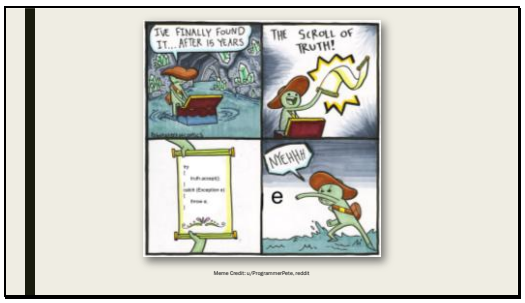
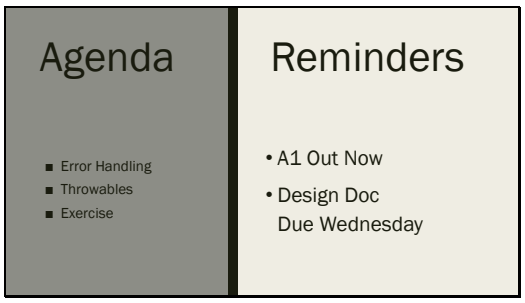
Slide 1



Slide 2

Slide 3



Slide 4

Slide 5

Motivating Example

```
List o = { "Latte", "Espresso" };

   int x = o.indexOf("Lattee")

             x ⇒ -1
```

If I spell the item I'm looking for wrong, indexOf returns -1

Slide 6

```
// many lines of code later


while (x != 0) {
   dispenseCoffee();
   x--;
}
```

This code is going to call dispenseCoffee() a few billion times. Your coffee cup is going to overflow.

Slide 7

```
// many lines of code later


while (x != 0) {
   dispenseRadiation();
   x--;
}
```

This is suddenly much less funny

Slide 8

Slide 9

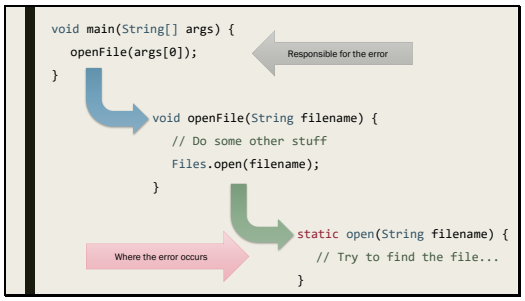
```
try {
   [redacted]
} catch (e) {
   // Silently ignore error and hope it goes away
}
```

It's tempting to view Exceptions as the problem that needs to be fixed. You may even be tempted to wrap everything in a try catch to make the Exceptions stop. But Exceptions are just the symptom of an underlying issue – that your code is broken. And ignoring that fact only invites more trouble.

Slide 10

Continuing execution in an invalid state can be catastrophic

Slide 11



Making things harder is that often times, the code that encounters the error (eg reading a nonexistent file) and the code that is responsible for the error (eg specifying which file to read) don't live in the same place and may not have even been written by the same person or team.

We need to solve this long range communication problem.

Slide 12

Wish List

- Represent abnormal execution status
- Delegate responsibility for handling problems
- Prevent execution in invalid state

As such, here are some features we'd like in our way of modelling problems

Slide 13

Slide 14

Error Codes

Case Study: OpenGL (C graphics library)

```
glutCreateWindow("Tutorial 01");
if (glGetError() != GL_NO_ERROR) { ... }
```

Some old, low level libraries will provide either a function or a return value that is some number if successful and a different number for errors

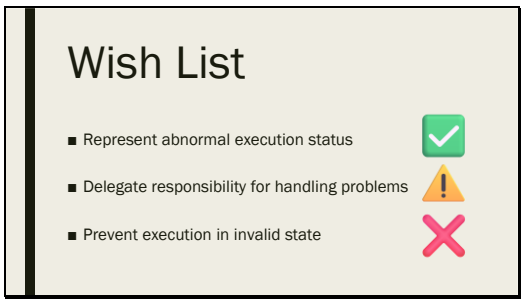
Slide 15

Error Handling

```
glutInit(&argc, argv);
     if (glGetError() != GL_NO_ERROR) { ... }
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA);
     if (glGetError() != GL_NO_ERROR) { ... }
glutInitWindowSize(1024, 768);
     if (glGetError() != GL_NO_ERROR) { ... }
glutCreateWindow("Tutorial 01");
     if (glGetError() != GL_NO_ERROR) { ... }
```

You end up checking for these error codes all over the place

Slide 16

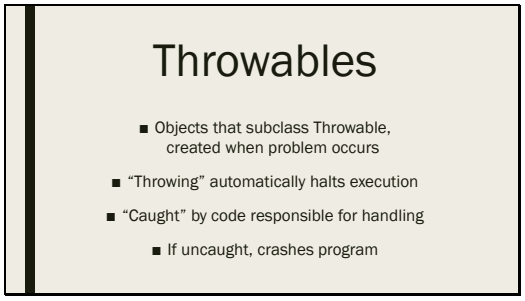


We do have a way of representing problems. However, if the place to handle these problems lives further away from the callsite, it's still not easy to delegate. And there's nothing that this does if the programmer forgets to check for the status code.

Slide 17

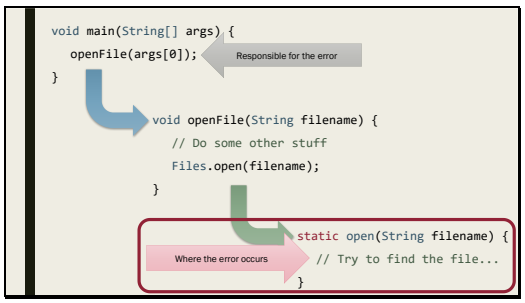
So instead this is how Java does it

Slide 18



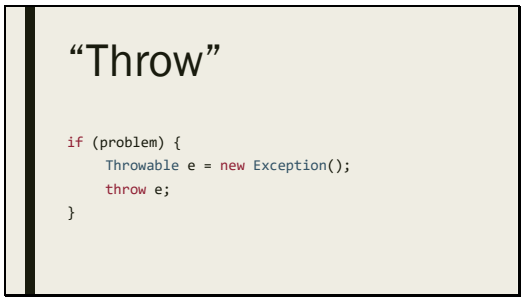
An object of type Throwable is a special object that can be "thrown" to halt execution due to an issue

Slide 19



Let's start with the perspective of the code where the problem occurs

Slide 20

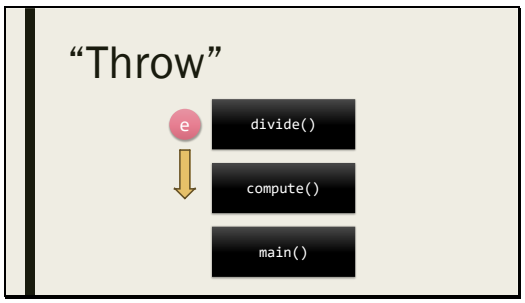


When a problem occurs that you'd like to pass off, create a new object of type Throwable like any other, and then pass that object to the "throw" keyword.

Slide 21

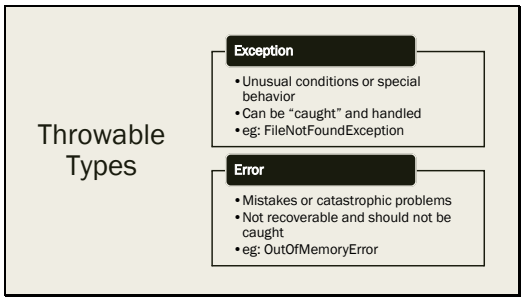
Often, we skip the step of assigning it to a variable first. However, don't forget the "new" as we are instantiating a new object.

Slide 22

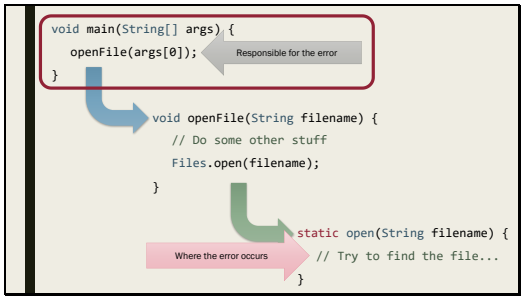


The throwable is then "thrown" down the call stack, waiting to be caught by the first method that tries to "catch" it. If it's not caught, the program crashes.

Slide 23



Slide 24



Now to look at the perspective of the code responsible for handling the error (for example, asking the user to pick a different file)

Slide 25

“Catch”

```
try {
    codeThatCanThrowException();
} catch (Exception e) {
    // Deal with the error
}
```

When you want to take responsibility for and handle the problems that may occur in another block of code, wrap that code in a “try” and you can catch the exception with a catch block.

Slide 26

Try vs. Catch

Where error occurs

```
void codeThatCanThrowException() {
   if (problem) {
      throw new Exception();
   }
}
```

Responsible for error

```
try {
     codeThatCanThrowException();
} catch (Exception e) {
     // Deal with the error
}
```

So to review, the place where a problem happens *throws* the exception, and the place responsible for handling the problem *catches* the exception. Often times, the place throwing the exception is in Java’s own code instead of what you wrote.

Slide 27

Semantics

```
try {
    <statement>
} catch (<class_1 e_1>) {
    <catch_statement_1>
} ...
  catch (<class_n e_n>) {
    <catch_statement_n>
}
```

If an exception is thrown, the first catch block whose declared type is a supertype of the thrown exception will run. Multiple catch blocks can be chained.

Slide 28

Finally

```
try {
     <statement>
} catch (<class e>) {
     <catch_statement>
} finally {
     // clean up here
}
```

- Optional `finally` block
- Can be used without `catch`
- Runs regardless of exception
- Even if a return happened

Slide 29

```
BufferedReader br = null;
try {
     br = Files.newBufferedReader(path);
     // Do stuff with br
} finally {
     if (br != null) {
          br.close();
     }
}
```

Finally blocks are often used to clean up resources when done

Slide 30

Try with Resources

```
try (
  BufferedReader br = Files.newBufferedReader(path)
) {
     // Do stuff with br
}
```

- `br` closed automatically
- Can declare multiple resources
- Resources implement `AutoCloseable`

This is newer syntax that does the same thing as the previous slide. The new syntax is preferable especially if you have multiple resources, as if the close() method on one of them throws an exception, this will still make sure the others are closed properly.

Slide 31

Checked Exceptions

Checked

- Must be explicitly handled or passed on
- Unusual but unpreventable circumstances
- Useful to factor out rare cases
- eg: `IOException`

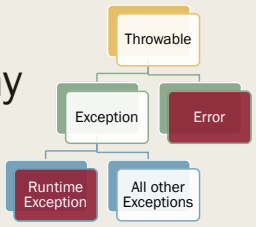
Unchecked

- Need not be handled
- Usually a programmer error (improper array use, call on null…)
- Subclass of Error or RuntimeException
- eg: `NullPointerException`

Exceptions come in two types

Slide 32

Type Hierarchy



Red = Unchecked

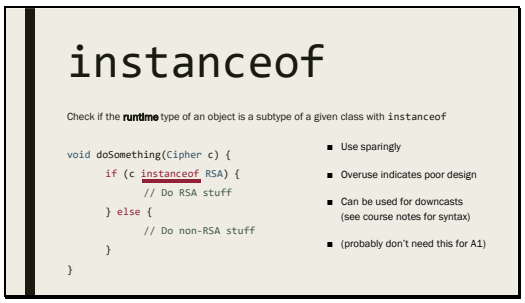
Slide 33

Checked Exception

```
/**
 * Encrypts the plaintext string {@code plaintext} and returns the
 * ciphertext as a {@code String}. Makes sense only for alphabetic ciphers.
 *
 * Throws UnsupportedOperationException if this cipher only works on
 * InputStreams.
 *
 * @param plaintext The plaintext to be encrypted
 * @return An encrypted ciphertext
 */
String encrypt(String plaintext) throws UnsupportedOperationException;
```

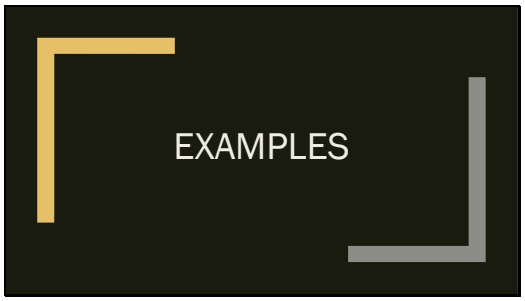
The A1 release code has an example of a checked exception in the Cipher interface. Here, the non alphabetic ciphers will throw an exception if this method is called.

Slide 34

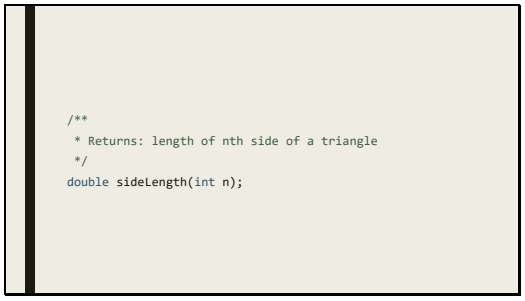


As a side note, Java does provide a way to check the runtime type of an object. Don't overuse this.

Slide 35



Slide 36



What's wrong with this?

Slide 37

```java
/**
 * Returns: length of nth side of a triangle
 * Requires: 0 <= n <= 2
 */
double sideLength(int n);


/**
 * Returns: length of nth side of a triangle
 * @throws OutOfBoundsException if n < 0 or n > 2
 */
double sideLength(int n) throws OutOfBoundsException;
```

The pros and cons of these two variants were discussed in class

Slide 38

Exercise

Download the code from the course website.

The `Rational` class represents a rational number.

- Write the `reciprocal()` method and design a more comprehensive specification for it.
  - *Hint: What type of exception might be appropriate?*
- Consider if you want to modify the constructor's behavior.
- Write a main() method (or JUnit test) to exercise your code.

Slide 39