Programming Design and Implementation

Lecture 6: Model Classes

Updated: 7th April, 2020

Mark Upston
Discipline of Computing
School of Electrical Engineering, Computing and Mathematical Sciences (EECMS)

Copyright © 2020, Curtin University
CRICOS Provide Code: 00301J

Outline

"Zero" Marks

- A student who does any of the following in a submitted, assessable answer will receive heavy penalties, up to and including **zero** marks for that question:
  - Uses **`continue`**
  - Uses **`break`** in any other place than a **`switch`** statement
  - Uses **`goto`**
  - Has more than one **`return`** statement in a method
  - Has a **`return`** statement in a method anywhere but the last statement of the method
  - Uses **`System.exit()`** anywhere but the last statement of the **`main()`** method
  - Uses global variables for anything other than class fields
  - Uses a ternary operator
- Note: similar efforts in pseudo code will also receive zero marks

Introduction to Objects

- What is the purpose of an Object?
- What are the properties of Classes and Objects?
- What is a Class?
- What is an Object?

What is the Purpose of an Object?

- The goals are achieved via:
  - Encapsulation
  - Data Hiding
  - Generic Code
  - Method Overloading
  - Polymorphism (Covered in DSA `COMP1002`)
  - Association (Covered in DSA `COMP1002`)
  - Inheritance (Covered in DSA `COMP1002`)

Clumsy Code Example

- Program for printing some Engine information

```
int engine1NumCylinders = 8;
double engine1HorsePower = 450.0;
String engine1FuelType = "98RON";

...

System.out.println("Engine has " + engine1NumCylinders + "cyls");
System.out.println("Engine has " + engine1HorsePower + "HP");
System.out.println("Engine uses " + engine1FuelType + "fuel");
```

- Imagine if we had lots more values **steel** or **alloyHead**, **overheadCam**, **fuelInjected** or **carb**, etc.
- Even worse still, multiple Engines

Objects Solution to Clumsy Code

```
Engine eng1 = new Engine(8, 450.0, "98RON");

System.out.println(eng1);
```

- We have bundled all of the engine information into a container (our own composite data type)
  - i.e., Encapsulation
- That container can now be passed to other submodules easily, and the information used as required

Class

- A class specifies the state and behaviour that an Object can have
  - **State** - What the Object is
    - Classfields
    - Member fields
    - Class members
    - Class attributes
  - **Behaviour** - What the Object does
    - Methods
    - Submodules
    - Functions

Encapsulation

- A (an object of a) class makes use of the information hiding principle
  - Communication with the rest of the software system is clearly defined
  - Its obligations to the software system are clearly defined
  - All details of implementation should be inaccessible to the parts of the software system outside the class

Classes and Objects

- A class specifies:
  - The communication with the rest of the software system
  - The exact data representation required
  - Exactly how the required functionality is to be achieved
- An object is an instance of a class
- In other words the class definition provides the template for the object and the object provides the details for a particular instance
  - This type of a class is called a **model** class
    - i.e., A model of a "real world thing"
- A class can also be used as a collection of related constants and methods which are available to users of the class
  - This is called a **static** class
  - This type of class will never have an object instance
  - An example of this type of class is the Java `Math` class

What Makes Up A Class?

- Each class which will be used to create objects will contain:
- A set of publicly accessible sub modules
- A set of private (or hidden) sub modules
- A set of variables, known as class fields, which:
  - Are hidden from the outside world
  - Are globally accessible anywhere in the class
  - Will contain the information required for objects of the class to perform the desired role
- The public sub modules will, for the most part, be concerned with initialising, accessing or modifying the class fields
- Every class is designed with a specific role in mind
  - In other words the total set of functional requirements for a software system is broken down into a set of tasks, collections of tasks are grouped together and mapped to roles and roles are mapped to specific classes

Class Responsibility

- Each class has a designated role (responsibility) which it must fulfil when the rest of the software system demands it
- The class will be composed of:
  - The data required to perform the role:
    - These are known as the **classfields**
  - The methods required to perform the role
    - The methods that the rest of the software system will invoke are declared as **`public`**:
    - They can be seen and invoked from outside the class
    - The methods that the public methods call when executing their algorithms will be declared as **`private`**:
    - They should not be accessible from outside the class

Class Responsibility (2)

- Take the requirements for a software application and:
  - Identify the class required
  - Assign specific Responsibilities to each class
  - Determine relationships between classes (see DSA `COMP1002`)
  - Repeat the above steps until the design is correct
  - Each responsibility should be handled by that class and no other
    - Example: If a responsibility for keeping track of a person's name is assigned to a class called **`Person`** then:
    - No other class should have this information (except as an object of **`Person`**)
    - Other classes which need this information should refer to this class when the information is required

Classes vs. Objects

- A class describes what is to be placed in an object
- An object is a thing which is created from the class specifications and placed in memory
- A class has no state information, an object does (in the form of the values stored in the class fields)
  - e.g. A class for **`Date`** will specify class fields for **`day`**, **`month`** and **`year`** but each date object will have their own specific values for **`day`**, **`month`** and **`year`**
- An analogy would be:
  - A set of construction plans for a House are the equivalent of a class
    - From these plans as many houses as required can be constructed
    - Each house will conform to the plans in exactly the same way
    - Each house will be built in a different location
    - Each house is equivalent to an object

Comparing Classes to Non-Object Algorithms

- In the past, we designed an algorithm by starting with a main module and using step wise refinement to determine the processing steps and the data types/structures made to fit these steps
- Some of these steps get refined into sub modules and the process repeats until the design is refined and tested well enough to code
- Under Object Orientation this all changes, before the algorithm is designed:
  - The classes are identified
  - Each class is assigned role(s) or responsibilities
  - The required sub modules are designed (i.e., Constructors, accessors, etc)
  - Each Class is thoroughly tested via a test harness
- Finally, the main algorithm (and any required submodules) making use of the previously designed classes in the process

Nouns and Verbs

- Like algorithm design, the determination of what classes should be used is still, by and large, an art form
- One shallow technique is the nouns and verb approach:
  - Nouns are mapped to classes
  - Verbs are mapped to sub modules within classes
  - Result is that:
    - sub module names should always describe an action (i.e., **`getName()`**)
    - Class names should always describe a thing (e.g., **`Person`**)
- It is important to note that the set of classes proposed will change over the time the software is being designed
- This is exactly the same principle as the steps in algorithms changing as the algorithm is being refined

Object Communication

- Sometimes referred to as message passing:
- When an object of one class calls an object of another class it is passing a message (i.e., A request to the object to perform some task)
- The public methods must provide the functionality required for the class to fulfill its role
- There are five categories of methods in a class:
  - The Constructors
  - The Accessor Methods (aka Interrogative Methods)
  - The Mutator Methods (aka Informative Methods)
  - Doing Methods (aka Imperative Methods)
  - Private methods

Java and Class Files

- The Java compiler creates a **`.class`** file for each class defined in the Java code (now the **`.class`** extension makes sense)
- Hence, each class should be entered into its own **`.java`** file whose name is the same as the class
- Many classes that you design and implement in Java could be useful across a number of different applications
- These general purpose classes should be grouped together in a library which is accessible to all of your applications
- In Java we call such a library a **package**
  - You have been using `import java.util.*;` for weeks

Classes in Java

- Each class in its own `.java` file
- Put the main method in a class by itself, the name for that class will be the name of the application
- All variables are **`private`**
- Methods are categorised as public or private and declared as such
- Order your Java code consistently. For PDI we will declare the components of each class in the following order:
  - Declarations for class constants.
  - Declarations for classfields
    - variables which are global to the class
  - Declarations for the Constructors
  - Accessor methods
  - Mutator methods
  - Doing methods (**`public`**)
  - Internal methods (**`private`**)

Classfields

- A class field is a variable which is accessible to all methods in the class (i.e., It is global to the class)
- Good class design means that class fields should not be visible outside the object
  - In Java, this means declaring class fields as **`private`**
  - Information hiding principle
- The class fields will contain the information required so that objects can fulfil their purpose
  - e.g. A class for administering an engine will have classfields to represent number of cylinders, horsepower and fuel type
- Constructors will initialise class fields, accessors will refer to the values stored in class fields and mutators will modify class fields

Object State

- When an object of a class is created (i.e., constructed):
- Memory is allocated to variables corresponding to the class fields specified in the class
- The values contained in these variables is the object **state**
- For example, a class whose role is to maintain information about engines:
    - The class field definitions would describe the variables required to describe an engine
    - Each engine object created would have its own copy of these variables
    - The values of the variables are usually different for each object
        - One class means one definition of what the variables should be
        - As many objects as are required can be made from that one class, each object having its own copy of the class field variables
        - State is only the same if it is a copy

Example

- Object **engine1**
  - Number of cylinders: 8
  - Horsepower: 450.0
  - Fuel type: 98RON
- Object **engine2**
  - Number of cylinders: 6
  - Horsepower: 163.0
  - Fuel type: Diesel
- Object **engine3**
  - Number of cylinders: 4
  - Horsepower: 36.5
  - Fuel type: ULP

Java: `private`

- Declaring a variable or method as **`private`** means that the variable can only be accessed in the block in which it is declared
- For methods this means the method is local to the class
- For **`private`** variables this means that if:
  - the variable is global to the class then it can be referred to anywhere within the class but is hidden from the outside world
  - the variable is local to a method then it can only be referred to within the method in which it is declared
- You should always explicitly state whether each method is **`public`** or **`private`**
- Variables declared within a method block will be **`private`** by default
- Variables declared as global to a class should always be explicitly declared as **`private`** (penalties apply if not)

Class - Pseudocode

```
CLASS Engine
    CLASS CONSTANTS:
        MAX_CYLINDERS := 16
    CLASSFIELDS:
        numCylinders (Integer)
        horsePower (Real)
        fuelType (String)
        hoses (ARRAY OF String)

    ...
```

Class - Java

```
/*********************************************************
 * Author:                                              *
 * Date:                                                *
 * Purpose:                                             *
 *********************************************************/
import java.anythingRequired.*;

public class Engine
{
    public static final int MAX_CYLINDERS = 16;

    private int numCylinders;
    private double horsePower;
    private String fuelType;
    private String[] hoses;

    ...

} // End Engine
```

Constructors

- Constructors are the methods used to create an object (i.e., to create an instance of a class)
- The role of a constructor is to ensure that the state of the object being created is correctly initialised
- There may be several constructors available
  - The difference between each of them is the IMPORT information used to initialise the state of the object being created
- There are three main categories of constructor:
  - **Default**
  - **Alternate** - IMPORT data used to initialise classfields
  - **Copy** - IMPORT another object of the same class i.e., new object will have a copy of the state from the IMPORT object

Constructors (2)

- Not all categories are required for every class
  - However, they are for this unit!
- All constructors must guarantee:
  - Classfields is initialised to VALID or special values

Default Constructor

- No **IMPORT**
- Initialises object state to defaults
- Defaults should be valid

Default Constructor - Pseudocode

```
DEFAULT CONSTRUCTOR
IMPORT: none
EXPORT: none   // Constructors never export
ASSERTION: Will create a Default state of an object
ALGORITHM:
    numCylinders := 1
    horsePower := 1.0
    fuelType := "ULP"
    hoses := NEW ARRAY OF EMPTY Strings
```

Default Constructor - Java

```
public Engine()
{
    numCylinders = 1;
    horsePower = 1.0;
    fuelType = "ULP";
    hoses = new String[MAX_STRING];
    for(int ii = 0; ii < hoses.length; ii++)
    {
        hoses[ii] = "";
    }
}
```

Alternate Constructor

- IMPORT values which are used to initialise class fields
- IMPORT should be validated (if possible)
  - If valid used to initialise class fields
  - If invalid then fail, or throw an exception (error)
- IMPORT information may be a direct reflection of class fields (e.g., **`DateClass -> inDay, inMonth, inYear`** to initialise class fields **`day`**, **`month`** and **`year`**).
- IMPORT information which is used to calculate, or otherwise obtain, values for class fields. For example:
  - A picture class which can be constructed:
    - As a blank image (number of rows and columns supplied as IMPORT to the constructor)
    - Read from a file (file name supplied as IMPORT to the constructor)

Alternate Constructor - Pseudocode

```
ALTERNATE CONSTRUCTOR
IMPORT: inNumCylinders (Integer), inHorsePower (Real),
        inFuelType (String), inHoses (ARRAY OF Strings)
EXPORT: none   // Constructors never export
ASSERTION: Will create an Alternate state of an object
ALGORITHM:
    IF NOT (1 < inNumCylinders < MAX_CYLINDERS) THEN
        FAIL
    IF inHorsePower < 1.0 THEN
        FAIL
    IF NOT validFuel <- inFuelType THEN
        FAIL
    IF inHoses = NULL THEN
        FAIL
    numCylinders := inNumCylinders
    horsePower := inHorsePower
    fuelType := inFuelType
    hoses := inHoses
```

Alternate Constructor - Java

```
public Engine(int inNumCylinders, double inHorsePower,
              String inFuelType, String[] inHoses)
{
    if(!(1 <= inNumCylinders && inNumCylinders <= MAX_CYLINDERS))
        throw new IllegalArgumentException("Invalid Cylinders");
    if(inHorsePower < 1.0)
        throw new IllegalArgumentException("Invalid Horsepower");
    if(!validFuel(inFuelType))
        throw new IllegalArgumentException("Invalid Fuel");
    if(inHoses == null)
        throw new IllegalArgumentException("Invalid Hoses");
    numCylinders = inNumCylinders;
    horsePower = inHorsePower;
    fuelType = inFuelType;
    hoses = inHoses   // You will need to make a copy
}
```

- **Note:** Exceptions are covered at the end

Copy Constructor

- IMPORT's an object of the same class
- Extracts the state information from the IMPORT object and uses that information to initialise the class fields for the new object
- The end result is that the new object will have a copy of the state information from the IMPORT object
- We have copied the IMPORT object

Copy Constructor - Pseudocode

```
COPY CONSTRUCTOR
IMPORT: inEngine (Engine)
EXPORT: none   // Constructors never export
ASSERTION: Will create a Copy state of an object
ALGORITHM:
    numCylinders := inEngine.getNumCylinders <- none
    horsePower := inEngine.getHorsePower <- none
    fuelType := inEngine.getFuelType <- none
    hoses := inEngine.getHoses <- none
```

Copy Constructor - Java

```
public Engine(Engine inEngine)
{
    numCylinders = inEngine.getNumCylinders();
    horsePower = inEngine.getHorsePower();
    fuelType = inEngine.getFuelType();
    hoses = inEngine.getHoses();
}
```

Accessor Methods

- AKA "Getters"
- Accessor methods are used to retrieve information from the object
- Accessor methods clearly define the data types of the information to be retrieved from the object
- Each Accessor method should retrieve exactly one piece of data
- Any public method which EXPORT's information which is either a copy of, or is calculated from, state information is, by definition, an accessor

Accessor Methods - Pseudocode

```
ACCESSOR: getNumCylinders
IMPORT: none
EXPORT: numCylinders (Integer)
ASSERTION: Will return the number of cylinders
ALGORITHM:
    EXPORT COPY OF numCylinders
```

- Note: **getHorsePower()**, **getFuelType()** and **getHoses()** are done in the same way

Accessor Methods - Java

```
public int getNumCylinders()
{
    return numCylinders;
}
public double getHorsePower()
{
    return horsePower;
}
public String getFuelType()
{
    return fuelType;
}
public String[] getHoses()
{
    String[] hoseCopy;
    hoseCopy = new String[hoses.length];
    for(int ii = 0; ii < hoses.length; ii++)   // Remember to add your
        hoseCopy[ii] = new String(hoses[ii]);  // blocks { ... }
    return hoseCopy;
}
```

Other Accessors

- A convention is that all classes should have additional two accessors:
  - **`equals()`**
    - Tests the equality of two objects
  - **`toString()`**
    - Generates a `String` representation of the state information
    - Various possibilities exist

equals - Pseudocode

```
ACCESSOR: equals
IMPORT: inObject (Object)
EXPORT: isEqual (Boolean)
ASSERTION: Will return true if the two objects are equivalent
    isEqual := FALSE
    IF inObject IS AN Engine THEN
        TRANSFORM inObject TO Engine NAMED inEngine
        IF numCylinders EQUALS inEngine.getNumCylinders <- none THEN
            IF horsePower EQUALS inEngine.getHorsePower <- none THEN
                IF fuelType EQUALS inEngine.getFuelType <- none THEN
                    IF hoses EQUALS inEngine.getHoses <- none THEN
                        isEqual := TRUE
```

equals - Java

```java
public boolean equals(Object inObject)
{
    boolean isEqual = false;
    Engine inEngine = null;
    if(inObject instanceof Engine)
    {
        inEngine = (Engine)inObject;
        if(numCylinders == inEngine.getNumCylinders())
            if(typeSame(horsePower, inEngine.getHorsePower()))
                if(fuelType.equals(inEngine.getFuelType()))
                    if(typeSame(hoses, inEngine.getHoses()))
                        isEqual = true;
    }
    return isEqual;
}
```

toString - Pseudocode

```
ACCESSOR: toString
IMPORT: none
EXPORT: engineString (String)
ASSERTION: Will return a String representation of the object
    engineString := "This engine has " + numCylinders +
                    " with " + horsePower +
                    " and uses " + fuelType
```

toString - Java

```java
public String toString()
{
    String engineString;
    engineString = "This engine has " + numCylinders +
                   " with " + horsePower +
                   " and uses " + fuelType);
    return engineString;
}
```

Back To Introduction

- Remember at the beginning:

```
Engine eng1 = new Engine(8, 450.0, "98RON");

System.out.println(eng1);
```

- `System.out.println()` calls `String.valueOf()` which in turn will call the appropriate `toString()` method if it exists
- It is sort of doing this:

```
System.out.println(eng1.toString());
```

- Is it starting to make sense yet?

Mutator Methods

- AKA "Setters"
- Mutator methods are used to send information into the object which is used to modify the values of classfields
- In other words, this new information *mutates* (changes) the object state
- Mutator methods clearly define the data types of the information to be passed into the object
- There will usually be a mutator method for each classfield
  - Except for when it doesn't make sense
- The actual representation of the information within the object is hidden and may not be the same as the IMPORT:
  - The data which is supplied to the method my be a direct reflection of variables in the object but this does not have to be the case

Mutators and IMPORT Validation

- If possible, the `IMPORT` used must be validated
  - If it is valid, then it is to be used to update the object state
  - If it is invalid then it is not used
    - The mutation is not allowed to occur
    - Throw an exception (covered at the end)
- If the validation of the `IMPORT` is done, then the rest of the software system can always assume that every object of that class has a valid state
- Of course, sometimes validation is not possible:
  - Person's name
  - Title of a book
  - A numeric value which has no upper or lower bounds on validity
- Generally we check **`null`** for `Strings`

Mutator Methods - Pseudocode

```
MUTATOR: setNumCylinders
IMPORT: inNumCylinders (Integer)
EXPORT: none
ASSERTION: Will update the state information of numCylinders if valid
ALGORITHM:
    IF NOT (1 < inNumCylinders < MAX_CYLINDERS) THEN
        FAIL
    numCylinders := inNumCylinders
```

Mutator Methods - Java

```java
public void setNumCylinders(int inNumCylinders)
{
    if(!(1 <= inNumCylinders && inNumcylinders <= MAX_CYLINDERS))
    {
        throw new IllegalArgumentException("Invalid Cylinders");
    }
    numCylinders = inNumCylinders;
}
```

Mutator Methods - Pseudocode (2)

```
MUTATOR: setHorsePower
IMPORT: inHorsePower (Real)
EXPORT: none
ASSERTION: Will update the state information of horsePower if valid
ALGORITHM:
    IF NOT (inHorsePower < 1.0) THEN
        FAIL
    horsePower := inHorsePower
```

Mutator Methods - Java (2)

```java
public void setHorsePower(double inHorsePower)
{
    if(inHorsePower < 1.0)
    {
        throw new IllegalArgumentException("Invalid Horse Power");
    }
    horsePower = inHorsePower;
}
```

Mutator Methods - Pseudocode (3)

```
MUTATOR: setFuelType
IMPORT: inFuelType (String)
EXPORT: none
ASSERTION: Will update the state information of fuelType if valid
ALGORITHM:
    IF NOT (validFuel <- inFuelType) THEN
        FAIL
    fuelType := inFuelType
```

Mutator Methods - Java (3)

```java
public void setFuelType(String inFuelType)
{
    if(!validFuel(inFuelType))
    {
        throw new IllegalArgumentException("Invalid Fuel Type");
    }
    fuelType = inFuelType;
}
```

One Mutator

- In certain circumstances, it doesn't make sense to have a mutator for each classfield
  - e.g., **`Date`**
    - Call **`someDate.setDay(31);`**
    - Then call **`someDate.setMonth(4);`** (April)
- In these cases, create one mutator for all fields.

Doing Methods

- The methods are used for performing some required task
- For example, a method which performs some calculation based on *more than* the state of the object, would be classified as a doing method

Private Methods

- These are the methods that are used within the object, but are never seen by the outside world
- They are hidden so that they can be modified or even replaced without causing problems in any other part of the software
- To perform a complex task, it is often easier to break the task down into a series of sub-tasks and then refine each sub task independently of the others
- i.e., The normal process of step-wise refinement still applies

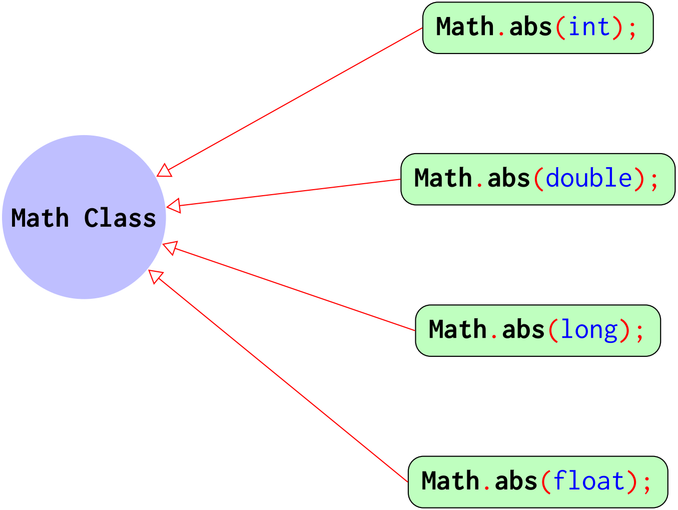
Private Methods - validFuel

```
PRIVATE: validFuel
IMPORT: inFuelType (String)
EXPORT: isValid (Boolean)
ASSERTION: Will return if inFuelType is valid
ALGORITHM:
    ...  // You can complete this in your own time
         // think about how you might validate a String
```

Method Overloading

- Method Overloading is the mechanism used to provide a variety of different possibilities within the same class
  - The method name and the arguments are known as the **Method Signature**
  - In Java it is implemented by matching the signature of the method call with the various possible signatures for the method

Method Overloading (2)



Method Overloading - typeSame

- Remember the method **typeSame()** from earlier?

```
private boolean typeSame(double a, double b)
{
   boolean same = false;
   if(Math.abs(a - b) < TOLERANCE) // Block braces omitted
      same = true;
   return same;
}
private boolean typeSame(String[] a, String[] b)
{
   int ii = 0;
   boolean same = true;
   if(a.length != b.length) // Block braces omitted
      same = false;
   else {
      while(a[ii].equals(b[ii]) && i < a.length) // Omitted
         ii++;
      if(ii < a.length) // Block braces omitted
         same = false;
   }
   return same;
}
```

Using a Class In a Program

- Somewhere else in the program, i.e., A different class

```
Engine eng1;
Engine eng2;
Engine eng3 = new Engine(4, 36.5, "ULP");

eng1 = new Engine(6, 46.0, "Diesel");
eng2 = new Engine(eng1);

if(eng1.equals(eng2))
{
    doSomething();
}
System.out.println(eng1.toString());

eng1 = new Engine(0, 123.4, "");  // What will happen?
```

Error

```
System.out.println(eng1.toString());

eng1 = new Engine(0, 123.4, "");  // What will happen?
```

- An exception will be thrown and the program will crash

Exceptions

- Error handling is a necessary task, but how do you do it elegantly?
  - Errors aren't "normal", you don't make a system that *expects* errors
  - But you **must** handle error situations
  - One solution: return an error code (see UCP `COMP1000`)
- Object-Oriented languages (such as Java) solve error handling with **exceptions**
  - An independent "return path" designed specifically for notifying the caller of an exceptional situation (error)
  - On an error, a method "throws" an exception
  - The calling method can "catch" the exception
    - If the caller doesn't catch it, the exception is thrown to the next highest caller
    - If no one catches it, the exception causes the program to crash

Exceptions (2)

- Java only lets objects of type **`Exception`** or its descendants to be thrown
  - Java has a range of classes descending (inheriting, extends) from **`Exception`**
    - e.g., **`IllegalArgumentException`** and **`ArrayIndexOutOfBoundsException`**
  - You may define your own **`Exception`** class, as long as it inherits from **`Exception`** (or one of its subclasses)
  - This will be covered in DSA (`COMP1002`)

Catching Exceptions

- Exceptions from different methods in different objects are often all caught at the one place in the calling method
- Somewhere close to **`main`**
  - Convenient: all error handling happens in one place
- Most languages use `try`, `catch()`, `finally` blocks
  - `try`: Define the set of statements whose exceptions will all be handled by the catch block associated with this `try`
  - `catch()`: Processing to do if an exception is thrown in the `try`
  - `finally`: Processing to always do, regardless of whether an exception occurs or not
    - Good for cleanup, e.g., closing files
    - This block is **optional** and executed *after* the `try` and `catch` blocks

Catching Exceptions - Example

```java
Engine eng1;
Engine eng2;
try
{
    Engine eng3 = new Engine(4, 36.5, "ULP");
    eng1 = new Engine(6, 46.0, "Diesel");
    eng2 = new Engine(eng1);
    if(eng1.equals(eng2))
    {
        doSomething();
    }
    System.out.println(eng1.toString());
    eng1 = new Engine(0, 123.4, "");
    // Code continues
}
catch(IllegalArgumentException e)
{
    // Do some fancy error handling here
}
```

Java Qualifiers

- `public`
- `private`
- `protected`
- `static`
- `final`

Next Week

- Next Lecture slot is your test. An announcement will be made shortly
- The next Lecture will address the following:
  - File I/O