Design for Robustness

Exceptions:

- Notify caller of exceptional circumstance (usually operational failure)
- Semantics
  - An exception propagates up the function
  - call stack until main() is reached (terminates program) or until the exception is caught
- Sources of exceptions:
  - Program throwing an exception
  - Exception thrown by the Java Virtual Machine

Java: Finally:

The finally block always runs after try/catch

Design choice: checked and unchecked:

- Unchecked exception: any subclass of RuntimeException
  - Error which is highly unlikely and/or unrecoverable
- check exception: any subclass of Exception that is not a subclass of RuntimeException
  - Error that every caller should be aware of and explicitly decide to handle or pass on
- Return values(- 1, false, null): If failure is common and expected possibility

Creating and throwing your own exceptions:

- Methods must declare any checked exceptions they might throw
- If your class extends java.lang.Throwable you can throw it:
  -
```
if (some__){
  throw new customException("Blah blah");
 }
```

Benefits of exceptions:

- High level summary of error and stack trace
  - Compare: core dumped in C
- Can't forget to handle common failure modes
  - Compare: using a flag or special return value
- Can optionally recover from failure
  - Compare: calling System.exit()
- Improve code structure
  -
  - Separate routine operations from error
  - handling
- Allow consisten clean
- up in both normal and exceptional operation

Guide for exceptions:

- Catch and handle all checked exceptions
  - Unless there is no good way
- Use runtime exceptions for programming error
- Other
  - Don't catch an exception without (at least somewhat) handling the error
  - When you throw an exception describe the error
  - if you re
  - throw an exception, always include the original exception as the clause

Modular Protection:

- Errors and bugs unavoidable but exceptions should not leak across modules (methods, classes) if possible
- Good modules handles exceptional conditions locally

  - Local input validation and local exception handling where possible
  - Explicit interfaces with clear pre/post conditions
- Explicitly documented and checked exceptions where exceptional conditions may propagate between modules
  - Information hiding -encapsulation of critical code (likely bugs, likely exceptions)

Problems when testing (sub -)systems:

- User interfaces and user interactions
  - Users click buttons, interpret output
  - Waiting/timing issues
- Test data vs. real data
- Testing against big infrastructure (database, web services,..)
- Testing with side effects (e.g. printing and mailing documents)
- Nondeterministic behavior
- Concurrency

Testing strategies in environments:

- Separate business logic and data representation from GUI for testing
- Test algorithms locally without large environment using stubs
- Advantage of stubs
  - Create deterministic response
  - Can reliably simulate spurious states (network error)
  - Can speed up test execution
  - Can simulate functionality not yet implemented
- Automate

Scaffolding:

- Catch bugs early: Before client code or service available
- Limit scope of debugging: Localize errors

- Improve coverage
  - System level tests may only cover 70
  - Simulate unusual error conditions
- Validate internal interface/API designs
  - Simulate clients in advance of their developement
  - Simulate services in advance of their development
- Capture developer intent (in absence of specification documentation)
  - A test suite formally captures elements of design intent
  - Developer documentation
- Improve low
- level design
  - Early attention to ability to test