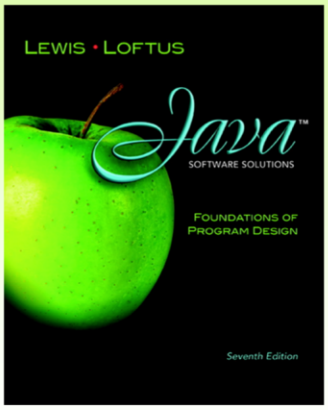
Chapter 10
Polymorphism



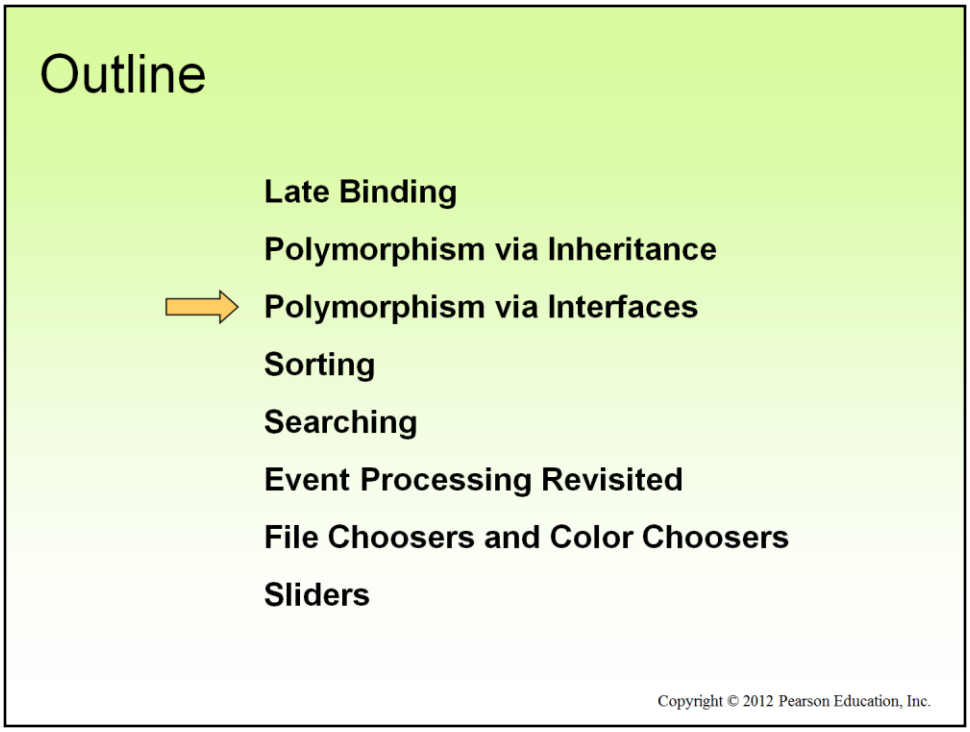
Java Software Solutions
Foundations of Program Design
Seventh Edition

John Lewis
William Loftus

Addison-Wesley
is an imprint of
PEARSON

Polymorphism

- Polymorphism is an object-oriented concept that allows us to create versatile software designs
- Chapter 10 focuses on:
  - defining polymorphism and its benefits
  - using inheritance to create polymorphic references
  - using interfaces to create polymorphic references
  - using polymorphism to implement sorting and searching algorithms
  - additional GUI components



-In addition to base and derived classes, we can also demonstrate and use polymorphism with interfaces

Polymorphism via Interfaces

- Interfaces can be used to set up polymorphic references as well
- Suppose we declare an interface called `Speaker` as follows:

```
public interface Speaker
{
   public void speak();
   public void announce (String str);
}
```


-Recall from our previous studies, that a Java interface contains constants and all **abstract** methods

-It is **implemented** by classes in order to include the functionality declared by the abstract methods

-In other words, classes provide the actual definitions for the abstract methods from the interfaces

-Above is an example of a Java interface with two abstract methods

Polymorphism via Interfaces

- An interface name can be used as the type of an object reference variable:

```
Speaker current;
```

- The `current` reference can be used to point to any object of any class that implements the `Speaker` interface
- The version of `speak` invoked by the following line depends on the type of object that `current` is referencing:

```
current.speak();
```


-Just as we use class types as object reference variables to store addresses, we can also use interface types

-We can use an interface type to store a reference to an object of a class **that implements the interface**

-Note the object reference we are storing **must** be from a class that actually **implements** the interface

-For example, if we had a class named Circle that implements an interface named Drawable, we can do the following:

```
Drawable d = new Circle();
```

-Think of this as “a Circle is Drawable” representing the “is a” relationship similar to what we saw with base and derived classes

Polymorphism via Interfaces

- Now suppose two classes, `Philosopher` and `Dog`, both implement the `Speaker` interface, providing distinct versions of the `speak` method
- In the following code, the first call to `speak` invokes one version and the second invokes another:

```
Speaker guest = new Philospher();
guest.speak();
guest = new Dog();
guest.speak();
```


-Similar to how we used a **base** reference variable, we can assign different objects to an **interface** reference variable

-Depending on the object pointed to (the address stored), the same method call from an interface reference could result in a different execution

-As we saw with inheritance, this is another example of polymorphism; our interface variable is **polymorphic**

-In the example above, the Philosopher and Dog class **both** implement the same interface named Speaker

-As such, we can use an interface reference variable to store both types of objects

-The first method call (guest.speak) from this interface reference above will call the method from the Philosopher class

-The second method call (guest.speak) from this interface reference above will call the method from the Dog class

-The same method call (speak) from the same interface reference (guest) results in **different** behaviors during the execution of a program

-This is an example of polymorphism

Polymorphism via Interfaces

- As with class reference types, the compiler will restrict invocations to methods in the interface
- For example, even if `Philosopher` also had a method called `pontificate`, the following would still cause a compiler error:

```
Speaker special = new Philospher();
special.pontificate();  // compiler error
```

- Remember, the compiler bases its rulings on the type of the reference

-When we use an interface variable, we can only call methods that are specified **in** the interface

-If a method existed in the class and not the interface, as in the example above, an error would result

-In these instances, we could type-cast if we wanted to call a method from the class that is not in the interface

-Recall we did the same thing with base and derived classes

-In this example, we would do the following:

```
Philosopher p = (Philosopher)special;
p.pontificate();
```

-We could do the same in one line:

```
((Philosopher)special).pontificate();
```

Quick Check

Would the following statements be valid?

```
Speaker first = new Dog();
Philosopher second = new Philosopher();
second.pontificate();
first = second;
```

Quick Check

Would the following statements be valid?

```
Speaker first = new Dog();
Philosopher second = new Philosopher();
second.pontificate();
first = second;
```

Yes, all assignments and method calls are valid as written