Error Handling

Many kinds of error conditions can occur during execution:

- invalid parameters passed to methods
- undefined operations: null pointer, array index out of bound, empty stack, etc.
- failures: disk full, cannot connect to remote host, etc.

We need a mechanism to flag errors.

Conventional way: return error codes, use a global "last error type" variable. Problems:

- users forget or refuse to check for errors
- checking for errors easily leads to messy code
- normal return values conflict with error codes

Error Handling

For example, recall `IntQueue` from the assignment.

```
public class IntQueue
{
//...
  public int peek() {
    return head.value;
    // what if queue is empty?
  }

  public int unqueue() {
    // ditto
  }
}
```

The assignment did not specify what `peek` and `unqueue` should do if the queue is empty.

What would be a sensible specification? (A special return code is not an option here.)

Exceptions

You would *throw exceptions*.

Exceptions are objects of type `Throwable`.

An important subclass is `Exception`. Most exception types, especially those you define, should be its subclasses.

Define an exception class for empty queues:

```
public class IQEmptyException extends Exception
{
  // implicit default constructor
}
```

Throw a new instance of it when unqueuing an empty queue:

```
public int unqueue() throws IQEmptyException {
  if (head == null)
    throw new IQEmptyException();
  // ...
  return x;
}
```

Flow of control: the `throw` statement stops `unqueue` immediately, and the caller receives the exception.

(Not) Handling Exceptions

A caller of `unqueue` must handle `IQEmptyException`.

Generally, if a method says it `throws` some exceptions, the caller must handle them.

One way is to just let the the exception propagate through. This is like you are throwing the exception.

```
public int foo(IntQueue q) throws IQEmptyException {
  int x;
  x = 10 * q.unqueue();
  x = x + q.unqueue();
  return x;
}
```

Here, `foo` throws the same exception received from `unqueue`. Callers of `foo` will then have to receive the exception and handle it.

Flow of control: if one of the calls to `unqueue` stops with the exception, `foo` receives it and also stops immediately, and `foo`'s caller receives the exception.

Handling Exceptions

You may also *catch* an exception.

```
public void bar(IntQueue q) {
  try {
    int y;
    y = foo(q);
    y = y * q.unqueue();
    if (q.empty()) throw new IQEmptyException();
    y = y + q.unqueue();
    System.out.println(y);
  }
  catch (IQEmptyException e) {
    System.out.println(e.toString());
  }
  System.out.println("bar done");
}
```

Flow of control: if the exception occurs, the `try` block stops immediately, and execution restarts at the `catch` block.

Handling Exceptions

Generally, the `try-catch` statement looks like this:

try { *statements* }
catch (*exception_type var*) { *statements* }
catch (*exception_type var*) { *statements* }
etc.

There can be zero or more `catch` blocks.

If no exception occurs in the `try` block, it executes to completion, the `catch` blocks are skipped, and execution continues afterwards.

If an exception occurs, it is matched against the `catch` blocks with `instanceof`. The *first* matching block will get executed, and the other ones are skipped.

If there is no match, the exception is propagated as normal. Hopefully some outer `try-catch` statement will catch it, or the method declares it in the `throws` clause.

Handling Exceptions

Usual things you can do in a `catch` block:

- do nothing
- print an error message
- if a computation was aborted, provide a default or special result
- re-throw the same exception
- throw your own exception

If a `catch` block throws an exception, it is *not* caught by the same `try-catch` statement. It is thrown like normal. Some outer `try-catch` statement or the method's `throws` clause will have to deal with it.

Exercises: Why would you re-throw the exception? Why would you throw your own exception?

Cleaning Up

The `try-catch` statement can be optionally followed by a `finally` statement:

try { *statements* }
catch (*exception_type var*) { *statements* }
etc.
finally { *statements* }

After the entire `try-catch` statement stops for whatever reason, the `finally` block will always be executed. Ideal for cleanup.

```
f1 = null; f2 = null;
try {
  f1 = openFile("f1"); f2 = openFile("f2");
  //...
}
catch (IOException e) { //... }
finally {
  if (f1 != null) f1.close();
  if (f2 != null) f2.close();
}
```

Control Flow and `try-finally`

Control flow statements like `return`, `break`, and `continue` can be used in `try` blocks and `catch` blocks. They will work as expected.

The `finally` block, if exists, will be executed just before leaving the `try` or `catch` block. E.g.,

```
while (...) {
  try {
    continue; // honoured after finally
  }
  finally {
    System.out.println("finally");
  }
}

public int foo() {
  try {
    return 0; // honoured after finally
  }
  finally {
    System.out.println("finally");
  }
}
```


Control Flow and `try-finally`

However, if the `finally` block has its own control flow statement or throws its exceptions, then it supersedes those of the try and catch blocks.

```
try {
  if (b) return 0; // superseded
  else throw new Exception();
}
catch (Exception e) {
  if (c) return 1; // superseded
  else throw e;    // superseded
}
finally {
  return 2;  // this takes over
}
```

Generally, if a `try-catch` statement stops (by exception or control flow), the "reason" is remembered and honoured after `finally`, unless `finally` executes a control flow statement or throws an exception.

The `throws` Clause

The `throws` clause can list several exception types:

```
public void f()
throws AnException, MyException, YourException {
  // ...
}
```

So seek to be specific when you list exceptions.

Recall that when a subclass overrides a method of its superclass, there is a rule concerning the `throws` clause. Here it is in detail:

> If an exception type is listed in the subclass's version, then it or a superclass of it must also appear in the superclass's version.

Thus, in effect, the subclass's version does not differ externally from the superclass's version.

Exercises: Why does it make sense? And what problems could it cause?

The `Throwable`'s

The `Throwable` hierarchy roughly looks like:

```
Throwable--Error--...
         |       |-...--InternalError
         |-Exception--...
                    |-RuntimeException--...
```

Subclasses of `Error` represent built-in system fatal errors, e.g., out of memory, byte code file corrupted, etc.

Subclasses of `RuntimeException` represent common runtime errors, e.g., array index out of bound, null pointer, invalid arithmetic (e.g., divide by zero), invalid class casting, etc.

Since they are omnipresent, you are not required to catch them or list them in your `throws` clauses. They are called *unchecked* exceptions.

All other types of exceptions are called *checked* exceptions. You must catch them or list them.

The Throwable's

Throwable provides these operations:

- Throwable(), Throwable(String m): constructors, with optional message m
- String getMessage(), String getLocalizedMessage(): gets the message given to the constructor
- String toString(): the string contains the name of the exception type and the message
- void printStackTrace(),
  void printStackTrace(PrintStream s),
  void printStackTrace(PrintWriter w):

  outputs the stack frame recorded when the exception was created
- Throwable fillInStackTrace(): record the current stack frame, useful when you re-throw an exception

The Throwable's

All existing exceptions have two constructors like Throwable does: one for no message and the other for a message.

When you define your exceptions, it is good practice to at least do the same.

```
public class IQEmptyException extends Exception
{
  public IQEmptyException() {
    super();
  }
  public IQEmptyException(String m) {
    super(m);
  }
}
```


Throwable and finalize

Recall the header of the finalize method:

```
protected void finalize() throws Throwable
```

By default, your finalize method may throw any type of exceptions and errors. You do not have to catch exceptions in the code.

But of course, sometimes you still want to:

```
f1.close();  // if IOException thrown here
f2.close();  // then this won't run
```

In this case you will want to protect each close with a try-catch statement.

The InternalError Type

The InternalError type stands for "this should not happen".

Many people throw this when the impossible happens. E.g.,

```
public void foo(IntQueue q) {
  q.queue(10);
  try {
    q.unqueue();
  }
  catch (IQEmptyException e) {
    throw new InternalError("IntQueue broken!");
  }
}
```