CSSE2002/7023

Semester 1, 2021

Programming in the Large

Week 3.2: Exceptions, Packages, Access Control

In this Session

- Exceptions
- Throwing Exceptions
- Inheritance and Exceptions
- Pros and Cons of Exceptions
- Packages
- Access Control

Exceptions

Exceptions are what happens in Java when something goes wrong:

```
System.out.println(5 / 0); // Infinity??
// Causes a "java.lang.ArithmeticException: / by zero"
```

We can catch the Exception to handle it:

```
try {
    System.out.println(5 / 0);

} catch (ArithmeticException e) {
    System.out.println(e); // print out error messsage
    // do something to recover
}
```

See `RuntimeExceptionsDemo.java`

Exceptions

- Don't just squash exceptions.
- Once an exception has been thrown, it will unwind the stack until caught.
  - return does not happen
- A `try` can have multiple `catch` blocks.
- `finally` happens whether or not an exception was caught.
- Trigger an exception with `throw`.

```
try {
    System.out.println(5 / 0);
} catch (ArithmeticException e) {
    // handle one type of error
} catch (FileNotFoundException) e) {
    // handle another type of error
} finally {
    // anything in here will always happen
}
```

```
// sometimes we need to show an error occured
throw new IOException();
```

If they aren't caught ...

If Java knows that some types of exceptions *could be thrown*, it insists you do something about them. You must either:

1. `catch` it
2. Declare that the method could throw the exception
   - making it the responsibility of the caller to deal with the exception

```
public int someFunction() throws FileNotFoundException {
        // some code which uses Files
        // and could throw an Exception
}
```

If they aren't caught ...

If Java knows that some types of exceptions *could be thrown*, it insists you do something about them. You must either:

1. `catch` it
2. Declare that the method could throw the exception
   - making it the responsibility of the caller to deal with the exception

Can use `throws` if can't be dealt with locally.

```java
public int someFunction() throws FileNotFoundException {
        // some code which uses Files
        // and could throw an Exception
}
```

If they aren't caught ...

If Java knows that some types of exceptions *could be thrown*, it insists you do something about them. You must either:

1. `catch` it
2. Declare that the method could throw the exception
   - making it the responsibility of the caller to deal with the exception

Can use `throws` if can't be dealt with locally.

Need to `catch` it somewhere.

```java
public int someFunction() throws FileNotFoundException {
        // some code which uses Files
        // and could throw an Exception
}
```

If they aren't caught . . .

If Java knows that some types of exceptions *could be thrown*, it insists you do something about them. You must either:

1. `catch` it
2. Declare that the method could throw the exception
   - making it the responsibility of the caller to deal with the exception

Can use `throws` if can't be dealt with locally.

Need to `catch` it somewhere.

See `ExceptionsDemo.java`

```java
public int someFunction() throws FileNotFoundException {
        // some code which uses Files
        // and could throw an Exception
}
```

Inheritance and Exceptions

Exceptions are objects (and hence described by classes).

Consequently, catching by a parent class exception type will catch any of its children exception objects.

```
try {


} catch (IOException e) {
    // FileNotFoundException,
    // UnknownHostException
    // EOFException, ...
}
```

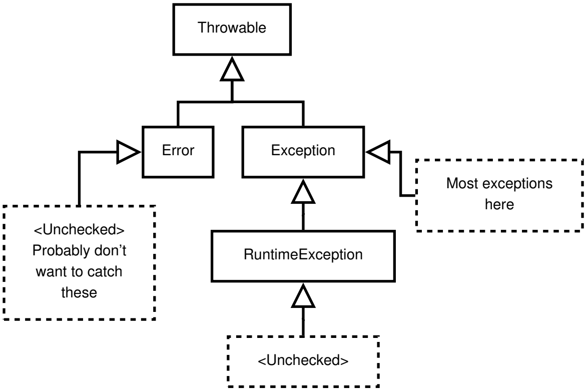
Inheritance and Exceptions

```
try {

} catch (FileNotFoundException e) {
    // This could execute.
} catch (IOException e)  {
    // This could execute.
} catch (EOFException e) {
    // This cannot execute.
    // Already dealt with by
    // superclass type above.
}
```

Be sure to put the most general class *last*.

Exception Heirachy — in `java.lang`



You don't need to declare methods throw things which are subclasses of `RuntimeException` or `Error`.
You could `catch Throwable`.

- Don't! Errors are generally very bad.

What about `catch Exception`? — Need a good reason.

Pros and Cons of Using Exceptions

- Code that detects the problem may not know what it should do about it (move IO to borders of the program).
- Exception propagation means decisions can be made elsewhere (without needing to code a return path all the way back).
- Can carry a lot of information
- Can't be ignored (unless squashed)
- Java likes them
- Did something go wrong (waves vaguely) somewhere in there.
- Not as good if the problem should be checked immediately.
- Less convienient where fine control is needed
- Better for "exceptional" circumstances
- If it can be checked for ahead of time, is it better to do that instead?

Packages

When the number of identifiers[1] increases and code from multiple libraries/authors is combined, the chance of clashing names increases.

- Old solution — really long names
- namespaces / modules / packages — allows duplicate names to exist provided they have separate contexts.

e.g. `java.util.List` and `java.awt.List` can coexist provided their use is not actually ambiguous.

[1]Names for things, e.g. classes, variables, . . .

Packages

You can declare the contents of a file as belonging to a package at the top of the file:

```java
package crawl;

public class Player {...}
```

The directory structure of the project **must** reflect the package naming. e.g. if the project root is `src/`, then `public class Sponge` from the package `noms.sweet` would be stored in:

```
src/
     noms/
           sweet/
                  Sponge.java
```

Packages

Early recommendation was that packages be named for your project domain (e.g. `org.junit`)

Packages and Access Control

- `protected` members are accessible to all methods in any classes in the same package.
  - *and subclasses anywhere*
- *package private/default/blank/...*
  Items with no explicit access specifier can be used by any class in the same package but not by subclasses.

```
private int a = 5; // only accessible in same class

// only accessible in same package, or subclasses
protected int b = 5;

int c = 5; // only accessible in same package

public int d = 5; // accessible from anywhere
```