LECTURE 2: STATIC ANALYSIS

Dr. Ehab Essa

Computer Science Department, Mansoura University

HAILSTONE SEQUENCE

Starting with a **number n,** the next number in the sequence is **n/2** if n is even, or **3n+1** if n is **odd**. The sequence ends when it reaches 1. Here are some examples:

- 2, 1
- 3, 10, 5, 16, 8, 4, 2, 1
- 4, 2, 1
- 5, 16, 8, 4, 2, 1
- 7, 22, 11, 34, 17, 52, 26, 13, 40, …? (where does this stop?)

Because of the odd-number rule, the sequence may bounce up and down before decreasing to 1.

Why is it called a **hailstone sequence**? Because hailstones form in clouds by bouncing up and down, until they eventually build enough weight to fall to earth.

COMPUTING HAILSTONES

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

print(n)
```

- Java requires **semicolons** at the ends of statements. The extra punctuation can be a pain, but it also gives you more freedom in how you organize your code – you can split a statement into multiple lines for more readability.
- Java requires **parentheses** around the conditions of the if and while.
- Java requires **curly braces** around blocks, instead of indentation.
- You should always indent the block, even though Java won't pay any attention to your extra spaces. Programming is a form of communication, and you're communicating not only to the compiler, but to human beings. Humans need that indentation.

TYPES

A type is a set of values, along with operations that can be performed on those values.

Java has several **primitive types**, stores the **actual data** directly in memory, among them:

- int (for integers like 5 and -200, but limited to the range $\pm 2^{31}$, or roughly $\pm$ 2 billion)
- long (for larger integers up to $\pm 2^{63}$)
- boolean (for true or false)
- double (for floating-point numbers, which represent a subset of the real numbers)
- char (for single characters like 'A' and '$')

Java also has **object types**, stores a memory address (reference) pointing to the object, for example:

- String: represents a sequence of characters.
- Wrapper Classes: Integer, Double, Boolean
- BigInteger represents an integer of arbitrary size.
- Arrays, Collections: List

TYPES

Operations are functions that take inputs and produce outputs (and sometimes change the values themselves). Here are three different syntaxes for an operation in Python or Java:

- As an infix, prefix, or postfix operator. For example, a + b invokes the operation + : int × int → int.
- As a method of an object. For example, bigint1.add(bigint2) calls the operation add: BigInteger × BigInteger → BigInteger.
- As a function. For example, Math.sin(theta) calls the operation sin: double → double. Here, Math is not an object. It's the class that contains the sin function.

Contrast Java's str.length() with Python's len(str). It's the same operation in both languages – a function that takes a string and returns its length – but it just uses different syntax.

Some operations are overloaded in the sense that the same operation name is used for different types. The arithmetic operators +, -, *, / are heavily overloaded for the numeric primitive types in Java. Methods can also be overloaded.

STATIC TYPING

Java is a **statically-typed language**. The types of all variables are known at compile time (before the program runs), and the compiler can therefore deduce the types of all expressions as well.

If a and b are declared as ints, then the compiler concludes that a+b is also an int. The Eclipse environment does this while you're writing the code, in fact, so you find out about many errors while you're still typing.

In **dynamically-typed languages** like Python, this kind of checking is deferred until runtime (while the program is running).

Static checking, means checking for bugs at compile time.

- Static typing is a particular kind of static checking.
- Static typing prevents bugs caused by applying an operation to the wrong types of arguments.
  - e.g. try "5" * "6"static typing will catch this error while you're still programming, rather than waiting until the line is reached during execution.

STATIC CHECKING, DYNAMIC CHECKING

Static checking: the bug is found automatically before the program even runs.

Dynamic checking: the bug is found automatically when the code is executed.

Static checking tends to be about types, errors that are independent of the specific value that a variable has. So if the error would be caused only by certain values, like divide-by-zero or index-out-of-range then the compiler won't raise a static error about it.

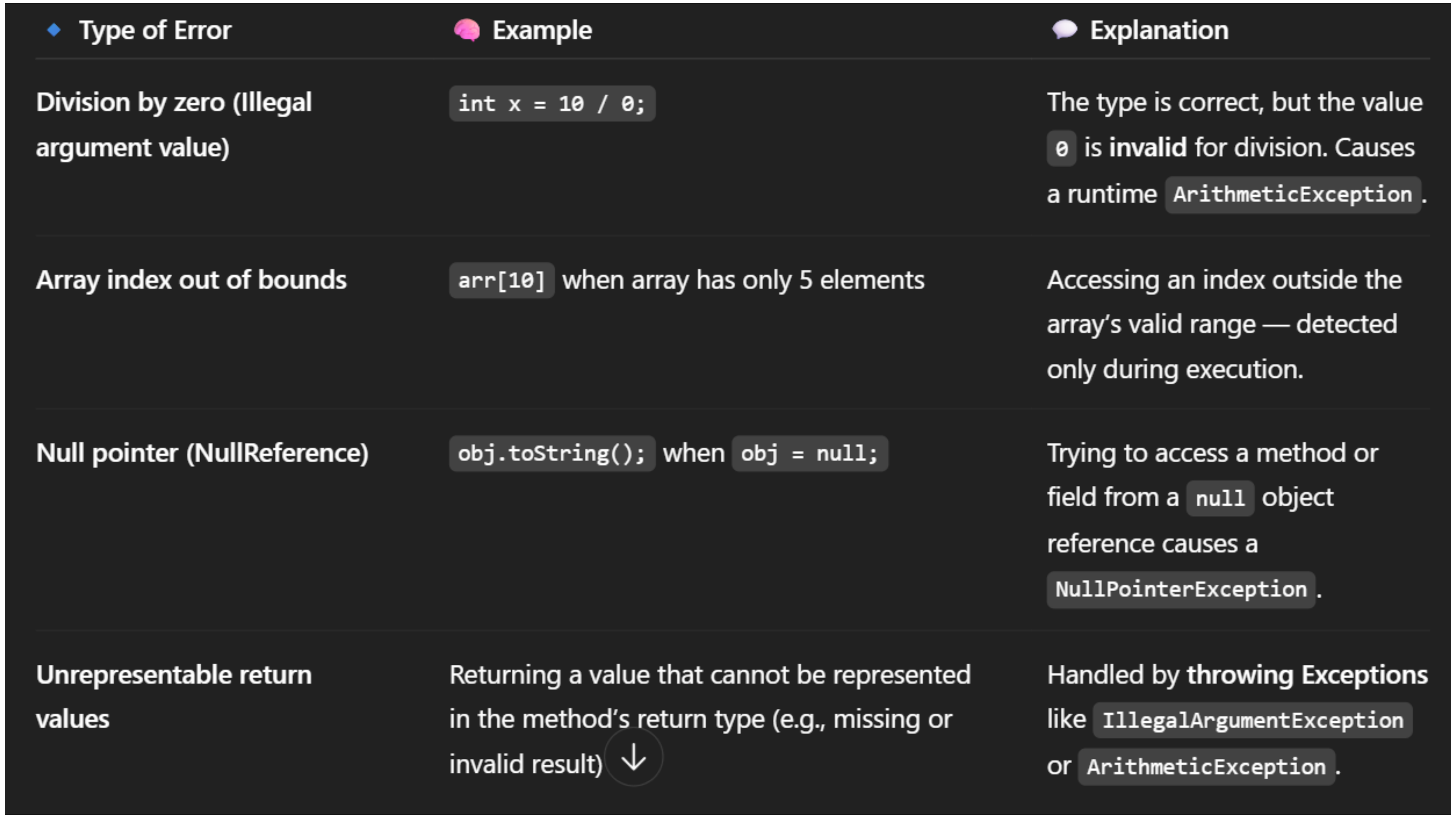
Dynamic checking, by contrast, tends to be about errors caused by specific values.

Static checking can catch:

- syntax errors, like extra punctuation or spurious words. Even dynamically-typed languages like Python do this kind of static checking. If you have an indentation error in your Python program, you'll find out before the program starts running.
- wrong names, like Math.sine(2). (The right name is sin.)
- wrong number of arguments, like Math.sin(30, 20).
- wrong argument types, like Math.sin("30").
- wrong return types, like return "30"; from a function that's declared to return an int.

STATIC CHECKING, DYNAMIC CHECKING

Dynamic checking can catch:



🔹 Type of Error	🧠 Example	💬 Explanation
Division by zero (Illegal argument value)	`int x = 10 / 0;`	The type is correct, but the value `0` is **invalid** for division. Causes a runtime `ArithmeticException`.
Array index out of bounds	`arr[10]` when array has only 5 elements	Accessing an index outside the array's valid range — detected only during execution.
Null pointer (NullReference)	`obj.toString();` when `obj = null;`	Trying to access a method or field from a `null` object reference causes a `NullPointerException`.
Unrepresentable return values	Returning a value that cannot be represented in the method's return type (e.g., missing or invalid result)	Handled by **throwing Exceptions** like `IllegalArgumentException` or `ArithmeticException`.

STATIC ANALYSIS

Analyzing code without executing it. Generally used to find bugs or ensure conformance to coding standards. The classic example is a compiler which finds lexical, and syntactic mistakes.

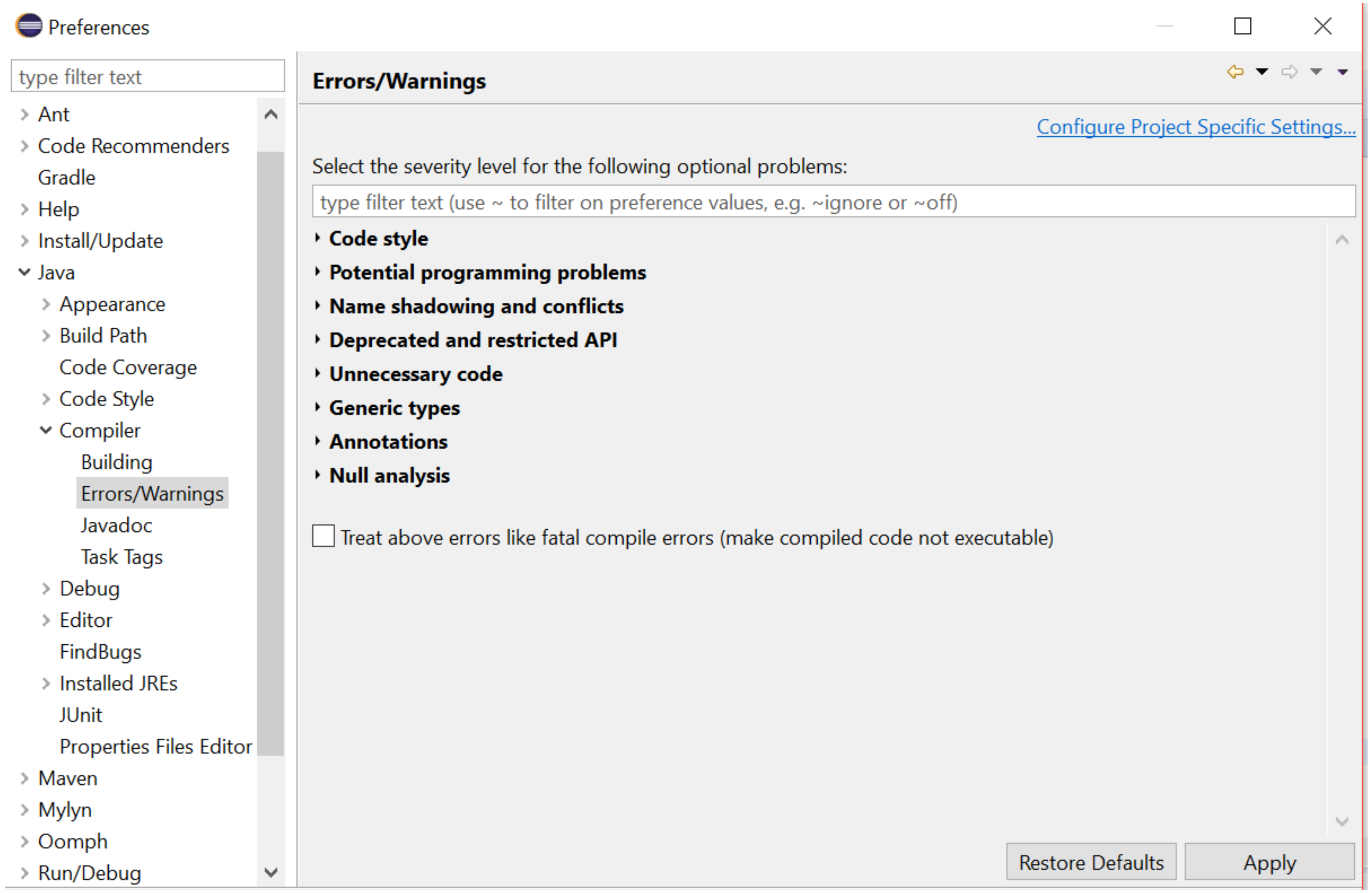
Some of benefits of static analysis:

- **Early Bug Detection**: By analyzing source code before it's executed or even compiled, static analysis tools can identify bugs, vulnerabilities, and other issues early on.
- **Security**: It's particularly useful for discovering security vulnerabilities in code that might be exploited by attackers, such as buffer overflows, SQL injection vulnerabilities.
- **Coding Standards Enforcement**: These tools can enforce coding standards and guidelines, ensuring that the codebase remains consistent and adheres to best practices.
- **Code Complexity Metrics**: Some static analysis tools measure the complexity of code, which can be a useful metric for understanding the maintainability and potential risks associated with certain components.

EXAMPLES OF WHAT STATIC ANALYSIS CAN CATCH

- **Type Mismatches**: Many languages have strong typing, and static analysis (often performed by the compiler in such languages) can identify when a variable of one type is used improperly with a variable of another type.
- **Use of Uninitialized Variables**: Accessing a variable before it's been initialized can be caught.
- **Dead Code**: Code that can never be executed under any circumstances.
- **Resource Leaks.**: For example, opening a file or database connection without closing it.
- **Concurrency issues:** arise when multiple threads access shared resources, potentially leading to unpredictable behavior if not handled properly.
- **Error Handling:** identify sections of the code that might throw exceptions but are not properly handled. E.g. Catch blocks that are empty.
- **Coding Standards and Style**

JAVA COMPILE ERRORS/WARNINGS



Preferences

type filter text

- Ant
- Code Recommenders
- Gradle
- Help
- Install/Update
- Java
  - Appearance
  - Build Path
  - Code Coverage
  - Code Style
  - Compiler
    - Building
    - Errors/Warnings
    - Javadoc
    - Task Tags
  - Debug
  - Editor
  - FindBugs
  - Installed JREs
  - JUnit
  - Properties Files Editor
- Maven
- Mylyn
- Oomph
- Run/Debug

Errors/Warnings

Configure Project Specific Settings...

Select the severity level for the following optional problems:

type filter text (use ~ to filter on preference values, e.g. ~ignore or ~off)

- **Code style**
- **Potential programming problems**
- **Name shadowing and conflicts**
- **Deprecated and restricted API**
- **Unnecessary code**
- **Generic types**
- **Annotations**
- **Null analysis**

☐ Treat above errors like fatal compile errors (make compiled code not executable)

Restore Defaults | Apply

https://help.eclipse.org/mars/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2Freference%2Fpreferences%2Fjava%2Fcompiler%2Fref-preferences-errors-warnings.htm

FINDBUGS

FindBugs: is a static analysis tool that examines your class or JAR files looking for potential problems by matching your bytecodes against a list of bug patterns. FindBugs divide defects in many categories:

- Correctness – gathers general bugs, e.g. infinite loops, inappropriate use of equals(), etc
- Bad practice, e.g. exceptions handling, opened streams, Strings comparison, etc
- Performance, e.g. idle objects
- Multithreaded correctness – gathers synchronization inconsistencies and various problems in a multi-threaded environment
- Malicious code vulnerability – gathers vulnerabilities in code, e.g. code snippets that can be exploited by potential attackers
- Security – gathers security holes related to specific protocols or SQL injections
- Dodgy – gathers code smells, e.g. useless comparisons, null checks, unused variables, etc

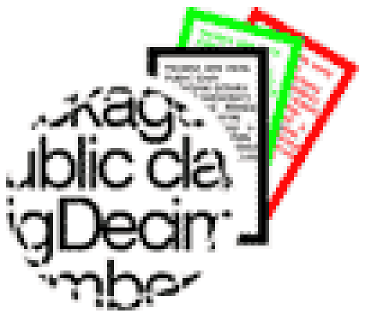
As of version 2, FindBugs started ranking bugs with a scale from 1 to 20 to measure the severity of defects:

- Scariest: ranked between 1 & 4.
- Scary: ranked between 5 & 9.
- Troubling: ranked between 10 & 14.
- Of concern: ranked between 15 & 20.

CHECKSTYLE

Checkstyle: A static analysis tool that focuses on Java coding style and standards.

- whitespace and indentation
- variable names
- Javadoc commenting
- code complexity
  - number of statements per method
  - levels of nested ifs/loops
  - lines, methods, fields, etc. per class
- proper usage
  - import statements
  - regular expressions
  - exceptions
  - I/O
  - thread usage, ...



SURPRISE: PRIMITIVE TYPES ARE NOT TRUE NUMBERS

One trap in Java – and many other programming languages – is that its primitive numeric types have corner cases that do not behave like the integers and real numbers we're used to. As a result, some **errors** that really should be dynamically checked **are not checked at all**. Here are the traps:

Integer division. 5/2 does not return a fraction, it returns a truncated integer. So this is an example of where what we might have hoped would be a dynamic error (because a fraction isn't representable as an integer) frequently produces the wrong answer instead.

Integer overflow. The int and long types are actually finite sets of integers, with maximum and minimum values. What happens when you do a computation whose answer is too positive or too negative to fit in that finite range? The computation quietly overflows (wraps around), and returns an integer from somewhere in the legal range but not the right answer.

Special values in floating-point types. Floating-point types like double have several special values that aren't real numbers: NaN (which stands for "Not a Number"), POSITIVE_INFINITY, and NEGATIVE_INFINITY. So when you apply certain operations to a double that you'd expect to produce dynamic errors, like dividing by zero or taking the square root of a negative number, you will get one of these special values instead. If you keep computing with it, you'll end up with a bad final answer.

ERROR CHECKING EXAMPLES

Let's try some examples of buggy code and see how they behave in Java.

Are these bugs caught as (a) static error (b) dynamic error (c) no error, wrong answer

1.
```
int n = 5;
if (n) {
  n = n + 1;
}
```

2.
```
int big = 200000; // 200,000
big = big * big;  // big should be 4 billion now
```

3.
```
double probability = 1/5;
```

4.
```
int sum =  7;
int n = 0;
int average = sum/n;
```

5.
```
double sum = 7;
double n = 0;
double average = sum/n;
```

6. double x = Math.sqrt(-1);

ARRAYS AND COLLECTIONS

let's change our hailstone computation so that it stores the sequence in a data structure, instead of just printing it out. Java has two kinds of list-like types that we could use: arrays and Lists.

Arrays are fixed-length sequences of another type T. For example, here’s how to declare an array variable and construct an array value to assign to it:

int[] a = new int[100];

The int[] array type includes all possible array values, but a particular array value, once created, can never change its length. Operations on array types include:

- indexing: a[2]
- assignment: a[2]=0
- length: a.length (note that this is different syntax from String.length() – a.length is not a method call, so you don’t put parentheses after it)

ARRAYS

```
int[] a = new int[100];        // <==== DANGER
int i = 0;
int n = 3;
while (n != 1) {
    a[i] = n;
    i++;  // very common shorthand for i=i+1
    if (n % 2 == 0) {
        n = n / 2;
    } else {
        n = 3 * n + 1;
    }
}
a[i] = n;
i++;
```

- Something should immediately smell wrong in this approach.
- What's that magic number 100?
- What would happen if we tried an n that turned out to have a very long hailstone sequence?

- It wouldn't fit in a length-100 array. We have a bug. Would Java catch the bug statically, dynamically, or not at all?

- Incidentally, bugs like these – overflowing a fixed-length array, which are commonly used in less-safe languages like C and C++ that don't do automatic runtime checking of array accesses – have been responsible for a large number of network security breaches and internet worms.

LIST

Instead of a fixed-length array, let's use the List type. Lists are variable-length sequences of another type T. Here's how we can declare a List variable and make a list value:

```
List<Integer> list = new ArrayList<Integer>();
```

And here are some of its operations:

- indexing: list.get(2)
- assignment: list.set(2, 0)
- length: list.size()

Note that List is an interface, a type that can't be constructed directly with new, but that instead specifies the operations that a List must provide. ArrayList is a class, a concrete type that provides implementations of those operations.

Note also that we wrote List<Integer> instead of List<int>. Unfortunately we can't write List<int> in direct analog to int[]. Lists only know how to deal with object types, not primitive types. In Java, each of the primitive types (which are written in lowercase and often abbreviated, like int) has an equivalent object type (which is capitalized, and fully spelled out, like Integer).

LIST

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

METHODS

In Java, statements generally have to be inside a method, and every method has to be in a class:

public means that any code, anywhere in your program, can refer to the class or method. Other access modifiers, like private, are used to get more safety in a program, and to guarantee immutability for immutable types.

static means that the method doesn't take a *this* parameter. Static methods can't be called on an object.

Hailstone.hailstoneSequence(83)

The comment before the method is a specification of the method, describing the inputs and outputs of the operation. The specification should be concise and clear and precise. The comment provides information that is not already clear from the method types.

It doesn't say, for example, that n is an integer, because the int n declaration just below already says that. But it does say that n must be positive, which is not captured by the type declaration but is very important for the caller to know.

```java
public class Hailstone {
    /**
     * Compute a hailstone sequence.
     * @param n  Starting number for sequence.  Assumes n > 0.
     * @return hailstone sequence starting with n and ending with 1.
     */
    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;
    }
}
```

MUTABLE VS. IMMUTABLE TYPE

MUTABLE VS. IMMUTABLE TYPE

Immutable

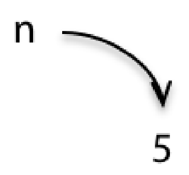
- An object value cannot be changed after it's created.
- If you want to "modify" an immutable object, you need to create a new instance.
- Examples such as String, primitive types and primitive wrapper classes (Boolean, Character, Byte, Short, Integer, Long, Float, Double).

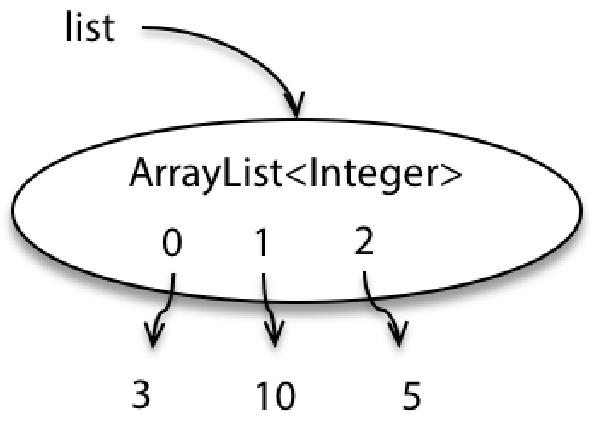
Mutable

- An object that can be changed after it's created.
- Examples StringBuilder, Arrays, ArrayList, and many others.

SNAPSHOT DIAGRAMS

Snapshot diagrams represent the internal state of a program at runtime. a snapshot diagram shows each variable with an arrow pointing to its value

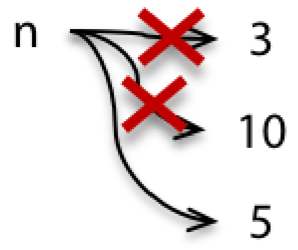




Snapshot diagram is a useful tool for understanding and visualizing the relationships between variables and the values they refer to.

When you assign to a variable, you're changing where the variable's arrow points. You can point it to a different value.

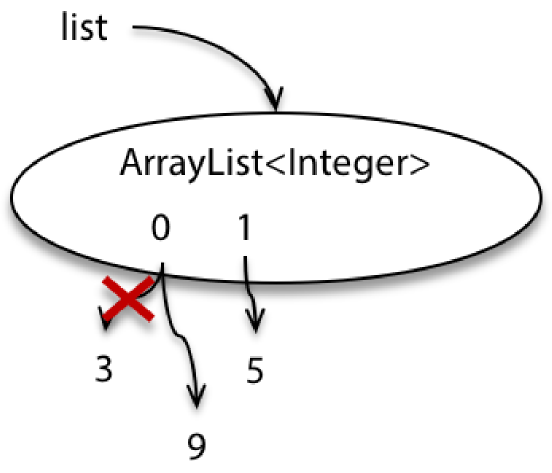
- int n = 3;
- n = 3*n + 1;
- n = n/2;



MUTATING VALUES

When you change the contents of a mutable value — such as an array or list — **you're changing references inside that value.** This is called mutating the value.

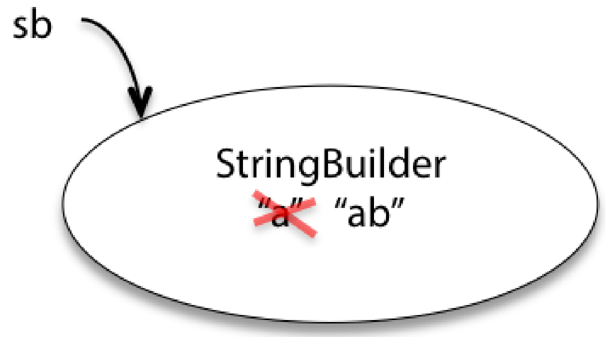
- List<Integer> list = new ArrayList<Integer>();
- list.add(3);
- list.add(5);
- list.set(0, 9);



Mutable values

StringBuilder (another built-in Java class) is a mutable object that represents a string of characters. It has methods that change the value of the object:

- StringBuilder sb = new StringBuilder("a");
- sb.append("b");



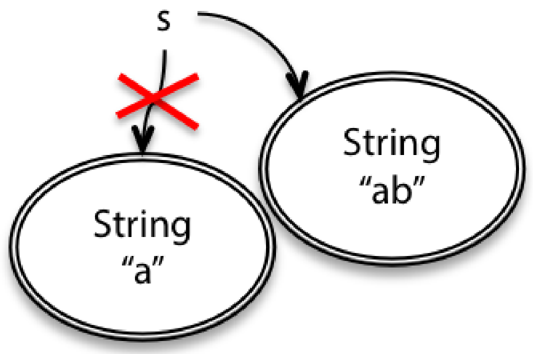
IMMUTABLE VALUES

Reassignment and immutable values

For example, if we have a String variable s, we can reassign it from a value of "a" to "ab".

- String s = "a";
- s = s + "b";

String is an example of an immutable type, a type whose values can never change once they have been created.



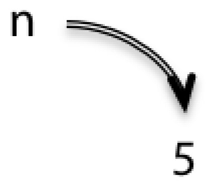
Immutable references

Java also gives us immutable references: variables that are assigned once and never reassigned. To make a reference immutable, declare it with the keyword final:

final int n = 5;

If the Java compiler isn't convinced that your final variable will only be assigned once at runtime, then it will produce a compiler error. So final gives you static checking for immutable references.

In a snapshot diagram, an immutable reference (final) is denoted by a double arrow.



IMMUTABLE REFERENCES

Notice that we can have an immutable reference to a mutable value. For example: final StringBuilder sb. This means that the StringBuilder value can change, but the sb variable's arrow can't change: it must always point to the same StringBuilder object.

```
final StringBuilder sb = new StringBuilder("Hello");

// This is allowed - changing the content of sb
sb.append(" World!");

// This is NOT allowed - reassigning the reference of sb
// sb = new StringBuilder("New String");
```

It's good practice to use final for declaring the parameters of a method and as many local variables as possible.

Like the type of the variable, these declarations are important documentation, useful to the reader of the code and statically checked by the compiler.

EXAMPLE ON FINAL

Suppose we want to add final to both variable declarations in the hailstoneSequence method, as shown:

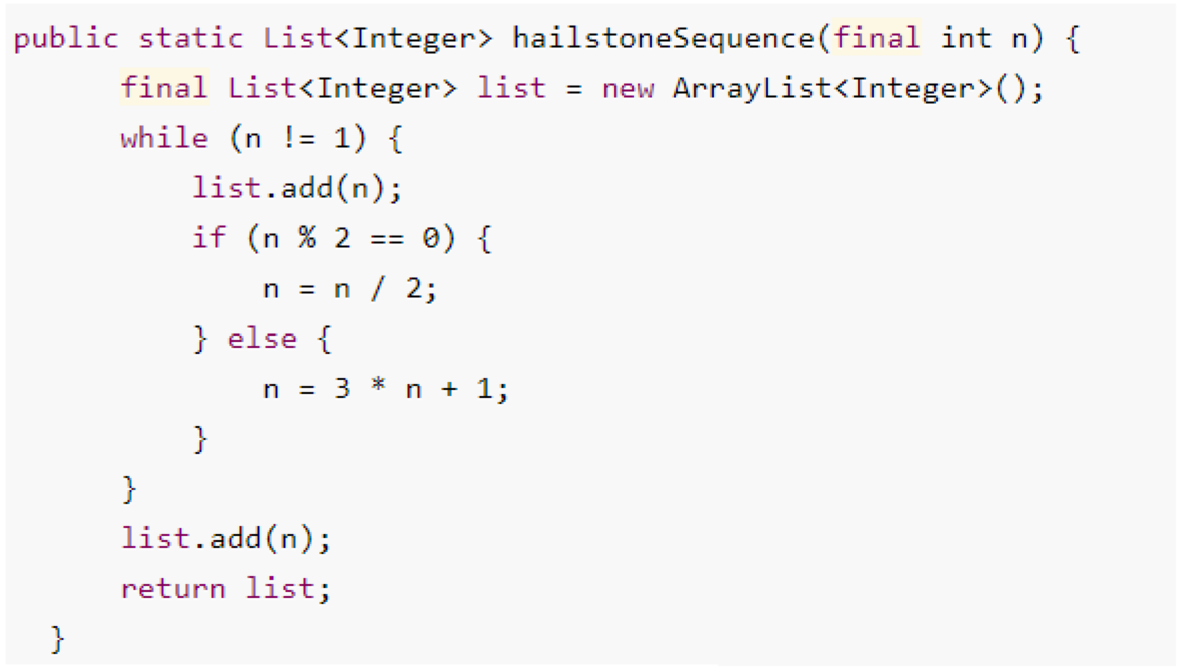
```
public static List<Integer> hailstoneSequence(final int n) {
      final List<Integer> list = new ArrayList<Integer>();
      while (n != 1) {
          list.add(n);
          if (n % 2 == 0) {
              n = n / 2;
          } else {
              n = 3 * n + 1;
          }
      }
      list.add(n);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

Which of the following are true statements about putting final on n?

- ☐ final can't be used on n because n is a parameter of a method
- ☐ final can't be used on n because its type is int and int is already immutable
- ☐ final can't be used on n because n is reassigned to other integer values in the body of the method
- ☐ final can be used on n, and it prevents n from being reassigned

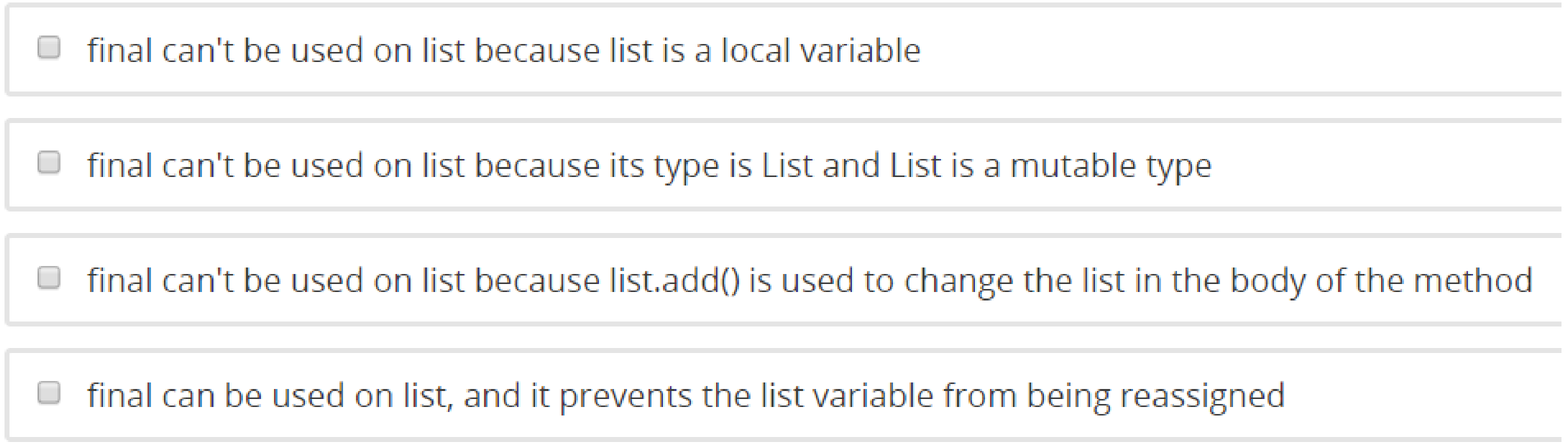
EXAMPLE ON FINAL

Suppose we want to add **final** to both variable declarations in the hailstoneSequence method, as shown:



```
public static List<Integer> hailstoneSequence(final int n) {
      final 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 of the following are true statements about putting final on list?



- ☐ final can't be used on list because list is a local variable
- ☐ final can't be used on list because its type is List and List is a mutable type
- ☐ final can't be used on list because list.add() is used to change the list in the body of the method
- ☐ final can be used on list, and it prevents the list variable from being reassigned

RISKS OF MUTATION

Mutable types seem much more powerful than immutable types. why on earth would you choose the immutable one? e.g. StringBuilder should be able to do everything that String can do, plus set() and append() and everything else.

The answer is **that immutable types are safer from bugs, easier to understand, and more ready for change**. Mutability makes it harder to understand what your program is doing, and much harder to enforce contracts. Here are two examples that illustrate why.

Let's start with a simple method that sums the integers in a list:

```
/** @return the sum of the numbers in the list */
public static int sum(List<Integer> list) {
    int sum = 0;
    for (int x : list)
        sum += x;
    return sum;
}
```

RISKY EXAMPLE #1: PASSING MUTABLE VALUES

Suppose we also need a method that sums the absolute values.

Following good DRY practice (Don't Repeat Yourself), the implementer writes a method that uses sum():

```java
/** @return the sum of the absolute values of the numbers in the list */
public static int sumAbsolute(List<Integer> list) {
    // let's reuse sum(), because DRY, so first we take absolute values
    for (int i = 0; i < list.size(); ++i)
        list.set(i, Math.abs(list.get(i)));
    return sum(list);
}
```

mutating the list directly

```java
// meanwhile, somewhere else in the code...
public static void main(String[] args) {
    // ...
    List<Integer> myData = Arrays.asList(-5, -3, -2);
    System.out.println(sumAbsolute(myData));
    System.out.println(sum(myData));
}
```

What will this code print? Will it be 10 followed by -10? Or something else?

RISKY EXAMPLE #2: RETURNING MUTABLE VALUES

Let’s consider Date, one of the built-in Java classes. Date happens to be a mutable type. Suppose we write a method that determines the first day of spring:

```
/** @return the first day of spring this year */
public static Date startOfSpring() {
    return askGroundhog();
}
```

```
// somewhere else in the code...
public static void partyPlanning() {
    Date partyDate = startOfSpring();
    // ...
}
```

If the code is rewritten to ask the groundhog at most once, and then cache the groundhog’s answer for future calls:

```
/** @return the first day of spring this year */
public static Date startOfSpring() {
    if (groundhogAnswer == null) groundhogAnswer = askGroundhog();
    return groundhogAnswer;
}
private static Date groundhogAnswer = null;
```

What just happened?

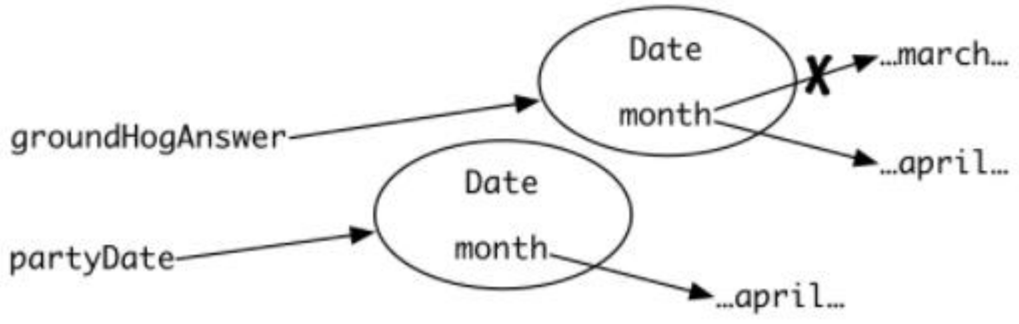
```
// somewhere else in the code...
public static void partyPlanning() {
    // let's have a party one month after spring starts!
    Date partyDate = startOfSpring();
    partyDate.setMonth(partyDate.getMonth() + 1);
    // ... uh-oh. what just happened?
}
```

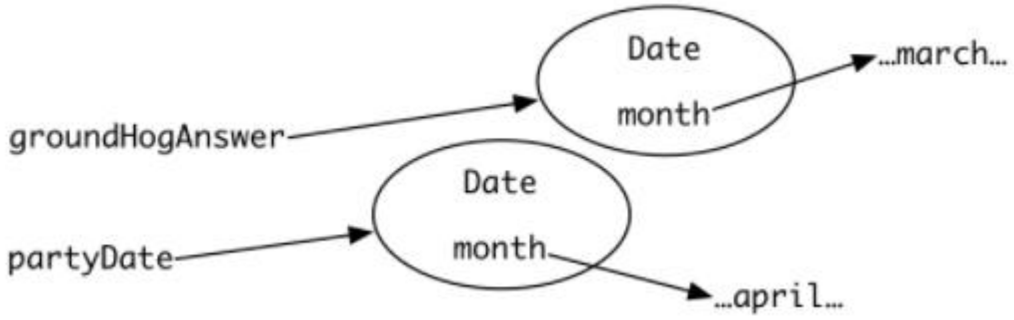
WHICH OF THESE SNAPSHOT DIAGRAMS SHOWS THE BUG?

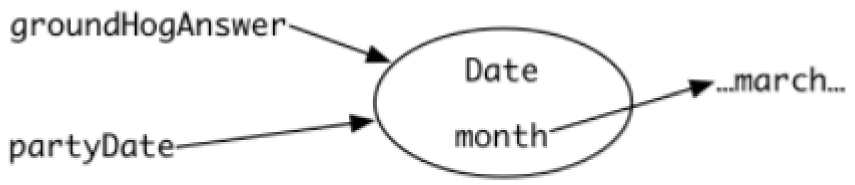
We don't know how Date stores the month, so we'll represent that with the abstract values ...march... and ...april... in an imagined month field of Date.

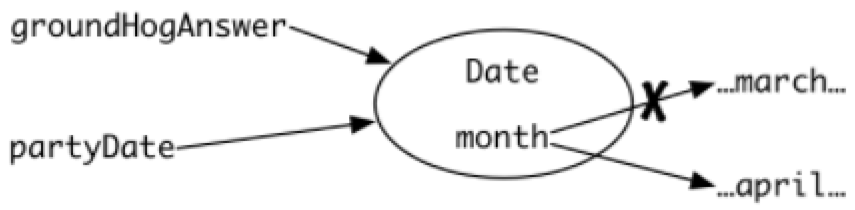
```java
/** @return the first day of spring this year */
public static Date startOfSpring() {
    if (groundhogAnswer == null) groundhogAnswer = askGroundhog();
    return groundhogAnswer;
}
private static Date groundhogAnswer = null;
```

```java
// somewhere else in the code...
public static void partyPlanning() {
    // let's have a party one month after spring starts!
    Date partyDate = startOfSpring();
    partyDate.setMonth(partyDate.getMonth() + 1);
    // ... uh-oh. what just happened?
}
```

1.


3.


2.


4.


ALIASING IS WHAT MAKES MUTABLE TYPES RISKY

Actually, using mutable objects is just fine if you are using them entirely locally within a method, and with only one reference to the object.

What led to the problem in the two examples we just looked at was having multiple references, also **called aliases , for the same mutable object.**

- In the List example, the same list is pointed to by both list (in sum and sumAbsolute) and myData (in main). One programmer (sumAbsolute 's) thinks it's ok to modify the list; another programmer (main 's) wants the list to stay the same. Because of the aliases, main 's programmer loses.
- In the Date example, there are two variable names that point to the Date object, groundhogAnswer and partyDate . These aliases are in completely different parts of the code, under the control of different programmers who may have no idea what the other is doing.

RISKS OF MUTATION

In both of these examples — the List<Integer> and the Date — the problems would have been completely avoided if the list and the date had been immutable types. The bugs would have been impossible by design.

This example also illustrates why using mutable objects can actually be bad for performance. The simplest solution to this bug, which avoids changing any of the specifications or method signatures, is for startOfSpring() to always return a copy of the groundhog's answer:

```
return new Date(groundhogAnswer.getTime());
```

ITERATING OVER ARRAYS AND LISTS

an **iterator** — a mutable object that steps through a collection of elements and returns the elements one by one.

Iterators are used **under the covers** in Java when you're using a for (... : ...) loop to step through a List or array. But you **must not** structurally **modify** the collection inside that loop.

```
List<String> lst = ...;
for (String str : lst) {
    System.out.println(str);
}
```

```
List<String> lst = ...;
Iterator iter = lst.iterator();
while (iter.hasNext()) {
    String str = iter.next();
    System.out.println(str);
}
```

What will happen when ty to remove list items?

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

you'll get a ConcurrentModificationException.

One Solution

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

USEFUL IMMUTABLE TYPES

The primitive types and primitive wrappers are all immutable. If you need to compute with large numbers, BigInteger and BigDecimal are immutable.

Don’t use mutable Dates, use the appropriate immutable type from java.time based on the granularity of timekeeping you need.

The usual implementations of Java’s collections types — List, Set, Map — are all mutable: ArrayList, HashMap, etc. The Collections utility class has methods for obtaining unmodifiable views of these mutable collections:

- Collections.unmodifiableList
- Collections.unmodifiableSet
- Collections.unmodifiableMap

You can think of the unmodifiable view as a wrapper around the underlying list/set/map. A client who has a reference to the wrapper and tries to perform mutations — add, remove, put, etc. — will trigger an UnsupportedOperationException.

Collections also provides methods for obtaining immutable empty collections: Collections.emptyList, etc. Nothing’s worse than discovering your definitely very empty list is suddenly definitely not empty!

DOCUMENTING ASSUMPTIONS

Writing the type of a variable down documents an assumption about it: e.g., this variable will always refer to an integer. Java actually checks this assumption at compile time, and guarantees that there's no place in your program where you violated this assumption.

Declaring a variable final is also a form of documentation, a claim that the variable will never change after its initial assignment. Java checks that too, statically.

We documented another assumption that Java (unfortunately) doesn't check automatically: that n must be positive.

Why do we need to write down our assumptions? Because programming is full of them, and if we don't write them down, we won't remember them, and other people who need to read or change our programs later won't know them. They'll have to guess.

Programs have to be written with two goals in mind:

- **communicating with the computer**. First persuading the compiler that your program is sensible – syntactically correct and type-correct. Then getting the logic right so that it gives the right results at runtime.
- **communicating with other people**. Making the program easy to understand, so that when somebody has to fix it, improve it, or adapt it in the future, they can do so.

DOCUMENTING ASSUMPTIONS

```
from math import sqrt
def funFactAbout(person):
  if sqrt(person.age) == person.age:
    print("The age of " + person.name + " is a perfect square: " + str(person.age))
```

If you were writing Java instead of Python, which of them could be documented by type declarations by the Java compiler?

- ☐ `person` must be an object with `age` and `name` instance variables
- ☐ `person` is not `null`
- ☐ `person.age` must be a nonnegative number
- ☐ `person.age` must be an integer
- ☐ `person.name` must be a string