

---

# CSE 331

# Software Design & Implementation

Hal Perkins

Autumn 2012

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

How would you categorize these?

Failure of a subcomponent

No return value (e.g., list element not found, division by zero)

# 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

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 size % 2 == 0 : "Bad size for " +  
                        toString();
```

Write the assertions as you write the code

# 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:

Pick Run>Run Configurations... then add `-ea` to VM arguments under (x)=arguments tab

# 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

Expensive `checkRep ()`s

Detailed checks can be too slow in production

Even if asserts are disabled, if `checkRep` has a deep loop nest it takes lots of time to do nothing

No great answers

Maybe call `checkRep` only if asserts are enabled?

(e.g., `assert checkRep ()`)

Maybe comment out expensive tests if needed?

# What to do when something goes wrong

---

Something goes wrong: an assertion fails (or would have failed if it 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:  $x \geq 0$   
// returns: approximation to square root of  $x$   
public double sqrt(double  $x$ ) {  
    ...  
}
```

# Square root with assertion

---

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

# Square root, specified for all inputs; Using try-catch

---

```
// 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();
    ...
}
```

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

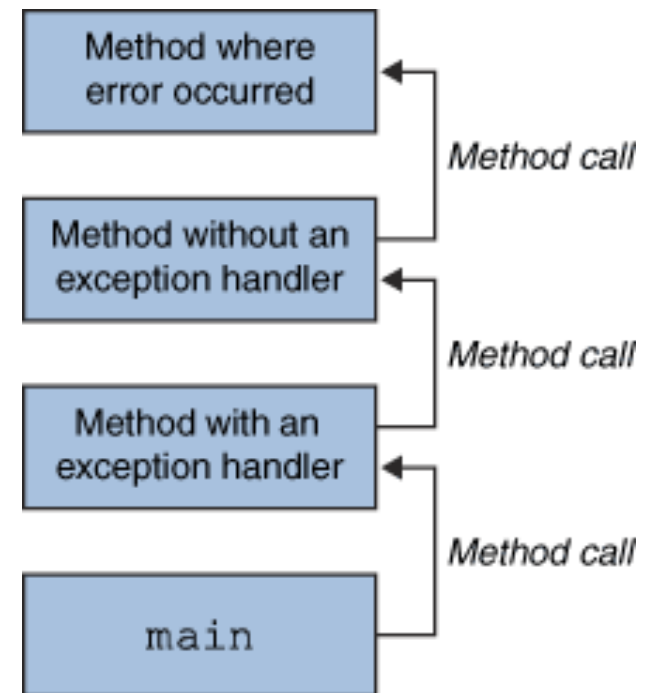
- No relation to static nesting of classes or packages or such

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

---

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

finally is often used for common “clean-up” code

```
try {
    // ... read from out; might throw
} 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?

# 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 to use exceptions for regular control flow. 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

Less efficient

A better solution: exceptions

# 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

# Why catch exceptions locally?

---

Failure to catch exceptions violates modularity

Call chain:  $A \rightarrow \text{IntegerSet.insert} \rightarrow \text{IntegerList.insert}$

$\text{IntegerList.insert}$  throws an exception

Implementer of  $\text{IntegerSet.insert}$  knows how list is being used

Implementer of  $A$  may not even know that  $\text{IntegerList}$  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

# 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

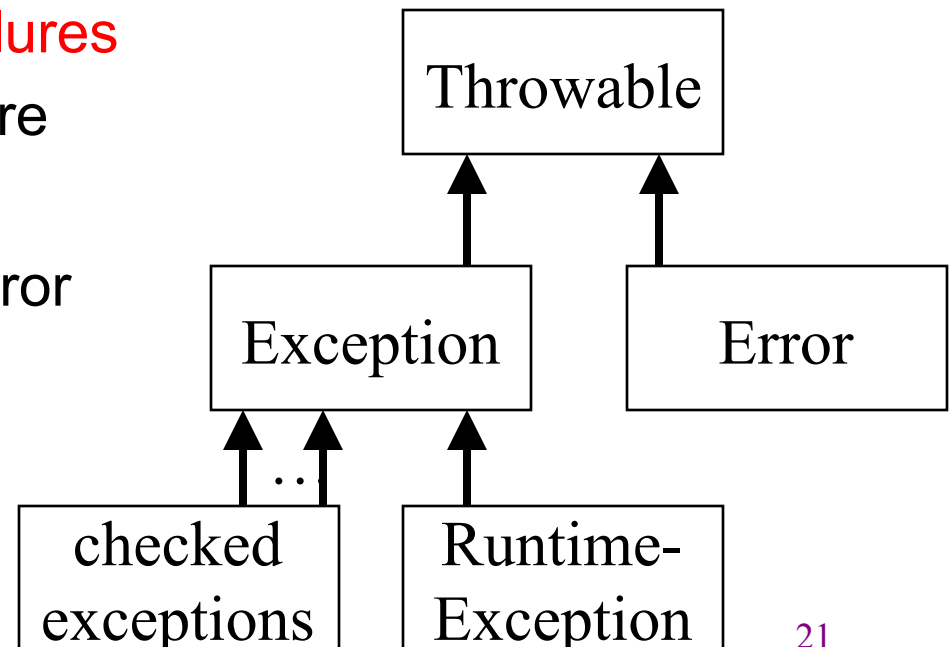
There is guaranteed to be a dynamically enclosing catch

## Unchecked exceptions for failures

Library: no need to declare

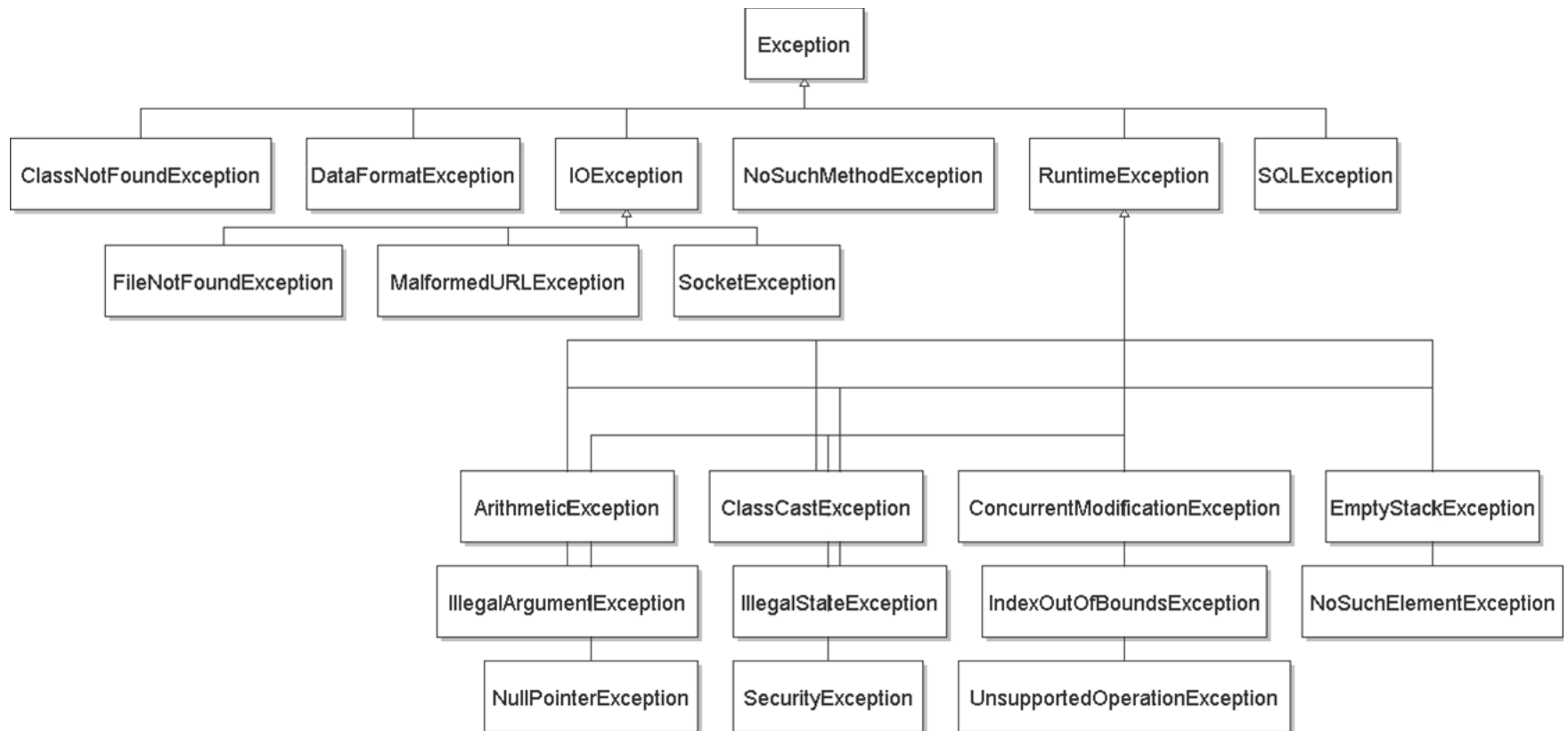
Client: no need to catch

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

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 in 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 in 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

# Exceptions vs assertions

---

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

## Exceptions

- Use for defensive programming, particularly checks at public API interfaces

- 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