EXCEPTIONS

Introduction

You've probably run a program that crashed and gave you a message about an "exception" such as:

- `ArithmeticException`: for example, if you try to divide by zero.
- `ArrayIndexOutOfBoundsException`: if you try to access an element beyond the beginning or end of an array.
- `NullPointerException`: if you try to access a member of an object through a `null` reference.

We're going to examine:

- how exceptions work
- when to *write* code that "throws" (signals) them
- how to *use* code that throws them

A program

```
public class AbuseArray {
    /** Set the first n elements of a to v.
        Requires: 0 <= n < a.length. */
    public static void init(int[] a, int n, int v) {
        for (int i=0; i != n; i++) {
            a[i] = v;
        }
    }

    public static void main(String[] args) {
        int[] myArray = new int[10];
        init(myArray, 25, -1);
    }
}
```

What happens when we run it?

```
Exception in thread "main"
    java.lang.ArrayIndexOutOfBoundsException
        at AbuseArray.init(AbuseArray.java:6)
        at AbuseArray.main(AbuseArray.java:12)
```

An exception is an unexpected event, interrupting the normal flow of control.

Think of this as a violation of a contract: something happened that prevented the code from meeting its specification.

Where Do Exceptions Come From?

Java has a class called `Exception`, with descendants like `ArrayIndexOutOfBoundsException` for more specific kinds of problems. You can also define your own descendants.

When something goes wrong, a called method can construct an instance of the appropriate kind of exception and throw it (using a `throw` statement, which we'll see later).

This causes the called method to abort immediately, without returning a value. The *calling* method then has a choice of dealing with the exception, or else it also aborts to the method that called it, and so on.

This game of "hot potato" continues until one of the methods on the call stack deals with the exception; otherwise it lands in the user's lap.

An Approach to Exception Handling

What do we mean by "dealing" with an exception? If we call a method and get an exception, then:

- we violated the method's preconditions, or
- the method, or a method called during execution of the method, violated some method's preconditions, or
- the method (or a method called during its execution) noticed it couldn't ensure its postconditions

Questions

Do we prepare for the worst every single time we call a method?

How do we 'recover'? Isn't that harder than writing correct code in the first place?

These are difficult and subtle questions, which we mostly avoid in this course.

Our Approach

In your assignments for this course, you will usually be expected to use exceptions to produce good debugging information.

Signalling Queue Overflow with Exceptions

With both `ArrayQueue` and `CircularQueue`, the array is of a fixed size. What if one calls `enqueue(Object)` on a full queue?

We might get an `ArrayIndexOutOfBoundsException`, or wrong values later when we call `dequeue()`.

Let's make sure the error gets noticed immediately, and the exception has a more descriptive name.

Defining our own kind of exception

Here's how:

```
public class QueueFullException
    extends Exception {}
```

That's it! There's no need to write any methods.

Rewriting `CircularQueue`'s `enqueue`

```
/** Append o to me. Throw a QueueFullException if
    there is no room because I am already full. */
public void enqueue(Object o)
                throws QueueFullException {

    if (size == contents.length) {
        throw new QueueFullException();
    }
    tail = (tail+1) % contents.length;
    contents[tail] = o;
    ++size;

}
```

Now when a program calls `enqueue` on a full `CircularQueue`, the program automatically stops and reports a `QueueFullException` to the user.

Reacting to an exception

Sometimes we may want to proceed anyway. For example, if we're queueing up requests we might simply want to ignore the rest.

To have our program react to an exception, we use a "`try`" block and "`catch`" the exception.

Catching an Exception

```
/** Add Strings (representing requests) from in to q,
    until the String "done", or the queue is full. */
public static void getRequests(Queue q, BufferedReader in)
                       throws IOException {
    String s = in.readLine();
    while (!s.equals("done") {
        try {
            q.enqueue(s);
            s = in.readLine();
        } catch (QueueFullException qfe) {
            return;
        }
    }
}
```

The `catch` block is skipped entirely if nothing goes wrong.

Question: Why bother with the `try` block? Won't the method return anyway if an exception occurs?

Exercise: Define a new type of exception: `TruncatedException`. Make `getRequests` throw it if the queue gets full, instead of proceeding.

Exception objects

Exceptions are objects, and have (at least) the members defined in `Exception`. We could have added more members to `QueueFullException`.

In a `catch` clause we can use the exception like a parameter.

When we construct an `Exception`, we can give it a string as a message. A `catch` clause can access the message via `getMessage()`. If the exception isn't caught, the message gets printed automatically.

A good example of this is in the Java API: if we call `add(Component)` on a `JFrame`, we get an exception with the helpful error message:

```
Do not use javax.swing.JFrame.add() use
 javax.swing.JFrame.getContentPane().add() instead
```

Exercise: Make `CircularQueue` put a message into the exception it throws, and `getRequests()` print it out.

IOException

If `readLine()` throws an `IOException` in the `try` block of `getRequests(Queue, BufferedReader)`, the exception won't get caught there and will propagate to the calling method as usual. It would be better style to move `readLine()` outside the `try` block.

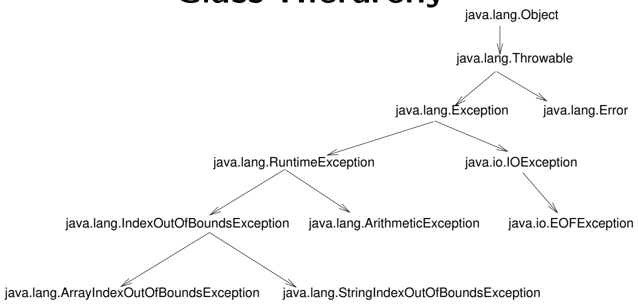
So `getRequests` might throw an `IOException`, and must declare this in its header with "`throws IOException`".

The header informs callers of `getRequests` that they might get an `IOException`; those callers must catch the exception or declare that they might throw an `IOException`.

Notice however that even though `q` or `in` could be `null` and thus `getRequests` could throw a `NullPointerException`, we don't have to declare this fact.

Why not?

Part of the Java API Class Hierarchy



A method is *not* required to catch (or declare that it might throw) any kind of `RuntimeException` or `Error`.

`NullPointerException` is a `RuntimeException`.

Okay, but why is it a `RuntimeException`?

Preventable Exceptions: `RuntimeException`

Writing `try` blocks, or declaring all exceptions we might throw (even ones we're pretty sure we won't) is time-consuming and buries the main logic of the program under all the exception code.

For people who use exceptions mainly for debugging, it's of even less use.

We want the compiler to assume that we're paying attention to the preconditions of methods we're calling. We shouldn't have to tell it that we're doing so every single time.

Question: We didn't make `QueueFullException` a `RuntimeException`. What changes does this force us to make to our programs?

Exercise: Add an `isFull()` method to `Queue`, modify `enqueue()` appropriately, and change `QueueFullException` to be a `RuntimeException`.

Non-`Runtime Exception`s and `Error`s

But there are still exceptions that are impossible or too inefficient to prevent.

For example when we call `readLine()`, we can't know if the user is going to rip the keyboard out of the computer in the middle of typing the line.

So we must prepare to deal with them in a `try` block, or tell the compiler that we don't care.

But what if there's no reasonable way to deal with a particular problem?

For example, what if the program runs out of memory? Then we probably don't even have memory space to run code that tries to deal with the problem.

These kinds of problems are signalled by throwing an `Error`.

More about Catching

You can have multiple `catch` clauses.

```
/** Add each line of input (representing requests)
    from in to q, until the String "done", or
    the queue is full, or an IOException occurs. */
public static void getRequests(Queue q,
                               BufferedReader in) {
    try {
        String s = in.readLine();
        while (!s.equals("done") {
            try {
                q.enqueue(s);
                s = in.readLine();
            } catch (QueueFullException qfe) {
                return;
            }
        }
    } catch (EOFException e) {
    } catch (IOException e) {
        System.out.println(
            "Problems reading input! " +
            "Processing previous requests.");
    }
}
```


At most one `catch` block is executed — the first one that matches.

Question: How could more than one ever match?

Notice that we removed "`throws IOException`". It's worth noting that the original version could have said "`throws Exception`"; the compiler assumes that (an instance of) any *descendant* of the listed class(es) might be thrown.

Question: Why was it better not to write "`throws Exception`" in the original version?

We could even leave the clause in, because the compiler doesn't assume that the class(es) *will* be thrown. It just cares that we've safely covered all those that *might* be thrown.

Question: Why would leaving the clause in be a bad idea?

Summary: When to Use Exceptions

Remember that an exception indicates a contract violation.

It occurs when something prevented the code from meeting its specification. There are two possibilities:

- The client failed to ensure that the preconditions were met before calling a method.
- The preconditions were met, but something else prevented the method from making its postconditions true.

These are the only times when you should throw an exception.

Often, an existing Java exception class is appropriate, so you can just throw an instance of it. If not, define your own exception class.

Aren't preconditions assumed to be true?

When a precondition hasn't been met, the contract leaves us under no obligation; we can do whatever we want — even crash.

But if we want to be kind to the client programmer, we can check the precondition and throw an exception.

This will show exactly where things went wrong; otherwise, an error can propagate for a while before it becomes apparent.

Avoiding inefficiency

Checking preconditions isn't kind to the client if it ruins the efficiency of your method. Can you think of an example where this would happen?

In some languages, checks can be turned on during debugging and testing and then turned off again when the code goes into use.

In other languages, a reasonable choice is to only check a precondition if it can be done in "constant time" (more on that later).