Java - P36

正则表达式

character class	匹配
[abc]	simple class : 'a' or 'b' or 'c'
[^abc]	negation : any character except 'a', 'b', 'c'
[a-zA-Z]	inclusive range : 'a' through 'z' or 'A' through 'Z'
[a-d[m-p]]	union : 'a' through 'd' or 'm' through 'p', 等价于[a-dm-p]
[a-z&&[def]]	intersection : 'd' or 'e' or 'f'
[a-z&&[^bc]]	substruction : 'a' through 'z', except 'b', 'c', 等价于[ad-z]
[a-z&&[^m-p]]	substruction : 'a' through 'z', and not 'm' through 'p', 等价于[a-lq-z]

predefined character class	匹配
.	any character, may or may not match line terminator
\d	digit : [0-9]
\D	not digit : [^0-9]
\h	horizontal whitespace character : [\t\xA0\u1680\u180e\u2000-\u200a\u202f\u205f\u3000]
\H	non-horizontal whitespace character, 等价于[^\h]
\s	whitespace character : [\t\n\x0B\f\r]
\S	non-whitespace character : [^\s]
\v	vertical whitespace character : [\n\x0B\f\r\x85\u2028\u2029]
\V	non-vertical whitespace character : [^\v]
\w	word character : [a-zA-Z_0-9]
\W	non-word character : [^\w]

boundary matcher	匹配
^	beginning of line
$	end of line
\b	word boundary
\B	non-word boundary
\A	beginning of input
\G	end of previous match
\Z	end of input but for final terminator (if any)
\z	end of input

Java - P37

正则表达式

POSIX char class	匹配 (US-ASCII only)
\p{Lower}	lowercase alphabetic character : [a-z]
\p{Upper}	uppercase alphabetic character : [A-Z]
\p{ASCII}	ASCII : [\x00-\x7F]
\p{Alpha}	alphabetic character : [\p{Lower}\p{Upper}]
\p{Digit}	decimal digit : [0-9]
\p{Alnum}	alphanumeric character : [\p{Alpha}\p{Digit}]
\p{Punct}	punctuation : one of !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
\p{Graph}	visible character : [\p{Alnum}\p{Punct}]
\p{Print}	printable character : [\p{Graph}\x20]
\p{Blank}	space or tab : [\t]
\p{Cntrl}	control character : [\x00-\x1F\x7F]
\p{XDigit}	hexadecimal digit : [0-9a-fA-F]
\p{Space}	whitespace character : [\t\n\x0B\f\r]

java.lang.Character class	匹配 (simple java character type)
\p{javaLowerCase}	等价于 java.lang.Character.isLowerCase()
\p{javaUpperCase}	等价于 java.lang.Character.isUpperCase()
\p{javaWhitespace}	等价于 java.lang.Character.isWhitespace()
\p{javaMirrored}	等价于 java.lang.Character.isMirrored()

Unicode script / block / category / binary property	匹配
\p{IsLatin}	Latin script character (script)
\p{InGreek}	character in Greek block (block)
\p{Lu}	uppercase letter (category)
\p{IsAlphabetic}	alphabetic character (binary property)
\p{Sc}	curreny symbol
\P{InGreek}	any character except in Greek block (negation)
[\p{L}&&[^\p{Lu}]]	any letter except uppercase letter (subtraction)

linebreak matcher	匹配
\R	any Unicode linebreak sequence
	等价于 \u000D\u000A|[\u000A\u000B\u000C\u000D\u0085\u2028\u2029]

Java - P38

正则表达式

greedy quantifier	匹配 (match as much as possible), backtrack
X?	once or not at all X
X*	zero or more times X
X+	one or more times X
X{n}	exactly n times X
X{n,}	at least n times X
X{n,m}	at least n and at most m times X

"non-greedy"

reluctant quantifier	匹配 (match as little as possible), backtrack
X??	once or not at all X
X*?	zero or more times X
X+?	one or more times X
X{n}?	exactly n times X
X{n,}?	at least n times X
X{n,m}?	at least n and at most m times X

possessive quantifier	匹配 no backtrack, 与 greed quantifier 类似
X?+	once or not at all X
X*+	zero or more times X
X++	one or more times X
X{n}+	exactly n times X
X{n,}+	at least n times X
X{n,m}+	at least n and at most m times X

greedy

如 ".*foo" : "xfooxxxxfoo"

.* match entire string and no character left

backtrack until match "foo", 即 ".*|foo" : "xfooxxxx|foo"

reluctant

".*?foo" : "xfoo xxxxfoo"

.* match nothing first and leave entire string unmatched

"backtrack" until match "foo", 即 ".*?foo" : "x|foo|xxxx|foo"

possesive

".*+foo" : "xfooxxxxfoo", no match

.* match entire string and no character left unmatched

no backtrack, then fail

Java - P39

正则表达式

logical operator	匹配
XY	X followed by Y
X|Y	either X or Y
(X)	X as capturing group

back reference	匹配
\n	n-th capturing group matched
\k<name>	named capturing group "named" matched

quotation	匹配
\	nothing but quote following character
\Q	nothing but quote all character until \E
\E	nothing but end quoting started by \Q

special construct	匹配 (named-capturing and non-capturing)
(?<name>X)	X as named-capturing group
(?:X)	X as non-capturing group
(?idmsux(U-idmsux(U)	nothing but turn match flag i d m s u x on (- off)
(?idmsux-idmsux:X)	X as non-capturing group with given flag i d m s u x on (- off)
(?=X)	X via zero-width positive lookahead
(?!X)	X via zero-width negative lookahead
(?<=X)	X via zero-width positive lookbehind
(?<!X)	X via zero-width negative lookbehind
(?>X)	X as independent non-capturing group

capturing group number, numbered by counting opening parentheses from left to right

如对于 ((A)(B(C)))

1: ((A)(B(C))), 2: (A), 3: (B(C)), 4: (C)

可以通过调用matcher对象的groupCount方法查看表达式的分组数

group zero always stand for entire expression

即表示为 (group(0))

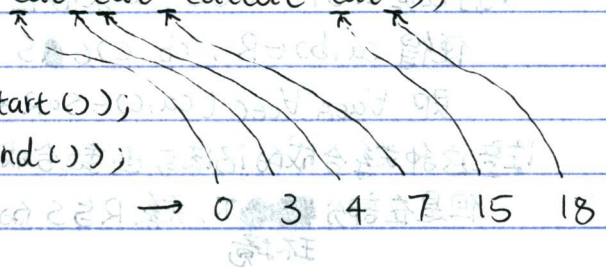
注意该组不包括在groupCount的返回值中

Java - P40

正则表达式

named-capturing group
capturing group can be assigned "name"
and be back-referenced later by "name"
during match each subsequence of input sequence match such group saved
may be used later in expression, via back reference
may be retrieved from matcher once match operation complete
group name composed of following character, first character must be letter
uppercase letter : A-Z (\u0041 - \u005a)
lowercase letter : a-z (\u0061 - \u007a)
digit : 0-9 (\u0030 - \u0039)
still numbered as capturing group number
matched
captured input associated with group always subsequence group most recently
if group evaluated second time because of quantification
previously-captured value retained if second evaluation fail

在Java中提供3 Matcher类进行正则表达式操作
public int start() : return start index of previous match
public int end() ; return offset after last character matched
public int start(int group)
public int end(int group)
return start index / offset after last character of subsequence captured
by given group during previous match operation
public int start(String name)
public int end(String name)
return start index / offset after last character of subsequence captured
by given named-capturing group during previous match operation
索引方法提供了有用的索引值，精确地表明输入字符串中匹配的位置
如 Pattern p = Pattern.compile("\\b cat \\b");
Matcher m = p.matcher("cat cat catcat cat");
while (m.find()){
System.out.println(m.start());
System.out.println(m.end());
}



Java-P41

正则表达式

在Java中，Matcher类中提供3个研究方法

用于检查一个字符串输入并返回一个布尔值，表示是否匹配模式

public boolean find()

attempt to find next subsequence of input sequence match the pattern start at the beginning of this matcher's region or at first character not matched by previous match

if previous invocation of method successful and matcher not reset

if match succeed, then more information obtained via start()/end()/group()

public boolean find(int start)

reset matcher and attempt to find next subsequence of input sequence match the pattern starting at specified index

public boolean lookingAt()

attempt to match input sequence, starting at beginning of region, against pattern

like matches() method, always start at beginning of region

unlike matches() method, not require entire region matched

public boolean matches()

attempt to match entire region against the pattern

if match succeed, then more information obtained via start()/end()/group()

即 matches方法和 lookingAt 方法都用于尝试匹配输入序列模式

差别在于 matches 要求整个序列匹配，而 lookingAt 不要求

相同在于都需要从第一个字符开始匹配

如

```
String reg1 = "foo";
String input1 = "fooxxxx";
String input2 = "xxxxfoo";
Pattern p1 = Pattern.compile(reg1);
Matcher m1 = p1.matcher(input1);
Matcher m2 = p1.matcher(input2);
m1.matches(); ⟶ false
m1.lookingAt(); ⟶ true   "foo xxxx"
m2.lookingAt(); ⟶ false
```

Java - P42

正则表达式

在Java中，Matcher类提供3替换方法用于替换输入字符串中的文本

public String replaceAll(String replacement)

public String replaceFirst(String replacement)

replace every / first subsequence of input sequence match pattern with given replacement string

first reset matcher, then scan input sequence looking for match of pattern

character not part of any match appended directly to result string

each match replaced in result by replacement string

replacement string may contain reference to captured subsequence as in appendReplacement method

backslash(\) and dollar sign($) in replacement string may cause result to be different than treated as literal replacement string

($) may be treated as reference to captured subsequence as described above

(\) used to escape literal character in replacement string

invoking this method change matcher's state

matcher should be first reset before used in further matching operation

```
String reg = "a*b";
String input = "aabfooaaabfooabfoob";
String replace = "-";
Pattern p = Pattern.compile(reg);
Matcher m = p.matcher(input);
m.replaceAll(replace) ⟶ "-foo-foo-foo-" : "aabfooaaabfooabfoob"
m.replaceFirst(replace) ⟶ "-fooaaabfooabfoob"
                        : "aabfooaaabfooabfoob"
```

public static String quoteReplacement(String s)

return literal replacement String for specified String

produce String work as literal replacement s in the appendReplacement method of Matcher class

String produced will match sequence of character in s treated as literal sequence

backslash(\) and dollar sign($) given no special meaning

Java - P43

正则表达式

在Java中，Matcher类提供了替换方法用于替换输入字符串中的文本

public Matcher appendReplacement(StringBuffer sb, String replacement)

implement non-terminal append-and-replace step

perform following action:

read character from input sequence starting at append position

and append character to given string buffer

stop after reading last character preceding previous match

character at index start() - 1

append given replacement string to string buffer to end()

set append position of this matcher to index of last character matched + 1

replacement may contain reference to subsequence captured during previous match

each occurence of ${name} or $g

will be replaced by result of evaluating corresponding group(name) / group(g)

for $g : first number after $ always treated as part of group reference

subsequent number incorporated into g if form legal reference

only numeral '0' through '9' considered as potential component

backslash(\) and dollar sign($) in replacement string may cause different result

($) may be treated as reference to captured subsequence as described above

(\) used to escape literal character in replacement string

intended to be used in loop together with appendTail() and find()

public StringBuffer appendTail(StringBuffer sb)

implement terminal append-and-replace step

intended to be invoked after one or more invocation of appendReplacement()

in order to copy remainder of input sequence

eg

```
Pattern p = Pattern.compile("\\b cat \\b");
Matcher m = p.matcher("one cat two cats in the yard");
StringBuffer sb = new StringBuffer();
while (m.find()){
    m.appendReplacement(sb, "dog"); }
m.appendTail(sb);
System.out.println(sb.toString());
```

⟶ "one dog two cats in the yard"

Java-P44

正则表达式

在Java中，Matcher类提供了操作group的方法

public String group(), return input subsequence matched by previous match
for Matcher m, with input sequence s
m.group() equivalent to s.substring(m.start(), m.end())
for pattern match empty string, return empty string

public String group(int g)
return input subsequence captured by given group during previous match operation
for Matcher m with input sequence s, and group index g
m.group(g) equivalent to s.substring(m.start(g), m.end(g))
capturing group indexed from left to right, starting at one
group zero denote entire pattern, m.group(0) equivalent to m.group()
if match successful but group specified failed to match any part of input sequence
return null
for pattern match empty string, return empty string

public String group(String name)
return input subsequence captured by given named-capturing group during previous match operation
if match successful but group specified failed to match any part of input sequence
return null
for pattern match empty string, return empty string

IllegalStateException: if no match yet attempted, or if previous match operation failed
IndexOutOfBoundsException: if no capturing group in pattern with given index
IllegalArgumentException: no capturing group in pattern with given name

public int groupCount()
return number of capturing group in this matcher's pattern
group zero denote entire pattern by convention
not included in count
any non-negative integer smaller than or equal to value returned
guaranteed to be valid group index for this matcher

Java - P45

正则表达式

在Java中，提供了非强制异常类(unchecked exception)指示正则表达式中的语法错误

public class PatternSyntaxException extends IllegalArgumentException
thrown to indicate syntax error in regular-expression pattern

public PatternSyntaxException(String desc, String regex, int index)
desc : description of error
regex : erroneous pattern
index : approximate index in pattern of error
or -1 if index not known

public int getIndex() : retrieve error index
public String getDescription() : retrieve description of error
public String getPattern() : retrieve erroneous regular-expression pattern
public String getMessage() :
return multi-line string containing description of syntax error
and index, erroneous regular-expression pattern
and visual indication of error index within pattern

方法

在Java中，方法是解决一类问题的步骤的有序组合
包含于类或对象中，在程序中被创建，在其他地方被引用
使程序变得简短而清晰明析，提高代码的重用性
可以提高程序开发的效率，有利于程序维护

method header [[<modifier>]* <return value type> <method name> ([<parameter type> <parameter name>]*)
method body [{ [<statement>]* }

modifier (修饰符)：可选地定义该方法的访问类型，告诉编译器如何调用方法
return value type (返回值类型)：方法返回值的数据类型
当类型为void时，方法执行定义的操作但没有返回值
method name (方法名)：方法的实际名称，与参数列表共同构成方法签名
通常其第一个单词以小写开头，之后单词以大写开头，不使用连接符
下划线可能出现在JUnit测试方法中用于分隔名称的逻辑组件
如 test<MethodUnderTest>_<state>
参数列表为方法的参数类型、顺序和参数个数

Java - P46

方法

在Java中，支持两种方法调用的方式，根据方法是否返回值来选择

当方法通过return关键字返回一个值时，

方法调用在程序中通常被视为一个右值或表达式的一部分

当方法的返回值类型为void，此时方法调用不返回值

方法调用一定是一条语句（statement）

当程序调用方法时，程序的控制权交给被调用的方法

而当被调用的方法的返回语句return执行或达到方法体的闭括号时

控制权由被调用方法交还给程序

在Java中，提供了函数重载（overload）

即对于同一个类的两个方法，拥有相同的方法名，但是拥有不同的参数列表

Java编译器根据方法签名判断应该调用哪个方法

执行密切相关的任务的方法应使用相同的名字

方法重载可以让程序变得更加清晰易读

注意，Java与C++一样，重载方法必须拥有不同的参数列表

而不能仅仅根据不同的修饰符或者返回类型来重载方法

在Java中，变量作用域（scope）指程序中该变量可被引用的部分

在方法内部定义的变量通常称为局部变量，必须声明才可以使用

局部变量的作用域从其声明开始，直到包含该变量的程序块终止

传入方法的参数的作用域可涵盖整个方法，

即传入的参数也视为局部变量

对于for循环，在其初始化时声明的变量，其作用域为整个循环

而循环体内部声明的变量，其作用域仅限于循环体

即从其声明开始，到该次循环迭代结束

在不同的嵌套块中声明有相同变量名的局部变量

但在相互嵌套的程序块中不能声明相同变量名的局部变量

在Java中，可以通过命令行参数向main()函数传递信息

```
如 public class CommandLine {
    public static void main(String args[]){
      for(int i=0; i<args.length; i++){
        System.out.println("args["+i+"]: "+args[i]);
    }}}
```

```
$ javac CommandLine.java
$ java CommandLine command line 11
→ args[0]: command
  args[1]: line
  args[2]: 11
```

Java - P47

可变参数

在Java中，支持向方法传递同类型的可变参数

基本语法结构为 <type name> ... <parameter name>

即在指定参数类型后加一个省略号(...)

注意在一个方法中只能定义一个可变参数，且必须是方法声明的最后一个参数

即方法的任何普通参数必须在可变参数前声明

```
eg public static void testVar(int a, int b, int ... ints){
        System.out.println("a: " + a);
        System.out.println("b: " + b);
        for(int i=0; i< ints.length; i++){
            System.out.println("ints[" + i + "]: " + ints[i]);
        }
    }
```

则 testVar(1,2,3,4) ⟶
- a: 1
- b: 2
- ints[0]: 3
- ints[1]: 4

在Java中，允许定义finalize()方法用于清除回收对象

在对象被垃圾回收器析构(回收)之前调用

finalize()方法中必须指定在对象销毁时应执行的操作

基本定义格式为 protected void finalize(){
...}

其中关键字protected为访问限定符，用确保finalize()不被该类以外的代码调用

Java的内存回收可以人通过JVM自动完成，也可以手动使用

```
eg public class Something extends Object{
        private int id;
        public Something(int id_){this.id = id_;}
        protected void finalize() throws java.lang.Throwable{
            super.finalize();
            System.out.println("id " + this.id + " destructed");
        }}
    Something st1 = new Something(1);
    Something st2 = new Something(2);
    st2 = null;
    System.gc(); ⟶ id 2 destructed
```

Java - P48

继承

在Java中，继承允许创建分等级层次的类

子类继承父类的特征和行为，使得子类对象（实例）具有父类的实例域和方法

或子类从父类继承方法，使得子类具有父类相同的行为

在Java中，通过extends关键字声明一个类从另一个类继承而来

如 class ParentClass { ... }

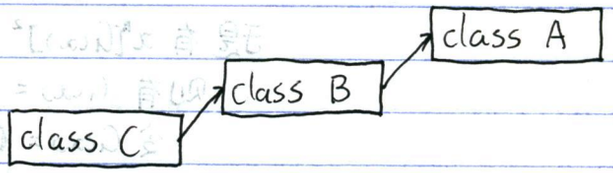
class ChildClass extends ParentClass { ... }

Java支持多重继承和公共基类

如 public class A { ... }

public class B extends A { ... }

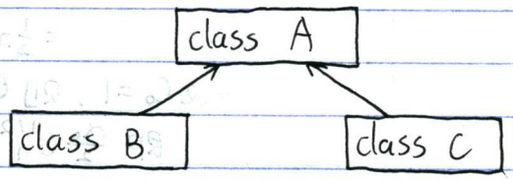
public class C extends B { ... }



public class A { ... }

public class B extends A { ... }

public class C extends A { ... }

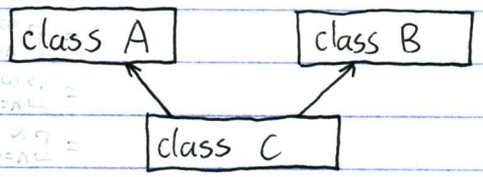


但是Java不支持多继承

如 public class A { ... }

public class B { ... }

public class C extends A, B { ... }



在Java中，所有的类都是继承于类java.lang.Object

Object类保存于java.lang包中，无需import即可使用

如果没有显式地通过关键字声明继承的基类，则类默认地继承自Object

Java的继承是单继承，即一个子类只能继承一个父类

子类拥有父类的非private的属性和方法

子类可以拥有自己的属性和方法，即子类对父类进行扩展

子类可以用自己的方式实现父类的方法

继承的缺点是提高了类之间的耦合性

高耦合度会造成代码之间的联系更紧密，降低代码的独立性

在Java中，可以使用implements关键字变相地实现类的多继承

即类可以同时继承多个接口

Java - P49

继承

在Java中，提供了this关键字和super关键字在类的定义中使用

this关键字：指向所属类自身的引用

super关键字：指向所属类的父类，用于对父类成员的访问

在Java中，final关键字用于修饰类/方法/变量

final关键字声明的类不能被其他类继承，即最终类

如 final class <class name> {...}

final关键字声明的方法不能被该类的派生类重写

如 public/private/default/protected final <return type> <method name>(...) {...}

final关键字声明的变量不能被修改，

声明为final的类的方法自动地声明为final，但实例变量不会自动声明为final

在Java中，子类不继承父类的构造器，而仅仅是隐式/显式地调用

如果父类包含无参数构造器，可能是默认的也可能是显式定义的

则子类的构造器会在java文件编译成class文件时

在子类构造器的首行默认地自动插入super()语句，用于调用父类的无参构造器

即在执行子类的构造函数前必定先执行父类的构造函数

于是如果父类定义了有参数构造器，则不再包含默认的无参构造器

则在生成子类实例时，会因为super()语句执行失败而报错

在子类的构造函数可以显式地调用父类的构造器

可以是无参构造器 super()，也可以是有参数构造器 super(...)

但是父类构造函数的调用必须在子类构造器内的首句，否则抛出错误

```
eg class Base {
       Base() { System.out.println("Base()"); }
       Base(int a) { System.out.println("Base(" + a + ")"); }
   }
   class Derived extends Base {
       Derived() { super(11);
                   System.out.println("Derived()"); }
       Derived(int a) { System.out.println("Derived(" + a + ")"); }
   }
```

`Derived d1 = new Derived();`	`Derived d2 = new Derived(11);`
→ Base(11)	→ Base()
Derived()	Derived(11)

Java - P50

继承　在Java中，存在向上转型（upcasting）和向下转型（down casting）

父类引用可以指向子类对象，而子类引用不可以指向父类对象

如 Base b = new Derived();

此时 b instanceof Base → true

Base b = new Base();

Derived d = (Derived) b;

由于子类引用不能指向父类对象，从而抛出错误

Derived d = new Base();

incompatible type: Base cannot be converted to Derived

将子类对象直接赋值给父类引用称为向上转型（upcasting）

而向上转型不用强制转换

将指向子类对象的父类引用赋值给子类引用称为向下转型（down casting）

向下转型需要使用强制转换

如 Base b = new Derived();

Derived d = (Derived) b;

重写　（override），在Java中指子类对父类的允许访问的方法实现过程进行重新编写

重写的过程中方法的返回值与参数列表均不能改变

优点是子类可以根据需要定义自身的行为，即根据需求实现父类的方法

注意 重写的方法不能抛出新的检查异常或比被重写的方法申明更加宽泛的异常

如 当父类方法申明了一个检查异常 IOException

则在子类重写该方法时不能抛出 Exception 异常

由于 Exception 异常是 IOException 的父类，而重写方法只能抛出 IOException 的子类异常

```
如 class Base {
       public void method() { System.out.println(11); }
   }
   class Derived extends Base {
       public void method() { System.out.println("overrided"); }
   }
   Base b1 = new Base(); b1.method() ⟶ 11
   Base b2 = new Derived(); b2.method() ⟶ overrided
   Derived d = new Derived(); d.method() ⟶ overrided
```