Objects First With Java
A Practical Introduction Using BlueJ

Handling errors

Main concepts to be covered

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

Some causes of error situations

- Incorrect implementation.
  - Does not meet the specification (you want the median, but you compute the mean value, e.g.).
- Inappropriate object request.
  - E.g., invalid index (index value outside valid range).
- Inconsistent or inappropriate object state.
  - An object is used in a way not anticipated by the class designer, arising through class extension or inheritance, e.g.

Not always programmer error

- Techniques of Ch. 6 (testing & debugging) help us to identify and eliminate logical (programmers') errors *before* runtime.
- However, errors often arise from the environment:
  - Incorrect URL entered to a browser.
  - Network interruption.
- File processing is particular error-prone:
  - Missing files.
  - Lack of appropriate permissions.

Exploring errors

- Example: *address-book*.
  - Application that stores contact details (name, address, and phone number).
  - Contact details indexed by both name and phone number.
  - Main classes: `AddressBook`, `ContactDetails`.
- Two aspects:
  - Error reporting.
  - Error handling.

Defensive programming

- Client-server interaction: `AddressBook` as a server object, acting on clients' requests only.
  - Should a server assume that clients are well-behaved?
  - Or should it rather assume that clients are potentially hostile (➔ defensive style)?
- Opposite extremes, with significant differences in implementation required.

Issues to be addressed

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

An example

- Try to remove an entry from the empty object.
- A runtime error results (due to method calls on a `null` value).
  - Whose ‘fault’ is this?
- Anticipation and prevention are preferable to apportioning blame.

Argument values

- Arguments represent a major ‘vulnerability’ for a server object.
  - Constructor arguments initialize state.
  - Method arguments often contribute to behaviour.
- Hence: valid argument values are vital!
- Argument checking is one defensive measure.

Checking the key

```
public void removeDetails(String key)
{

        ContactDetails details = book.get(key);
        book.remove(details.getName());
        book.remove(details.getPhone());
        numberOfEntries--;

}
```

What if `key` is not a valid key?

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--;
    }
}
```

Better: check whether `key` really is a valid key!

Server error reporting

- Circumventing illegal arguments is one thing – to avoid them in the future, someone should be informed.
- How and to whom to report illegal arguments?
  - To the user?
    - Is there any human user? What to do if not?
    - Can they solve the problem? (Imagine a bank customer at the teller machine getting a `NullPointerException`?!)
  - To the client object?
    - Return a diagnostic value (typically via return value).
    - *Throw an exception (see later).*

Return values

- If void: change to boolean.
- Otherwise:
  - Return `null` value instead of a valid object.
  - Use an additional value not occurring normally.
- Different purposes:
  - Inform the client about success (the server's responsibility/fault).
  - Inform the client about invalid requests (the client's responsibility/fault).

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;
    }
}
```

Now the client knows if something went wrong, and the client *can* react (via `if` statements in its code, e.g.)!

Client responses

- Possibility 1: Test the return value ☺.
  - Attempt recovery on error.
  - Avoid program failure.
- Possibility 2: Ignore the return value ☹.
  - Cannot be prevented.
  - Likely to lead to program failure.
- Therefore, exceptions are preferable to the simple use of return values.

Exception-throwing principles

- A special Java language feature:
  - Special exception classes.
- No ‘special’ return value needed.
- Errors cannot be ignored in/by the client.
  - The normal flow-of-control is interrupted.
- Specific recovery actions are encouraged: application stops otherwise.

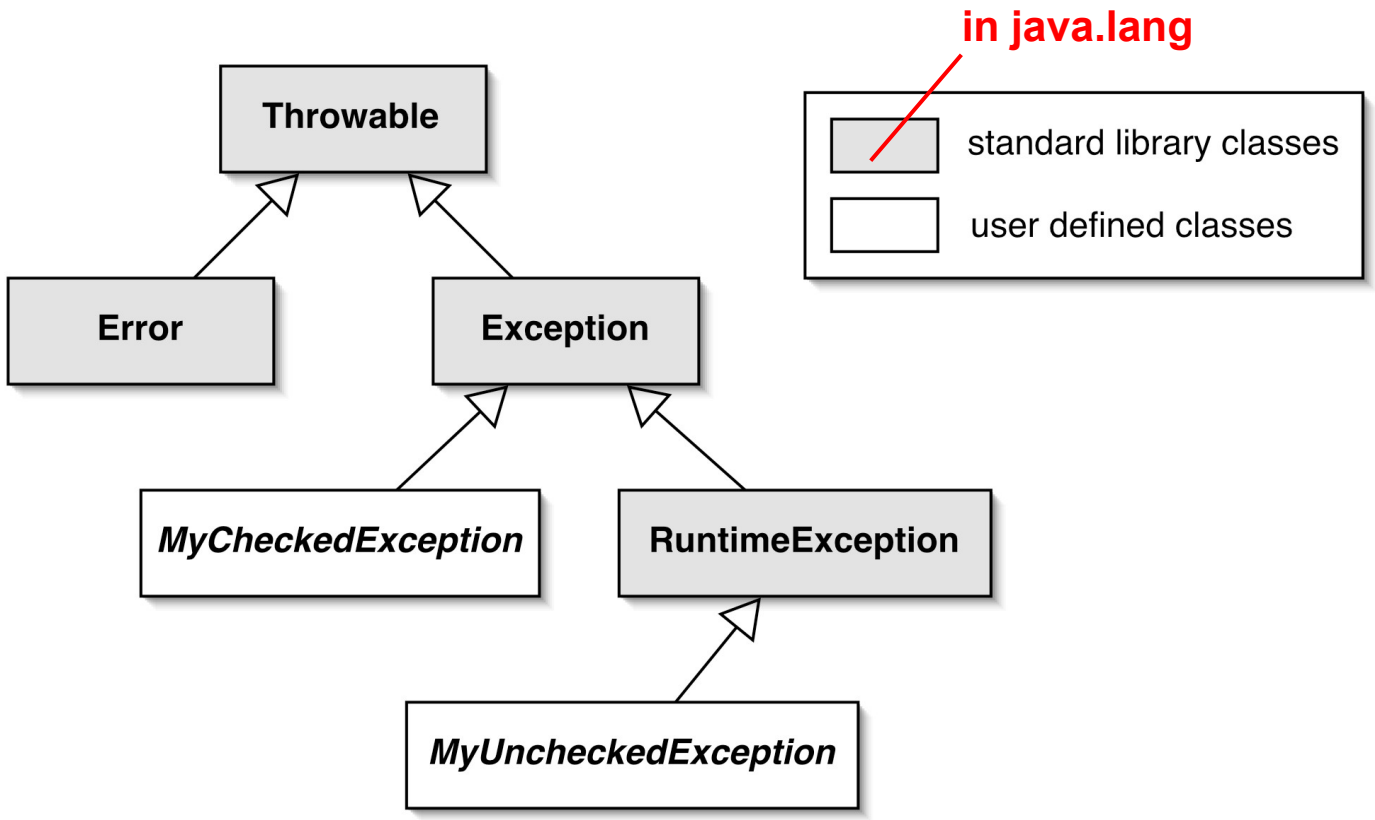
Throwing an exception

```
/**
 * 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 NullPointerException if the key is null.
 */
public ContactDetails getDetails(String key)
{
    if(key == null){
        throw new NullPointerException(
                              "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 (writing to a full disk, e.g.).
  - Where recovery may be possible.
  - Client should be forced to check.
- Unchecked exceptions
  - Subclass of `RuntimeException`
  - Use for unanticipated failures (there should never be any failure in normal operation).
  - Where recovery is unlikely (program error).

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.
  - If not caught: termination!

Unchecked exceptions

- The easiest to use from the programmer's point of view.
- 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 NullPointerException(
                            "null key in getDetails");
    }
    if(key.trim().length() == 0) {
        throw new IllegalArgumentException(
                            "Empty key passed to getDetails");
    }
    return (ContactDetails) 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.");
    }
}
```

If no valid initial state is possible (due to inappropriate constructor arguments, e.g.), no object should be created!

Checked Exceptions

- Are meant to be caught.
- The compiler ensures that their use is tightly controlled.
  - In both server and client.
  - More compiler action, more rules.
- Used properly, failures may be recoverable.

The throws clause

- Methods throwing a checked exception must include a throws clause (here `throws`, not `throw`!):

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

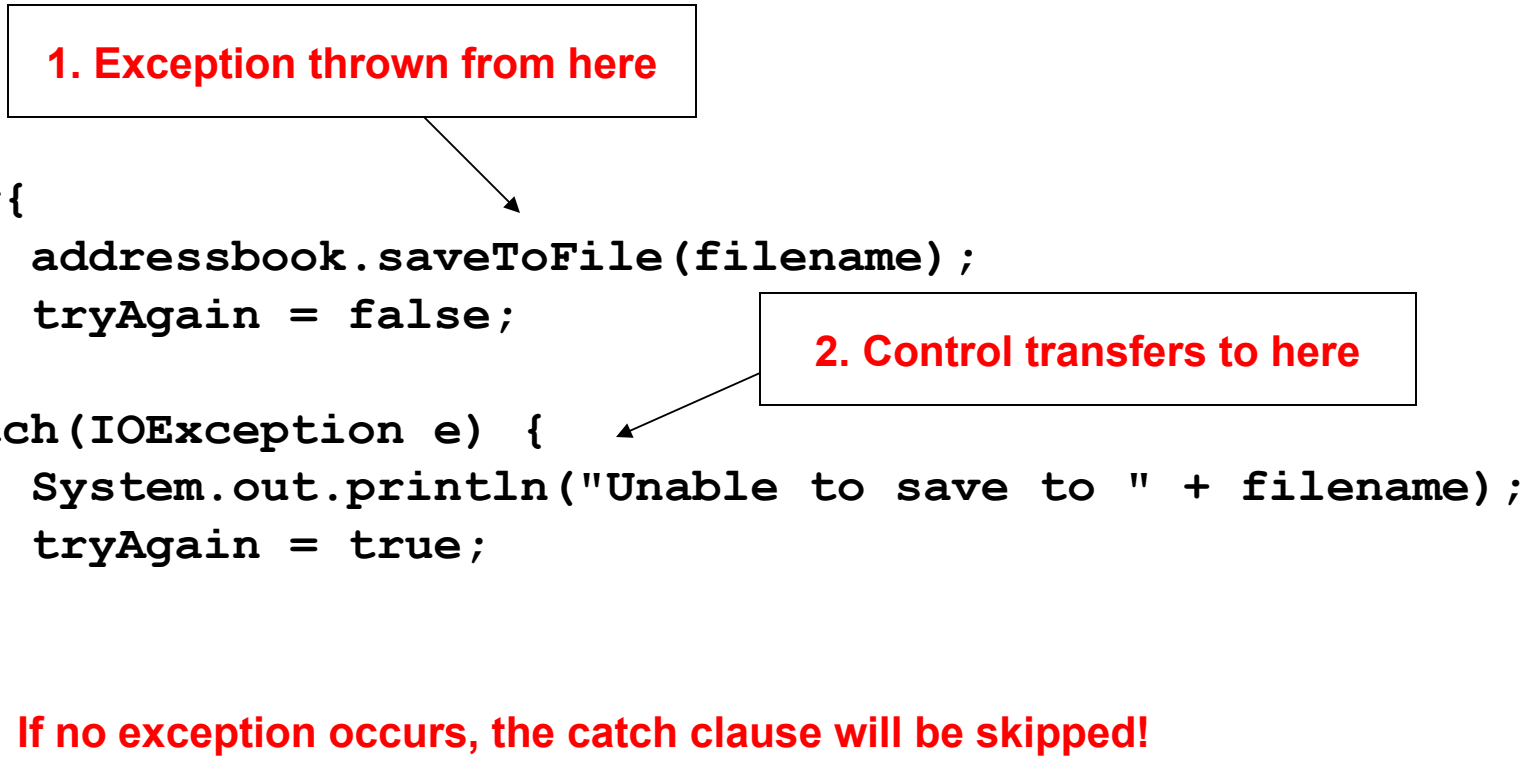
- Use of throws clauses in unchecked exceptions is allowed, but not required.

The try statement

- Clients catching an exception must protect the call with a try statement (make provision for dealing with the exception):

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

The try statement




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

If no exception occurs, the catch clause will be skipped!

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.
    ...
}
```

All types of checked exceptions must be listed in the throws clause, separated by commas.

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.
- An uncaught or propagated exception still exits via the finally clause.
- The finally clause is optional.

Defining new exceptions

- Extend either `Exception` (checked) or `Runtime-Exception` (unchecked).
- Define new types to give better (i.e. more detailed/specific) diagnostic information.
  - Include reporting and/or recovery information (i.e. information beyond the inclusion of a fixed diagnostic string).

```
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.";
    }
}
```

constructor receives `key`

dedicated accessor for `key`

diagnostic string including `key`

Assertions

- Used for *internal* consistency checks.
  - E.g. object state following mutation.
- Used during development and normally removed in production version.
  - E.g. via a compile-time option.
- Java has an *assert statement*.

Java Assertion Statement

- Two forms available:
  - **`assert`** *boolean-expression*
  - **`assert`** *boolean-expression* **`:`** *expression*
- The boolean-expression expresses something that should be true at this point.
- An AssertionError is thrown if the assertion is false.

Assert Statement

```
public void removeDetails(String key)
    {
        if(key == null){
            throw new IllegalArgumentException("...");
        }
        if(keyInUse(key)) {
            ContactDetails details = book.get(key);
            book.remove(details.getName());
            book.remove(details.getPhone());
            numberOfEntries--;
        }
        assert !keyInUse(key);
        assert consistentSize() :
                "Inconsistent book size in removeDetails";
    }
```

Guidelines for Assertions

- They are not an alternative to throwing exceptions.
- Use for internal checks.
- Remove from production code.
- Don't include normal functionality:

```
// Incorrect use:
assert book.remove(name) != null;
```

Error recovery

- Clients should take note of error notifications.
  - Check return values.
  - Don't 'ignore' exceptions – or just include a `println` command in the catch clause.
- Include code to attempt recovery.
  - Will often require a loop over the try and catch clauses.

Attempting recovery

```
// Try to save the address book.
boolean successful = false;
int attempts = 0;
do {
    try {
        addressbook.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;
}
```

Error recovery

- Anticipating an error and recovering from it requires a more complex flow of control.
- Crucial: statements in the catch clause.
- Recovery often involves trying again – without a guaranteed success.
- Resign at some point!

Error avoidance

- Clients can often use server query methods to avoid errors.
  - More robust clients mean servers can be more trusting.
  - Unchecked exceptions can be used.
  - Simplifies client logic.
- May increase client-server coupling.

Text input-output

- Input-output is particularly error-prone.
  - It involves interaction with the external environment.
- The `java.io` package supports input-output.
- `java.io.IOException` is a checked exception.

Readers, writers, streams

- Readers and writers (classes) deal with textual input (i.e. text files).
  - Based around the `char` type.
- Streams (classes) deal with binary data (i.e. binary files).
  - Based around the `byte` type.
- The *address-book-io* project illustrates textual IO.

Text output

- Use the `FileWriter` class.
  - Open a file.
  - Write to the file.
  - Close the file.
- Failure at any point results in an `IOException`.
- Reasons for a failure might be:
  - User has no permission to write a file.
  - Given file name does not match a valid location.

Text output

```
try {
    FileWriter writer = new FileWriter("name of file");
    while(there is more text to write) {
        ...
        writer.write(next piece of text);
        ...
    }
    writer.close();
}
catch(IOException e) {
    something went wrong with accessing the file
}
```

Text input

- Use the `FileReader` class.
- Augment with `BufferedReader` for line-based input.
  - Open a file.
  - Read from the file.
  - Close the file.
- Failure at any point results in an `IOException`.

Text input

```
try {
    BufferedReader reader = new BufferedReader(
                            new FileReader("filename"));
    String line = reader.readLine();
    while(line != null) {
        // do something with line
        line = reader.readLine();
    }
    reader.close();
}
catch(FileNotFoundException e) {
    // the specified file could not be found
}
catch(IOException e) {
    // something went wrong with reading or closing
}
```

Review

- Runtime errors arise for many reasons.
  - An inappropriate client call to a server object.
  - A server unable to fulfil 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.