Everybody plays the fool …
…there’s no exception to the rule

CSE 331
SOFTWARE DESIGN & IMPLEMENTATION
EXCEPTIONS AND ASSERTIONS

Autumn 2011

Failure: String.reverse(“sneppah tihs”)

- Galloping Gurdy
- Industrial: Chernobyl, Three Mile Island, Bhopal, Fukushima Daiichi, …
- Aerospace: Challenger, Columbia, Soyuz I, Apollo I, Ariane 5
- Aviation: AF4590 (Concorde), AA587
- Construction: Hyatt Regency walkway (KC, 1981), the last Husky Stadium remodel (1987)
- And many, many more

- Henry Petroski ⓘ has written broadly on the role of failure in engineering

Software errors are inevitable, too

- Not famous software failures, but how to think more about reducing the chances of failure and the consequences of failure
  - Reducing the chances of failure is usually considered software reliability
  - Reducing the consequences of failure is usually considered software safety
  - “A car that doesn’t start is unreliable; a car that doesn’t stop is unsafe.:
- Software failure causes include
  - Misuse of your code (e.g., precondition violation)
  - Errors in your code (e.g., bugs, representation exposure, …re)
  - Unpredicted/unpredictable external problems (e.g., 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 and possibly 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

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); // modifies behavior if
                       // assertion checking disabled
// Better:
boolean found = list.remove(x);
assert found;
```

- Turn them off in rare circumstances (e.g., production code)
  - Eclipse: set in compiler preferences
  - Command line
    - `java -ea` runs Java with assertions enabled
    - `java` runs Java with assertions disabled (default)
  - Most assertions should always be enabled

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 (and perform or make it easier for them to perform cleanup actions, loging the error, etc.)
  - Re-try: problem might be transient
  - Skip a subcomputation: permit rest of program to continue
  - Fix the problem during execution (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) {
  double result;
  ... // compute result
  assert (Math.abs(result*result – x) < .0001);
  return result;
}
```

Square root, specified for all inputs

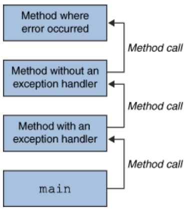
```
// throws: IllegalArgumentException if x < 0
// returns: approximation to square root of x
public double sqrt(double x) throws IllegalArgumentException
{
  if (x < 0)
    throw new IllegalArgumentException();
  ...
}

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

Exception caught by `catch` associated with nearest dynamically enclosing `try`

- Top-level default handler: stack trace, program terminates
- Note: this is really a form of inversion-of-control

Throwing and catching

- At any time, your program has an active call stack of methods
  - The call stack is **not** the same as nesting of classes or packages or such – it reflects which methods called which methods during this specific execution
- When an exception is thrown, the JVM looks up the call stack until it finds a method with a matching catch block for it
  - If one is found, control jumps back to that method
  - If none is found, the program crashes
- Exceptions allow non-local error handling
  - A method many levels up the stack can handle a deep error


Method where error occurred
Method call
Method without an exception handler
Method call
Method with an exception handler
Method call
main

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 whether 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();
  }
}
```

Exception chaining

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

Special values

- Special values are often used to inform a client of a problem
  - `null` `Map.get`
  - `-1` `indexOf`
  - `NaN` `sqrt` of negative number
- Problems with using special value
  - Hard to distinguish from real results
  - Error-prone
    - The programmer may forget to check the result?
    - The value should not be legal – should cause a failure later
  - Ugly
  - Often inefficient

Can use exceptions instead

- Special results through exceptions
  - Expected
  - Unpredictable or unpreventable by client
  - Take special action and continue computing
  - Should always check for this condition
  - Should handle locally

Exceptions for failure

- Different from use for special values
- Failures are
  - Unexpected
  - Should be rare with well-written client and library
  - Can be the client's fault or the library's
  - Usually unrecoverable
  - Usually can't recover
  - If the condition is not checked, the exception propagates up the stack
  - The top-level handler prints the stack trace

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

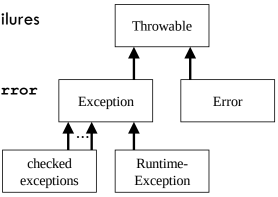
```
try {
  // ... read from out;  might throw
} catch (IOException e) {
  System.out.println("Caught IOException: "
                     + e.getMessage());
} finally {
  out.close();
}
```

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" – show earlier
  - 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

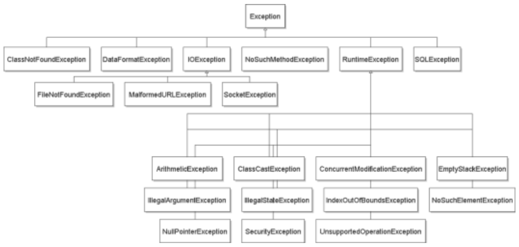
Java **`throwable`** hierarchy

- □ 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

20



UW CSE331 Autumn 2011

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
  - But too easy to forget to check

Ignoring 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

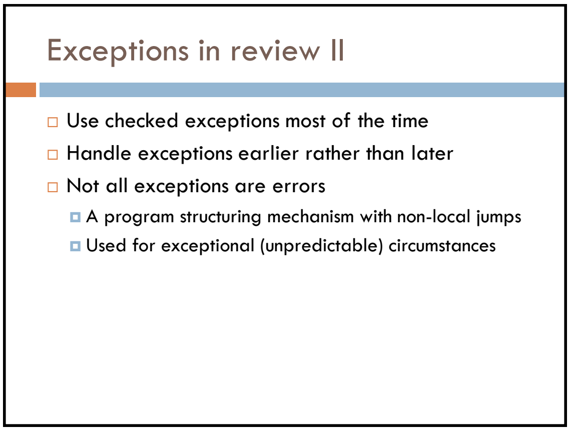
```
try {
  readFile(filename);
} catch (IOException e) {}  // do nothing on error
```

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

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

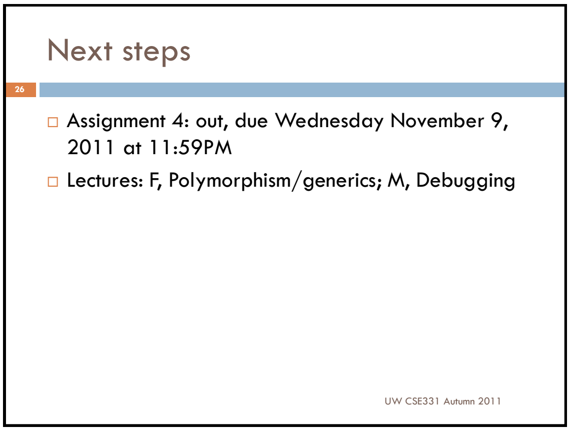
Exceptions in review I

- □ 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 II

- Use checked exceptions most of the time
- Handle exceptions earlier rather than later
- Not all exceptions are errors
  - A program structuring mechanism with non-local jumps
  - Used for exceptional (unpredictable) circumstances



Next steps

26

- Assignment 4: out, due Wednesday November 9, 2011 at 11:59PM
- Lectures: F, Polymorphism/generics; M, Debugging

UW CSE331 Autumn 2011