Objectives

- Getting familiar with
  - Method Overriding
  - Polymorphism
    - Overriding Vs Overloading
    - Dynamic binding
    - UpCasting and DownCasting objects
    - `instanceOf` operator
  - ArrayList and Vector
Reconsider the geometry hierarchy. The method `toString()` is defined in the `Shape` class and overridden in both the `Rectangle` and the `Circle` classes.
Method Overriding (cont)

- Method overriding involves the creation of another method (**in a subclass**) which has the same signature, and its return type is compatible with that of the method in the superclass.

```java
public class Shape{
    public String toString(){
        return "This is a Shape object";
    }
}

public class Square extends Shape{
    public String toString(){
        return "This is a Square object";
    }
}
```
Method Overriding (cont)

- It is not possible to override a method which is marked as `final` or as `static`.
- The access level cannot be more restrictive than that of the overridden method’s. However it can be less.
- In essence, if a method cannot be inherited for some reason, then it cannot be overridden.
Method Overriding (cont)

- Implementation of toString in the Shape class
  ```java
  public String toString(){
      return getColour() + " shape created on " + getDateCreated();
  }
  ```

- Implementation of toString in the Circle class
  ```java
  public String toString(){
      return getColour() + " Circle created on " + getDateCreated();
  }
  ```

- Therefore this is possible
  ```java
  Shape s = new Shape();
  Circle c = new Circle();
  s.toString();
  //invoke the overridden version
  c.toString();
  ```
Better use of overriding

- Re-implementation of the `toString` in the Shape class

```java
public String toString(){
    return "Colour: " + getColour() +
           "\n DateCreated: " + getDateCreated();
}
```

- Re-implementation of the `toString` in the Circle class

```java
public String toString(){
    return "Circle Details \n" + super.toString();
}
```

- In this way the parent’s `toString` method is reused in an effective manner
Overriding Vs Overloading

```java
public class Test{
    public static void main(String[] args){
        A a = new A();
        a.p(10);
    }
}

public class B{
    public void p(int i){
        System.out.println("B prints "+i);
    }
}

public class A extends B{
    public void p(int i){
        System.out.println("A prints "+i);
    }
}
```

A prints 10
Overriding Vs Overloading

```java
public class Test{
    public static void main(String[] args){
        A a = new A();
        a.p(10);
    }
}

public class B{
    public void p(int i){
        System.out.println("B prints "+i);
    }
}

public class A extends B{
    public void p(double i){
        System.out.println("A prints "+i);
    }
}
```

B prints 10
Consider the following method invocation:

```java
obj.doIt();
```

At some point, this invocation is bound to the definition of the method that it invokes.

If this binding occurred at compilation time, then that line of code would call the same method every time.

However, Java defers method binding until run time -- this is called dynamic binding or late binding.

Late binding provides flexibility in program design.
Polymorphism

- **Means** “many forms”
- It is the ability to treat an object of any subclass as if it were an object of the base/super class.
- A base class will therefore have many forms
  - as itself
  - as any of its subclasses
- Polymorphism makes effective use of
  - *dynamic binding* or *late binding*
- Why is polymorphism important?
  - allows for improved code organisation and readability
  - for creating extensible programs (generic programming): programs that “grow” in time with little effect on existing code
Example

```
Shape s = new Shape();
```

- through polymorphism the following is also possible

```
Shape s1 = new Circle();
Shape s2 = new Rectangle();
```

- This is called **upcasting** (i.e. going from a subclass up, towards a superclass)
- Upcasting is always possible since a subclass can always behave like its superclass.
Example (cont)

- Therefore this is also possible

```java
Shape s1 = new Circle();
Shape s2 = new Rectangle();
s1.toString();
s2.toString();
```

- `s1.toString()` will invoke the Circle’s `toString()` method

```
Circle Details
Colour: white
DateCreated: 12/12/07
```

- `s2.toString()` will invoke the Rectangle’s `toString()` method

```
Rectangle Details
Colour: white
DateCreated: 12/12/07
```
Dynamic Binding

- The compiler checks the declared reference types at compilation time, however the JVM checks the real object at runtime.

- When the `toString` method is called via the `Shape` reference variable:
  - the JVM looks at the real objects at the other end of the reference,
  - “sees” that the method has been overridden by both the `Circle` and `Rectangle` classes and
  - invokes the method in the object’s class.
  - This is also referred to as **virtual method invocation**
Dynamic Binding II

- Dynamic binding works as follows: Suppose an object \( o \) is an instance of classes \( C_1, C_2, ..., C_{n-1}, \) and \( C_n, \) where \( C_1 \) is a subclass of \( C_2, \) \( C_2 \) is a subclass of \( C_3, ..., \) and \( C_{n-1} \) is a subclass of \( C_n. \)
  - \( C_n \) is the most general class
  - \( C_1 \) is the most specific class.
  - In Java, \( C_n \) is the Object class.
  - If \( o \) invokes a method \( p, \) the JVM searches the implementation for the method \( p \) in \( C_1, C_2, ..., C_{n-1} \) and \( C_n, \) in this order, until it is found.
  - Once an implementation is found, the search stops and the first-found implementation is invoked.
Dynamic Binding III

Since o is an instance of C_1, o is also an instance of C_2, C_3, ...C_{n-1} and C_n.
Polymorphism in Animals

Each object in the array is referenced by a more generic reference variable.

At runtime, however, the actual object is seen by the JVM, and the appropriate object’s eat method is invoked.
Polymorphic Arguments

**What happens in this code?**

```java
public class Vet{
    public void giveShot(Animal a){
        //assume all animals have this method
        a.makeNoise();
    }
}

public class PetOwner{
    public void takeToVet(){
        Vet v = new Vet();
        Dog d = new Dog();
        Horse h = new Horse();
        v.giveShot(d);
        v.giveShot(h);
    }
}
```

Dog’s `makeNoise()` invoked
Horse’s `makeNoise()` invoked
Polymorphic Arguments II

What is the output of this code?

```java
public class Animal{}
public class Horse extends Animal{}
public class UseAnimals{
    public void doStuff(Animal a){
        System.out.println("Animal version");
    }
    public void doStuff(Horse h){
        System.out.println("Horse version");
    }
    public static void main(String[] args){
        UseAnimals ua = new UseAnimals();
        Animal animalObj = new Animal();
        Horse horseObj = new Horse();
        ua.doStuff(animalObj);
        ua.doStuff(horseObj);
    }
}
```

Animal version
Horse version
Polymorphic Arguments II (cont)

- What if an Animal reference is assigned to a Horse object?

  ```java
  Animal animalRef = new Horse();
  ua.doStuff(animalRef);
  ```

- Which overloaded version is invoked?

  *Animal version*

- Remember:
  - The **reference type** determines which **overloaded** method is invoked, at compilation time.
  - The **object type** is used to determine which **overridden** version of a method is called at runtime.
Overloaded and Overridden methods

What happens when methods are both overloaded and overridden?

```java
public class Animal{
    public void eat(){
        System.out.println("Generic eating");
    }
}
public class Horse extends Animal{
    public void eat(){
        System.out.println("Horse eating hay");
    }
    public void eat(String s){
        System.out.println("Horse eating "+ s);
    }
}
```
Overloaded and Overridden methods II

Method Invocation

```java
Animal a = new Animal();
a.eat();

Horse h = new Horse();
h.eat();

Animal ah = new Horse();
ah.eat();

Horse he = new Horse();
he.eat(“apples”);  

Animal a2 = new Animal();
a2.eat(“treats”);

Animal ah2 = new Horse();
ah2.eat(“Carrots”);
```

Results

Generic eating

Horse eating hay; due to polymorphism the actual object type is used to determine method

Horse eating hay; overloaded eat(String s) method is invoked

Compiler error! Compiler tries to find eat(String s) within Animal and fails

Compiler error! Compiler still tries to find an eat(String s) in Animal and fails
**Downcasting**

- In the previous example the following code produced a compilation error

  ```java
  Animal ah2 = new Horse();
  ......
  ah2.eat("Carrots");
  ```

- There is however a solution

  ```java
  Animal ah2 = new Horse();
  ......
  ((Horse)ah2).eat("Carrots");
  ```

- **Downcast** (explicit casting) the reference variable to a Horse object and invoke its `eat(String s)` method.

- The compiler will not complain since he is forced to trust the developer.
Explicit casting **must** be used when casting an object from a superclass to a subclass. This type of casting may not always succeed.

```java
Apple x = (Apple)fruit;
Orange x = (Orange)fruit;
```
Downcasting could produce a `ClassCastException` if not done correctly.

```java
public class Animal{ }
public class Dog extends Animal{}
public class DogTest{
    public static void main(String[] args){
        Animal animal = new Animal();
        Dog fido = (Dog)animal;
    }
}
```

Though this compiles, it will fail at runtime, giving an exception, since we are trying to downcast an animal reference to a `Dog`. However the animal reference, was previously referring to an `Animal` object not a `Dog` object.
Use the `instanceof` operator to check whether an object is an instance of a class or not:

```java
Object myObject = new Circle();
... // Some lines of code
/** Perform casting if myObject is an instance of Circle */
if (myObject instanceof Circle){
    System.out.println("The circle diameter is "+ ((Circle)myObject).getDiameter());
}
```
Static methods cannot be overridden. However they can be redefined in a subclass. A redefined static method (in the subclass), hides the static method in its parent class.

```java
class Animal{
    public static void doStuff(){
        System.out.print("a");
    }
}
class Dog extends Animal{
    public static void doStuff(){
        System.out.print("b");
    }
    public static void main(String[] args){
        Animal[] a={new Animal(), new Dog()};
        for(int i=0; i < a.length; i++)
            a[i].doStuff();
    }
}
```
ArrayList & Vector

- The ArrayList and Vector classes implement dynamic arrays.
- The two classes are very similar except for some performance-related features.
- Both classes allow for adding, removing and accessing elements.

- Declaring/Initialising Vectors and ArrayLists

```java
Vector v = new Vector();
Vector v2 = new Vector(10);
ArrayList a = new ArrayList(2);
ArrayList<String> a = new ArrayList<String>();
```
Adding/Accessing elements

```java
import java.util.ArrayList;

ArrayList a = new ArrayList();
a.add("String");
a.add(1); //possible due to AutoBoxing
a.add(2.3); //possible due to AutoBoxing
for(int i=0; i< a.size(); i++){
    System.out.println(a.get(i).toString());
}
a.remove(1);
System.out.println(a.size());
System.out.println(a.contains(2.3));
```
import java.util.ArrayList;

ArrayList clist = new ArrayList();
clist.add(new Circle(3.4));
clist.add(new Circle(4.5));
clist.add(new Circle(2.3));
for(int i=0; i < clist.size(); i++){
    System.out.println("Area of circle: "+ ((Circle)clist.get(i)).getArea());
}

import java.util.ArrayList;

ArrayList<Cat> cats = new ArrayList<Cat>(3);
cats.add(new Lion("micio"));
cats.add(new Cheetah("runner"));
cats.add(new Tiger("grahhh"));
for(int i=0; i < cats.size(); i++){
    //no need to typecast
    System.out.println(cats.get(i).getName());
}