Learning Outcomes:

After doing this exercise, the learner…

1. **has seen inheritance in action**
2. **has seen and used a method override.**
3. **has seen that through inheritance, a super-class reference can refer to a sub-class object**
4. **has seen that through polymorphism, the same reference can call the same method but behave differently depending on the Object to which it refers.**

Task:

Copy and compile the codes below yourself and experiment briefly to discover what each scenario teaches (oh!, and ask questions).

Scenario 1: Learning Inheritance – method "override"

The following is an example which shows inheritance in action in that a `WonkaChocolateBar` "inherits" the method `taste()` from its super class `ChocolateBar`. **Try the code and try to understand how it works.**

```
class ChocolateBar
{
	void taste()
	{
		System.out.println("Mmnn...Chocolate Taste \n");
	}
}

class WonkaChocolateBar extends ChocolateBar
{
	void taste()		//A method "override"
	{
		System.out.println("Oh!... Wonka chocolate! Yum! \n");
	}

	void checkForGoldenTicket()
	{
		//note: use of a "ternary expression" inside the ...println()
		System.out.println( (Math.random() > .2)? "you win" : "you loose" );
	}

}

class RunChocolateBars
{
	public static void main(String[] args)
	{
		ChocolateBar plain = new ChocolateBar();
		plain.taste(); //prints "plainChocolate"

		WonkaChocolateBar wonka = new WonkaChocolateBar();
		wonka.taste(); //prints "…Wonka chocolate…" - method "override"
	}
}
```

Explanation

Above, the `WonkaChocolateBar` class doesn't need to implement a `taste()` method. Because it "`extends ChocolateBar`" it gets a copy for free; that is – it is as if the WonkaChocolateBar (the "sub" class) **had** declared the method `taste()` from the class `ChocolateBar` (the "super" class).
However, declaring a `taste()` method means that this re-definition will execute (it is said to "override" the `taste()` method inherited from the super-class.

Scenario 1, continuation 1: "Re-use" the super-class implementation with `super`

Below, the `super` keyword is used to access the inherited `taste()` method. This will cause the `ChocolateBar.taste()` method to execute, and is a way of "re-using" the implementation; it also serves to show that the method has been inherited. **Try the code and try to understand how it works.**

```
//Now "re-uses" the ChocolateBar.taste() (or "super-class") implementation
class WonkaChocolateBar extends ChocolateBar
{
	void taste()
	{
		super.taste();
		System.out.println("Oh!... Wonka chocolate! Yum! \n");
	}

	void checkForGoldenTicket()
	{
		System.out.println((Math.random() > .2)? "you win"  : "you loose");
	}

}
```

Scenario 1, continuation 2: Understanding the term "interface"

Here we alter the `RunChocolatBars` to show inheritance in action again. **Try the code and try to understand how it works.**

```
class RunChocolateBars
{
	public static void main(String[] args)
	{
		ChocolateBar bar1 = new ChocolateBar();
		ChocolateBar bar2 = new WonkaChocolateBar();
		bar1.taste();
		bar2.taste();
		bar2.checkForGoldenTicket();//will not compile
		// A ChocolateBar reference (super-class) can "refer to"
		//  a WonkaChocolateBar Object (sub-class Object)
		//   but only use the ChocolateBar interface
	}
}
```

Explanation

Through inheritance a "super" class reference can refer to a "sub" class Object. Note – it is the Object type that determines which implementation of `taste()` gets executed (i.e. **not** the reference type).

Scenario 1, continuation 3: Polymorphism

Below, we see an example of "polymorphism" the same reference is used to call the same method, but behaves differently depending on the Object referred to. **Try the code and try to understand how it works.**

```
class RunChocolateBars
{
	//...now showing polymorphism in use with an array of sub-class objects,
	//        referred to by the super-class interface.
	//...also showing dynamic type-checking with Java's 'instanceof' operator
	public static void main(String[] args)
	{
		ChocolateBar[] bars = {new ChocolateBar(), new WonkaChocolateBar()};
		//bars[1].taste();

		for( ChocolateBar b: bars)
		{
			b.taste();

			//if it's a Wonka, check for a Golden ticket
			if( b instanceof WonkaChocolateBar){
				WonkaChocolateBar wonkaRef;
				wonkaRef = (WonkaChocolateBar) b; //Note: a "downcast"

				wonkaRef.checkForGoldenTicket();

			}
		//Note: this 'instanceof' kind of type-checking is not recommended.
		//    It is shown here, purely to to demonstrate the 'instanceof'
		// operator and to illustrate the polymorphism in use.
		}
	}
}
```

Explanation

Above, the `ChocolateBar[]` (read as: "*ChocolateBar array*") is initialized to contain references to two objects: one of type `ChocolateBar` (the super-class) and one of type `WonkaChocolateBar` (the sub-class).

The super-class reference `b` is of type `ChocolateBar` and so, it must refer to an object which either inherits, or has its own implementation of the `taste()` method; and as such, it works. The call through reference `b` to `taste()` is poly morphic – meaning, it takes many forms (or behaves in different ways, depending on the actual *Object* it refers to).