CS 228: Introduction to Data Structures
Lecture 8

`static` Fields and Methods

The keyword **`static`** in Java means “associated with the class as a whole, not with an instance”. Fields and methods can be static.

A ***static field*** is a single variable shared by a whole class of objects; its value does not vary from object to object. Thus, static fields are also called ***class variables***. If we declare a field `static`, there is just one field for the whole class. One common use of static fields is to define constants, such as `Math.PI`, that are **`static`** and **`final`**. Here is another example.

Example. Suppose we want to keep track of the number of `Person` objects that we have constructed. It does not make sense for each object to have its own copy of this number: we would have to update every `Person`’s number whenever a new `Person` is created. It makes more sense to have a single variable, a static field, for the entire class that counts the number of people created thus far. The constructor increments this static field, called `numberOfPeople`, by one.

```
class Person {
  public static int numberOfPeople;
  public String name;
  public Person(String name) {
    this.name = name;
    numberOfPeople++;
  }
}
```

If we want to look at the variable `numberOfPeople` from another class, we write it in the usual notation, but we prefix it with the class name rather than the name of a specific object. For example,

```
1.  int kids = Person.numberOfPeople / 4;
```

A ***static method*** does not implicitly pass an object as a parameter — in contrast, for example, the call `p.foo(q)` implicitly passes p as a parameter to `foo`. Thus, a static method can be used without creating an instance of the class. One example is `Math.cos()`. Here's another one.

```
class Person {
  ...
  public static void printPopulation() {
    System.out.println(numberOfPeople);
  }
}
```

Now, we can call "`Person.printPopulation()`" from another class.

The `main()` method is always `static`, because when we run a program, we are not passing it an object.

Important: In a `static` method, there is no "`this`"! Any attempt to reference "`this`" will cause a compile-time error.

Notes

Instead of statement 1 in the previous page, we could write the following.

```
2.  int kids = joe.numberOfPeople / 4;
```

Surprisingly, statement 2 works as well! Note, though, that `numberOfPeople` has nothing to do with `joe` specifically, and, in fact, `joe` will ***not*** be passed along as "`this`". Thus, although statement 2 works, it is bad style, because it confusing.

Note that you can also call `joe.printPopulation()` instead of `Person.printPopulation()`, but this is also bad style.

Lifetimes of Variables

- A local variable — i.e., one declared within a method — is gone forever as soon as the method in which it is declared finishes executing. (If it references an object, the object might continue to exist, though.)
- An instance variable — i.e., a non-static field — lasts as long as the object exists. An object lasts as long as there's a reference to it.
- A class variable — i.e., a static field — lasts as long as the program runs.

Exceptions

An ***exception*** is any special condition that alters the normal flow of program execution, such as attempting to divide by zero or to open a file that does not exist. These errors can cause your program to terminate immediately. Your perspective on exceptions depends on whether you are API ***developer*** or an API ***client***.

- As a ***developer***, your main concern is to ***throw*** the right exception at the right time. This is often just a matter of putting in a `throw` statement and creating the right `Exception` object.

- As a ***client***, you may have to ***handle*** the exceptions thrown by methods you call. This is done with a `try`/ `catch` block. This prevents an error message from printing and the program from terminating.

Exceptions are a way to escape a “sinking ship”. By letting the system throw an exception, or by throwing your own, you can move program execution out of a method whose purpose has been defeated. Throwing an exception is different from a `return` statement because

- You don't have to return anything.

- An exception can fly several stack frames down the stack, not just one.

Catching Exceptions

To handle an exception, you have to ***catch*** it. As an example, here is a program that tries to open a file named by the first command-line argument.

```
public static void main(String[] args)
{
   InputStream istream;
   File        inputFile;

   try {
      inputFile = new File(args[0]);
      istream = new InputStream(inputFile);
   }
    catch (FileNotFoundException ex) {
        System.out.println("file " +
           args[0] + " not found");
    }
}
```

If the user runs the program with a bad file name `foo`, the program will print the message

```
file foo not found
```

and then halt. Note that the `try/catch` for the `FileNotFoundException` is needed for the program to compile: `FileNotFoundException` is what is known as a “checked exception”. If the compiler sees the possibility that a checked exception may be thrown, it will signal an error if the program either does not handle it or does not list that exception as one that might be thrown.

Comments

- The program should probably take the additional precaution of checking that there *is* a command-line argument before attempting to use `args[0]`.

- It probably makes more sense to put the `try` block in a loop, so that if the file is not found the user can be asked to enter a new file name.

In general, we catch exceptions using `try/catch` blocks:

```
try {
   // statements that might cause
   // exceptions
}
catch ( exception-1 id-1 ) {
   // statements to handle this exception
}
catch ( exception-2 id-2 ) {
   // statements to handle this exception
}
   ⋮
```

```
finally {
   // statements to execute every time this
   // try block executes
}
```

This code does the following:

1. It executes the code inside the `try` braces.

2. If the `try` code executes normally, it skips over the `catch` clauses.

3. If the `try` code throws an exception, it jumps directly (without finishing the `try` code) to the first `catch` clause that ***matches*** the exception, and executes that `catch` clause. There is a ***match*** if the actual exception object thrown is the same class as, or a subclass of, the exception type listed in the `catch` clause.

4. When the `catch` clause finishes executing, the `finally` clause, if there is one (it is optional), is executed, ***whether or not there was an exception***. A `finally` clause is typically used to ensure that some clean-up (e.g., closing opened files) is done. If there is no `finally` clause, we go to the next step.

5. Java jumps to the next line of code immediately after the `try/catch` block.

The code within a `catch` clause is called an ***exception handler***. Note that you don't need a `catch` clause for every exception that can occur. You can catch some exceptions and let others propagate. If an exception propagates all the way out of `main()` without being caught, the Java Virtual Machine prints an error message and halts.

In general, there can be one or more catch clauses. If there is a `finally` clause, there can be zero `catch` clauses.

Further details (not covered in class). An exception may occur in a `catch` or `finally` clause. An exception thrown in a `catch` clause will terminate the `catch` clause, but the `finally` clause will still get executed before the exception goes on. An exception thrown in a `finally` clause replaces the old exception, and terminates the `finally` clause and the method immediately. However, you can nest a `try` clause inside a `catch` or `finally` clause, thereby catching those exceptions as well.

If a `finally` clause includes a transfer of control statement — i.e., a `return`, `break`, `continue`, or `throw` — then that statement overrides any transfer of control initiated in the `try` or in a `catch` clause.

First, let's assume that the finally clause does not include any transfer of control. Here are the situations that can arise:

1. No exception occurs during execution of the `try`, and the `try` does no transfer of control.

   ⇒ The `finally` clause executes, then the statement following the `try` block.

2. No exception occurs during execution of the `try`, but it does execute a transfer of control.

   ⇒ The `finally` clause executes, then the transfer of control takes place.

3. An exception occurs during execution of the `try`, and there is no `catch` clause for that exception.

   ⇒ The `finally` clause executes, then the uncaught exception is passed up to the next enclosing `try` block, possibly in a calling method.

4. An exception occurs during execution of the `try`, and there is a `catch` clause for that exception. The `catch` clause does not execute a transfer of control.

   ⇒ The `catch` clause executes, then the `finally` clause, then the statement following the try block.

5. An exception occurs during execution of the `try`, there is a `catch` clause for that exception, and the `catch` clause does execute a transfer of control.

   ⇒ The `catch` clause executes, then the `finally` clause, then the transfer of control takes place.

If the `finally` block ***does*** include a transfer of control, then that takes precedence over any transfer of control executed in the `try` or in an executed catch clause. So for all of the cases listed above, the finally clause would execute, then its transfer of control would take place. Here's one example:

```
try {
    return 0;
}
finally {
    return 2;
}
```

The result of executing this code is that 2 is returned. Note that this is rather confusing! The moral is that you probably do not want to include transfer-of-control statements in both the try statements and the `finally` clause, or in both a catch clause and the `finally` clause.

Checked and Unchecked Exceptions

Every exception is either a ***checked*** exception or an ***unchecked*** exception. If a method has instructions that could cause a ***checked*** exception to be thrown, then the method must acknowledge the possibility that such a checked exception may be thrown. This is done by either

- putting a `throws` clause in the method header to declare that the exception could be thrown, or

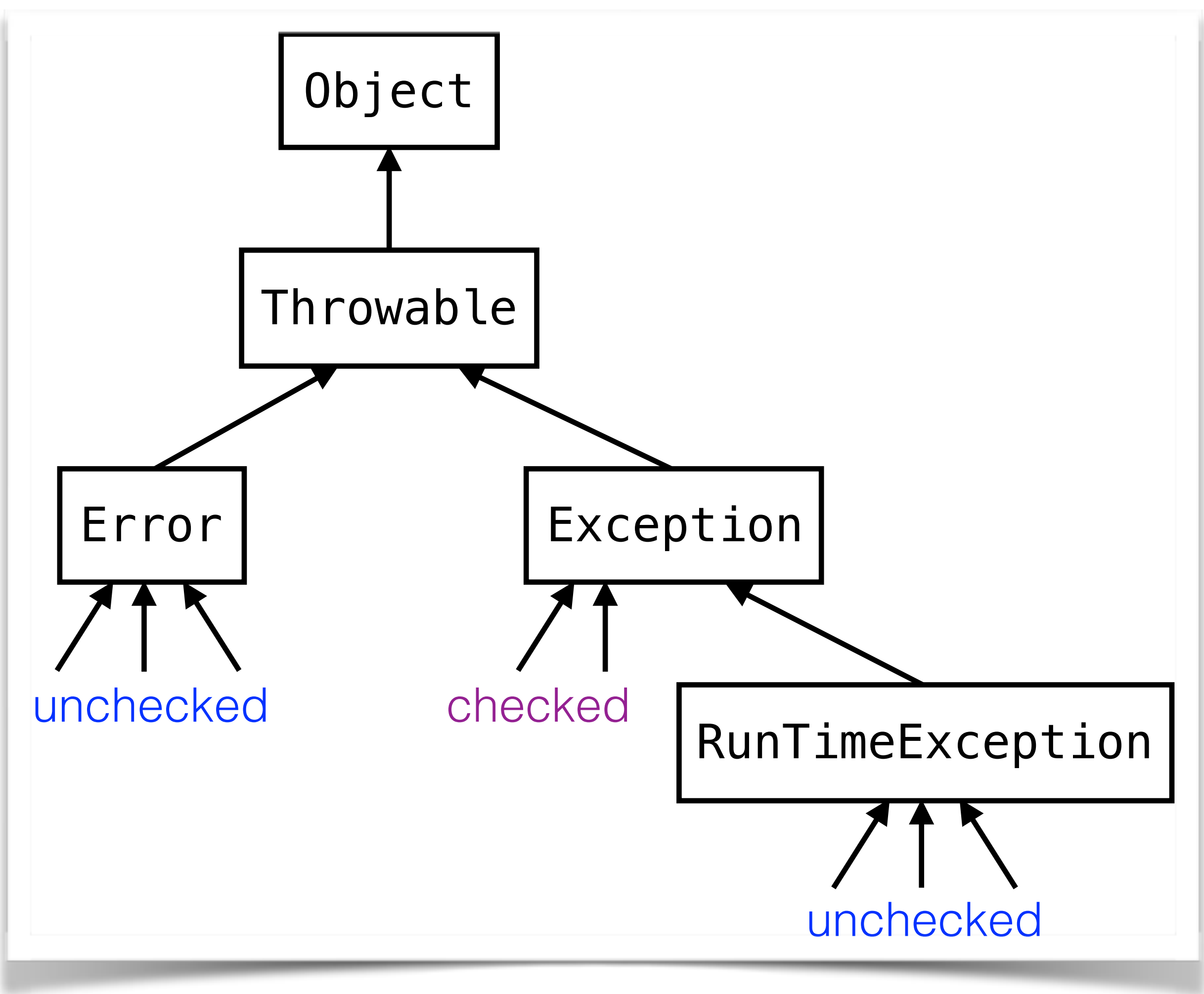
- enclosing the code that might cause the exception within a `try` block with a `catch` clause for that exception.

If you fail to do this, you will get a compiler error.

The `Exception` Hierarchy

Exceptions in Java are objects belonging to the `Exception` class, which is itself a subclass of `Throwable`, the class of things that you can `throw` and `catch` (see the figure). The other subclass of Throwable is `Error`. An `Error` generally represents a fatal error, like running out of memory or stack space. Although you can throw or catch any kind of `Throwable`, catching an `Error` is rarely appropriate.

Most `Exceptions`, unlike `Errors`, signify problems you could conceivably recover from. The subclass `RunTimeException` is made up of exceptions that might be thrown by the Java Virtual Machine, such as `NullPointerException`, `ArrayIndexOutOfBoundsException`, and `ClassCastException`.



Exception Constructors

Java exceptions are objects; they are defined by defining a class, for example:

```
public class EmptyStackException
extends Exception { }
```

New exceptions are usually subclasses of `Exception`, so that they are checked. The exceptions you define do not have to be public classes; however, remember that if you do not make them public, then they can only be used in the package in which they are defined.

By convention, most `Throwables` (including `Exceptions`) have two constructors. One takes no parameters, and one takes an error message in the form of a `String`.

```
class MyException extends Exception {
  public MyException() { super(); }
  public MyException(String s)
  { super(s); }
}
```

The error message will be printed if it propagates out of `main()`. The constructors usually call the superclass constructors, which are defined in `Throwable`.

Throwing Exceptions

To throw an exception, we use a `throw` statement:

```
public class Stack {
   ...
   public Object pop() throws
   EmptyStackException
   {
      if (Empty())
         throw new EmptyStackException();
      ...
   }
}
```

Observe that

- since exceptions are objects, you must use `new` to create an `Exception` object, and
- since the `pop` method might throw the (checked) exception `EmptyStackException`, that must be indicated in `pop`'s `throws` clause.

Introduction to the Analysis of Algorithms

An ***algorithm*** is a step-by-step method for solving a problem. A well-specified algorithm can be directly translated into working code; however, an algorithm is independent of the programming language used to code it. For example, consider the following problem.

ARRAY EQUALITY

Input: Two arrays A and B, of the same length and without duplicates.
Question: Do A and B contain the same elements?

Here is pseudocode for two possible algorithms.

Algorithm 1

```
for each position i in array A
    if element A[i] does not appear in array B
        return false
return true
```

Algorithm 2

```
Sort A
Sort B
for each position i
    if A[i] is different from B[i]
        return false
return true
```

Which algorithm is better? What does “better” mean for an algorithm? We will focus on comparing relative speeds. In the next few lectures, we will see how to analyze an algorithm to determine its ***time complexity***, a measure of how the algorithm’s running time *scales* as a function of the size of its input. We will sketch the main ideas of algorithm analysis using the two algorithms above. First, we need to decide on two things.

(a) What is our unit for measuring time?

(b) How do we measure the size of the input?

Algorithms 1 and 2 are based on comparisons, so we will measure the time complexity of each algorithm in terms of how many comparisons executes. For the array equality problem, a good measure of the input size is the number of items in each array, which we denote by n.

Analysis of Algorithm 1. Suppose we implement line 2 using a sequential scan of B. That is, we successively compare `A[i]` with `B[0]`, `B[1]`, `...`, `B[n−1]`, stopping as soon as there is a mismatch or we have exhausted all entries of B. The worst-case number of comparisons for each such sequential scan is n. Since the **for** loop of lines 1—3 iterates at most n times, the worst-case running time is proportional to (i.e., scales like) $n \times n = n^2$. Formally, we say that the time complexity of Algorithm 1 is $O(n^2)$ (read "big-O of n^2"). We will explain this notation next time.

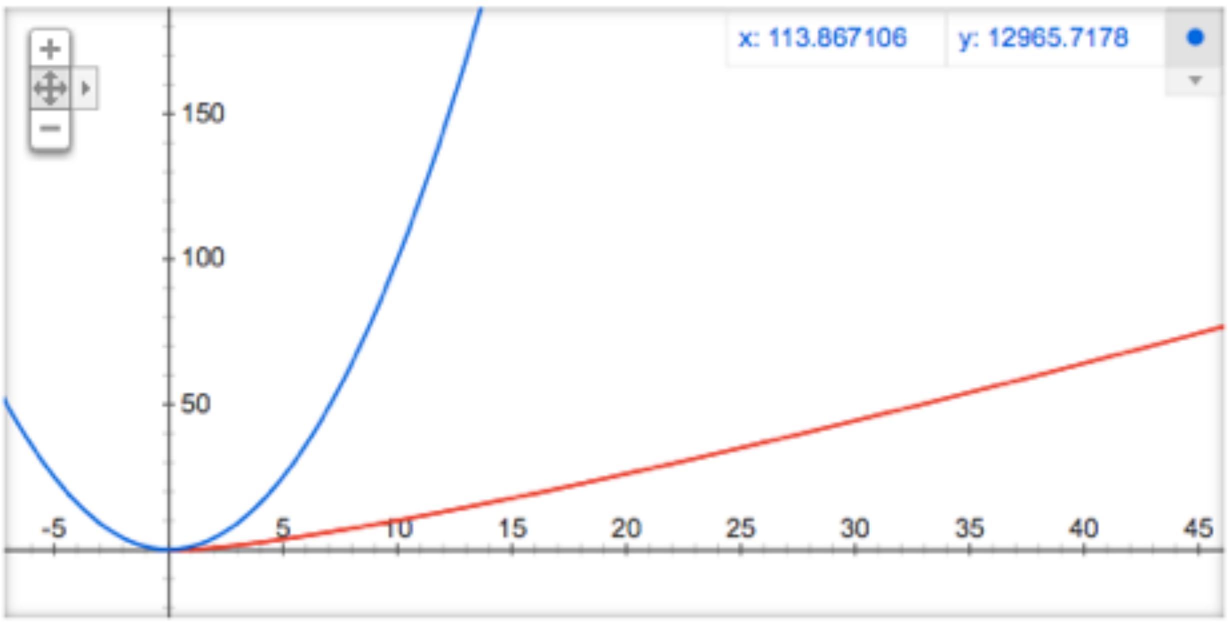
Analysis of Algorithm 2. For a moment, let us ignore lines 1 and 2. Then, the **for** loop of lines 3—5 iterates at most n times, and each iteration does only one comparison. Thus, the time taken by the loop is proportional to n in the worst case. As we will see later this semester, it is possible to sort an n-element array using no more than $n \log_2 n$ comparisons, where is the logarithm[1] to base 2 of n. Hence we can perform in lines 1 and 2 in time proportional to $n \log_2 n$, and the total running time of the entire algorithm scales like $2\, n \log_2 n + n$. The

---

[1] We will review logarithms next week. For now, you can consult Wikipedia.

dominant term is $n \log_2 n$, so we can say that the algorithm runs in time proportional to n log n. Formally, we say that the time complexity of Algorithm 2 is $O(n \log n)$ (read "big-O of n log n").

Which algorithm is faster? The graph below[2] shows that $T_1(n) = n^2$ grows more quickly than $T_1(n) = n \log_2 n$.



Thus, the larger n is, the better Algorithm 2 is compared to Algorithm 1.

Comment. The rough analysis we just performed ignores several details. For example, the true worst-case running time of Algorithm 2 is more like $2 n \log_2 n + n$. So, why do we disregard the n term and the factor of 2 in the $2 n \log_2$

[2] The plot was generated using Google.

n term to get $T_1(n) = n \log_2 n$? We will see the reason next week.