Day 8

COMP1006/1406
Summer 2016

M. Jason Hinek
Carleton University
today’s agenda

- assignments
  - Assignment 4 is out and due on Tuesday

- Bugs and Exception handling
Bugs...

often use the word *bug* when there is a problem with our program
Bugs...

0800 Andon started - Andon stopped - Andon stopped
     033) PPO2 2.130476415
     034) CODE 2.13062415
Relays 6-2 in 033 failed special speed test
Relays changed
Relays changed

1100 Started Cosine Tape (Sine check)

1525 Started Multy Adder Test

1545 Started

First actual case of bug being found.

1630 Changed started
1700 closed down.
Bugs...

- **bug** is an error in our code (or hardware)
> **bug** is an error in our code (or hardware)

> **debugging** is a methodical process of finding and reducing the number of bugs
Bugs...

especially 3 types of bugs/errors

· compile errors
  - program does not compile but compiler tells you why
  - syntax errors, type mismatch errors, ...
  - these are your fault

· runtime errors
  - cannot be determined at compile time
  - sometimes require re-design of code
  - these may or may not be your fault

· logical errors
  - non-syntax mistakes in your code
  - Java cannot detect or explain these errors
  - may be very hard to find...
  - these are definitely your fault
Bugs...

essentially 3 types of bugs/errors

- compile errors
- runtime errors
- logical errors
Bugs...

essentially 3 types of bugs/errors

- **compile errors**
  - program does not compile but compiler tells you why
  - syntax errors, type mismatch errors, ...

- **runtime errors**
  
- **logical errors**
  


Bugs...

essentially 3 types of bugs/errors

- **compile errors**
  - program does not compile but compiler tells you why
  - syntax errors, type mismatch errors, ...

- **runtime errors**
  - cannot be determined at compile time
  - sometimes require re-design of code

- **logical errors**
Bugs...

essentially 3 types of bugs/errors

- compile errors
  - program does not compile but compiler tells you why
  - syntax errors, type mismatch errors, ...

- runtime errors
  - cannot be determined at compile time
  - sometimes require re-design of code

- logical errors
  - non-syntax mistakes in your code
  - Java cannot detect or explain these errors
  - may be very hard to find...
Bugs...

essentially 3 types of bugs/errors

- **compile errors**
  - program does not compile but compiler tells you why
  - syntax errors, type mismatch errors, ...
  - these are your fault

- **runtime errors**
  - cannot be determined at compile time
  - sometimes require re-design of code

- **logical errors**
  - non-syntax mistakes in your code
  - Java cannot detect or explain these errors
  - may be very hard to find...
Bugs...

essentially 3 types of bugs/errors

▶ compile errors
  ▶ program does not compile but compiler tells you why
  ▶ syntax errors, type mismatch errors, ...
  ▶ these are your fault

▶ runtime errors
  ▶ cannot be determined at compile time
  ▶ sometimes require re-design of code
  ▶ these may or may not be your fault

▶ logical errors
  ▶ non-syntax mistakes in your code
  ▶ Java cannot detect or explain these errors
  ▶ may be very hard to find...
Bugs...

essentially 3 types of bugs/errors

▶ compile errors
  ▶ program does not compile but compiler tells you why
  ▶ syntax errors, type mismatch errors, ...
  ▶ these are your fault

▶ runtime errors
  ▶ cannot be determined at compile time
  ▶ sometimes require re-design of code
  ▶ these may or may not be your fault

▶ logical errors
  ▶ non-syntax mistakes in your code
  ▶ Java cannot detect or explain these errors
  ▶ may be very hard to find...
  ▶ these are definitely your fault
Bugs...

- first we need to discover that a bug exists
Bugs...

- first we need to discover that a bug exists
  - compile errors are found for free*
Bugs...

- first we need to discover that a bug exists
  - compile errors are found for free*

---

*xkcd - 303*
Bugs...

- first we need to discover that a bug exists
  - compile errors are found for free
  - test, test, test, ...

- JVM might tell you where code crashed (runtime error)
  - observable error may have been caused elsewhere though...

- play computer! backtrack...
  - might be easy or hard

- next we fix the bug
  - if we know exactly where the bugs originates, often easy to fix
  - serious bugs may require rethinking your code/objects
Bugs...

- first we need to discover that a bug exists
  - compile errors are found for free★
  - test, test, test, ...
  - `println` is your friend

- next we need to find in your code where they occur
  - JVM might tell you where code crashed (runtime error)
  - observable error may have been caused elsewhere though...
  - play computer! backtrack...
  - might be easy or hard

- next we fix the bug
  - if we know exactly where the bugs originates, often easy to fix
  - serious bugs may require rethinking your code/objects
Bugs...

- first we need to discover that a bug exists
  - compile errors are found for free
  - test, test, test, ...
  - `println` is your friend

- next we need to find in your code where they occur
Bugs...

- first we need to discover that a bug exists
  - compile errors are found for free
  - test, test, test, ...
  - `println` is your friend

- next we need to find in your code where they occur
  - JVM might tell you where code crashed (runtime error)
Bugs...

- first we need to discover that a bug exists
  - compile errors are found for free
  - test, test, test, ...
  - `println` is your friend

- next we need to find in your code where they occur
  - JVM might tell you where code crashed (runtime error)
  - observable error may have been caused elsewhere though...
Bugs...

- first we need to discover that a bug exists
  - compile errors are found for free
  - test, test, test, ...
  - `println` is your friend

- next we need to find in your code where they occur
  - JVM might tell you where code crashed (runtime error)
  - observable error may have been caused elsewhere though...
  - play computer! backtrack...
Bugs...

- first we need to discover that a bug exists
  - compile errors are found for free
  - test, test, test, ...
  - `println` is your friend

- next we need to find in your code where they occur
  - JVM might tell you where code crashed (runtime error)
  - observable error may have been caused elsewhere though...
  - play computer! backtrack...
  - might be easy or hard
Bugs...

- first we need to discover that a bug exists
  - compile errors are found for free*
  - test, test, test, ...
  - `println` is your friend

- next we need to find in your code where they occur
  - JVM might tell you where code crashed (runtime error)
  - observable error may have been caused elsewhere though...
  - play computer! backtrack...
  - might be easy or hard

- next we fix the bug
Bugs...

- first we need to discover that a bug exists
  - compile errors are found for free
  - test, test, test, ...
  - `println` is your friend

- next we need to find in your code where they occur
  - JVM might tell you where code crashed (runtime error)
  - observable error may have been caused elsewhere though...
  - play computer! backtrack...
  - might be easy or hard

- next we fix the bug
  - if we know exactly where the bugs originates, often easy to fix
Bugs...

- first we need to discover that a bug exists
  - compile errors are found for free
  - test, test, test, ...
  - `println` is your friend

- next we need to find in your code where they occur
  - JVM might tell you where code crashed (runtime error)
  - observable error may have been caused elsewhere though...
  - play computer! backtrack...
  - might be easy or hard

- next we fix the bug
  - if we know exactly where the bugs originates, often easy to fix
  - serious bugs may require rethinking your code/objects
Bugs...

part of the problem is that we don’t live in a perfect world...

- people do not follow API specifications (preconditions)
- people do weird things with your code...
- files get corrupted

you have no control over how people use your code
you have no control over the universe your code is running in

- need to write robust code

robustness is the ability of the code to cope with errors during execution and cope with erroneous input (from wiki)
Bugs...

we could write code to check for all possibilities in our code

- **error-checking** is code added to your code to look for bad data
- **error-handling** is what you do when bad data is found
  
  → everything needs to be an **Object**

in Java we use **Exceptions**...

- an **Exception** is an error that occurs in your code
- **Exception Handling** is what you do when an exception is found

goal of using exceptions

- you want your program to die gracefully
- you want to program to recover from bad data if possible
Exceptions...

in Java, exceptions are objects

- the JVM automatically does this, or
- you explicitly do this in your code, or
- another method will explicitly do this

in Java, thrown exceptions are always caught

- explicitly caught by your code, or
- delegated to someone else to be caught, or
- caught by the JVM if everyone delegates
in Java, exceptions are objects

- the JVM automatically does this, or
- you explicitly do this in your code, or
- another method will explicitly do this

in Java, thrown exceptions are always caught

- explicitly caught by your code, or (graceful)
- delegated to someone else to be caught, or (potentially graceful)
- caught by the JVM if everyone delegates (ugly)
Exceptions...

in Java, exceptions are objects

Error Class

- unrecoverable errors
  - `java.lang.StackOverflowError`
  - `java.lang.OutOfMemoryError`

- you do not generally catch these yourself
  (you fix your code so it doesn’t happen again!)
Exceptions...

In Java, exceptions are objects

Exception Class

- less severe errors (we might be able to recover from these)
- there are checked and unchecked exceptions
Exceptions...

in Java, exceptions are **objects**

**Exception Class**

- less severe errors (we might be able to recover from these)
- there are **checked** and **unchecked** exceptions

**checked** exceptions (**Exception Class**)

- compiler checks that these are explicitly caught (if there is a throw there must be a catch)
- **IllegalAccess.Exception** (try to violate access modifier)
- **FileNotFoundException** (file is not found)
Exceptions...

in Java, exceptions are **objects**

**Exception Class**

- less severe errors (we might be able to recover from these)
- there are **checked** and **unchecked** exceptions

**checked** exceptions (**Exception Class**)

- compiler checks that these are explicitly caught
  (if there is a throw there must be a catch)
- **IllegalAccessException** (try to violate access modifier)
- **FileNotFoundException** (file is not found)

**unchecked** exceptions (**RuntimeException** subclass)

- compiler does not check for a catch
  (typically left for JVM to crash program)
- you should rethink your code to avoid these...
- **ArithmeticException** (divide by zero for example)
- **NullPointerException**
- **IndexOutOfBoundsException**
Exceptions...
let’s take a break...
for 1.2 minutes
Exceptions...

new Java keywords

- **throws**
  - used in method declaration
  - says that this method is delegating this exception

- **try/catch/finally**
  - used to execute code and handle exceptions thrown
  - "try" to execute this code...
  - "catch" any thrown exceptions and handle them
  - "finally" execute some code after everything is done

- **throw**
  - explicitly throw an exception
void openFile(String fname) throws java.io.FileNotFoundException{
    ...
}

tells the compiler that this method might
Throws

void openFile(String fname) throws java.io.FileNotFoundException{
...
}

tells the compiler that this method might
  ▶ explicitly throw a java.io.FileNotFoundException object
void openFile(String fname) throws java.io.FileNotFoundException {
  ...
}

tells the compiler that this method might
  ▶ explicitly throw a java.io.FileNotFoundException object
  or
  calls a method that might
Throws

`void openFile(String fname) throws java.io.FileNotFoundException{ ... }
`
throws java.io.FileNotFoundException{
...}

tells the compiler that this method might

- explicitly throw a java.io.FileNotFoundException object
  or
  calls a method that might
  
  - explicitly throw a java.io.FileNotFoundException object
    or
    calls a method that might
Throws

```java
void openFile(String fname) throws java.io.FileNotFoundException{
...
}
```

tells the compiler that this method might

- explicitly throw a `java.io.FileNotFoundException` object
- or
- calls a method that might
  - explicitly throw a `java.io.FileNotFoundException` object
  - or
  - calls a method that might
    - explicitly throw a `java.io.FileNotFoundException` object
Throws

void openFile(String fname) throws java.io.FileNotFoundException{
...
}

tells the compiler that this method might
  ▶ explicitly throw a java.io.FileNotFoundException object
  or
calls a method that might
  ▶ explicitly throw a java.io.FileNotFoundException object
  or
calls a method that might
  ▶ explicitly throw a java.io.FileNotFoundException object
  or
calls a method that might
void openFile(String fname) throws java.io.FileNotFoundException {
...
}

tells the compiler that this method might:
\[\begin{itemize}
\item explicitly throw a java.io.FileNotFoundException object
\item or calls a method that might:
\[\begin{itemize}
\item explicitly throw a java.io.FileNotFoundException object
\item or calls a method that might:
\[\begin{itemize}
\item explicitly throw a java.io.FileNotFoundException object
\item or calls a method that might:
\[\begin{itemize}
\item explicitly throw a java.io.FileNotFoundException object
\end{itemize}
\end{itemize}
\end{itemize}
\end{itemize}
...
void openFile(String fname) throws java.io.FileNotFoundException{
    ...
}

let's look at an example
Try/Catch

- use `throws` to delegate exception handling
- use `try` and `catch` to handle the exception yourself
Try/Catch

- use `throws` to delegate exception handling
- use `try` and `catch` to handle the exception yourself

```java
try{
    block of code to try
}
catch(Exception e){
    block of code
to execute if exception is caught
}
```
Try/Catch

- use `throws` to delegate exception handling
- use `try` and `catch` to handle the exception yourself

```java
try{
    block of code to try
}
catch(Exception e){
    block of code
to execute if exception is caught
}
catch(RuntimeException re){
    block of code
to execute if exception is caught
}
```

Note: this won’t compile
Try/Catch

- use `throws` to delegate exception handling
- use `try` and `catch` to handle the exception yourself

let’s look at the same example...
Try/Catch

order matters! class hierarchy matters!

```
java.lang.Object
↑
java.lang.Throwable
↑
java.lang.Exception
↑
java.io.IOException
↑
java.io.FileNotFoundException
```

19
Try/Catch

order matters! class hierarchy matters!

```
Try/Catch

Object
↑
Throwable
↑
Exception
↑
java.io.IOException
↑
java.io.FileNotFoundException
```

19
Try/Catch

```java
try{
    block of code to try
}
catch(IOException ioe){
    block of code to execute if exception thrown
}
catch(RuntimeException re){
    block of code to execute if exception thrown
}
catch(Exception e){
    block of code to execute if exception thrown
}
```
Try/Catch

order matters! class hierarchy matters!

```
Object
  ↑
Throwable
  ↑
Exception
  ↑
java.io.IOException
  ↑
java.io.FileNotFoundException
```
Try/Catch/Finaly

- **try** executes a block of code
- if exception thrown, **catch** it and handle it
- after try and possibly catch code executes, we **finally** execute some finishing code
Try/Catch/Finaly

- **try** executes a block of code
- if exception thrown, **catch** it and handle it
- after try and possibly catch code executes, we **finally** execute some finishing code

```java
try{
    block of code to try
}
catch( Throwable e ){
    block of code
    to execute if exception is caught
}
```
Try/Catch/Finaly

- **try** executes a block of code
- if exception thrown, **catch** it and handle it
- after try and possibly catch code executes, we **finally** execute some finishing code

```java
try{
    block of code to try
}
catch(Throwables e){
    block of code
to execute if exception is caught
}
finally{
    block of code to execute // like a funny else
    REGARDLESS of what happens above
}
```
Throw

`throw` is the mechanism to explicitly throw an exception.

- `throw new Exception("something bad here...");` creates a new `Exception` object and throws it.
- `throw e;` throws an existing exception (`e` must be a `Throwable` object).
Throw

throw is the mechanism to explicitly throw an exception.

- throw new Exception("something bad here..."); creates a new Exception object and throws it
Throw

`throw` is the mechanism to explicitly throw an exception.

- `throw new Exception("something bad here...");` creates a new Exception object and throws it
- `throw e;` throws an existing exception (e must be a Throwable object)
Making Your Own Exceptions

you create your own custom exceptions by extending an appropriate Throwable class
Making Your Own Exceptions

you create your own custom exceptions by extending an appropriate Throwable class
Making Your Own Exceptions

you create your own custom exceptions by extending an appropriate Throwable class

```java
public class BadCardRankException extends Exception {

    public BadCardRankException() {
        super("Rank of card is invalid");
    }
}
```
Throwable

the Throwable class is the root of all exceptions in Java

- `toString()`
  - returns short description of this object

- `getMessage()`
  - returns the message as input with the constructor

- `printStackTrace()`
  - prints the stack trace to standard error