



Programming and Software Development
COMP90041

Lecture 11

Exceptions



- Error handling is difficult because the code that knows what to do when a problem happens is different from the code that detects the problem
- Two traditional ways to handle errors:
 - ① return a result indicating whether operation succeeded
 - ② Print an error message and exit
- Approach 1 is terrible;
 - ▶ **Very** common for programmers to fail to check return code, and just assume everything is OK
 - ▶ A method can only return one result; if it already needs to return one, how to also return the success code?
- Approach 2 is worse;
 - ▶ No way to recover from the program exiting!
 - ▶ This means every call to a method that could experience a problem must first test if there will be a problem



- Exceptions give the best of both worlds:
- Write code mostly without worrying about what to do if something goes wrong
- In places where you know what to do if something goes wrong, you can still catch it and handle it
- If you don't handle an exception at all, it aborts the program
- Two sides to exception handling:
 - ▶ Throwing
 - ▶ Catching



- An exception is an object indicating what went wrong
- The class of the exception object indicates broadly what's wrong
- Extra information in the form of a message string gives details
- Some methods of all exception classes:
 - ▶ `toString()` returns a string describing the exception
 - ▶ `getMessage()` returns a string with detail about the error, or `null`
 - ▶ `printStackTrace()` print a backtrace of what was happening when the exception happened (only useful to programmers)



- Throwing an exception signals an error has occurred
- Currently executing code is interrupted and abandoned
- If exception is not caught in the current method, it will flow up to the calling method
- If caller doesn't catch it, it percolates up until it is caught
- If it's never caught, the program is aborted, printing a backtrace
- Use exceptions only for exceptional circumstances, not as a quick way to jump around your code



- Form: `throw exceptionObject`
- Typically create the exception at the same time:
`throw new ExceptionClass(detailString);`
- Or: `throw new ExceptionClass();`
- For example:

```
public Person(int age, String name) {  
    if (name == null) {  
        throw new NullPointerException(  
            "Creating person with null name");  
    }  
    this.age = age;  
    this.name = name;  
}
```



- Handling an exception means providing code to execute to recover from code going wrong
- Form:

```
try {  
    code that may go wrong...  
} catch (ExceptionClass var) {  
    code to handle exception...  
}
```

- **try** part specifies code that may throw an exception
- **catch** parameter specifies the kind of exception to catch
- Code inside the **catch** clause specifies what to do if that exception occurs



- Use `try...catch` like an ordinary statement

```
try {  
    code that may go wrong...  
} catch (ExceptionClass var) {  
    code to handle exception...  
}
```

- First execute *code that may go wrong...*
- If it completes without exception, ignore (skip over) *code to handle exception*
- If it throws an exception matching *ExceptionClass*, bind *var* to exception object and execute *code to handle exception*
- Either way, then go on to following statements



- **catch** parameter has two roles:
 - ▶ To specify the kind of exception to catch
 - ▶ To name a variable to hold the exception object
(`e` is often used as variable name)
- **catch** parameter is scoped to (only defined in) that exception handler
- Can have multiple **catches** in immediate succession
 - ▶ Only one handler is executed, others are ignored
 - ▶ First one that matches the thrown exception is used
- If no **catch** clause exception class matches thrown exception, it is not caught by that **try...catch**
 - ▶ Propagates to caller as if code were not in a **try...catch**



THE UNIVERSITY OF
MELBOURNE

QuickPoll

MELBOURNE

qp.unimelb.edu.au/tchristy



MELBOURNE

What will this code print?

```
try {  
    int i = 1;  
    if (i > 0) throw new Exception();  
    System.out.print("X");  
} catch (Exception e) {  
    System.out.print("Y");  
}  
System.out.println("Z");
```

- A X
- B XZ
- C Y
- D YZ
- E XYZ



MELBOURNE

What will this code print?

```
try {  
    int i = 1;  
    if (i > 0) throw new Exception();  
    System.out.print("X");  
} catch (Exception e) {  
    System.out.print("Y");  
}  
System.out.println("Z");
```

- A X
- B XZ
- C Y
- D YZ
- E XYZ



- Always put more specific (descendant class) **catches** before more general (ancestor class) ones
- More general one will always match when the more specific one would, so more specific one will never be used

```
try {  
    ...  
} catch (Exception e) {  
    // This code will be used for any  
    // descendent of Exception class  
} catch (NumberFormatException e) {  
    // Never used because NumberFormatException is a  
    // descendent of Exception  
}
```



- `catch` block should try to recover from error
- Can use exception object (commonly parameter `e`)
- `e.getMessage()` returns exception message
- Can use known class of exception to decide action
- If `catch` block cannot resolve the problem, it can always throw the same exception (just `throw e;`)
- Or `catch` block can throw a different exception
- Exception thrown inside `catch` block will not be caught by that `try...catch`
 - ▶ Only exceptions thrown inside the `try` block are caught by that `try...catch`



- A `finally` block allows you to perform clean ups regardless of whether or not an exception is thrown
- `finally` block is attached to a `try...catch`, following all the `catches`
- Form:

```
try {  
    ...  
} catch (...) {  
    :  
} finally {  
    code to execute regardless  
}
```

- Can omit the `catches`, if they're not needed



When `finally` Block is Executed

- `finally` block is executed almost no matter what
- If `try` block completes without error, `catches` are skipped and `finally` block is executed
- If exception with matching `catch` is thrown in `try` block, `catch` is executed, then `finally` block
- If no matching `catch`, `finally` block executes, then exception propagates to caller
- If matching `catch` block throws exception, `finally` block executes, then exception propagates to caller
- Only if `try` or `catch` is an infinite loop, or calls `System.exit`, will `finally` block be missed
 - ▶ So don't do that!



- Java distinguishes a few different kinds of exceptions:
- Everything that can be `thrown` is a descendent of the `Throwable` class
- Subclasses of `Throwable`:
 - ▶ `Error`: a serious error that could occur at any time, such as JVM failure
 - ▶ `Exception`: exceptional circumstance in executing your code
- Subclasses of `Exception`:
 - ▶ `RuntimeException`: an exception that could happen at any time in normal Java code (e.g. null pointer)
 - ▶ ... and many, many more specific exception classes



- You should not try to catch descendants of `Error`; there is not much you can do about such failures
- Descendants of `RuntimeException` indicate an error that your code has failed to prevent
- Rather than catch `RuntimeExceptions`, you should fix your code to avoid them, e.g.:
 - ▶ `NullPointerException`: check that variables are not null before sending them a message
 - ▶ `ArrayIndexOutOfBoundsException`: ensure that index is within bounds before accessing array element
 - ▶ *Etc.*
- (But it's fine to throw `RuntimeExceptions`)
- Other descendants of `Exception` should be caught and handled where you know how to handle them



THE UNIVERSITY OF
MELBOURNE

QuickPoll

MELBOURNE

qp.unimelb.edu.au/tchristy



MELBOURNE

What will this code print?

```
try {  
    String s = null;  
    System.out.print(s.toUpperCase());  
    System.out.print("W");  
} catch (Exception e) { System.out.print("X");}  
} finally { System.out.print("Y");}  
}  
System.out.println("Z");
```

- A WXYZ
- B XYZ
- C XZ
- D YZ
- E a NullPointerException error message



MELBOURNE

What will this code print?

```
try {  
    String s = null;  
    System.out.print(s.toUpperCase());  
    System.out.print("W");  
} catch (Exception e) { System.out.print("X");}  
} finally { System.out.print("Y");  
}  
System.out.println("Z");
```

- A WXYZ
- B XYZ
- C XZ
- D YZ
- E a NullPointerException error message



- Descendents of `Error` and `RuntimeException` are called unchecked exceptions
- Other descendents of `Exception` are called checked exceptions
- Each method must declare any checked exceptions that may be propagated out
- “Catch or declare” rule: each method must declare that it may throw:
 - ▶ Any checked exceptions that may be thrown by the method, and
 - ▶ Any checked exceptions declared to be thrown by methods it calls
 - ▶ Except those that are caught and handled by the method body



- Declare what checked exceptions a method can throw with a throws clause
- Form: *throws ExceptionClass*
- Place this after method signature in concrete or abstract method declaration
- Declare multiple throws by listing all of them separated by commas
- *E.g.,*

```
public LostPerson readLostPersonFile(String fileName)
    throws FileNotFoundException, EOFException { ... }
```



- A method may not expand overridden method's **throws** clause (but may reduce it)
- Method may throw any descendent of exceptions listed in **throws** clause
- So **throws** clause can generalise multiple exception classes to a common ancestor
- *E.g.*, **IOException** is the base class of both **FileNotFoundException** and **EOFException**, so:

```
public LostPerson readLostPersonFile(String fileName)
    throws IOException { ... }
```

- But this obliges callers to catch or declare all kinds of **IOExceptions**



- Can define your own exception classes
- Must be descendent of `Exception` class
- Pick the right place in the exception taxonomy
 - ▶ E.g., if it's an I/O related error, make it a descendent of `IOException`
- Make sure `getMessage` works right
- Usually you should define a constructor with no arguments and one with a single `String` argument
- Many user exception classes just have constructors:

```
public MyException(String msg) { super(msg); }
public MyException() {
    super("default description string");
}
```



- *Cheops Law: Nothing ever gets built on schedule or within budget.* — Robert Heinlein
- Goes doubly for software projects
- Statistics vary depending on how you define success/failure, but roughly:
 - ▶ 1 in 6 software development projects fail outright
 - ▶ 1 in 2 projects complete over budget
 - ▶ 1 in 3 projects succeed
- Costs 10s of \$Billions every year
- Reasons for failure include:
 - ▶ Unrealistic or unarticulated goals
 - ▶ Inaccurate estimation of needed resources
 - ▶ Badly defined system requirements
 - ▶ Poor project management
 - ▶ Inability to manage software complexity



- Project started in 2005 to replace the aging LEAP police database Budgeted \$50 Million
- In 2009, budget blew out to \$130M
- In 2011, new business case saw budget blow out to almost \$200M
- Project halted because it didn't handle changing policing requirements
- They're still using LEAP



- Project started in 2005 to replace the aging LEAP police database Budgeted \$50 Million
- In 2009, budget blew out to \$130M
- In 2011, new business case saw budget blow out to almost \$200M
- Project halted because it didn't handle changing policing requirements
- They're still using LEAP

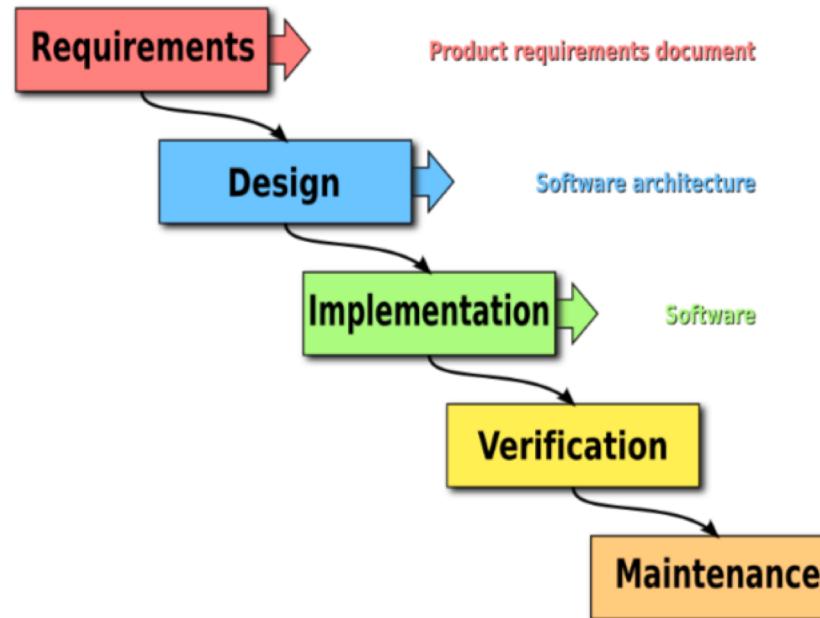




- Software Engineering evolved beginning in the 1960s to solve the “Software Crisis”
- Software development practices and processes are still evolving
- Basic idea: consider phases of software development (software development life cycle):
 - ▶ **Requirements specification:** determine what the system must do
 - ▶ **Design:** determine how to make the system do that
 - ▶ **Implementation:** code the system
 - ▶ **Validation:** make sure the system actually does that
 - ▶ **Maintenance:** keep the system doing what it must do as needs change or problems are discovered



- Classic life cycle model is the waterfall model
- Phases of development are followed in order



- Phases can backtrack to earlier phase if (when) necessary



- The waterfall model often breaks down because:
 - ▶ Client rarely know what they need or want
 - ▶ Clients and developers rarely communicate visions well
 - ▶ Needs change as software is developed
 - ▶ Clients get impatient
- Agile development is more popular now
- Agile software development is an incremental software development methodology
- Continuous delivery — delivery every 1–4 weeks
- Focus on communication — daily short meetings
- Focus on client satisfaction and adaptation to changing client needs



- `throw new ExceptionClass(...)` to signal error
- `try { ... }`
`catch (ExceptionClass e) { ... }`
to catch and handle exception
- Use `finally { ... }` to give cleanup code that will almost always be executed
- Must declare uncaught checked exceptions with `throws ExceptionClass` in method declaration
- Unchecked exceptions shouldn't be caught
 - ▶ Fix bugs that lead to `RuntimeExceptions`
- Software engineering focuses on best processes to develop working software on time and within budget



THE UNIVERSITY OF

MELBOURNE