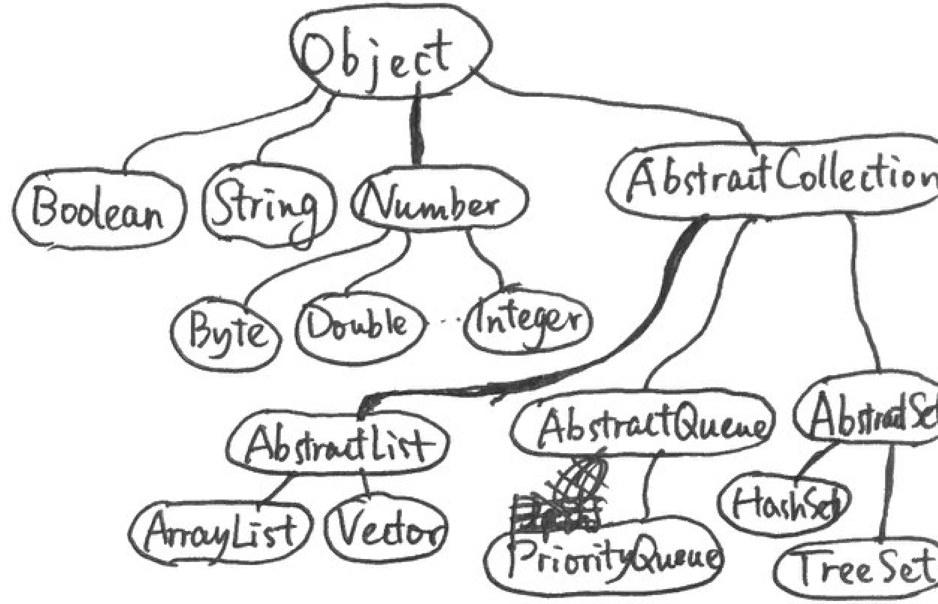
# 子类关系

在以Java为代表的「面向对象」语言中，父类与子类的概念尤为重要。顾名思义，子类继承父类的衣钵，是父类的特殊化，它在父类之基础上，添加众多独有的功能，实现更为精密的操作。本章的主题，正是把这种父子关系引入类型系统中。

Java的顶层类称作 Object，所有其它类均直接或间接地为 Object 的子类。Object 仅提供一些最通用的操作，例如 clone()，equals()，toString() 等「放之四海皆准」的函数。在 Object 类的基础上，我们可逐级细化，搭建出各式各样的子类，达成不同的功能。下面展示了一棵关系树：




这只是Java语言庞大关系树的冰山一角，但已足以带来直观感受了。以下是两点显而易见的观察：

**Observation**
从下往上行走（亦即从子类往父类走），有
(1) 功能越来越少，支持的操作越来越贫乏；
(2) 抽象程度越来越高，适用的范围越来越宽泛。

这两点观察实是一个硬币的两面，是没有矛盾、相互等价的。它们恰恰印

证了「外延愈大，内涵愈小」的原理。

我们在剩下部分所做的，无非就是定义出一种排序关系 $\le$，以刻画子类与父类之间的关系。我们的目标是：使 $T_1 \le T_2$ 意味着「$T_2$ 的功能比 $T_1$ 少（准确而言，$T_2$ 的功能是 $T_1$ 的功能之子集），适用范围比 $T_1$ 宽，从而 $T_1$ 是子类，$T_2$ 是父类」。

在把 $\le$ 定义妥当以后，我们再引入一条新的类型推导规则：

$$\frac{\Gamma \vdash t : T_1 \qquad T_1 \le T_2}{\Gamma \vdash t : T_2}\ \text{[T-SUB]}$$

意为「我们可随时将类型 $T_1$ 向上转化为类型 $T_2$」。假若 $\le$ 的定义确实符合预期，那么这种转化相当于「忘记」$T_1$ 所携带的精细信息，而只保留部分通用的信息，从而将对象泛化了。

泛化以后，对象就以类型 $T_2$ 存在；而凡是 $T_2$ 具有的信息/操作，该对象必然拥有，因此，从直觉上来说，类型系统的安全性理应不被破坏。而且，不难见得，允许类型之向上转化将使我们的程序书写（尤其是函数书写）更加灵活。

先来进行第一步：定义 $\le$。

既然要定义一种排序，那么自然而然须满足以下两条抽象性质：

$$\frac{}{T \le T}\ \text{[REFLEX]} \qquad \frac{T_1 \le T_2 \qquad T_2 \le T_3}{T_1 \le T_3}\ \text{[TRANS]}$$

然后考虑元组之间的关系：

$$\frac{m \in \mathbb{N}_0}{T_1 \times T_2 \times \cdots \times T_{n+m} \le T_1 \times T_2 \times \cdots \times T_n}\ \text{[TUPLE-WIDTH]}$$

$$\frac{T_1 \le T_1' \quad \cdots \quad T_n \le T_n'}{T_1 \times \cdots \times T_n \le T_1' \times \cdots \times T_n'}\ \text{[TUPLE-DEPTH]}$$

我们仔细分析一下 [TUPLE-WIDTH] 为何捕捉到了我们的直觉：

(1) 从 $T_1 \times \cdots \times T_{n+m}$ 到 $T_1 \times \cdots \times T_n$，信息量从 $n+m$ 维降低到 $n$ 维了，支持的操作也从原来的 $.i$ $(i=1,\ldots,n+m)$

减少者.$i$ $(i=1,\cdots,n)$

(2) 从 $T_1\times\cdots\times T_{n+m}$ 到 $T_1\times\cdots\times T_n$，约束条件放宽了（原有 $n+m$ 个，现为 $n$ 个）。

请你仿此论证第一规则的合理性。

接着考察柔性类型：

$$\frac{m\in\mathbb{N}_0}{T_1+\cdots+T_n \le T_1+\cdots+T_{n+m}}\ [\text{SOFT-WIDTH}]$$

$$\frac{T_1\le T_1' \quad \cdots \quad T_n\le T_n'}{T_1+\cdots+T_n \le T_1'+\cdots+T_n'}\ [\text{SOFT-DEPTH}]$$

注意从 $T_1+\cdots+T_n$ 到 $T_1+\cdots+T_{n+m}$，信息量不增反减——可能性变多，适用范围宽泛了。前者支持 $n$ 分支的case，后者支持 $n+m$ 分支的case，看似后者的操作能力更强，实则不然。请想想看，前者难道不支持 $n+1/n+2/\cdots/n+m/\cdots$ 分支的case吗？非也！而后者却的确不支持小于 $n+m$ 分支的case，否则会丧失安全性。更精确的说法是这样的：$T_1+\cdots+T_n$ 支持 $\ge n$ 分支的case（显然若 $>n$ 则有若干分支无用），而 $T_1+\cdots+T_{n+m}$ 却仅支持 $\ge n+m$ 分支的case。显然，后者支持的操作少于前者。

**problem**

请延续这样的思路，讨论「记录」之间的子类关系并论证之。注意记录标签的次序是可打乱的。

函数之间的子类关系又如何呢？

$$\frac{T_1'\le T_1}{T_1\to T_2 \le T_1'\to T_2}\ [\text{FUNC-ARG}]$$

$$\frac{T_2\le T_2'}{T_1\to T_2 \le T_1\to T_2'}\ [\text{FUNC-RET}]$$

[FUNC-ARG]较难理解。从 $T_1\to T_2$ 到 $T_1'\to T_2$，由于 $T_1'\le T_1$，故参数部分被精细化了，从而，函数支持的操

作也变少了。何也？请先想想，作为一个函数本身，它的特异性操作是什么？显然只有「调用」(application)。从调用操作的角度来说，$T_1' \blacktriangleright T_2$ 仅接受一些精细的参数，因而将许多对象排除在外了，是故调用操作的丰富度降低了。

e.g. 设 $f: Int \times Int \blacktriangleright Int$ $\quad (T_1 \blacktriangleright T_2)$

$g: Int \times Int \times Int \blacktriangleright Int$ $\quad (T_1' \blacktriangleright T_2)$

多余的信息不碍事

$f(\{0,0\})$，$f(\{1,2,3\})$，$f(\{9,8,false\})$ 都是合法的，但 $g(\{0,0\})$ 与 $g(\{9,8,false\})$ 却不合法。

缺少信息或信息不当将造成灾难

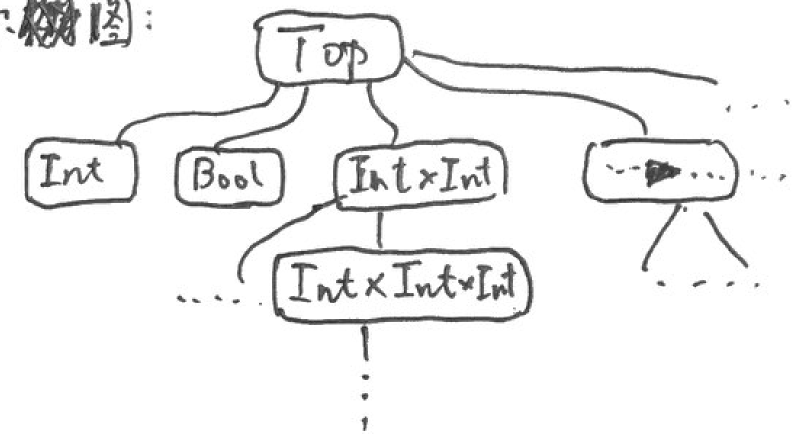
[FUNC-RET] 的作用则与 [TUPLE-DEPTH] 类似，较易理解。

最后，我们引入类型 Top，承担「顶级类」的大任：

$$\frac{}{T \le Top}\ [TOP]$$

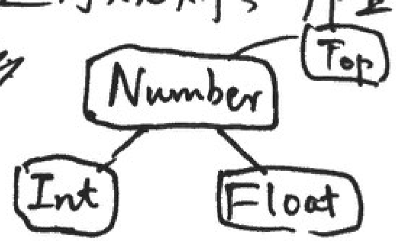
Top是没有任何操作的，因而也是最抽象的。

现在，我们已定义出了如下的关系图：




与Java的一大区别是：这里的 ≤ 关系是无穷的，且构成图而非树。换用Java的术语，此处的 ≤ 支持无穷的多继承。

作为练习，请你引入类型 Number 和 Float，定义其类型推导规则、单步运行规则，并且定义子类关系如下图的：



接下来，进行第二步：引入[T-SUB]，并讨论类型安全性。

[第一步]

$$\frac{}{T \le T}\,[\text{REFLEX}] \qquad \frac{T_1 \le T_2 \quad T_2 \le T_3}{T_1 \le T_3}\,[\text{TRANS}]$$

$$\frac{m \in \mathbb{N}_0}{T_1 \times \cdots \times T_{n+m} \le T_1 \times \cdots \times T_n}\,[\text{TUPLE-WIDTH}]$$

$$\frac{T_1 \le T_1' \quad \cdots \quad T_n \le T_n'}{T_1 \times \cdots \times T_n \le T_1' \times \cdots \times T_n'}\,[\text{TUPLE-DEPTH}]$$

$$\frac{m \in \mathbb{N}_0}{T_1 + \cdots + T_n \le T_1 + \cdots + T_{n+m}}\,[\text{SOFT-WIDTH}]$$

$$\frac{T_1 \le T_1' \quad \cdots \quad T_n \le T_n'}{T_1 + \cdots + T_n \le T_1' + \cdots + T_n'}\,[\text{SOFT-DEPTH}]$$

$$\frac{T_1' \le T_1}{T_1 \blacktriangleright T_2 \le T_1' \blacktriangleright T_2}\,[\text{FUNC-ARG}]$$

$$\frac{T_2 \le T_2'}{T_1 \blacktriangleright T_2 \le T_1 \blacktriangleright T_2'}\,[\text{FUNC-RET}]$$

[第二步]

$$\frac{\Gamma \vdash t : T_1 \quad T_1 \le T_2}{\Gamma \vdash t : T_2}\,[\text{T-SUB}]$$

**Lemma 1**

(1) 若 $T \le T_1 \times \cdots \times T_n$，则 $T = S_1 \times \cdots \times S_{n+m}$ 且 $S_1 \le T_1, \ldots, S_n \le T_n$.

(2) 若 $T \le T_1 + \cdots + T_n$，则 $T = S_1 + \cdots + S_{n-m}$ $(n > m)$ 且 $S_1 \le T_1, \ldots, S_{n-m} \le T_{n-m}$

(3) 若 $T \le T_1 \blacktriangleright T_2$，则 $T = S_1 \blacktriangleright S_2$，且 $T_1 \le S_1$，$S_2 \le T_2$.

*proof.* 习题. ■

**Lemma 2**

(1) 若 $\Gamma \vdash v : T_1 \times \cdots \times T_n$，则 $v$ 的形式必为 $\{v_1, \ldots, v_{n+m}\}$

(2) 若 $\Gamma \vdash v : T_1 + \cdots + T_n$，则 $v$ 的形式必为 $v$ as $S_1 + \cdots + S_{n-m}$.

(3) 若 $\Gamma \vdash v : T_1 \blacktriangleright T_2$，则 $v$ 的形式必为 $\lambda x : S_1 . t$.

*proof.* 习题. ■

**Theorem 3 (Progress)**

若 $t$ 通过了类型检查，则 $t$ 要么已是一个值，要么可单步执行。

proof. 运用 Lemma 2 即可归纳证得. ■

**Lemma 4**

(1) 若 $\Gamma \vdash \{t_1, \dots, t_n\} : T_1 \times \cdots \times T_{n+m}$，
则 $\forall i \in [n]$ 有 $\Gamma \vdash t_i : T_i$

(2) 若 $\Gamma \vdash (\lambda x : S_1 .\ t) : T_1 \to T_2$，
则 $T_1 \le S_1$ 且 $\Gamma \cup \{x : S_1\} \vdash t : T_2$.

(3) 若 $\Gamma \vdash t \text{ as } (T_1 + \cdots + T_n) : S_1 + \cdots + S_{n+m}$
则 $\forall i \in [n]$ 有 $T_i \le S_i$，且
$\exists i \in [n]\quad \Gamma \vdash t : T_i$.

**Lemma 5 (Substitution)** 与 **Theorem 6 (Preservation)** 请自行给出。