Introduction to polymorphism

- **Polymorphism** enables you to “program in the general” rather than “program in the specific.”
- Polymorphism enables you to write programs that **process objects that share the same superclass as if they were all objects of the superclass**; this can simplify programming.

Introduction to polymorphism

- Example: Suppose we create a program that **simulates the movement of several types of animals** for a biological study. Classes **Fish**, **Frog** and **Bird** represent the **three types of animals** under investigation.
- Each class extends **superclass Animal**, which contains a method ***move*** and maintains an animal's ***current location*** as x-y coordinates. Each subclass implements method ***move***.
- A program maintains *an Animal array containing references to objects of the various Animal subclasses*.
- To simulate the animals' movements, the program *sends each object the same message* once per second - namely, *move*.

Introduction to polymorphism

- Each specific type of **Animal** responds to a move message in a unique way:
  - a **Fish** might swim three feet
  - a **Frog** might jump five feet
  - a **Bird** might fly ten feet.
- The program issues the same message (i.e., move) to each animal object, but each object knows how to modify its x-y coordinates appropriately for its specific type of movement.
- Relying on each object to know how to "do the right thing" in response to the same method call is the **key concept of polymorphism**.
- The **same message sent to a variety of objects has "many forms" of results** - hence the term polymorphism.

Introduction to polymorphism

- With polymorphism, we can **design and implement systems that are easily extensible**.
- New classes can be added with little or no modification to the **general portions** of the program, as long as the new classes are part of the inheritance hierarchy that the program processes generically.
- The new classes simply **"plug right in."**
- The only parts of a program that must be altered to accommodate new classes are those that require direct knowledge of the new classes that we add to the hierarchy.

Introduction to polymorphism

- Once a class implements an interface, all objects of that class have an **is-a relationship with the interface type**, and all objects of the class are guaranteed to provide the functionality described by the interface.
- This is true of all subclasses of that class as well.
- **Interfaces** are particularly **useful for assigning common functionality to possibly unrelated classes**.
- Allows objects of unrelated classes to be processed **polymorphically** - objects of classes that implement the same interface can respond to all of the interface method calls.

Polymorphism Examples

- Example: **Quadrilaterals**
- If **Rectangle** is derived from **Quadrilateral**, then a **Rectangle** object is a more specific version of a **Quadrilateral**.
- Any operation that can be performed on a **Quadrilateral** can also be performed on a **Rectangle**.
- These operations can also be performed on other **Quadrilaterals**, such as **Squares**, **Parallelograms** and **Trapezoids**.
- Polymorphism occurs when a program invokes a method through a superclass **Quadrilateral** variable - at execution time, the correct subclass version of the method is called, based on the type of the reference stored in the superclass variable.

Polymorphism Examples (Cont.)

- Example: **Space Objects in a Video Game**
- A video game manipulates objects of classes **Martian**, **Venusian**, **Plutonian**, **SpaceShip** and **LaserBeam**. Each inherits from **SpaceObject** and overrides its ***draw*** method.
- A screen manager maintains a **collection of references to objects** of the various classes and periodically sends each object the same message—namely, draw.
- Each object responds in a unique way.
- A **Martian** object might draw itself in red with green eyes and the appropriate number of antennae.
- A **SpaceShip** object might draw itself as a bright silver flying saucer.
- A **LaserBeam** object might draw itself as a bright red beam across the screen.
- The same message (in this case, draw) sent to a variety of objects has “many forms” of results.

Polymorphism Examples (Cont.)

- A screen manager might use polymorphism to facilitate **adding new classes to a system with minimal modifications to the system's code**.
- To add new objects to our video game:
  - Build a class that extends SpaceObject and provides its own *draw* method implementation.
  - When objects of that class appear in the SpaceObject collection, the screen-manager code invokes method *draw*, exactly as it does for every other object in the collection, regardless of its type.
- So the new objects simply "plug right in" without any modification of the screen manager code by the programmer.

Demonstrating Polymorphic Behavior

- In the next example, we aim a **superclass reference at a subclass object.**
- Invoking a method on a subclass object via a superclass reference invokes the subclass functionality
- The **type of the referenced object**, not the type of the variable, **determines which method is called**.
- This example demonstrates that an object of a subclass can be treated as an object of its superclass, enabling various interesting manipulations.
- A program can create an array of superclass variables that refer to objects of many subclass types.
- Allowed because each subclass object is an object of its superclass.

Demonstrating Polymorphic Behavior (Cont.)

- A superclass object cannot be treated as a subclass object, because a superclass object is not an object of any of its subclasses.
- The **is-a relationship applies only up the hierarchy** from a subclass to its direct (and indirect) superclasses, and not down the hierarchy.
- The Java compiler does allow the assignment of a superclass reference to a subclass variable if you **explicitly cast the superclass reference to the subclass type**.
- A technique known as **downcasting** that enables a program to invoke subclass methods that are not in the superclass.

Demonstrating Polymorphic Behavior (Cont.)

- When a superclass variable contains a reference to a subclass object, and that reference is used to call a method, the **subclass version of the method is called**.
- The Java compiler allows this “crossover” because an object of a subclass is an object of its superclass (but not vice versa).
- When the compiler encounters a method call made through a variable, the compiler **determines if the method can be called by checking the variable’s class type**.
- If that class contains the proper method declaration (or inherits one), the call is compiled.
- At execution time, the type of the object to which the variable refers determines the actual method to use.
- This process is called **dynamic binding**.

Abstract Classes and Methods

- **Abstract classes**
- Sometimes it's useful to declare classes for which you never intend to create objects.
- Used only **as superclasses in inheritance hierarchies**, so they are sometimes called abstract superclasses.
- Cannot be used to instantiate objects **- abstract classes are incomplete**.
- Subclasses must declare the "missing pieces" to become "concrete" classes, from which you can instantiate objects; otherwise, these subclasses, too, will be abstract.
- An abstract class provides a superclass from which other classes can inherit and thus share a common design.

Abstract Classes and Methods (Cont.)

- Classes that can be used to instantiate objects are called **concrete classes**.
- Such classes **provide implementations of every method they declare** (some of the implementations can be inherited).
- Abstract superclasses are too general to create real objects - they specify only what is common among subclasses.
- Concrete classes provide the specifics that make it reasonable to instantiate objects.
- Not all hierarchies contain abstract classes.

Abstract Classes and Methods (Cont.)

- Programmers often write client code that uses only abstract superclass types to reduce client code's dependencies on a range of subclass types.
- You can write a method with a parameter of an abstract superclass type.
- When called, such a method can receive an object of any concrete class that directly or indirectly extends the superclass specified as the parameter's type.
- Abstract classes sometimes constitute several levels of a hierarchy.

Abstract Classes and Methods (Cont.)

- You make a class abstract by declaring it with keyword abstract.
- An abstract class normally contains one or more abstract methods.
- An abstract method is an instance method with keyword abstract in its declaration, as in
- public abstract void draw(); // abstract method
- Abstract methods **do not provide implementations**.
- A class that contains abstract methods must be an abstract class even if that class contains some concrete (nonabstract) methods.
- Each concrete subclass of an abstract superclass also **must provide concrete implementations** of each of the superclass's abstract methods.
- **Constructors and static methods** cannot be declared abstract.

Abstract Classes and Methods (Cont.)

- **Cannot instantiate objects of abstract superclasses**, but you can use abstract superclasses to declare variables
- These can hold references to objects of any concrete class derived from those abstract superclasses.
- We'll use such variables to manipulate subclass objects polymorphically.
- Can use abstract superclass names to invoke static methods declared in those abstract superclasses.

Abstract Classes and Methods (Cont.)

- Polymorphism is particularly effective for implementing so-called **layered software systems**.
- Example: Operating systems and device drivers.
- Commands to read or write data from and to devices may have a certain uniformity.
- Device drivers control all communication between the operating system and the devices.
- A write message sent to a device-driver object is interpreted in the context of that driver and how it manipulates devices of a specific type.
- The write call itself really is no different from the write to any other device in the system - place some number of bytes from memory onto that device.

Abstract Classes and Methods (Cont.)

- An object-oriented operating system might use an abstract superclass to provide an “interface” appropriate for all device drivers.
- Subclasses are formed that all behave similarly.
- The **device-driver methods** are declared as abstract methods in the abstract superclass.
- The implementations of these abstract methods are provided in the subclasses that correspond to the specific types of device drivers.
- New devices are always being developed.
- When you buy a new device, it comes with a device driver provided by the device vendor and is immediately operational after you connect it and install the driver.
- This is another elegant example of how polymorphism makes systems extensible.

Case Study: Payroll System Using Polymorphism

- Use an abstract method and polymorphism to perform payroll calculations based on the type of inheritance hierarchy headed by an employee.
- Enhanced employee inheritance hierarchy requirements:
- *A company pays its employees on a weekly basis. The employees are of four types: Salaried employees are paid a fixed weekly salary regardless of the number of hours worked, hourly employees are paid by the hour and receive overtime pay (i.e., 1.5 times their hourly salary rate) for all hours worked in excess of 40 hours, commission employees are paid a percentage of their sales and base-salaried commission employees receive a base salary plus a percentage of their sales. For the current pay period, the company has decided to reward salaried-commission employees by adding 10% to their base salaries. The company wants you to write a Java application that performs its payroll*

Case Study: Payroll System Using Polymorphism (Cont.)

- abstract class Employee represents the general concept of an employee.
- **Subclasses**: SalariedEmployee, CommissionEmployee , HourlyEmployee and BasePlusCommissionEmployee (an indirect subclass)
- Fig. 10.2 shows the inheritance hierarchy for our polymorphic employee-payroll application.
- Abstract class names are italicized in the UML.

Case Study: Payroll System Using Polymorphism (Cont.)

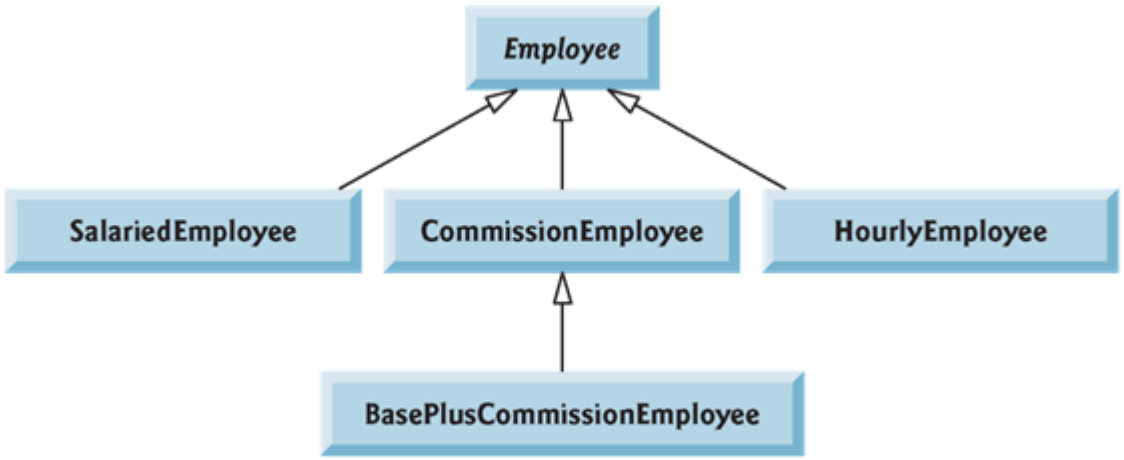

Fig. 10.2 | Employee hierarchy UML class diagram.

Case Study: Payroll System Using Polymorphism (Cont.)

- Abstract superclass **Employee** declares the "interface" to the hierarchy - that is, the set of methods that a program can invoke on all Employee objects.
- We use the term "interface" here in a general sense to refer to the various ways programs can communicate with objects of any Employee subclass.
- Each employee has a *first name*, a *last name* and a *social security number* defined in abstract superclass Employee.

Abstract Superclass Employee

- Class Employee (Fig. 10.4) provides methods *earnings* and *toString*, in addition to the get and set methods that manipulate Employee's instance variables.
- An *earnings* method applies to all employees, but each *earnings* calculation depends on the employee's class.
- An abstract method - there is not enough information to determine what amount earnings should return.
- Each subclass overrides *earnings* with an appropriate implementation.
- Iterate through the array of Employees and call method *earnings* for each Employee subclass object.
- Method calls processed **polymorphically**.

Abstract Superclass Employee (Cont.)

- The diagram in Fig. 10.3 shows each of the five classes in the hierarchy down the left side and methods earnings and *toString* across the top.
- For each class, the diagram shows the desired results of each method.
- Declaring the earnings method abstract indicates that each concrete subclass must provide an appropriate earnings implementation and that a program will be able to use superclass Employee variables to invoke method earnings polymorphically for any type of Employee.

Abstract Superclass Employee (Cont.)

	earnings	toString
Employee	abstract	*firstName lastName* social security number: *SSN*
Salaried-Employee	weeklySalary	salaried employee: *firstName lastName* social security number: *SSN* weekly salary: *weeklySalary*

Hourly-Employee	if (hours <= 40)
	  wage * hours
	else if (hours > 40)
	{
	  40 * wage +
	  (hours - 40) *
	  wage * 1.5
	}
		hourly employee: *firstName lastName*
		social security number: *SSN*
		hourly wage: *wage*; hours worked: *hours*

Commission-Employee	commissionRate * grossSales	commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales*; commission rate: *commissionRate*
BasePlus-Commission-Employee	(commissionRate * grossSales) + baseSalary	base salaried commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales*; commission rate: *commissionRate*; base salary: *baseSalary*

Fig. 10.3 | Polymorphic interface for the `Employee` hierarchy classes.

Payroll example

- **Abstract superclass** – Employee
- **Concrete subclasses**:
  - CommissionEmployee
  - BasePlusCommissionEmployee
  - HourlyEmployee
  - SalariedEmployee

Polymorphic Processing, Operator instanceof and Downcasting

- Fig. 10.9 creates an object of each of the four concrete.
- Manipulates these objects nonpolymorphically, via variables of each object's own type, then polymorphically, using an array of Employee variables.
- While processing the objects polymorphically, the program increases the base salary of each BasePlusCommissionEmployee by 10%
- Requires determining the object's type at execution time.
- Finally, the program polymorphically determines and outputs the type of each object in the Employee array.

Polymorphic Processing, Operator instanceof and Downcasting (Cont.)

- All calls to method *toString* and *earnings* are resolved at execution time, based on the type of the object to which currentEmployee refers.
- Known as **dynamic binding** or **late binding**.
- Java decides which class's *toString* method to call at execution time rather than at compile time
- A **superclass reference** can be used to invoke only methods of the superclass - the subclass method implementations are invoked polymorphically.
- Attempting to invoke a subclass-only method directly on a superclass reference is a compilation error.

Polymorphic Processing, Operator *instanceof* and Downcasting (Cont.)

- Every object *knows its own class* and can access this information through the getClass method, which all classes inherit from class Object.
  - The getClass method returns an object of type Class (from package java.lang), which contains information about the object's type, including its class name.
  - The result of the getClass call is used to invoke getName to get the object's class name.

Summary of the Allowed Assignments Between Superclass and Subclass Variables

- There are three proper ways to assign superclass and subclass references to variables of superclass and subclass types.
  - Assigning a **superclass reference to a superclass variable** is straightforward.
  - Assigning **a subclass reference to a subclass variable** is straightfor-ward.
  - Assigning **a subclass reference to a superclass variable** is safe, because the subclass object is an object of its superclass.
- The superclass variable can be used to refer only to superclass members.
- If this code refers to subclass-only members through the superclass variable, the compiler reports errors.

final Methods and Classes

- A final method in a superclass cannot be overridden in a subclass.
- Methods that are declared private are implicitly final, because it's not possible to override them in a subclass.
- Methods that are declared static are implicitly final.
- A final method's declaration can never change, so all subclasses use the same method implementation, and calls to final methods are resolved at compile time - this is known as **static binding**.

final Methods and Classes (Cont.)

- A final class cannot be extended to create a subclass.
- All methods in a final class are implicitly final.
- Class **String** is an example of a final class.
- If you were allowed to create a subclass of **String**, objects of that subclass could be used wherever Strings are expected.
- Since class **String** cannot be extended, programs that use Strings can rely on the functionality of String objects as specified in the Java API.
- Making the class final also prevents programmers from creating subclasses that might bypass security restrictions.

A Deeper Explanation of Issues with Calling Methods from Constructors

- Do not call **overridable methods** from constructors.
- When creating a subclass object, this could lead to an overridden method being called before the subclass object is fully initialized.
- Recall that when you construct a subclass object, its constructor first calls one of the direct superclass's constructors.
- If the superclass constructor calls an overridable method, the subclass's version of that method will be called by the superclass constructor - before the subclass constructor's body has a chance to execute.
- This could lead to **subtle, difficult-to-detect errors** if the subclass method that was called depends on initialization that has not yet been performed in the subclass constructor's body.
- It's acceptable to call a static method from a constructor.

Creating and Using Interfaces

- Our next example reexamines the **payroll system** of Section 10.5.
- Suppose that the company involved wishes to perform several accounting operations in a single accounts payable application
- Calculating the earnings that must be paid to each employee
- Calculate the payment due on each of several invoices (i.e., bills for goods purchased)
- Both operations have to do with obtaining some kind of payment amount.
- For an employee, the payment refers to the employee's earnings.
- For an invoice, the payment refers to the total cost of the goods listed on the invoice.

Creating and Using Interfaces (Cont.)

- Interfaces offer a capability requiring that **unrelated classes implement a set of common methods**.
- Interfaces define and standardize the ways in which things such as people and systems can interact with one another.
- Example: The controls on a radio serve as an interface between radio users and a radio's internal components.
- Can perform only a limited set of operations (e.g., change the station, adjust the volume, choose between AM and FM)
- Different radios may implement the controls in different ways (e.g., using push buttons, dials, voice commands).

Creating and Using Interfaces (Cont.)

- The interface **specifies what operations** a radio must permit users to perform but does not specify how the operations are performed.
- A Java interface **describes a set of methods that can be called on an object**.

Creating and Using Interfaces (Cont.)

- An interface declaration begins with the keyword interface and contains only constants and abstract methods.
- All interface members must be public.
- Interfaces may not specify any implementation details, such as concrete method declarations and instance variables.
- All methods declared in an interface are **implicitly public abstract methods**.
- All fields are **implicitly public**, static and final.

Creating and Using Interfaces (Cont.)

- To use an interface, **a concrete class must specify that it implements the interface** and must **declare each method in the interface with specified signature**.
- Add the implements keyword and the name of the interface to the end of your class declaration's first line.
- A class that does not implement all the methods of the interface is an **abstract class** and must be declared abstract.
- Implementing an interface is like signing a contract with the compiler that states, "I will declare all the methods specified by the interface or I will declare my class abstract."

Creating and Using Interfaces (Cont.)

- An interface is often used when disparate classes (i.e., unrelated classes) need to share common methods and constants.
- Allows objects of unrelated classes to be processed polymorphically by responding to the same method calls.
- You can create an interface that describes the desired functionality, then implement this interface in any classes that require that functionality.

Creating and Using Interfaces (Cont.)

- An interface is often used in place of an abstract class when there is no default implementation to inherit - that is, no fields and no default method implementations.
- Like public abstract classes, interfaces are typically public types.
- A public interface must be declared in a file with the same name as the interface and the **.java** filename extension.

Developing a Payable Hierarchy

- Next example builds an application that can determine payments for employees and invoices alike.
- Classes Invoice and **Employee** both represent things for which the company must be able to calculate a payment amount.
- Both classes implement the **Payable** interface, so a program can invoke method *getPaymentAmount* on *Invoice* objects and *Employee* objects alike.
- Enables the polymorphic processing of Invoices and Employees.

Developing a Payable Hierarchy (Cont.)

- Fig. 10.10 shows the accounts payable hierarchy.
- The UML distinguishes an interface from other classes by placing «interface» above the interface name.
- The UML expresses the relationship between a class and an interface through a realization.
- A class is said to “realize,” or implement, the methods of an interface.
- A class diagram models a realization as a dashed arrow with a hollow arrowhead pointing from the implementing class to the interface.
- A subclass inherits its superclass’s realization relationships.

Developing a Payable Hierarchy (Cont.)

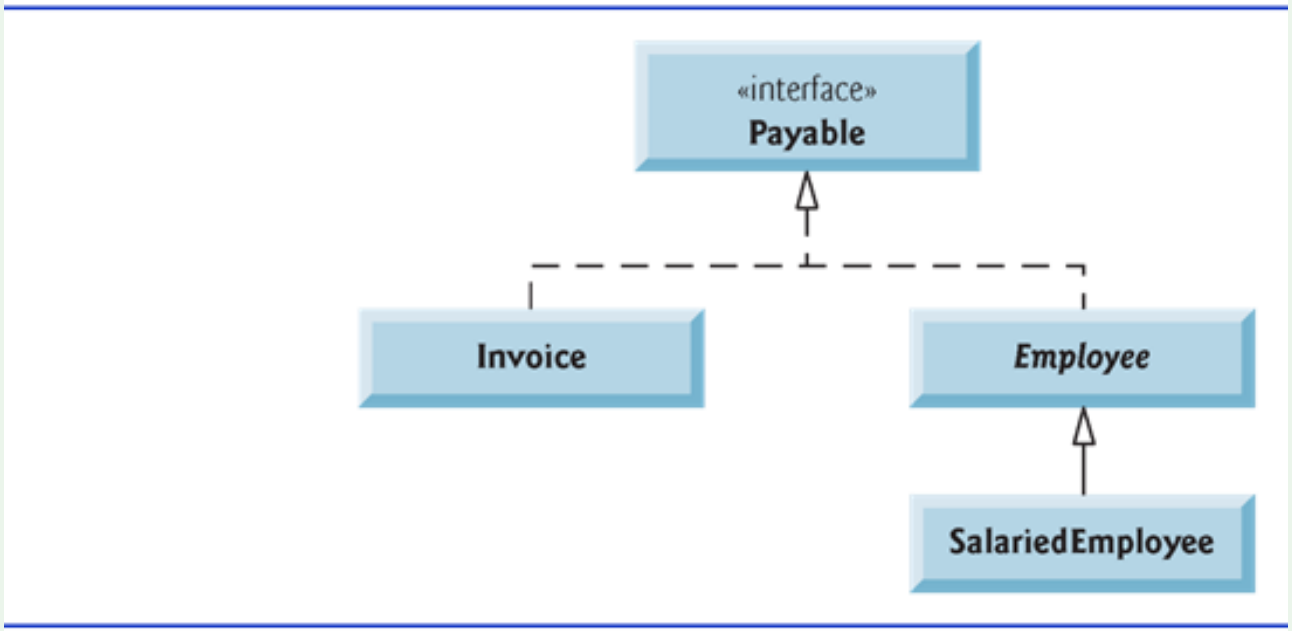

Fig. 10.10 | Payable hierarchy UML class diagram.

Interface Payable

- Fig. 10.11 shows the declaration of interface Payable.
- Interface methods are always public and abstract, so they do not need to be declared as such.
- Interfaces can have any number of methods.
- Interfaces may also contain final and static. constants

Class Invoice

- Java does not allow subclasses to inherit from more than one superclass, but it allows a class to inherit from one superclass and implement as many interfaces as it needs.
- To implement more than one interface, use a comma-separated list of interface names after keyword implements in the class declaration, as in:

```
public class ClassName extends SuperclassName
   implements FirstInterface, SecondInterface, …
```

Modifying Class Employee to Implement Interface Payable

- When a class implements an interface, it makes a contract with the compiler
- The class will implement each method in the interface or the class will be declared abstract.
- Because class **Employee** does not provide a *getPaymentAmount* method, the class must be declared abstract.
- Any concrete subclass of the abstract class must implement the interface methods to fulfill the contract.
- If the subclass does not do so, it too must be declared abstract.
- Each direct **Employee** subclass inherits the superclass's contract to implement method *getPaymentAmount* and thus must implement this method to become a concrete class for which objects can be instantiated.

Modifying Class SalariedEmployee for Use in the Payable Hierarchy

- Figure 10.14 contains a modified **SalariedEmployee** class that extends **Employee** and fulfills superclass Employee's contract to implement **Payable** method *getPayment-Amount*.

Modifying Class SalariedEmployee for Use in the Payable Hierarchy (Cont.)

- Objects of any subclasses of a class that implements an interface can also be thought of as objects of the interface type.
- Thus, just as we can assign the reference of a **SalariedEmployee** object to a superclass **Employee** variable, we can assign the reference of a SalariedEmployee object to an interface **Payable** variable.
- Invoice implements **Payable**, so an Invoice object also is a **Payable** object, and we can assign the reference of an Invoice object to a **Payable** variable.

Using Interface Payable to Process Invoices and Employees Polymorphically

- **PayableInterfaceTest** (Fig. 10.15) illustrates that interface **Payable** can be used to process a set of Invoices and Employees polymorphically in a single application.

Java SE 8 Interface Enhancements

- Prior to Java SE 8, interface methods could be only public abstract methods.
- An interface specified what operations an implementing class must perform but not how the class should perform them.
- In Java SE 8, **interfaces also may contain public default methods with concrete default implementations that** specify how operations are performed when an implementing class does not override the methods.
- If a class implements such an interface, the class also receives the interface's default implementations (if any).
- To declare a default method, place the keyword default before the method's return type and provide a concrete method implementation.

default Interface Methods (Cont.)

- Adding Methods to Existing Interfaces
- Any class that implements the original interface will not break when a default method is added.
- The class simply receives the new default method.
- When a class implements a Java SE 8 interface, the class “signs a contract” with the compiler that says,
- “I will declare all the abstract methods specified by the interface or I will declare my class abstract”
- The implementing class is not required to override the interface’s default methods, but it can if necessary.

default Interface Methods (Cont.)

- Interfaces vs. abstract Classes
- Prior to Java SE 8, an interface was typically used (rather than an abstract class) when there were no implementation details to inherit - no fields and no method implementations.
- With default methods, you can instead declare common method implementations in interfaces, which gives you more flexibility in designing your classes.

static Interface Methods (Cont.)

- Prior to Java SE 8, it was common to associate with an interface a class containing static helper methods for working with objects that implemented the interface.
- In Chapter 16, you'll learn about class Collections which contains many static helper methods for working with objects that implement interfaces **Collection**, **List**, **Set** and more.
- Collections method sort can sort objects of any class that implements interface **List**.
- With static interface methods, such helper methods can now be declared directly in interfaces rather than in separate classes.

References

- Textbook
- Java documentation
  - https://docs.oracle.com/javase/tutorial/java/IandI/index.html