Outline

• General concepts about dealing with errors and failures

• Assertions: what, why, how
  – for things you believe will/should never happen

• Exceptions: what, how
  – how to throw, catch, and declare exceptions in Java
  – subtyping of exceptions
  – checked vs. unchecked exceptions

• Exceptions: why *in general*  
  – for things you believe are bad and should rarely happen  
  – and many other style issues

• Alternative with trade-offs: Returning special values

• Summary and review
Not all “errors” should be failures

Some “error” cases:

1. Misuse of your code
   - e.g., precondition violation
   - **should** be a failure (i.e., made visible to the user)

2. Errors in your code vs reasoning
   - e.g., representation invariant fails to hold
   - **should** be a failure

3. Unexpected resource problems
   - e.g., missing file, server offline, …
   - not an error in the sense above (... these are not bugs)
   - **should not** be a failure (i.e., do try to recover)
What to do when failing

Fail fast and fail friendly

Goal 1: *Prevent harm*
  – stop before anything worse happens
  – (do still need to perform cleanup: close open resources etc.)

Goal 2: *Give information about the problem*
  – failing quickly helps localize the defect
  – a good error message is important for debugging
Outline

• General concepts about dealing with errors and failures

• Assertions: what, why, how
  – for things you believe will/should never happen

• Exceptions: what, how
  – how to throw, catch, and declare exceptions in Java
  – subtyping of exceptions
  – checked vs. unchecked exceptions

• Exceptions: why in general
  – for things you believe are bad and should rarely happen
  – and many other style issues

• Alternative with trade-offs: Returning special values

• Summary and review
Square root

// requires: x >= 0
// returns: approximation to square root of x
public double sqrt(double x) {
    ...
}

Square root with assertion

```java
// requires: x >= 0
// returns: approximation to square root of x
public double sqrt(double x) {
    assert x >= 0.0;
    double result;
    ... compute result ...
    assert Math.abs(result*result - x) < .0001;
    return result;
}

• These two assertions serve different purposes

(Note: the Java library Math.sqrt method returns NaN for x<0. We use different specifications in this lecture as examples.)
Outline

• General concepts about dealing with errors and failures

• Assertions: what, why, how
  – for things you believe will/should never happen

• Exceptions: what, how
  – how to throw, catch, and declare exceptions in Java
  – subtyping of exceptions
  – checked vs. unchecked exceptions

• Exceptions: why in general
  – for things you believe are bad and should rarely happen
  – and many other style issues

• Alternative with trade-offs: Returning special values

• Summary and review
Square root, specified for all inputs

// throws: NegativeArgumentException if x < 0
// returns: approximation to square root of x
public double sqrt(double x)
    throws NegativeArgumentException {
    if (x < 0)
        throw new NegativeArgumentException();
    ...
}

• **throws** is part of a method signature: “it might happen”
  – comma-separated list
  – like @modifies, promises are in what is **not listed**
• **throw** is a statement that actually causes exception-throw
  – immediate control transfer [like return but different]
Using try-catch to handle exceptions

```java
public double sqrt(double x) throws NegativeArgumentException {
    ...
}
```

Client code:
```
try {
    y = sqrt(...);
    ... other statements ...
} catch (NegativeArgumentException e) {
    e.printStackTrace(); // or other actions
}
```

- Handled by nearest *dynamically* enclosing `try/catch`
  - top-level default handler: print stack trace & crash
try {
    code...
} catch (FileNotFoundException fnfe) {
    code to handle a file not found exception
} catch (IOException ioe) {
    code to handle any other I/O exception
} catch (Exception e) {
    code to handle any other exception
}

- A `SocketException` would match the second block
- An `ArithmeticException` would match the third block
- (Subsequent catch blocks need not be supertypes like this)
Throwing and catching

• Executing program has a stack of currently executing methods
  – dynamic: reflects runtime order of method calls
  – no relation to static nesting of classes, packages, etc.
• When an exception is thrown, control transfers to nearest method with a matching catch block
  – if none found, top-level handler used
• Exceptions allow non-local error handling
  – a method many levels up the stack can handle a deep error
Code Paths with Exceptions

Three potential paths through the code below:

```java
try {
    y = foo(...);
    ... more code ...
} catch (Type name) {
    ... code to handle the exception ...
}
```

1. `sqrt` returns normally
2. `sqrt` throws an exception caught by this catch
3. `sqrt` throws an exception not caught here
The **finally** block

**finally** block is always executed
- whether an exception is thrown or not

```java
try {
    y = foo(...);
    ... more code ...
} catch (Type name) {
    ... code to handle the exception ...
} finally {
    ... code to run after the try or catch finishes
}
```
What **finally** is for

**finally** is used for common “must-always-run” or “clean-up” code
- avoids duplicated code in catch branch[es] and after
- avoids having to catch all exceptions

```java
try {
    // ... write to out; might throw exception
} catch (IOException e) {
    System.out.println("Caught IOException: " + e.getMessage());
} finally {
    out.close();
}
```
(Abridged) Exception Hierarchy
Java’s checked/unchecked distinction

Checked exceptions (style: for special cases / abnormal cases)
- callee must declare in signature (else type error)
- client must either catch or declare (else type error)
  - even if you can prove it will never happen at run time, the type system does not “believe you”
- guaranteed to be a matching enclosing catch at runtime

Unchecked exceptions (style: for never-expected)
- library has no need to declare
- client has no need to catch
- these are subclasses of:
  - RuntimeException
  - Error (rarely caught)
Outline

• General concepts about dealing with errors and failures

• Assertions: what, why, how
  – for things you believe will/should never happen

• Exceptions: what, how
  – how to throw, catch, and declare exceptions in Java
  – subtyping of exceptions
  – checked vs. unchecked exceptions

• Exceptions: why *in general*
  – for things you believe are bad and should rarely happen
  – and many other style issues

• Alternative with trade-offs: Returning special values

• Summary and review
Two distinct uses of exceptions

• Errors that should be failures
  – unexpected (ideally, should not happen at all)
  – should be rare with high quality client and library
  – can be the client’s fault or the library’s
  – often unrecoverable

• Special cases (a.k.a. exceptional cases)
  – expected, just not the common case
  – possibly unpredictable or unpreventable by client
Handling exceptions

- Errors that should be failures
  - usually can’t recover
  - unchecked exceptions the better choice (avoids much work)
  - if condition not checked, exception propagates up the stack
    - top-level handler prints the stack trace

- Special cases
  - take special action and continue computing
  - should always check for this condition
  - should handle locally by code that knows how to continue
  - checked exceptions the better choice
Checked vs. unchecked

• No perfect answer to the question “should clients be forced to catch (or declare they throw) this exception?”
  – Java provided both options

• Advantages to checked exceptions:
  – Static checking of callee: only declared exceptions are thrown
  – Static checking of caller: exception is caught or declared

• Disadvantages:
  – impedes implementations and overrides (can’t add exceptions)
    • prevents truly giving no promises when @requires is false
  – often in your way when prototyping
  – have to catch or declare even if the exception is not possible
Propagating an exception

// returns: x such that ax^2 + bx + c = 0
// throws: NegativeArgumentException if no real soln exists
double solveQuad(double a, double b, double c)
    throws NegativeArgumentException {
    // No need to catch exception thrown by sqrt
    return (-b + sqrt(b*b - 4*a*c)) / (2*a);
}

Aside: does “negative argument” make sense to the caller?
Why catch exceptions locally?

Problems:

1. Failure to catch exceptions often violates modularity
   - call chain: A -> IntSet.insert -> IntList.insert
   - IntList.insert throws some exception
     • implemeter of IntSet.insert knows how list is being used
     • implemeter of A may not even know that IntList exists

2. Possible that a method on the stack may think that it is handling an exception raised by a different call

Alternative: catch it and throw again
   - “chaining” or “translation”
   - do this even if the exception is better handled up a level
   - makes it clear to reader of code that it was not an omission
Exception translation

// returns: x such that ax^2 + bx + c = 0
// throws: NotRealException if no real solution exists

double solveQuad(double a, double b, double c)
{
    try {
        return (-b + sqrt(b*b - 4*a*c)) / (2*a);
    } catch (NegativeArgumentException e) {
        throw new NotRealException(); // "chaining"
    }
}

class NotRealException extends Exception {
    NotRealException() { super(); }
    NotRealException(String message) { super(message); }
    NotRealException(Throwerable cause) { super(cause); }
    NotRealException(String msg, Throwable c) { super(msg, c); }
}
Don’t ignore exceptions

*Effective Java* Tip: Don't ignore exceptions

Empty catch block is poor style

```java
try {
    readFile(filename);
} catch (IOException e) {} // silent failure
```

At a minimum, print out the exception so you know it happened

– and exit if that’s appropriate for the application

```java
} catch (IOException e) {
    e.printStackTrace();
    System.exit(1);
}
```
Outline

• General concepts about dealing with errors and failures

• Assertions: what, why, how
  – for things you believe will/should never happen

• Exceptions: what, how in Java
  – how to throw, catch, and declare exceptions
  – subtyping of exceptions
  – checked vs. unchecked exceptions

• Exceptions: why in general
  – for things you believe are bad and should rarely happen
  – and many other style issues

• Alternative with trade-offs: Returning special values

• Summary and review
Informing the client of a problem

Special value:
- `null` for `Map.get`
- `-1` for `indexOf`
- `NaN` for `sqrt` of negative number

Advantages:
- can be less verbose than try/catch machinery

Disadvantages:
- error-prone: callers forget to check, forget spec, etc.
- need “extra” result: doesn’t work if every result could be real
  - example: if a map could store `null` keys
- has to be propagated manually one call at a time

General Java style advice: exceptions for exceptional conditions
Outline

• General concepts about dealing with errors and failures

• Assertions: what, why, how
  – For things you believe will/should never happen

• Exceptions: what, how in Java
  – How to throw, catch, and declare exceptions
  – Subtyping of exceptions
  – Checked vs. unchecked exceptions

• Exceptions: why in general
  – For things you believe are bad and should rarely happen
  – And many other style issues

• Alternative with trade-offs: Returning special values

• Summary and review
Exceptions: review

Use an **assertion** for internal consistency checks that should not fail
  – when checking at runtime is possible

Use **only** a **precondition** when
  – used in a context in which calls can be checked via reasoning
  – but checking at runtime would be prohibitive
    • e.g., requiring that a list be sorted

Use an **exception** when
  – used in a dynamic / unpredictable context (client can’t predict)
  – for exceptional cases only

Use a **special value** when
  – it is a common case (not really exceptional)
  – clients are likely (?) to remember to check for it
Exceptions: review, continued

Use *checked* exceptions most of the time
  – static checking is helpful! (*tools*, inspection, & testing)

Avoid checked exceptions if there is probably no way to recover

Handle exceptions sooner rather than later

Good reference: Effective Java chapter
  – a whole chapter: exception-handling design matters!