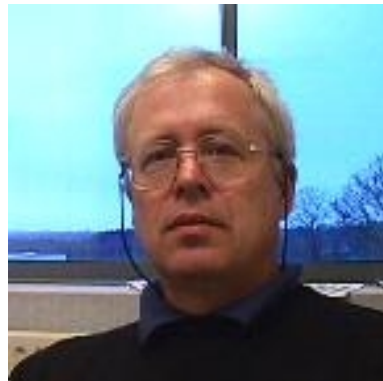




Programming with Java

Inheritance and Recursion

P. Bakowski



bako@ieee.org



Inheritance – base class

Now let us describe the class to represent a **bank account**. It should at least know the holder - his name, the account number, and the amount available.

For the operations on an account, we limit ourselves to **withdrawals**, **transfers** and **consultation**. In the case where a withdrawal is not possible we **throw an exception** of type **insufficientProvisionError** defined separately.

In addition to the default constructor, we also provide a constructor to build more complete account by determining its number, the holder and the original balance.

We also propose a method to transfer money from one account to another.



Inheritance – base class

```
public class BankAccount {
String nameHolder; char[] number; double balance;
    public BankAccount(){}
    public BankAccount(String hold, char[] num, double amount){
        this.nameHolder=hold; this.number=num; this.balance=amount;
    }
    public double getBalance(){return this.balance;}
    public void deposit(double amount)
    { this.balance=amount+this.balance;}
    public void withdraw(double amount) {
        System.out.println("Withdrawal");
        if(this.balance<amount){
            throw new insufficientProvisionError;
        } else { this.balance=this.balance-amount;}
    }
    public void transferTo(BankAccount c,double amount){
        c.withdraw(amount); this.deposit(amount); }
}
class insufficientProvisionError extends Error { }
```



Inheritance – base class

```
class Test1BankAccounts {  
public static void main(String[] args){  
BankAccount c1,c2,c3;  
String num1 = "123456789";  
String num2 = "145775544";  
String num3 = "A4545AA54";  
c1=new BankAccount("Paul",num1.toCharArray(),1000.00);  
c2=new BankAccount("Paul",num2.toCharArray(),2300.00);  
c3=new BankAccount("Henri",num3.toCharArray(),5000.00);  
c1.deposit(100.00);  
c2.transferTo(c1,1000.00);  
System.out.println(c2.getBalance());  
}  
}
```



Inheritance – data extension

Now suppose you want to take into consideration the possibility of having an **overdraft** on an account (withdrawal leads to a negative balance) or the possibility of associating payed deposits on account.

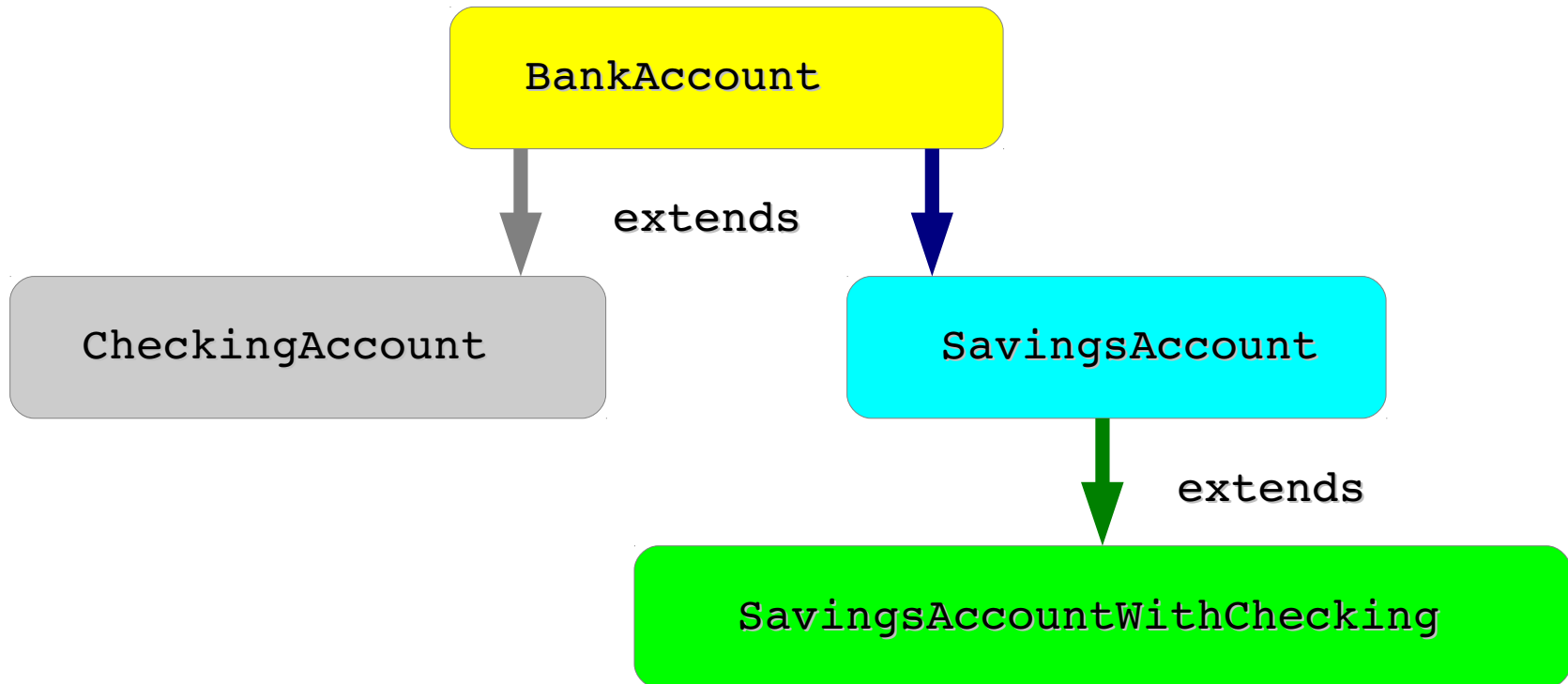
These cases are not included in the class we defined.

With object-oriented programming this is done naturally with the notion of **inheritance**. The principle is to define a new class from an existing class by "adding" the data to the class and **reusing implicitly** all data from the base class.

In the construction of a new class by extending an existing class, there are three kinds of methods (or variables):

- methods (and variables) that are **specific to the new class**, which is **extended**
- methods (and variables) which are **derived from the parent class**, this is **inheritance**
- methods (or variables, but this is more rare) that **redefine methods** (variables) that exist in the parent class, it is **masking**.

Inheritance – data extension





Extension of BankAccount (1)

```
public class CheckingAccount extends BankAccount{
    double MaxNegativeBalance;
    public void fixMaxNegativeBalance(double amount) {
        this.MaxNegativeBalance=amount ;
    }
    public CheckingAccount(String hold,char[] num,
        double amount,double MaxNegativeBalance) {
        super(hold,num,amount) ;
        this.MaxNegativeBalance = MaxNegativeBalance ;
    }
    public void withdraw(double amount) {
        if (this.balance-amount<MaxNegativeBalance) {
            throw new insufficientProvisionError();
        } else { balance= balance-amount;}}
    }
```

For the definition of the constructor we use the constructor of the superclass (by a call to **super**). Note that **super** is always called implicitly in the constructor or explicitly by a call to **super**.



Extension of BankAccount (2)

```
public class SavingsAccount extends BankAccount{
    double rate; double interest;
    public void fixRate(double rate) {
        this.rate=rate ;
    }
    public SavingsAccount(String hold,char[] num,
        double amount,double rate) {
        super(hold,num,amount) ;
        fixRate(rate); interest =0.0 ;
    }
    public void withdraw(double amount) {
        if (this.balance<amount) {
            throw new insufficientProvisionError();
        } else { balance = balance-amount;}
    }
    public void calculateInterest() {
        interest = interest +1 ; // in reality more complex
    }
}
```




Extension of BankAccount (3)

```
public class SavingsAccountWithChecking extends
SavingsAccount {
    double MaxNegativeBalance;
    public void SavingsAccountWithChecking(double amount) {
        this.MaxNegativeBalance=amount ; }
    public SavingsAccountWithChecking(String hold,
        char[] num, double amount, double rate) {
        super(hold,num,amount,rate);
        fixMaxNegativeBalance(amount);
    }
    public void withdraw(double amount) {
        if (this.balance-amount<MaxNegativeBalance) {
            throw new insufficientProvisionError();
        } else { balance= balance-amount;}
    }
}
```



Trans-typing

Rule related to object-oriented programming is that **every class is compatible with its subclasses** that is to say that if B is a subclass of A, then any variable of type A can be affected by a value of type B (an assignment statement or a parameter passing).

For example, if **c1** is a **BankAccount** and if **d1** is a **CheckingAccount** then the assignment **c1 = d1** is correct.

To contrary the assignment **d1 = c1** is incorrect.

The same instruction **c1.transferTo(d1, 100.0)** will be correct and there will be an implicit conversion of a value **d1** type **BankAccount** into **CheckingAccount** when calling the **transferTo** method .

We can also execute the :

```
d1.transferTo(c1, 100.0).
```



Dynamic links (binding)

In the previous examples, the methods called in the program are known at compile time.

Now suppose you declare a bank account `c` **without initializing it**:

```
BankAccount c ;  
String num = "AAA4AA54" ;
```

The typing rules of Java makes it possible to associate `c` by creating a **BankAccount** or **CheckingAccount**. In fact, every class is compatible with its subclasses, and therefore `c` is the class **BankAccount** class is compatible with **CheckingAccount**.

The two statements below are correct:

```
c=new BankAccount("Marie",num.toCharArray(),10000.00);  
c=new CheckingAccount("Marie",num.toCharArray(),10000.00,0.0);
```

So the instruction `c.withdraw(100.00)` will call the method of withdrawal accounts with overdraft (second case) or the withdrawal method of ordinary accounts (first case). This link (choice) between method call can be done during the execution of the program ; it is called **late binding (dynamic link)**.



Dynamic links (binding)

```
public class LateBinding{
public static void main(String[] args) {
BankAccount c ; String num = "AAA4AA54" ;
System.out.println("You wish to create checking account Y/N:");
char response; response = Keyboard.readChar();
if ((response!='Y')&&(response!='y')) {
System.out.println("Creation of ordinary account");
c = new BankAccount("Marie",num.toCharArray(),10000.00);}
else {
System.out.println("Creation of checking account");
System.out.println("Authorized negative balance:");
double max; max = Keyboard.readDouble();
c=new CheckingAccount("Marie",num.toCharArray(),10000.00,max);
}
System.out.println("Balance before withdrawal"+c.getBalance());
System.out.println("Withdrawal of 11000.00");
c.withdraw(11000.00);
System.out.println("New balance="+c.getBalance()); }
}
```



Dynamic links (binding)

```
pi@raspberrypi ~/java/MyClasses $ java LateBinding
You wish to create checking account Y/N:
Y
Creation of checking account
Authorized negative balance:
100
Balance before withdrawal : 10000.0
Withdrawal of 11000.00
Exception in thread "main" insufficientProvisionError
    at CheckingAccount.withdraw(CheckingAccount.java:13)
    at LateBinding.main(LateBinding.java:17)
```



Recursion

In some cases, the subproblem is an illustration of the initial problem, but for an "easier" case. Therefore, the solution of the problem is expressed in relation to itself! This is called **recursion**.

Example:

Calculating the **factorial** of a positive integer n ($n! = 1 * 2 * \dots * n$)

But, $n! = (1 * 2 * \dots * (n-1)) * n$, then $n! = (n-1)! * n$.

To calculate the value of $n!$, it suffices to know how to calculate $(n-1)!$ and then multiply the result by n . The sub-problem of the calculation of $(n-1)!$ is the same as the original problem, but for a "simpler" case, because $n-1 < n$.

```
int factorial(int n){
    int result;
    if(n==1) resul=1 ;
    else {
        int subresult = factorial(n-1); // recursive call
        result = subresult*n; }
    return result;
}
```



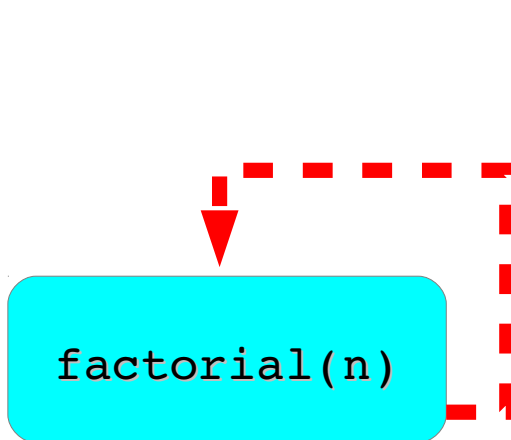
Recursion: test of factorial()

```
public class FactorialTest{
static int factorial(int n){
int result;
if (n<0) throw new BadParameter();
else if (n==0) result=1;
else {
int subresult = factorial(n-1);
result = subresult * n;}
return result;
}
public static void main(String[] args) {
System.out.println("Give a positive integer:");
int x = Keyboard.readInt();
System.out.println(x+"!="+factorial(x));
}
}
class BadParameter extends Error{}
```

Recursion : direct and indirect

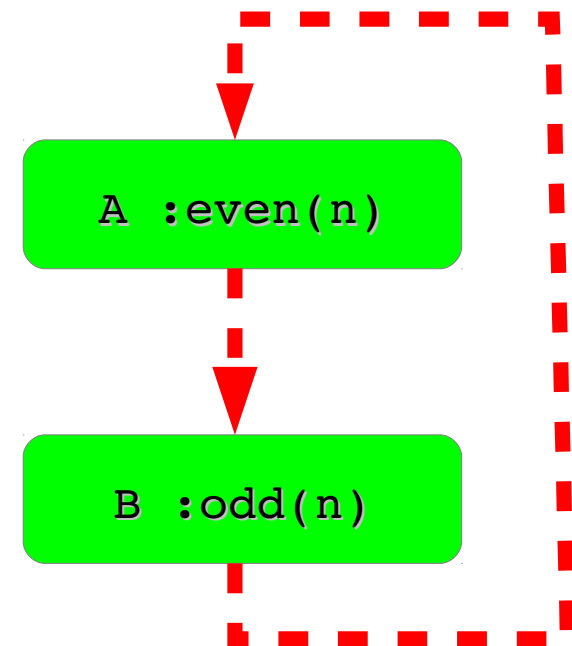
When a subroutine uses itself, as in the case of factorial, it is called **direct recursion**.

Sometimes, it is possible that subroutine A makes a call to subroutine B, which in turn calls the subroutine A. So the call from A reaches A again after being passed through B. This is called indirect recursion (in fact, following calls from A can pass through several other routines before coming back to A).



direct recursion

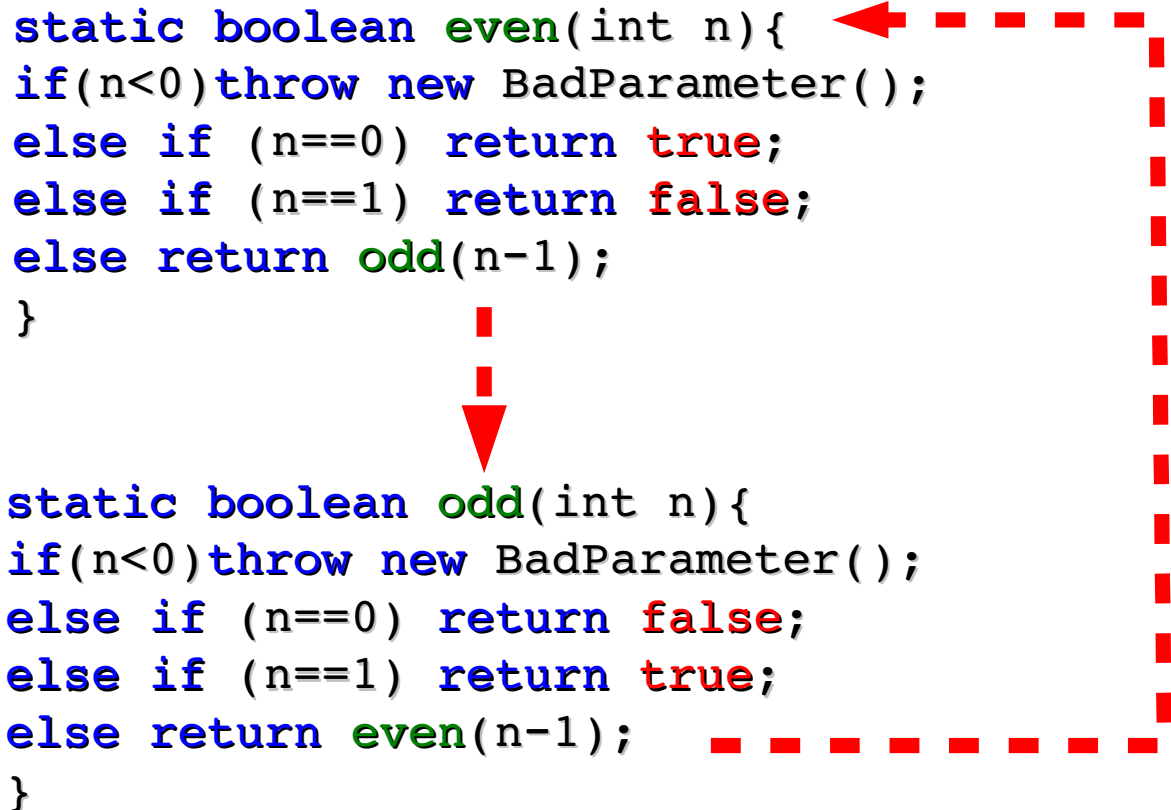
indirect
recursion





Indirect recursion

```
static boolean even(int n){  
    if(n<0) throw new BadParameter();  
    else if (n==0) return true;  
    else if (n==1) return false;  
    else return odd(n-1);  
}
```



```
static boolean odd(int n){  
    if(n<0) throw new BadParameter();  
    else if (n==0) return false;  
    else if (n==1) return true;  
    else return even(n-1);  
}
```



Indirect recursion : test

```
public class TestEvenOdd {
    static boolean even(int n){
        ..
    }
    static boolean odd(int n){
        ..
    }
    public static void main(String[] args) {
        System.out.println("Give a positive integer:");
        int x = Keyboard.readInt();
        if(pair(x))System.out.println("even number");
        else Keyboard.writeString("odd number");
    }
}
class BadParameter extends Error{}
// end of execution
```

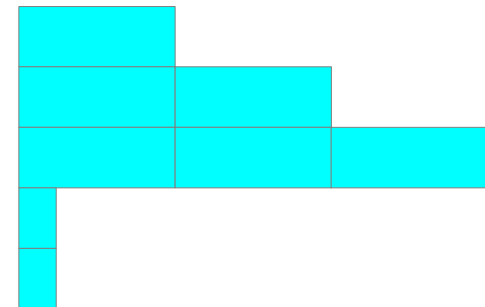
Memory model

How is it possible to call a subroutine while it is being run? How is it that the data managed by the calls to the same subroutine does not "mix" ? To answer these questions we must first understand the structure of the **memory model** maintained during the execution of Java subroutines. Take for example the following function:

```
boolean example(int x, double y) {  
    int[] t = new int[3];  
    char c ;  
    ...  
}
```

the stack

int
double
int[3]
char
boolean

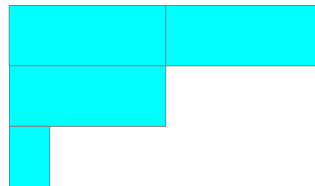


Memory model

```
public class Principal {  
    static boolean example(int x, double y ) { . . . }  
    public static void main(String[] args) {  
        double x ;  
        int n ;  
        boolean c ;  
        .....  
        c = example(n,x);  
        .....  
    }  
}
```



x
n
c



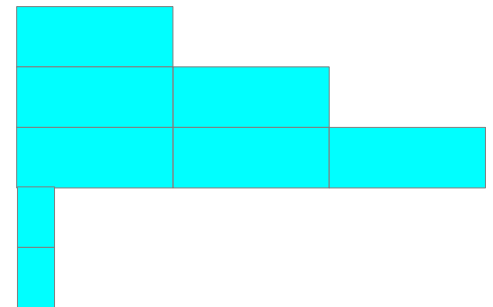
stacking



unstacking

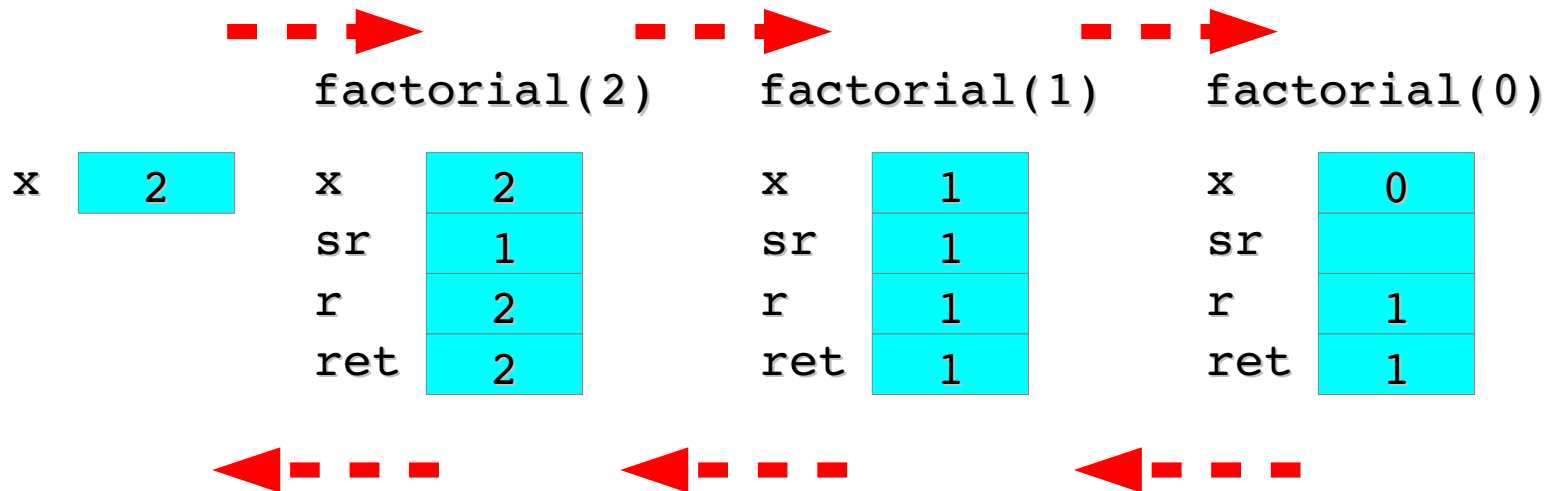


x
y
t
c
return



Execution progress of recursive calls

In the case of recursive programs, the stack is filled by calling the same subroutine (self-called). Take the case of calculating factorial (2): the main program calls factorial (2), which calls factorial (1), which calls factorial (0), which can calculate its return value without further recursive call. The value returned by factorial (0) factor allows (1) to compute its result, which allows factorial (2) to calculate his and return to the main program.





Recursion and Iteration

Depending on the problem, the solution is expressed more naturally by recursion or by iteration.

```
int nbOccurrences(char c, Strings ) {  
    int accu = 0 ;  
    for(int i=0;i<s.length();i ++)  
        if(s.charAt(i)==c) accu++;  
    return accu ;  
}
```

```
int nbOccurrences(char c, Strings ) {  
    if (s.length()== 0 ) return 0 ;  
    // stop condition : empty String  
    else {  
        int subresult=nbOccurrences(c,s.substring(1,s.length()-1));  
        if(s.charAt(i)==c) return 1 + subresult;  
        else return subresult;}  
}
```



Summary

- Inheritance - base class
- Data extension (**extends**)
- **BankAccount** and 2 extension classes
- Casting
- Dynamic linking
- Recursion
- Direct and indirect recursion
- Recursion and Iteration