

Handling errors

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.

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.
- A runtime error results.
 - 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 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.

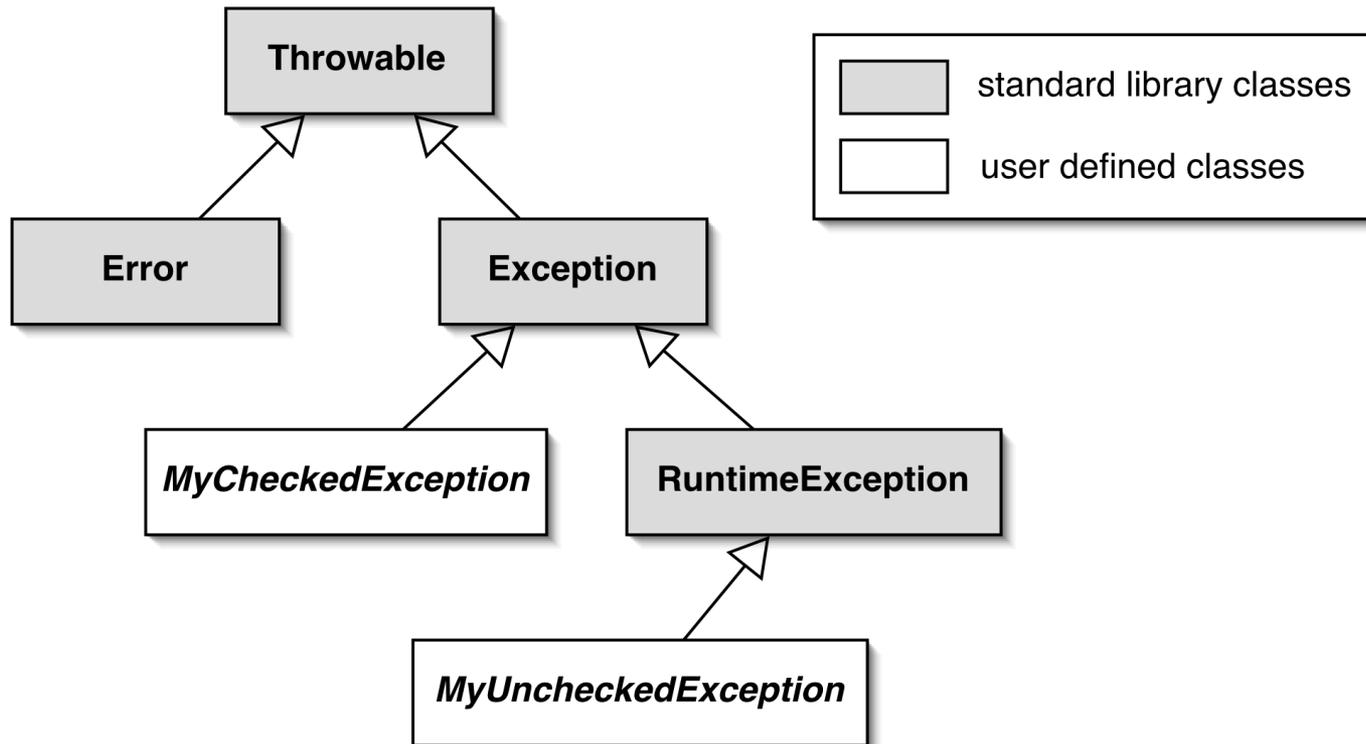
Throwing an exception

```
/**
 * 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

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 - a Java 7 feature

```
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.

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.";
    }
}
```

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 expression evaluates to 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.
- 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;
}
```

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.

Avoiding an exception

```
// Use the correct method to put details
// in the contacts list.
if (contacts.keyInUse (details.getName () ||
    contacts.keyInUse (details.getPhone ())) {
    contacts.changeDetails (details);
}
else {
    contacts.addDetails (details);
}
```

The `addDetails` method could now throw an *unchecked* exception.

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.

File-based input-output

- Input-output is particularly error-prone because it involves interaction with the external environment.
- The `java.io` package supports input-output.
- `java.io.IOException` is a checked exception.
- Java 7 adds the `java.nio` packages.

File and Path

- `java.io.File` provides information about files and folders/directories.
- `java.nio.file.Path` is the Java 7 alternative.
- `File` is a class; `Path` is an interface.
- The `Files` and `Paths` classes in `java.nio.file` are also used.

Readers, writers, streams

- Readers and writers deal with textual input.
 - Based around the `char` type.
- Streams deal with binary data.
 - Based around the `byte` type.
- The *address-book-io* project illustrates textual I/O.

File output

- The three stages of file output.
 - Open a file.
 - Write to the file.
 - Close the file (but see ARM in Java 7).
- Failure at any point results in an **IOException**.
- Use **FileWriter** for text files.

Text output to file

```
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  
}
```

See the *weblog-analyzer* project

Try-with-resource - Java 7

- Used for ensuring ‘resources’ are closed after use.
- Removes need for explicit closure on both successful and failed control flows.
- Also known as ‘automatic resource management’ (ARM).

Try-with-resource

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

**No close() call required in either clause.
See the *weblog-analyzer-v7* project.**

Text input from file

- Use the `FileReader` class.
- Augment with `BufferedReader` for line-based input.
 - Open a file.
 - Read from the file.
 - Close the file (but see ARM in Java 7).
- Failure at any point results in an `IOException`.

Text input from file

```
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
}
```

[See tech-support-io](#)

Text input from file - Java 7

- **BufferedReader** created via static **newBufferedReader** method in the **java.nio.file.Files** class.
- Requires a **Charset** from **java.nio.charset**, e.g.:
 - "US-ASCII"
 - "ISO-8859-1"

Text input - Java 7

```
Charset charset =
    Charset.forName("US-ASCII");
Path path = Paths.get("file");
try(BufferedReader reader =
    Files.newBufferedReader(path, charset)) {
    use reader to process the file
}
catch(FileNotFoundException e) {
    deal with the exception
}
catch(IOException e) {
    deal with the exception
}
```

Text input from the terminal

- `System.in` maps to the terminal:
 - Its type is `java.io.InputStream`
- It is often wrapped in a `java.util.Scanner`.
- `Scanner` with `File` is an alternative to `BufferedReader` with `FileReader`.

Scanner: parsing input

- **Scanner** supports *parsing* of textual input.
 - `nextInt`, `nextLine`, etc.
- Its constructors support **String**, **File** and **Path** (Java 7) arguments.

Review

- Input/output is an area where errors cannot be avoided.
- The environment in which a program is run is often outside a programmer's control.
- Exceptions are typically *checked*.

Review

- Key classes for text input/output are **FileReader**, **BufferedReader**, **FileWriter** and **Scanner**.
- Binary input/output involves **Stream** classes.
- Java 7 introduces the **Path** interface as an alternative to **File**.
- Java 7 introduces try-with-resource.