Using Inheritance

Objectives

- After completing this lesson, you should be able to:
  - Define inheritance in the context of a Java class hierarchy
  - Create a subclass
  - Override a method in the superclass
  - Use a keyword to reference the superclass
  - Define polymorphism
  - Use the `instanceof` operator to test an object's type
  - Cast a superclass reference to the subclass type
  - Explain the difference between abstract and nonabstract classes
  - Create a class hierarchy by extending an abstract class

Topics

- Overview of inheritance
- Working with superclasses and subclasses
- Overriding superclass methods
- Introducing polymorphism
- Creating and extending abstract classes

Java Puzzle Ball

Have you played through **Inheritance Puzzle 3**?

Consider the following:

What is inheritance?

Why are these considered “Inheritance” puzzles?

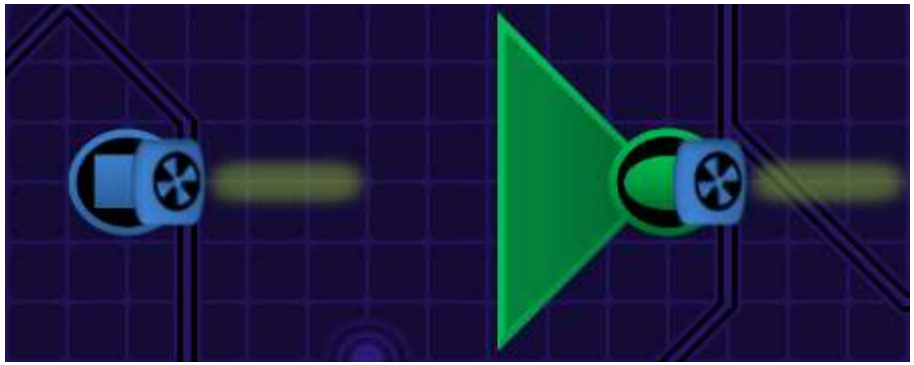
Java Puzzle Ball Debrief

- What is inheritance?
  - **Inheritance** allows one class to be derived from another.
    - A child inherits properties and behaviors of the parent.
    - A child *class* inherits the fields and method of a parent *class.*
  - In the game:
    - Blue shapes also appear on *green* bumpers

Inheritance in Java Puzzle Ball

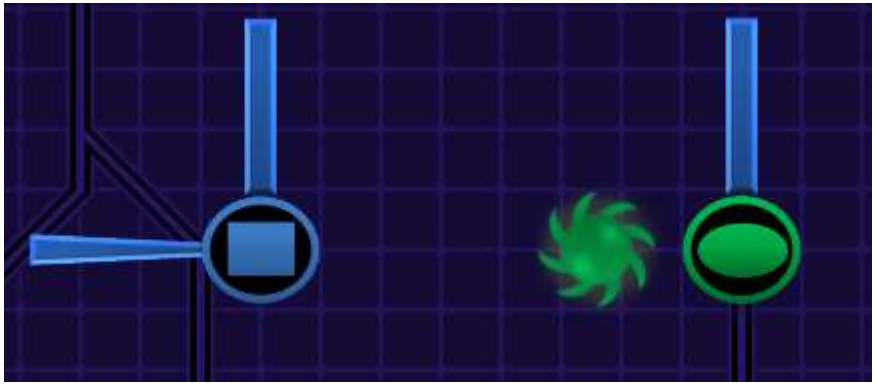
- Inheritance Puzzle 1:
  - Methods for deflecting the ball that were originally assigned to Blue Bumpers are also found on Green Bumpers.

- Inheritance P
  - Green Bumpers contain methods from Blue Bumpers, PLUS methods unique to Green Bumpers.



Inheritance in Java Puzzle Ball

- Inheritance Puzzle 3:
  - If Green Bumpers inherit unwanted Blue Bumper methods, it is possible to **override,** or replace those methods.



Implementing Inheritance

```
public class Clothing {
    public void display(){…}
    public void setSize(char size){…}
}
```

- `public class Shirt extends Clothing {…}`

Use the `extends` keyword.

```
Shirt myShirt = new Shirt();
myShirt.setSize ('M');
```

This code works!

More Inheritance Facts

- The parent class is the **superclass**.
- The child class is the **subclass**.
- A subclass may have unique fields and methods not found in the superclass.

subclass superclass

```
public class Shirt extends Clothing {
    private int neckSize;
    public int getNeckSize(){
        return neckSize;
    }
    public void setNeckSize(int nSize){
       this.neckSize = nSize;
    }
}
```

Topics

- Overview of inheritance
- Working with superclasses and subclasses
- Overriding superclass methods
- Introducing polymorphism
- Creating and extending abstract classes

Duke's Choice Classes: Common Behaviors

Shirt	Trousers
getId() getPrice() getSize() getColor() getFit()	getId() getPrice() getSize() getColor() getFit() getGender()
setId() setPrice() setSize() setColor() setFit()	setId() setPrice() setSize() setColor() setFit() setGender()
display()	display()

Code Duplication

Shirt

```
getId()
display()
getPrice()
getSize()
getColor()
getFit()
```

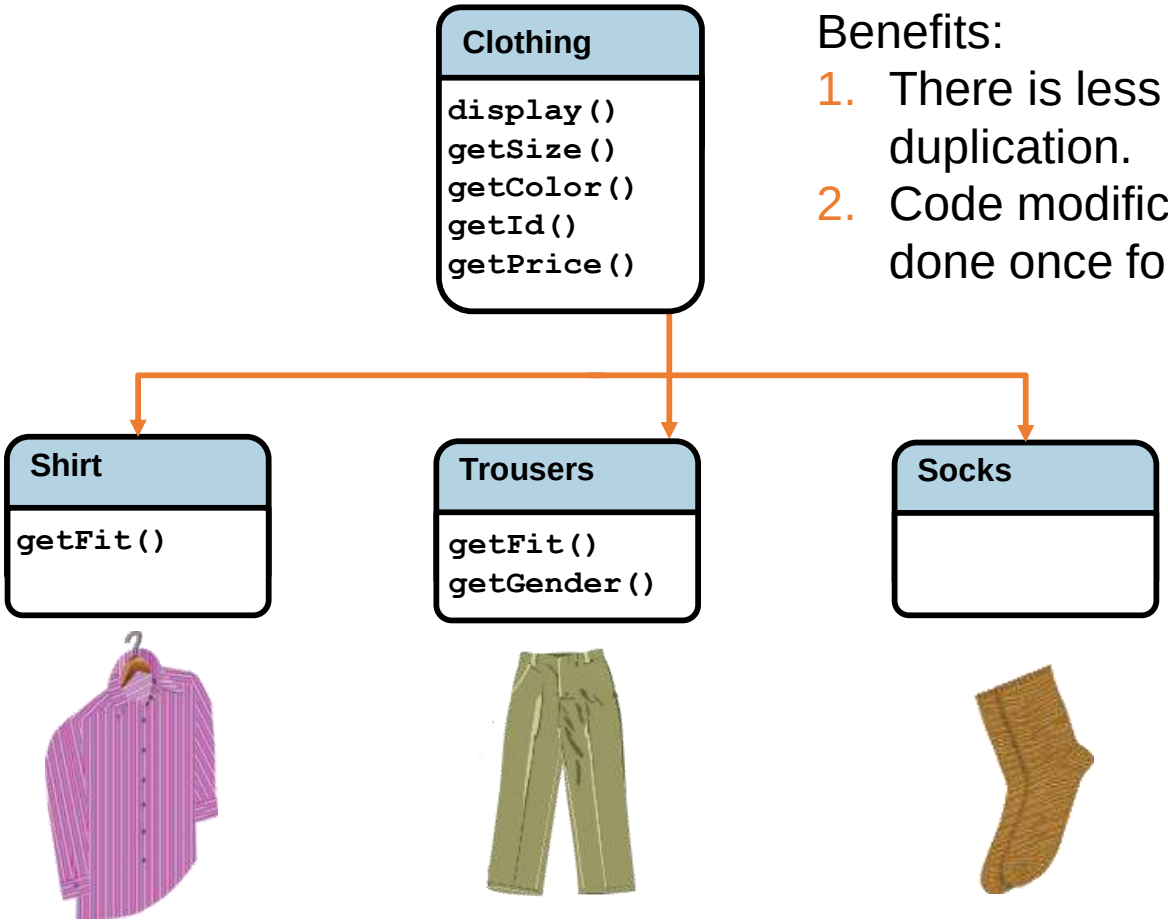
Trousers

```
getId()
display()
getPrice()
getSize()
getColor()
getFit()
getGender()
```

Socks

```
getId()
display()
getPrice()
getSize()
getColor()
```

Inheritance



Benefits:

1. There is less code duplication.
2. Code modification can be done once for all subclasses.

Clothing Class: Part 1

```
01 public class Clothing {
02   // fields given default values
03   private int itemID = 0;
04   private String desc = "-description required-";
05   private char colorCode = 'U';
06   private double price = 0.0;
07
08   // Constructor
09   public Clothing(int itemID, String desc, char color,
10      double price ) {
11       this.itemID = itemID;
12       this.desc = desc;
13       this.colorCode = color;
14       this.price = price;
15   }
16 }
```

Shirt Class: Part 1

```
• 01 public class Shirt extends Clothing {
• 03   private char fit = 'U';
• 04
• 05   public Shirt(int itemID, String description, char
• 06           colorCode, double price, char fit) {
• 07       super (itemID, description, colorCode, price);
• 08
• 09       this.fit = fit;
• 10   }
• 12   public char getFit() {
• 13       return fit;
• 14   }
• 15   public void setFit(char fit) {
• 16       this.fit = fit;
17   }
```

Reference to the superclass constructor

Reference to this object

Constructor Calls with Inheritance

```
public static void main(String[] args){
        Shirt shirt01 = new Shirt(20.00, 'M');          }
```

```
public class Shirt extends Clothing {
        private char fit = 'U';

public Shirt(double price, char fit) {
    super(price);                                   //MUST call superclass constructor
        this.fit = fit;                             }}
```

```
public class Clothing{
        private double price;

        public Clothing(double price){
                    this.price = price;
}}
```

Inheritance and Overloaded Constructors

```
public class Shirt extends Clothing {
      private char fit = 'U';

public Shirt(char fit){
                  this(15.00, fit);              //Call constructor in same class
      }                           //Constructor is overloaded

      public Shirt(double price, char fit) {
    super(price);                          //MUST call superclass constructor
    this.fit = fit;                        }}
```

```
public class Clothing{
      private double price;

      public Clothing(double price){
                  this.price = price;
}}
```

Exercise 12-1: Creating a Subclass

- In this exercise, you create the `Shirt` class, which extends the Item class.
  - Add two fields that are unique to the `Shirt` class.
  - Invoke the superclass constructor from the `Shirt` constructor.
  - Instantiate a `Shirt` object and call the `display` method.

Output

```
Item description: Shirt
        ID: 1
        Price: 25.99
```

Topics

- Overview of inheritance
- Working with superclasses and subclasses
- Overriding superclass methods
- Introducing polymorphism
- Creating and extending abstract classes

More on Access Control

- Access level modifiers determine whether other classes can use a particular field or invoke a particular method
  - At the top level—public, or *package-private* (no explicit modifier).
  - At the member level—public, private, protected, or *package-private* (no explicit modifier).

Stronger access privileges

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
No modifier	Y	Y	N	N
private	Y	N	N	N

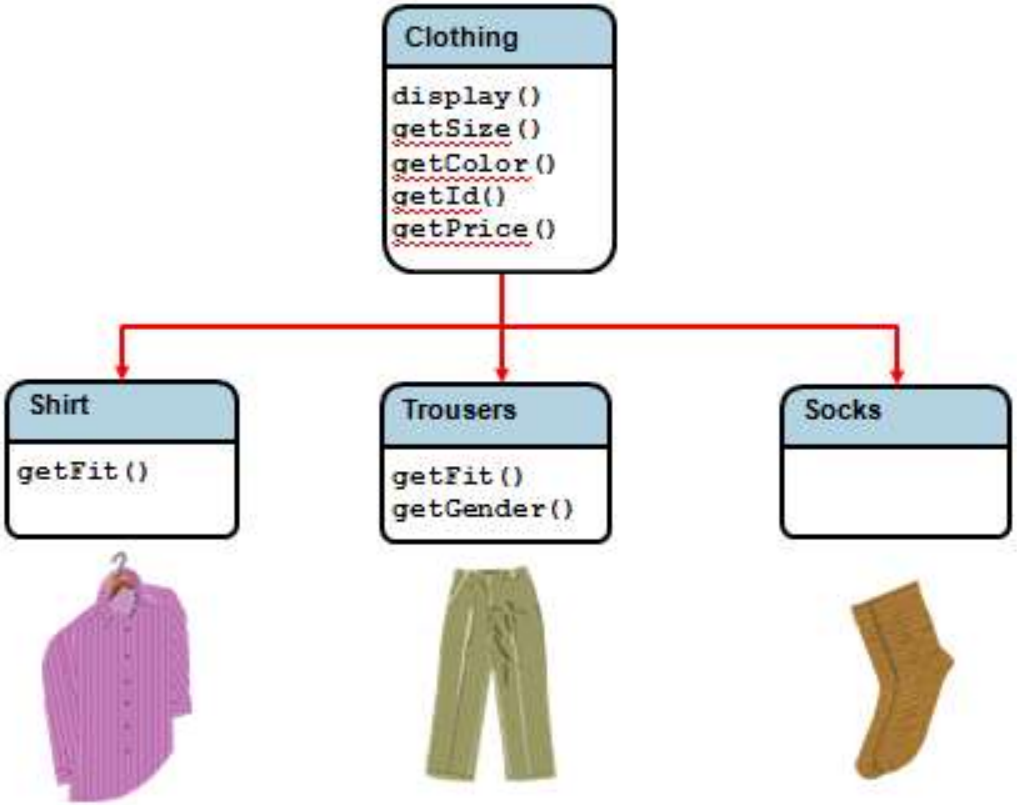
Overriding Methods

- **Overriding:** A subclass implements a method that already has an implementation in the superclass.

- **Access Modifiers:**
  - The method can only be overridden if it is accessible from the subclass
  - The method signature in the subclass cannot have a more restrictive (stronger) access modifier than the one in the superclass

Review: Duke's Choice Class Hierarchy

- Now consider these classes in more detail.



Clothing Class: Part 2

```
29   public void display() {
30     System.out.println("Item ID: " + getItemID());
31     System.out.println("Item description: " + getDesc());
32     System.out.println("Item price: " + getPrice());
33     System.out.println("Color code: " + getColorCode());
34   }
35   public String getDesc (){
36       return desc;
37   }
38   public double getPrice() {
39       return price;
40   }
41   public int getItemID() {
42       return itemID;
43   }
44   protected void setColorCode(char color){
45       this.colorCode = color; }
```

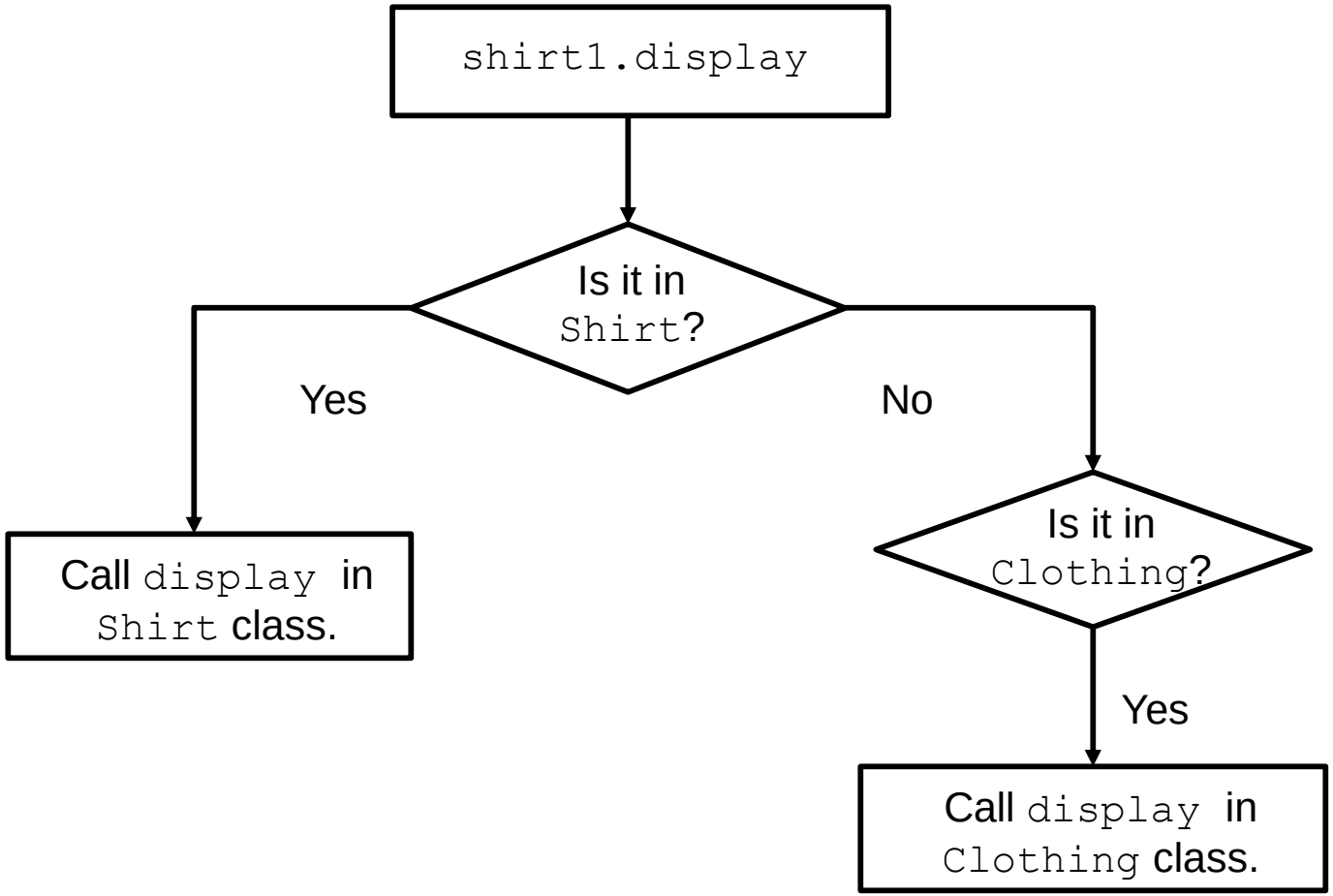
Assume that the remaining `get/set` methods are included in the class.

Shirt Class: Part 2

```
17  // These methods override the methods in Clothing
18    public void display() {
19          System.out.println("Shirt ID: " + getItemID());
20          System.out.println("Shirt description: " + getDesc());
21          System.out.println("Shirt price: " + getPrice());
22          System.out.println("Color code: " + getColorCode());
23          System.out.println("Fit: " + getFit());
24  }
• 25
• 26  protected void setColorCode(char colorCode) {
27     //Code here to check that correct codes used
28     super.setColorCode(colorCode);
29  }
30}
```

Call the superclass's version of `setColorCode`.

Overriding a Method: What Happens at Run Time?



Topics

- Overview of inheritance
- Working with superclasses and subclasses
- Overriding superclass methods
- Introducing polymorphism
- Creating and extending abstract classes

Polymorphism

- Polymorphism means that the same message to two different objects can have different results.
  - “Good night” to a child means “Start getting ready for bed.”
  - “Good night” to a parent means “Read a bedtime story.”
- In Java, it means the same method is implemented differently by different classes.
  - This is especially powerful in the context of inheritance.
  - It relies upon the “*is a*” relationship.

Superclass and Subclass Relationships

- Use inheritance only when it is completely valid or unavoidable.
  - Use the *"is a"* test to decide whether an inheritance relationship makes sense.
  - Which of the phrases below expresses a valid inheritance relationship within the Duke's Choice hierarchy?

O.K. A Shirt *is a* piece of Clothing.

– A Hat *is a* Sock.

– Equipment *is a* piece of Clothing.

– Clothing and Equipment *are* Items. O.K.

Using the Superclass as a Reference

- So far, you have referenced objects only with a reference variable of the same class:
  - To use the `Shirt` class as the reference type for the `Shirt` object:
    ```
    Shirt myShirt = new Shirt();
    ```
  - But you can also use the superclass as the reference:
    ```
    Clothing garment1 = new Shirt();
    Clothing garment2 = new Trousers();
    ```

Shirt **is a** (type of) Clothing.
Trousers **is a** (type of) Clothing.

Polymorphism Applied

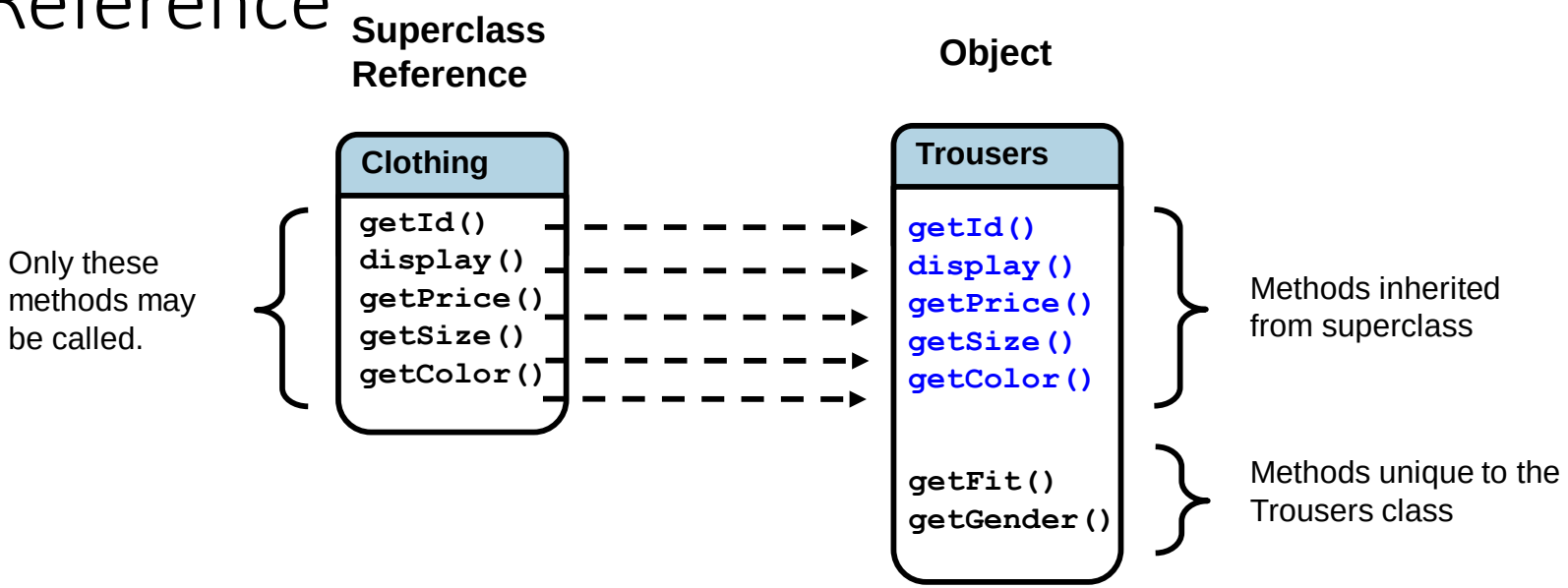
```
Clothing c1 = new ??();

c1.display();
c1.setColorCode('P');
```

c1 could be a Shirt, Trousers, or Socks object.

- The method will be implemented differently on different types of objects. For example:
  - Trousers objects show more fields in the display method.
  - Different subclasses accept a different subset of valid color codes.

Accessing Methods Using a Superclass Reference




```
Clothing c1 = new Trousers();
c1.getId();   OK
c1.display(); OK
c1.getFit();  NO!
```

Casting the Reference Type

Superclass reference

```
Clothing c1 = new Trousers();
```

Clothing

```
getId()
display()
getPrice()
getSize()
getColor()
```

Cast the object.

```
((Trousers)c1).getFit()
```

Trousers

```
getId()
display()
getPrice()
getSize()
getColor()

getFit()
getGender()
```

The object

Topics

- Overview of inheritance
- Working with superclasses and subclasses
- Overriding superclass methods
- Introducing polymorphism
- Creating and extending abstract classes

Abstract Classes

Shirt

```
getFit()
```

=

Clothing

```
getId()
display()
getPrice()
getSize()
getColor()
```

=

Trousers

```
getFit()
getGender()
```

=

Socks

=

Abstract Classes

Use the `abstract` keyword to create a special class that:

- Cannot be instantiated
- May contain concrete methods
- May contain abstract methods that **must** be implemented later by any nonabstract subclasses

```
Clothing cloth01 = new Clothing()
```

```
public abstract class Clothing{
    private int id;

    public int getId(){
        return id;
    }

    public abstract double getPrice();
    public abstract void display();
}
```

Concrete method

Abstract methods

Extending Abstract Classes

```
public abstract class Clothing{
      private int id;

      public int getId(){
       return id;
      }
      protected abstract double getPrice();        //MUST be
  implemented
      public abstract void display();   }
      //MUST be implemented
```

```
public class Socks extends Clothing{
      private double price;

      protected double getPrice(){
                  return price;
    }
      public void display(){
                  System.out.println("ID: " +getID());
                  System.out.println("Price: $" +getPrice());
}}
```

Summary

- In this lesson, you should have learned the following:
  - Creating class hierarchies with subclasses and superclasses helps to create extensible and maintainable code by:
    - Generalizing and abstracting code that may otherwise be duplicated
    - Allowing you to override the methods in the superclass
    - Allowing you to use less-specific reference types
  - An abstract class cannot be instantiated, but it can be used to impose a particular interface on its descendants.