Chapter 9
Inheritance and Polymorphism

Extending classes and implementing interfaces
There are two hierarchies in OOP that help manage the complexity of large software systems.

- An **object hierarchy** introduced in Chapter 4 that uses composition (aggregation) to express complex classes in terms of simpler ones.

- A **class hierarchy** that is defined by inheritance where we define subclasses which inherit all the functionality of their parents and can modify this functionality or introduce new functionality.
What is inheritance?

★ Inheritance defines a relationship between two classes

- One is called the superclass or parent class
- The other is called the subclass or child class

★ Each subclass can also have subclasses so we get an inheritance hierarchy in which each class is a subclass of the class above it in the hierarchy

★ In Java keyword `extends` means "is a subclass of"
### Importance of inheritance

- Each subclass can be constructed incrementally from its superclass.

- This promotes code reuse since a subclass only specifies how its objects differ from those of its parent class.

- These differences are of three types:
  - New (additional) data fields
  - New (additional) methods
  - New versions of existing superclass methods
Domestic animal hierarchy

- DomesticAnimal
  - Dog
    - Terrier
    - Bulldog
  - Cat
    - Persion
    - Cheshire
BankAccount hierarchy

Every Java class inherits from the top level `Object` class. This class is the ultimate parent of every class.

We will write this class later.
JointBankAccount

BankAccount has data fields for an account number, owner name, and balance

JointBankAccount will do the following

- provide its own constructors
- inherit all BankAccount methods
- provide a new data field for a joint owner name
- provide a new get method for the joint owner
- override toString to provide for joint owner name
Employee hierarchy

- Employee
  - Manager
  - HourlyWorker
  - PartTimeWorker
  - CommissionWorker

We don't always show the Object class
Everything common to all employees
These classes differ only in how the salary is calculated
Employee example

- The **Employee** class represents everything that is common to all types of employees such as name, id, date hired, etc.

- Such a class is often called a **base class**

- Each type of employee such as manager or hourly worker is represented by a subclass

- Each subclass will provide methods for calculating an employee's gross and net monthly salary
"is-a" and "has-a" relations

Inheritance is often called the "is-a" or "is-a-type-of" relation (reflexive, transitive and not symmetric)

A JointBankAccount object is a type of BankAccount object

Aggregation (Composition) is often called the "has-a" relation

A Circle object "has a" Point object which represents the center of the circle
Template for a subclass

public class SubclassName extends SuperclassName {
    declarations for new data fields, if any
    constructor declarations (never inherited)
    method declarations for new methods, if any
    method declarations for overridden methods, if any
}

Subclass rules (1)

Superclass data fields

- A superclass data field is automatically a data field of any subclass
- There is no need to re-declare it in a subclass
- In fact it is an error if you do re-declare it

Example: The JointBankAccount class inherits the number, name and balance data fields of the BankAccount class
Subclass rules (2)

Access to superclass data fields

- A **public** data field can be directly accessed by any class, subclass or not

- A **private** data field can never be directly accessed by any class, subclass or not

- A **protected** data field can be directly accessed by a subclass, otherwise protected is like private

Example: data fields of **BankAccount** cannot be directly accessed by **JointBankAccount**
Subclass rules (3)

Declaring new data fields

They are declared in the subclass just like any data field

Example: In the JointBankAccount class it is necessary to declare a new data field for the joint owner name
Constructors are not inherited

- Each subclass must declare its own constructors
- In doing so a subclass constructor may call a superclass constructor as its **first** statement using the syntax `super(actualArgList)` to construct the superclass part of an object

**Example:** In the `JointBankAccount` class we have a new data field so it is necessary to declare a new constructor
Subclass rules (5)

Inheriting superclass methods

- All public and protected superclass methods are automatically inherited by a subclass

Example: In the `JointBankAccount` class we automatically inherit all methods:

- `getNumber`, `getName`, `getBalance`, `withdraw`, `deposit`, and `toString` methods are inherited
Overriding superclass methods

- Any public or protected superclass method can be re-defined (overridden) in a subclass to provide additional or new functionality.

- In doing so, the subclass method can call the superclass version of the method using the syntax `super.methodName(actualArgList)`.

- If you don't override a superclass method then do not declare it again in the subclass (this is an error).

Example: override `toString` in `JointBankAccount`
Subclass rules (7)

Declaring new methods

A subclass can declare new methods

Example: In the `JointBankAccount` class we need to

- override the `toString` method to include the joint owner
- provide a new method called `getJointName` that returns the name of the joint owner
public class MyGraphicsFrame extends JPanel {
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        ...
    }
    // other methods
}
MyGraphicsClass hierarchy
From Chapter 3 CircleCalculator calculates the area and circumference of a circle given the radius. Now split it into a hierarchy of two classes

CircleCalculatorA
- Just do the area part given the radius

CircleCalculatorB
- Calculate area and circumference by extending the CircleCalculatorA class
CircleCalculatorA class

```java
package chapter9.geometry;
public class CircleCalculatorA {
    protected double radius;
    private double area;

    public CircleCalculatorA(double r) {
        radius = r;
        area = Math.PI * radius * radius;
    }

    public double getRadius() {
        return radius;
    }

    public double getArea() {
        return area;
    }
}
```

- Will need it in subclass
- Don't need it in subclass
- These methods are available in subclasses
package chapter9.geometry;
public class CircleCalculatorB extends CircleCalculatorA {

    private double circumference;

    public CircleCalculatorB(double r) {
        super(r);
        circumference = 2.0 * Math.PI * radius;
    }

    public double getCircumference() {
        return circumference;
    }
}

CircleCalculatorB class
new data field
new method
let superclass do its part
protected
package chapter9.geometry;
public class CircleCalculatorTester
{
    public void doTest()
    {
        CircleCalculatorB circle =
            new CircleCalculatorB(3.0);
        double radius = circle.getRadius();
        double area = circle.getArea();
        double circ = circle.getCircumference();
        System.out.println("Radius: " + radius);
        System.out.println("Area: " + area);
        System.out.println("Circumference: " + circ);
    }
    public static void main(String[] args)
    {
        new CircleCalculatorTester().doTest();
    }
}
Inheritance in BlueJ

- Solid arrow points to superclass
- Inherited methods

```
<table>
<thead>
<tr>
<th>Inherited from Object</th>
<th>double getArea() double getRadius()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inspect Remove</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Inherited from Object</th>
<th>double getArea() double getRadius()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inherited from CircleCalculatorA</td>
<td></td>
</tr>
<tr>
<td>Inspect Remove</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Inherited from Object</th>
<th>double getArea() double getRadius()</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
public class BankAccount
{
    private int number;
    private String name;
    private double balance;

    public BankAccount(int accountNumber, String ownerName, double initialBalance) {...}

    public void deposit(double amount) {...}
    public void withdraw(double amount) {...}
    public int getNumber() {...}
    public String getName() {...}
    public double getBalance() {...}
    public String toString() {...}
}
JointBankAccount class (1)

public class JointBankAccount extends BankAccount {
    // new data field for joint owner name goes here
    // constructors go here
    // new getJointName method goes here
    // overridden version of toString goes here
}

The new data field is specified by

private String jointName;
public JointBankAccount(int accountNumber,
   String ownerName, String jointOwnerName,
   double initialBalance)
{
    number = accountNumber;
    name = ownerName;
    balance = initialBalance;
    jointName = jointOwnerName;
}
JointBankAccount class (3)

Constructor (correct version using super)

```java
public JointBankAccount(int accountNumber,
            String ownerName, String jointOwnerName,
            double initialBalance)
{
    super(accountNumber, ownerName, initialBalance);

    jointName = jointOwnerName;
}
```

Ask superclass constructor to do its part
public JointBankAccount extends BankAccount
{
    private String jointName;

    public JointBankAccount(int accountNumber, String ownerName, String jointOwnerName, double initialBalance)
    {
        super(accountNumber, ownerName, initialBalance);
        if (jointOwnerName.equals("") || jointOwnerName == null)
            throw new IllegalArgumentException(...);
        this.jointName = jointOwnerName;
    }
}
public String getJointName() {
    return jointName;
}

public String toString() {
    return "JointBankAccount[" +
           super.toString() + ", " +
           jointName + "]";
}
} // end of class JointBankAccount
BlueJ project for inheritance

- New Class...
- Compile

**BlueJ: bank_account**

- BankAccount
- AccountTester
- AccountTester2
- JointBankAccount

**bankAccount : BankAccount**

- **inherited from Object**
  - void deposit(double amount)
  - double getBalance()
  - String getName()
  - int getNumber()
  - String toString()
  - void withdraw(double amount)

- **inherited from BankAccount**
  - String getJointName()
  - String toString()  [redefined in JointBankAccount]

- void deposit(double amount)
- double getBalance()
- String getName()
- int getNumber()
- String toString()
Polymorphism

Polymorphic types
Polymorphic methods
Abstract classes
Polymorphic types

- Polymorphism means many forms

- A hierarchy of classes defined by inheritance is a polymorphic type

- The subclasses are different but we can think of them all as being of a similar type

- Every object is of type `Object`

- "is a", "is a kind of", or "is a type of"
The following two examples do not make use of inheritance: they simply construct two objects and assign their references to variables of the same type.

```java
BankAccount fred = new BankAccount(123, "Fred", 345.50);
JointBankAccount fredMary = new JointBankAccount(345, "Fred", "Mary", 456, 60);
```

The following example uses inheritance to express that the `JointBankAccount` is also a "type of" `BankAccount`.

```java
BankAccount ellenFrank = new JointBankAccount(456, "Ellen", "Frank", 234.50);
```
The following example is illegal because a `BankAccount` object is not a type of `JointBankAccount` object:

```java
JointBankAccount fred =
    new BankAccount(123, "Fred", 345.50);
```

The rule is simple:

- You can assign a subclass reference to a superclass type but not the other way around.
- For example, a `JointBankAccount` is a type of `BankAccount` but the converse is not true.
Type casting (1)

Define the following account

```
JointBankAccount fredMary =
    new JointBankAccount(345,"Fred","Mary",450);
```

Then the following statements are legal

```
String owner = fredMary.getName();
String jointOwner = fredMary.getJointName();
```
Define the following account

```java
BankAccount ellenFrank = new JointBankAccount(345,"Fred","Mary",450.65);
```

Then the following statement is legal

```java
String name = ellenFrank.getName();
```

The following statement is not legal: as a `BankAccount` object `ellenFrank` doesn't have a `getJointName` method

```java
String jointName = ellenFrank.getJointName();
```

It is necessary to do a typecast (note parentheses)

```java
String jointName = ((JointBankAccount)ellenFrank).getJointName();
```
Object amnesia

Define the following account

```java
BankAccount ellenFrank = new JointBankAccount(345, "Fred", "Mary", 450.65);
```

Here the object forgets it is a `JointBankAccount` since it's assigned to a variable of type `BankAccount`. This is called **object amnesia**

It is necessary to do a typecast to remind the object that it is really a `JointBankAccount` object

```java
String jointName = ((JointBankAccount) ellenFrank).getJointName();
```
AccountTester class (1)

```java
package chapter9.bank_account;
import custom_classes.BankAccount;
import custom_classes.JointBankAccount;

public class AccountTester
{
    public void doTest()
    {
        JointBankAccount fredMary = new 
            JointBankAccount(123, "Fred", "Mary", 1000);
        BankAccount ellenFrank = new 
            JointBankAccount(345, "Ellen", "Frank", 1000);
    }
}```
String jointName1 = fredMary.getJointName();
String jointName2 =
((JointBankAccount) ellenFrank).getJointName();

System.out.println("Joint name 1 is " + jointName1);
System.out.println("Joint name 2 is " + jointName2);
} // end of doTest

public static void main(String[] args)
{
    new AccountTester().doTest();
}
}
Example of object amnesia

Point and Circle objects are types of Object so we can make the following assignments

```java
Object p = new Point(3,4);
Object c = new Circle((Point)p, 5);
```

Both objects have now forgotten their original types and it would be necessary to use a typecast

- `p.getX()` and `c.getCenter()` are now illegal
- `((Point p).getX())` and `((Center)c).getCenter()` are legal
Graphics example

We used the following statement in our graphics programs in Chapter 5

```java
Graphics2D g2D = (Graphics2D) g;
```

The variable `g` is a reference to the original graphics class called `Graphics`. In later versions of Java this class was extended to `Graphics2D` which included object oriented graphics called Java2D

Doing it this way allows users to use the original graphics methods or the newer ones simply by typecasting from `Graphics` to `Graphics2D`
Polymorphic methods

In a class hierarchy we can have an instance method that has many different forms, one for each subclass in the hierarchy since each subclass can provide an overridden version of this method.

Such a method is called a polymorphic method.

The standard example is the toString method.
Overridden methods

- Method overriding is not the same as method overloading

- Overloading (same name and class, different arg lists)
  Overriding (same name, same arg lists, different subclasses)

- Method overloading example

  ```java
  public void println()
  public void println(String s)
  public void println(int n)
  ```

  Several methods in the same class can have the same name as long as they have distinguishable argument lists.
In the Java2D graphics classes we used statements such as

```java
Point2D.Double bottomRight =
    new Point2D.Double(300.0,200.0);
```

However `Point2D` is the superclass of `Point2D.Double` so we can write

```java
Point2D bottomRight =
    new Point2D.Double(300.0,200.0);
```
Account transfer method

Problem: write a method that has two account references as arguments, one for the "from" account and one for the "to" account, and an argument for an amount to transfer from the "from" account to the "to" account.

The method should work for both BankAccount and JointBankAccount objects

Solution: use polymorphism
Non-polymorphic solution

Without polymorphism we need 4 methods

```java
public void transfer(BankAccount from,
                     BankAccount to, double amount) {...}

public void transfer(BankAccount from,
                     JointBankAccount to, double amount) {...}

public void transfer(JointBankAccount from,
                     BankAccount to, double amount) {...}

public void transfer(JointBankAccount from,
                     JointBankAccount to, double amount) {...}
```
Polymorphic solution

With polymorphism we need only one method

```java
public void transfer(BankAccount from,
                    BankAccount to, double amount)
{
    from.withdraw(amount);
    to.deposit(amount);
}
```

use base class here

Here the `withdraw` and `deposit` methods are polymorphic within the bank account hierarchy

Every `JointBankAccount` object "is a type of" `BankAccount` object
Polymorphic toString method

In the bank account hierarchy there are three versions of the `toString` method

- the default one in the `Object` class. It would have been used if we didn't override `toString` in any subclasses
- the one in the `BankAccount` class
- the one in the `JointBankAccount` class

The Java run-time system chooses the correct version at run-time
package chapter9.bank_account;
import custom_classes.BankAccount;
import custom_classes.JointBankAccount;
public class AccountTester2
{
    public void doTest()
    {
        BankAccount fred = new BankAccount(456, "Fred", 500);
        JointBankAccount fredMary = new JointBankAccount(123, "Fred", "Mary", 1000);
        BankAccount ellenFrank = new JointBankAccount(345, "Ellen", "Frank", 1000);
        System.out.println(fred);
        System.out.println(fredMary);
        System.out.println(ellenFrank);
    }
    public static void main(String[] args)
    {
        new AccountTester2().doTest();
    }
}
AccountTester2 output

Output from the doTest method

BankAccount[456, Fred, 500.0]
JointBankAccount[BankAccount[123, Fred, 1000.0], Mary]
JointBankAccount[BankAccount[345, Ellen, 1000.0], Frank]

Even though the third account is assigned to a superclass reference (static compile-time type is BankAccount) the dynamic run-time type is still JointBankAccount so the toString method in this class is called
Abstract classes and polymorphism

An abstract class is a class that declares at least one method without providing a method body (no implementation).

Example:

```java
abstract public double grossSalary();

abstract public double netSalary();
```

- Abstract keyword indicates method is abstract.
- Note the semicolon.
Purpose of an abstract class

☆ It sits at the top of a class hierarchy and specifies just what should be common to all subclasses

☆ It can also declare one or more abstract methods to be implemented by each subclass

☆ These methods will be polymorphic since each non-abstract subclass will need to provide an implementation of each abstract method.
### Employee class hierarchy

**Employee class**

- Abstract class to encapsulate employee name
- Contains two abstract methods: `grossSalary`, to calculate and return the gross monthly salary, and `netSalary` to calculate and return the net monthly salary after deductions
- Constructor: `public Employee(String name)`
- `toString` method to represent the name
- Each subclass will implement the two abstract methods and the non-abstract `toString` method
Manager subclass

- Employee with a gross monthly salary from which 10% is deducted to get net monthly salary
- Uses this information to implement the abstract methods
- Constructor prototype:

  `public Manager(String name, double salary)`
HourlyWorker subclass

Employee with a gross monthly salary given by hours worked times hourly rate and net salary obtained using a 5% deduction

Uses this information to implement the abstract methods

Constructor prototype:

```java
public HourlyWorker(String name, double hoursWorked, double hourlyRate)
```
PartTimeWorker subclass

- Employee like an hourly worker but with no deductions to get the net salary

- Uses this information to implement the abstract methods

- Constructor prototype:

  ```java
  public PartTimeWorker(String name, double hoursWorked, double hourlyRate)
  ```
CommisionWorker subclass

Employee who receives a base monthly salary with a sales bonus added to get gross salary. From this 10% is deducted to get net salary. Gross salary is base + (monthly sales) * (commissionRatePercent/100)

Uses this information to implement the abstract methods

Constructor prototype:

```java
public CommissionWorker(String name, double salary, double monthlySales, double commissionRate)
```
Employee class

```java
package chapter9.employee;
abstract public class Employee
{
    private String name;

    public Employee(String name)
    {
        this.name = name;
    }

    public String getName()
    {
        return name;
    }

    abstract public double grossSalary();
    abstract public double netSalary();
}
```
package chapter9.employee;
public class Manager extends Employee {
    private double gross;  // gross monthly salary
    private double net;    // net monthly salary

    public Manager(String name, double salary) {
        super(name);
        gross = salary; net = 0.9 * gross;
    }

    public double grossSalary() { return gross; }  
    public double netSalary() { return net; } 

    public String toString() {
        return "Manager[" + "name = " + getName() + ", gross = " + grossSalary() + ", net = " + netSalary() + "]";
    }
}

Manager class
Other subclasses of Employee

Do the other subclasses yourself (see Exercise 9.1)

Each subclass will be like Manager except it will implement the two abstract methods and the toString method in a different way
Employee polymorphism

All classes in the Employee hierarchy have three polymorphic methods:

- `grossSalary`
- `netSalary`
- `toString`

The following EmployeeProcessor class illustrates the importance of polymorphism:

- put some employees in an Employee array and process them polymorphically
package chapter9.employee;
public class EmployeeProcessor
{
    private Employee[] staff;
    private double totalGrossSalary;
    private double totalBenefits;
    private double totalNetSalary;
    public void doTest()
    {
        staff = new Employee[5];
        staff[0] = new Manager("Fred", 800);
        staff[1] = new Manager("Ellen", 700);
        staff[2] = new HourlyWorker("John", 37, 13.50);
        staff[3] = new PartTimeWorker("Gord", 35, 12.75);
        staff[4] = new CommissionWorker("Mary", 400, 15000, 3.5);
totalGrossSalary = 0.0;
totalNetSalary = 0.0;
for (int i = 0; i < staff.length; i++)
{
    totalGrossSalary = totalGrossSalary +
                staff[i].grossSalary();
    totalNetSalary = totalNetSalary +
                staff[i].netSalary();

    System.out.println(staff[i]);
}
EmployeeProcessor class (3)

totalBenefits = totalGrossSalary -
    totalNetSalary;
System.out.println("Total gross salary: " +
    totalGrossSalary);
System.out.println("Total benefits: " +
    totalBenefits);
System.out.println("Total net salary: " +
    totalNetSalary);
} // end of doTest method

public static void main(String[] args)
{ new EmployeeProcessor().doTest();
}
} // end of EmployeeProcessor class

display results
EmployeeProcessor output

Manager[name = Fred, gross = 800.0, net = 720.0]
Manager[name = Ellen, gross = 700.0, net = 630.0]
HourlyWorker[name = John, gross = 499.5, net = 474.525]
PartTimeWorker[name = Gord, gross = 446.25, net = 446.25]
CommissionWorker[name = Mary, gross = 925.0, net = 832.5]
Total gross salary: 3370.75
Total benefits: 267.4749999999999
Total net salary: 3103.275

Important idea:

(1) It is not necessary to know anything about the kinds of employees when writing the loop. The run-time system knows the type and will call the correct version in the polymorphic loop.

(2) If new kinds of employees are added to the hierarchy it is not necessary to make any changes to the polymorphic loop that calculates the salaries and benefits
The Object class

It is at the top of any hierarchy: an object of any class is of type `Object`. Some methods are:

```java
public String toString()
public boolean equals(Object obj)
public Object clone()
public Class<?> getClass()
```

Can define an array of type `Object[]` and store references to any kinds of objects in it:

```java
Object[] a = new Object[3];
a[0] = new BankAccount(123, "Fred", 3400);
a[1] = new Point(3,4);
a[2] = new Manager("Fred", 4000);
```
Overriding Object methods

The only method we have overridden so far is `toString`. If we hadn't done this we would get the following `Object` class version which isn't so meaningful.

Manager@310d42
Manager@5d87b2
HourlyWorker@77d134
PartTimeWorker@47e553
CommissionWorker@20c10f
Total gross salary: 3370.75
Total benefits: 267.47
Total net salary: 3103.275
Overriding the equals method

What does it mean to say two objects are equal?

equals method in Object class compares refs

```java
Point p = new Point(3,4);
Point q = p;
```

```java
p.equals(q) is true
```

```java
Point p = new Point(3,4);
Point q = new Point(3,5);
```

```java
p.equals(q) is not true
```

```java
Point p = new Point(3,4);
Point q = new Point(3,4);
```

```java
p.equals(q) is not true
```

In the last case we want a true result: even though the references are unequal, the objects are equal.
package chapter9.equals;
public class PointEqualsTester {
    public void doTest() {
        Point p = new Point(3,4);
        Point q = new Point(3,4);
        Point r = new Point(3,5);

        if (p.equals(q)) {
            System.out.println("p and q are equal");
        } else {
            System.out.println("p and q are not equal");
        }

        if (q.equals(r)) {
            System.out.println("q and r are equal");
        } else {
            System.out.println("q and r are not equal");
        }
    }

    public static void main(String[] args) {
        new PointEqualsTester().doTest();
    }
}
Recall the Point class

```java
package chapter9.equals;
public class Point {
    private double x;
    private double y;

    public Point() {...}
    public Point(double x, double y) {...}

    public double getX() {...}
    public double getY() {...}

    public boolean equals(Object obj) {...}

    public String toString() {...}
}
```
Override equals (1)

First attempt

```java
public boolean equals(Point p) {
    return (x == p.x && y == p.y);
}
```

This doesn't override the `Object` class `equals` method since it has a different prototype (`Point` instead of `Object` as an argument). The `Object` class method prototype is

```java
public boolean equals(Object obj)
```
Override equals (2)

Second attempt

```java
public boolean equals(Object obj)
{
    return (x == obj.x && y == obj.y);
}
```

This doesn't work since there are no x and y data fields for an object of type Object (object amnesia)
Override equals (3)

Third attempt

```java
public boolean equals(Object obj) {
    Point p = (Point) obj;
    return (x == p.x && y == p.y);
}
```

This works if you call the `equals` method with an object of type `Point`. If you call it with another type of object then the typecast throws a `ClassCastException`
Override equals (4)

Correct version

```java
public boolean equals(Object obj)
{
    if (obj instanceof Point)
    {
        Point p = (Point) obj;
        return (x == p.x && y == p.y);
    }
    return super.equals(obj);
}
```

Put this method into the `Point` class and the `PointEqualsTester` class will work properly.
Override equals (best)

A Java 5 version that uses `getClass` in `Object` class

```java
public boolean equals(Object obj)
{
    if (obj == null) return false;
    if (! this.getClass().equals(obj.getClass()))
        return false;
    Point p = (Point) obj;
    return (x == p.x && y == p.y);
}
```

Put this method into the `Point` class and the `PointEqualsTester` class will work properly.
Final classes

A final class cannot be extended to have subclasses. This means that the methods of a final class cannot be overridden.

```java
public final MyFinalClass
{
    // ...
}
```

Final classes are usually more efficient. Many of the standard classes such as `String` are declared final. Behaviour of final classes cannot be modified.
Interfaces

purely abstract class
independent of inheritance
provides polymorphic types and
methods
Multiple interfaces
What is an interface?

| ★ | It is a kind of purely abstract class |
| ★ | It contains only method prototypes |
| ★ | No implementations can be provided |
| ★ | A class can implement an interface |
| ★ | If several classes implement an interface then with respect to the interface the objects of these classes have the same type (polymorphic type) |
Syntax for an interface

```java
public interface MyInterface {
    // method prototypes go here
}
```

instead of using the class keyword we use interface
Syntax for an implementing class

```java
public class MyClass implements MyInterface {
    // data fields
    // constructors
    // methods not related to interface, if any
    // implementations of interface methods
}
```

`implements` is used instead of `extends`
Assume classes **MyClass1** and **MyClass2** both implement **MyInterface**

```java
MyInterface myObject1 = new MyClass1(...);
MyInterface myObject2 = new MyClass2(...);
```

Even though **myObject1** and **myObject2** are from different classes they are of the same type with respect to **MyInterface**: they are **MyInterface** objects.

This is like inheritance with the important difference that **MyClass1** and **MyClass2** do not need to be related in any other way. In particular they don't need to be related by inheritance
Extending an interface

It is possible to have interface inheritance hierarchies

Example

```java
public interface MySubInterface extends MyInterface {
    // new method prototypes go here
}
```
Extending and implementing

★ A class can only extend one other class. This is called single inheritance of classes. But it can implement several interfaces

★ Example

```java
public class MyClass extends MySuperClass
    implements MyInterface1, MyInterface2, ...
{
    // MyClass data fields and constructors
    // methods not related to interfaces
    // implementation of all interface methods
}
```
The Measurable interface

Suppose you are writing several classes that deal with the geometry of 2-dimensional objects such as circles, ellipses, rectangles, and so on.

Suppose each class has a method for calculating the area and perimeter of its objects. Then these classes can be related if they implement the Measurable interface:

```java
public interface Measurable {
    public double area();
    public double perimeter();
}
```

these are like abstract methods
The Scalable interface

Scale factor is same in both directions

```java
public interface Scalable {
    public void scale(double s);
}
```

Scale factor is different in each direction

```java
public interface Scalable2D extends Scalable {
    public void scale(double sx, double sy);
}
```
public class Circle implements Measurable
{
    private Point center;
    private double radius;
    public Circle(double xc, double yc, double r)
    {
        center = new Point(xc, yc);
        radius = r;
    }
    public Circle(Point p, double r)
    {
        center = new Point(p.getX(), p.getY());
        radius = r;
    }
    // other non-interface methods go here
    public double area()
    {
        return Math.PI * radius * radius;
    }
    public double perimeter()
    {
        return 2.0 * Math.PI * radius;
    }
}
Circle as Measurable object

Declare 2 Circle objects, one of type Circle and the other of type Measurable

Circle c1 = new Circle(0.0, 0.0, 1.0);
Measurable c2 = new Circle(0.0, 0.0, 1.0);

Without typecasting c2 can only access the area and perimeter interface methods in Circle class

double a1 = c1.area();
double r1 = c1.getRadius();
double a2 = c2.area();
double r2 = c2.getRadius();
double r2 = ((Circle) c2).getRadius();

this is illegal
this is legal
The Java2D graphics classes such as `Line2D`, `Rectangle2D` and `Ellipse2D` that draw geometrical shapes all implement an interface called `Shape`.

The fill and draw methods have a polymorphic `Shape` argument:

```java
public void draw(Shape s)
public void fill(Shape s)
```
Shape polymorphism

Since `Line2D` and `Rectangle2D` are of type `Shape` we can write either

```java
Line2D.Double line = new Line2D.Double(...);
Rectangle2D.Double rect = new Rectangle2D.Double(..);
```

or

```java
Shape line = new Line2D.Double(...);
Shape rect = new Rectangle2D.Double(...);
```
We can store any kind of graphics object in a `Shape` array (array of references to `Shape` objects) and process them in a polymorphic loop.

```java
Shape[] shape = new Shape[5];
shape[0] = new Line2D.Double(...);
shape[1] = new Rectangle2D.Double(...);
shape[2] = new RoundRectangle2D.Double(...);
shape[3] = new Ellipse2D.Double(...);
shape[4] = new Ellipse2D.Double(...);

for (int k = 0; k < shape.length; k++)
{
    g2D.draw(shape[k])
}
```
package chapter9.shapetest;
import custom_classes.GraphicsFrame;  // Chapter 5
import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

public class ShapeTester extends JPanel
{
    Shape[] shape = new Shape[5];
    public ShapeTester()
    {
        shape[0] = ...
        shape[1] = ...
        shape[2] = ...
        shape[3] = ...
        shape[4] = ...
    }
}
public void paintComponent(Graphics g) {
    super.paintComponent(g);
    Graphics2D g2D = (Graphics2D) g;

    double xMax = getWidth() - 1;
    double yMax = getHeight() - 1;
    AffineTransform at = new AffineTransform();
    at.translate(xMax / 2, yMax / 2);
    at.scale(xMax / 200, yMax / 150);
    at.translate(-100,-75);
    g2D.transform(at);
// polymorphic loop to draw shapes

for (int k = 0; k < shape.length; k++)
{
    g2D.setPaint(Color.pink);
    g2D.fill(shape[k]);
    g2D.setPaint(Color.black);
    g2D.draw(shape[k]);
}

} // end of PaintComponent method
public void draw()
{
    new GraphicsFrame("Some shapes",
                      this, 201, 151);
}

public static void main(String[] args)
{
    new ShapeTester().draw();
}
} // end of ShapeTester class
Multiple Interfaces

```java
public interface Measurable {
    public double area();
    public double perimeter();
}

public interface Translatable {
    public void translate(double dx, double dy);
}

public interface Scalable {
    public void scale (double s);
}
```
package chapter9.multiple_interfaces;

public class Circle implements Measurable, Translatable, Scalable {

private double x, y, radius;
...

public double area() {
    return Math.PI * radius * radius;
}

public double perimeter() {
    return 2.0 * Math.PI * radius;
}

public void translate(double dx, double dy) {
    x = x + dx; y = y + dy;
}

public void scale(double s) {
    radius = radius * s;
}
}
package chapter9.multiple_interfaces;

public class Rectangle implements Measurable,
    Translatable, Scalable
{
    private double x, y, width, height;
    ...
    public double area()
    { return width * height; }
    public double perimeter()
    { return 2.0 *(width + height); }
    public void translate(double dx, double dy)
    { x = x + dx; y = y + dy; }
    public void scale(double s)
    { width = width * s; height = height * s; }
}
package chapter9.interfaces;
public class MeasurableTester
{
    private Measurable[] a = new Measurable[3];

    public void test()
    {
        a[0] = new Circle(0,0,1);
        a[1] = new Circle(1,1,2);
        a[2] = new Rectangle(5,5,20,10);
    }
}
double areaSum = 0.0; double perimeterSum = 0.0;
for (int k = 0; k < a.length; k++) {
    areaSum = areaSum + a[k].area();
    perimeterSum = perimeterSum + a[k].perimeter();
    System.out.println(a[k]);
    System.out.println("Perimeter = ", a[k].perimeter() + ", Area = ", a[k].area());
}
System.out.println("Total area is ", areaSum);
System.out.println("Total perimeter is ", perimeterSum);
} // end of test method
public static void main(String[] args) {
    new MeasurableTester().test();
} // end of MeasurableTester class
package chapter9.multiple_interfaces;
public class MultipleInterfaceTester
{
  private Object[] a = new Object[3];
  public void test()
  {
    a[0] = new Circle(0,0,1);
    a[1] = new Circle(1,1,2);
    a[2] = new Rectangle(5,5,20,10);
    for (int k = 0; k < a.length; k++)
    {
      ((Translatable) a[k]).translate(1,1);
      ((Scalable) a[k]).scale(2);
      System.out.println(a[k]);
    }
  }
  public static void main(String[] args)
  {
    new MultipleInterfaceTester().test();
  }
}
The **Shape** interface declares methods that are needed to draw or fill a 2-dimensional shape.

Examples of **Shape** objects are:

- `Line2D.Double`
- `Rectangle2D.Double`
- `Ellipse2D.Double`
- `GeneralPath`
public interface Shape
{
    public boolean contains(Point2D p);
    public boolean contains(Rectangle2D r);
    public boolean contains(double x, double y);
    public boolean contains(double x, double y, double w, double h);
    public Rectangle getBounds();
    public Rectangle2D getBounds2D();
    public PathIterator getPathIterator(AffineTransform at);
    public PathIterator getPathIterator(AffineTransform at, double flatness);
    public boolean intersects(Rectangle2D r);
    public boolean intersects(double x, double y, double w, double h);
}

complex interface with 10 methods
It would be complicated to figure out how to implement these 10 methods.

Fortunately it is not necessary since `GeneralPath` implements the `Shape` interface.

We can write adapter classes that use `GeneralPath` to implement our own graphics objects that can be used as arguments to the `draw` and `fill` methods.
package chapter9.shapes;
import java.awt.*;
import java.awt.geom.*;

public class ShapeAdapter implements Shape {
    /** The path used to define the Shape */
    protected GeneralPath path;

    /** Construct an empty path */
    public ShapeAdapter()
    {
        path = new GeneralPath();
    }

    // Now use path to implement the Shape interface
public boolean contains(Point2D p)
{ return path.contains(p); }
public boolean contains(Rectangle2D r)
{ return path.contains(r); }
public boolean contains(double x, double y)
{ return path.contains(x,y); }
public boolean contains(double x, double y,
           double w, double h)
{ return path.contains(x,y,w,h); }
public java.awt.Rectangle getBounds()
{ return path.getBounds(); }
public Rectangle2D getBounds2D()
{ return path.getBounds2D(); }
public PathIterator getPathIterator(AffineTransform at)  
{ return path.getPathIterator(at); } 
public PathIterator getPathIterator(AffineTransform at, double flatness)  
{ return path.getPathIterator(at, flatness); } 
public boolean intersects(Rectangle2D r)  
{ return path.intersects(r); } 
public boolean intersects(double x, double y, double w, double h)  
{ return path.intersects(x,y,w,h); } 

Note how the **Shape** interface method implementations just call the corresponding ones that are available with the **GeneralPath** object called **path**
public class MyGraphics extends ShapeAdapter
{
    // data fields, if any
    // constructors using the inherited path object
    // to define the path of our shape
    // methods, if any
}

Now to draw a shape we can use statements such as

Shape s = new MyGraphicsShape(...);
...
g2D.draw(s);
g2D.fill(s);
Implementing Shape directly

If the `MyGraphicsClass` already extends some class then we cannot use `ShapeAdapter` since we can only extend one class using inheritance in Java. Therefore we implement shape directly as in `ShapeAdapter`.

```java
public class MyGraphics extends AnotherClass
    implements Shape
{
    GeneralPath path;

    // other data fields, if any
    // constructors and methods not in Shape interface
    // implementation of the 10 shape methods here
    // using path (as we did in ShapeAdapter)
}
```
package chapter9.shapes;
import java.awt.geom.*;

public class Triangle2D extends ShapeAdapter {
    private Point2D.Double v1, v2, v3;

    public Triangle2D()
    {
        this(new Point2D.Double(0,0),
             new Point2D.Double(1,0),
             new Point2D.Double(0.5,1));
    }
}
public Triangle2D(double x1, double y1, double x2, double y2, double x3, double y3)
{
    this(new Point2D.Double(x1,y1),
         new Point2D.Double(x2,y2),
         new Point2D.Double(x3,y3));
}
public Triangle2D(Point2D.Double p1, 
   Point2D.Double p2, Point2D.Double p3) 
{
    v1 = (Point2D.Double) p1.clone();
    v2 = (Point2D.Double) p2.clone();
    v3 = (Point2D.Double) p3.clone();

    // psthi is inherited from ShapeAdapter
    path.moveTo((float) v1.x, (float) v1.y);
    path.lineTo((float) v2.x, (float) v2.y);
    path.lineTo((float) v3.x, (float) v3.y);
    path.closePath();
}
Using the Triangle2D class

We can now use the Triangle2D class in the same way as we did classes such as Rectangle2D.

The only difference is that we did not create both Float and Double versions:

```java
Triangle2D nose =
    new Triangle2D(100,80, 90,110, 110,110);
...

g2D.setPaint(Color.green);
g2D.fill(nose);
```
Other examples

<table>
<thead>
<tr>
<th>RandomTriangles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Turtle2D</td>
</tr>
<tr>
<td>PentagonSpinner</td>
</tr>
<tr>
<td>RecursiveTreeMaker</td>
</tr>
</tbody>
</table>
public class Test
{
    System.out.println("Hello");
}

public class Test
{
    System.out.println("Hello");
}
Template 2

left justified text

- sub item
- sub item
- sub item
- sub item