CS 251
Intermediate Programming
Exceptions

Brooke Chenoweth

University of New Mexico

Spring 2024

Expecting the Unexpected

Most of the time our programs behave well, however sometimes unexpected things happen. Java's way of handling these types of problems is called *exception handling*

When do exceptions occur?

- When you least expect them to...
- When there's something wrong with the hardware, or other things that you can't control from your program.
  - Input from files (or other streams)
  - Communication of various kinds (internet, users)
- Or...When you create them yourself
  - Custom problem space may need custom exceptions

What is an exception?

- Like everything else in java, they are `Objects`
- Objects can be created and customized, extended and inherited
- Many exceptions are already predefined in java
  - `ArrayIndexOutOfBoundsException` is one of them
  - Extends the class `RuntimeException`
  - For more refer to the `Exception` class in the Java API.

Custom exception class

```
public class BadThingHappenedException
  extends Exception {

  public BadThingHappenedException () {
    super("Something bad happened");
  }

  public BadThingHappenedException ( String msg ) {
    super(msg);
  }
}
```

Creating an exception

- Create a new exception object:

```
Exception myEx = new BadThingHappenedException();
```

- Creating an exception, doesn't mean you caused an exceptional event.
- Nothing happens until you "throw" it...

Exception keywords

- `try` – clause for testing potential exception code.
- `catch` – catching the exceptions, if they happen
- `throws` – used in method headers to indicate method might cause exception
- `throw` – used by a method to "throw" (cause) an exception
- `finally` – code executed after the try-catch clauses, regardless of whether exception happened or not.

Catching an Exception

- Try to compile the following:

```
public class Sleeper {
  public void sleep10Secs () {
    Thread.sleep(10000);
  }
}
```

- Will not work. . . Why?

Methods throwing Exceptions

- Methods can define that they wish to be able to throw one or more exceptions

```
int myMethod() throws SomeException
```

```
int myMethod() throws SomeException, SomeOtherException
```

- Note that these exception classes must exist and be defined as the prior `BadThingHappenedException` for the program to compile
- `Thread.sleep()` is defined like this, it throws an `InterruptedException` in case its sleep is disturbed.

Catching exceptions

- Calls to methods that potentially throw exceptions must be "padded" to allow compilation, to allow for the exception to happen
- There are two basic approaches:
  - Ignoring the exceptions, and passing them on to the caller of your method.
  - Catching the exception and dealing with it yourself

Ignoring (passing on) exceptions

- To avoid dealing with the exceptions yourself, while still calling methods that might throw exceptions - your method must also be declared to throw those same exceptions.

```
public class Sleeper {
  public void sleep10Secs()
    throws InterruptedException {
    Thread.sleep(10000);
  }
}
```

- Only viable if caller is prepared to handle exceptions

Catching exceptions

Other solution: catch the exception and handle it yourself

```
public class Sleeper {
  public void sleep10Secs() {
    try {
      Thread.sleep(10000);
    } catch ( InterruptedException ie ) {
      System.out.println ( "Woke up early!" );
    }
  }
}
```

When you do this, make sure you really handle the exception.

Don't eat exceptions!

```
try {
  bigRedButton.pushIt();
} catch (EndOfTheWorldException ex) {
  // Silently ignoring Armageddon...
}
```

At the very least, add some debugging output in case the "impossible" exception happens.

Which one to use?

- Both above methods are allowed
- Only use "passing on" when you are sure that caller can handle exception, or if ok to ignore exceptions
- If you can handle exception within – then do!
- Makes *your* program more robust

Throwing an exception

- If necessary, you can create and `throw` an exception:

```
throw new SomeException("Explanation");
```

- Assumes the `SomeException` class exists, and has a constructor taking a `String` argument
- Aborts the execution of the current method, no return value is provided
- Exception must be handled by the caller of your method
- Your method must be declared as:

```
public int myMethod() throws SomeException
```

Throwing while catching

- Can throw an exception in a catch clause, if you want to create your own Exception messages, or provide an abstraction for the "real" exception:

```
try {
  Thread.sleep(10000);
} catch ( InterruptedException ie ) {
  throw new SomeException("Awakened");
}
```

- The class `SomeException` must exist

`try ...catch ...finally`

- Similar to an if statement
- Can have only one `try` clause, but...
- Any number of `catch` clauses
- Catch clauses should be ordered in decreasing order of specialization, i.e., if `catch (Exception e)` is the first, it will catch **all** exceptions.
- `finally` clause to be used if something must happen, even if exception will be thrown (and method exit)

try/catch/finally example

```
try {
  driver.getInCar();
  driver.driveToWork();
} catch (DeadBatteryException ex) {
  driver.callAAA();
} catch (NoKeysException ex) {
  driver.takeBus();
} finally {
  if(driver.isInCar()) {
    driver.getOutOfCar();
  }
}
```

Checked vs Unchecked

- Most exceptions are *checked* exceptions, which means you must handle them somehow.
- Exception types that extend `RuntimeException` are *unchecked* exceptions, which means you don't have to handle them, but may choose to do so.
  - Usually these are programmer errors, like divide by zero, index out of bounds, referencing null.
  - Generally can avoid through good coding.
  - Still might catch, just in case, but shouldn't be first choice.

A note on usage...

- Many types of problems can be detected and prevented inside your code. When possible this is preferred since exceptions run slower than "normal" code.
- Exception code is executed in special mode, and exiting by normal means is faster in high-performance requirements