W07:L01: Dealing with errors

- Our code could encounter many types of errors
  - User input: enter invalid filenames or URLs
  - Resource limitations: disk full
  - Code errors: invalid array index, key not present in hash map, refer to a variable that is null, divide by zero, ...
- If we could anticipate what is going to happen, we would rather signal the error than let the program crash
- Exception handling: gracefully recover from errors that occur when running code

W07:L01: Classification of errors

- All exceptions descend from class `Throwable`
  - `Error` : relatively rare, "not the programmer's fault"
    - Internal errors, resource limitations within Java runtime
    - No realistic corrective action possible, notify caller and terminate gracefully
  - `Exception` : Gracefully recover from anticipated issues
    - Checked exceptions: `extends Exception`
      - Typically user-defined, code assumptions violated. Example: in a list of orders, quantities should be positive integers
    - Unchecked Exceptions: `extends RuntimeException`
      - Programming errors that should have been caught by code. Eg: Array index out of bounds, invalid hash key, ...

W07:L02: Exceptions: Catching and handling

- `try-catch` block

```
try{
    //Error-prone code
}
catch(ExceptionType 1){
    //Code to handle Exception 1
}
catch(ExceptionType 2){
    //Code to handle Exception 1
}
....
catch(ExceptionType k){
    //Code to handle Exception 1
}
```

W07:L02: Exceptions: Catching and handling

- Enclose code that may generate exception in a `try` block
- Code to handle the exception to be added in the `catch` block
- If `try` encounters an exception, rest of the code in the `try` block is skipped
- If exception matches the type in any of the `catch` blocks, the handler code executes
- Otherwise, uncaught exception is passed back to the caller code
- Possible to catch more than one type of exception with multiple `catch` blocks
- `catch (ExceptionType e)` catches any subtype of `ExceptionType`
- `catch` blocks are tried in sequence, match exception type against each one in turn
- Order `catch` blocks by argument type, in the order more specific to less specific

W07:L02: Throwing exceptions

- A method can throw one among many types of exceptions, and it declares all the exceptions that it is likely to throw

```
public void myMethod throws IOException, ArithmeticException(){}
```

- Throws the exception using the throw statement that requires a single argument: a throwable object.

```
throw new EmptyStackException();
```

- Can throw any subtype of the declared exception type
- While calling such a method that throws an exception, the caller code must handle it
- . . . or pass it on such that the caller method also advertises that it throws the same exception
- Customized exceptions - Define a new class extending exception to create a checked exception

W07:L02: Cleaning up after exceptions

- Cleaning up resources happen in the `finally` block
  - When exception occurs, rest of the `try` block is skipped
  - May need to do some clean up (close files, deallocate resources, etc.)
  - `finally` block is always executed except when there is a system failure

```
try{
    //Error-prone code
}
catch(Exception e1){
    //Code to handle the exception
}
finally{
    //Cleanup code
}
```

W07:L03: Packages

- Java has an organizational unit called package
- Can use import to use packages directly

```
import java.util.*;
```

- If we omit modifiers, the default visibility is public within the package
  - This applies to both methods and variables
- Can also restrict visibility with respect to inheritance hierarchy
  - protected means visible within all subclasses

W07:L04: Assertions

- Assertion checks are supposed to flag fatal, unrecoverable errors
- This should not be caught – Abort and print diagnostic information (stack trace)
- If assertion fails, code throws `AssertionError`

```
public static double myfn(double x){
    assert x >= 0;
}
```

- Can provide additional information to be printed with diagnostic message

```
public static double myfn(double x){
    assert x >= 0 : x;
}
```

- If you need to flag the error and take corrective action, use exceptions instead
- Turned on only during development and testing
  - Not checked at run time after deployment

W07:L04: Assertions (Cont.)

- Assertions are enabled or disabled at runtime – does not require recompilation
- Use the following flag to run with assertions enabled
  ```
  java -enableassertions MyCode
  ```
- Can use `-ea` as abbreviation for `-enableassertions`
- Can selectively turn on assertions for a class
  ```
  java -ea:Myclass MyCode
  ```
- ... or a package
  ```
  java -ea:in.ac.iitm.onlinedegree MyCode
  ```
- Similarly, disable assertions globally or selectively
  ```
  java -disableassertions MyCode
  java -da:MyClass MyCode
  ```
- Can combine the two
  ```
  java -ea in.ac.iitm.onlinedegree -da:MyClass MyCode
  ```
- Separate switch to enable assertions for system classes
  ```
  java -enablesystemassertions MyCode
  java -esa MyCode
  ```

W07:L05: Logging

- Logging gives us more flexibility and control over tracking diagnostic messages than simple print statements
- Example: call `info()` method of global logger:

```
Logger.getGlobal().info("Edit->Copy menu item selected");
```

- Can define a hierarchy of loggers
- Seven levels of messages — `SEVERE`, `WARNING`, `INFO`, `CONFIG`, `FINE`, `FINER`, `FINEST`
  - By default, first three levels are logged
- Can set a different level

```
logger.setLevel(Level.FINE);
```

- Turn on all levels, or turn off all logging

```
logger.setLevel(Level.ALL);
logger.setLevel(Level.OFF);
```

- Control logging from within code or through external configuration file