Inheritance

- Derive new classes (**subclass**) from existing ones (**superclass**).

- Only the **Object** class (**java.lang**) has no superclass.

- Every class = **subclass** of **Object**.
Inheritance

- Code reuse – *methods* and *properties*.
- Use classes to model objects of the same type
  - Define general class
    - Then define specialized classes.
Inheritance

- Can only inherit from one class.
- Can inherit from multiple classes in other languages (C++)
Example

- Let’s say you create the following classes:
  - Circle
  - Rectangle
  - Triangle
  - etc.

- Share certain properties, but also specialized properties
public class GeometricObject {
    private String Color;
    private String name;
    private float area;

    // constructors...
    public GeometricObject(String color, String name) {
        ...
    }

    // get/set methods, etc.
}
Example

class Circle extends GeometricObject {
    private double radius; // specific property

    public Circle(double radius, String color, String name) {
        this.radius = radius;
        this.color = color;
        this.name = name;
    }

    // get/set methods, etc.

    public double getArea() {
        return radius * radius * Math.PI;
    }
}

Example

- An error in the constructor!

  ```java
  public Circle(double radius, String color, String name) {
      this.radius = radius;
      this.color = color;
      this.name = name;
  }
  ```

- Why can't we use “this.color” and “this.name”?
Example

- They are **private** properties.

- NOT accessible by subclasses.

- Should use `setColor()` instead.

- **public** methods/properties inherited.
Example

```java
public class Rectangle extends GeometricObject {
    private double width, height; // specific properties

    public Rectangle(double w, double h, String color, String name) {
        this.height = h;
        this.width = w;
        setColor(color);
        setName(name);
    }

    // get/set methods, etc.

    public double getArea() {
        return width * height;
    }
}
```
The super keyword

- Are constructors inherited?
  - Yes. super used for:
    - calling a super class constructor
    - calling a super class method
The **super** keyword

- Constructors:
  - `super()` calls the default constructor
  - `super(arguments)` calls the constructors according to the arguments
The super keyword

- Constructors:

```java
public Circle(double radius, String color, String name) {
    super(color, name);
    this.radius = radius;
}
```
The super keyword

- Constructors:

```java
public Circle(double radius, String color, String name) {
    super(color, name);
    this.radius = radius;
}
```

```java
public GeometricObject(String color, String name) {
    this.color = color;
    this.name = name;
}
```
The `super` keyword

What if we don’t add the `super(..)` constructor call?
The super keyword

```java
public Circle(double radius, String color, String name) {
    this.radius = radius;
}
```
public Circle(double radius, String color, String name) {
    this.radius = radius;
}

*Is the same as:*

public Circle(double radius, String color, String name) {
    super();
    this.radius = radius;
}
The super keyword

Called Constructor Chaining
The **super** keyword

- Calling the super class methods

- `super.method(parameters)`...

- Optional
Using **get/set** methods to access properties of the superclass is cumbersome...
The **protected** keyword

- Often want subclasses to directly access properties/methods

- Use **protected** instead of **private/public**

- Subclasses can now directly access
public class GeometricObject {
    private String Color;
    private String name;
    private float area;

    // constructors...

    // get/set methods, etc.

}
The protected keyword

```java
public class GeometricObject {
    protected String Color;
    protected String name;
    protected float area;

    // constructors...

    // get/set methods, etc.
}
```
The **protected** keyword

```java
public class Circle extends GeometricObject {
    private double radius;  // specific property

    public Circle(double radius, String color, String name) {
        this.radius = radius;
        this.color = color;
        this.name = name;
    }

    // Now works!
}
```
Polymorphism

- Important aspect of OOP
Polymorphism

- Important aspect of OOP

“capability of an action or method to do different things based on the object that it is acting upon.”
Polymorphism

- Important aspect of OOP

- “capability of an action or method to do different things based on the object that it is acting upon.”

- “characteristic of being able to assign a different meaning or usage to something in different contexts - specifically, to allow an entity such as a variable, a function, or an object to have more than one form.”
Polymorphism

1. Overriding Methods
2. Overloaded methods
3. Dynamic (late) method binding
Overriding Methods

- Sometimes need to overwrite methods written in the superclass

- Re-define the method in the subclass.
Overriding Methods

- The **Object** class and **toString()**

- **toString()** = String output when you print the object
Overriding Methods

- The **Object** class and **toString()**

  - **toString()** = String output when you print the object

- **Object** class has default **toString()** method
The `toString()` method

```java
public String toString() {
    String str;
    ...
    return str;
}
```
The **toString()** method

```java
Circle circle1 = new Circle(“Circle1”, “Red”, 3.5);

System.out.println(circle1);
```

Output:

?
The `toString()` method

```java
Circle circle1 = new Circle("Circle1", "Red", 3.5);

System.out.println(circle1);
```

Output:

```java
Circle@19821f
```
The `toString()` method

- So override the method

```java
public String toString() {
    String str = "";
    str += "Circle: " + getName() + "",
    str += "Color:  " + getColor() + "",
    str += "Radius: " + getRadius();
    return str;
}
```
The `toString()` method

```java
Circle circle1 = new Circle("Circle1", "Red", 3.5);
System.out.println(circle1);
```

Output:

```
Circle: Circle1,Color: Red,Radius: 3.5
```
The `equals()` method

- The `Object` class and `equals()`

- `equals (Object obj) =` returns whether the object passed in and this one are the same
The `equals()` method

- Default implementation:

```java
public boolean equals(Object obj) {
    return (this == obj);
}
```

*Compares references*
The `equals()` method

```java
Circle circle1 = new Circle(3.5);
Circle circle2 = new Circle(3.5);
```
The `equals()` method

```java
Circle circle1 = new Circle(3.5);
Circle circle2 = new Circle(3.5);

circle1.equals(circle2); // ?
```
The `equals()` method

```java
Circle circle1 = new Circle(3.5);
Circle circle2 = new Circle(3.5);

circle1.equals(circle2); // returns false!
```
The `equals()` method

```java
Circle circle1 = new Circle(3.5);
Circle circle2 = new Circle(3.5);

circle1.equals(circle2); // returns false!

Circle circle3 = circle1;

circle1.equals(circle3); // ?
```
The `equals()` method

```java
Circle circle1 = new Circle(3.5);
Circle circle2 = new Circle(3.5);

circle1.equals(circle2); // returns false!

Circle circle3 = circle1;

circle1.equals(circle3); // returns true!
```
The equals() method

Override it

```java
public boolean equals(Object obj) {
    if (obj instanceof Circle)
        return this.radius == ((Circle) obj).getRadius();
    else
        return false;
}
```
The `instanceof` operator

- Test if object is of a specified type

```java
if (object instanceof Class)
```
The `instanceof` operator

- Test if object is of a specified type

```java
if (object instanceof Class)
```

- Example:

```java
Circle circle1 = new Circle(3.4);
if (circle1 instanceof Circle) {
    ...
}
```
The `instanceof` operator

- Testing with the SuperClass:

  ```java
  if (subclassObj instanceof SuperClass) {
      ...
  }
  ```
The `instanceof` operator

- Testing with the SuperClass:

```java
if (Circle instanceof GeometricObject) {
    ...
}
```

The `instanceof` operator

- Testing with the SuperClass:

```java
if (subclassObj instanceof SuperClass) {
    ...
}
```

returns `true`!
The `instanceof` operator

- Example:

```java
Circle circle1 = new Circle(3.4);
if (circle1 instanceof GeometricObject) {
    ...
}
```

returns `true`!
The `instanceof` operator

What if we write:

```java
if (SuperClassObj instanceof subclass) {
    ...
}
```
The `instanceof` operator

```java
if (SuperClassObj instanceof subclass) {
    ...
}
```

- returns `true` as long as the object is an instance of the subclass
The *instanceof* operator

- **Example:**

`Object` is a superclass of `Circle`

```java
public boolean equals(Object obj) {
    if (obj instanceof Circle)
        return this.radius == ((Circle) obj).getRadius();
    else
        return false;
}
```
The `instanceof` operator

- Be careful with `null` objects

```java
String s = null;

s instanceof String = false
```
Object casting

- So we can determine if (SuperClassObj instanceof subclass)

- Convert it to subclass by casting
Object casting

- So we can determine
  \[
  \text{if (SuperClassObj instanceof subclass)}
  \]

- Convert it to subclass by *casting*

- Can now access members/properties
Object casting

Once again...

```java
public boolean equals(Object obj) {
    if (obj instanceof Circle) {
        return this.radius == ((Circle)obj).getRadius();
    } else {
        return false;
    }
}
```
Once again...

```java
public boolean equals(Object obj) {
    if (obj instanceof Circle) {
        Circle circle = (Circle)obj;  // casting
        return this.radius == circle.getRadius();
    }
    else
        return false;
}
```
Overloading Methods

- Looked at it before.

- Can have multiple methods with the same name.

- But different parameters, return type
Dynamic binding

- Object type is determined at run-time
- Ability of a program to resolve references to subclass methods at runtime.
Dynamic binding

```java
public class GeometricObject {
    protected String Color;
    protected String name;
    protected float area;

    public float getArea() {
        return -1;
    }
}
```
Dynamic binding

// Circle
public class Circle {
    public float getArea() {
        return Math.PI * radius * radius;
    }
}

// Rectangle
public class Rectangle {
    public float getArea() {
        return width * height;
    }
}
Dynamic binding

`GeometricObject[] objs = new GeometricObject[2];`
Dynamic binding

```java
GeometricObject[] objs = new GeometricObject[2];
objs[0] = new Circle(3);
objs[1] = new Rectangle(1, 2);
```
Dynamic binding

```java
GeometricObject[] objs = new GeometricObject[2];
objs[0] = new Circle(3);
objs[1] = new Rectangle(1, 2);

for (GeometricObject i : objs) {
    System.out.println(i.getArea());
}
```
Dynamic binding

```
GeometricObject[] objs = new GeometricObject[2];
objs[0] = new Circle(3);
objs[1] = new Rectangle(1, 2);

for (GeometricObject i : objs) {
    System.out.println(i.getArea());
}
```

What’s the output?
Dynamic binding

Output:
9.0
2.0

The subclass getArea() methods are called.

At run-time, Java *dynamically* determines the class.
final Classes

- Sometimes classes shouldn’t be allowed to be extended.

- The **String** class for example. Use the keyword “final”

```java
final class String {
    ...
}
```
final Methods

- Methods which can’t be overridden in the subclasses.
Summary...

- Inheritance
  - super
  - protected
- Polymorphism
  - Overloading
  - Overwriting
  - Dynamic binding
- instanceof
- Object casting
- final classes and methods
Next lecture...

- Abstract Classes
- Interfaces