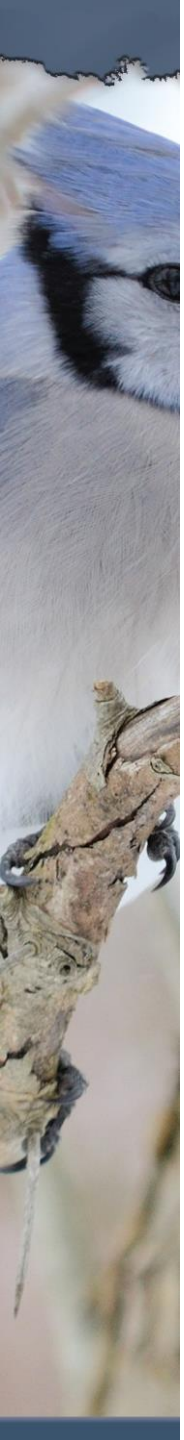
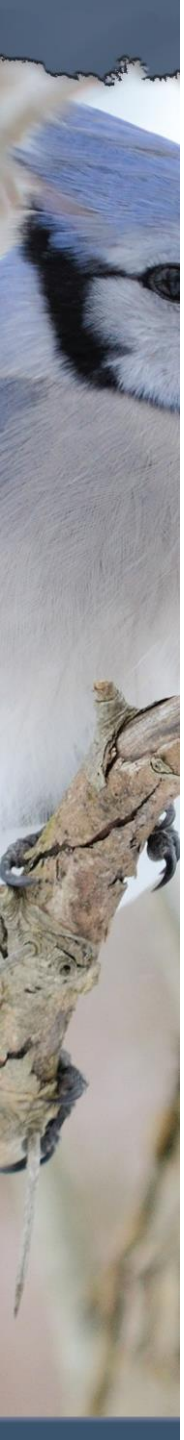


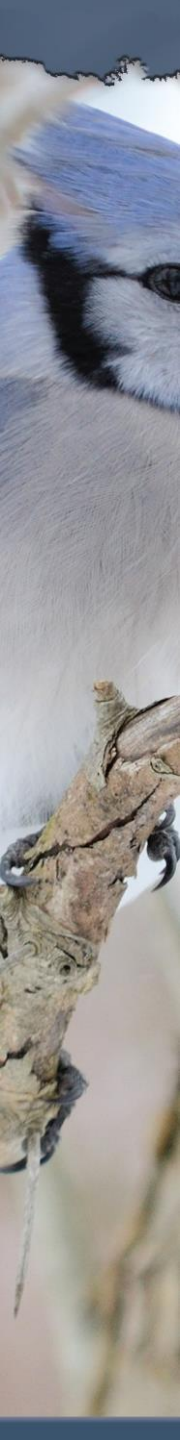
Exception Handling

Chapter 14-BlueJ and Chapter 11-Java How to Program
Edited by Ehsan Edalat



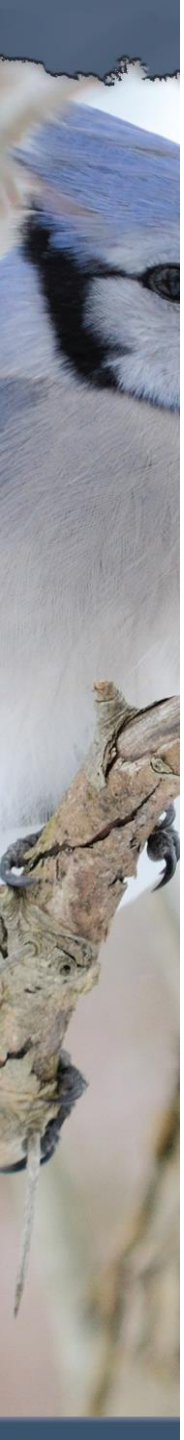
Runtime Errors

- Divide by zero
  - Result = first / second;
- Casting errors
  - Computer c = (Computer) d;
- Invalid index
  - list.get(index);



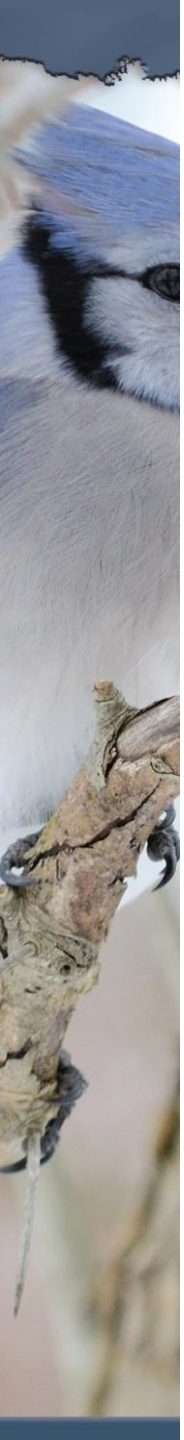
Not always programmer error

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



Exploring errors

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



An example

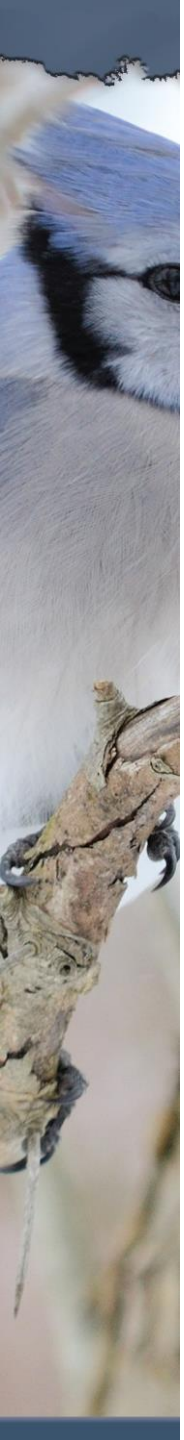
- Create an **`AddressBook`** object.
- Try to remove an entry.
- A runtime error results.
  - Whose ‘fault’ is this?
- Anticipation and prevention are preferable to apportioning blame.

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--;
    }
}
```

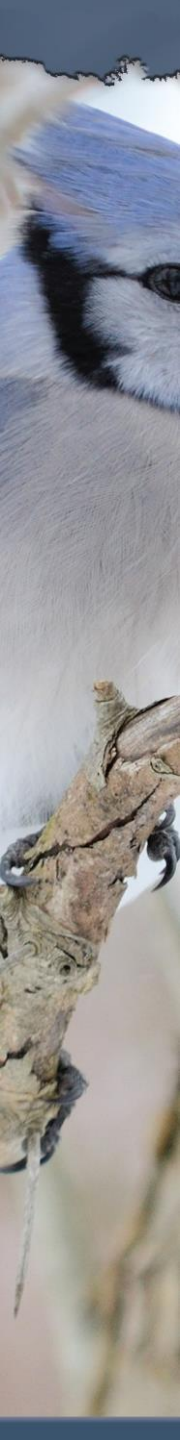
Returning a diagnostic

```
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;
    }
}
```



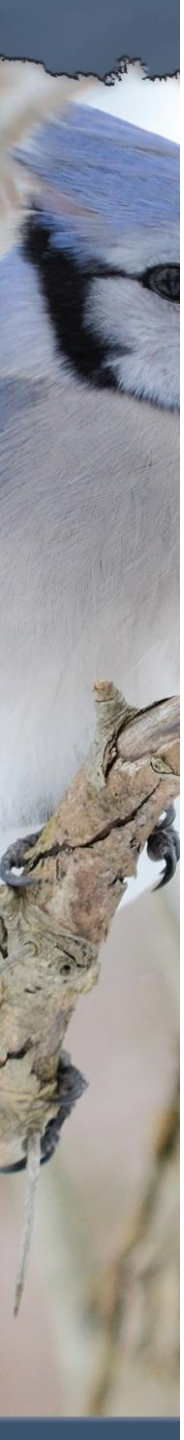
Client can check for success

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



Potential Programmer 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.

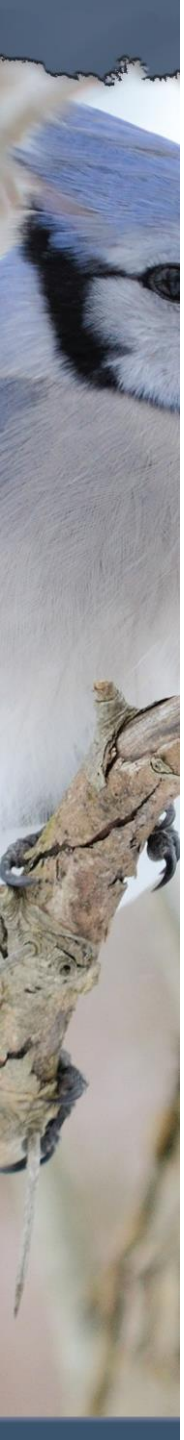


Exception-throwing principles

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

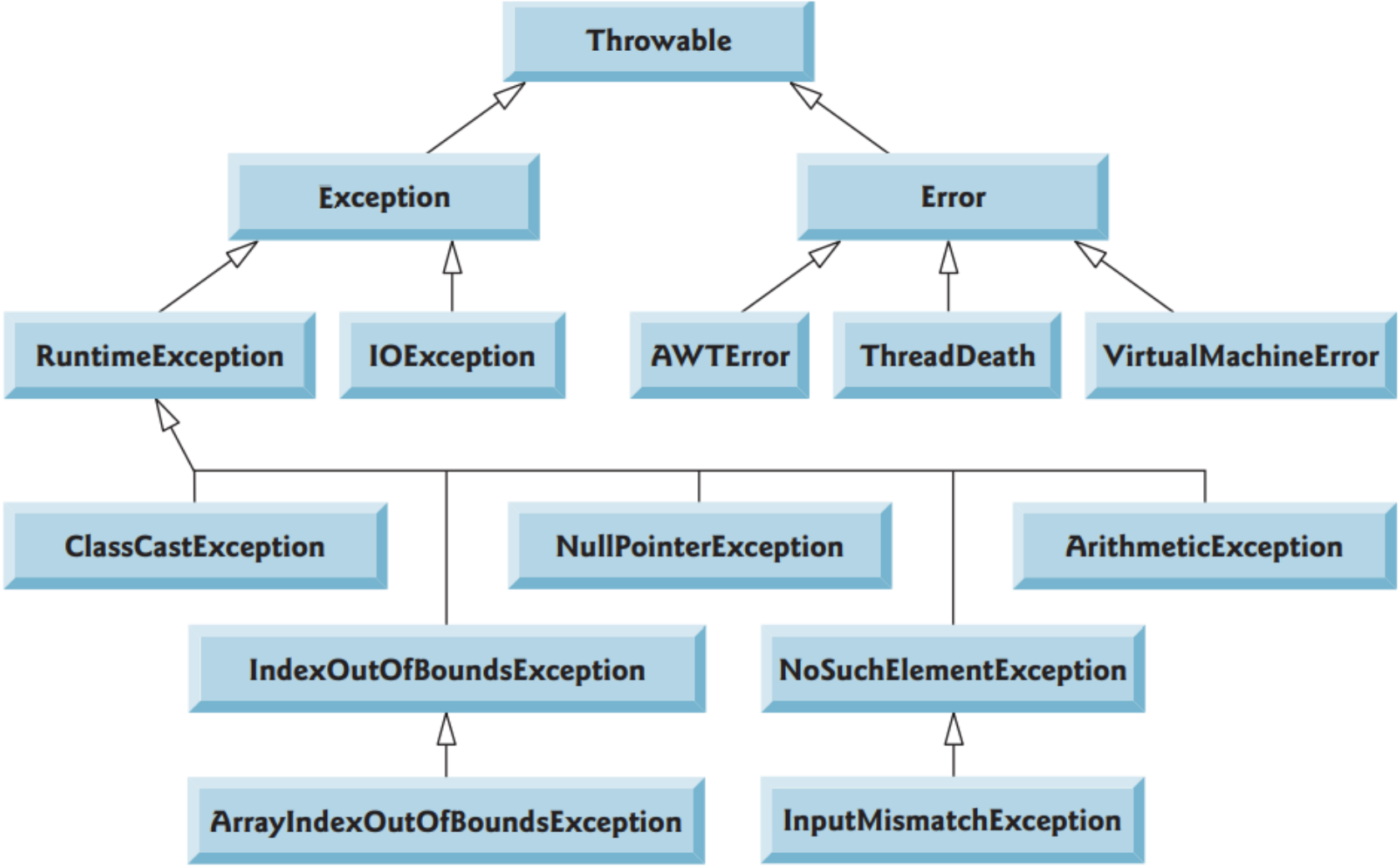
Throwing an exception

```
/**
 * 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

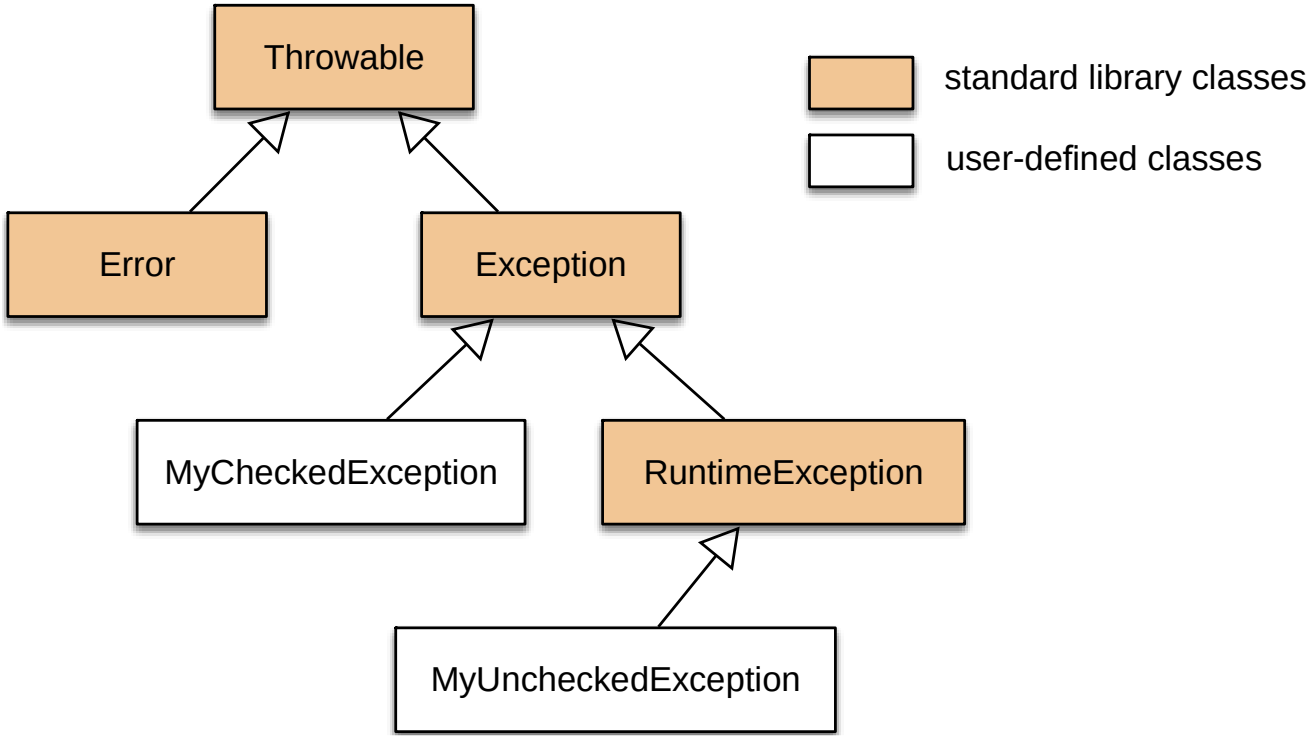
- An exception object is constructed:
  - `new ExceptionType("...")`
- The exception object is thrown:
  - `throw ...`
- Javadoc documentation:
  - `@throws ExceptionType reason`

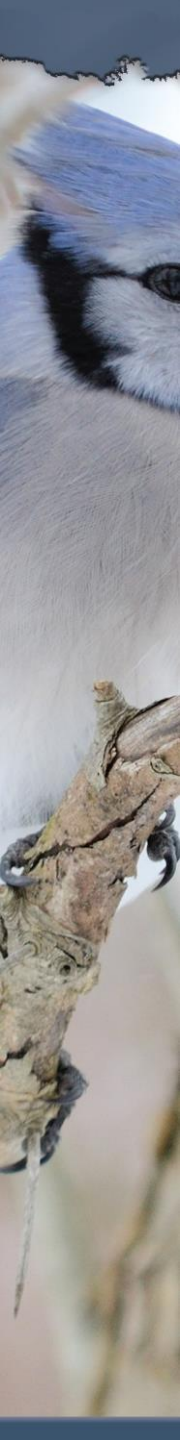


`Exception` (package `java.lang`) and `IOException` (package `java.io`)—represent exceptional situations that can occur in a Java program and that can be caught by the application. Class `Error` and its subclasses represent *abnormal situations* that happen in the JVM. Most `Errors` *happen infrequently and should not be caught by applications—it's usually not possible for applications to recover from* `Errors`.

https://stackoverflow.com/questions/5217712/error-class-in-java

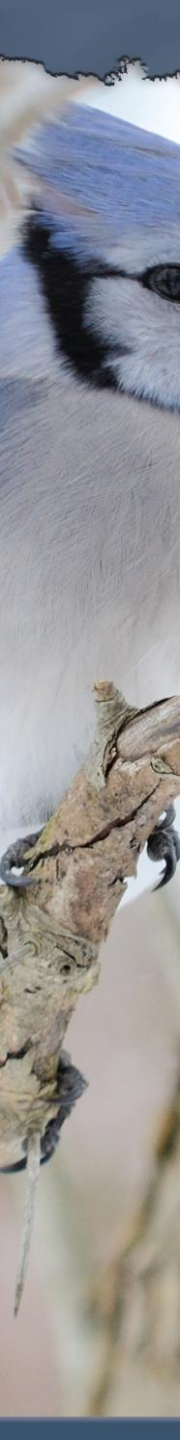
The exception class hierarchy





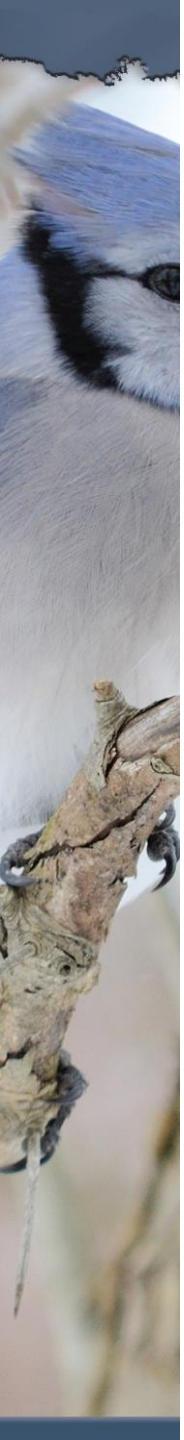
Exception categories

- Checked exceptions
  - Subclass of `Exception`
  - Use for anticipated failures.
  - Where recovery may be possible.
  - When a environmental problem exist.
- Unchecked exceptions
  - Subclass of `RuntimeException`
  - Use for unanticipated failures.
  - Where recovery is unlikely.
  - When a logical problem exist.



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.



- Checked exceptions must be explicitly handled.
- Unchecked exceptions do not have this requirement. They don't have to be caught or declared thrown.

Unchecked exceptions

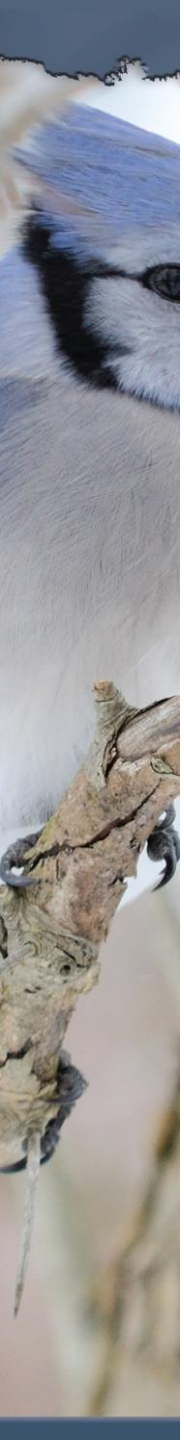
- Use of these is ‘unchecked’ by the compiler.
- Cause program termination if not caught.
  - This is the normal practice.
- **`IllegalArgumentException`** is a typical example.

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);
}
```

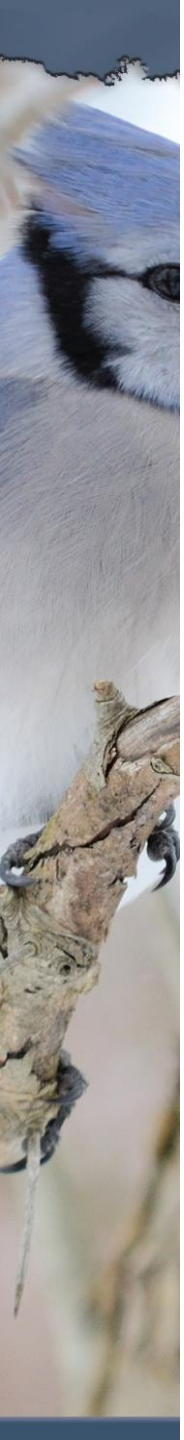
Preventing object creation

```
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

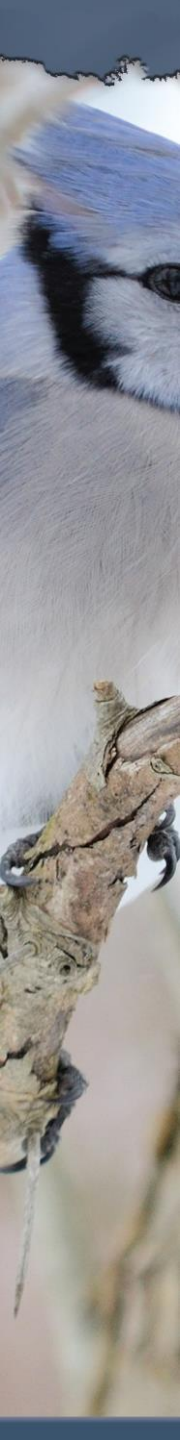
- Checked exceptions are meant to be caught and responded to.
- The compiler ensures that their use is tightly controlled.
  - In both server and client objects.
- Used properly, failures may be recoverable.



The throws clause

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

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



The try statement

- Methods 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

1. Exception thrown from here

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

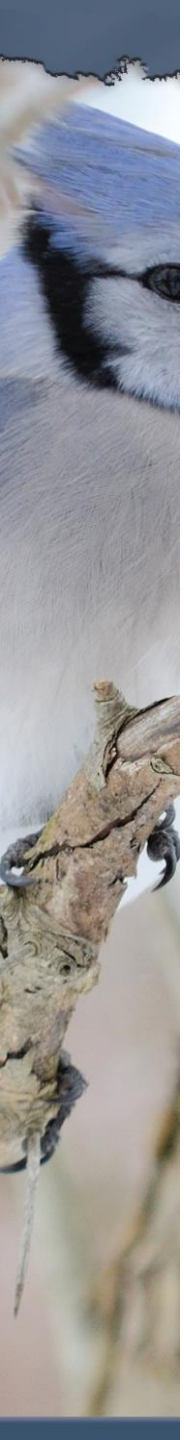
2. Control transfers to here

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.
    ...
}
```

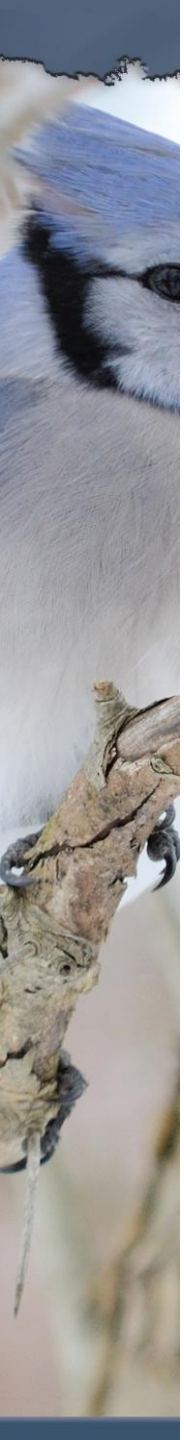
Multi-catch

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



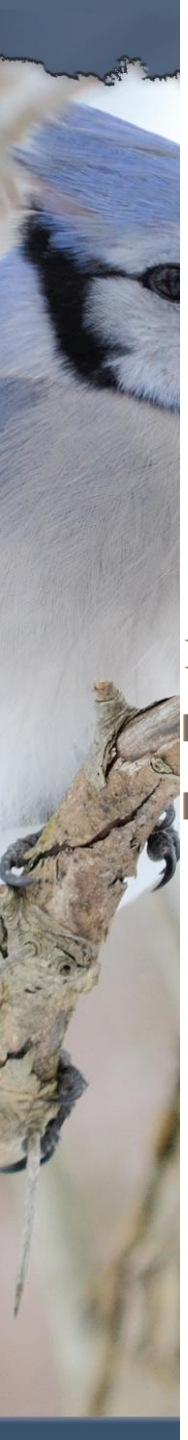
The finally clause

```
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.

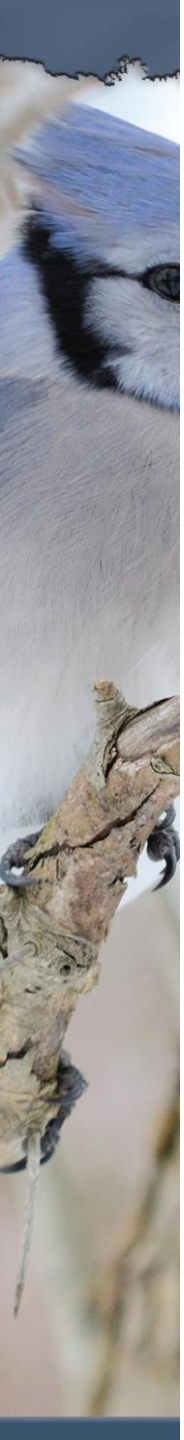


```
try {
    Protect one or more statements here.
}
catch(Exception e) {
    Report and recover from the exception here.
}
Perform any actions here common to whether or not an exception is thrown.
```

In fact, there are at least two cases where these two examples would have different effects:

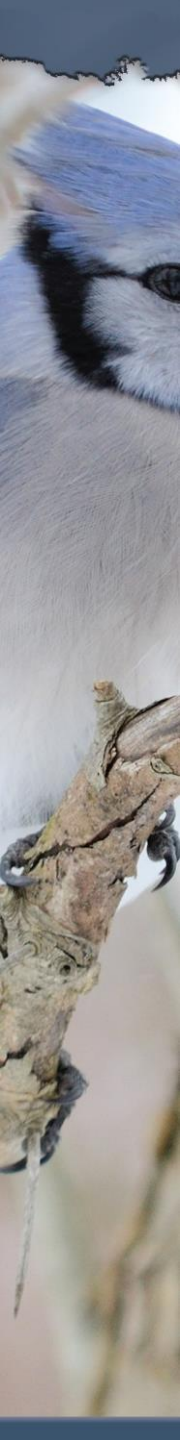
- A finally clause is executed even if a return statement is executed in the try or catch blocks.
- If an exception is thrown in the try block but not caught, then the finally clause is still executed.

```
try {
    Protect one or more statements here.
}
finally {
    Perform any actions here common to whether or not
    an exception is thrown.
}
```

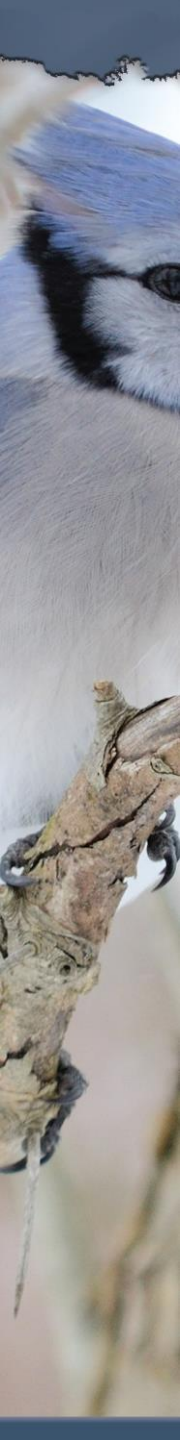


Defining new exceptions

- Extend **`RuntimeException`** for an unchecked or **`Exception`** for a checked exception.
- Define new types to give better diagnostic information.
  - Include reporting and/or recovery information.



```
public class NoMatchingDetailsException extends Exception
{
    private String key;

    public NoMatchingDetailsException(String key)
    {
        this.key = key;
    }

    public String getKey()
    {
        return key;
    }

    public String toString()
    {
        return "No details matching '" + key +
               "' were found.";
    }
}
```

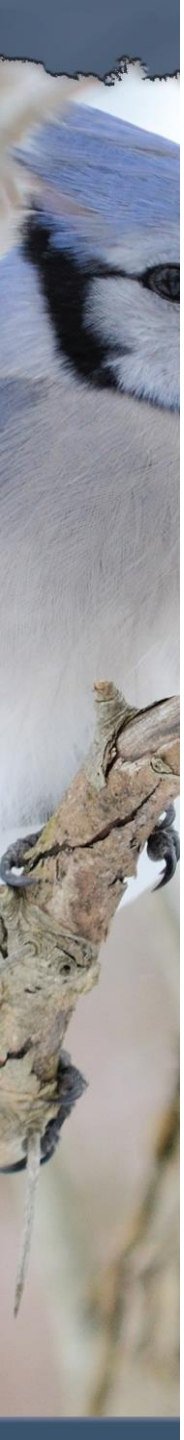


Error recovery

- Methods should take note of error notifications.
  - Check return values of calling methods.
  - Don't 'ignore' exceptions.
- Include code to attempt recovery.
  - Will often require a loop.

Attempting recovery

```
// Try to save the address book.
boolean successful = false;
int attempts = 0;
do {
    try {
        contacts.saveToFile(filename);
        successful = true;
    }
    catch(IOException e) {
        System.out.println("Unable to save to " + filename);
        attempts++;
        if(attempts < MAX_ATTEMPTS) {
            filename = an alternative file name;
        }
    }
} while(!successful && attempts < MAX_ATTEMPTS);

if(!successful) {
    Report the problem and give up;
}
```



```
public class MyException extends Exception
{

        public MyException(String msg)
        {
           super("details of the problem:"+ msg);
        }
 }
 ------------------------------------------------

public class Test {

static int  divide(int  first, int second) throws MyException
{
  if(second==0)
    throw new MyException("can't be divided by zero");

  return first/second;
 }

 public static void main(String[] args) {
  try {
         System.out.println(divide(4,0));
      }
   catch (MyException exc) {
        exc.printStackTrace();
    }
  }
}
```

Stack Unwinding

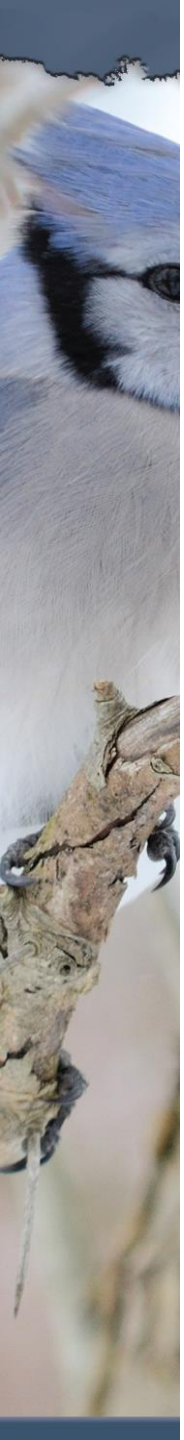
```
Exception thrown in method3

java.lang.Exception: Exception thrown in method3
        at UsingExceptionsStack.method3(UsingExceptionsStack.java:35)
        at UsingExceptionsStack.method2(UsingExceptionsStack.java:31)
        at UsingExceptionsStack.method1(UsingExceptionsStack.java:27)
        at UsingExceptionsStack.main(UsingExceptionsStack.java:4)

Stack trace from getStackTrace:
Class                   File                          Line    Method
UsingExceptionsStack    UsingExceptionsStack.java     35      method3
UsingExceptionsStack    UsingExceptionsStack.java     31      method2
UsingExceptionsStack    UsingExceptionsStack.java     27      method1
UsingExceptionsStack    UsingExceptionsStack.java     4       main
```

```java
// Fig. 13.1: DivideByZeroNoExceptionHandling.java
// An application that attempts to divide by zero.
import java.util.Scanner;

public class DivideByZeroNoExceptionHandling
{
   // demonstrates throwing an exception when a divide-by-zero occurs
   public static int quotient( int numerator, int denominator )
   {
      return numerator / denominator; // possible division by zero
   } // end method quotient

   public static void main( String args[] )
   {
      Scanner scanner = new Scanner( System.in ); // scanner for input

      System.out.print( "Please enter an integer numerator: " );
      int numerator = scanner.nextInt();
      System.out.print( "Please enter an integer denominator: " );
      int denominator = scanner.nextInt();

      int result = quotient( numerator, denominator );
      System.out.printf(
         "\nResult: %d / %d = %d\n", numerator, denominator, result );
   } // end main
} // end class DivideByZeroNoExceptionHandling
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at DivideByZeroNoExceptionHandling.quotient(
           DivideByZeroNoExceptionHandling.java:10)
        at DivideByZeroNoExceptionHandling.main(
           DivideByZeroNoExceptionHandling.java:22)
```

```java
// Fig. 13.2: DivideByZeroWithExceptionHandling.java
// An exception-handling example that checks for divide-by-zero.
import java.util.InputMismatchException;
import java.util.Scanner;

public class DivideByZeroWithExceptionHandling
{
   // demonstrates throwing an exception when a divide-by-zero occurs
   public static int quotient( int numerator, int denominator )
      throws ArithmeticException
   {
      return numerator / denominator; // possible division by zero
   } // end method quotient

   public static void main( String args[] )
   {
```

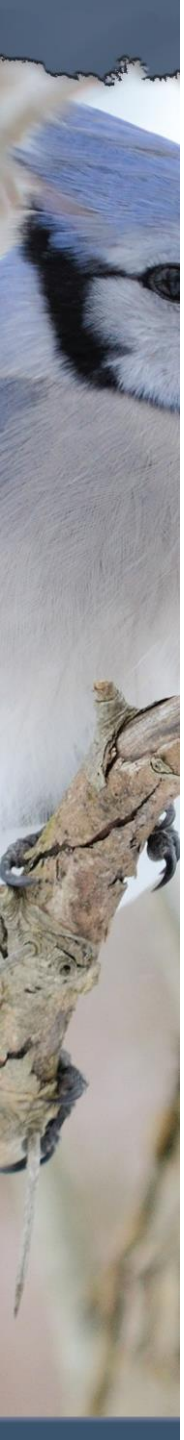
```
17          Scanner scanner = new Scanner( System.in ); // scanner for input
18          boolean continueLoop = true; // determines if more input is needed
19
20          do
21          {
22             try // read two numbers and calculate quotient
23             {
24                System.out.print( "Please enter an integer numerator: " );
25                int numerator = scanner.nextInt();
26                System.out.print( "Please enter an integer denominator: " );
27                int denominator = scanner.nextInt();
28
29                int result = quotient( numerator, denominator );
30                System.out.printf( "\nResult: %d / %d = %d\n", numerator,
31                   denominator, result );
32                continueLoop = false; // input successful; end looping
33             } // end try
34             catch ( InputMismatchException inputMismatchException )
35             {
36                System.err.printf( "\nException: %s\n",
37                   inputMismatchException );
38                scanner.nextLine(); // discard input so user can try again
39                System.out.println(
40                   "You must enter integers. Please try again.\n" );
41             } // end catch
42             catch ( ArithmeticException arithmeticException )
43             {
44                System.err.printf( "\nException: %s\n", arithmeticException );
45                System.out.println(
46                   "Zero is an invalid denominator. Please try again.\n" );
47             } // end catch
48          } while ( continueLoop ); // end do...while
49       } // end main
50    } // end class DivideByZeroWithExceptionHandling
```



```
Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14
```

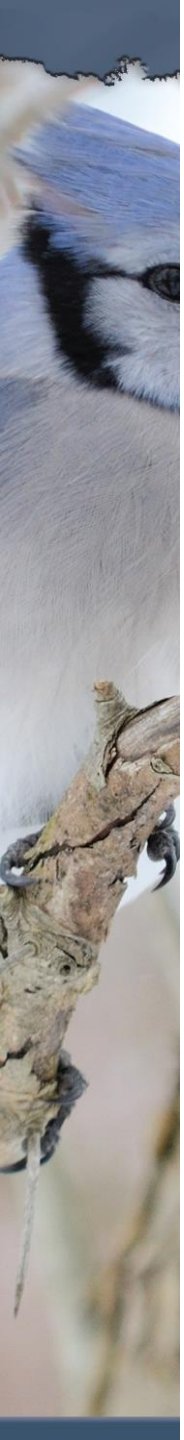
```
Please enter an integer numerator: 100
Please enter an integer denominator: 0

Exception: java.lang.ArithmeticException: / by zero
Zero is an invalid denominator. Please try again.

Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: hello

Exception: java.util.InputMismatchException
You must enter integers. Please try again.

Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14
```



Review

- Runtime errors arise for many reasons.
  - An inappropriate client call to a server object.
  - A server unable to fulfill a request.
  - Programming error in client and/or server.



Review

- Runtime errors often lead to program failure.
- Defensive programming anticipates errors – in both client and server.
- Exceptions provide a reporting and recovery mechanism.