继承是面向对象程序设计的一个基本概念。继承的基本思想是，可以基于已有的类创建新的类。继承已存在的类就是复用（继承）这些类的方法，而且可以增加一些新的方法和字段，使新类能够适应新的情况。

1 定义子类

在定义新类时，使用关键字`extends`表示继承。例如：

```
public class Manager extends Employee
{
    private double bonus;

    public void setBonus(double bonus)
    {
        this.bonus = bonus;
    }
}
```

关键词`extends`表明正在构造的新类派生于一个已经存在的类。这个已存在的类称为**超类**、**基类**或**父类**，新类称为**子类**或**派生类**。在上面的例子中，`Manager`类派生于`Employee`类，`Employee`类是超类，`Manager`类是子类。

在`Manager`类中，增加了一个用于存储奖金信息的字段，以及一个用于设置这个字段的新方法。这里定义的方法和字段并没有什么特别之处，如果有一个`Manager`对象，就可以使用`setBonus`方法。但是，由于`setBonus`方法不是在`Employee`类中定义的，所以`Employee`类的对象不能使用它。

尽管在`Manager`类中没有显式地定义`getName`和`getHireDay`等方法，但是可以对`Manager`对象使用这些方法，因为`Manager`类自动地继承了超类`Employee`中的这些方法。

类似地，从超类中还继承了`name`、`salary`、`hireDay`这3个字段。这样一来，每个`Manager`对象就包含了4个字段：`name`、`salary`、`hireDay`、`bonus`。

通过扩展超类定义子类的时候，只需要指出子类与超类的不同之处。因此在设计类的时候，应该将最一般的方法放在超类中，而将更特殊的方法放在子类中。

2 覆盖方法

超类中的有些方法对子类并不一定适用，为此需要提供一个新的方法来覆盖超类中的这个方法。只需要在子类的定义中重写超类中的方法，就可以完成覆盖。例如：

```
public class Manager extends Employee
{
    ...
    public double getSalary()
    {
        ...
    }
    ...
}
```

`Employee`类中有`getSalary`方法，在`Manager`类中重写这个方法就可以覆盖它。

如果超类中的方法被覆盖，又想在子类的定义中调用这个方法，可以使用关键字`super`。例如`Manager`类中的`getSalary`方法可以定义如下：

```
public double getSalary()
{
    double baseSalary = super.getSalary(); // 调用 Employee 类中的 getSalary 方法
    return baseSalary + bonus;
}
```

`super`关键字也可以用于访问超类的实例字段，用法与调用超类方法类似。

3 子类构造器

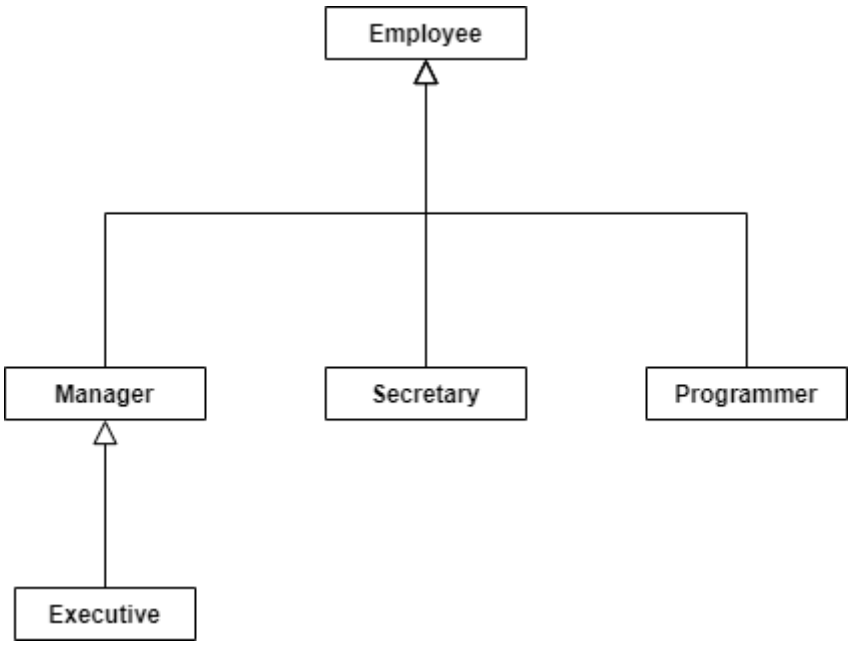
由于子类的构造器不能访问超类的私有字段，所以必须通过超类的构造器来初始化这些私有字段。可以利用`super`语法调用这个构造器。使用`super`调用构造器的语句必须是子类构造器的第一条语句。例如`Manager`类的构造器可以定义如下：

```
public Manager(String name, double salary, int year, int month, int day)
{
    super(name, salary, year, month, day);
    bonus = 0;
}
```

如果子类的构造器没有显式地调用超类的构造器，将自动调用超类的无参数构造器。如果超类没有无参数构造器，并且在子类中又没有显式地调用超类的其他构造器，Java编译器就会报告一个错误。

4 继承层次

继承并不仅限于一个层次。由一个公共超类派生出来的所有类的集合称为**继承层次**。在继承层次中，从某个特定的类到其祖先的路径称为该类的**继承链**。例如：



`Employee`作为公共超类，派生出`Manager`、`Secretary`、`Programmer`三个子类，`Manager`类又派生出`Executive`类。`Manager`、`Secretary`、`Programmer`、`Executive`这四个类的祖先都是`Employee`类，称为`Employee`类的继承层次。从`Executive`类到祖先的路径是`Executive -> Manager -> Employee`，这是`Executive`类的继承链。

5 多态

里氏替换原则（Liskov Substitution Principle，LSP）：程序中出现超类对象的任何地方都可以使用子类对象替换。

在Java中，对象变量是多态的。一个类的变量既可以引用本类对象，也可以引用它的任何一个子类的对象。例如上面的`Employee`类变量既可以引用`Employee`类对象，也可以引用`Manager`、`Secretary`、`Programmer`、`Executive`类的对象。

当子类覆盖了超类的方法时，虚拟机可以在运行时根据对象变量引用的对象类型正确地调用相应类的方法。

下面用一个例子展示多态性的应用。

```
/* ManagerTest.java */
package inheritance;

public class ManagerTest
{
    public static void main(String[] args)
    {
        Manager boss = new Manager("Carl Cracker", 80000, 1987, 12, 15);
        boss.setBonus(5000);

        Employee[] staff = new Employee[3];
        staff[0] = boss;
        staff[1] = new Employee("Harry", 50000, 1989, 10, 1);
        staff[2] = new Employee("Tommy", 40000, 1990, 3, 15);

        for(Employee e : staff)
        {
            System.out.println("name=" + e.getName() + ",salary=" +
e.getSalary());
```

```
        }
    }
}
```

```
/* Employee.java */
package inheritance;

import java.time.*;

public class Employee
{
    private String name;
    private double salary;
    private LocalDate hireDay;

    public Employee(String name, double salary, int year, int month, int day)
    {
        this.name = name;
        this.salary = salary;
        hireDay = LocalDate.of(year, month, day);
    }

    public String getName()
    {
        return name;
    }

    public double getSalary()
    {
        return salary;
    }

    public LocalDate getHireDay()
    {
        return hireDay;
    }

    public void raiseSalary(double byPercent)
    {
        double raise = salary * byPercent / 100;
        salary += raise;
    }
}
```

```
/* Manager.java */
package inheritance;

public class Manager extends Employee
{
    private double bonus;

    public Manager(String name, double salary, int year, int month, int day)
    {
        super(name, salary, year, month, day);
        bonus = 0;
    }
```

```
    public double getSalary()
    {
        double baseSalary = super.getSalary();
        return baseSalary + bonus;
    }

    public void setBonus(double b)
    {
        bonus = b;
    }
}
```

在`Employee`数组`staff`中，`staff[0]`引用了一个`Manager`对象，`staff[1]`和`staff[2]`各引用一个`Employee`对象。在输出对象信息时，`Employee`类和`Manager`类都有`getSalary`方法，对`staff[0]`引用的`Manager`对象，使用`Manager`类的`getSalary`方法；对`staff[1]`和`staff[2]`引用的`Employee`对象，使用`Employee`类的`getSalary`方法。

在这个例子中，`staff[0]`与`boss`引用同一个`Manager`对象，但编译器只将`staff[0]`看做`Employee`对象。这意味着，`boss`可以调用`setBonus`方法，而`staff[0]`不可以，因为`setBonus`不是`Employee`类的方法。

对象变量不能引用超类对象。因为对象变量是子类，它可以使用子类的方法，而这些方法可能在超类中不存在。例如，不能用`Manager`类变量引用`Employee`对象，因为变量可以使用`setBonus`方法，而`Employee`类中没有这个方法，如果调用`setBonus`方法就会发生运行时错误。

> 警告：在Java中，子类引用的数组可以转换成超类引用的数组，而不需要使用强制类型转换。例如：

```
Manager[] managers = new Manager[10];
Employee[] staff = managers;
```

> 这样的操作是合法的，但是会发生一些不好的事情。例如：

```
staff[0] = new Employee("Harry", 50000, 1989, 10, 1);
```

> 这条语句是合法的，此时`staff[0]`和`managers[0]`引用同一个`Employee`对象。`Manager`类变量`manager[0]`引用了超类`Employee`的对象，这是我们不希望看到的。
>
> 为了确保不发生这样的破坏，所有数组都要牢记创建时的元素类型，并负责监督仅将类型兼容的引用存储到数组中。

6 final类和方法

如果希望阻止某个类派生子类，就可以使用`final`关键字。例如：

```
public final class Executive extends Manager
{
    ...
}
```

类中的某个特定方法也可以被声明为`final`，这样子类就不能覆盖这个方法。`final`类中的所有方法自动成为`final`方法。

字段也可以声明为`final`，构造对象之后不能改变`final`字段的值。`final`类中的字段不会自动成为`final`字段。

7 理解方法调用

假设要调用`x.f(args)`，隐式参数`x`声明为类C的一个对象。下面是调用过程的详细描述：

1. 编译器查看对象的声明类型和方法名，一一列举C类中所有名为`f`的方法和其超类中所有名为`f`而且可访问的方法。
2. 编译器确定方法调用中提供的参数类型，如果在所有名为`f`的方法中存在一个与所提供参数类型完全匹配的方法，就选择这个方法。这个过程称为**重载解析**。如果编译器没有找到与参数类型匹配的方法，或者发现经过类型转换后有多个方法与之匹配，编译器就会报告一个错误。

> 前面曾经说过，方法的名字和参数列表称为方法的签名。如果在子类中定义了一个与超类签名相同的方法，那么子类中的这个方法就会覆盖超类中这个相同签名的方法。
> 返回类型不是签名的一部分，不过在覆盖一个方法时，需要保证返回类型的兼容性。
> 允许子类将覆盖方法的返回类型改为原返回类型的子类型。例如，假设`Employee`类有以下方法：

```
public Employee getBuddy()
```

> 在子类`Manager`中，可以如下覆盖这个方法：

```
public Manager getBuddy()
```

> 称这两个`getBuddy`方法有**可协变的**返回类型。

3. 如果是`private`方法、`static`方法、`final`方法或者构造器，那么编译器可以准确地知道应该调用哪个方法，这称为静态绑定。与此相对，如果要调用的方法依赖于隐式参数的实际类型，那么必须在运行时使用动态绑定。在上面的例子中，编译器会利用动态绑定生成一个调用`f(String)`的指令。
4. 程序运行并且采用动态绑定调用方法时，虚拟机必须调用与`x`所引用对象的实际类型对应的那个方法。假如`x`的实际类型是D，它是C类的子类，如果D类定义了方法`f(String)`，就会调用这个方法；否则，将在D类的超类中寻找`f(String)`，以此类推。

每次调用方法都要完成这个搜索，时间开销相当大。因此，虚拟机预先为每个类计算了一个方法表，其中列出了所有方法的签名和要调用的实际方法。这样一来，在真正调用方法的时候，虚拟机仅查找这个表就可以了。

下面分析前面多态应用的例子中调用`e.getSalary()`的过程。（注：这个调用在`for each`循环的输出语句中）

`e`声明为`Employee`类型，`Employee`类只有一个名叫`getSalary`的方法，因此这里不存在重载解析的问题。由于`getSalary`不是`private`方法、`static`方法或`final`方法，所以将采用动态绑定。虚拟机为`Employee`和`Manager`类生成方法表：

```
Employee:
    getName() -> Employee.getName()
    getSalary() -> Employee.getSalary()
    getHireDay() -> Employee.getHireDay()
    raiseSalary(double) -> Employee.raiseSalary(double)
Manager:
    getName() -> Employee.getName()
    getSalary() -> Manager.getSalary()
    getHireDay() -> Employee.getHireDay()
    raiseSalary(double) -> Employee.raiseSalary(double)
    setBonus(double) -> Manager.setBonus(double)
```

在运行时，调用 `e.getSalary()` 的解析过程为：

1. 首先，虚拟机获取 `e` 的实际类型的方法表。这可能是 `Employee` 或 `Manager` 的方法表。
2. 接下来，虚拟机查找定义了 `getSalary()` 签名的类。此时，虚拟机已经知道应该调用哪个方法。
3. 最后，虚拟机调用这个方法。

> 警告：在覆盖一个方法的时候，子类方法不能低于超类方法的可见性。
>
> 例如，如果超类方法是 `public`，子类方法必须也要声明为 `public`。

8 强制类型转换

有时可能需要将某个类的对象引用转换成另外一个类的对象引用。对象引用的强制类型转换与数值表达式的强制类型转换类似，只需要用一对圆括号将目标类名括起来，并放置在需要转换的对象引用之前即可。例如：

```
Manager boss = (Manager) staff[0];
```

一个对象变量可以引用本类对象或子类对象，但不能引用超类对象。如果将一个子类的引用赋值给超类变量，符合里氏替换原则，不需要类型转换，在此不做讨论；如果将超类的引用赋值给子类变量，需要类型转换，将超类变量强制转换成子类变量，但这种转换未必合法，下面分两种情况讨论。

第一种情况，当待转换的对象变量引用子类对象时，例如：

```
Employee staff = new Manager("Harry", 50000, 1989, 10, 1);
```

`staff` 变量是 `Employee` 类型，通过 `staff` 变量只能调用 `Employee` 类中的方法，不能使用 `Manager` 类中派生出的新方法和实例字段。为了能够访问新添加的实例字段和方法，需要将这个变量复原成 `Manager` 类型，为此需要强制类型转换：

```
Manager manager = (Manager) staff; // 合法的类型转换
manager.setBonus(5000);
```

第二种情况，当待转换的对象变量引用本类对象时，例如：

```
Employee employee = new Employee("Harry", 50000, 1989, 10, 1);
```

如果要把这个变量强制转换为 `Manager` 类型：

```
Manager boss = (Manager) employee; // 运行时错误
```

这个转换是不合法的，会在运行时产生`ClassCastException`异常。因为这种转换意味着`boss`变量要引用一个超类对象，这是不允许的。

对这两种情况总结如下：将超类变量强制转换成子类变量时，如果转换后变量引用本类或子类对象，转换就是合法的；如果转换后出现变量引用超类对象的情况，就是不合法的，但编译不报错，会在运行时抛出异常。

在进行强制类型转换之前，要先查看是否能够成功转换，为此只需要使用`instanceof`操作符。例如：

```
if (staff instanceof Manager)
{
    boss = (Manager) staff;
}
```

`instanceof`操作符前面是待转换的变量，后面是目标类型。如果转换合法，结果为`true`，执行`if`里面的语句；如果转换不合法，结果为`false`，跳过类型转换语句。如果对象变量为`null`，结果也是`false`。

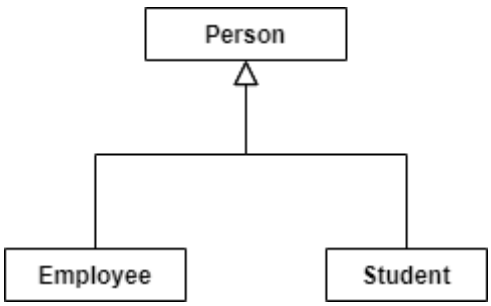
最后，只能在继承层次内进行强制类型转换。例如，`String`类不是`Employee`类的子类，因此下面的转换是不合法的：

```
String c = (String) staff; // 编译错误
```

通过强制类型转换来转换对象的类型通常不是一种好的做法。大多数情况下，当超类和子类有同样的方法时，通过多态性的动态绑定机制可以自动找到正确的方法，不需要强制类型转换。只有在使用子类特有的方法时才需要强制类型转换。一般情况下，最好尽量少用强制类型转换。

9 抽象类

在继承层次中，祖先类更具有一般性，也更加抽象。在某些情况下，祖先类中的方法无法确定具体操作，例如：



增加一个`getDescription`方法，返回对一个人的简短描述，例如：

> an Employee with a salary of $50,000.00
>
> a student majoring in computer science

对一个人的描述依赖于这个人的身份，但在`Person`类中还不知道人的身份，无法给出方法的具体操作。这时可以使用`abstract`关键字，这样就不用实现这个方法了：

```
public abstract String getDescription(); // 不需要给出具体实现
```

为了提高清晰度，包含一个或多个抽象方法的类必须被声明为抽象类。例如：

```
public abstract class Person
{
    ...
    public abstract String getDescription();
}
```

除了抽象方法之外，抽象类还可以包含字段和具体方法（注：具体方法与抽象方法相对，是指给出实现的方法）。即使不含抽象方法，也可以将一个类声明为抽象类。

抽象方法充当占位方法的角色，它们在子类中具体实现。扩展抽象类可以有两种选择。一种是在子类中保持部分或所有抽象方法仍未定义，这样子类仍然是抽象类；另一种做法是定义全部方法，这样子类就不是抽象类了。

抽象类不能实例化，不能创建这个类的对象。但是可以定义抽象类的对象变量，这个变量只能引用非抽象子类的对象。

下面用一个例子展示抽象类的使用。

```
/* PersonTest.java */
package abstractClasses;

public class PersonTest
{
    public static void main(String[] args)
    {
        Person people = new Person[2];
        people[1] = new Employee("Harry", 50000, 1989, 10, 1);
        people[2] = new Student("Maria", "computer science");

        for(Person p : people)
        {
            System.out.println(p.getName() + "," + p.getDescription());
        }
    }
}
```

```
/* Person.java */
package abstractClasses;

public abstract class Person
{
    public abstract String getDescription();
    private String name;

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

    public String getName()
    {
        return name;
    }
}
```

```java
/* Employee.java */
package abstractClasses;

import java.time.*;

public class Employee extends Person
{
    private double salary;
    private LocalDate hireDay;

    public Employee(String name, double salary, int year, int month, int day)
    {
        super(name);
        this.salary = salary;
        hireDay = LocalDate.of(year, month, day);
    }

    public double getSalary
    {
        return salary;
    }

    public LocalDate getHireDay()
    {
        return hireDay;
    }

    public String getDescription()
    {
        return String.format("an employee with a salary of $%.2f", salary);
    }

    public void raiseSalary(double byPercent)
    {
        double raise = salary * byPercent / 100;
        salary += raise;
    }
}
```

```java
/* Student.java */
package abstractClasses;

public class Student extends Person
{
    private String major;

    public Student(String name, String major)
    {
        super(name);
        this.major = major;
    }

    public String getDescription()
    {
        return "a student majoring in " + major;
    }
}
```

10 受保护访问

如果希望超类中的某个方法只允许子类访问，或者希望允许子类的方法访问超类的某个字段，可以将这些方法或字段声明为受保护（`protected`）。例如，如果将超类`Employee`中的`hireDay`字段声明为`protected`，子类`Manager`中的方法就可以直接访问这个字段。

在Java中，保护字段只能由同一个包中的类访问。有了这个限制，就能避免滥用保护机制，不能通过派生子类来访问受保护的字段。

在实际应用中，要谨慎使用受保护字段。假设你的类要提供给其他程序员使用，而你在设计这个类时设置了一些受保护字段，其他程序员可能会由这个类派生出新类，并访问受保护字段。这种情况下，如果你想修改你的类的实现，就势必会影响那些程序员，这违背了数据封装的精神。

受保护的方法更具有实际意义。如果需要限制某个方法的使用，就可以将它声明为`protected`，这表明子类得到了信任，可以正确地使用这个方法，而其他类不行。

下面对Java中的访问控制做个总结：

1. 仅对本类可见：`private`
2. 对外部完全可见：`public`
3. 对本包和所有子类可见：`protected`
4. 对本包可见：默认，不需要修饰符

11 继承的设计技巧

1. 将公共操作和字段放在超类中。
2. 不要使用受保护字段。
3. 使用继承实现“is-a”关系。
4. 除非所有继承的方法都有意义，否则不要使用继承。
5. 在覆盖方法时，不要改变预期的行为。
6. 使用多态，而不要使用类型信息。