12

Using Inheritance

ORACLE

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

ORACLE

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

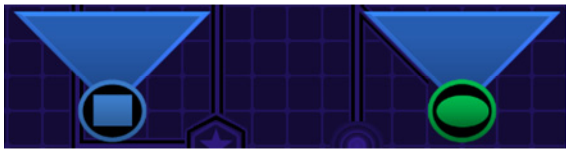
In this lesson, you will examine the object-oriented concept of inheritance.

- Inheritance is a mechanism by which a class can be derived from another class, just as a child derives certain characteristics from the parent.
- You can see this reflected in the game. When you drag an icon to the blue wheel, it affects green objects as well as blue objects in the field of play. The green wheel derives its characteristics from the blue wheel.

Inheritance in Java Puzzle Ball

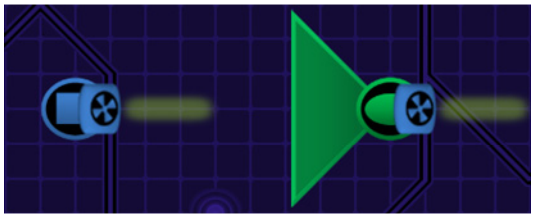
Inheritance Puzzle 1:

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



Inheritance Puzzle 2:

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



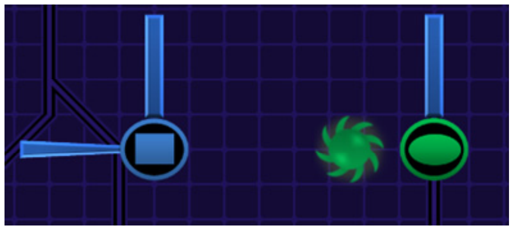
In the first image, both Blue Bumpers and Green Bumpers share the Triangle Wall.

In the second image, both Blue Bumpers and Green Bumpers share the fan. In addition, Green Bumpers uniquely have a Triangle Wall.

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.



Green Bumpers have overridden the rotation wall with a blade.

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!

In the code example above, an inheritance relationship between the `Shirt` class and its parent, the `Clothing` class, is defined. The keyword `extends` creates the inheritance relationship:

```
public class Shirt extends Clothing…
```

As a result, `Shirt` objects share the `display` and `setSize` methods of the `Clothing` class. Although these methods are not actually written in the `Shirt` class, they may be used by all `Shirt` objects. Therefore, the following code can be successfully compiled and run:

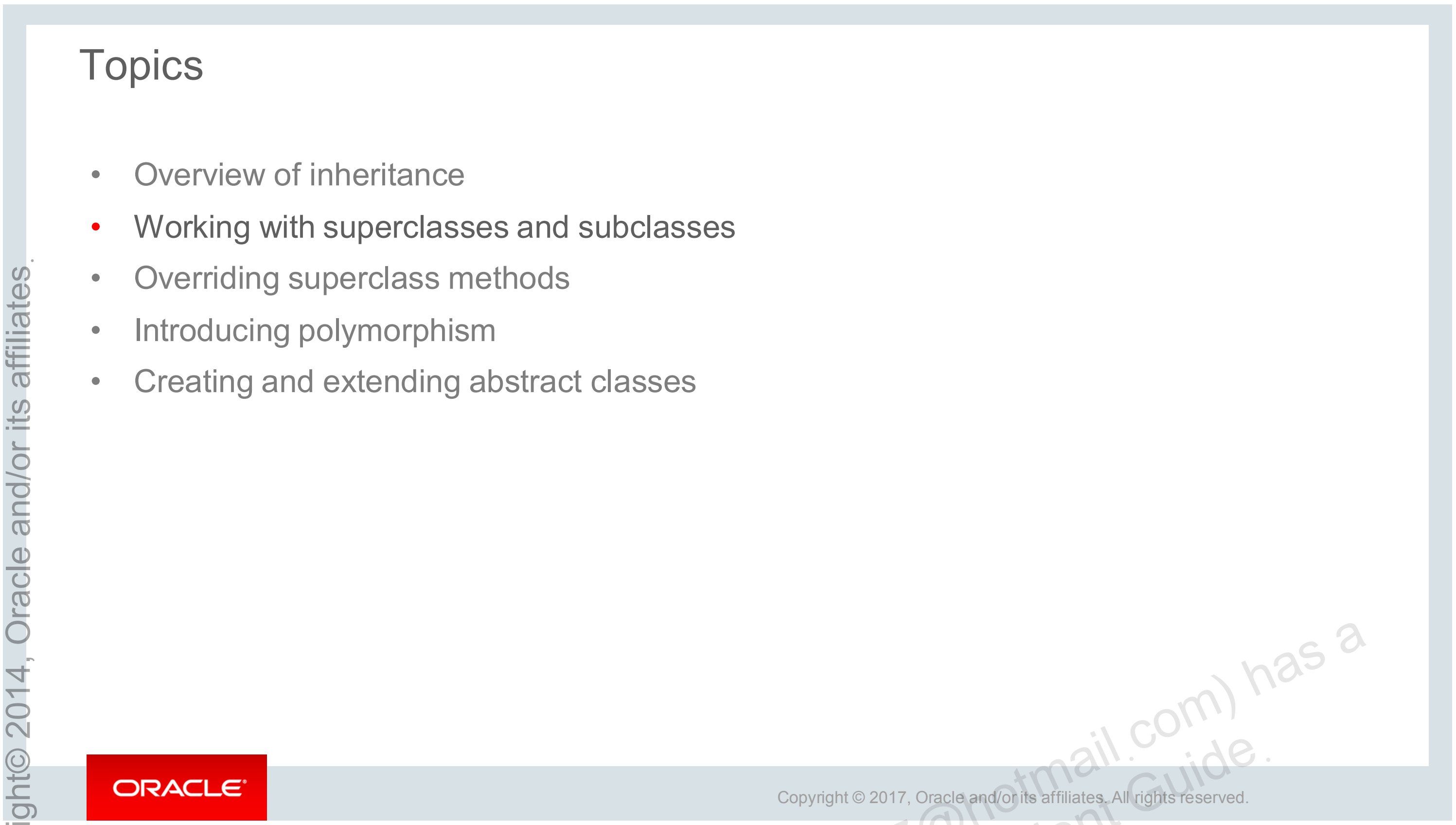
```
Shirt myShirt = new Shirt();
myShirt.setSize('M');
```

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.

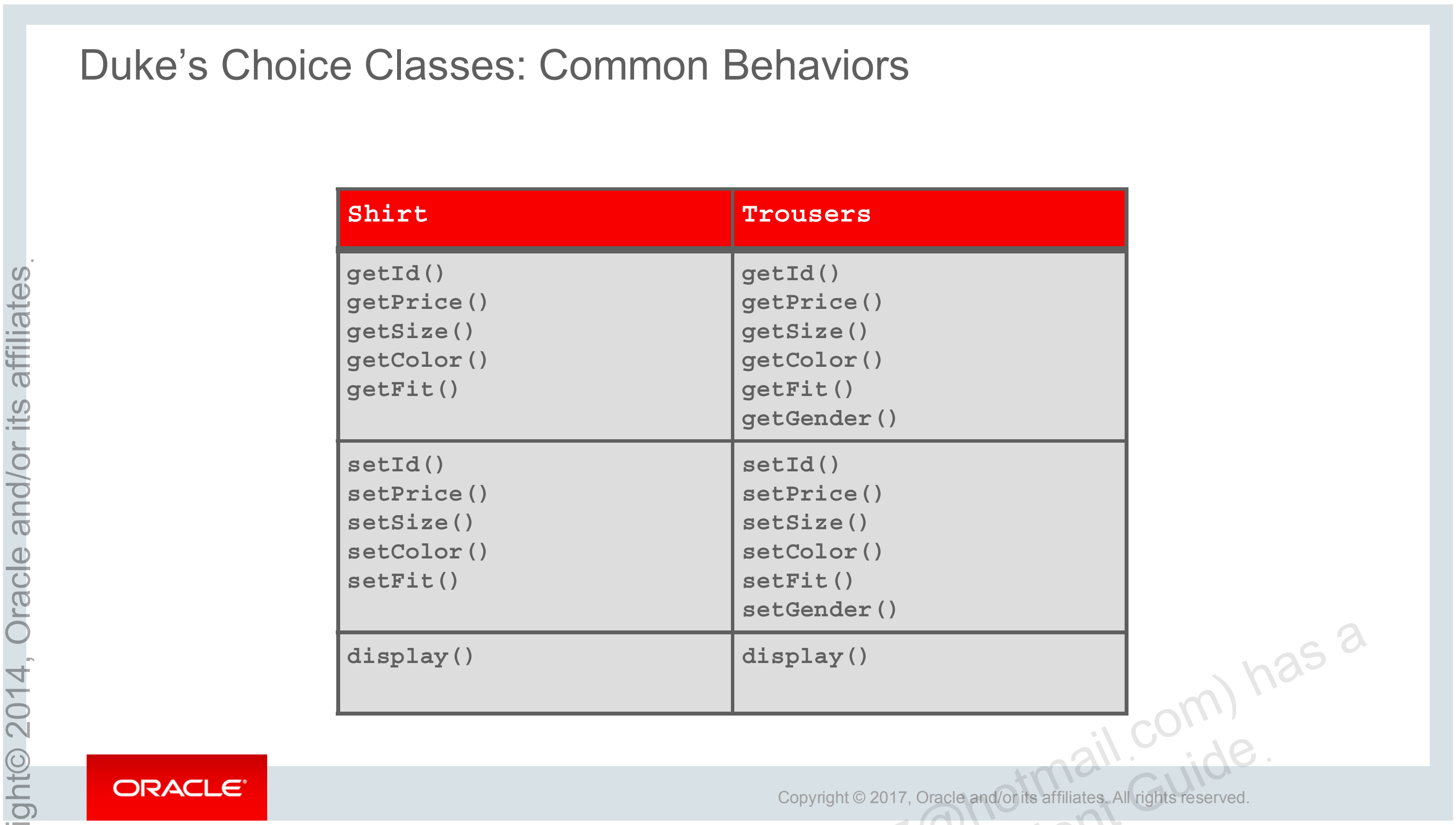
```
                  subclass                 superclass
public class Shirt extends Clothing {
    private int neckSize;
    public int getNeckSize(){
       return neckSize;
    }
    public void setNeckSize(int nSize){
       this.neckSize = nSize;
    }
}
```

The subclass not only has access to all of the public fields and methods of the superclass, but it can declare additional fields and methods that are specific to its own requirements.



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

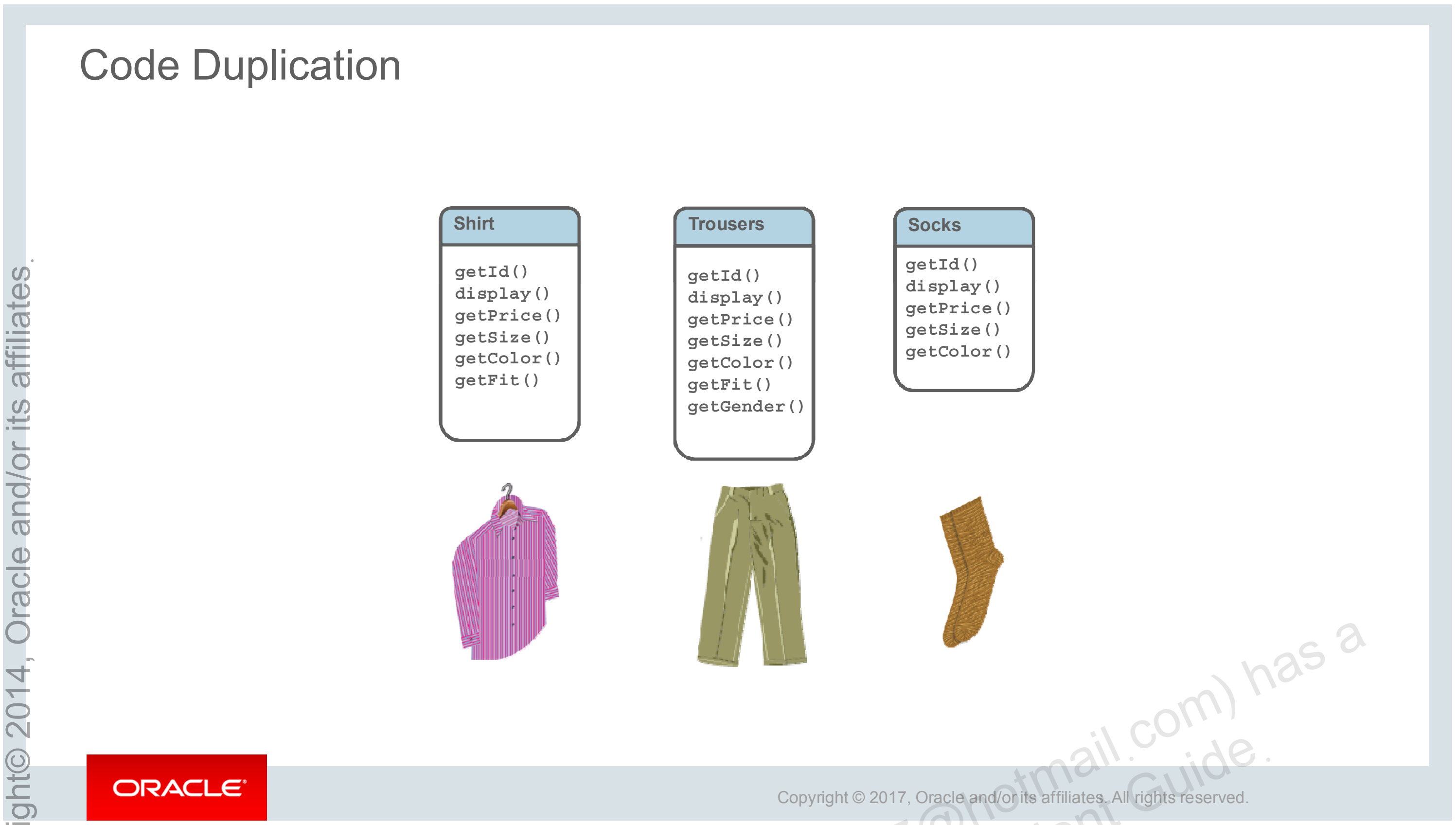
Shirt	Trousers

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

setId()
setPrice()
setSize()
setColor()
setFit()
	setId()
	setPrice()
	setSize()
	setColor()
	setFit()
	setGender()

display()	display()

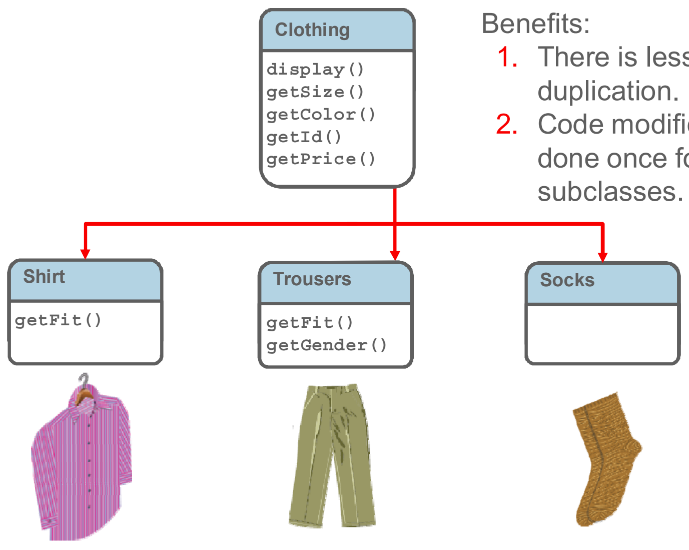
The table in the slide shows a set of behaviors for some classes belonging to the Duke’s Choice shopping cart application, the `Shirt` class and the `Trousers` class. The classes are shown fully encapsulated so that all field values are accessible only through setter and getter methods. Notice how both classes use many of the same methods; this may result in code duplication, making maintenance and further expansion more difficult and error-prone.



Code Duplication

If Duke's Choice decides to add a third item, socks, as well as trousers and shirts, you may find even greater code duplication. The diagram in the slide shows only the getter methods for accessing the properties of the new objects.

Inheritance



Benefits:

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

You can eliminate code duplication in the classes by implementing inheritance. Inheritance enables programmers to put common members (fields and methods) in one class (the superclass) and have other classes (the subclasses) inherit these common members from this new class.

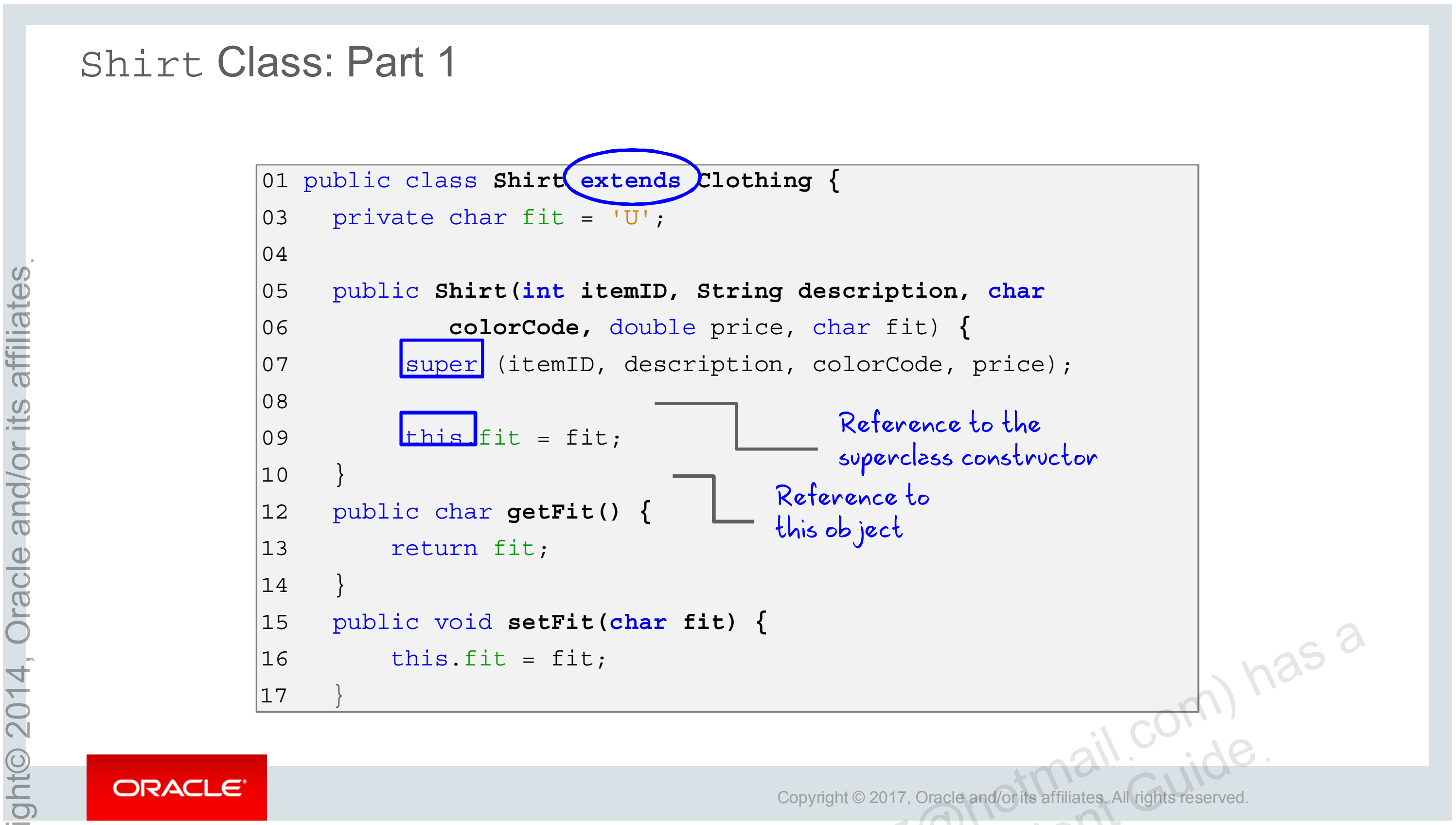
An object instantiated from a subclass behaves as if the fields and methods of the subclass were in the object. For example,

- The `Clothing` class can be instantiated and have the `getId` method called, even though the `Clothing` class does not contain `getId`. It is inherited from the `Item` class.
- The `Trousers` class can be instantiated and have the `display` method called even though the `Trousers` class does not contain a `display` method; it is inherited from the `Clothing` class.
- The `Shirt` class can be instantiated and have the `getPrice` method called, even though the `Shirt` class does not contain a `getPrice` method; it is inherited from the `Clothing` class.

Clothing Class: Part 1

```java
public class Clothing {
  // fields given default values
  private int itemID = 0;
  private String desc = "-description required-";
  private char colorCode = 'U';
  private double price = 0.0;

  // Constructor
  public Clothing(int itemID, String desc, char color,
     double price ) {
      this.itemID = itemID;
      this.desc = desc;
      this.colorCode = color;
      this.price = price;
  }
}
```

The code in the slide shows the fields and the constructor for the `Clothing` superclass.

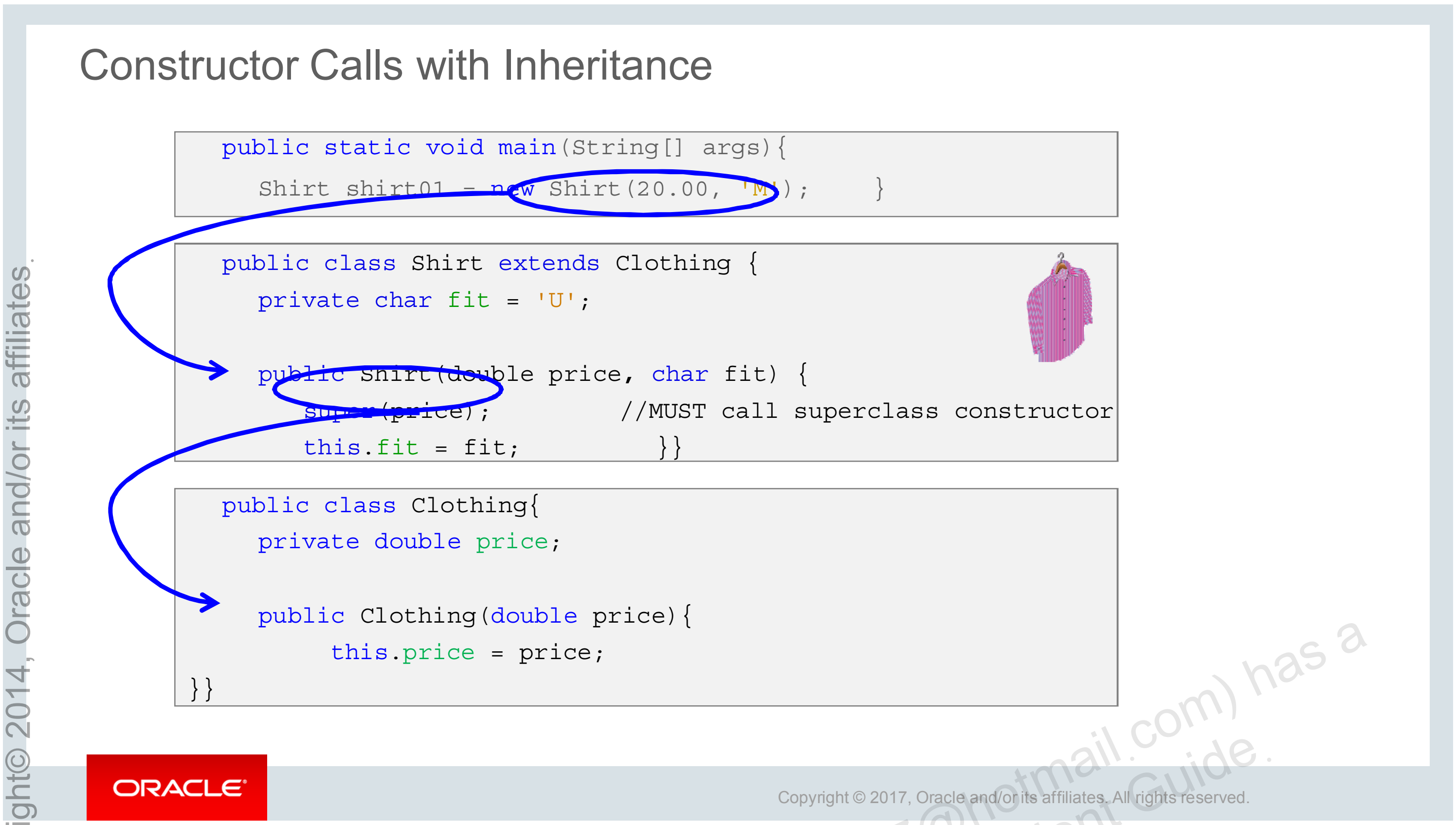


Shirt Class: Part 1

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

  public Shirt(int itemID, String description, char
          colorCode, double price, char fit) {
      super (itemID, description, colorCode, price);

      this.fit = fit;
  }
  public char getFit() {
      return fit;
  }
  public void setFit(char fit) {
      this.fit = fit;
  }
```

The slide shows the code of the `Shirt` subclass. As you have seen in an earlier example, the `extends` keyword enables the `Shirt` class to inherit all the members of the `Clothing` class. The code declares attributes and methods that are unique to this class. Attributes and methods that are common with the `Clothing` class are inherited and do not need to be declared. It also includes two useful keywords and shows a common way of implementing constructors in a subclass.

- `super` refers to the superclass. In the example in the slide, the `super` keyword is used to invoke the constructor on the superclass. By using this technique, the constructor on the superclass can be invoked to set all the common attributes of the object being constructed. Then, as in the example here, additional attributes can be set in following statements.
- `this` refers to the current object instance. The only additional attribute that `Shirt` has is the `fit` attribute, and it is set after the invocation of the superclass constructor. The `this` keyword is necessary here as the constructor method parameter is also named `fit`). However, even when not strictly necessary, it is good programming practice because it makes the code more readable.



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;
 }}
```

Within the constructor of a subclass, you must call the constructor of the superclass. If you call a superclass constructor, the call must be the first line of your constructor. This is done using the keyword `super`, followed by the arguments to be passed to the superclass constructor.

The constructor of the subclass sets variables that are unique to the subclass. The constructor of the superclass sets variables that originate from the superclass.

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;
}}
```

Use the `this` keyword to call another constructor within the same class. This is how you call an overloaded constructor.

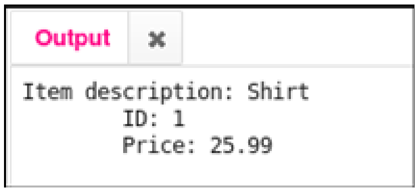
Use the `super` keyword to call a constructor in the superclass. When you have overloaded subclass constructors, all of your constructors must eventually lead to the superclass constructor. If you call a superclass constructor, the call must be the first line of your constructor.

If your superclass constructors are overloaded, Java will know which superclass constructor you are calling based on the number, type, and order of arguments that you supply.

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.



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

- In the Java Code Console, access Lessons > 12-Inheritance > Exercise1.
- Follow the instructions below the code editor to create the `Shirt` class and then instantiate it and call the display method (in the Item class).
- Run the `ShoppingCart` class to test your code. Your output should look similar to the screenshot above.
- If you need help, click the Solution link. To go back to your code, click the Exercise link again. Any changes that you have made will have been saved.

Topics

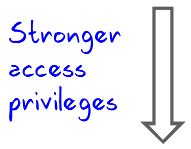
- Overview of inheritance
- Working with superclasses and subclasses
- Overriding superclass methods
- Introducing polymorphism
- Creating and extending abstract classes

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).

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



In the diagram, the first data column indicates whether the class itself has access to the member defined by the access level. As you can see, a class always has access to its own members. The second column indicates whether classes in the same package as the class (regardless of their parentage) have access to the member. The third column indicates whether subclasses of the class declared outside this package have access to the member. The fourth column indicates whether all classes have access to the member

Note: packages will be covered in greater detail in lesson titled "Deploying and Maintaining the Soccer Application".

Overriding Methods

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

Access Modifiers:

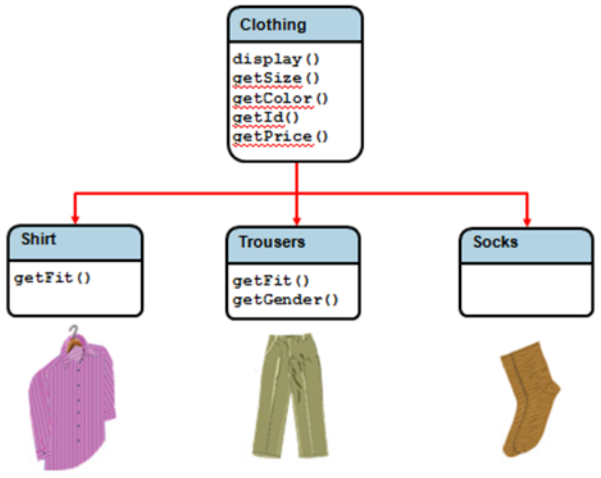
- The method can only be overridden if it is accessible from the subclass
- The method signature in the subclass cannot have a more restrictive (stronger) access modifier than the one in the superclass

Subclasses may implement methods that already have implementations in the superclass. In this case, the methods in the subclass are said to override the methods from the superclass.

- For example, although the `colorCode` field is in the superclass, the color choices may be different in each subclass. Therefore, it may be necessary to override the accessor methods (getter and setter methods) for this field in the individual subclasses.
- Although less common, it is also possible to override a field that is in the superclass. This is done by simply declaring the same field name and type in the subclass.

Review: Duke's Choice Class Hierarchy

Now consider these classes in more detail.



Clothing Class: Part 2

```
 public void display() {
  System.out.println("Item ID: " + getItemID());
  System.out.println("Item description: " + getDesc());
  System.out.println("Item price: " + getPrice());
  System.out.println("Color code: " + getColorCode());
}
public String getDesc (){
    return desc;
}
public double getPrice() {
    return price;
}
public int getItemID() {
    return itemID;
}
protected void setColorCode(char color){
    this.colorCode = color; }
```

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

The code in the slide shows the display method for the Clothing superclass and also some of the getter methods and one of the setter methods. The remaining getter and setter methods are not shown here.

Of course, this display method prints out only the fields that exist in Clothing. You would need to override the display method in Shirt in order to display all of the Shirt fields.

Shirt Class: Part 2

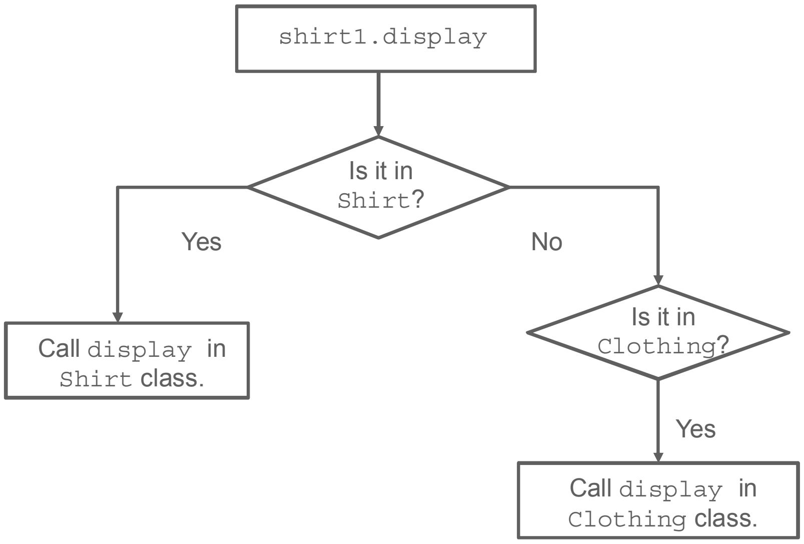
```
// These methods override the methods in Clothing
 public void display() {
    System.out.println("Shirt ID: " + getItemID());
    System.out.println("Shirt description: " + getDesc());
    System.out.println("Shirt price: " + getPrice());
    System.out.println("Color code: " + getColorCode());
    System.out.println("Fit: " + getFit());
}

protected void setColorCode(char colorCode) {
   //Code here to check that correct codes used
   super.setColorCode(colorCode);
}
}
```

Call the superclass's version of setColorCode.

Notice that the display method overrides the display method of the superclass and is more specific to the Shirt class because it displays the shirt's fit property.

- The Shirt class does not have access to the private fields of the Clothing class such as itemID, desc, and price. The Shirt class's display method must, therefore, call the public getter methods for these fields. The getter methods originate from the Clothing superclass.
- The setColorCode method overrides the setColorCode method of the superclass to check whether a valid value is being used for this class. The method then calls the superclass version of the very same method.

Overriding a Method: What Happens at Run Time?



The `shirt01.display` code is called. The Java VM:

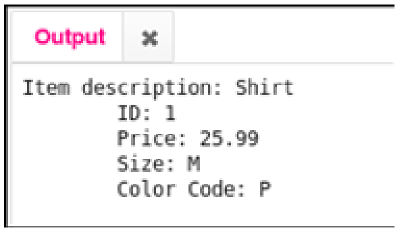
- Looks for `display` in the `Shirt` class
  - If it is implemented in `Shirt`, it calls the `display` in `Shirt`.
  - If it is not implemented in `Shirt`, it looks for a parent class for `Shirt`.
- If there is a parent class (Clothing in this case), it looks for `display` in that class.
  - If it is implemented in `Clothing`, it calls `display` in `Clothing`
  - If it is not implemented in `Clothing`, it looks for a parent class for `Clothing`... and so on.

This description is not intended to exactly portray the mechanism used by the Java VM, but you may find in helpful in thinking about which method implementation gets called in various situations.

Exercise 12-2: Overriding a Method in the Superclass

In this exercise, you override a method in the Item class.

- Override the `display` method to show the additional fields from the `Shirt` class.
- Run the `ShoppingCart` to see the result.



```
Item description: Shirt
        ID: 1
        Price: 25.99
        Size: M
        Color Code: P
```

- In the Java Code Console, access Lessons > 12-Inheritance > Exercise12.
- Follow the instructions below the code editor to override the `display` method.
- Run the `ShoppingCart` class to test your code. Your output should look similar to the screenshot above.
- If you need help, click the Solution link. To go back to your code, click the Exercise link again. Any changes that you have made will have been saved.

Topics

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

Polymorphism

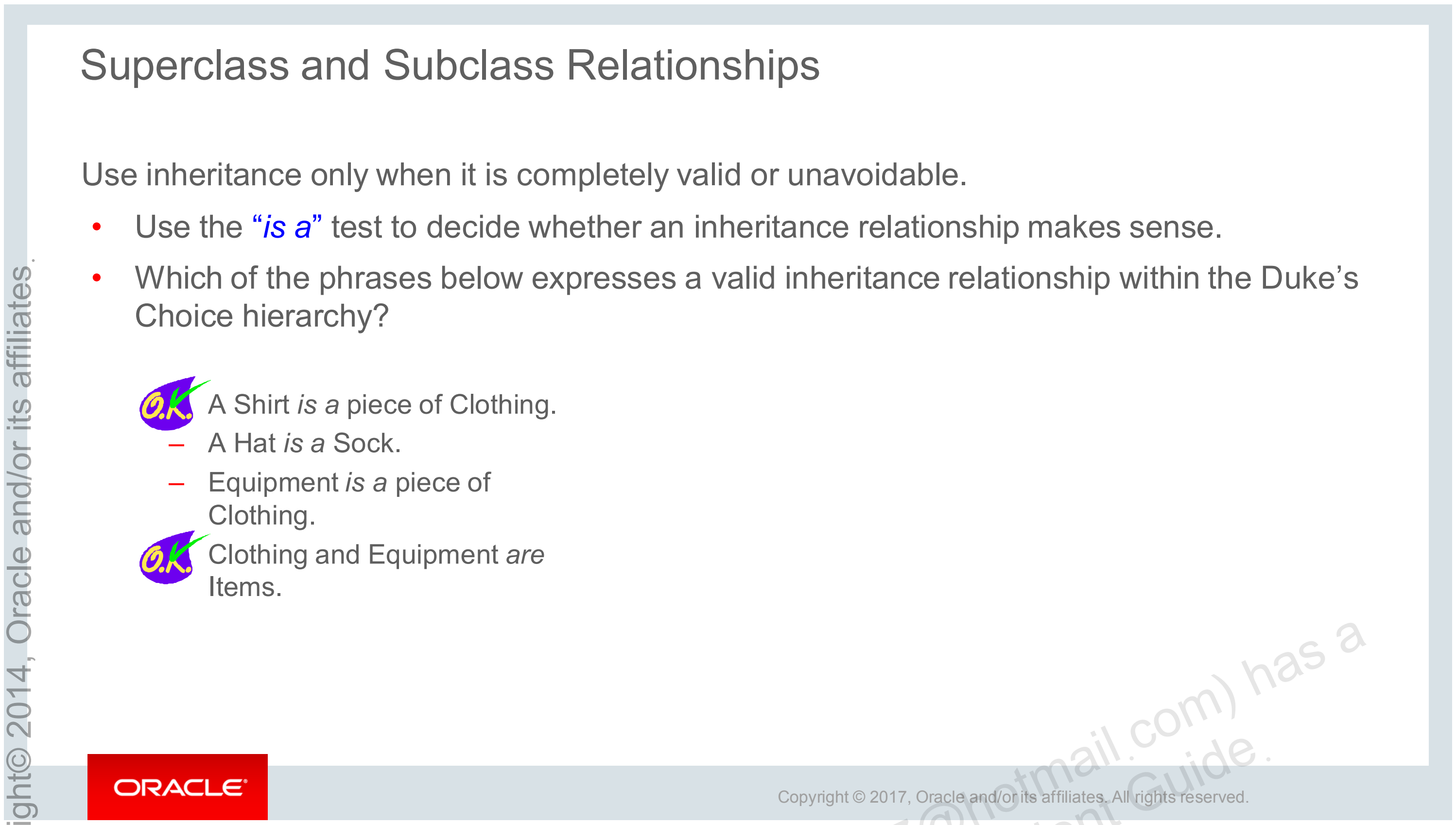
Polymorphism means that the same message to two different objects can have different results.

- "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.

You already used polymorphism when you overrode a method in the superclass, thereby allowing two classes to have the same method name but with a different outcome. In this lesson, we will examine this relationship in more detail and also introduce some other ways of implementing polymorphism.



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?
  - ✓ A Shirt *is a* piece of Clothing.
  - – A Hat *is a* Sock.
  - – Equipment *is a* piece of Clothing.
  - ✓ Clothing and Equipment *are* Items.

In this lesson, you have explored inheritance through an example:

- In the Duke's Choice shopping cart, shirts, trousers, hats, and socks are all types of clothing. So `Clothing` is a good candidate for the superclass to these subclasses (types) of clothing.
- Duke's Choice also sells equipment, but a piece of equipment is *not* a piece of clothing. However, clothing and equipment are both items, so `Item` would be a good candidate for a superclass for these classes.

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.

A very important feature of Java is this ability to use not only the class itself but any superclass of the class as its reference type. In the example shown in the slide, notice that you can refer to both a `Shirt` object and a `Trousers` object with a `Clothing` reference. This means that a reference to a `Shirt` or `Trousers` object can be passed into a method that requires a `Clothing` reference. Or a `Clothing` array can contain references to `Shirt`, `Trousers`, or `Socks` objects as shown below.

- `Clothing[] clothes = {new Shirt(), new Shirt(), new Trousers(), new Socks()};`

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.

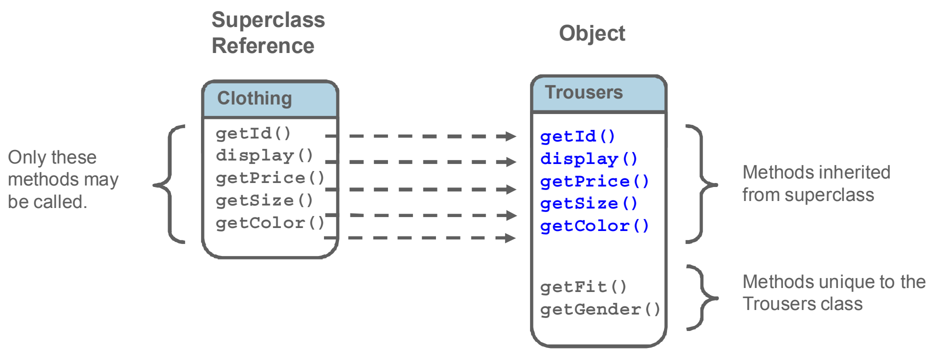
Polymorphism is achieved by invoking one of the methods of the superclass—in this example, the `Clothing` class.

This is a polymorphic method call because the runtime engine does not know, or *need* to know, the type of the object (sometimes called the *runtime* type). The correct method—that is, the method of the actual object—will be invoked.

In the example in the slide, the object could be any subclass of `Clothing`. Recall that some of the subclasses of `Clothing` implemented the `display` and `setColorCode` methods, thereby overriding those methods in the `Clothing` class.

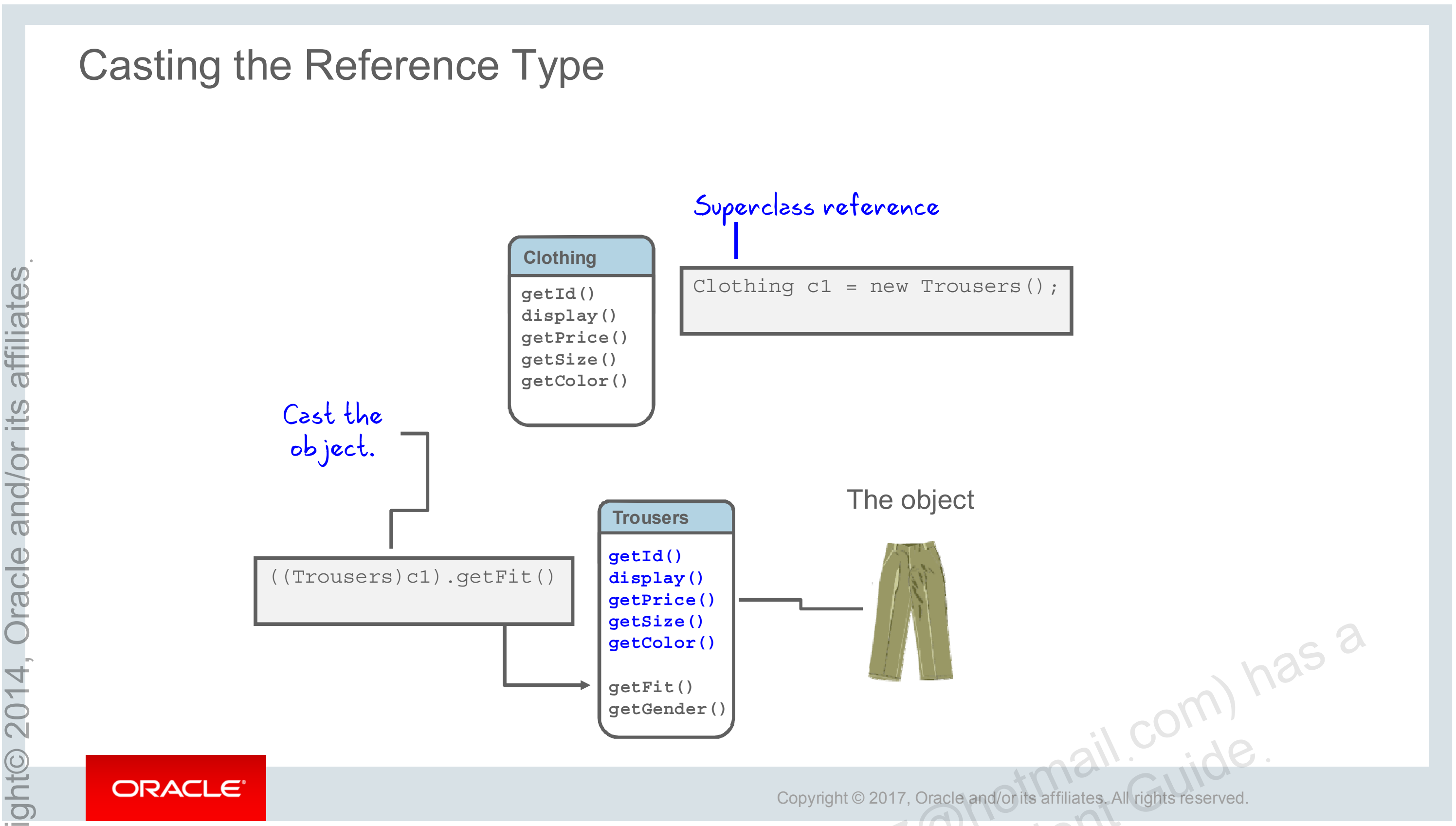
Here you begin to see the benefits of polymorphism. It reduces the amount of duplicate code, and it allows you to use a common reference type for different (but related) subclasses.

Accessing Methods Using a Superclass Reference

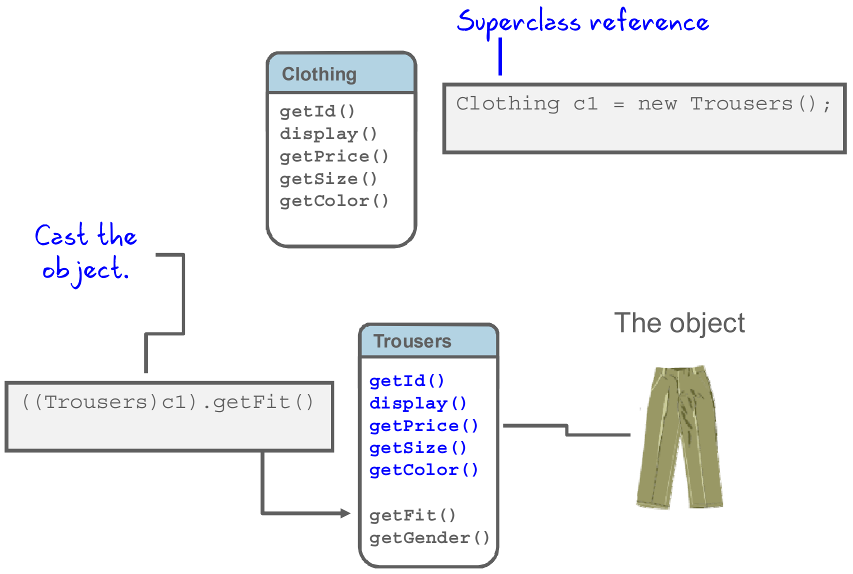


```
Clothing c1 = new Trousers();
c1.getId();   OK
c1.display(); OK
c1.getFit();  NO!
```

Using a reference type `Clothing` does not allow access to the `getFit` or `getGender` method of the `Trousers` object. Usually this is not a problem, because you are most likely to be passing `Clothing` references to methods that do not require access to these methods. For example, a `purchase` method could receive a `Clothing` argument because it needs access only to the `getPrice` method.



Casting the Reference Type

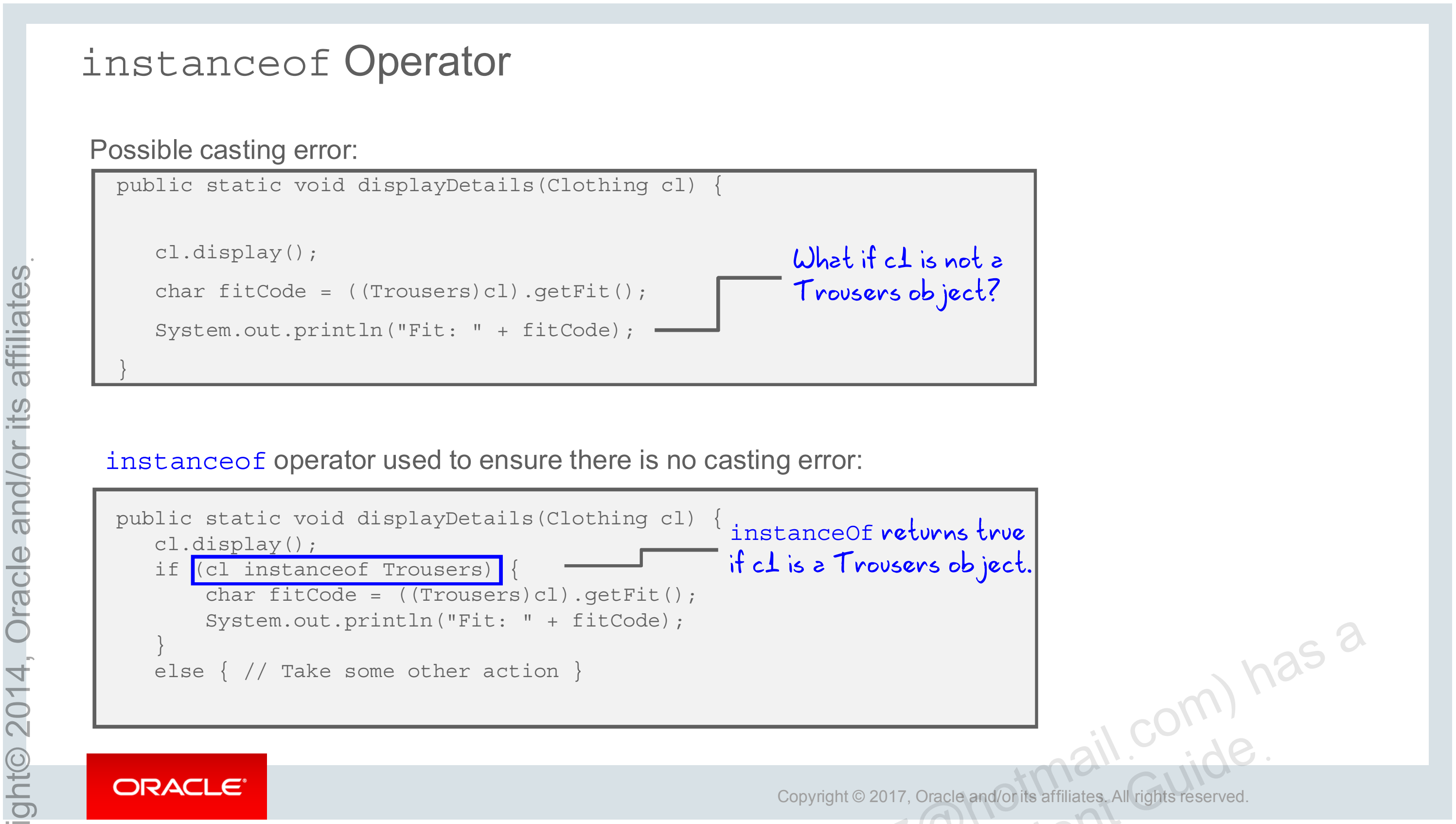


```
Clothing c1 = new Trousers();
```

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

Given that a superclass may not have access to all the methods of the object it is referencing, how can you access those methods? The answer is that you can do so by replacing the superclass reference with:

- A reference that is the same type as the object. The code in this example shows a `Clothing` reference being cast to a `Trousers` reference to access the `getFit` method, which is not accessible via the `Clothing` reference. Note that the inner parentheses around `Trousers` are part of the cast syntax, and the outer parentheses around `(Trousers)c1` are there to apply the cast to the `Clothing` reference variable. Of course, a `Trousers` object would also have access to the nonprivate methods and fields in its superclass.
- An Interface that declares the methods in question and is implemented by the class of the object. Interfaces are covered in the next lesson.



instanceof Operator

Possible casting error:

```
public static void displayDetails(Clothing cl) {

   cl.display();
   char fitCode = ((Trousers)cl).getFit();
   System.out.println("Fit: " + fitCode);
}
```

What if cl is not a Trousers object?

instanceof operator used to ensure there is no casting error:

```
public static void displayDetails(Clothing cl) {
   cl.display();
   if (cl instanceof Trousers) {
      char fitCode = ((Trousers)cl).getFit();
      System.out.println("Fit: " + fitCode);
   }
   else { // Take some other action }
```

instanceOf returns true if cl is a Trousers object.

The first code example in the slide shows a method that is designed to receive an argument of type `Clothing`, and then cast it to `Trousers` to invoke a method that exists only on a `Trousers` object. But it is not possible to know what object type the reference, `cl`, points to. And if it is, for example, a `Shirt`, the attempt to cast it will cause a problem. (It will throw a `ClassCastException`. Throwing exceptions is covered in the lesson titled "Handling Exceptions.")

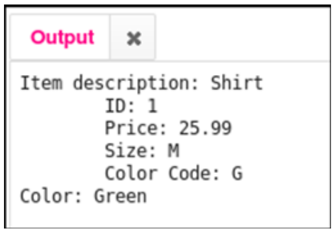
You can code around this potential problem with the code shown in the second example in the slide. Here the `instanceof` operator is used to ensure that `cl` is referencing an object of type `Trousers` before the cast is attempted.

If you think your code requires casting, be aware that there are often ways to design code so that casting is not necessary, and this is usually preferable. But if you do need to cast, you should use `instanceof` to ensure that the cast does not throw a `ClassCastException`.

Exercise 12-3: Using the `instanceof` Operator

In this exercise, you use the `instanceof` operator to test the type of an object before casting it to that type.

- Add a `getColor` method to the `Shirt` class.
- Instantiate a `Shirt` object using an `Item` reference type.
- Call the `display` method on the object.
- Cast the `Item` reference as a Shirt and call `getColor`.



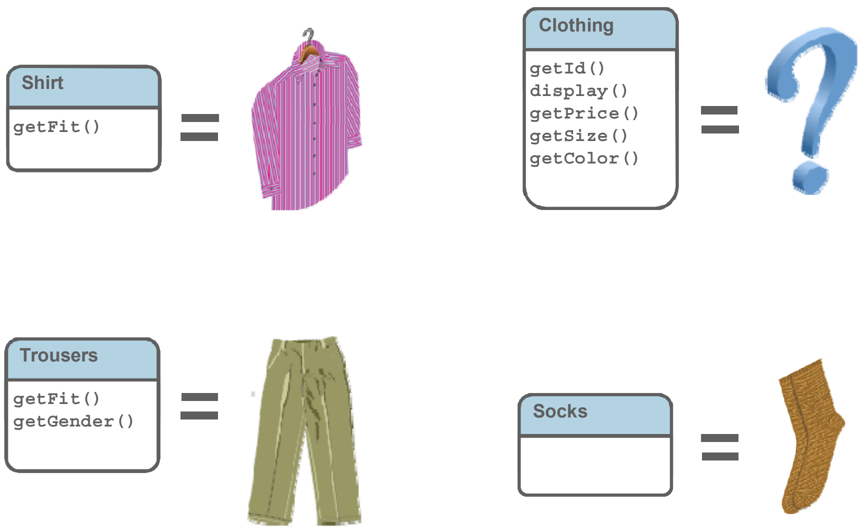
```
Item description: Shirt
        ID: 1
        Price: 25.99
        Size: M
        Color Code: G
Color: Green
```

- In the Java Code Console, access Lessons > 13-Interfaces > Exercise1.
- Follow the instructions below the code editor to modify the `Shirt` class with a new method, and the `ShoppingCart` to instantiate a `Shirt` object with an `Item` reference. Then call the new method by casting the reference as a `Shirt`. You will need to test the object type with the `instanceof` operator first.
- Run the `ShoppingCart` class to test your code. Test the use case where `Item` is not a `Shirt`, too. Your output should look similar to the screenshot above.
- If you need help, click the Solution link. To go back to your code, click the Exercise link again. Any changes that you have made will have been saved.

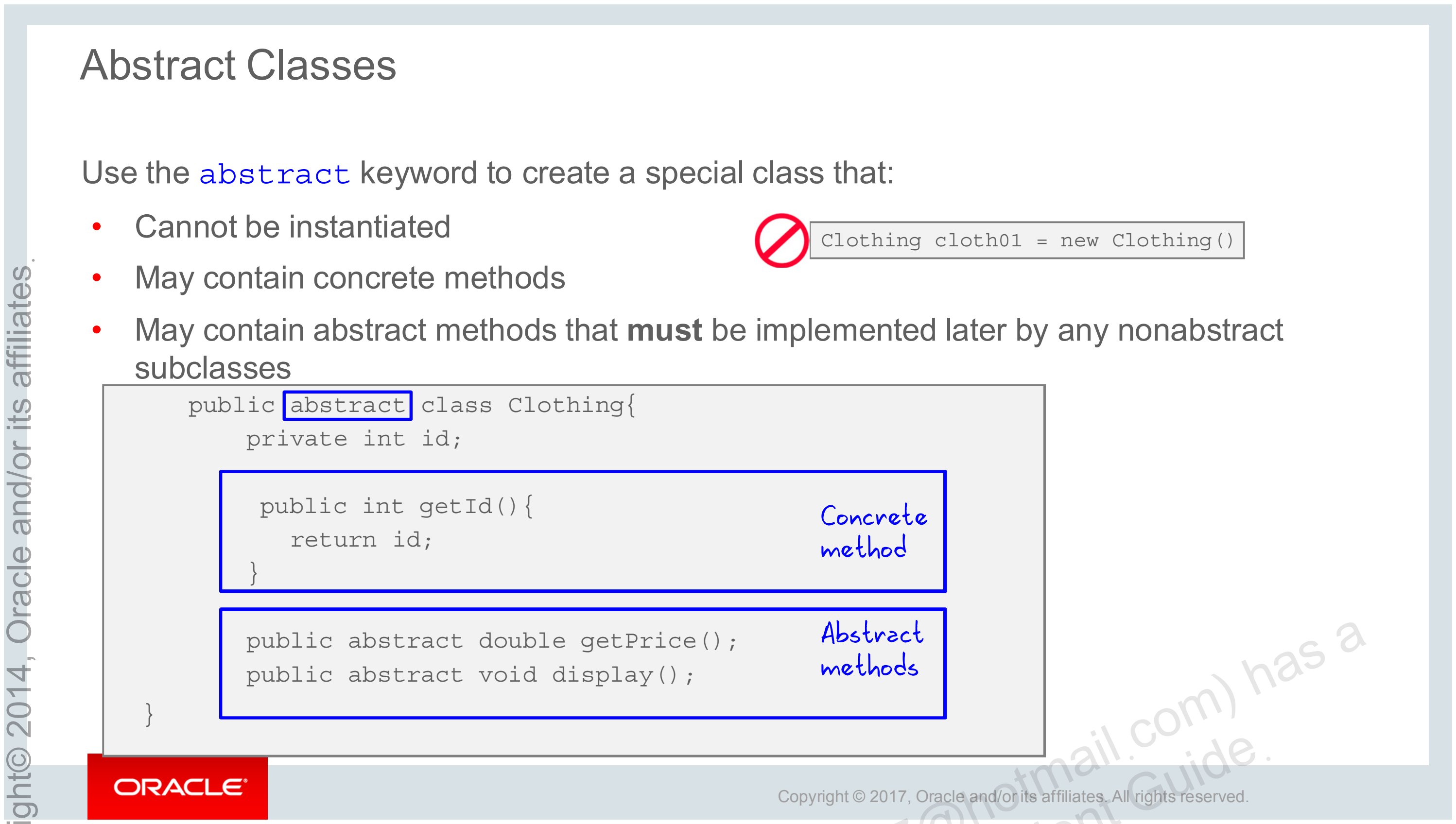
Topics

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

Abstract Classes



Sometimes a superclass makes sense as an object, and sometimes it does not. Duke's Choice carries shirts, socks, and trousers, but it does not have an individual item called "clothing." Also, in the application, the superclass `Clothing` may declare some methods that may be required in each subclass (and thus can be in the superclass), but cannot really be implemented in the superclass.



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();
 }
```

An abstract class cannot be instantiated. In fact, in many cases it would not make sense to instantiate them (Would you ever want to instantiate a `Clothing`?). However these classes can add a helpful layer of abstraction to a class hierarchy. The abstract class imposes a requirement on any subclasses to implement all of its abstract methods. Think of this as a contract between the abstract class and its subclasses.

- The example above has a concrete method, `getId`. This method can be called from the subclass or can be overridden by the subclass.
- It also contains two abstract methods: `getPrice` and `display`. Any subclasses of `Clothing` must implement these two 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());
}}
```

The `Socks` class extends the Clothing class. The `Socks` class implements the abstract `getPrice` and `display` methods from the `Clothing` class. A subclass is free to call any of the concrete methods or newly implemented abstract methods from an abstract superclass, including within the implementation of an inherited abstract method, as shown by the call to `getID` and `getPrice` within the `display` method.

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.

Inheritance enables programmers to put common members (variables and methods) in one class and have other classes *inherit these common members* from this new class.

The class containing members common to several other classes is called the *superclass* or the *parent class*. The classes that inherit from, or extend, the superclass are called *subclasses* or *child classes*.

Inheritance also allows object methods and fields to be referred to by a reference that is the type of the object, the type of any of its superclasses, or an interface that it implements.

Abstract classes can also be used as a superclass. They cannot be instantiated but, by including abstract methods that must be implemented by the subclasses, they impose a specific public interface on the subclasses.

Challenge Questions: Java Puzzle Ball

Bumpers share many of the same properties and behaviors. Is there a way to design code for these objects that minimizes duplicate code and take advantage of polymorphism?

- Common properties: color, shape, x-position, y-position…
- Common behaviors: `flash()`, `chime()`, `destroy()`…

When you have an opportunity to play the game, run any of the inheritance puzzles and answer the questions above.

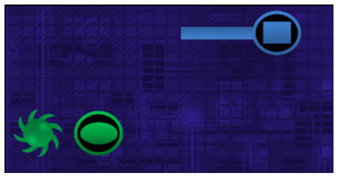
For some possible answers to these questions and more discussion, see "Appendix A: Java Puzzle Ball Challenge Questions Answered."

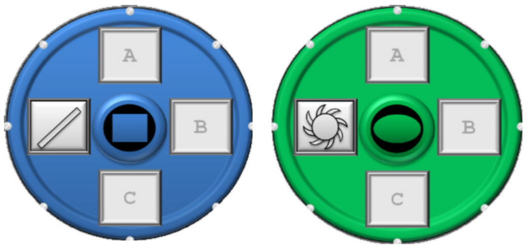
Challenge Questions: Java Puzzle Ball

For a method to be overridden, a subclass must provide a method with the same name and signature as the superclass's method. Only the implementation may differ.

To make overriding possible, which game components best represent:

- A method name and signature?
- A method implantation?





When you have an opportunity to play the game, run any of the inheritance puzzles and answer the questions above.

For some possible answers to these questions and more discussion, see "Appendix A: Java Puzzle Ball Challenge Questions Answered."

Practice 12-1 Overview: Creating a Class Hierarchy

This practice covers rewriting the playGame method so that it will eventually be able to work with GameEvent objects and not just Goal objects, which is the case at present.

Practice 12-2 Overview: Creating a `GameEvent` Hierarchy

This practice covers adding game events to the application. To do this, you create a hierarchy of game event classes that extend an abstract superclass. Some of the game events are:

- `Goal`
- `Pass`
- `Kickoff`

A game event is something that happens and that can be reported as game statistics by the soccer league application. You create an abstract superclass from which all of the game event classes are inherited.