Java Part 1

变量和对象

- byte(8), short(16), int(32), long(64), float(32), double(64), boolean(1, 32一个整体), char(16).
- 保留关键字

boolean	byte	char	double	float	int	long	short	public	private
protected	abstract	final	native	static	strictfp	synchronized	transient	volatile	if
else	do	while	switch	case	default	for	break	continue	assert
class	extends	implements	import	instanceof	interface	new	package	super	this
catch	finally	try	throw	throws	return	void	const	goto	enum

- primitive 主数据类型变量只有8种. 其他都是引用变量(类似C语言中通过指针操作的变量).
- 数组是一个对象.
- Java是通过值传递的, 也就是说通过拷贝传递.
- 传入与传出方法的值类型可以隐含地放大或是明确地缩小.
- 如果方法声明了非void的返回类型, 那就一定要返回和声明类型相同的值.
- 要记得以面向对象的方式思考问题, 专注于程序中出现的事物而不是过程.
- 极限编程.
- 左脑活动: 循序渐进的工作, 解决逻辑问题于分析.
- 右脑活动: 隐喻, 创造性思维, 模式匹配, 可视化.
- Java**不能**对primitive类型做引用(取别名).
- public的类, 只能定义在以类名命名的文件中.
- 注意区分词语, 引用和对象, 引用是指向对象的指针. 所有非基础类型的对象, 实际上都是引用.

函数库

- ArrayList类似vector.
- 在Java的Api中, 类是被包装在包中.
- Java的包的好处:
    - 帮助组织一大堆零散的类.
    - 包可以制造出名称空间, 以便错开相同名称的类.
    - 包可以通过限制同一个包中的类才能相互存储以维护安全性.

- 虽然ArrayList只能携带对象, 而不能是primitive主类型数据类型, 但编译器能自动将主类型数据包装秤object以存放在ArrayList中.
- import和C语言的include并不相同, 运用import只是帮你省下每个类名前的包的名称而已, 程序不会因为用了import而变大或变慢.

- java.lang是一个预先被引用的包，不需要显示的引用，像String，System这些类是独一无二的，存在于java.lang中，Java知道去哪里找他们.

继承与多态

- A extends B，表示A继承自B，A拥有B的所有成员变量和方法，A可以覆盖这些方法，以及添加新的方法和变量.
- 继承的类，调用方法，最低阶的方法会被采用. 就是自下而上，一直找到这个方法为止.
- super这个关键词可以用于取用父类的方法或者成员变量.
- 4种继承中的存取权限，private，default，protected，public，越往右，限制程度越小.
- public的成员会被继承，private的成员不会被继承.
- 当某个类会比其父亲更具有特定意义时使用继承.
- 继承虽然是面向对象程序设计的一个关键特征，但是却不是达成**重用**行为的最简方式.
- Is A 和 Has A，前者是继承，后者是组合.
- 继承下来的方法可以被覆盖，但是实例变量不能被覆盖.
- 在多态下，引用和对象可以是不同类型. 运用多态时，引用类型可以是实际对象类的父类. 例如一个Animal数组，可以放进去Dog，Cat等对象.
- 函数的参数，返回类型也可以多态，参数或者返回值是Animal，可以传入或返回更具体的Cat，Dog.
- 继承树的层次少一点是合理的，但是实际上没有严格的限制，GUI类这边，就比较深.
- 并不是所有的类都可以被继承，有三种方式防止一个类被继承:
    - 存取控制，类不能被标记成私有，但是可以不被比较成公有，非公有的类只能被同一个包内的类做出子类.
    - 使用final修饰符，表示他是继承树的末端，不能被继承.
    - 让类只拥有private的构造过程.

- 如果想要防止某个方法被覆盖，可以用final修饰符修饰这个方法.
- 覆盖方法的规则:
    - 覆盖的方法和原来的方法具有相同的参数(必须要一样)和返回值(返回类型要兼容). 总之就是自雷对象要保证能执行父类的一切.
    - 不能降低方法的存取权限. 也就是说存取权限必须相同或者更加开放，不能覆盖一个公有的方法成私有.

- 方法的**重载**: 重载和多态没有任何关系. 重载的意义是两个方法名称相同，但是参数不同. 重载不是用来满足定义在父类的多态合约的. 重载的方法和覆盖的方法不一样.
    - 返回类型可以不同.
    - 不能只改变返回类型，参数一样是不允许的. 以及如果是继承类中的重载，也要让返回类型是父类版本的返回类型的子类. (???)
    - 重载的条件是使用不同的参数，此时返回类型可以自由定义.
    - 可以任意更改存取权限.

深入多态

- 抽象类，不应该被初始化的类，在类声明前加上abstract修饰符即可．例如Animal类，就不应该被初始化(被new出来)．可以被声明，但是不能被初始化(new Animal()是不对的)．加上abstract修饰符，如果初始化，编译阶段会报错．
- 抽象类除了被继承之外，是没有用途，没有目的，没有值的．除了抽象类可以有static成员以外．
- 除了类以外，可以对方法进行抽象，抽象的方法代表这个方法一定要被覆盖过．**抽象的方法没有实体，直接 以分号结束**

```
public abstract void eat();
```

- 如果声明出一个抽象的方法，就必须将类也标记成抽象的，不能在非抽象的类中拥有抽象的方法．
- 必须实现所有抽象的方法．
- 抽象的方法没有内容，它只是为标记多态而存在，这表示在继承树下的第一个具体的类必须要实现出所有抽象的方法．但是仍然可以通过抽象机制，将负担传给下一层．例如将Animal和Canine都设成抽象的．只有到Dog这层才需要实现抽象方法．
- 抽象类可以带有抽象和非抽象的方法．
- 所有的类都继承于Object这个类．这使得ArrayList这种可以处理各种各样的类．
- Object的一些成员：equals，getClass，hashCode，toString．
- Object没有抽象的方法，也可以被初始化，不是抽象的类．
- 并非所有Object的方法都可以被继承，有些被修饰成final了．
- 当某个对象是以Object类型的实例来引用时，Java会把他当做Object实例，只能调用Object中声明的方法.下面是不对的．也就是说，看起来像是类型退化了．

```
Object test = new Ferrai();
test.goFast();
```

- 一个Dog放进Animal的数组，没有问题，用数组元素调用方法，也是调用的Dog的方法，但是如果你想把这个Dog对象取出，并赋值给另一个引用的时候，编译器会报错，它无法确认这个对象是不是一个Dog，它只认为它是一个Animal．
- 不能将一个Animal的对象引用赋值给一个Dog对象．
- instanceof 运算符，检查a是不是b类的一个实例．如果是，可以强制转化，例如一个被声明成Animal的引用的Dog，用instanceof运算，确认是Dog的实例之后，可以强制转化，赋值给一个Dog的引用．如果类型转化错误，会有ClassCastException．并且程序终止．
- 接口，interface这个关键字，可以用来解决多态继承，而不出现致命方块问题．用implements这个关键字．

```
public interface Pet {...}
public class Dog extends Canine implements Pet {...};
```

- 接口，更像是一个集合，里面放着一些通用的方法，能被不同的类使用，这些类的相对关系有很复杂，可能和原有的继承树的关系交错．例如，Pet这个接口，里面定义了Play函数，Dog和Cat和RobotDog类都可以实现这个接口．这表明不同的继承树中的类可以实现相同的接口．
- 类可以同时实现多个接口，但是却只能继承一个类．

```
public class Dog extends Animal implements Pet, Saveable, Runnable {...}
```

- 一个Dog类型的引用实例不能调用接口Pet里的函数，只有被声明成Pet的引用的时候，可以调用．Dog能通过IS-A Pet的测试．能用instanceof运算符判断是不是实现了某个接口．
- 要如何判断应该设计一个类，还是子类，还是抽象类，还是接口呢?
    - 如果新的类无法对其他类通过IS-A测试，那么就设计成不继承其他类的类．
    - 只有在需要某个类的特殊化版本时，以覆盖或者增加新的方法来继承现有的类．
    - 当需要定义一群子类的模板的时候，有不想让程序员初始化这个模板，设计出一个抽象类给他们用．
    - 如果想要定义出**类可以扮演的角色**，使用接口．(例如，Dog可以扮演Pet)

内存管理和垃圾回收

- 方法调用和局部变量在栈上(包括引用变量)，对象内存在堆上(引用对象指向的实际内容)．
- 实例变量，是被声明在类里，而不是方法里．例如Dog.size．
- 局部变量，方法的参数都是被声明在方法中的，他们是暂时的，且生命周期只限于方法被放在栈上这段时间(也就是方法调用到方法执行结束这段时间按)．
- 不管是实际变量还是引用变量，对象本身都会在堆上．
- 一个类中带有另一个对象，例如CellPhone里有一个Antenna，那么在new一个CellPhone会在堆上分配多少空间?
    - 基础数据类型，会根据所占内存大小，直接分配．
    - Antenna这种对象，CellPhone中只会分配空间保存它的引用变量的值，而不是对象本身，只有Antenna这个对象真正被new出来的时候，会在堆上重新分配内存．(实际上就像，CellPhone里只有一个Antenna的指针．Java里的引用都这样理解就可以了．)

- 构造函数没有返回类型，构造函数名称一定要和类名相同．构造函数让你有机会介入new的过程．
- 构造函数不会被继承．
- 编译器只会在你完全没有设定构造函数的时候，帮你写一个没有参数，什么也不做的构造函数．其他任何时候，只要你自己实现了构造函数，编译器都不会多此一举来帮你实现一个没有参数的构造函数．
- 一个以上构造函数，参数一定要不同．
- 基础数据类性的默认值是 0/0.0/false/null．
- 构造函数可以是公有私有或者不指定的．
- 一个继承而来的类，包含里所有父类的所有成员，所以内存上也会有分配有那些父类成员变量的空间．
- 父类的构造函数都会在子类的对象的创建过程中执行，就算是抽象类也会有．super()，就是

调用父类的构造函数. super(), 跳到父类的时候, 又会调用他的父类, 直到Object这个类, 然后再一路执行构造函数里的内容, 然后弹出, 回到原来的构造函数.

- 构造函数里面, 如果没有显示调用super(), 这个父类构造函数, 编译器会帮我们默认加上. 以及如果有显示地调用super()必须是构造函数里的第一个语句.
- 如果父类有很多个构造函数, super(), 这个语句也只会调用那个没有参数的构造函数. 或者就是super(args), 显示调用有参数的构造函数.
- this对对象本身的一个引用, this()只能用在构造函数中, 且它必须是第一行语句. 不能和super()同时出现在一个构造函数里. (???)
- 对象的生命周期完全要看引用到它的引用. life和scope不同, 只要变量的堆栈块还存在在堆栈块上, 变量或者对象就还活着. scope是说, 局部变量的范围, 只限于声明它的方法之内. 可以活着, 但不在目前的范围内.

数字与静态

- Java中没有东西是全局的.
- 无法声明成一个Math的对象. Math()的构造函数被声明成私有的. 你不能new一个Math对象. 然而Math的所有方法都是静态的.
- 静态方法就是不依靠实例变量, 也不需要对象的行为. 例如Math.abs(), Math.round(); 静态的标识符是static.
- 以类的名称调用静态的方法. 用引用变量的名称调用非静态的方法.
- 取得新对象的方法只有:
    - new
    - 序列化(deserialization)
    - Java Reflection API

- 静态的方法不能调用非静态的变量和方法.
- 可以用对象引用名称去调用静态的方法, 例如 Dog dog = new Dog(); dog.main(); 但是**合法的东西, 未必是好的**.
- 静态变量, 是在类被加载的时候初始化的. 类会被加载是因为Java虚拟机认为它该被加载了. 通常Java虚拟机会加载某个类是因为第一次有人尝试创建该类的新实例. 程序员也可以强制Java虚拟机去加载某个类, 但是一般不太需要这么做.
- 静态变量会在该类的任何对象创建之前完成初始化. 会在该类的任何静态方法执行前初始化.
- 静态的final变量, 是常数. 一旦被初始化了之后, 就不会改动. final是唯一的表示一个变量为不变的常数的方法. 常量通常用大写字母命名, 单词间用下划线分割.

```
public static final double PI = 3.1415;
```

- 静态的**final**变量的初始化方式:
    - `public static final double PI = 3.1415;` 直接在声明的时候给初始化的值.
    - 或者在一个static标识的代码块中初始化, 这段代码会在类被加载的时候初始化:

```
static {
    PI = (double) Math.random();
}
```

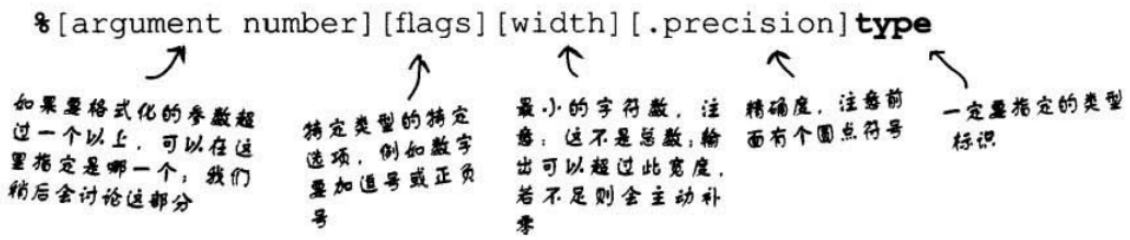
- 如果没有用以上两种方法给静态final变量初始化的值，编译器会发现这个错误，报一个error. 非final的变量不给初始化值，会给一个默认的值. 完全没有问题，不会报错.
- 前面的那个static的代码快会在类的构造函数之前执行. (亲测).
- final不止用在静态变量上. final类似C语言中的const，表示，值不能变动. 以及可以防止方法在继承过程中被覆盖或者创建子类.
- 如果一个类是final，就不用把他的方法标记成final防止被覆盖了，因为本身就不可能有子类.
- final变量必须被显示地赋值，可以在声明的时候赋值. 可以在类的(每个)构造函数中被赋值. 不允许没有显式的赋值. 编译器会报错. 如果不是静态的，赋值语句不能放在前面那个static的语句块中，那里只能初始化静态的变量.
- 注意Java代码中的Shadow. 如下代码是不会报错的，能够通过编译. 这里的x，shadow了外面的变量，所以这个static函数没有调用非static的成员变量. 没有问题.

```
class Foo {
  int x = 12;
  public static void go(final int x) {
    System.out.println(x);
  }
}
```

- 每个perimitive的类型，首字母大写，就是对应的包装好的类. 注意char对应的是Character.
- 基础类型的包装对象实例，如果没有初始化，回事一个null的引用，这时候，如果尝试引用这个对象的值，会有异常抛出.
- 格式化输出字符串和C语言类似，可以给数字输出格式加’,’号(三位一个逗号这样).

```
System.out.println("%,.2f", 12345.789);
12,345.78
```

- 格式化输出.



- 静态import. 可以让程序员少打几个单词，例如Math.abs，可以直接打abs. 但是容易出现

名称冲突. 语法是: import static java.util.*;

- 构造函数不能用static修饰, 也不用final修饰, 因为构造函数不能被覆盖.