第05天 API

今日内容介绍

* Object类 & System类
* 日期相关类
* 包装类&正则表达式

# Object类 & System类

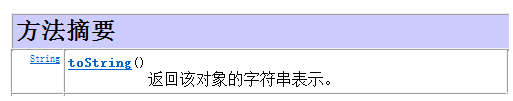
## Object类

### 概述

Object类是Java语言中的根类，即所有类的父类。它中描述的所有方法子类都可以使用。所有类在创建对象的时候，最终找的父类就是Object。

在Object类众多方法中，我们先学习equals方法与toString方法，其他方法后面课程中会陆续学到。

### toString()方法



由于toString方法返回的结果是内存地址，而在开发中，经常需要按照对象的属性得到相应的字符串表现形式，因此也需要重写它。

#### 案例代码一:

**package** com.itheima\_01;

/\*

\* String toString() : 返回该对象的字符串表示

\* return getClass().getName() + "@" + Integer.toHexString(hashCode());

\* getClass():返回一个字节码对象

\* Integer.toHexString():返回指定参数的十六进制字符串形式

\* hashCode()：返回该对象的哈希码值（内部地址）

\*

\*

\*

\* boolean equals(Object obj)

\*

\*/

**public** **class** ObjectDemo {

**public** **static** **void** main(String[] args) {

Student s = **new** Student();

s.name = "zhangsan";

s.age = 18;

System.*out*.println(s.toString());//com.itheima\_01.Student@737951b0

System.*out*.println(s);//说明我们输出一个对象就是默认输出这个对象的toString()方法

}

}

**class** Student **extends** Object {

String name;

**int** age;

/\*

public String toString() {

return name + "@" + age;

}

\*/

@Override

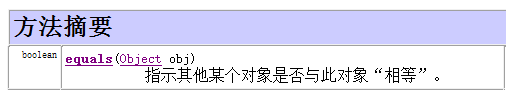
**public** String toString() {

**return** "Student [name=" + name + ", age=" + age + "]";

}

}

### equals()方法



equals方法，用于比较两个对象是否相同，它其实就是使用两个对象的内存地址在比较。Object类中的equals方法内部使用的就是==比较运算符。

在开发中要比较两个对象是否相同，经常会根据对象中的属性值进行比较，也就是在开发经常需要子类重写equals方法根据对象的属性值进行比较。

#### 案例代码二:

**package** com.itheima\_01;

**import** java.util.ArrayList;

/\*

\* boolean equals(Object obj)

\* 使用==来比较两个对象是否相等，则比较地址值是否相等

\*/

**public** **class** ObjectDemo2 {

**public** **static** **void** main(String[] args) {

Person p = **new** Person("zhangsan",18);

Person p2 = **new** Person("zhangsan",19);

//boolean flag = p.equals(p2);

**boolean** flag = p.equals(**new** ArrayList());

System.*out*.println(flag);

}

}

**class** Person {

String name;

**int** age;

**public** Person(String name,**int** age) {

**this**.name = name;

**this**.age = age;

}

@Override

**public** **boolean** equals(Object obj) {

//提高效率

**if** (**this** == obj)

**return** **true**;

**if** (obj == **null**)

**return** **false**;

//提高健壮性

**if** (getClass() != obj.getClass())

**return** **false**;

//向下转型

Person other = (Person) obj;

**if** (age != other.age)

**return** **false**;

**if** (name == **null**) {

**if** (other.name != **null**)

**return** **false**;

} **else** **if** (!name.equals(other.name))

**return** **false**;

**return** **true**;

}

/\*@Override

public boolean equals(Object o) {

//提高效率 当前对象和传递进来的对象地址值一样，则不用比较成员

if(this == o) {

return true;

}

//提高代码的健壮性

if(this.getClass() != o.getClass()) {

return false;

}

//向下转型

Person other = (Person) o;

if(!this.name.equals(other.name)) {

return false;

}

if(this.age != other.age) {

return false;

}

return true;

}\*/

}

## System类

System 类包含一些有用的类字段和方法。它不能被实例化。

### 成员方法

static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length) :

从src源数组的srcPos索引开始,复制length个元素

从destPost位置开始将这些元素放至到dest数组中

static long currentTimeMillis()

返回以毫秒为单位的当前时间

static void exit(int status)

终止当前正在运行的 Java 虚拟机

static void gc()

运行垃圾回收器

### 案例代码三:

**package** com.itheima\_02;

/\*

\* System:包含一些有用的类字段和方法。它不能被实例化

\* static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)

\* static long currentTimeMillis()

\* static void exit(int status)

static void gc()

\*

\*/

**public** **class** SystemDemo {

**public** **static** **void** main(String[] args) {

//method();

//method2();

//method3();

//static void gc()

//Demo d = new Demo();

**new** Demo();

System.*gc*();

}

**private** **static** **void** method3() {

//static void exit(int status) :终止虚拟机

**for** (**int** i = 0; i < 100000; i++) {

System.*out*.println(i);

**if**(i == 100) {

System.*exit*(0);

}

}

}

**private** **static** **void** method2() {

/\*

\* static long currentTimeMillis() :以毫秒值返回当前系统时间

\* 这个毫秒的时间是相对时间，相对于1970-1-1 00:00:00 ： 0

\* 1970-1-1 00:00:01 : 1000

\* 1970-1-1 00:01:00: 1000 \* 60

\* 1970-1-1 01:00:00: 1000 \* 60 \* 60

\* 1000毫秒 = 1秒

\*

\*/

//System.out.println(System.currentTimeMillis());

**long** start = System.*currentTimeMillis*();

**for** (**int** i = 0; i < 100000; i++) {

System.*out*.println(i);

}

**long** end = System.*currentTimeMillis*();

System.*out*.println(end - start);

}

**private** **static** **void** method() {

/\*

\* static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)

\* 复制数组

\* 参数1：源数组

\* 参数2：源数组的起始索引位置

\* 参数3：目标数组

\* 参数4：目标数组的起始索引位置

\* 参数5：指定接受的元素个数

\*/

**int**[] src = {1,2,3,4,5};

**int**[] dest = **new** **int**[5];

System.*arraycopy*(src, 2, dest, 4, 3);

**for** (**int** i = 0; i < dest.length; i++) {

System.*out*.print(dest[i]);

}

}

}

**class** Demo {

@Override

**protected** **void** finalize() **throws** Throwable {

System.*out*.println("我被回收了");

}

}

# 日期相关类

## Date类

Date: 表示特定的瞬间，精确到毫秒，他可以通过方法来设定自己所表示的时间，可以表示任意的时间

### Date类的构造方法

Date() ：创建的是一个表示当前系统时间的Date对象

Date(long date) ：根据"指定时间"创建Date对象

### 案例代码四:

**package** com.itheima\_03;

**import** java.util.Date;

/\*

\* Date: 表示特定的瞬间，精确到毫秒，他可以通过方法来设定自己所表示的时间，可以表示任意的时间

\* System.currentTimeMillis():返回的是当前系统时间，1970-1-1至今的毫秒数

\*

\* 构造方法：

\* Date() ：创建的是一个表示当前系统时间的Date对象

Date(long date) ：根据"指定时间"创建Date对象

\*/

**public** **class** DateDemo {

**public** **static** **void** main(String[] args) {

//Date()

//Date d = new Date();

//System.out.println(d);//Thu Aug 26 14:17:28 CST 2049

//System.out.println(d.toLocaleString());

//Date(long date)

Date d2 = **new** Date(1000 \* 60 \* 60 \* 24);//时区 有时差

System.*out*.println(d2.~~toLocaleString~~());

}

}

### Date类常用方法

void setTime(long time)

long getTime()

### 案例代码五:

**package** com.itheima\_03;

**import** java.util.Date;

/\*

\* Date的常用用方法

毫秒值 --- Date

设置

返回值是void，参数long

void setTime(long time)

Date(long date)

Date --- 毫秒值

获取

返回long，无参数

long getTime()

\*/

**public** **class** DateDemo2 {

**public** **static** **void** main(String[] args) {

Date d = **new** Date();//默认当前系统时间

//d.setTime(1000 \* 60 \* 60 \* 24 \* 2);

System.*out*.println(d.~~toLocaleString~~());

System.*out*.println(d.getTime());//172800000

d.setTime(172800000L);

System.*out*.println(d.~~toLocaleString~~());

}

}

## DateFormat类 & SimpleDateFormat

DateFormat 是日期/时间格式化子类的抽象类，它以与语言无关的方式格式化并解析日期或时间。日期/时间格式化子类（如 **SimpleDateFormat类**）允许进行格式化（也就是日期 -> 文本）、解析（文本-> 日期）和标准化。

我们通过这个类可以帮我们完成日期和文本之间的转换。

继续阅读API，DateFormat 可帮助进行格式化并解析任何语言环境的日期。对于月、星期，甚至日历格式（阴历和阳历），其代码可完全与语言环境的约定无关。

### DateFormat&SimpleDateFormat的常用方法

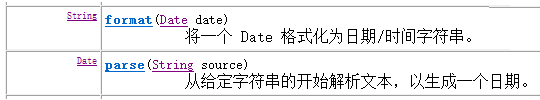
要格式化一个当前语言环境下的日期也就是日期 -> 文本），要通过下面的方法来完成。DateFormat是抽象类，我们需要使用其子类SimpleDateFormat来创建对象。

A:SimpleDateFormat构造方法





B:DateFormat类方法



### 案例代码六:

package com.itheima\_04;

import java.text.ParseException;

import java.text.SimpleDateFormat;

import java.util.Date;

/\*

\* SimpleDateFormat:

\* 格式化：

\* Date --- String

\* 2049-8-26 2049年8月26日

\* String format(Date date)

\* 解析：

\* String --- Date

\* "2049-8-26"

\* Date parse(String source)

\*

\* 构造方法：

\* SimpleDateFormat() ：使用默认的模式进行对象的构建

\* SimpleDateFormat(String pattern) ：使用的指定的模式进行对象的构建

\*

\* 注意：Exception in thread "main" java.text.ParseException: Unparseable date: "49年9月26日 下午1:29"

\* 解析的字符串，模式必须和构建对象的模式一样

\*

\*/

public class SimpleDateFormatDemo {

public static void main(String[] args) throws ParseException {

//method();

//method2();

//使用指定的模式进行对象的构建

//1999年9月1日 10:10:10

//4个小姨2个大美眉和2个小弟弟

SimpleDateFormat sdf = new SimpleDateFormat("yyyy年MM月dd日 HH:mm:ss");

//格式化

Date date = new Date();

String s = sdf.format(date);

System.out.println(s);//2049年08月26日 13:39:12

//解析

Date d = sdf.parse("2049年08月26日 13:39:12");

System.out.println(d.toLocaleString());

}

private static void method2() throws ParseException {

//使用指定的模式进行对象的构建

//1999年9月1日

SimpleDateFormat sdf = new SimpleDateFormat("yyyy年MM月dd日");

//格式化

Date date = new Date();

String s = sdf.format(date);

System.out.println(s);//2049年08月26日

//解析

Date d = sdf.parse("2049年08月26日");

System.out.println(d.toLocaleString());

}

private static void method() throws ParseException {

//使用默认模式进行对象的构建

SimpleDateFormat sdf = new SimpleDateFormat();

//创建日期对象

Date date = new Date();

//格式化 把日期对象转换成字符串

String s = sdf.format(date);

System.out.println(s);//49-8-26 下午1:29

//解析 把字符串转换成日期对象

Date d = sdf.parse("49年9月26日 下午1:29");

System.out.println(d.toLocaleString());

}

}

## Calendar类

### Calendar类概述

Calendar是日历类，在Date后出现，替换掉了许多Date的方法。该类将所有可能用到的时间信息封装为静态成员变量，方便获取。

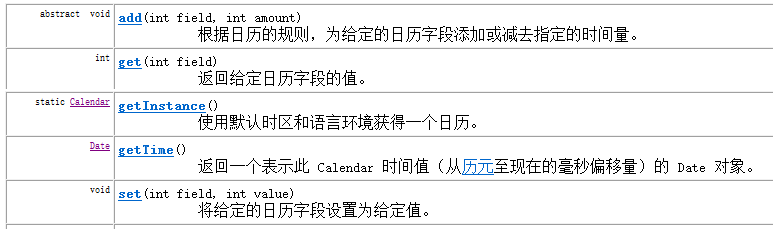
Calendar为抽象类，由于语言敏感性，Calendar类在创建对象时并非直接创建，而是通过静态方法创建，将语言敏感内容处理好，再返回子类对象，如下：

Calendar类静态方法



Calendar c = Calendar.getInstance(); //返回当前时间

### Calendar类常用方法



### 案例代码七:

**package** com.itheima\_05;

**import** java.util.Calendar;

/\*

\* Calendar：日历，提供了一些操作年月日时的方法

\*

\* 获取

\* 修改

\* 添加

\*

\*

\*/

**public** **class** CalendarDemo {

**public** **static** **void** main(String[] args) {

//static Calendar getInstance()

Calendar c = Calendar.*getInstance*();

//void set(int field, int value) ：把指定的字段修改成指定的值

//c.set(Calendar.DAY\_OF\_MONTH, 20);

//void add(int field, int amount): 在指定的字段上加上指定的值

c.add(Calendar.*DAY\_OF\_MONTH*, -1);

//int get(int field) // 返回给定日历字段的值

//public static final int YEAR 1

//System.out.println(Calendar.YEAR);

//int year = c.get(1);

**int** year = c.get(Calendar.*YEAR*);

**int** month = c.get(Calendar.*MONTH*) + 1;

**int** day = c.get(Calendar.*DAY\_OF\_MONTH*);

System.*out*.println(year + "年" + month + "月" + day + "日");

}

}

# 包装类&正则表达式

## 包装类

在实际程序使用中，程序界面上用户输入的数据都是以字符串类型进行存储的。而程序开发中，我们需要把字符串数据，根据需求转换成指定的基本数据类型，如年龄需要转换成int类型，考试成绩需要转换成double类型等。那么，想实现字符串与基本数据之间转换怎么办呢？

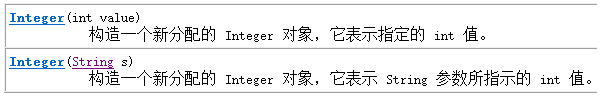
Java中提供了相应的对象来解决该问题，基本数据类型对象包装类：java将基本数据类型值封装成了对象。封装成对象有什么好处？可以提供更多的操作基本数值的功能。

8种基本类型对应的包装类如下：

其中需要注意int对应的是Integer，char对应的Character，其他6个都是基本类型首字母大写即可。

## 包装类的常用方法

A:构造方法:



B:成员方法:







### 案例代码八:

**package** com.itheima\_06;

/\*

\* 需求：判断一个数是否符合int类型的范围

\* 由于基本数据类型只能做一些简单的操作和运算，所以Java为我们封装了基本数据类型，为每种基本数据类型提供了包装类

\* 包装类就是封装了基本数据类型的类，为我们提供了更多复杂的方法和一些变量

\*

\* byte Byte

\* short Short

\* char Character

\* int Integer

\* long Long

\* float Float

\* double Double

\* boolean Boolean

\*

\* Integer:

\* String --- int

\* 方式1：int intValue()

\* 方式2： static int parseInt(String s)

\* int --- String

\* 方式1： + ""

\* 方式2：String toString()

\*

\* 构造方法：

\* Integer(int value)

\* Integer(String s)

\*/

**public** **class** IntegerDemo {

**public** **static** **void** main(String[] args) {

/\*int n = 10;

if(n >= Math.pow(-2, 31) && n <= Math.pow(2, 31) -1) {

System.out.println("符合");

}

else {

System.out.println("不符合");

}\*/

Integer i = **new** Integer("10");

System.*out*.println(i);

**int** a = i.intValue();

System.*out*.println(a + 10 );

**int** b = Integer.*parseInt*("20");

System.*out*.println(b + 30);

Integer i2 = **new** Integer(40);

String s = i2.toString();

System.*out*.println(s);

String s2 = Integer.*toString*(50);

System.*out*.println(s2);

}

}

## 包装类的自动装箱与拆箱

在需要的情况下，基本类型与包装类型可以通用。有些时候我们必须使用引用数据类型时，可以传入基本数据类型。

比如：

基本类型可以使用运算符直接进行计算，但是引用类型不可以。而基本类型包装类作为引用类型的一种却可以计算，原因在于，Java”偷偷地”自动地进行了对象向基本数据类型的转换。

相对应的，引用数据类型变量的值必须是new出来的内存空间地址值，而我们可以将一个基本类型的值赋值给一个基本类型包装类的引用。原因同样在于Java又”偷偷地”自动地进行了基本数据类型向对象的转换。

自动拆箱：对象转成基本数值

自动装箱：基本数值转成对象

### 案例代码九:

**package** com.itheima\_06;

**import** java.util.ArrayList;

/\*

\* JDK1.5特性：自动装箱和拆箱

\*

\*/

**public** **class** IntegerDemo2 {

**public** **static** **void** main(String[] args) {

//Integer i = new Integer(10);

//自动装箱

//相当于： Integer i = new Integer(10);

//Integer i = 10;

//自动拆箱

//相当于 int a = i.intValue();

//Integer i = 10;

//int a = i;

Integer i = 10;

Integer i2 = 20;

Integer i3 = i + i2;

/\*

\* Integer i3 = new Integer(i.intValue() + i2.intValue());

\*

\*/

ArrayList list = **new** ArrayList();

list.add(1);//自动装箱，list.add(new Integer(1));

}

}

## 正则表达式

### 正则表达式概述

正则表达式是专门解决字符串规则匹配的工具。

正则表达式也是一个字符串，用来定义匹配规则。

参照帮助文档，在Pattern类中有简单的规则定义，可以结合字符串类的方法使用。

### 正则表达式匹配规则

参照帮助文档，在Pattern类中有正则表达式的的规则定义，正则表达式中明确区分大小写字母。我们来学习语法规则。

正则表达式的语法规则：

字符：x

含义：代表的是字符x

例如：匹配规则为 "a"，那么需要匹配的字符串内容就是 ”a”

字符：\\

含义：代表的是反斜线字符'\'

例如：匹配规则为"\\" ，那么需要匹配的字符串内容就是 ”\”

字符类：[abc]

含义：代表的是字符a、b 或 c

例如：匹配规则为"[abc]" ，那么需要匹配的内容就是字符a，或者字符b，或字符c的一个

字符类：[^abc]

含义：代表的是除了 a、b 或 c以外的任何字符

例如：匹配规则为"[^abc]"，那么需要匹配的内容就是不是字符a，或者不是字符b，或不是字符c的任意一个字符

字符类：[a-zA-Z]

含义：代表的是a 到 z 或 A 到 Z，两头的字母包括在内

例如：匹配规则为"[a-zA-Z]"，那么需要匹配的是一个大写或者小写字母

字符类：[0-9]

含义：代表的是 0到9数字，两头的数字包括在内

例如：匹配规则为"[0-9]"，那么需要匹配的是一个数字

字符类：[a-zA-Z\_0-9]

含义：代表的字母或者数字或者下划线(即单词字符)

例如：匹配规则为" [a-zA-Z\_0-9] "，那么需要匹配的是一个字母或者是一个数字或一个下滑线

预定义字符类：.

含义：代表的是任何字符

例如：匹配规则为" . "，那么需要匹配的是一个任意字符。如果，就想使用 . 的话，使用匹配规则"\\."来实现

预定义字符类：\d [0-9]

含义：代表的是 0到9数字，两头的数字包括在内，相当于[0-9]

例如：匹配规则为"\d "，那么需要匹配的是一个数字

预定义字符类：\w [a-zA-Z\_0-9]

含义：代表的字母或者数字或者下划线(即单词字符)，相当于[a-zA-Z\_0-9]

例如：匹配规则为"\w "，，那么需要匹配的是一个字母或者是一个数字或一个下滑线

数量词：X?

含义：代表的是X出现一次或一次也没有

例如：匹配规则为"a?"，那么需要匹配的内容是一个字符a，或者一个a都没有

数量词：X\*

含义：代表的是X出现零次或多次

例如：匹配规则为"a\*" ，那么需要匹配的内容是多个字符a，或者一个a都没有

数量词：X+

含义：代表的是X出现一次或多次

例如：匹配规则为"a+"，那么需要匹配的内容是多个字符a，或者一个a

数量词：X{n}

含义：代表的是X出现恰好 n 次

例如：匹配规则为"a{5}"，那么需要匹配的内容是5个字符a

数量词：X{n,}

含义：代表的是X出现至少 n 次

例如：匹配规则为"a{5, }"，那么需要匹配的内容是最少有5个字符a

数量词：X{n,m}

含义：代表的是X出现至少 n 次，但是不超过 m 次

例如：匹配规则为"a{5,8}"，那么需要匹配的内容是有5个字符a 到 8个字符a之间

### 案例代码十:

**package** com.itheima\_07;

/\*

\* 校验qq号码

\* 要求必须是5-15位

\* 0不能开头

\* 必须都是数字

正则表达式：就是一套规则，可以用于匹配字符串

boolean matches(String regex) ：判断当前字符串是否匹配指定的正则表达式，如果匹配则返回true，否则返回false

\*

\*

\*/

**public** **class** RegexDemo {

**public** **static** **void** main(String[] args) {

String qq = "12a345";

/\*boolean flag = checkQQ(qq);

System.out.println(flag);\*/

**boolean** flag = qq.matches("[1-9][0-9]{4,14}");

System.*out*.println(flag);

}

**public** **static** **boolean** checkQQ(String qq) {

**int** length = qq.length();

//要求必须是5-15位

**if**(length < 5 || length > 15) {

**return** **false**;

}

//0不能开头

**if**(qq.startsWith("0")) {

**return** **false**;

}

//必须都是数字

**for** (**int** i = 0; i < length; i++) {

//得到参数的每一个字符

**char** c = qq.charAt(i);

**if**(c < '0' || c > '9') {

**return** **false**;

}

}

**return** **true**;//符合要求

}

}