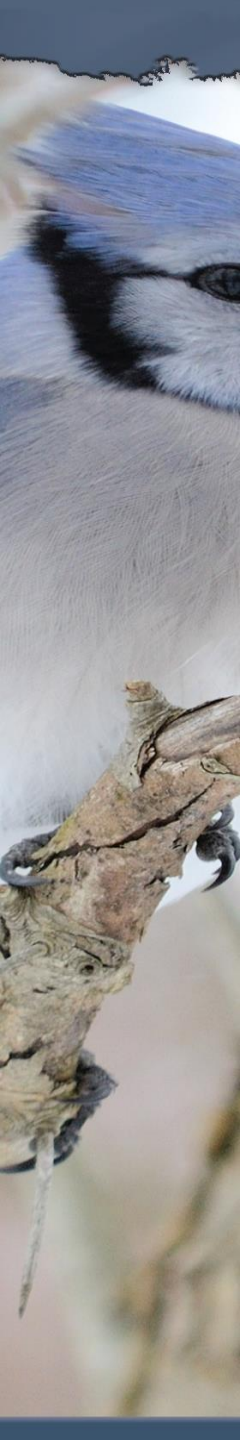
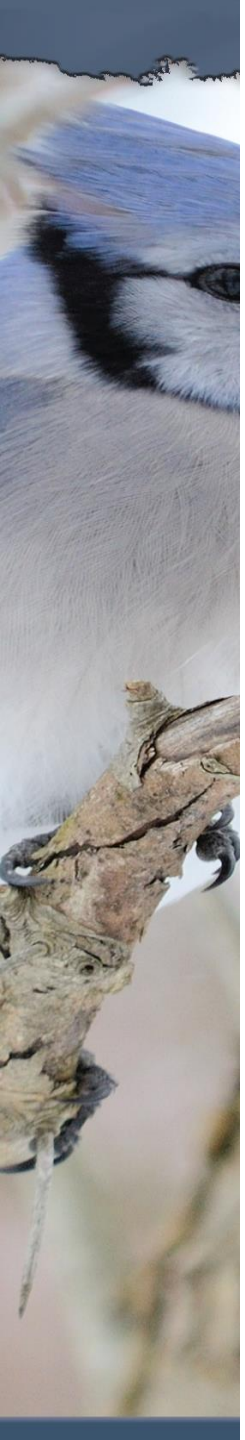
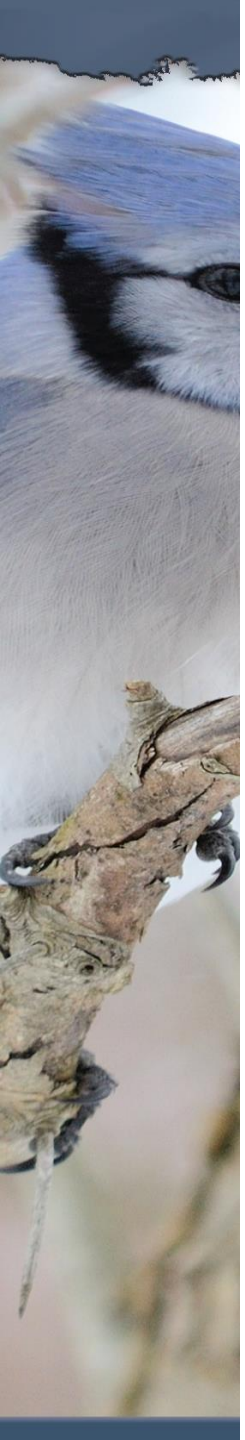


Handling errors



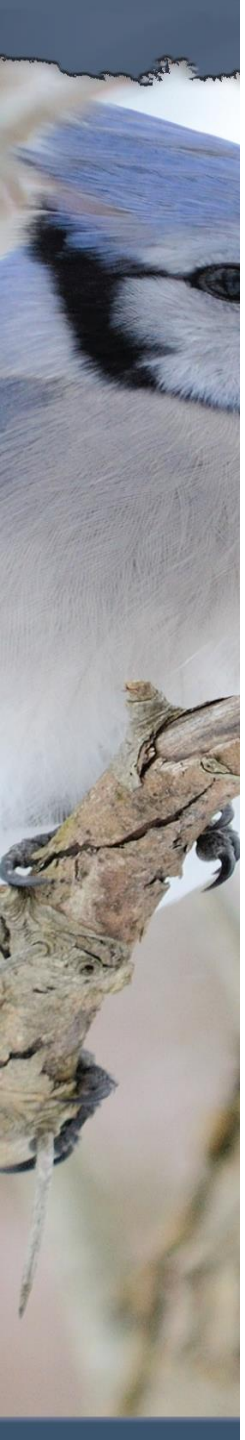
Arbeidskrav 1

- Skal nå være vurdert.
- Spørsmål?



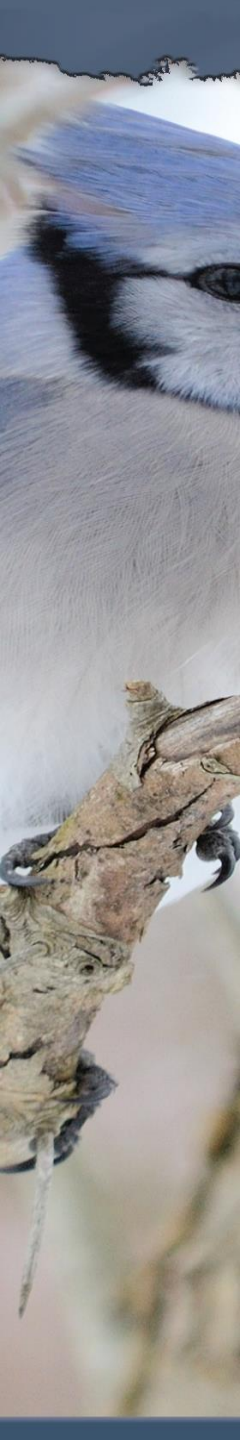
Arbeidskrav 2

- Det er satt av tid til å jobbe med arbeidskrav 2 i denne øvingen (også).
- Fortsett å bruke forumet!
- Ser ut til å bli vurdert mandag 23.4.
- Noen spørsmål?



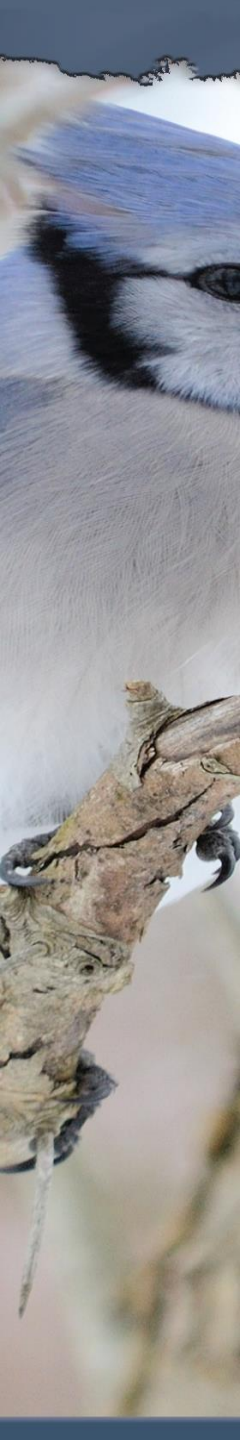
JavaFX og eksamen

- Det blir ikke oppgaver om å kode et GUI.
- Det kan bli oppgaver som:
  - Tegne opp et GUI basert på JavaFX-koden som foreligger. (Altså skjønne koden)
  - Om JavaFX og GUI som ikke besvares med kode.



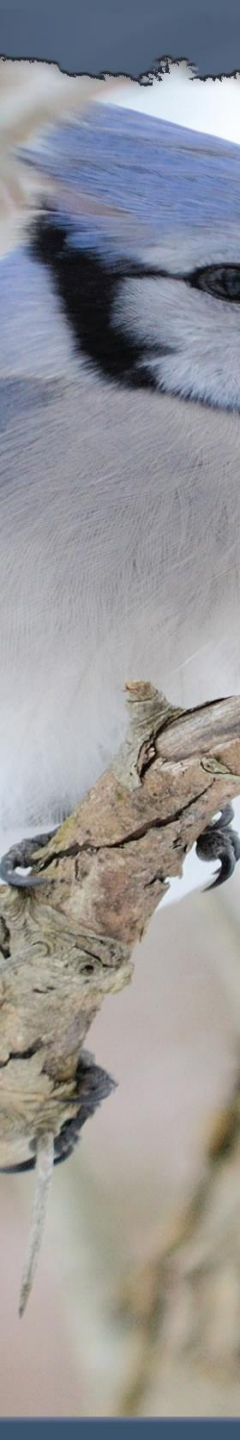
Main concepts to be covered

- Defensive programming.
  - Anticipating that things could go wrong.
- Exception handling and throwing.
- Error reporting.
- Simple file processing.



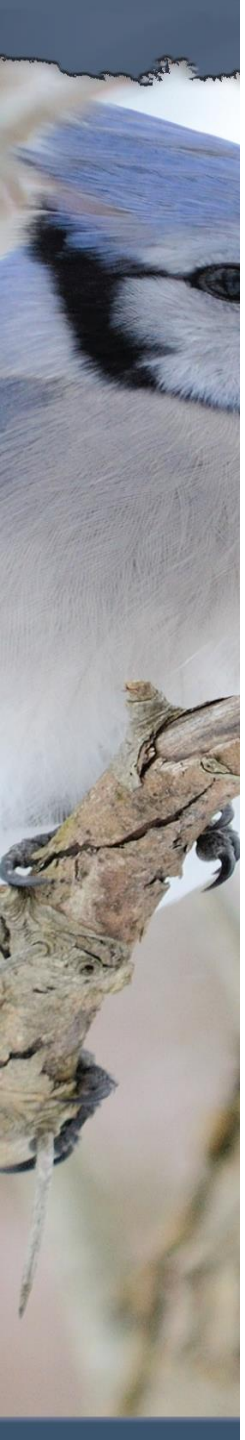
Typical error situations

- Incorrect implementation.
  - Does not meet the specification.
- Inappropriate object request.
  - E.g., invalid index.
- Inconsistent or inappropriate object state.
  - E.g. arising through class extension.



Not always programmer error

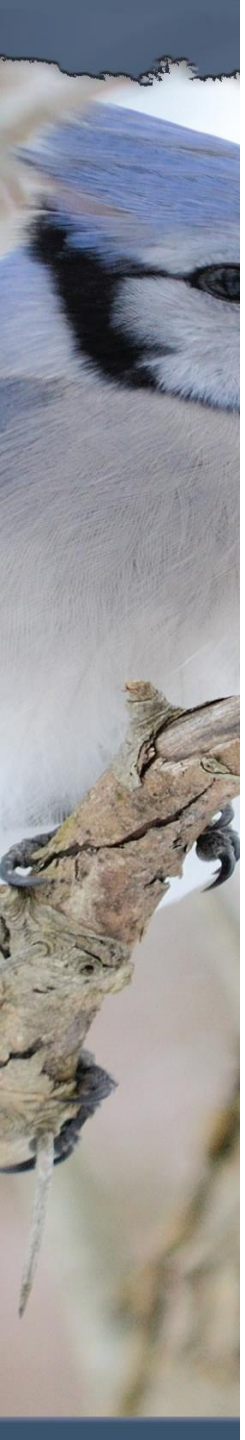
- Errors often arise from the environment:
  - Incorrect URL entered.
  - Network interruption.
- File processing is particular error-prone:
  - Missing files.
  - Lack of appropriate permissions.



Exploring errors

- Explore error situations through the *address-book* projects.
- Two aspects:
  - Error reporting.
  - Error handling.

TreeMap



java.util

Class TreeMap<K,V>

java.lang.Object
    java.util.AbstractMap<K,V>
        java.util.TreeMap<K,V>

Type Parameters:

`K` - the type of keys maintained by this map

`V` - the type of mapped values

All Implemented Interfaces:

`Serializable, Cloneable, Map<K,V>, NavigableMap<K,V>, SortedMap<K,V>`

```
public class TreeMap<K,V>
extends AbstractMap<K,V>
implements NavigableMap<K,V>, Cloneable, Serializable
```

A Red-Black tree based `NavigableMap` implementation. The map is sorted according to the natural ordering of its keys, or by a `Comparator` provided at map creation time, depending on which constructor is used.

This implementation provides guaranteed log(n) time cost for the `containsKey`, `get`, `put` and `remove` operations. Algorithms are adaptations of those in Cormen, Leiserson, and Rivest's *Introduction to Algorithms*.

tailMap

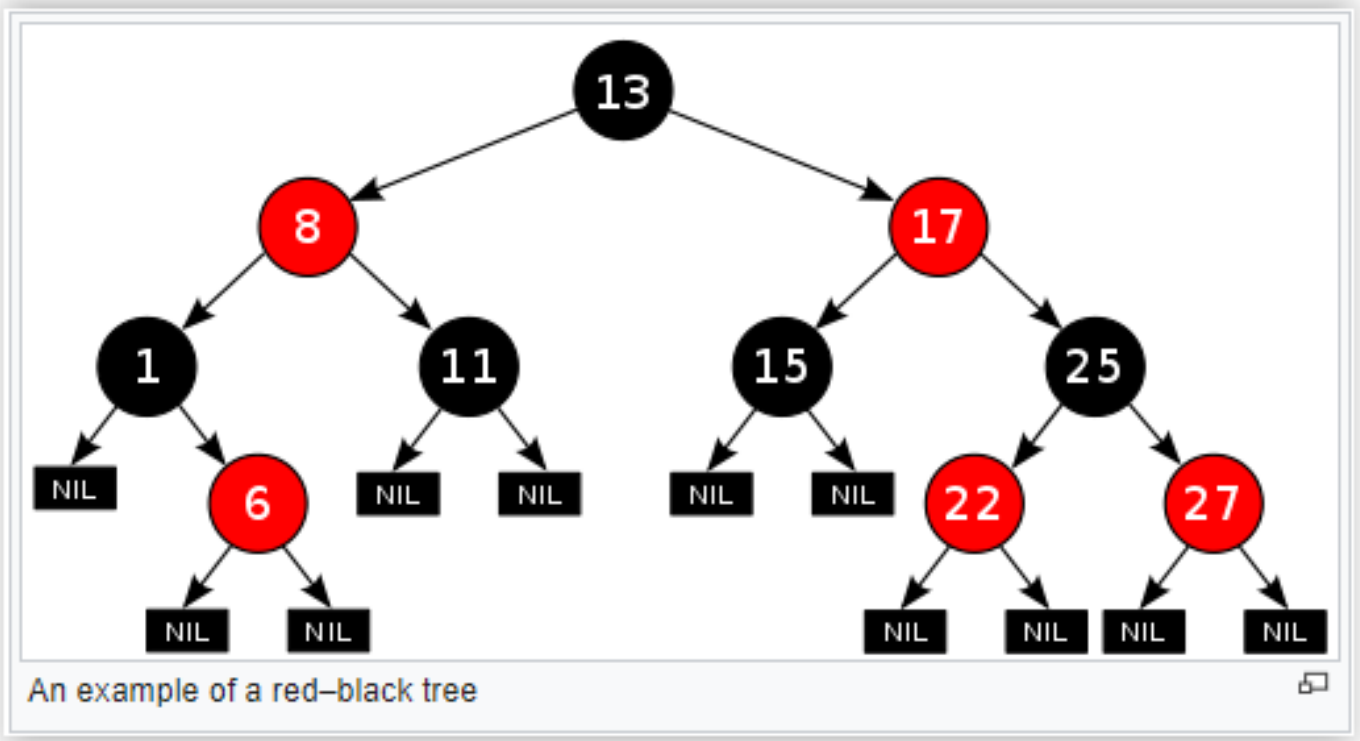
```
public SortedMap<K,V> tailMap(K fromKey)
```

Description copied from interface: `NavigableMap`

Returns a view of the portion of this map whose keys are greater than or equal to `fromKey`. The returned map is backed by this map, so changes in the returned map are reflected in this map, and vice-versa. The returned map supports all optional map operations that this map supports.

The returned map will throw an `IllegalArgumentException` on an attempt to insert a key outside its range.

Red-black tree



An example of a red–black tree

- Each node is either red or black.
- The root is black. This rule is sometimes omitted. Since the root can always be changed from red to black, but not necessarily vice versa, this rule has little effect on analysis.
- All leaves (NIL) are black.
- If a node is red, then both its children are black.
- Every path from a given node to any of its descendant NIL nodes contains the same number of black nodes. Some definitions: the number of black nodes from the root to a node is the node's **black depth**; the uniform number of black nodes in all paths from root to the leaves is called the **black-height** of the red-black tree.

wikipedia

SortedMap

java.util

Interface SortedMap<K,V>

Type Parameters:

K - the type of keys maintained by this map

V - the type of mapped values

All SuperInterfaces:

Map<K,V>

All Known SubInterfaces:

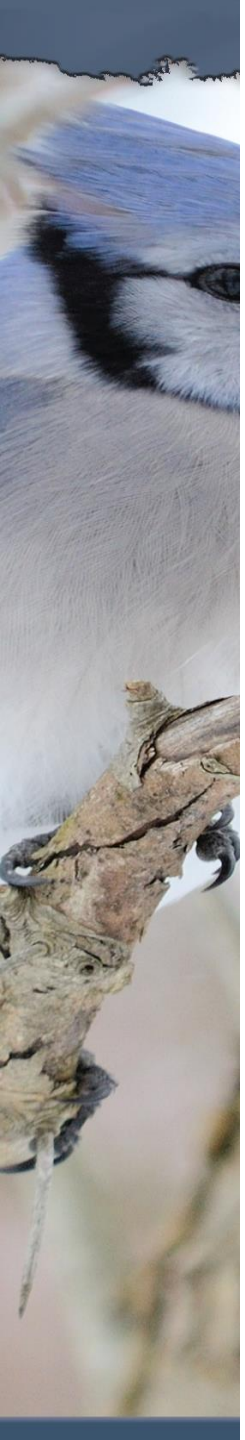
ConcurrentNavigableMap<K,V>, NavigableMap<K,V>

All Known Implementing Classes:

ConcurrentSkipListMap, TreeMap

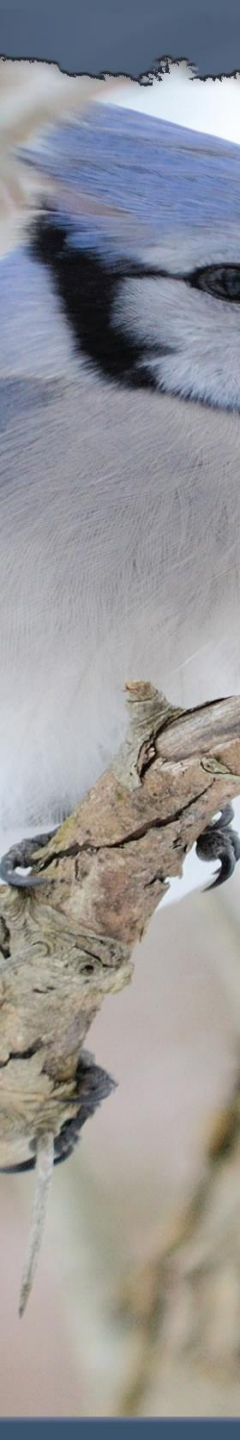
```
public interface SortedMap<K,V>
extends Map<K,V>
```

A Map that further provides a *total ordering* on its keys. The map is ordered according to the natural ordering of its keys, or by a Comparator typically provided at sorted map creation time. This order is reflected when iterating over the sorted map's collection views (returned by the entrySet, keySet and values methods). Several additional operations are provided to take advantage of the ordering. (This interface is the map analogue of SortedSet.)



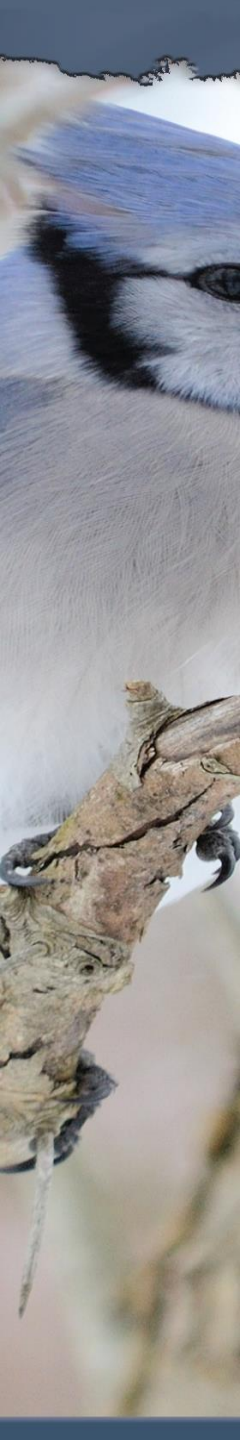
Defensive programming

- Client-server interaction.
  - Should a server assume that clients are well-behaved?
  - Or should it assume that clients are potentially hostile?
- Significant differences in implementation required.



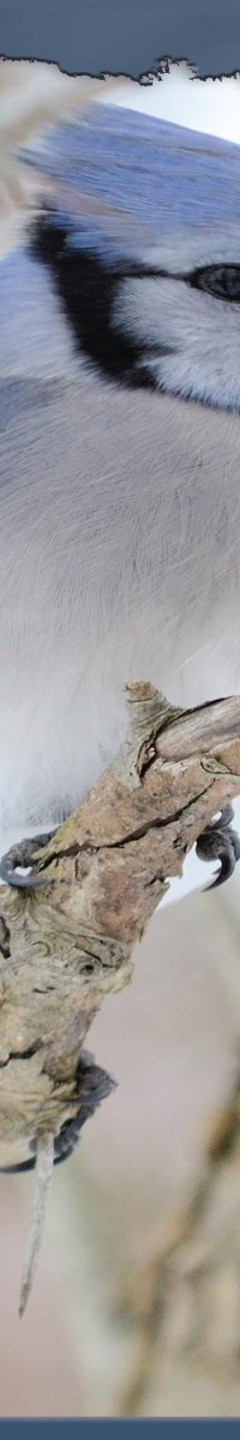
Issues to be addressed

- How much checking by a server on method calls?
- How to report errors?
- How can a client anticipate failure?
- How should a client deal with failure?



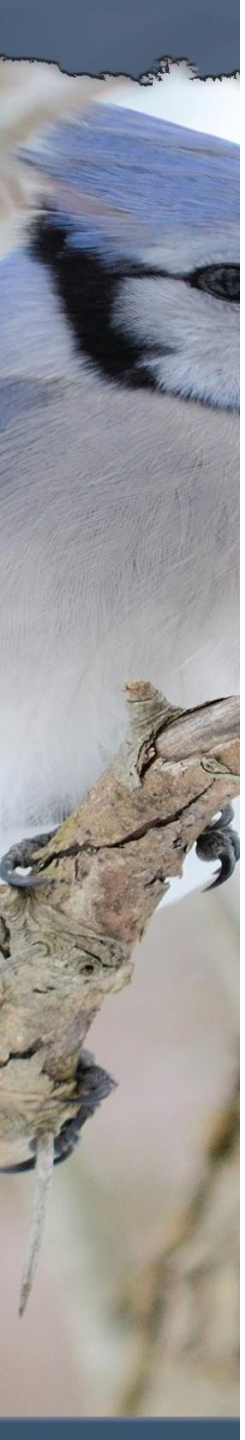
An example

- Create an **`AddressBook`** object.
- Try to remove an entry.
- Answer this:
  - What happens?
  - Why?



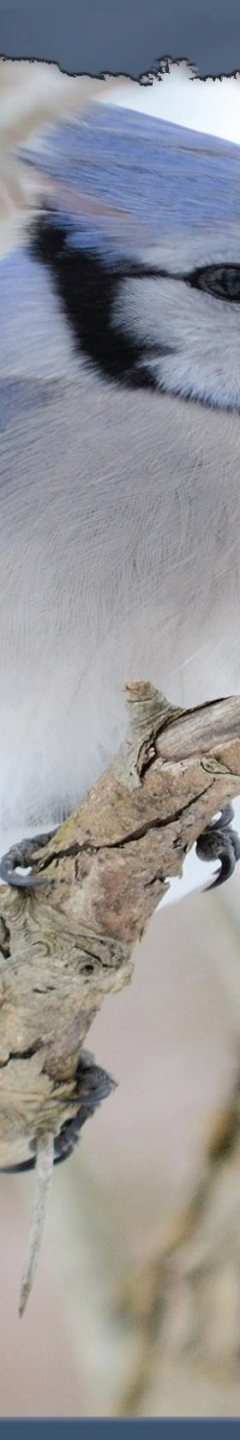
Argument values

- Arguments represent a major ‘vulnerability’ for a server object.
  - Constructor arguments initialize state.
  - Method arguments often contribute to behavior.
- Argument checking is one defensive measure.



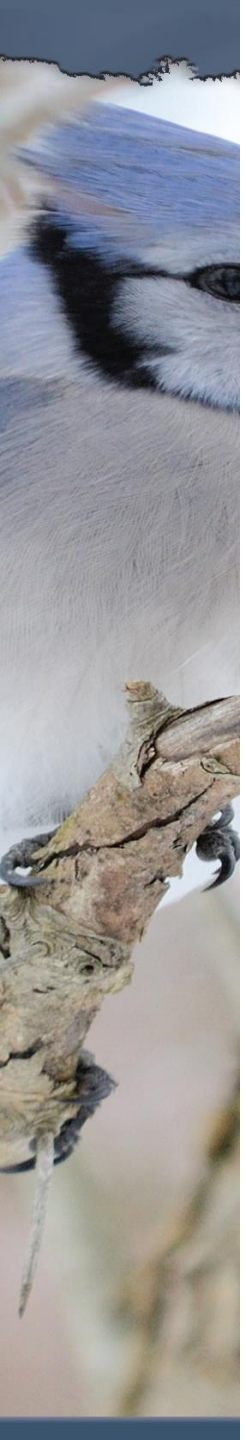
Checking the key

```
public void removeDetails(String key)
{
    if(keyInUse(key)) {
        ContactDetails details = book.get(key);
        book.remove(details.getName());
        book.remove(details.getPhone());
        numberOfEntries--;
    }
}
```



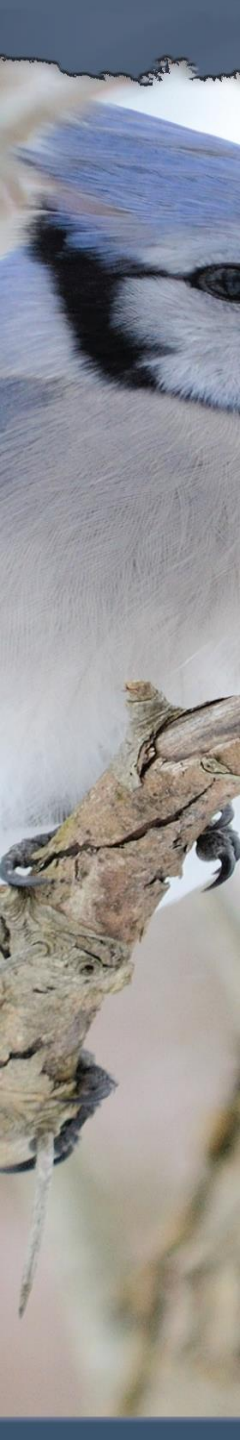
Server error reporting

- How to report illegal arguments?
  - To the user?
    - Is there a human user?
    - Can they solve the problem?
  - To the client object?
    - Return a diagnostic value.
    - *Throw an exception.*



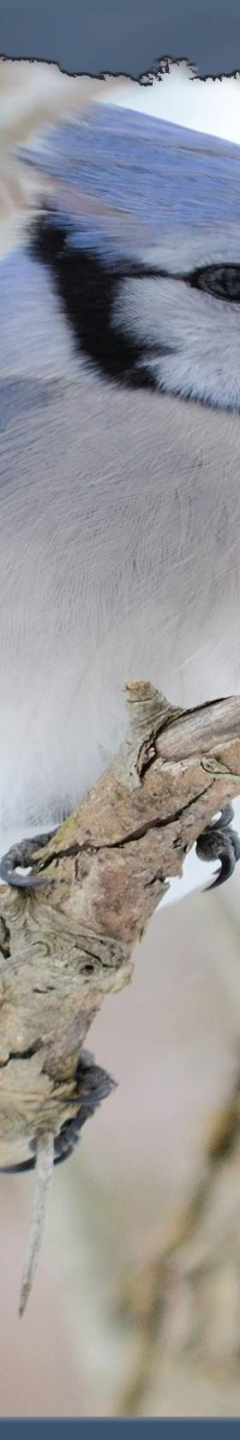
Returning a diagnostic

```
public boolean removeDetails(String key)
{
    if(keyInUse(key)) {
        ContactDetails details = book.get(key);
        book.remove(details.getName());
        book.remove(details.getPhone());
        numberOfEntries--;
        return true;
    }
    else {
        return false;
    }
}
```



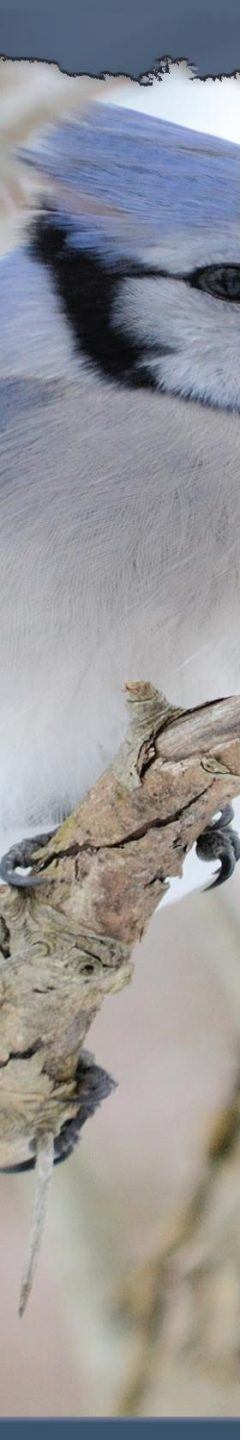
Client can check for success

```
if(contacts.removeDetails("…")) {
    // Entry successfully removed.
    // Continue as normal.
    …
}
else {
    // The removal failed.
    // Attempt a recovery, if possible.
    …
}
```



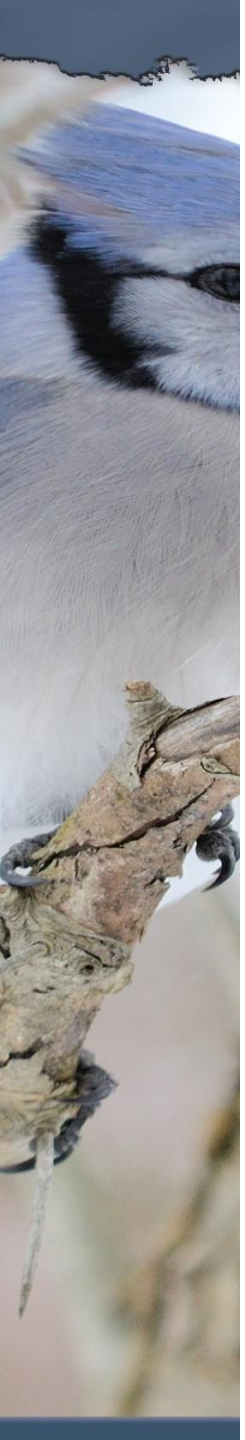
Potential client responses

- Test the return value.
  - Attempt recovery on error.
  - Avoid program failure.
- Ignore the return value.
  - Cannot be prevented.
  - Likely to lead to program failure.
- ‘Exceptions’ are preferable.



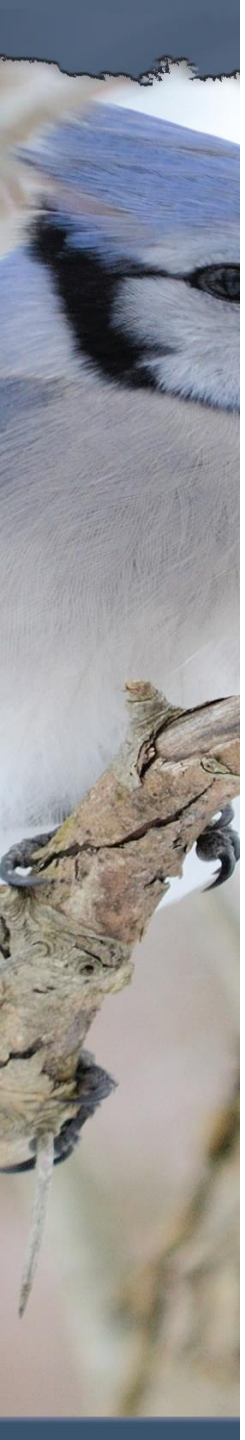
Exception-throwing principles

- A special language feature.
- No ‘special’ return value needed.
- Errors cannot be ignored in the client.
  - The normal flow-of-control is interrupted.
- Specific recovery actions are encouraged.



Throwing an exception

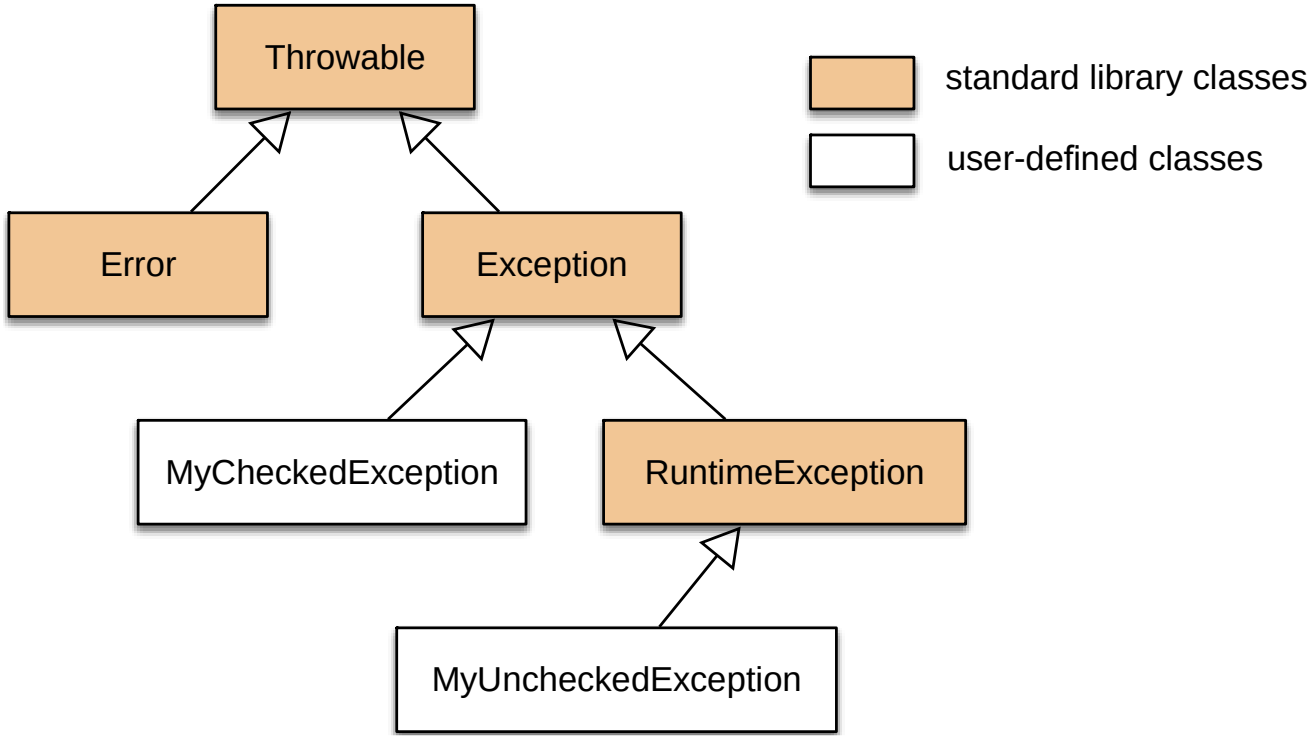
```
/**
 * Look up a name or phone number and return the
 * corresponding contact details.
 * @param key The name or number to be looked up.
 * @return The details corresponding to the key,
 *         or null if there are none matching.
 * @throws IllegalArgumentException if
 *         the key is invalid.
 */
public ContactDetails getDetails(String key)
{
    if(key == null) {
        throw new IllegalArgumentException(
                              "null key in getDetails");
    }
    return book.get(key);
}
```

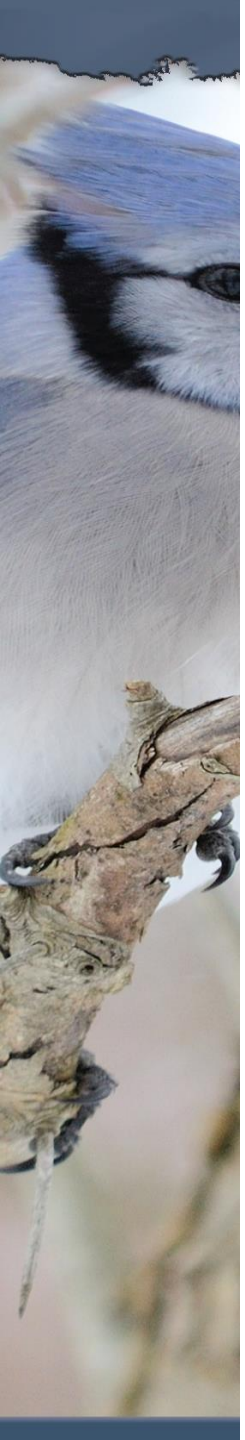


Throwing an exception

- An exception object is constructed:
  - `new ExceptionType("...")`
- The exception object is thrown:
  - `throw ...`
- Javadoc documentation:
  - `@throws ExceptionType reason`

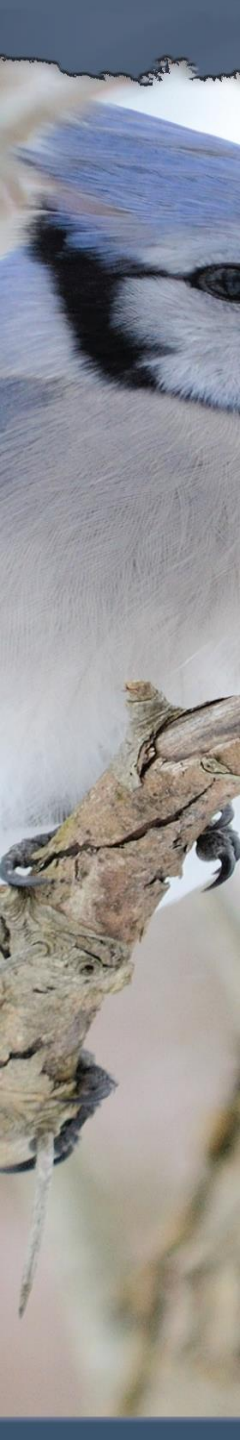
The exception class hierarchy





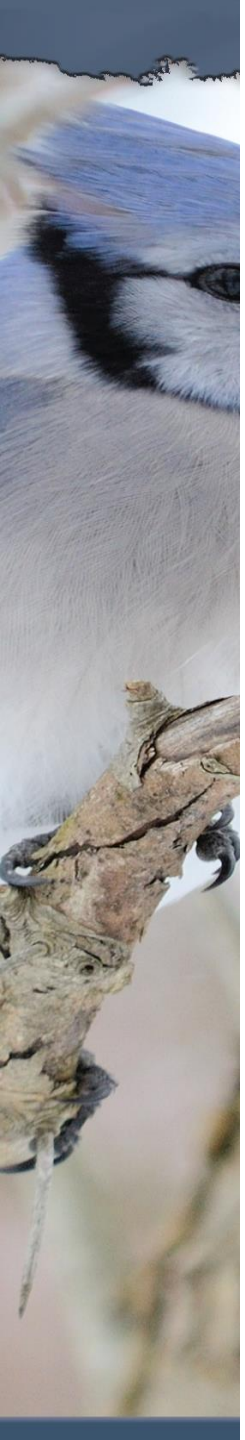
Exception categories

- Checked exceptions
  - Subclass of **`Exception`**
  - Use for anticipated failures.
  - Where recovery may be possible.
- Unchecked exceptions
  - Subclass of **`RuntimeException`**
  - Use for unanticipated failures.
  - Where recovery is unlikely.



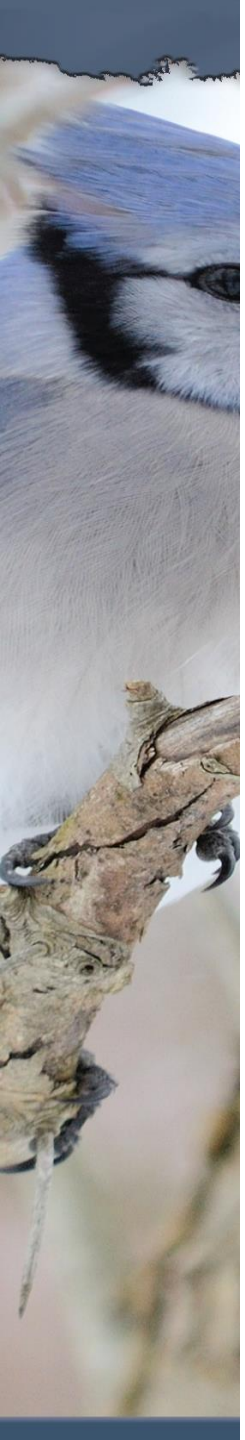
The effect of an exception

- The throwing method finishes prematurely.
- No return value is returned.
- Control does not return to the client's point of call.
  - So the client cannot carry on regardless.
- A client may 'catch' an exception.



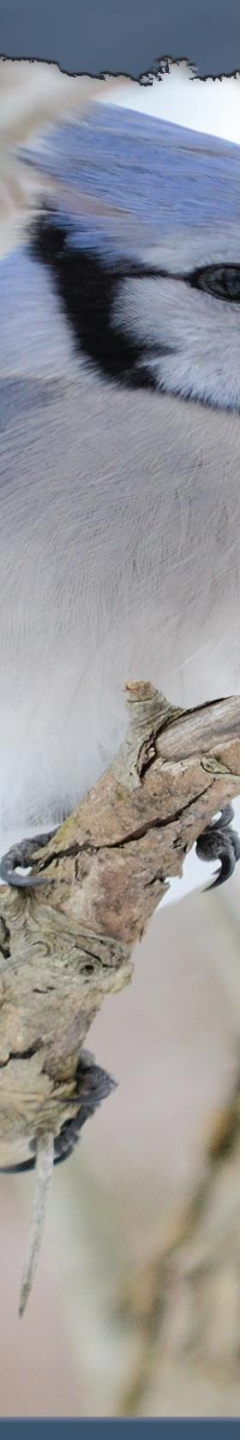
Unchecked exceptions

- Use of these is ‘unchecked’ by the compiler.
- Cause program termination if not caught.
  - This is the normal practice.
- **`IllegalArgumentException`** is a typical example.



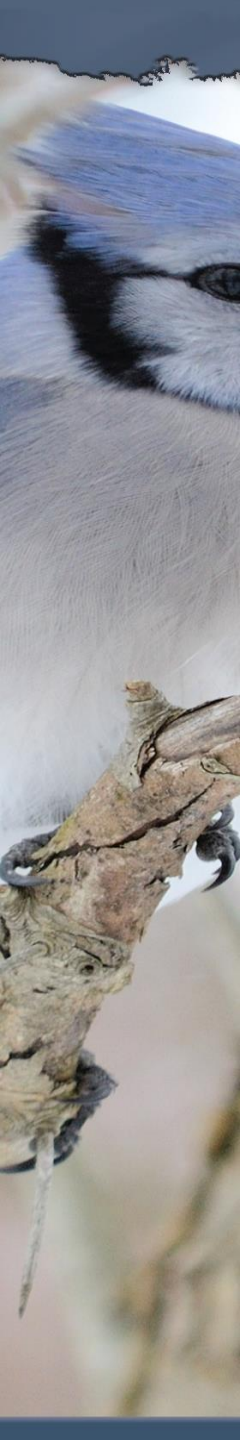
Argument checking

```
public ContactDetails getDetails(String key)
{
    if(key == null) {
        throw new IllegalArgumentException(
                            "null key in getDetails");
    }
    if(key.trim().length() == 0) {
        throw new IllegalArgumentException(
                          "Empty key passed to getDetails");
    }
    return book.get(key);
}
```



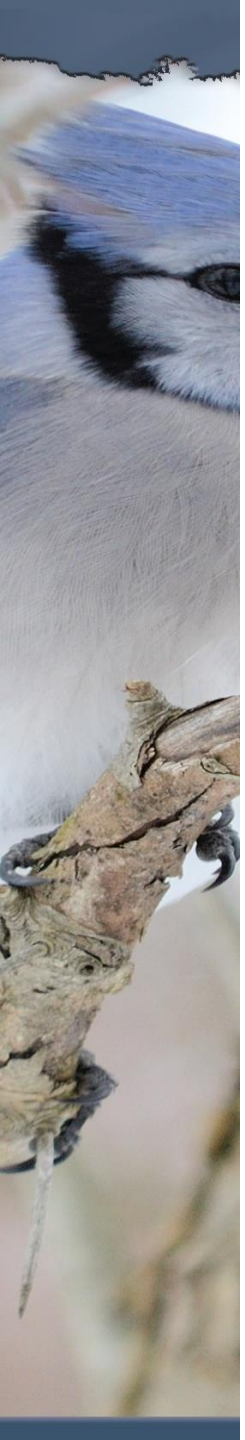
Preventing object creation

```
public ContactDetails(String name, String phone, String address)
{
    if(name == null) {
        name = "";
    }
    if(phone == null) {
        phone = "";
    }
    if(address == null) {
        address = "";
    }

    this.name = name.trim();
    this.phone = phone.trim();
    this.address = address.trim();

    if(this.name.length() == 0 && this.phone.length() == 0) {
        throw new IllegalStateException(
                "Either the name or phone must not be blank.");
    }
}
```



Exception handling

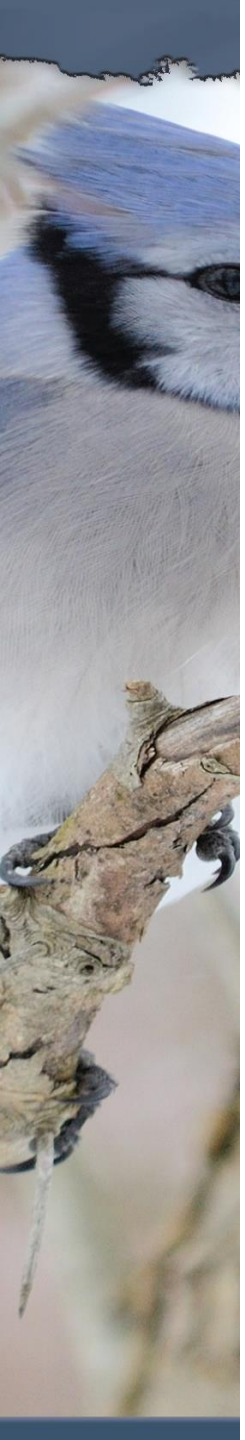
- Checked exceptions are meant to be caught and responded to.
- The compiler ensures that their use is tightly controlled.
  - In both server and client objects.
- Used properly, failures may be recoverable.



The throws clause

- Methods throwing a checked exception must include a throws clause:

```
public void saveToFile(String destinationFile)
    throws IOException
```

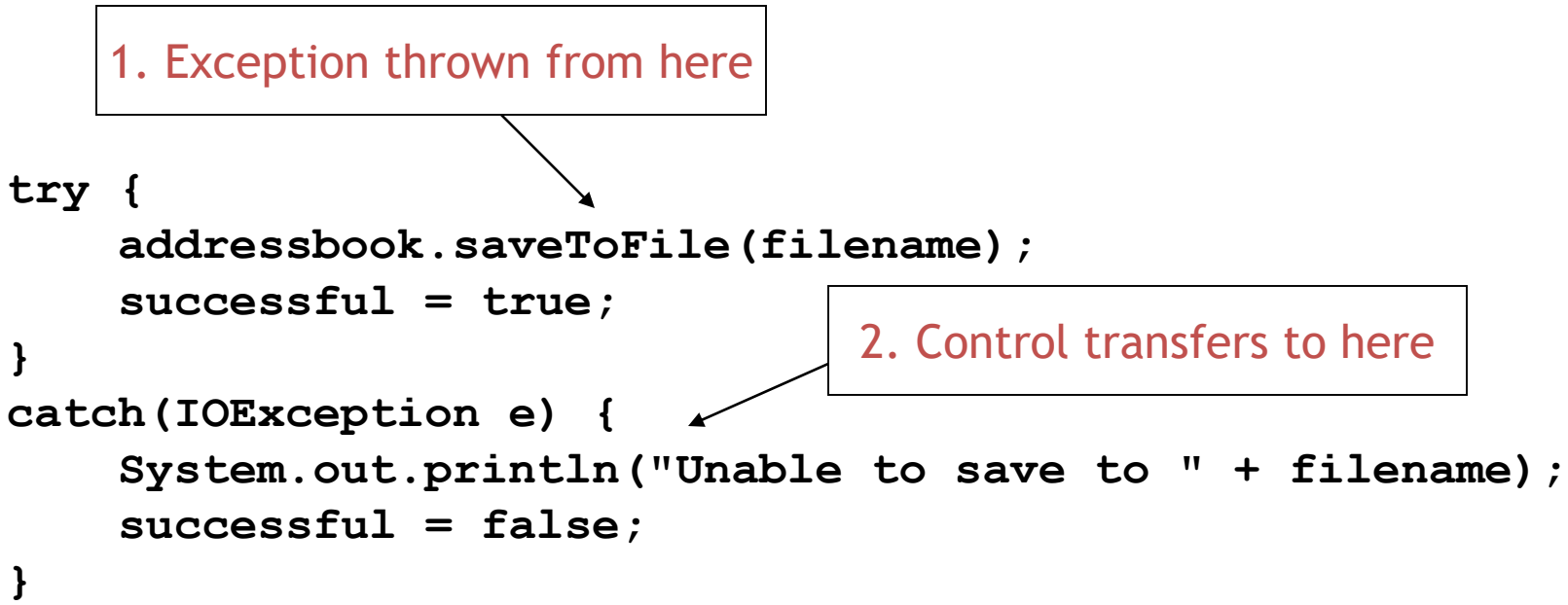


The try statement

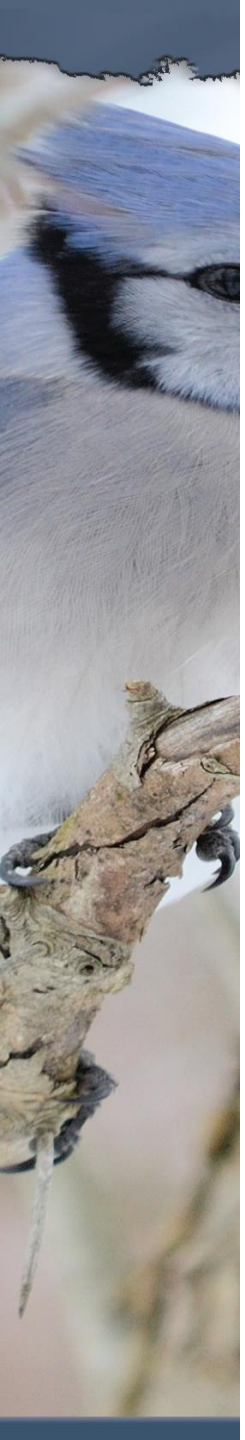
- Clients catching an exception must protect the call with a try statement:

```
try {
    Protect one or more statements here.
}
catch(Exception e) {
    Report and recover from the
    exception here.
}
```

The try statement

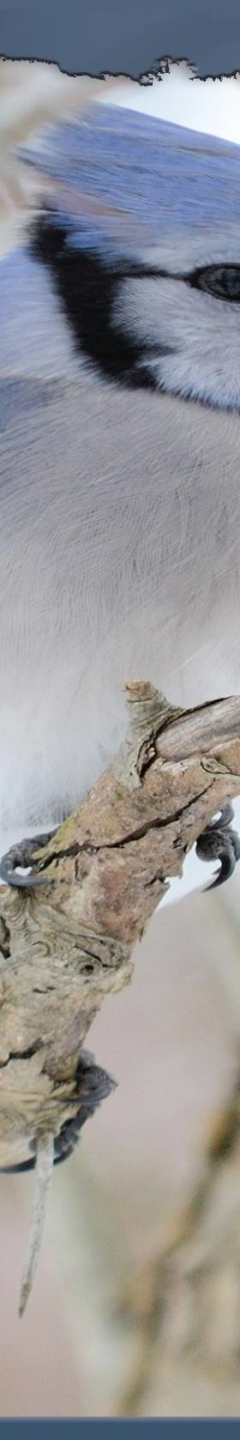



```
try {
    addressbook.saveToFile(filename);
    successful = true;
}
catch(IOException e) {
    System.out.println("Unable to save to " + filename);
    successful = false;
}
```



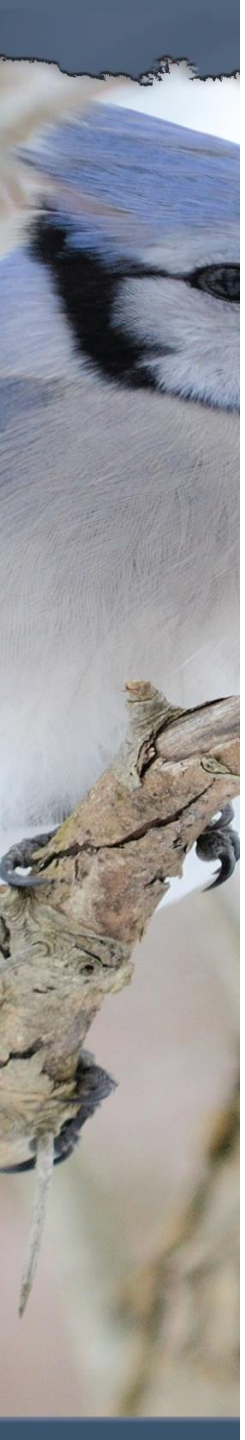
Catching multiple exceptions

```
try {
    ...
    ref.process();
    ...
}
catch(EOFException e) {
    // Take action on an end-of-file exception.
    ...
}
catch(FileNotFoundException e) {
    // Take action on a file-not-found exception.
    ...
}
```



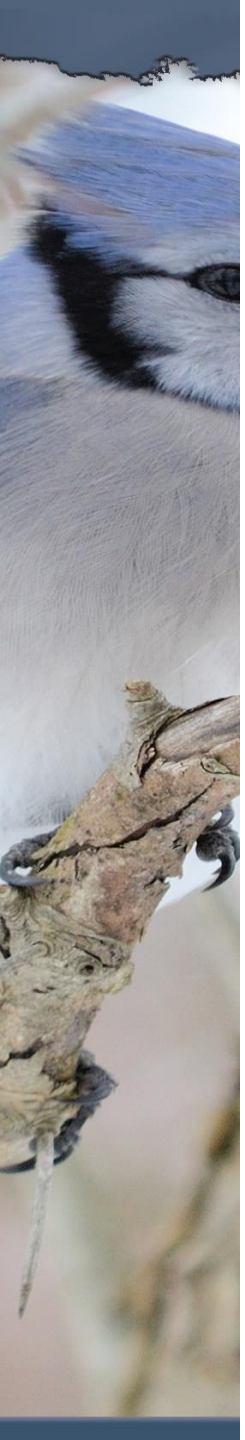
Multi-catch

```
try {
    ...
    ref.process();
    ...
}
catch(EOFException | FileNotFoundException e) {
    // Take action appropriate to both types
    // of exception.
    ...
}
```



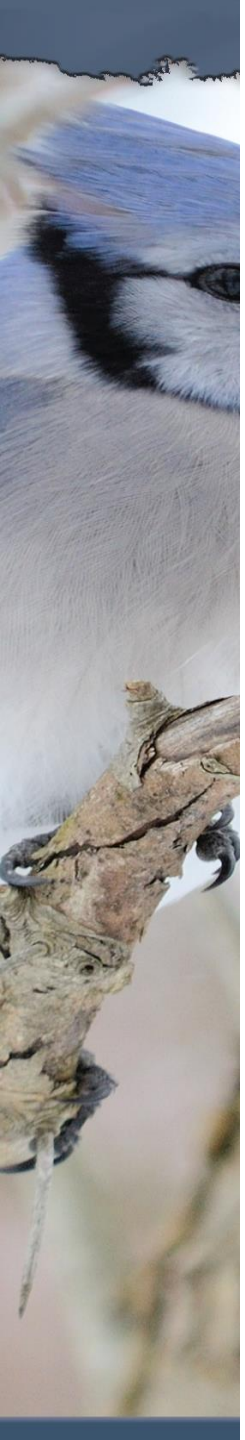
The finally clause

```
try {
     Protect one or more statements here.
}
catch(Exception e) {
     Report and recover from the exception here.
}
finally {
     Perform any actions here common to whether or
     not an exception is thrown.
}
```



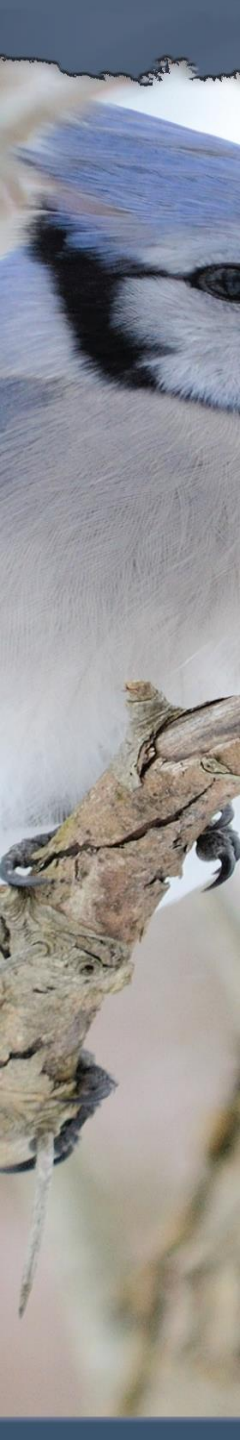
The finally clause

- A finally clause is executed even if a return statement is executed in the try or catch clauses.
- A uncaught or *propagated* exception still exits via the finally clause.



Propagating an exception

- We do not have to catch an exception. But if we do not catch a checked exception, we must declare to let the exception propagate up the call stack.
- We do so by using *throws* in the method signature.



Nå

- Kahoot☺
- Deretter øving her på Fjerdingen (sjekk TimEdit for rom)
  - Jobb med arbeidskrav 2 i øvingen…
- OBS! Neste uke ser det ut til å bli forelesning på Vulkan!