第7章　7.3節
オブジェクト指向の特徴

OOP特征：封装[カプセル化]（Encapsulation）

- 封装[カプセル化]可确保信息的隐蔽性
  - 对象的内部状态不需要被外部访问的可以对外部不可见
  - 只对外公开必要的属性和方法
  - 增强数据安全性

- 访问修饰符[アクセス修飾子]的作用

获取和设置[ゲッターとセッター]（getter & setter）

- private变量只能在同一类中访问（外部类无权访问）。但是，如果我们提供公共获取和设置方法，则可以访问它们。
- get方法返回变量值，set方法设置该值。
- 两者的语法是，它们以get或set开头，后跟变量名，首字母大写：

```
public class Person {
  private String name; // private = restricted access

  // Getter
  public String getName() {
    return name;
  }

  // Setter
  public void setName(String newName) {
    this.name = newName;
  }
}
```

获取和设置[ゲッターとセッター]（getter & setter）

- get方法返回变量的值name。
- set方法采用参数（newName）并将其分配给 name变量。 this关键字指当前对象。
- 但是，由于name变量声明为private，因此我们不能从此类外部访问它：

```
public class MyClass {
  public static void main(String[] args) {
    Person myObj = new Person();
    myObj.setName("John"); // Set the value of the name variable to "John"
    System.out.println(myObj.getName());
  }
}

// Outputs "John"
```

为什么要封装[カプセル化]?

- 更好地控制类的属性和方法
- 可以将类属性设置为只读[読み取り専用]（如果仅使用get方法）或只写[書き込み専用]（如果仅使用set方法）
- 灵活性：程序员可以更改代码的一部分而不影响其他部分
- 增强数据安全性

Java包和API

- Java中的软件包用于对相关类进行分组。可以将其视为文件目录中的文件夹。我们使用软件包来避免名称冲突，并编写更好的可维护代码。软件包分为两类：
  - 内置软件包[組み込みパッケージ]（来自Java API的软件包）
  - 用户定义的软件包（自己创建的软件包）

内置软件包[組み込みパッケージ]（Built-in Packages）

- Java API是Java开发环境中包含的，可免费使用的预编写类的库。
- 该库包含用于管理输入，数据库编程等的组件。完整列表可以在Oracle网站上找到：https：//docs.oracle.com/javase/8/docs/api/。
- 该库分为package和class。这意味着你既可以导入单个类（以及其方法和属性），也可以导入包含所有类的整个包。

- 要导入库中的类或包，需要使用import 关键字：

```
import package.name.Class;   // Import a single class
import package.name.*;   // Import the whole package
```

导入class

- 如果导入要使用的类，例如用于获取用户输入的Scanner类，请按如下操作导入：

```
import java.util.Scanner;

class MyClass {
  public static void main(String[] args) {
    Scanner myObj = new Scanner(System.in);
    System.out.println("Enter username");

    String userName = myObj.nextLine();
    System.out.println("Username is: " + userName);
  }
}
```

- 尝试代码： ImportClass.java

导入Package

- 有很多软件包可供选择。在前面的示例中，我们使用了java.util包中的Scanner类。该软件包还包含日期和时间工具，随机数生成器和其他实用程序类。
- 要导入整个程序包，请在句子后加上星号（*）。下面的示例将导入java.util包中的所有类：

```
import java.util.*;
```

用户定义的程序包

- 要创建自己的软件包，你需要知道Java使用文件系统目录来存储它们。就像你计算机上的文件夹一样：

- 要创建一个包，请使用package关键字：

```
package mypack;
class MyPackageClass {
  public static void main(String[] args) {
    System.out.println("This is my package!");
  }
}
```

创建包

- 尝试代码： MyPackageClass.java
- 对此文件进行编译：

```
C:\Users\Your Name>javac MyPackageClass.java
```

- 然后编译该包：

```
C:\Users\Your Name>javac -d . MyPackageClass.java
```

- 这将强制编译器创建“mypack”包。
- -d关键字指定在哪里保存类文件的目标。可以使用任何目录名称，例如c：/ user（windows），或者，如果要将软件包保留在同一目录中，则可以使用点“.”，如上例所示。
- 注意：软件包名称应使用小写字母，以避免与类名称冲突。

创建包

- 当我们在前面的示例中编译软件包时，创建了一个新文件夹，称为“mypack”。
- 要运行MyPackageClass.java文件，请编写以下内容：

```
C:\Users\Your Name>java mypack.MyPackageClass
```

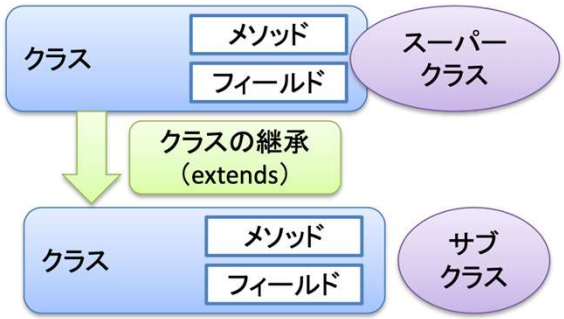
- 自己动手试一下！

Java 继承

- 在Java中，可以将属性[フィールド]和方法[メソッド]从一个类继承到另一个类。我们将“继承概念”分为两类：
  - 子类[サブクラス]（subclass）-继承另一个类的类
  - 超类/父类[スーパークラス]（superclass）-被继承的类
- 要继承类，请使用extends 关键字。

```
class NemedRectangle extends Rectangle {
   String name;
   NamedRectangle() {
      name = “No Name”;
   }
   NamedRectangle(String name) {
      this.name = name;
   }
}

 public static void main(String[] args) {
      NamedRectangle nr = new NamedRectangle();
      nr.setSize(123, 45);
      System.out.println(nr.getArea());
   }
```



Java 继承

- 子类[サブクラス]里即使不记述超类的属性和方法，也可以调用它们
- 尝试代码： Vehicle.java
- 对代码进行一下改动：
  - 将brand从protected改成private的变量，会怎么样?
  - 将Vehicle改成final类，会怎么样?

- 为什么以及何时使用“继承”?

-这对于代码可重（chóng）用性很有用：在创建新类时重用现有类的属性和方法。

重写[オーバライド]（Override）

- 重写[オーバライド]：超类里定义的非静态方法，在子类[サブクラス]里面重新定义，也可以叫做覆盖。
- 重写[オーバライド]方法的规则：
  - 参数，返回值类型，方法名必须一致
  - 在继承的过程中，如果产生了方法的重写，那么重写的方法的访问修饰符一定不小于被重写方法的访问修饰符

```
class A {
    method(Object o) {
        ...
    }
}
```

```
class B extends A {
    method(Object o) {
        ...
    }
}
```

```
A a = new A();
a.method();

B b = new B();
b.method();
```

継承されたClassA の method() ではなく，
ClassB で定義された
新しい method() が呼ばれる

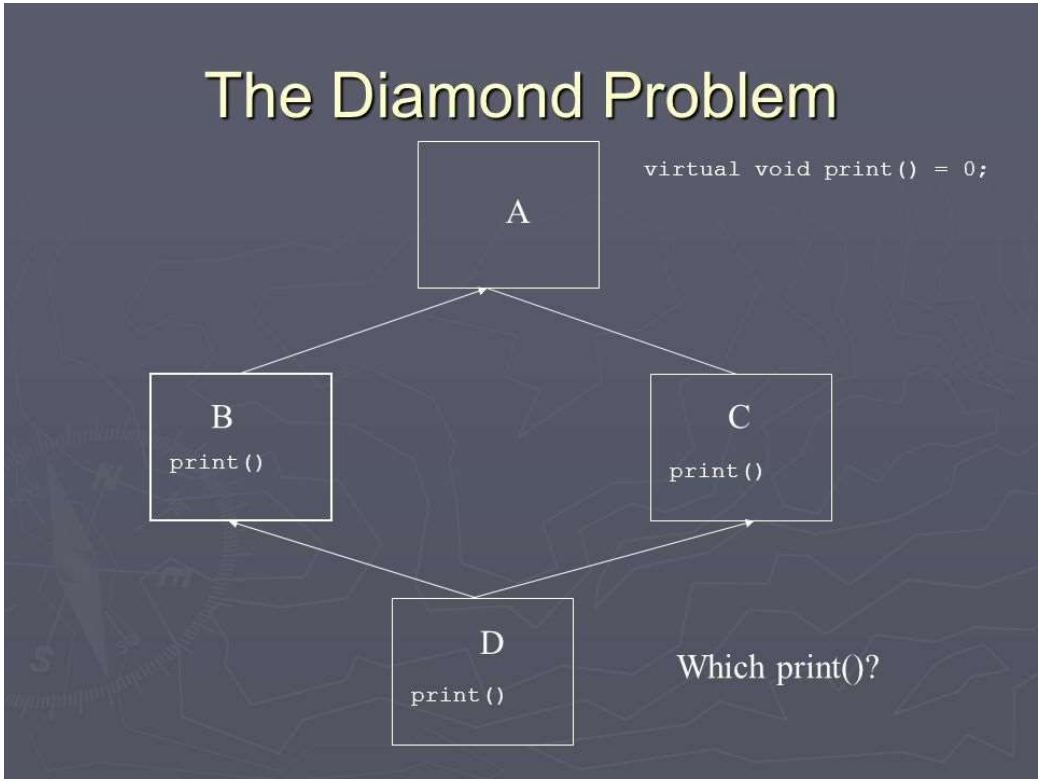
重载[オーバーロード]（Overload）

- 方法名一样，返回值类型和参数不一样
- Java的重载[オーバーロード]不可以只是返回值类型不一样，思考这是为什么

```
class  Triangle {
    void setSides(double x) { ... }
    void setSides(double x, double y) { ... }
    void setSides(double x, double y, double z) { ... }
}
```

多重继承

- 一个子类[サブクラス]继承多个父类
- 向上追踪会变得很困难
- Java不支持多重继承
- Java用接口[インターフェース]代替多重继承
- 参考：钻石继承问题



Java 多态性[ポリモーフィズム]（Polymorphism）

- 多态性[ポリモーフィズム]的意思是“许多形式”，当我们有许多通过继承相互关联的类时，就会引发多态性[ポリモーフィズム]。
- 例如，考虑一个名为的超类Animal，该类具有animalSound()方法。Animal的子类[サブクラス]可以是Pig，Cat，Dog，Bird。并且它们也具有自己的动物声音实现方式：

```
class Animal {
  public void animalSound() {
    System.out.println("The animal makes a sound");
  }
}

class Pig extends Animal {
  public void animalSound() {
    System.out.println("The pig says: wee wee");
  }
}

class Dog extends Animal {
  public void animalSound() {
    System.out.println("The dog says: bow wow");
  }
}
```

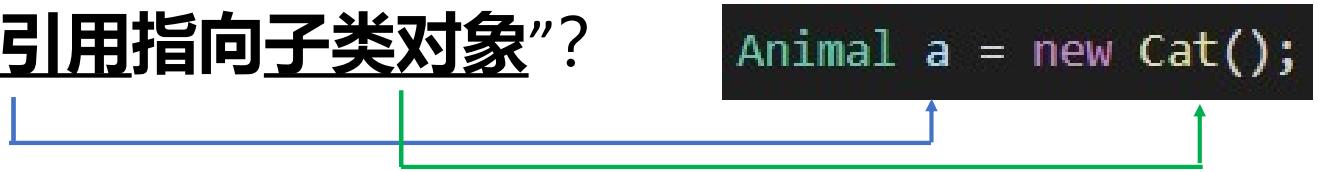
Java 多态性[ポリモーフィズム]（Polymorphism）

- 尝试代码：Animal.java

- 为什么以及何时使用“继承”和“多态”？
 -这对于代码可重用性很有用：在创建新类时重用现有类的属性和方法。

Java 多态性[ポリモーフィズム]（Polymorphism）

- 如何理解“**父类引用指向子类对象**”？



- 使用父类类型的引用指向子类的对象的时候：
  - 该引用只能调用父类中定义的方法和变量；
  - 如果子类中重写了父类中的一个方法，那么在调用这个方法的时候，将会调用子类中的这个方法；（动态连接、动态调用）
  - 变量不能被重写（覆盖），”重写“的概念只针对方法，如果在子类中”重写“了父类中的变量，那么在编译时会报错。
- **多态的3个必要条件**：
  - 继承
  - 重写
  - 父类引用指向子类对象

尝试代码：Polymorphism.java

Java 内部类[内部クラス]（inner class）

- 在Java中，也可以嵌套类（类中类）。嵌套类的目的是对应该在一起的类进行分组，这会使代码更具可读性和可维护性。
- 要访问内部类，请创建外部类的对象，然后创建内部类的对象：

```
class OuterClass {
  int x = 10;

  class InnerClass {
    int y = 5;
  }
}

public class MyMainClass {
  public static void main(String[] args) {
    OuterClass myOuter = new OuterClass();
    OuterClass.InnerClass myInner = myOuter.new InnerClass();
    System.out.println(myInner.y + myOuter.x);
  }
}

// Outputs 15 (5 + 10)
```

- 与“常规”类不同，内部类可以是private或protected或static的。如果你不希望外部对象访问内部类，则将该类声明为private。

Java 内部类[内部クラス]（inner class）

- 与“常规”类不同，内部类可以是private或protected的。如果你不希望外部对象访问内部类，则将该类声明为private：

```
class OuterClass {
  int x = 10;

  private class InnerClass {
    int y = 5;
  }
}

public class MyMainClass {
  public static void main(String[] args) {
    OuterClass myOuter = new OuterClass();
    OuterClass.InnerClass myInner = myOuter.new InnerClass();
    System.out.println(myInner.y + myOuter.x);
  }
}
```

Java 内部类[内部クラス]（inner class）

- 内部类也可以是static，这意味着你可以在不创建外部类的对象的情况下访问它：

```
class OuterClass {
  int x = 10;

  static class InnerClass {
    int y = 5;
  }
}

public class MyMainClass {
  public static void main(String[] args) {
    OuterClass.InnerClass myInner = new OuterClass.InnerClass();
    System.out.println(myInner.y);
  }
}

// Outputs 5
```

- 注意：就像static属性和方法一样，static内部类不能访问外部类的成员

Java 内部类[内部クラス]（inner class）

- 内部类[内部クラス]（非静态）的优点之一是，它们可以访问外部类的属性和方法：

```
class OuterClass {
  int x = 10;

  class InnerClass {
    public int myInnerMethod() {
      return x;
    }
  }
}

public class MyMainClass {
  public static void main(String[] args) {
    OuterClass myOuter = new OuterClass();
    OuterClass.InnerClass myInner = myOuter.new InnerClass();
    System.out.println(myInner.myInnerMethod());
  }
}

// Outputs 10
```