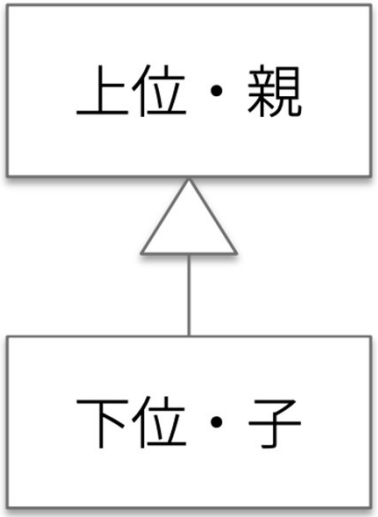
第7章　7.4節
オブジェクト指向知識補充

泛化关系[汎化]（Generalization）（⇔特化关系）

- 泛化关系[汎化]
  - 和具有共同特性的上位概念的关系表现
  - a-kind-of关系（is-a关系）



泛化关系练习

- 下列正确的是?
  - 多边形是三角形的特化
  - 狗是金毛犬的泛化
  - 博士生是学生的特化
  - 圆是椭圆的泛化

型和泛化关系[汎化]

- 泛化关系[汎化]表现的是向上转型
  - 教员拥有作为人的属性 = 可以当人来对待 = 教员【型】的对象可以用人【型】来声明

- 面向对象编程可以用超类来创造子孙后代类的对象
  - 右边的代码，只要Y extends X，则不会报错
    (Y型的对象可以作为X型的对象来使用)

```
X x;
…
x = new Y();
```

型和重写[オーバーライド]

- 可以作为超类[スーパークラス]的对象使用 = 可以使用超类[スーパークラス]的特性（继承过来的属性和方法）
- 那么，被重写[オーバーライド]的方法会怎么样?

注释（annotation[アノテーション]）

Java可以用@开始的特殊的关键字加注释
@override表示的是下面的方法是重写方法，可以避免拼写错误等无意识的重写错误

```
class X {
    …
    void method () {
      … // A
    }
}

class Y extends X {
    …
    @override
    void method() {
      … // B
    }
}
```

```
X x;
…
x = new Y();
x.method();
…
```

这里执行的是A和B里的哪个呢?

练习

- 下面代码的黄色部分会怎么样（X和Y和前一页相同）

```
X x;  Y y;

x = new Y();
x.method();

x = new X();
x.method();

y = new Y();
y.method();

y = new X();
y.method();
```

绑定（Binding）

- 绑定是一个做出决定的过程。该决定需要确定的是在程序运行的时侯，哪些方法或属性会被访问。
- 编译[コンパイル]时，为早期绑定，又称静态绑定。
- 运行[実行]时，为晚期绑定，又称动态绑定。

静态绑定（Static binding）

- 由编译器在编译[コンパイル]时就已经决定调用哪个方法或属性。

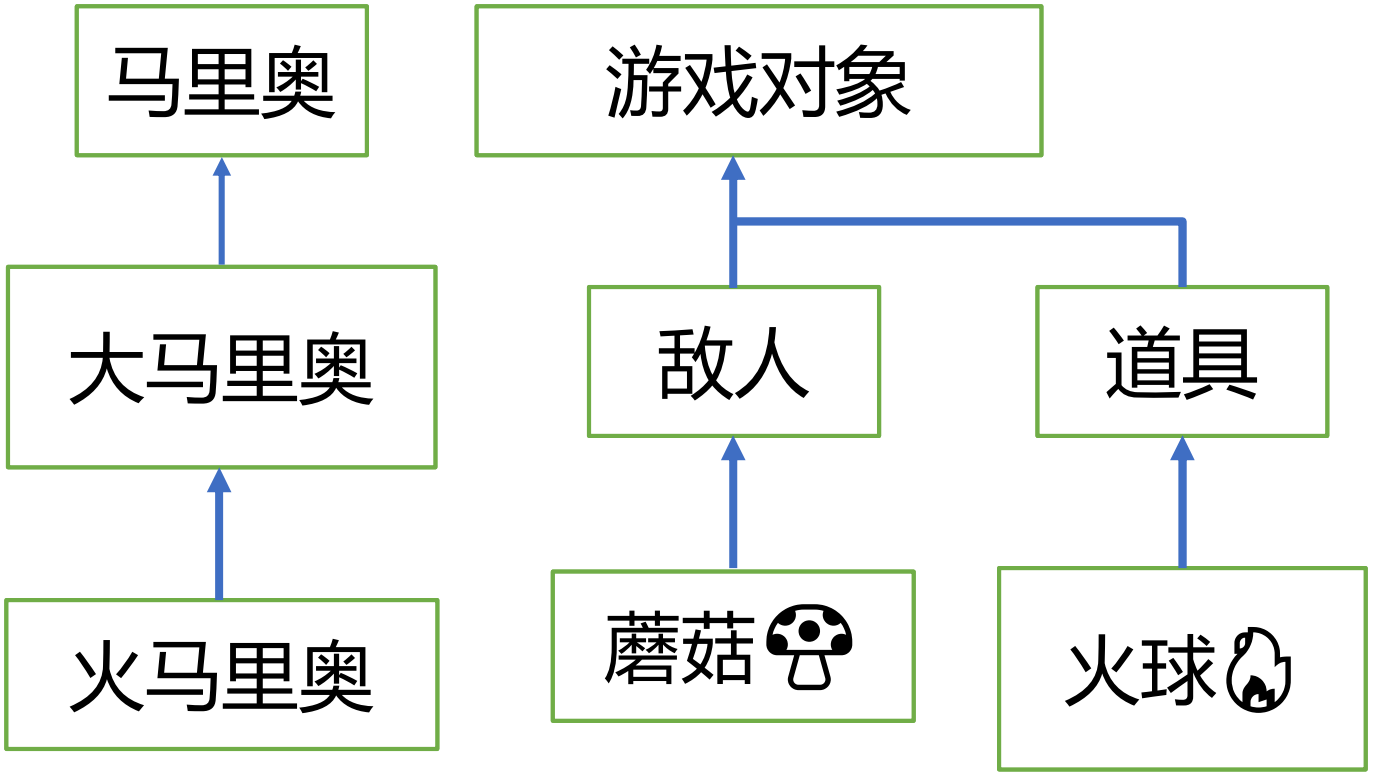
- 适用于：
  - 所有类的属性(静态或非静态)
  - static方法
  - 非static的final方法

动态绑定（Dynamic binding）

- 由JVM（Java虚拟机）在运行[実行]时才决定调用哪个方法或属性。
- 这是多态性[ポリモーフィズム]的优势
- 适用于：
  - 非static方法
  - 非final方法

动态调度（Dynamic dispatch[ディスパッチ]）

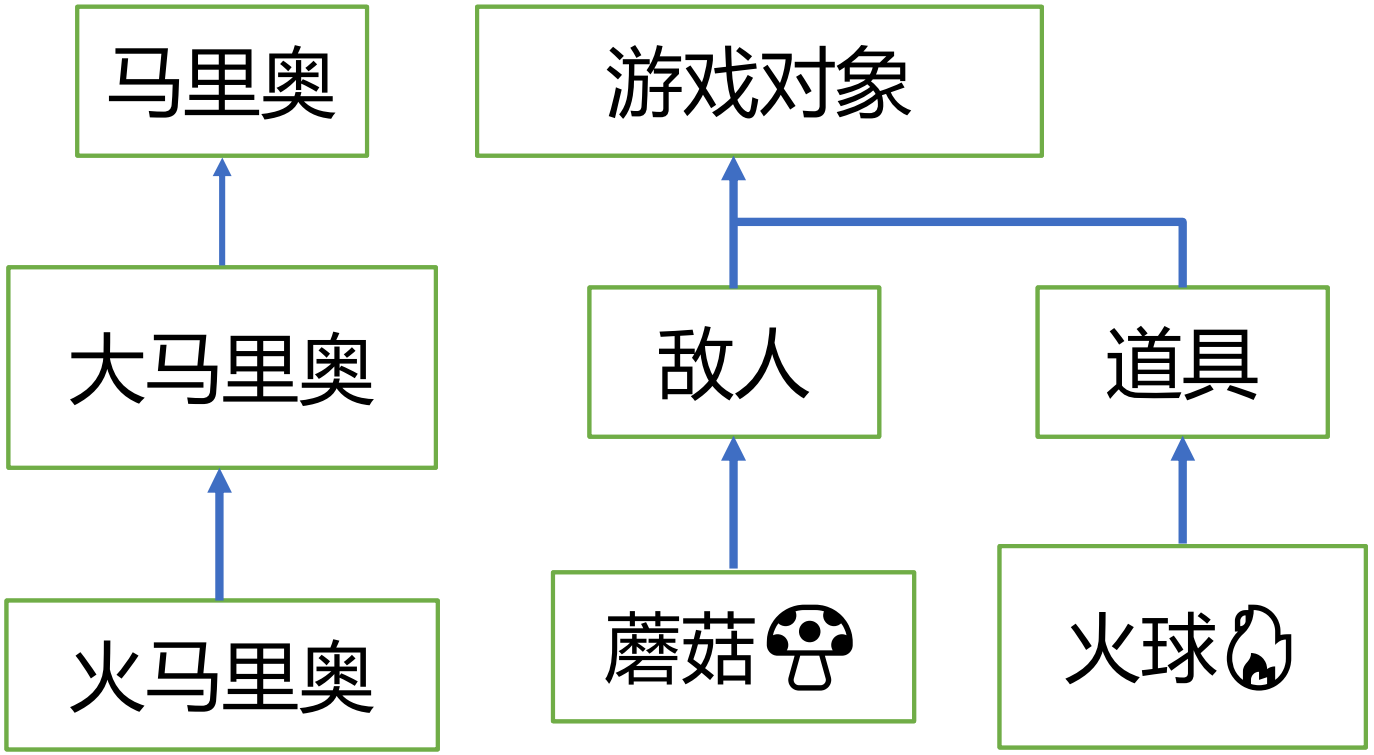
- 动态调度是指运行[実行]时选择哪一个方法来调用的过程。



单调度（Single dispatch[ディスパッチ]）

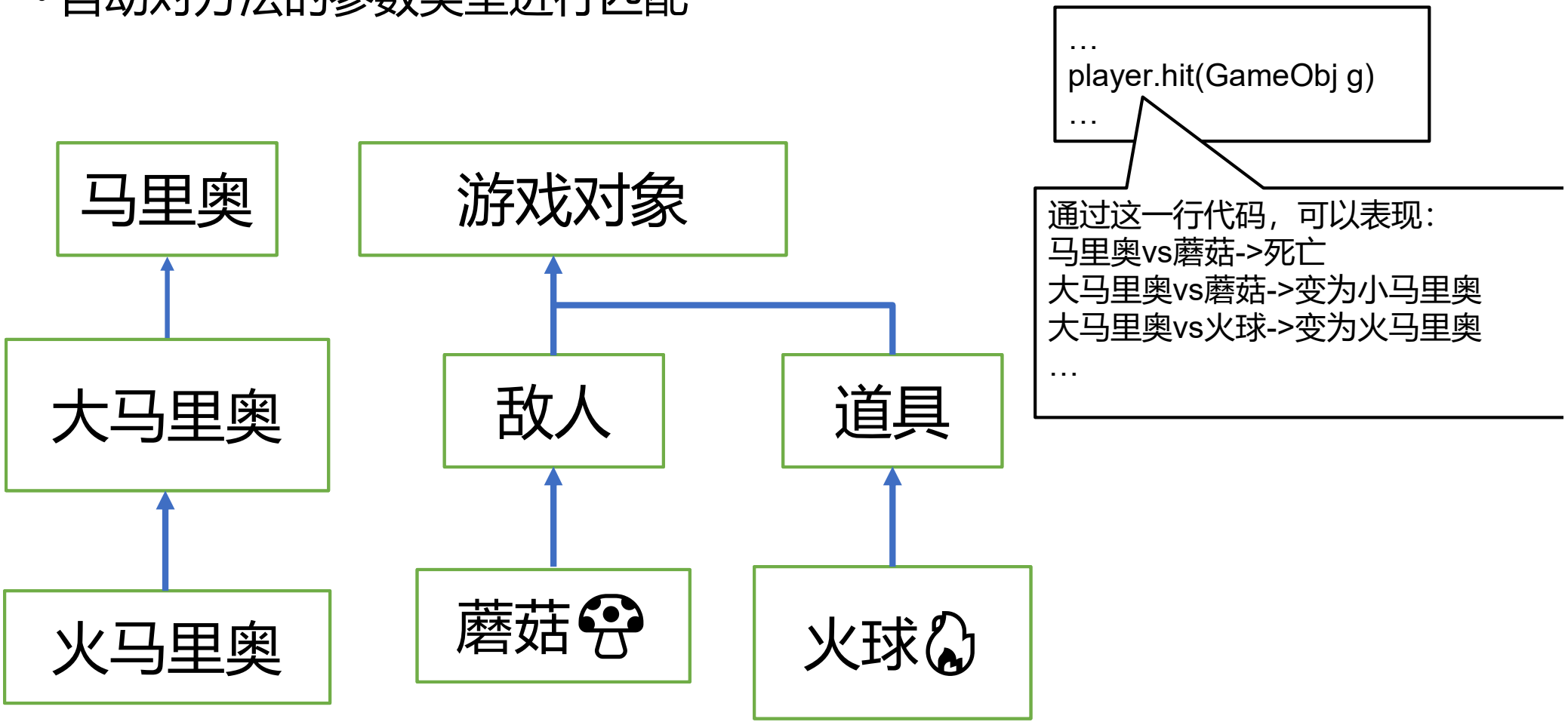
```
…
player.hit(GameObj g)
…
```

- Java是单调度
- player是Mario/BigMario/FireMario的哪个，就运行哪个的hit
- hit(GameObj g)不会对hit(Enemy e), hit(FireFlower f)进行自动匹配。



双调度

- java不支持
- 自动对方法的参数类型进行匹配



Java this关键字

- this关键字是用在方法或构造方法[コンストラクター]里指代当前对象。
- this关键字最常见的用法是类属性和具有相同名称的方法参数之间的混淆。
- this 也可以用于：
  - 调用当前类的构造方法[コンストラクター]
  - 调用当前类方法
  - 返回当前类的对象
  - 在方法调用中传递参数
  - 在构造方法[コンストラクター]调用中传递参数

Java this关键字

- 形参与成员变量名字重名，用 this 来区分：

```
public class MyClass {
  int x;

  // Constructor with a parameter
  public MyClass(int x) {
    this.x = x;
  }

  // Call the constructor
  public static void main(String[] args) {
    MyClass myObj = new MyClass(5);
    System.out.println("Value of x = " + myObj.x);
  }
}
```

Java super关键字

- super关键字是表示超类[スーパークラス]的对象。
- 它用于调用超类[スーパークラス]的方法，以及访问超类[スーパークラス]的构造方法[コンストラクター]。
- super关键字最常见的用法是消除超类[スーパークラス]和子类[サブクラス]所具有的相同名称的方法的混淆。

Java super关键字

- 使用super调用超类[スーパークラス]的方法：

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

class Dog extends Animal { // Subclass (child)
  public void animalSound() {
    super.animalSound(); // Call the superclass method
    System.out.println("The dog says: bow wow");
  }
}

public class MyMainClass {
  public static void main(String args[]) {
    Animal myDog = new Dog(); // Create a Dog object
    myDog.animalSound(); // Call the method on the Dog object
  }
}
```

尝试代码：
AnimalSuper.java

Java super关键字

- 使用super调用超类[スーパークラス]的某一个构造方法[コンストラクター]（必须在当前类构造方法[コンストラクター]里的第一行调用）
- 尝试代码：Person.java

Java的型（复习）

- 面向对象语言本来应该在内存上只保存对象
- 在Java里会对基本型[プリミティブ型]（primitive）特别处理

- Java的型=基本型[プリミティブ型] ・ 引用型[参照型]（reference）
- 基本型[プリミティブ型]
  - 布尔[真偽値型]：boolean
  - 字符[文字]：char
  - 整数：byte(8bit), short(16bit), int(32bit), long(64bit)
  - 浮点数[フロート]：float(32bit 单精度), double(64bit 双精度)

Java引用练习

- 尝试代码：Test1.java，这个代码会输出什么？为什么？

```
class MyObject{
  int data;
  void setData(int i){
    data=i;
  }
  int getData(){
    return data;
  }

  public static void main(String[] argv){
    MyObject a,b;
    a = new MyObject();
    a.setData(1);
    b = a;
    b.setData(2);
    System.out.println(a.getData());
  }
}
```

Java引用练习

- 尝试代码：Test2.java, 这个代码会怎么样?

```
class MyObject{
  int data;
  void setData(int i){
    data=i;
  }
  int getData(){
    return data;
  }

  public static void main(String[] argv){
    MyObject[] a = new MyObject[5];
    a[0].setData(2);
    System.out.println(a[0].getData());
  }
}
```

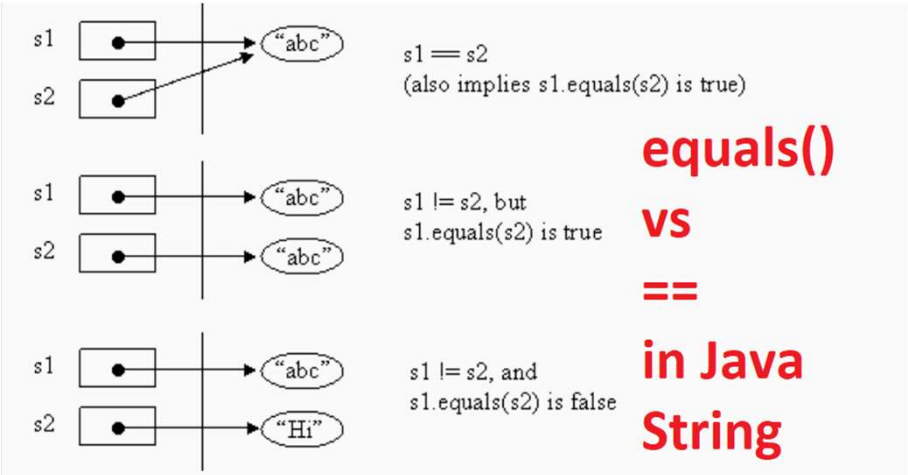
```
class MyObject{

  public static void main(String[] argv){
    int[] a = new int[5];
    a[0] = 2;
    System.out.println(a[0]);
  }
}
```

对比：这段代码又会怎么样?

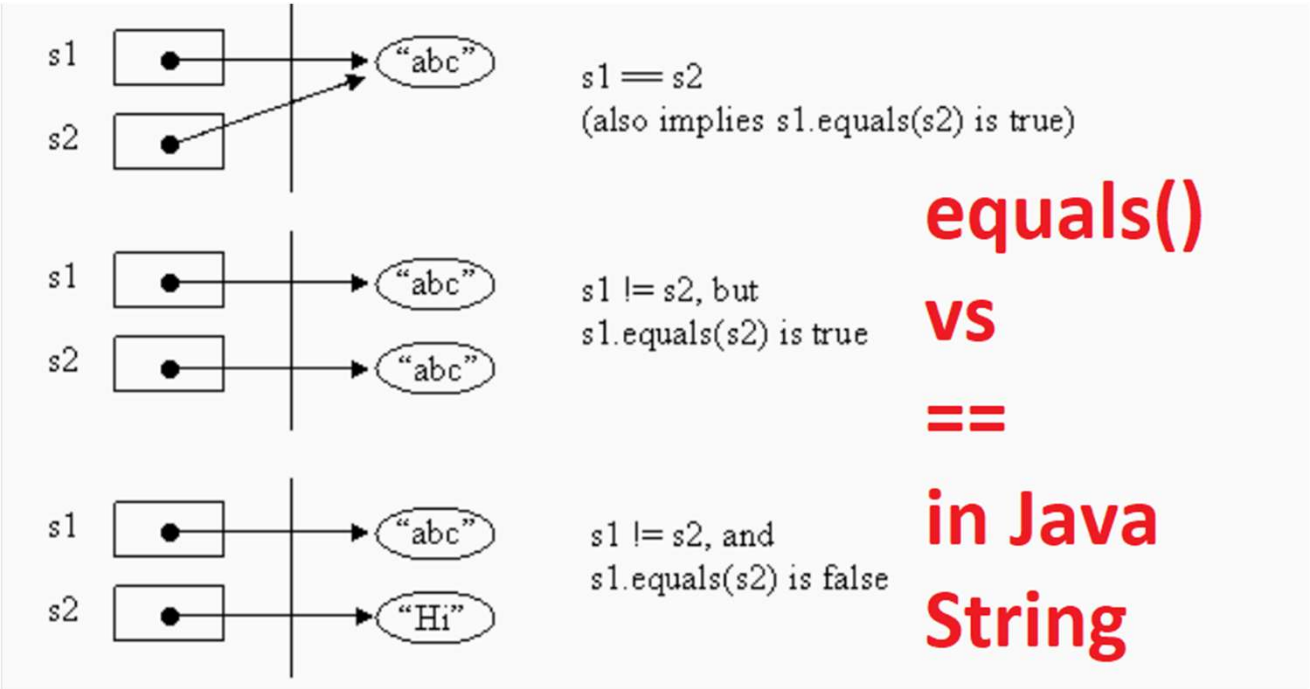
Java中的「一致」：same和equals

- Object.equals(Object o) 方法
  - 表示的是不同对象的内容上的等价关系
  - 此方法可以被重写[オーバーライド]（override）再定义
    - 所有类的超类[スーパークラス]Object里面已经定义了equals方法
  - 例：对于同一个人的定义
    - 满足 「名字和出生年月日一样」 的话，就是同一个人?
    - 满足 「名字，籍贯和出生年月日一样」 的话，就是同一个人?



Java中的「一致」：same和equals

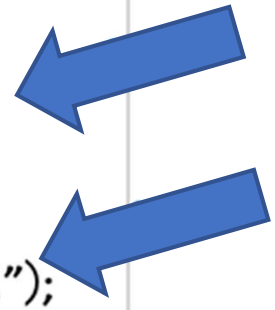
- 运算符 ==
  - Java的运算符==表示same（同一个对象）
    - 也就是说在指向同一个内存空间里的同一个对象
  - 对于基本型[プリミティブ型]，只要表现的值一样，就是same关系，1==(1+1)-1



Java中容易出错的字符串处理

- Java里的字符串是String类的对象
  - 因为String类是java.lang包的类所以不需要import
  - 已经生成的String对象不可以再更改
- 注意：字符串要用equals（）判断一致性
  - ‘==’判断的是：是不是同一个对象

```
String x = “abc”; // new String(“abc”) と同じ
String y = “ab”;
y = y + “c”;
if (x == y) {
    System.out.println(“Same String!”);
} else {
    System.out.println(“Different String!”);
}
```



x，y都是“abc”

这里会输出什么？

尝试代码：StringEquals.java

Java copy

- 尝试代码:Copy1.java
- 这个代码会输出什么?

```
  :
  :
int[] a = new  int[2];
int[] b;
a[0] =0;
a[1] =10;
b = a.clone();
System.out.print(b[0]+",");
b[0] = 2;
System.out.print(b[0]+",");
System.out.println(a[0]);
```

注意这个copy

Java copy

- 尝试代码：Copy2.java
- 这个代码会输出什么?

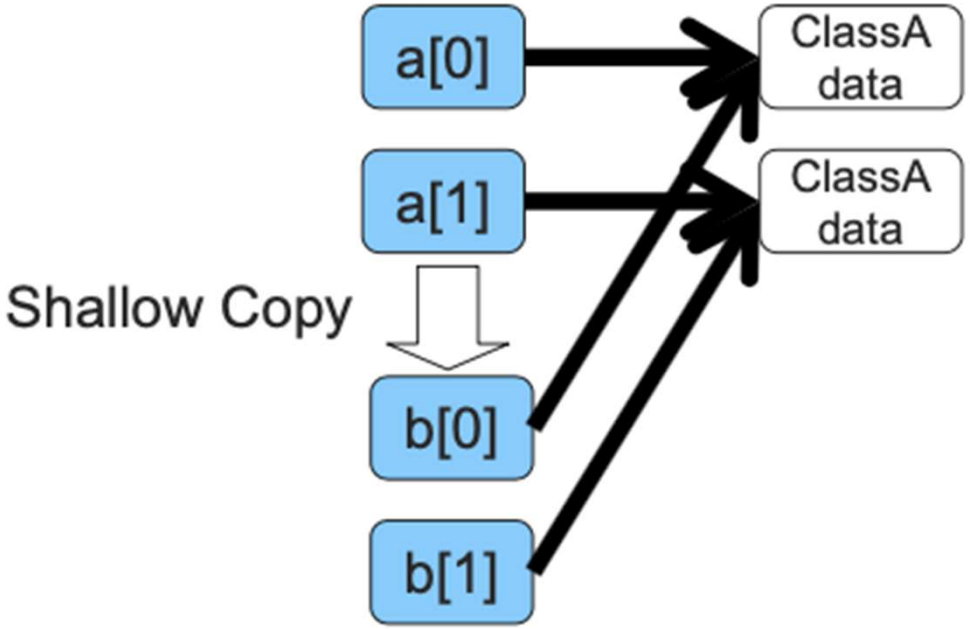
```
ClassA[] a = new ClassA[2];
ClassA[] b;
a[0] = new ClassA(0);
a[1] = new ClassA(10);
b = a.clone();
System.out.print(b[0]+",");
b[0].set(2);
System.out.print(b[0]+",");
System.out.println(a[0]);
```

注意这个copy

```
class ClassA{
  int data;
  ClassA(int i){
    data = i;
  }
  void set(int i){
    data = i;
  }
  public String toString(){
    return Integer.toString(data);
  }
}
```

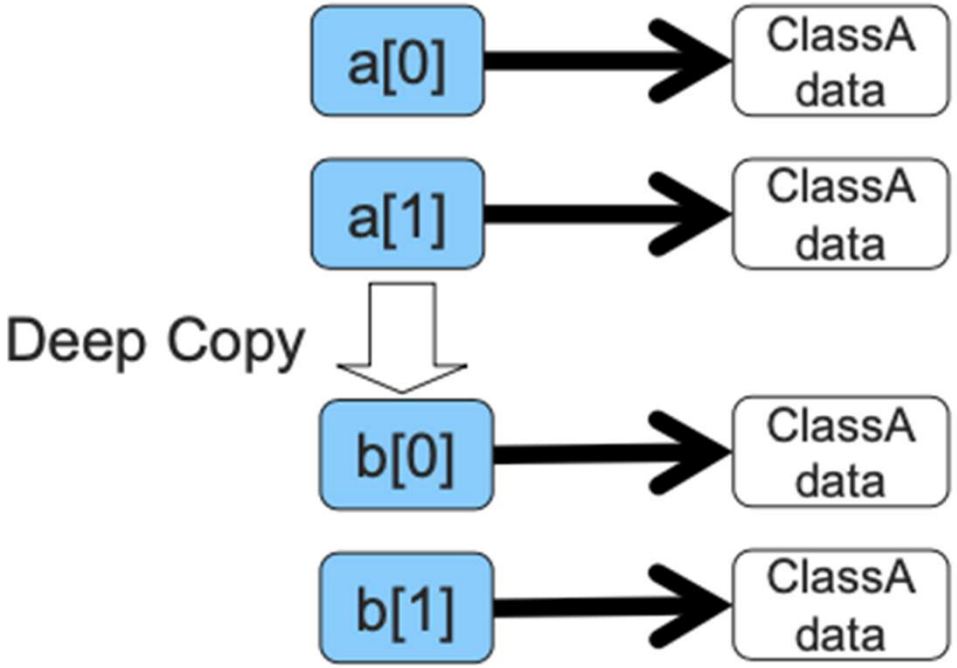
浅Copy[シャローコピー]（Shallow Copy）

- Shallow Copy 引用的目标是共通的（Object的clone方法即这个）
  - 复制的是引用
  - 和copy元共有引用目标的数据
  - 短时间内完成copy，内存耗用小



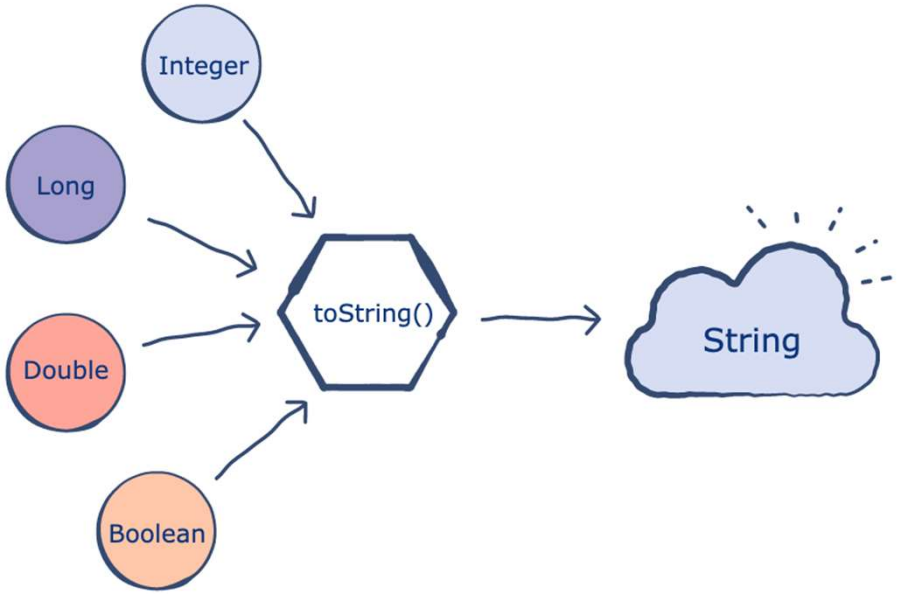
深Copy[ディープコピー]（Deep Copy）

- Deep Copy 直接复制引用目标
  - 独立于copy元引用不同的数据
  - 消耗时间长，需要内存多



Object toString()

- Object的toString（）方法
  - 通常用于Debug
  - 用字符串表现对象
  - 在Object类里面定义，所有的类都可以对其重写[オーバーライド]。



String字符串连结的问题

- String字符串的连结=新字符串的生成=高cost
  - 从内存角度考虑，频繁的字符串连结推荐使用StringBuffer类（特别是循环里面的连结）
- java.lang.StringBuffer 的使用方法
  - new出来的是空字符串
  - 用append/insert等方法进行字符串的追加/插入/删除操作
  - 最后通过toString()方法将其变成String类的对象

```
StringBuffer sb = new StringBuffer();
sb.append(“abc”);
sb.append(“123”).append(“efg”);
sb.insert(0, “xyz”);
System.out.println(sb.toString());
```

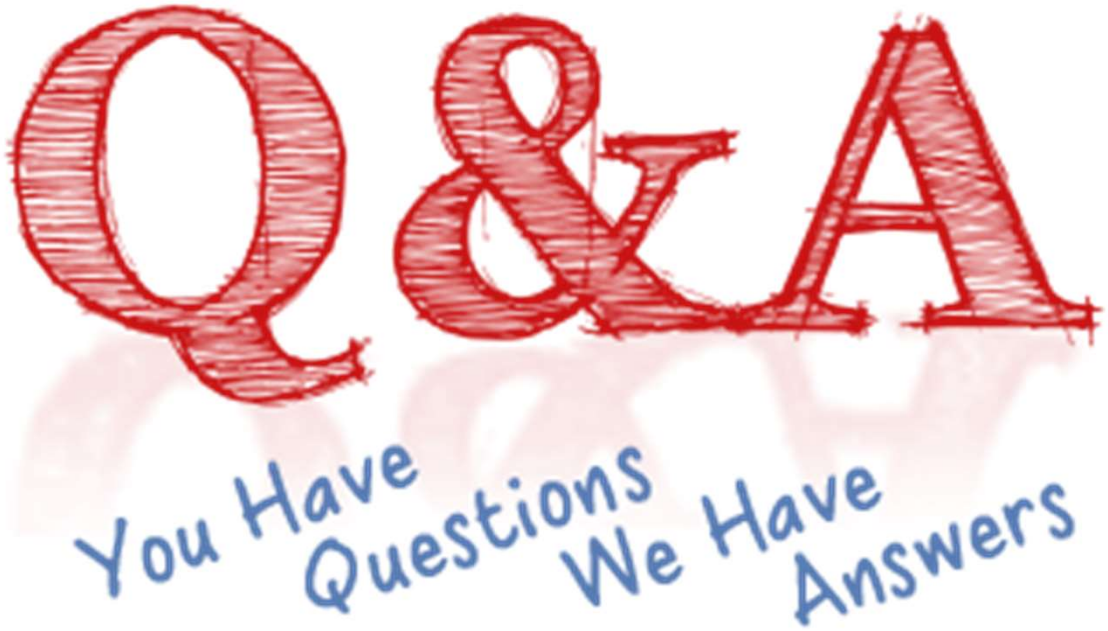
“xyzabc123efg”

经常使用的String方法

- public String toUpperCase(), toLowerCase() :
  - 英文大小写转换
- public String[] split(String rex, int limit) :
  - 返回通过正则表达式rex分割的字符串的数组
  - limit参数指定分割的次数。limit是0以下表示无次数限制，limit是0的话会破坏空字符串
- public String trim :
  - 返回消除掉开头和末尾的空白文字的字符串
  - 经常用于消除split方法返回的字符串数组里面的字符串前后的空白

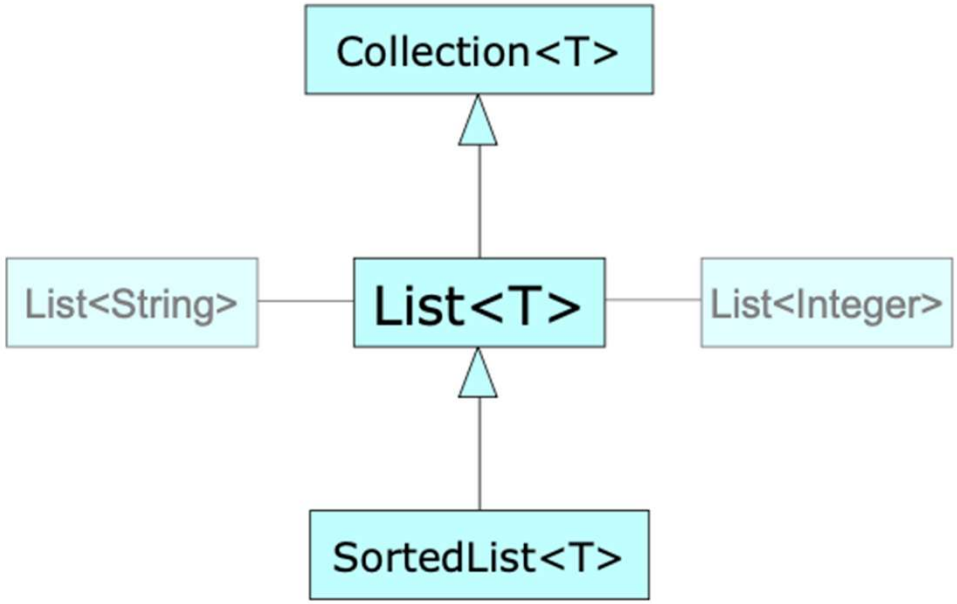
经常使用的String方法

- public String substring(int from, int to):
  - 以0为基准点，生成第（from）位字符到第（to-1）位字符的部分字符串并且返回。
- public String replaceFirst(String rex, String rpl) :
  - 将字符串中和正则表达式rex的第一次匹配到的字符串部分置换成rpl
- public String replaceAll(String rex, String rpl) :
  - 将字符串中和正则表达式rex的所有匹配到的字符串部分置都换成rpl



Java 泛型[総称型]（Generics）

- 泛型[総称型]的本质是参数化类型，也就是说所操作的数据类型被当作一个参数。
- 从Java 5.0引入的特性

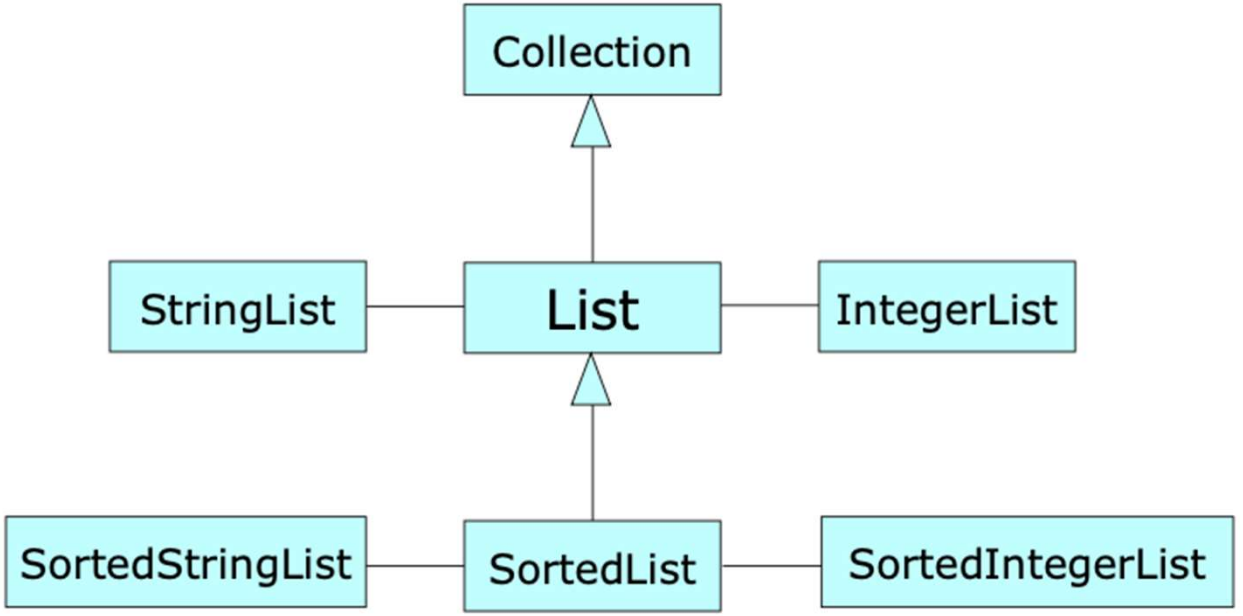


泛型[総称型]的必要性

- 问题：如何定义一个实现栈[スタック]的Stack类
- 针对多种数据类型有多种Stack：
  - 保存Integer的Stack
  - 保存String的Stack
  - 保存Long的Stack
  - …

解决办法1:制作专用的类

- 数据结构总数 * 数据类型总数 个类--->效率很低



解决办法2:利用Object类

- Object类（Java里所有类的超类[スーパークラス]，所有的类都自动地继承它）

```
class Stack {
    void push(Object o) {…}
    Object pop() {…}
}
```

- 问题点
  - 不支持静态检查类型

```
stack.push("Hello");
stack.push(new Integer(1));
String str = (String) stack.pop();
```

明明是Integer型，
却用String型对待

最优解决办法：泛型[総称型]

- 将类型参数化

```
//定义
class Stack<T> {
 void push(T o)
 T pop() {…}
}
```

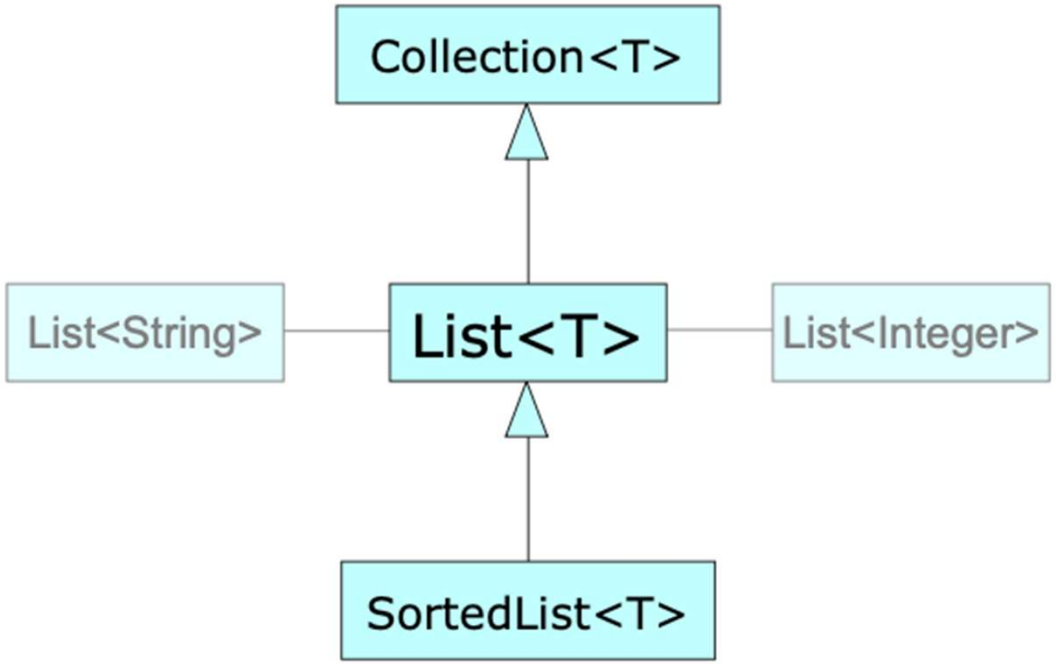
Stack内部保存的数据类型是T型

```
//使用
Stack<String> strStack
  = new Stack<>();
…
String str = strStack.pop();
```

T=String，也就是说声明了一个只能放String的Stack

继承和泛型[総称型]

- 通过使用泛型[総称型]，继承能够吸收参数的类型



继承和泛型[総称型]

- 继承（抽象数据类型）
  - 对象的型相同，算法（内部实现方式）不同
    - 例）用链表实现的栈[スタック]，用数组实现的栈[スタック]
- 泛型[総称型]
  - 算法相同，对象的型不同
    - 例）存String的栈[スタック]，存Integer的栈[スタック]