# 静态检查

用一个例子引入什么是静态检查：
冰雹队列，队列第一个数字是n，如果n是偶数，下一个数字是n/2，如果n是奇数，下一个数字是3n+1，直到为1后停止（这个队列应该不会进入死循环，但是目前好像没有人能证明。）
用java和python编写一下大致的程序：

```java
// Java
int n = 3;
while (n != 1) {
    System.out.println(n);
    if (n % 2 == 0) {
        n = n / 2;
    } else {
        n = 3 * n + 1;
    }
}
System.out.println(n);
```

```python
# Python
n = 3
while n != 1:
    print(n)
    if n % 2 == 0:
        n = n / 2
    else:
        n = 3 * n + 1


print(n)
```

不难看出java是静态类型语言（所有变量都提前用int double等 定义好了是什么类型，编译时机器可以自动检查是否存在类型的错误） python是动态类型语言（所有的变量的类型检查都会推迟到程序运行时进行）

# 静态检查，动态检查，无检查

静态检查：
1.语法错误：

收起 ^

java

```java
public class SyntaxErrorExample {
    public static void main(String[] args) {
        // 多余的逗号，这是语法错误，编译器在编译时会报错
        int x = 5,;
    }
}
```

2.拼写错误的名称：

java

```java
import java.lang.Math;

public class NameSpellingErrorExample {
    public static void main(String[] args) {
        // 方法名拼写错误，正确的是 sin
        double result = Math.sine(30);
    }
}
```

3.参数数量错误

java

```java
import java.lang.Math;

public class ArgumentCountErrorExample {
    public static void main(String[] args) {
        // sin 方法只接受一个参数，这里传递了两个，会在编译时出错
        double result = Math.sin(30, 20);
    }
}
```

4.参数类型错误

java

```java
import java.lang.Math;

public class ArgumentTypeErrorExample {
    public static void main(String[] args) {
        // sin 方法需要 double 类型参数，这里传递了字符串，编译时会报错
        double result = Math.sin("30");
    }
}
```

5.返回类型错误

```java
public class ReturnTypeErrorExample {
    public static int returnInteger() {
        // 方法声明返回 int 类型，但返回了字符串，编译时会出错
        return "30";
    }

    public static void main(String[] args) {
        int result = returnInteger();
    }
}
```

动态检查

1.非法参数值（除零错误）

```python
def divide(x, y):
    return x / y

# 当 y 为 0 时会引发 ZeroDivisionError，运行时才会发现
result = divide(10, 0)
```

2.java中的非法转换

```java
public class IllegalConversionExample {
    public static void main(String[] args) {
        // 字符串 "hello" 不能解析为整数，运行时会抛出 NumberFormatException
        int num = Integer.valueOf("hello");
    }
}
```

3.python中的越界索引

```python
my_list = [1, 2, 3]
# 列表索引越界，运行时会引发 IndexError
element = my_list[10]
```

4.java中的空对象引用调用方法

```java
public class NullReferenceExample {
    public static void main(String[] args) {
        String str = null;
        // 对空对象引用调用方法，运行时会抛出 NullPointerException
        int length = str.length();
    }
}
```

## 意外之处：基本数据类型并非真正的数字

Java 以及许多其他编程语言中的一个陷阱是，其基本数值类型存在一些特殊情况，表现与我们所熟悉的整数和实数不同。因此，一些实际上本应进行动态检查的错误根本没有得到检查。以下是这些陷阱：

- **整数除法**：`5/2` 不会返回一个分数，而是返回一个截断后的整数。所以在这种情况下，我们原本希望是一个动态错误（因为分数不能用整数表示），但结果却常常产生错误的答案。
- **整数溢出**：`int` 和 `long` 类型实际上是有限的整数集合，有最大值和最小值。当计算结果太大（正或负）而无法容纳在这个有限范围内时会发生什么？计算会悄然溢出（循环回绕），并从合法范围内的某个地方返回一个整数，但并非正确答案。
- **浮点类型中的特殊值**：像 `double` 这样的浮点类型有几个特殊值，它们并非真正的数字：`NaN`（表示“非数字”）、`POSITIVE_INFINITY` 和 `NEGATIVE_INFINITY`。所以当你对 `double` 类型执行某些你预期会产生动态错误的操作时，比如除以零或对负数取平方根，你得到的将是这些特殊值之一。如果你继续使用它进行计算，最终会得到一个错误的结果。

# 巧用final

对于认定的不变的量用final修饰，防止意外修改
例子：

## READING EXERCISES

### final

Consider the variables in our `hailstoneSequence` method:

```
public static List<Integer> hailstoneSequence(int n) {
    List<Integer> list = new ArrayList<Integer>();
    while (n != 1) {
        list.add(n);
        if (n % 2 == 0) {
            n = n / 2;
        } else {
            n = 3 * n + 1;
        }
    }
    list.add(n);
    return list;
}
```

Which variables can we declare `final`, because they are never reassigned in the code?

- [ ] `int n`
- [x] `List<Integer> list`

`n` is reassigned many times in the code, for example by `n = n / 2`. So we can't declare it `final`.

But `list` is only assigned once, to `new ArrayList<Integer>()`. Subsequently, even though we call `add()` on the list to add newly-discovered elements of the hailstone sequence, we never reassign the `list` *variable*

> to refer to a different list. So `list` can be declared `final`.

## 声明假设

比如年龄，这个量不可能是负数，所以你要用注释等声明age>=0这一假设

## 本课程的核心目的：

- **Safe from bugs.** Static checking helps with safety by catching type errors and other bugs before runtime.
- **Easy to understand.** It helps with understanding, because types are explicitly stated in the code.
- **Ready for change.** Static checking makes it easier to change your code by identifying other places that need to change in tandem. For example, when you change the name or type of a variable, the compiler immediately displays errors at all the places where that variable is used, reminding you to update them as well.