迭代器：

```
public static void dropCourse6(ArrayList<String> subjects) {
    MyIterator iter = new MyIterator(subjects);
    while (iter.hasNext()) {
        String subject = iter.next();
        if (subject.startsWith("6.")) {
            subjects.remove(subject);
        }
    }
}
```

结果错误

```
for (String subject : subjects) {
    if (subject.startsWith("6.")) {
        subjects.remove(subject);
    }
}
```

报错

```
Iterator iter = subjects.iterator();
while (iter.hasNext()) {
    String subject = iter.next();
    if (subject.startsWith("6.")) {
        iter.remove();
    }
}
```

正确

To create immutable collections from some known values, use
List.of, Set.of, and Map.of.
List<String> a = List.of("lion", "tiger", "bear");
Alternatively, use List.copyOf (Java 10) to create unmodifiable shallow copies of mutable collections

作为 find 的实现者，如果违反了前提条件，为什么要选择抛出异常？
当我们的前提条件被违反时，客户端就有一个 bug。我们可以通过快速失败使这个 bug 更容易发现和修复，即使我们没有义务这样做。

precondition 前置条件：对客户端的约束，在使用方法时必须满足的条件
使用@param annotation 说明每个参数的前置条件
postcondition 后置条件：对开发者的约束，方法结束时必须满足的条件

使用@return annotation 说明后置条件
使用@throws annotation 说明出现异常的时候会发生什么
在方法声明中使用 static 等关键字声明，可据此进行静态类型检查

1.Try 不能没有 cache 或 finally
2.try-catch 块中的变量遵循正常的作用域规则。当 birthdate 在 try 块中声明时，一旦该块结束，它就会超出范围。我们不能指派给它，或者根本不能引用它，无论是在那里还是在 catch 别处。

是否使用前置条件取决于
(1) check 的代价；
(2) 方法的使用范围
- 如果只在类的内部使用该方法(private)，那么可以不使用前置条件，在使用该方法的各个位置进行 check——责任交给内部 client；
- 如果在其他地方使用该方法(public)，那么必须要使用前置条件，若 client 端不满足则方法抛出异常。

四类 ADT 操作
Creators 构造器
实现：构造函数 constructor 或静态方法（也称 factory method）
Producers 生产器
需要有“旧对象”
return 新对象
eg. String.concat()
Observers 观察器
eg. List 的.size()
Mutators 变值器
改变对象属性
若返回值为 void，则必然改变了对象内部状态（必然是 mutator）

Java 接口不能具有构造函数

静态工厂方法：

```
Public static ClassName valueOf(boolean b){
    Return new newClassName(b);
}
```

valueOf/getInstace

```java
public interface MyString {

    /** @param b a boolean value
     *  @return string representation of b, either "true" or "false" */
    public static MyString valueOf(boolean b) {
        return new FastMyString(true);
    }

    // ...
```

```java
MyString s = MyString.valueOf(true);
System.out.println("The first character is: " + s.charAt(0));
```

接口的优点

1. 防漏洞
   (1) 一个 ADT 是由其操作来定义的，而接口就是这样做的。
   (2) 当客户端使用接口类型时，静态检查可确保它们只使用由接口定义的方法。
   (3) 如果实现类公开了其他方法，或者更糟的方法，具有可见表示，客户端不会意外看到或依赖它们。
   (4) 当我们有多个数据类型的实现时，接口就提供了对方法签名的静态检查。
2. 易于理解
   (1) 客户和维护人员确切地知道该在哪里寻找 ADT 的规范。
   (2) 由于该接口不包含实例字段或实例方法的实现，因此更容易将该实现的详细信息排除在规范之外。
3. 便于更改
   (1) 我们可以通过添加实现接口的类来轻松地添加一种类型的新实现。
   (2) 如果我们避免构造函数支持静态工厂方法，客户机只能看到接口。
   (3) 这意味着我们可以在不更改代码的情况下切换实现类客户端使用的内容。

信息隐藏的优点

1. 解耦组成一个系统的类
   允许它们被单独地开发、测试、优化、使用、理解和修改
2. 加快系统开发速度
   类可以并行开发
3. 减轻维护工作的负担
   类可以更快地理解并进行调试，几乎不担心伤害其他模块
4. 启用有效的性能调整
   “热”类可以单独进行优化
5. 增加了软件的可重用性
   松散耦合的类在其他上下文中通常被证明是有用的

构造函数调用必须是构造函数中的第一个语句

Override：dynamic checking，不能抛出新的异常，不能有更严格的访问权限（private、public、protected）

Overlord：static checking，必须改变参数列表

	Overloading	Overriding
Argument list	Must change	Must not change
Return type	Can Change	Must not change
Exceptions	Can Change	Can reduce or eliminate Must not throw new or broader checked exception
Access	Can Change	Must not make more restrictive (can be less restrictive)
Invocation	Reference type determines which overloaded version (based on declared argument types) is selected. Happens at compile time. The actual method that's invoked is still a virtual method invocation that happens at runtime, but the compiler will always know the signature of the method that is to be invoked. So at runtime, the argument match will have already been nailed down, just not the actual class in which the method lives	Object type (in other words, the type of the actual instance on the heap) determines which method is selected. Happens at runtime.

泛型的实现细节

1. 可以有多个类型的参数
2. 通配符，只在使用泛型的时候出现，不能在定义中出现
   (1) ? Extend E 上界通配符
   (2) ? Super E 下界通配符
3. 泛型信息将被删除。（仅限编译时使用）
4. 无法创建常规数组

```
Pair<String>[] foo = new Pair<String>[42]; // won't compile
```

每个类只能直接扩展一个父类；一个类可以实现多个接口

子类型的规约不能弱化超类型的规约。

为什么向下转型是不安全的？

当用父类变量指向子类类型的时候,变量只有父类的方法而不能调用子类的方法,若想调用子类的方法就要把父类变量转换为子类变量(向下转型)。但当父类变量转换为一个与真实对象不相符的子类变量的时候,就会抛出 ClassCastException,所以向下转型是不安全的,正确的做法是在强制类型转换之前先使用 Instance of 关键字进行判断转型后的类型和真实对象的类型是否一致,一致才执行类型转换操作;

如何编写一个不可变的类

1.不提供任何致突变器

2.确保不会覆盖任何方法

3.使所有字段设置为 final private
4.确保任何可变部件的安全性（避免代表曝光）
5.实现 toString(), hashCode(), clone(), equals()等

Equals:

```
@Override
public boolean equals(Object that) {
    return that instanceof Duration && this.sameValue((Duration)that);
}

// returns true iff this and that represent the same abstract value
private boolean sameValue(Duration that) {
    return this.getLength() == that.getLength();
}
```

Hashcode
相等的对象必须具有相等的哈希码
如果覆盖了 equal，则必须覆盖 hashCode

一个简单而有效的方法来确保契约得到满足，就是 hashCode 总是返回一些常量值，所以每个对象的 hash 码都是相同的。这满足了对象契约，但会对性能产生灾难性的影响，因为每个键都将存储在同一个槽中，并且每个查找都将退化为沿长列表的线性搜索。

标准是为用于确定相等性的对象的每个组件计算一个哈希码（通常通过调用每个组件的 hashCode 方法），然后将它们组合起来，抛出一些算术运算。

对可变类型，实现行为等价性即可：也就是说，只有指向同样内存空间的 objects，才是相等的。所以对可变类型来说，无需重写这两个函数，直接继承 Object 的两个方法即可。如果一定要判断两个可变对象看起来是否一致，最好定义一个新的方法。

对于不可变类型：

1.行为平等与观察平等相同。
2.equals() 必须覆盖以比较抽象值。
3.hashCode() 必须重写以将抽象值映射到整数。

对于可变类型：

1.行为平等不同于观察平等。
2.equals()通常不应被覆盖，而是从 Object 比较引用继承实现，就像==.
3.hashCode()同样不应被覆盖，但应该继承 Object 将引用映射到整数的实现。