions
  munitionsList.addElement(peaceMaker1);
  munitionsList.addElement(peaceMaker2);
  //Print the list of munitions
  for(int i=0;i<munitionsList.size();i++) {
    Gun currentGun = (Gun)munitionsList.elementAt(i);
    System.out.println(currentGun.getReloadInstructions());
    System.out.println(currentGun.getSerialNumber() + "\n");
  }
 }
}
```

Dynamic binding looks after the mapping to the correct method

Interfaces in Java

- Interfaces in Java are like classes except every method inside them is public abstract and every class field is pubic static final

- This will be the case whether or not you declare them this way (i.e. leaving out the final static will still mean fields are final static)

Interfaces in Java

- Interfaces look quite bizarre, but they have an important function, i.e. they can pass down behaviours without tying the implementer to a specific implementation

- For example, ActionListener allows you to click buttons and respond but it doesn’t tell you what you need to do once a button is pushed…and why would it!!!...only you know!

Interface example

```
public interface Phone {

    long number=387777777;

    public void makeCall();
    public void receiveCall();
}
```

If you implement the interface you commit to acting like a phone, but you’re not told how to do it

Passed down the behaviour of a phone, implementers are of type Phone!

Conclusions

- Using Polymorphism combined with dynamic binding can reduce the amount of static type checking needed in programs
- This will reduce if statements and will make the code more adaptable to changes, there may be hundreds of places in a program where you need to type check
- To extend the existing system all that is required is to write a new subclass and implement the abstract method, the rest of the code remains unchanged
- Instead of using a class at the top of the hierarchy in polymorphism, you can replace the abstract base class with an interface if no behaviours need to be passed down