

方法 Method

刘 钦

南京大学软件学院

Outline

- 方法的概念
- 方法的属性
- 方法的行为
- 方法的实现
- 方法的特例

Outline

- 方法的概念
 - 概念
 - 属性
 - 行为
- 方法的实现
- 方法的特例

方法的概念

- 方法的概念
 - 物理的角度：指令块
 - 逻辑的角度：抽象指令单元
 - 语义的角度：行为
- 分类
 - 类的行为：静态方法
 - 对象的行为：成员方法

方法对象的属性

- 名字
- 所有者
- 地址
- 接口
- 实现
- 运行期
- 可见性

方法的命名

- 方法名中的第一个单词小写，其后每个单词的第一个字母大写
- 一般用动词，或者动宾短语
- 跟boolean相关的， isValid
- getValue,setValue

所有者

- 类的行为
 - 静态方法
 - 类名来调用
- 对象的行为
 - 成员方法
 - 对象的引用来调用

Program Memory

- Data Segment
 - Data
 - The data area contains global and static variables used by the program that are explicitly initialized with a value. This segment can be further classified into a read-only area and read-write area.
 - `const char* string = "hello world"`
 - BSS
 - uninitialized data, starts at the end of the data segment and contains all global variables and static variables that are initialized to zero or do not have explicit initialization in source code
 - `static int i`
 - Heap
- Stack
- Code segment

Linux地址空间

- 代码段的位置

代码区（只读）



JVM Run-Time Data Areas

- The pc Register
- Java Virtual Machine Stacks
- Heap
- Method Area
- Run-Time Constant Pool
- Native Method Stacks

Method Areas

- The Java Virtual Machine has a method area that is shared among all Java Virtual Machine threads.
- The method area is analogous to the storage area for compiled code of a conventional language or analogous to the "text" segment in an operating system process.
- It stores per-class structures such as the run-time constant pool, field and method data, and the code for methods and constructors, including the special methods (§2.9) used in class and instance initialization and interface initialization.

接口

- 返回值
- 方法名
- 参数
 - 个数
 - 类型
 - 顺序

实现

- {}内的语句

运行期

- 程序执行的时间段
- 方法调用
 - 同步
 - 调用方法和被调用方法相关（等待被调用方法执行完）
 - 异步
 - 调用方法和被调用方法不相关

可见性

- Public
- Protected
- 缺省
- Private

方法对象的行为

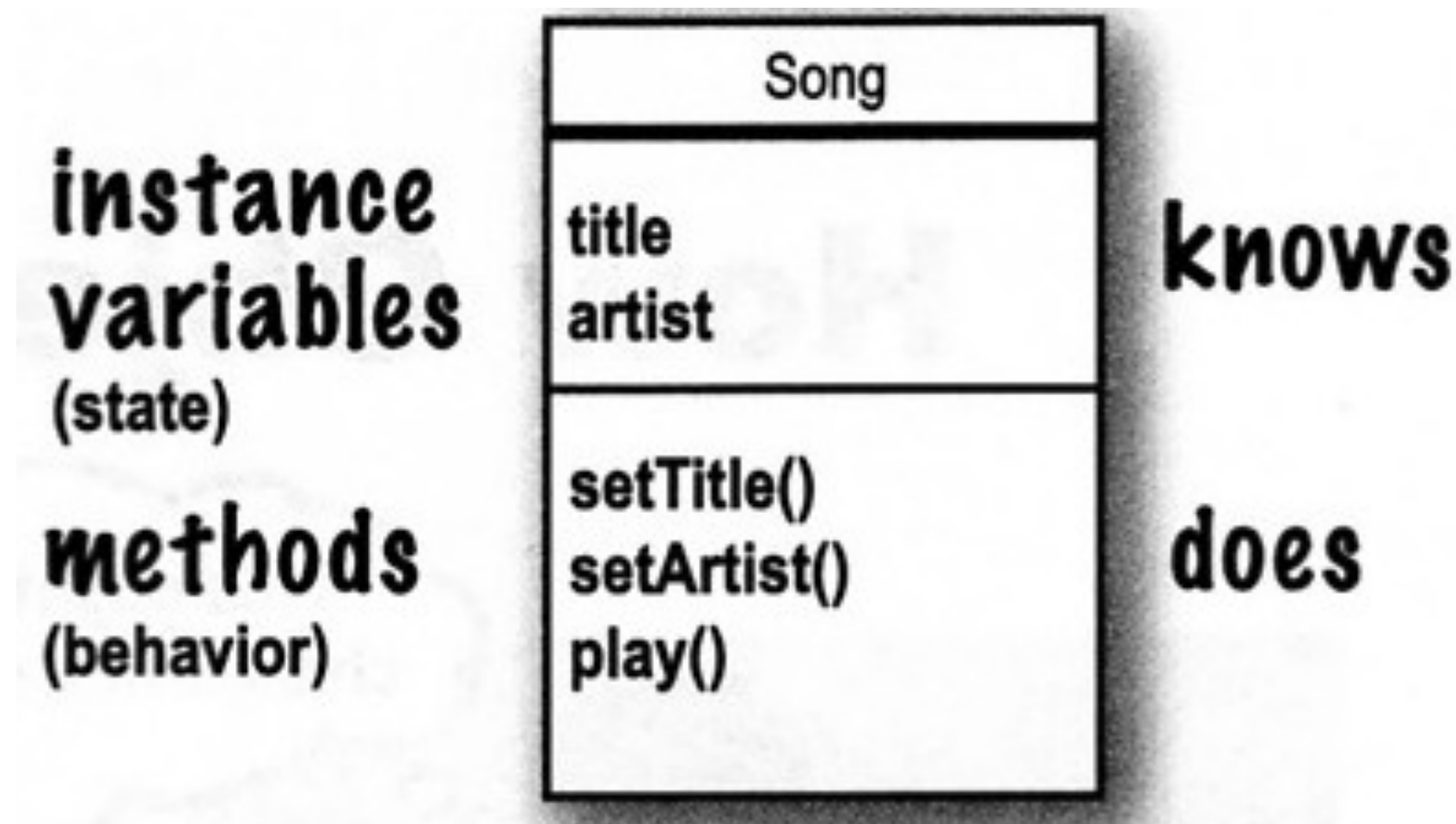
- 执行（被调用）

Outline

- 方法的概念
 - 方法的属性
 - 方法的行为
- 方法的实现
 - 成员方法的声明
 - 成员方法的调用机制
 - 方法的结构
 - 顺序（表达式，语句，块）
 - 选择
 - 循环
 - 局部变量的初始化
- 方法的特例
 - 递归方法
 - 常用数据类型的方法

方法的声明

- 修饰词 返回类型 方法名称(参数列表) {
- 方法体
- }
-



A class describes what an object knows and what an object does

成员方法的调用

Every instance of a particular class has the same methods, but the methods can *behave* differently based on the value of the instance variables.

The Song class has two instance variables, *title* and *artist*. The `play()` method plays a song, but the instance you call `play()` on will play the song represented by the value of the *title* instance variable for that instance. So, if you call the `play()` method on one instance you'll hear the song "Politik", while another instance plays "Darkstar". The method code, however, is the same.

```
void play() {  
    soundPlayer.playSound(title);  
}
```

方法的被同步调用

- 方法被同步调用的特性
 - 每个方法都只有一个入口。
 - 当执行被调用的方法的时候，调用方法暂停。
 - 当方法结束时，程序的控制权交还给调用处。

```
class Dog {  
    int size;  
    String name;
```

```
    void bark() {  
        if (size > 60) {  
            System.out.println("Woof! Woof!");  
        } else if (size > 14) {  
            System.out.println("Ruff! Ruff!");  
        } else {  
            System.out.println("Yip! Yip!");  
        }  
    }  
}
```

Dog
size name
bark()

```
class DogTestDrive {  
    public static void main (String[] args) {  
        Dog one = new Dog();  
        one.size = 70;  
        Dog two = new Dog();  
        two.size = 8;  
        Dog three = new Dog();  
        three.size = 35;  
  
        one.bark();  
        two.bark();  
        three.bark();  
    }  
}
```

File Edit Window Help Playdead

```
%java DogTestDrive  
Woof! Woof!  
Yip! Yip!  
Ruff! Ruff!
```


Parameters and arguments

A method uses parameters. A caller passes arguments.

- 1 Call the bark method on the Dog reference, and pass in the value 3 (as the argument to the method).

```
Dog d = new Dog();
```

```
d.bark(3);
```

argument

- 2 The bits representing the int value 3 are delivered into the bark method.

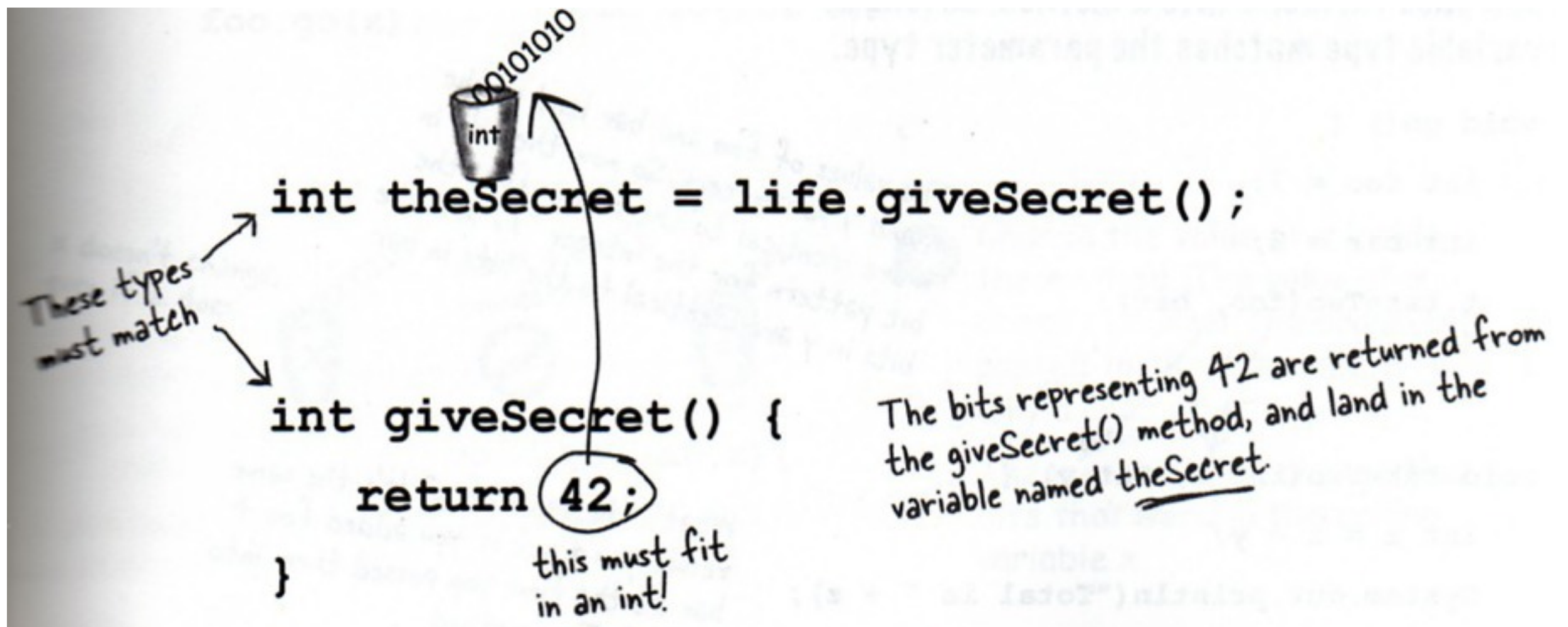
- 3 The bits land in the numOfBarks parameter (an int-sized variable).

```
void bark(int numOfBarks) {  
    while (numOfBarks > 0) {  
        System.out.println("ruff");  
        numOfBarks = numOfBarks - 1;  
    }  
}
```

- 4 Use the numOfBarks parameter as a variable in the method code.

Return values

If you define a return value, you must return one.



Multi arguments

Calling a two-parameter method, and sending it two arguments.

```
void go() {  
    TestStuff t = new TestStuff();  
    t.takeTwo(12, 34);  
}
```

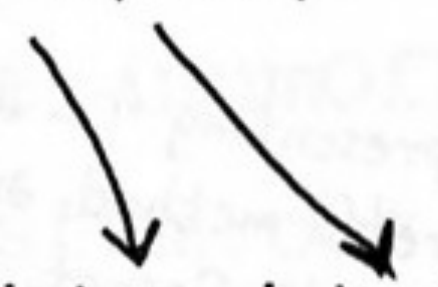
```
void takeTwo(int x, int y) {  
    int z = x + y;  
    System.out.println("Total is " + z);  
}
```

The arguments you pass land in the same order you passed them. First argument lands in the first parameter, second argument in the second parameter, and so on.

Pass variables into a method

You can pass variables into a method, as long as the variable type matches the parameter type.

```
void go() {  
    int foo = 7;  
    int bar = 3;  
    t.takeTwo(foo, bar);  
}  
  
void takeTwo(int x, int y) {  
    int z = x + y;  
    System.out.println("Total is " + z);  
}
```

A diagram with two arrows. One arrow starts at the 'foo' parameter in the 'takeTwo' method signature and points to the 'foo' variable in the 'go' method body. The other arrow starts at the 'bar' parameter in the 'takeTwo' method signature and points to the 'bar' variable in the 'go' method body.

The values of `foo` and `bar` land in the `x` and `y` parameters. So now the bits in `x` are identical to the bits in `foo` (the bit pattern for the integer '7') and the bits in `y` are identical to the bits in `bar`.

What's the value of `z`? It's the same result you'd get if you added `foo` + `bar` at the time you passed them into the `takeTwo` method

Java is pass-by-value.

That means pass-by-copy.

```
int x = 7;
```



1

Declare an int variable and assign it the value '7'. The bit pattern for 7 goes into the variable named x.

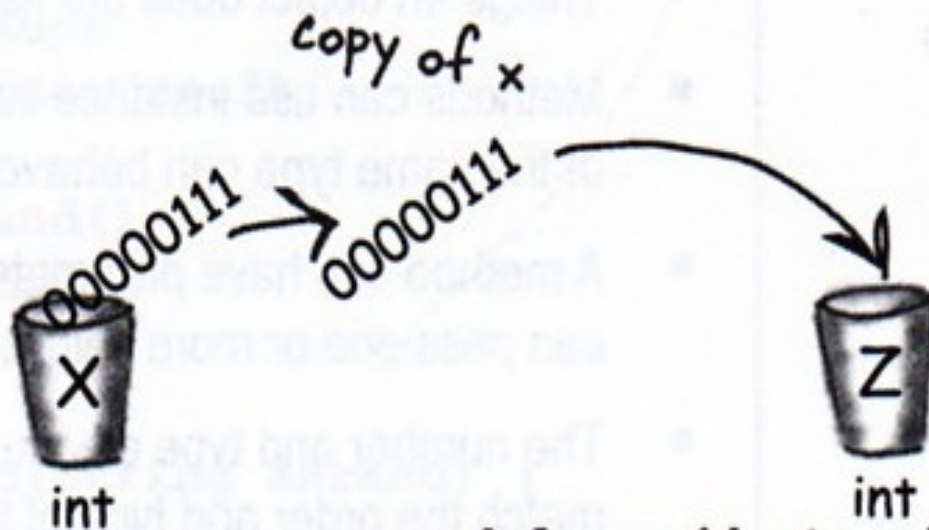


```
void go(int z) { }
```



2

Declare a method with an int parameter named z.



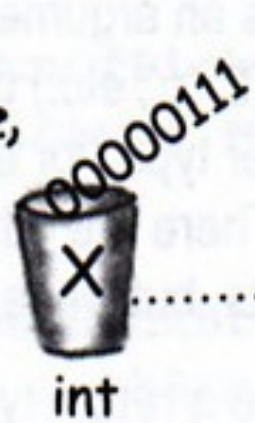
3

Call the go() method, passing the variable x as the argument. The bits in x are copied, and the copy lands in z.

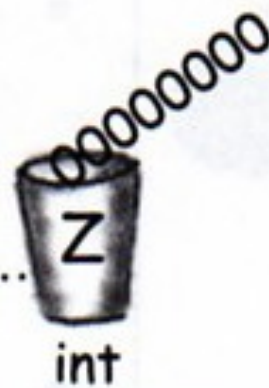
```
foo.go(x);
```

```
void go(int z) { }
```

x doesn't change,
even if z does.



x and z aren't
connected

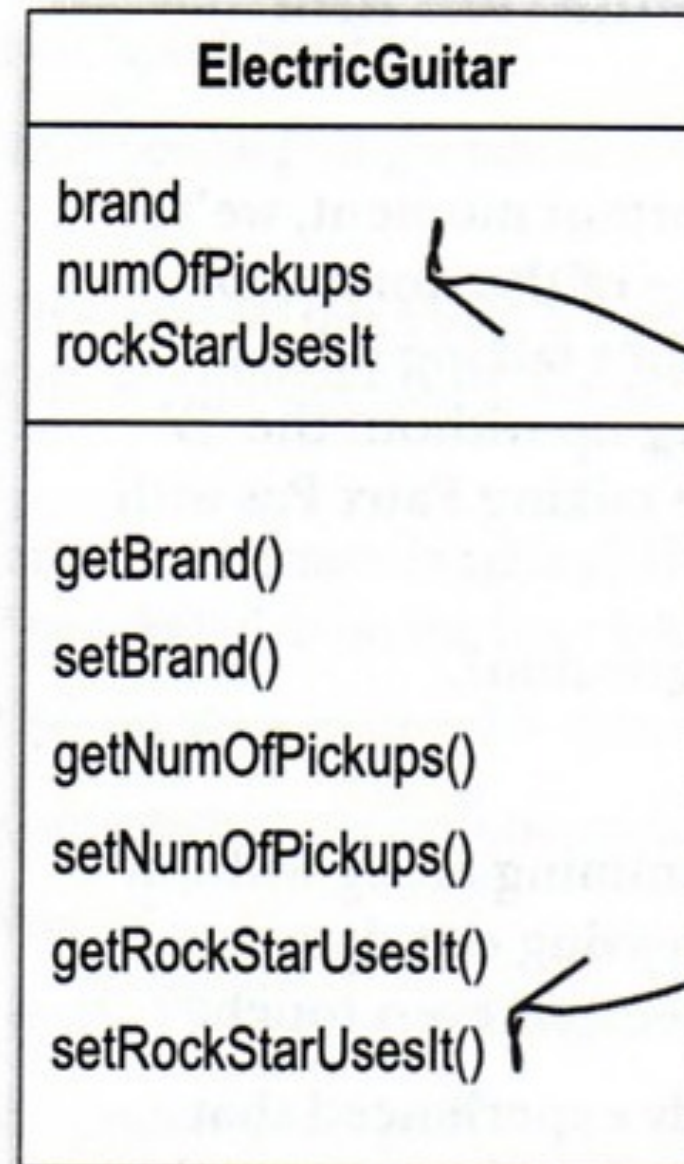


```
void go(int z) {  
    z = 0;  
}
```

4 Change the value of `z` inside the method. The value of `x` doesn't change! The argument passed to the `z` parameter was only a copy of `x`.

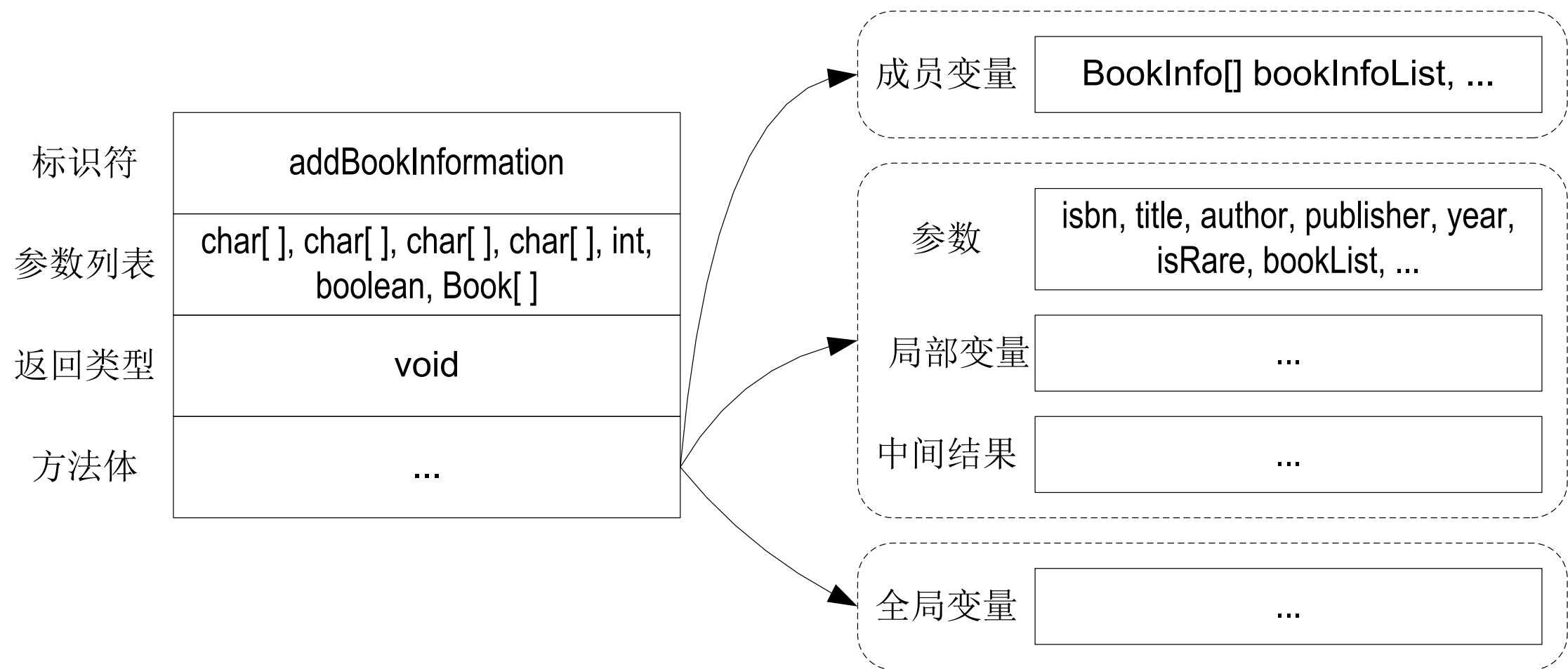
The method can't change the bits that were in the calling variable `x`.

Getter and Setter



Note: Using these naming conventions means you'll be following an important Java standard!

```
class ElectricGuitar {  
  
    String brand;  
    int numOfPickups;  
    boolean rockStarUsesIt;  
  
    String getBrand() {  
        return brand;  
    }  
  
    void setBrand(String aBrand) {  
        brand = aBrand;  
    }  
  
    int getNumOfPickups() {  
        return numOfPickups;  
    }  
  
    void setNumOfPickups(int num) {  
        numOfPickups = num;  
    }  
  
    boolean getRockStarUsesIt() {  
        return rockStarUsesIt;  
    }  
  
    void setRockStarUsesIt(boolean yesOrNo) {  
        rockStarUsesIt = yesOrNo;  
    }  
}
```



方法访问数据的示例

Outline

- 方法的概念
- 方法的实现
 - 成员方法的调用机制
 - 方法的结构
 - 顺序（表达式，语句，块）
 - 选择
 - 循环
 - 局部变量的初始化
- 方法的特例

Expressions

- An expression is a construct made up of variables, operators, and method invocations, which are constructed according to the syntax of the language, that evaluates to a single value.

Statements

- Expression statement
 - Statements are roughly equivalent to sentences in natural languages. A statement forms a complete unit of execution.
 - The following types of expressions can be made into a statement by terminating the expression with a semicolon (;).
 - Assignment expressions
 - Any use of ++ or --
 - Method invocations
 - Object creation expressions
- Declaration statement
- Control flow statement

Blocks

- A block is a group of zero or more statements between balanced braces and can be used anywhere a single statement is allowed.

If else

```
| if(Boolean-expression)  
|     statement
```

or

```
| if(Boolean-expression)  
|     statement  
| else  
|     statement
```

switch

```
//: control/VowelsAndConsonants.java
// Demonstrates the switch statement.
import java.util.*;
import static net.mindview.util.Print.*;

public class VowelsAndConsonants {
    public static void main(String[] args) {
        Random rand = new Random(47);
        for(int i = 0; i < 100; i++) {
            int c = rand.nextInt(26) + 'a';
            printnb((char)c + ", " + c + ": ");
            switch(c) {
                case 'a':
                case 'e':
                case 'i':
                case 'o':
                case 'u': print("vowel");
                        break;
                case 'y':
                case 'w': print("Sometimes a vowel");
                        break;
                default: print("consonant");
            }
        }
    }
} /* Output:
```

```
y, 121: Sometimes a vowel
n, 110: consonant
z, 122: consonant
b, 98: consonant
r, 114: consonant
n, 110: consonant
y, 121: Sometimes a vowel
g, 103: consonant
c, 99: consonant
f, 102: consonant
o, 111: vowel
w, 119: Sometimes a vowel
z, 122: consonant
...
*///:~
```

Java SE 7 新特性

```
public class StringSwitchDemo {
```

```
    public static int getMonthNumber(String month) {
```

```
        int monthNumber = 0;
```

```
        if (month == null) {  
            return monthNumber;  
        }
```

```
        switch (month.toLowerCase()) {
```

```
            case "january":  
                monthNumber = 1;  
                break;
```

```
            case "february":  
                monthNumber = 2;  
                break;
```

```
            case "march":  
                monthNumber = 3;  
                break;
```

```
        ...
```

```
            case "december":  
                monthNumber = 12;  
                break;
```

```
            default:  
                monthNumber = 0;  
                break;
```

```
        }
```

```
        return monthNumber;
```

```
    }
```

```
    ...  
}
```

Iteration

```
| while(Boolean-expression)  
|     statement
```

```
| do  
|     statement  
| while(Boolean-expression);
```

```
| for(initialization; Boolean-expression; step)  
|     statement
```

for each syntax

```
//: control/ForEachFloat.java
import java.util.*;

public class ForEachFloat {
    public static void main(String[] args) {
        Random rand = new Random(47);
        float f[] = new float[10];
        for(int i = 0; i < 10; i++)
            f[i] = rand.nextFloat();
        for(float x : f)
            System.out.println(x);
    }
} /* Output:
```

```
//: control/ForEachString.java

public class ForEachString {
    public static void main(String[] args) {
        for(char c : "An African Swallow".toCharArray() )
            System.out.print(c + " ");
    }
} /* Output:
A n   A f r i c a n   S w a l l o w
*///:~
```


break & continue

- break
 - breaks out of the inner iteration and you end up in the outer iteration.
 - break label (breaks all the way out to label, but it does not reenter the iteration)
- continue
 - the continue moves back to the beginning of the inner iteration.
 - continue label (reenter the iteration)

break标签

- 下面的例子中采用break加标签的方式，可以跳出三层的循环语句，继续执行Loop4所标识的语句：
- ```
int i = 1, j = 1, k = 1, n = 0;
```
- ```
boolean flag = false;
```
- ```
Loop1: for (i = 1; i <= 10; i++) {
```
- ```
    Loop2: for (j = 1; j <= 10; j++) {
```
- ```
 Loop3: for (k = 1; k <= 10; k++) {
```
- ```
            if (i + j + k == 10) {
```
- ```
 flag = true;
```
- ```
                break Loop1;
```
- ```
 if (k >= 1){
```
- ```
                    Loop4: n = 1;
```
- ```
 }
```
- ```
            }
```
- ```
 }
```
- ```
    }
```
- ```
}
```
- ```
/* 结果为：i的值为1，j的值为1，k的值为8，flag的值为true */
```
- ```
Loop5: for (int m = 1; m <= 10; m++) {
```
- ```
    ...
```
- ```
}
```
- 需要注意的是，break加标签只能用于跳出包含break语句的块，而不能用于跳入块或是跳转到其它地方。例如在上面的例子中，break Loop4或者break Loop5都是非法的。

# return

---

- Specifies what value a method will return
- Causes the current method to exit

# Infamous goto

---

- 1968
- Edsger W. Dijkstra
- A Case against the GO TO Statement

# Outline

---

- 方法的概念
- 方法的实现
  - 成员方法的调用机制
  - 方法的结构
    - 顺序（表达式，语句，块）
    - 选择
    - 循环
  - 局部变量的初始化
- 方法的特例

# Encapsulating the GoodDog class

Make the instance variable private.

Make the getter and setter methods public.

```
class GoodDog {
```

```
 private int size;
```

```
 public int getSize() {
```

```
 return size;
```

```
 }
```

```
 public void setSize(int s) {
```

```
 size = s;
```

```
 }
```

```
 void bark() {
```

```
 if (size > 60) {
```

```
 System.out.println("Woof! Woof!");
```

```
 } else if (size > 14) {
```

```
 System.out.println("Ruff! Ruff!");
```

```
 } else {
```

```
 System.out.println("Yip! Yip!");
```

```
 }
```

```
 }
```

```
}
```

| GoodDog                          |
|----------------------------------|
| size                             |
| getSize()<br>setSize()<br>bark() |

Even though the methods don't really add new functionality, the cool thing is that you can change your mind later. you can come back and make a method safer, faster, better.

**Any place where a particular value can be used, a *method call that returns that type* can be used.**

**instead of:**

```
int x = 3 + 24;
```

**you can say:**

```
int x = 3 + one.getSize();
```

```
class GoodDogTestDrive {

 public static void main (String[] args) {
 GoodDog one = new GoodDog();
 one.setSize(70);
 GoodDog two = new GoodDog();
 two.setSize(8);
 System.out.println("Dog one: " + one.getSize());
 System.out.println("Dog two: " + two.getSize());
 one.bark();
 two.bark();
 }
}
```



# What is the value before you initialize a variable?

---

## Declaring and initializing instance variables

```
int size = 420;
String name = "Donny";
```

```
class PoorDog {
```

```
 private int size;
 private String name;
```

declare two instance variables,  
but don't assign a value

```
 public int getSize() {
 return size;
 }
```

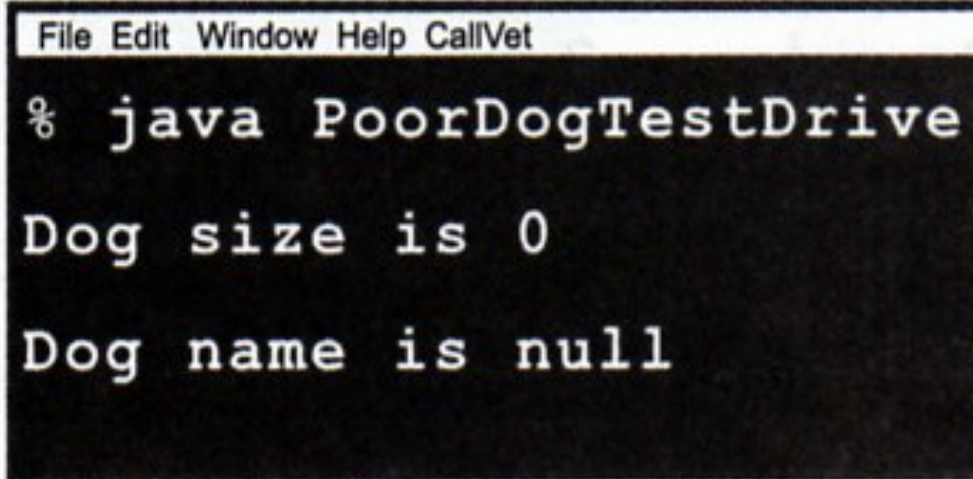
What will these return??

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

```
}
```

```
public class PoorDogTestDrive {
 public static void main (String[] args) {
 PoorDog one = new PoorDog();
 System.out.println("Dog size is " + one.getSize());
 System.out.println("Dog name is " + one.getName());
 }
}
```

What do you think? Will this even compile?



The screenshot shows a terminal window with a menu bar at the top containing 'File', 'Edit', 'Window', 'Help', and 'CallVet'. The terminal output is as follows:

```
% java PoorDogTestDrive
Dog size is 0
Dog name is null
```

You don't have to initialize instance variables, because they always have a default value. Number primitives (including char) get 0, booleans get false, and object reference variables get null.

(Remember, null just means a remote control that isn't controlling / programmed to anything. A reference, but no actual object.)



**Instance variables  
always get a  
default value. If  
you don't explicitly  
assign a value  
to an instance  
variable, or you  
don't call a setter  
method, the  
instance variable  
still has a value!**

|                 |       |
|-----------------|-------|
| integers        | 0     |
| floating points | 0.0   |
| booleans        | false |
| references      | null  |

# The difference between instance and local variables

- 1 **Instance** variables are declared inside a class but not within a method.

```
class Horse {
 private double height = 15.2;
 private String breed;
 // more code...
}
```

- 2 **Local** variables are declared within a method.

```
class AddThing {
 int a;
 int b = 12;

 public int add() {
 int total = a + b;
 return total;
 }
}
```



### 3 Local variables MUST be initialized before use!

```
class Foo {
 public void go() {
 int x;
 int z = x + 3;
 }
}
```

Won't compile!! You can declare x without a value, but as soon as you try to USE it, the compiler freaks out.

File Edit Window Help Yikes

```
% javac Foo.java
```

```
Foo.java:4: variable x might
not have been initialized
```

```
 int z = x + 3;
1 error ^
```

---

**Local variables do NOT get a default value! The compiler complains if you try to use a local variable before the variable is initialized.**

# Outline

---

- 方法的概念
- 方法的实现
- 方法的特例
  - main方法
  - 递归方法
  - 方法重载
  - 函数副作用

# main方法

---

- Java程序的入口
- `public static void main (String[] args) {`
- 方法体
- `}`
- 当多个类中均包含main方法时，可以指定Java虚拟机从哪个类的main方法开始运行。其余的类中所包含的main方法将不被执行。



# 递归

---

- `// A simple example of recursion.`
- `class Factorial {`
- `// this is a recursive function`
- `int fact(int n) {`
- `int result;`
- `if(n==1) return 1;`
- `result = fact(n-1) * n;`
- `return result;`
- `}`
- `}`
- `class Recursion {`
- `public static void main(String args[]) {`
- `Factorial f = new Factorial();`
- `System.out.println("Factorial of 3 is " + f.fact(3));`
- `System.out.println("Factorial of 4 is " + f.fact(4));`
- `System.out.println("Factorial of 5 is " + f.fact(5));`
- `}`

该程序产生的输出如下所示：

**Factorial of 3 is 6**  
**Factorial of 4 is 24**  
**Factorial of 5 is 120**

# 方法重载 (Method Overloading)

---

- `void printInformation(int i) {...}`
- `void printInformation(char[ ] cArray) {...} /*  
与上面的方法参数类型不同 */`
- `void printInformation(char[ ] cArray, int i)  
{...}/* 与上面的方法参数个数不同 */`

# 非法的重载

---

- 参数标示符不同
  - `void printInformation(int i) {...}`
  - `void printInformation(int j) {...}` /\* 与上面的方法参数标识符不同 \*/
- 需要注意的是，在C++、Java、C#等语言中不同的返回类型不能用于区分方法。下面的方法重载是不允许的：
  - `int setBorrowedNum(int i) {...}` /\* 用布尔型表示操作结果 \*/
  - `boolean setBorrowedNum(int j) {...}` /\* 用整型表示操作结果 \*/

重载：不同的方法，  
恰好名字相同

# 编程范式 - Programming Paradigm

---

- A programming paradigm is a fundamental style of computer programming. There are four main paradigms: object-oriented, imperative, functional and declarative. [1]
- Their foundations are distinct models of computation: Turing machine for object-oriented and imperative programming, lambda calculus for functional programming, and first order logic for logic programming.

# Overview of the four main programming paradigm

---

- Imperative paradigm
  - First do this and next do that
- Functional paradigm
  - Evaluate an expression and use the resulting value for something
- Logic paradigm
  - Answer a question via search for a solution
- Object-Oriented paradigm
  - Send messages between objects to simulate the temporal evolution of a set of real world phenomena

# 示例

---

- $(1 + 2) * 3 - 4$
- 过程式编程
  - `var a = 1 + 2;`
  - `var b = a * 3;`
  - `var c = b - 4;`
- 函数式
  - `var result = subtract(multiply(add(1,2), 3), 4);`
- 面向对象式
  - `Integer i = 1;`
  - `Integer result;`
  - `result = i.add(2).multiply(3).subtract(4);`

Markup languages  
(only Datastructures)

More declarative  
Paradigms

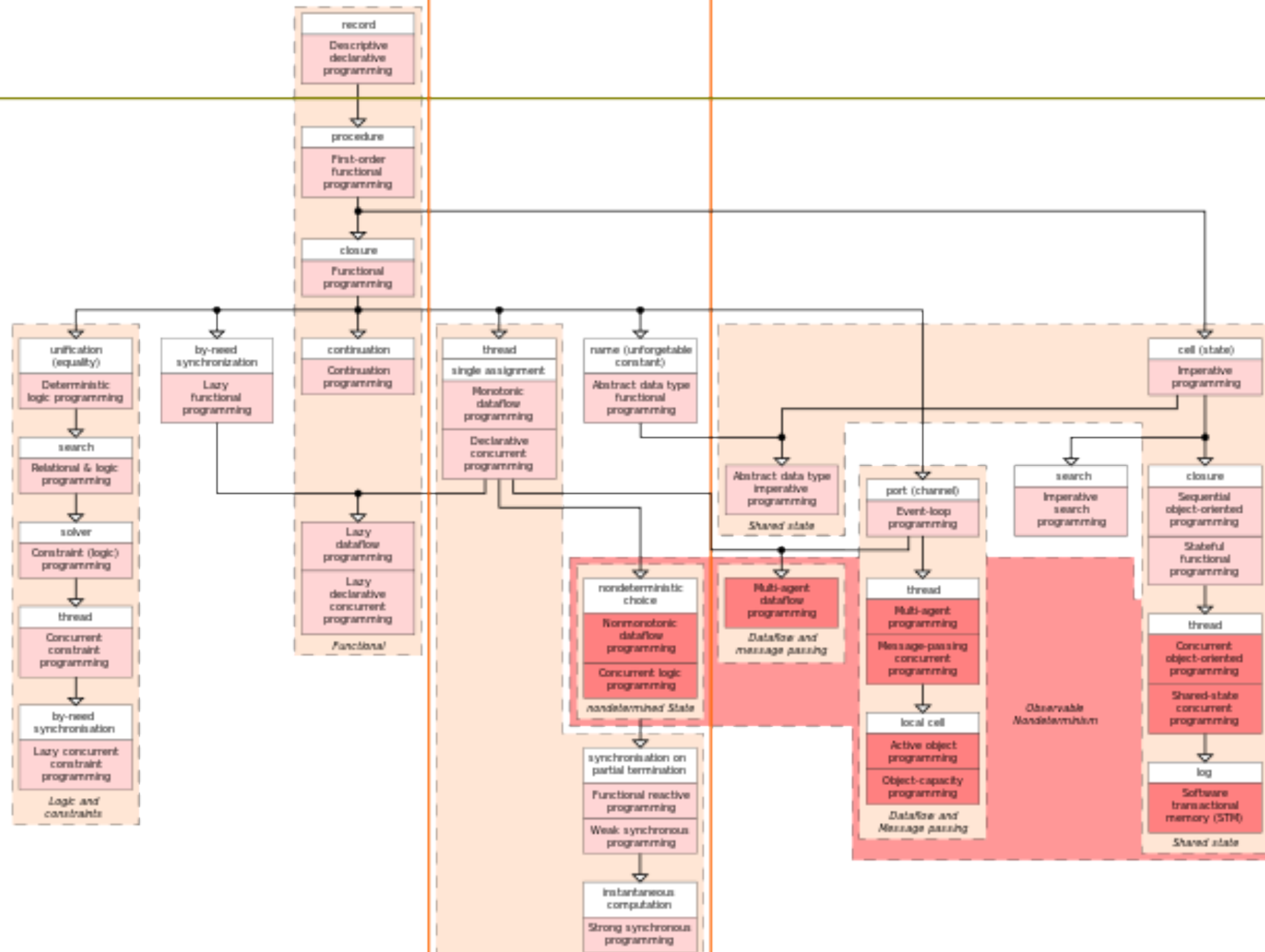
Unnamed state  
(sequential or concurrent)

Undeterministic  
state

Named  
state

More Imperative  
Paradigms

Turing complete  
Languages





# 函数副作用

---

- 在计算机科学中，函数副作用指当调用函数时，除了返回函数值之外，还对主调用函数产生附加的影响。例如修改全局变量（函数外的变量）或修改参数。
- 函数副作用会给程序设计带来不必要的麻烦，给程序带来十分难以查找的错误，并且降低程序的可读性。

# 相关概念

---

- 纯函数
  - 纯函数(Pure Function)是这样一种函数——输入输出数据流全是显式(Explicit)的。
  - 显式(Explicit)的意思是，函数与外界交换数据只有一个唯一渠道——参数和返回值；函数从函数外部接受的所有输入信息都通过参数传递到该函数内部；函数输出到函数外部的所有信息都通过返回值传递到该函数外部。
- 非纯函数
  - 如果一个函数通过隐式(Implicit)方式，从外界获取数据，或者向外部输出数据，那么，该函数就不是纯函数，叫作非纯函数(Impure Function)。
  - 隐式(Implicit)的意思是，函数通过参数和返回值以外的渠道，和外界进行数据交换。比如，读取全局变量，修改全局变量，都叫作以隐式的方式和外界进行数据交换；比如，利用I/O API（输入输出系统函数库）读取配置文件，或者输出到文件，打印到屏幕，都叫做隐式的方式和外界进行数据交换。
- 引用透明
  - 引用透明(Referential Transparent)的概念与函数的副作用相关，且受其影响。如果程序中两个相同值得表达式能在该程序的任何地方互相替换，而不影响程序的动作，那么该程序就具有引用透明性。它的优点是比非引用透明的语言的语义更容易理解，不那么晦涩。纯函数式语言没有变量，所以它们都具有引用透明性。

# 范例

---

- Java范例
  - `f(x)`
  - `{`
  - `return x + 1`
  - `}` // `f(x)`函数就是纯函数。
  - `a = 0`
  - `q(x) {`
  - `b = a`
  - `}` // `q(x)` 访问了函数外部的变量。`q(x)`是非纯函数。
  - `p(x){`
  - `print "hello"`
  - `}` // `p(x)`通过I/O API输出了一个字符串。`p(x)`是非纯函数。
  - `c(x) {`
  - `data = readConfig() // 读取配置文件`
  - `}` // `c(x)`通过I/O API读取了配置文件。`c(x)`是非纯函数。
- 纯函数内部有隐式(Implicit)的数据流，这种情况叫做副作用(Side Effect)。我们可以把副作用想象为潜规则。上述的I/O，外部变量等，都可以归为副作用。因此，纯函数的定义也可以写为，没有副作用的函数，叫做纯函数。
- I/O API可以看作是一种特殊的全局变量。文件、屏幕、数据库等输入输出结构可以看作是独立于运行环境之外的系统外全局变量，而不是应用程序自己定义的全局变量。

# 特殊的函数副作用

---

- 上述只讨论了一般的情况，还有一种特殊的情况，我们没有讨论。有些函数的参数是一种In/Out作用的参数，即函数可能改变参数里面的内容，把一些信息通过输入参数，夹带到外界。这种情况，严格来说，也是副作用。也是非纯函数。比如下面的函数。
- `process(context) {`
- `a = context.getInfo()`
- `result = calculate( a )`
- `context.setResult( result )`
- `}`
-

# 纯函数的优点

---

- 无状态。线程安全。不需要线程同步。
- 纯函数相互调用组装起来的函数，还是纯函数。
- 应用程序或者运行环境（Runtime）可以对纯函数的运算结果进行缓存，运算加快速度。

# 为什么函数式编程越来越流行

---

- 代码简洁，开发快速
- 接近自然语言，易于理解
- 更方便代码管理（单元测试和调试）
- 易于并发编程
- 代码的热升级

# 递归与函数式语言

---

- function reverse(string) {
- if(string.length == 0) {
- return string;
- } else {
- return reverse(string.substring(1, string.length)) +
- string.substring(0, 1);
- }
- }
- }
-