# CSE 331
# Software Design & Implementation

Hal Perkins

Autumn 2013

Exceptions and Assertions

(Slides by Mike Ernst and David Notkin)

# Failure causes

Partial failure is inevitable

- Goal:  prevent complete failure
- Structure your code to be reliable and understandable

Some failure causes:

1. Misuse of your code
   - Precondition violation
2. Errors in your code
   - Bugs, representation exposure, many more
3. Unpredictable external problems
   - Out of memory
   - Missing file
   - Memory corruption

# Avoiding errors

A precondition prohibits misuse of your code

- Adding a precondition weakens the spec

This ducks the problem

- Does not address errors in your own code
- Does not help others who are misusing your code

Removing the precondition requires specifying the behavior (often a good thing, but there are tradeoffs)

- Strengthens the spec
- Example: specify that an exception is thrown

# Defensive programming

Check
- precondition
- postcondition
- representation invariant
- other properties that you know to be true

Check statically via reasoning (& tools)

Check dynamically at run time via assertions

```
assert index >= 0;
assert items != null : "null item list argument"
assert size % 2 == 0 : "Bad size for " +
                                   toString();
```

Write the assertions as you write the code

Prefer assertions with descriptive messages

# Enabling assertions

In Java, assertions can be enabled or disabled at runtime without recompiling the program

Command line:

- **`java -ea`** runs code with assertions enabled
- **`java`** runs code with assertions disabled (default ☹)

Eclipse:

- Select Run>Run Configurations… then add **`-ea`** to VM arguments under (x)=arguments tab
- (Demo and details in sections)

# When *not* to use assertions

Don't clutter the code

```
x = y + 1;
assert x == y + 1;            // useless, distracting
```

Don't perform side effects

```
assert list.remove(x);  // won't happen if disabled

// Better:
boolean found = list.remove(x);
assert found;
```

Turn them off in rare circumstances (production code(?) )

Most assertions should always be enabled

# `assert` and `checkRep()`

CSE 331's **`checkRep()`** is another dynamic check

Strategy: use **`assert`** in **`checkRep()`** to test and fail with meaningful traceback/message if trouble found

Be sure to enable asserts when you do this!

Asserts should be enabled always for CSE 331 projects

We will enable them for grading

# Expensive `checkRep()` tests

Detailed checks can be too slow in production

Even if asserts are disabled, an **`assert`** in a deep loop nest can take lots of time to do nothing

But complex tests can be very helpful, particularly during testing/debugging (let the computer find problems for you!)

No perfect answers; suggested strategy for **`checkRep`**:

– Create a static, global “debug” or “debugLevel” variable

– Run expensive tests when this is enabled

– Turn it off in graded / production code if tests are too expensive

Often helpful: put expensive / complex tests in separate methods and call as needed

# What to do when something goes wrong

Something goes wrong: an assertion fails
    (or would have failed if an assertion were there)
Fail early, fail friendly
Goal 1: Give information about the problem
    To the programmer – a good error message is key!
    To the client code
Goal 2: Prevent harm from occurring
    Abort: inform a human
        Perform cleanup actions, log the error, etc.
    Re-try:
        Problem might be transient
    Skip a subcomputation:
        Permit rest of program to continue
    Fix the problem (usually infeasible)
        External problem: no hope; just be informative
        Internal problem: if you can fix, you can prevent

# Square root without exceptions

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

# Square root with assertion

```
// 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;
}
```

# Square root, specified for all inputs

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

# Using try-catch to handle exceptions

```
// throws: IllegalArgumentException if x < 0
// returns: approximation to square root of x
public double sqrt(double x)
                    throws IllegalArgumentException
   ...
```

Client code:

```
try {
  y = sqrt(-1);
} catch (IllegalArgumentException e) {
  e.printStackTrace(); // or take some other action
}
```

Handled by **`catch`** associated with nearest dynamically enclosing **`try`**

Top-level default handler:  stack trace, program terminates

# Throwing and catching

At run time, your program has a stack of currently executing methods

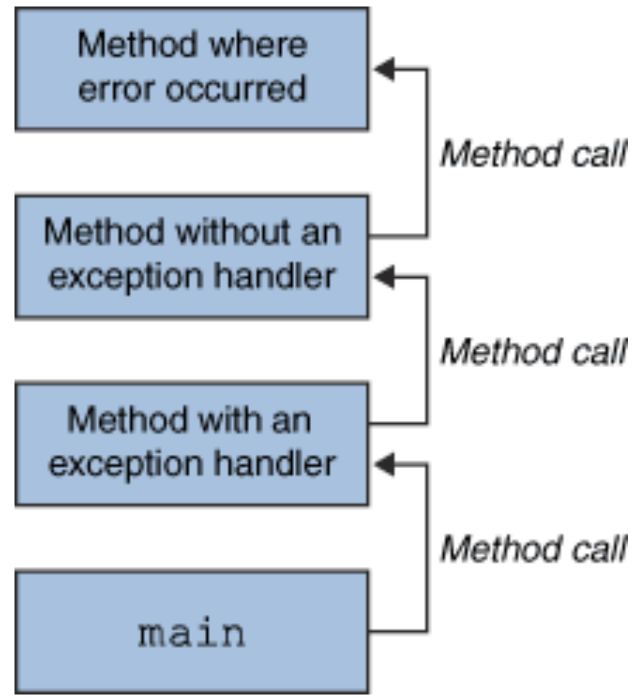
- Dynamic: reflects runtime order of method calls
- No relation to static nesting of classes or packages or such

When an exception is thrown, control transfers to nearest method with a matching catch block

- If none found, top-level handler prints a stack trace & terminates

Exceptions allow non-local error handling

- A method many levels up the stack can handle a deep error



# The `finally` block

**`finally`** block is always executed

- Whether an exception is thrown or not

```
try {
  code…
} catch (type name) {
    code… to handle the exception
} finally {
    code… to run after the try or catch finishes
}
```

# One use for the **`finally`** block

**`finally`** is often used for common, “must-always-run” or “clean-up” code

```
try {
  // ... write to out;  might throw exception
} catch (IOException e) {
  System.out.println("Caught IOException: "
                           + e.getMessage());
} finally {
  out.close();
}
```

# Propagating an exception

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

How can clients know if a set of arguments to **`solveQuad`** is illegal?

# Why catch exceptions locally?

- Failure to catch exceptions violates modularity
  - Call chain: A → IntegerSet.insert → IntegerList.insert
  - IntegerList.insert throws an exception
    - Implementer of IntegerSet.insert knows how list is being used
    - Implementer of A may not even know that IntegerList exists
- Procedure on the stack may think that it is handling an exception raised by a different call
- Better alternative: catch it and throw it 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) throws NotRealException
{
  try {
    return (-b + sqrt(b*b - 4*a*c)) / (2*a);
  } catch (IllegalArgumentException e) {
    throw new NotRealException();
  }
}

class NotRealException extends Exception {
  NotRealException() { super(); }
  NotRealException(String message) { super(message); }
  NotRealException(Throwable cause) { super(cause); }
  NotRealException(String msg, Throwable c) { super(msg, c); }
}
```

Exception chaining:

```
throw new NotRealException(e);
```

# Exceptions as non-local control flow

```
void compile() {
  try {
    parse();
    typecheck();
    optimize();
    generate():
  } catch (RuntimeException e) {
    Logger.log("Failed: " + e.getMessage());
  }
}
```

Not common – usually bad style, particularly at small scale
Java/C++, etc. exceptions are expensive if thrown/caught
Reserve exceptions for exceptional conditions

# Informing the client of a problem

Special value:

- `null` – Map.get
- `-1` – indexOf
- `NaN` – sqrt of negative number

Problems with using special value:

- Hard to distinguish from real results
- Error-prone: what if the programmer forgets to check result?
  - Needs to be a value that cannot be a legal result and best if it will trigger a failure later
- Ugly

A better solution (but not always?): exceptions

- Not highly recommended for routine control flow

# Two distinct uses of exceptions

- Failures
  - Unexpected
  - Should be rare with well-written client and library
  - Can be the client's fault or the library's
  - Usually unrecoverable
- Special results
  - Expected
  - Unpredictable or unpreventable by client

# Handling exceptions

Failures

- Usually can’t recover
- If the condition is not checked, the exception propagates up the stack
- The top-level handler prints the stack trace

Special results

- Take special action and continue computing
- Should always check for this condition
- Should handle locally

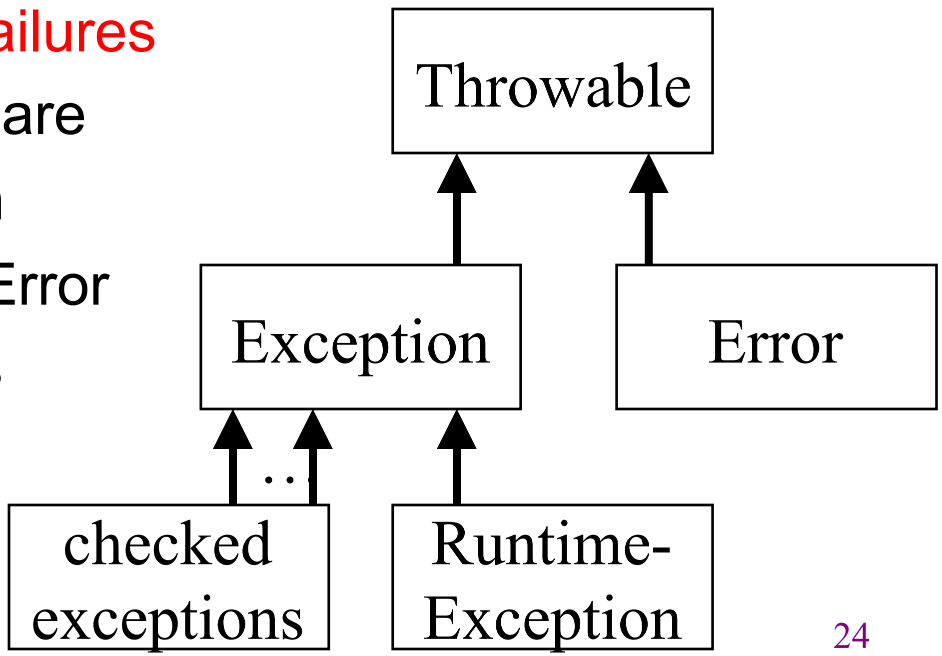
# Java exceptions for failures and for special cases

Checked exceptions for special cases

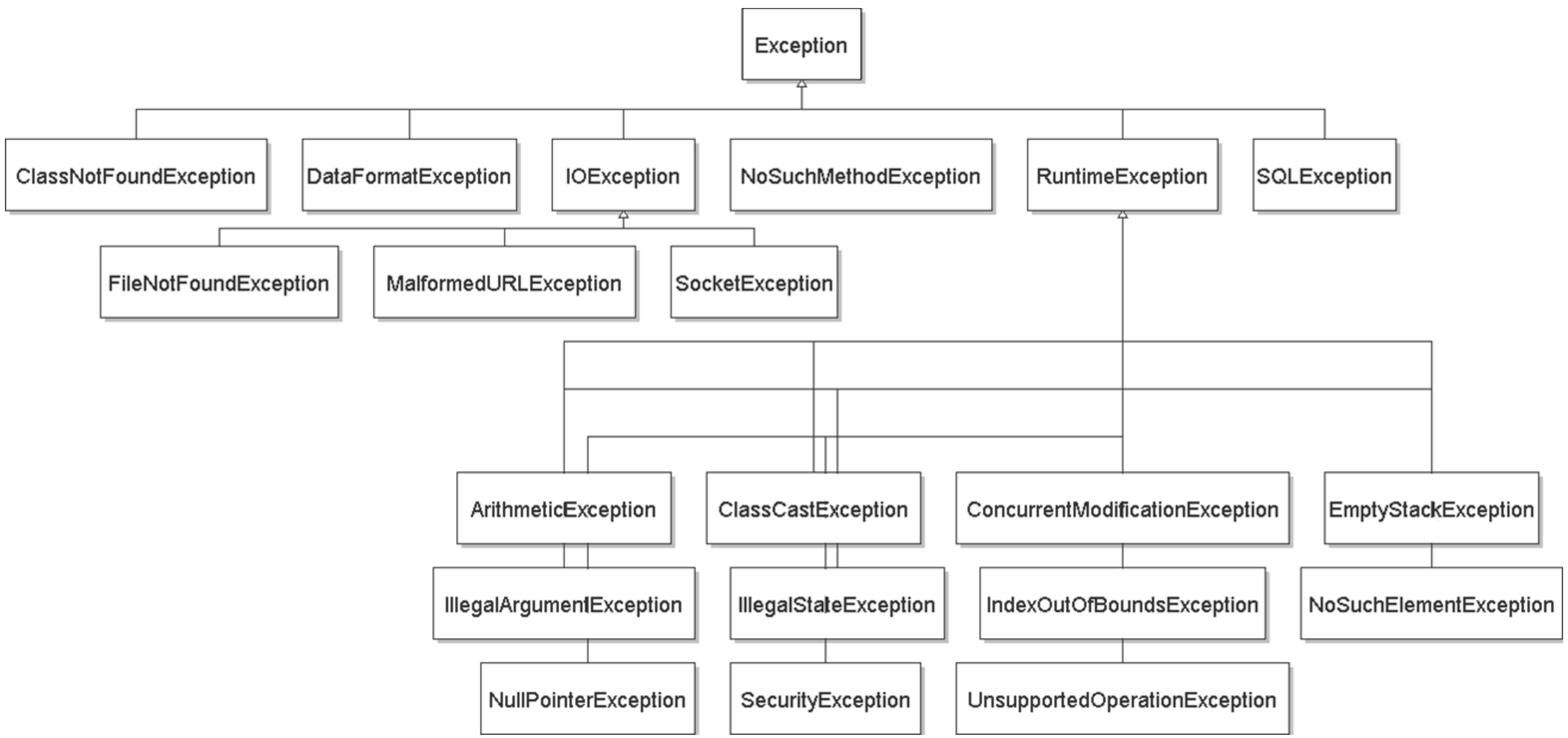
- Library: must declare in signature
- Client: must either catch or declare
  - Even if you can prove it will never happen at run time
- There is guaranteed to be a dynamically enclosing catch

Unchecked exceptions for failures

- Library: no need to declare
- Client: no need to catch
- RuntimeException and Error
  - and their subclasses



# exception hierarchy



# Catching with inheritance

```
try {
  code…
} catch (FileNotFoundException fnfe) {
    code… to handle the 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

# Avoid proliferation of checked exceptions

Unchecked exceptions are better if clients will usually write code that ensures the exception will not happen

- i.e., there is a convenient and inexpensive way to avoid it
- The exception reflects unanticipatable failures

Otherwise use a checked exception

- Must be caught and handled – prevents program defects
- Checked exceptions should be locally caught and handled
- Checked exceptions that propagate long distances suggests bad design (failure of modularity)

Java sometimes uses **`null`** (or NaN, etc.) as a special value

- Acceptable if used judiciously, carefully specified
- Easy to forget to check

# Don’t ignore exceptions

Effective Java Tip #65: Don't ignore exceptions

An empty catch block is (a common) poor style – often done to get code to compile or hide an error

```
try {
  readFile(filename);
} catch (IOException e) {}  // do nothing on error
```

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

```
} catch (IOException e) {
  e.printStackTrace();    // just in case
}
```

# Exceptions: review

Use an exception when
- Used in a broad or unpredictable context
- Checking the condition is feasible

Use a precondition when
- Checking would be prohibitive
  - E.g., requiring that a list be sorted
- Used in a narrow context in which calls can be checked

Avoid preconditions because
- Caller may violate precondition
- Program can fail in an uninformative or dangerous way
- Want program to fail as early as possible

How do preconditions and exceptions differ, for the client?

# Exceptions: review, continued

Use checked exceptions most of the time

Handle exceptions sooner rather than later

Not all exceptions are errors

- A program structuring mechanism with non-local jumps (expensive, should be rare)
- Used for exceptional (unpredictable) circumstances (IOException, similar things)

# Exceptions vs assertions

Both can be used to check for errors. No universal consensus on which to use where. More general guidelines:

Exceptions

- Use for defensive programming, particularly checks at public API interfaces when assertions could be disabled
- Use to signal when client can or could recover, or otherwise handle a situation

Assertions

- Use for internal consistency checks – things that should "never happen"
- Use to catch things that are bugs and should be fixed
- Use for expensive checks during development/debugging

Good reference on all of this: Bloch *Effective Java*, ch. 9