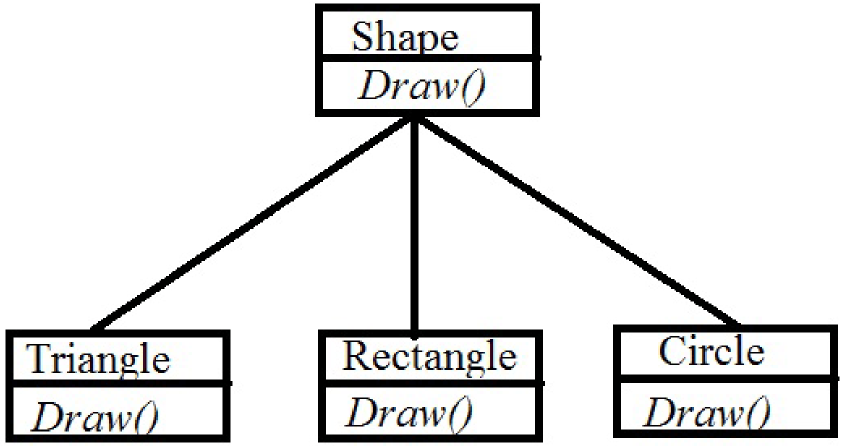
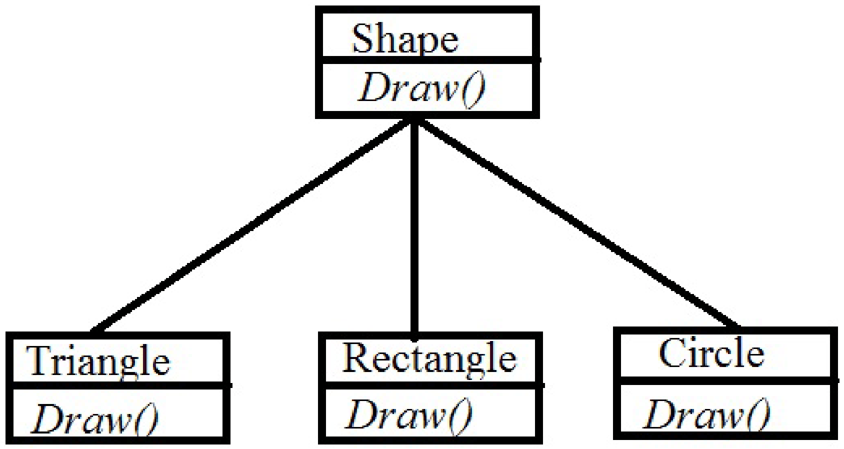
# Principle of Polymorphism



## Polymorphism

- Recall that an instance of a subclass is an instance of the superclass!
- Polymorphism is the ability to reference instances of a subclass from references of the superclass.



## Polymorphism

There are two types of Polymorphism:

- static polymorphism → method overloading!
- dynamic polymorphism → method overriding!

Static Polymorphism (or *static binding*) is what allows us to implement multiple methods using the same name but having different signatures. The signature of the method allows the compiler to identify which method is to be called and to bind the call with that method at compile time.

Dynamic Polymorphism (or *dynamic binding*) is what allows subclasses to override methods written in the superclass. The method is bound to the call at run-time, and the JVM will call the appropriate method depending on the type of the object that the method is being called on.

# Polymorphism

- We've been using reference variables like this:

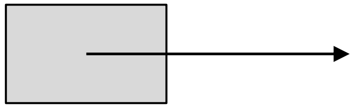
```
Rectangle r1 = new Rectangle(20, 30);
```

  - variable `r` is declared to be of type `Rectangle`
  - it holds a reference to a `Rectangle` object

- But a square *is a* Rectangle:

```
Rectangle r1 = new Square(50, "cm");
```

Creates an instance of a **Square**



Referenced by a variable of type **Rectangle**

# Polymorphism

- We've been using reference variables lik

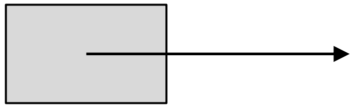
```
Rectangle r1 = new Recta
```

  - variable `r` is declared t
  - it holds a reference to a

*Dynamic* Polymorphism ensures that methods invoked on r1 at run-time are methods appropriate to the object that is being referenced.

- But a square *is a* Rectangle:

```
Rectangle r1 = new Square(50, "cm");
```



Creates an instance of a **Square**

Referenced by a variable of type **Rectangle**

# Polymorphism and Collections of Objects

- Polymorphism is useful when we have a collection of objects of different but related types.

- Example:

For each element of the array, the appropriate `toString()` method is called!

`myShapes[0]`: the **Rectangle** version of `toString()` is called
`myShapes[1]`: the **Square** version of `toString()` is called

etc.

```
 for (int i = 0; i < myShapes.length; i++) {
    System.out.println(myShapes[i]);
}
```

# Polymorphism

- We've been using reference variables like this:

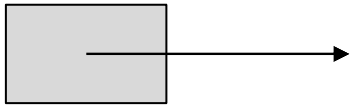
```
Rectangle                                  0);
```

  - varia                                    tangle
  - it hol                                   ject

Establishes the `type` and binds the object to that type!

- But a squar    s a Rect

```
Rectangle r1 = new Square(50, "cm");
```



Creates an instance of a **Square**

Referenced by a variable of type **Rectangle**

## Polymorphism

- We've been using reference variables like this:

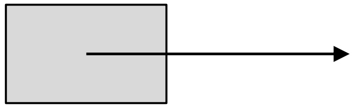
```
Rectangle                              0);
```

  - varia                              tangle
  - it hol                              ject

This allows the compiler to ensure that only methods known to the type can be invoked.

- But a squar

```
Rectangle r1 = new Square(50, "cm");
```



Creates an instance of a **Square**

Referenced by a variable of type **Rectangle**

## Polymorphism and Collections of Objects

- Polymorphism is useful when we have a collection of objects of different but related types.
- The only methods that can be called on our collection of Shape objects are the methods that are known by all types of Shapes!

```
Shape[] myShapes = new Shape[5];
myShapes[0] = new Rectangle(20, 30);
myShapes[1] = new Square(50, "cm");
myShapes[2] = new Triangle(10, 8);
myShapes[3] = new Circle(10);
myShapes[4] = new Rectangle(50, 100);

for (int i = 0; i < myShapes.length; i++) {
   System.out.println(myShapes[i]);
}
```

## Polymorphism and Collections of Objects

- Polymorphism is useful when we have a collection of objects of different but related types.
- Invoking a method defined in a subclass through a reference bound to the superclass **will not** be allowed by the compiler!

```
Shape[] myShapes = new Shape[5];
myShapes[0] = new Rectangle(20, 30);
myShapes[1] = new Square(50, "cm");
myShapes[2] = new Triangle(10, 8);
myShapes[3] = new Circle(10);
myShapes[4] = new Rectangle(50, 100);

for (int i = 0; i < myShapes.length; i++) {
   System.out.println(myShapes[i]);
}
```

## New Methods defined in Sub-class

- Let’s say we added some method to the Square sub-class that was not known to the super class Rectangle. Example:

```
public class Square extends Rectangle {
   private String unit;

   public Square(int side, String unit) {
       this.width = this.height = side;
   }

   // inherits methods from Rectangle
   // overrides the toString method
   // implements some new method
   public void someMethodForSquare() { ... }

}
```

```
// some other body of code
{
   Square sq = new Square(5);
   sq.someMethodForSquare(); // accepted
}
```

# New Methods defined in Sub-class

- Let’s say we added some method to the Square sub-class that was not known to the super class Rectangle. Example:

```
public class Square extends Rectangle {
   private String unit;

   public Square(int side, String unit) {
       this.width = this.height = side;
   }

   // inherits methods from Recta
   // overrides the toString meth
   // implements some new meth
   public void someMethodForSq

}
```

Calling a sub-class method on a reference to the subclass!

```
// some other body of code
{
   Square sq = new Square(5);
   sq.someMethodForSquare(); // accepted
}
```

## New Methods defined in Sub-class

- Let’s say we added some method to the Square sub-class that was not known to the super class Rectangle. Example:

```
public class Square extends Rectangle {
   private String unit;

   public Square(int side, String
      this.width = this.height
   }

   // inherits methods fro
   // overrides the toStri
   // implements some new m
   public void someMethodFo

}
```

Calling a sub-class method on a reference to the superclass! Methods written in the sub-class are not known by the super-class.

```
// some other body of code
{
   Rectangle sq = new Square(5);
   sq.someMethodForSquare(); // error
}
```

# Polymorphism and Collections of Objects

- Polymorphism is useful when we have a collection of objects of different but related types.

- **What if** you call a method defined **only** in the subclass through a superclass reference?

```
Shape[] myShapes = new
myShapes[0] = new Recta
myShapes[1] = new Squa
myShapes[2] = new Trian
myShapes[3] = new Circle(10
myShapes[4] = new Rectangle(50, 100);
```

Java only allows methods to be called that are known to the type of the reference.

```
for (int i = 0; i < myShapes.length; i++) {
   myShapes[i].someMethod();
}
```

# Scenarios

1. Super class reference to a superclass object:

```
Rectangle r = new Rectangle(5, 10);
r.area();    // method of superclass
```

2.

```
Square s = new Square(5);
s.area();                   // subclass - inherited
System.out.println(s);   // subclass - overridden
```

3. Superclass reference to a subclass object:

1. Super class

```
Rectangle
r.area();    //
```

Can invoke all inherited methods of the superclass and **all** methods overridden and implemented in the subclass!

2.

```
Square s = new Square(5);
s.area();                   // subclass - inherited
System.out.println(s);      // subclass - overridden
```

3. Superclass reference to a subclass object:

## Scenarios

1. Super class reference to a superclass object:

```
Rectangle r
r.area();
```

Can invoke all inherited methods of the superclass and all methods **overridden** in the subclass!

2.

```
Square s = ne
s.area();                 // subclass - inherited
System.out.printl  s);    // subclass - overridden
```

3.

```
Rectangle s = new Square(5);
s.area();                 // method of superclass
System.out.println(s);    // subclass - overridden
```

## Scenarios

1. Super class reference to a superclass object:

```
Rectangle r
r.area();
```

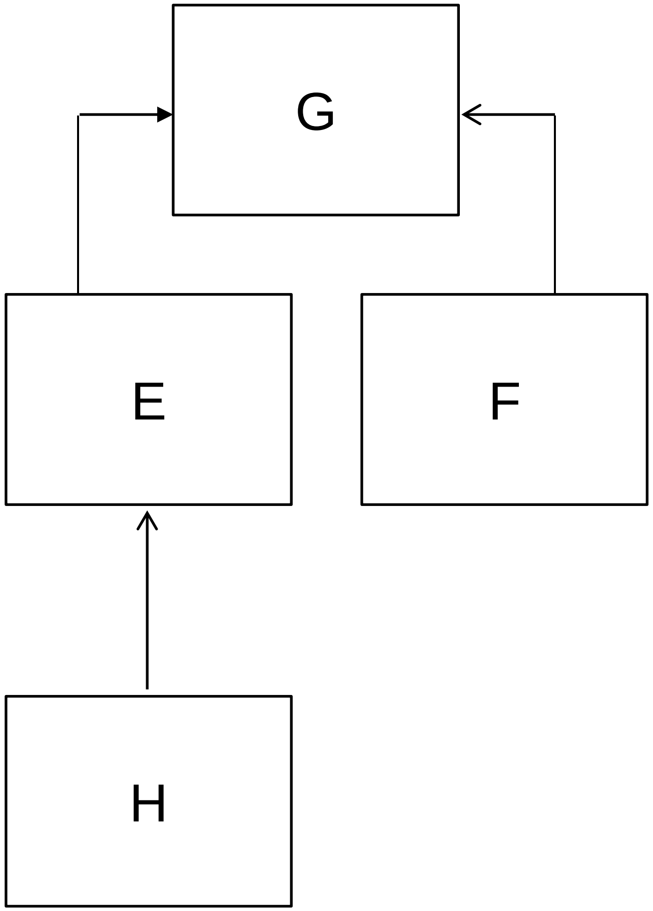
But you cannot invoke methods written in the subclass that are not known in the superclass.

2.
```
Square s = ne
s.area();                   // subclass - inherited
System.out.printl   s);     // subclass - overridden
```

**3.**
```
Rectangle s = new Square(5);
s.area();                   // method of superclass
System.out.println(s);      // subclass - overridden
```

# A Simple Heirarchy

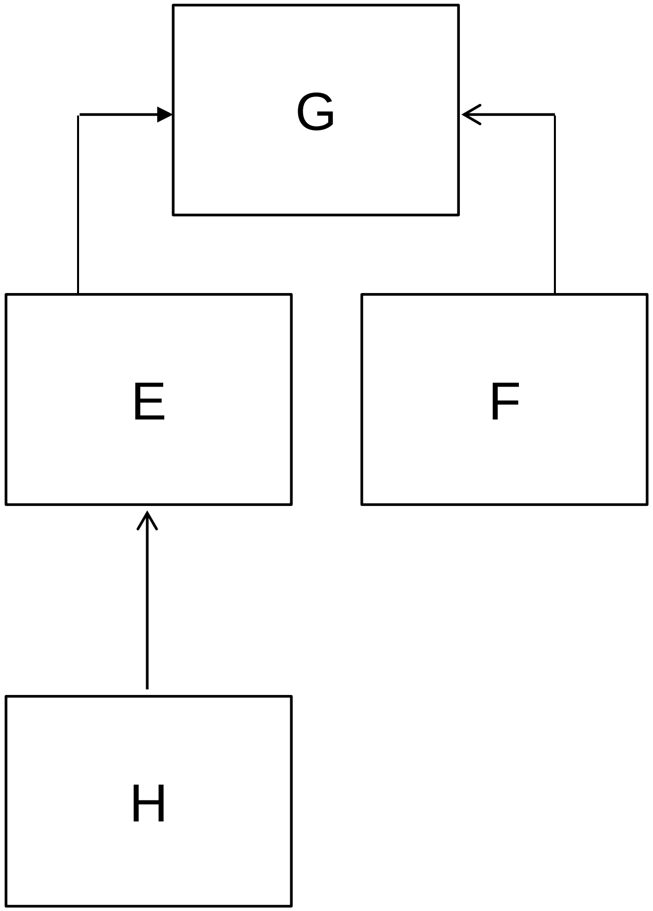
- Class `G` has two non-static methods:
  - method1, method2
- `E extends G`
  - it inherits G's fields and methods
  - it ***overrides*** method2 with its own version
  - it adds a new method called method3
- `F also extends G`
  - it inherits G's fields and methods
  - it ***overrides*** method2
- `H extends E`
  - it inherits E's fields and methods
  - it ***overrides*** method1



```
{
    H h = new H();
    h.method1(); // ?
}
```

# A Simple Heirarchy

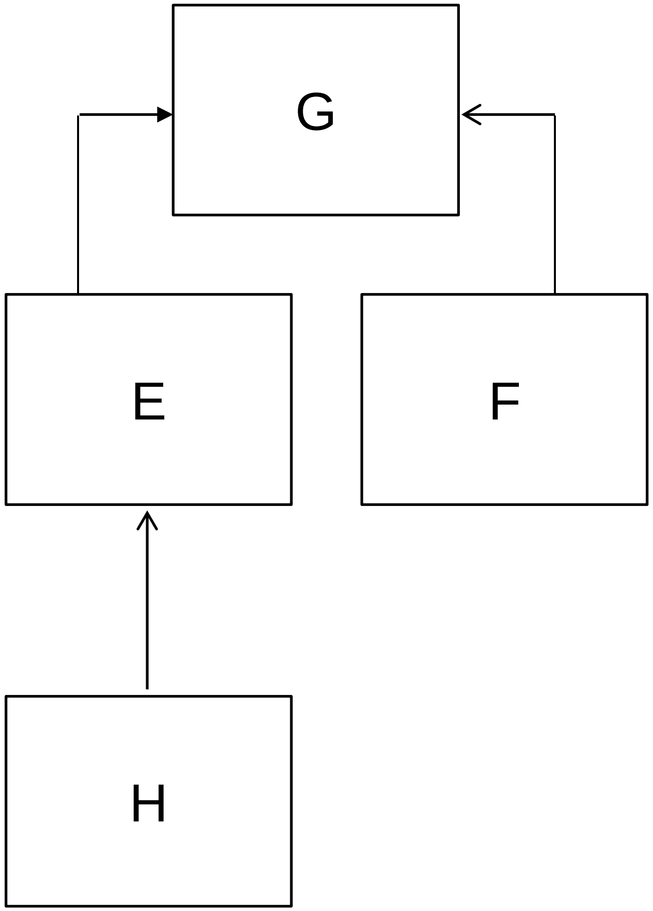
- Class `G` has two non-static methods:
  - method1, method2
- `E extends G`
  - it inherits G's fields and methods
  - it ***overrides*** method2 with its own version
  - it adds a new method called method3
- `F also extends G`
  - it inherits G's fields and methods
  - it ***overrides*** method2
- `H extends E`
  - it inherits E's fields and methods
  - it ***overrides*** method1



```
{
    H h = new H();
    h.method2(); // ?
}
```

# A Simple Heirarchy

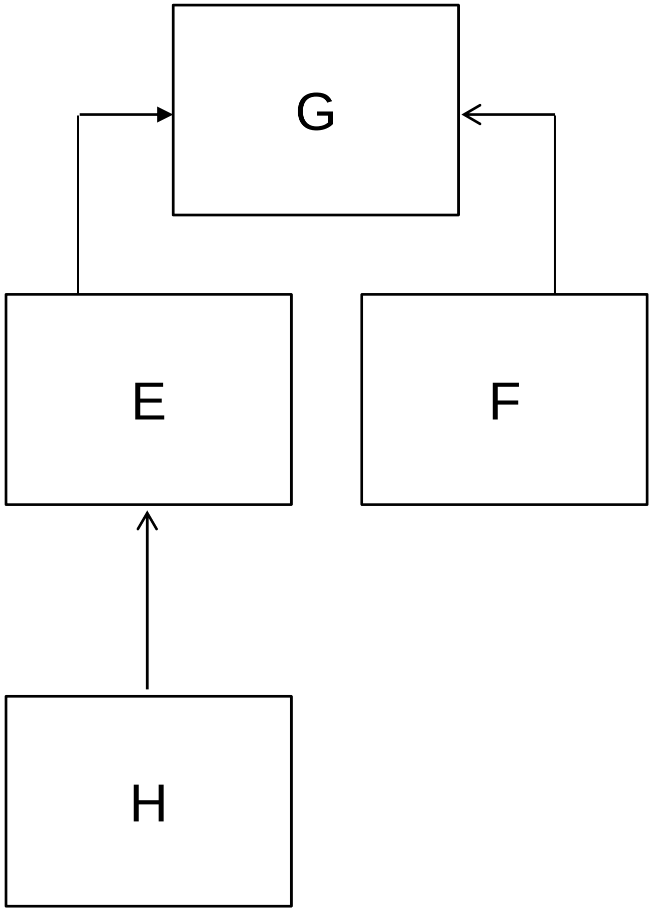
- Class `G` has two non-static methods:
  - method1, method2
- `E extends G`
  - it inherits G's fields and methods
  - it ***overrides*** method2 with its own version
  - it adds a new method called method3
- `F also extends G`
  - it inherits G's fields and methods
  - it ***overrides*** method2
- `H extends E`
  - it inherits E's fields and methods
  - it ***overrides*** method1



```
{
    H h = new H();
    h.method3(); // ?
}
```

# A Simple Heirarchy

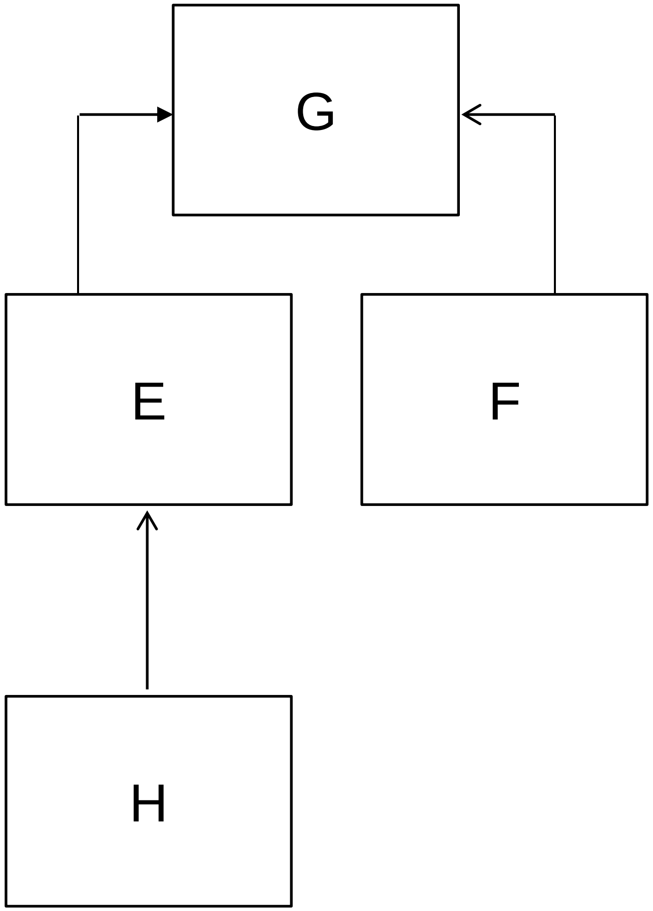
- Class `G` has two non-static methods:
  - method1, method2
- `E extends G`
  - it inherits G's fields and methods
  - it ***overrides*** method2 with its own version
  - it adds a new method called method3
- `F also extends G`
  - it inherits G's fields and methods
  - it ***overrides*** method2
- `H extends E`
  - it inherits E's fields and methods
  - it ***overrides*** method1



```
{
    E h = new H();
    h.method3(); // ?
}
```

# A Simple Heirarchy

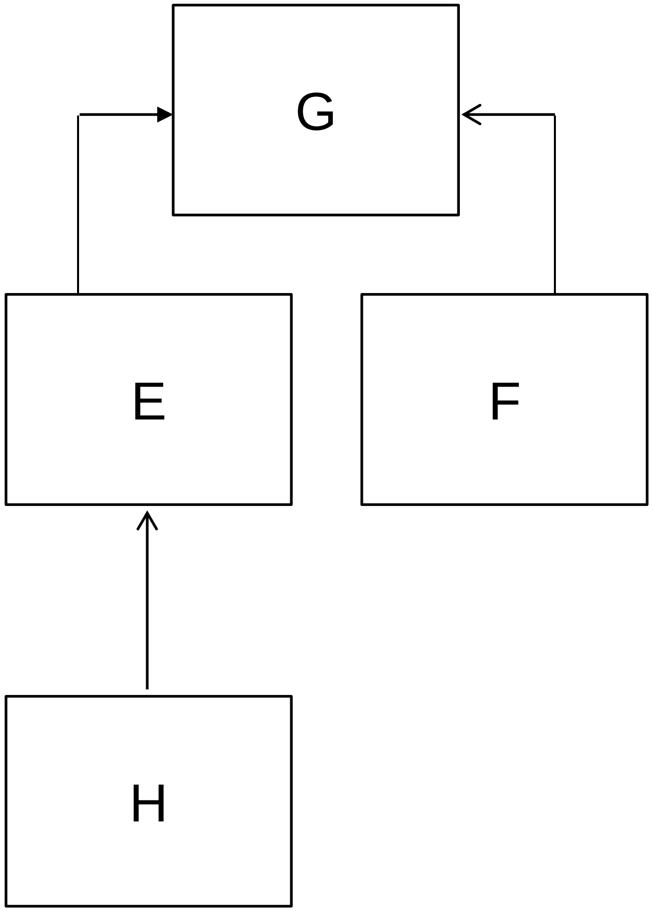
- Class `G` has two non-static methods:
  - method1, method2
- `E extends G`
  - it inherits G's fields and methods
  - it ***overrides*** method2 with its own version
  - it adds a new method called method3
- `F also extends G`
  - it inherits G's fields and methods
  - it ***overrides*** method2
- `H extends E`
  - it inherits E's fields and methods
  - it ***overrides*** method1



```
{
    G h = new H();
    h.method3(); // ?
}
```

# A Simple Heirarchy

- Class `G` has two non-static methods:
  - method1, method2
- `E extends G`
  - it inherits G's fields and methods
  - it ***overrides*** method2 with its own version
  - it adds a new method called method3
- `F also extends G`
  - it inherits G's fields and methods
  - it ***overrides*** method2
- `H extends E`
  - it inherits E's fields and methods
  - it ***overrides*** method1



```
{
    F h = new H();
    h.method1(); // ?
}
```