Object Oriented Software Engineering (COMP2003)

Lecture 3: Separation of Concerns and Error Handling

Updated: 19th March, 2018

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

Outline

Separation of Concerns

- A design ideal – implies low coupling and high cohesion.
- A "concern" can be virtually any aspect of the system's functionality.
- As much as possible, different concerns should be handled by different sections of code.
- Makes it easier for the human brain to process!

Encapsulation

- Encapsulation is how separation of concerns is achieved in OO.
- A class represents a "concern", and hence some aspect of the system's functionality.
- A class has:
  - A **public interface**: all the publicly-visible method and constructor signatures, and any public constants.
  - An **implementation**: all the private fields/methods and the code itself.
  - (You could say it also has a "protected interface", visible to any subclasses.)
- Classes communicate via their public interfaces only.
  - The word "interface" is overloaded a bit.
  - Think of a Java interface as a kind of class that *only* has a public interface, and no implementation.

Access Modifiers

- Java, C++ and other languages use access modifiers:

  `public` – accessible everywhere.

  `protected` – accessible to subclasses only.

  `private` – accessible within this class only.
- Python has no equivalent.
  - But the principles are the same.
  - Fields and some methods should be *considered* private.
  - Some methods should be *considered* protected.
  - Simply don't access things you're not meant to!

An Interesting(ish) Example

```
public class Point
{
    private int x;
    private int y;

    public void add(Point other)
    {
        x += other.x; // But other.x is private...
        y += other.y;
    }
}
```

- Right? Wrong? Will it compile?
- (Same behaviour in Java and C++.)

What does `private` really mean?

- Access modifiers (`private`, `protected` and `public`) apply to classes, not to objects.
  - If a field/method is `private`, other classes cannot access it.
  - Other *objects* of the same class *can*!
- Is this a horrible flaw in both Java and C++?
- Access modifiers help enforce low class coupling.
  - High coupling exists if classes interfere in the internal workings of other classes.
  - This reduces readability and maintainability, and will increase the risk of creating faults.
- Coupling between *individual objects* is meaningless – they don't have their own code.
  - Two objects of the same class are described by the *same code*.
  - That class must access its own private fields anyway.

Architectural Separation

- A few SE patterns are large-scale, "architectural" patterns
  - Especially for GUI/web/mobile applications.
- They govern the overall structure of the software.
- Some examples for a:
  - Model-View-Controller (MVC)
  - Model-View-Presenter (MVP)
  - Model-View-Adapter (MVA)
  - Model-View-ViewModel (MVVM)
- Spot the common theme. (We won't worry about the differences yet.)
- They all divide a software system into:
  - The "model" – the classes that store information.
  - The "view" – the user interface.
  - The leftover bit.

The Model

- The model represents real-world concepts that the system deals with.
- The model stores information.
  - Classes in the model often just have simple accesses and mutators.
  - But the model may also load/save itself.
    - If there's a database, the model connects to it.
- For example, in a class registration system:
  - `Class`, with subclasses `Lecture` and `Practical`;
  - `Student`;
  - `Venue`;
  - `Unit`;
  - `ClassInfoLoader` (or something similar).

The View

- Also called the user interface (UI).
- Basically handles user input/output.
- The UI/View can be radically different depending on the type of software you're making:
    - Console-based programs often take all their input "up-front" on the command-line.
    - Desktop and mobile apps have back-and-forth user interaction.
    - Web applications need to transmit and receive UI information across the Internet.
    - A lot of software has no real UI at all.

The Leftover Bit

- Not everything is part of the View or the Model.
- There's also the decision-making/business-logic part (i.e. the brains).
- We often call this the "Controller".
  - You may have several controllers, one related to each part of the UI.
- This might be split up in other ways too.
  - If the app is not driven by user interaction, "model-view-controller" is *not* a good way to think about it.

Architectural Separation of Concerns

- Don't mix up the responsibilities of the model and view.
- Tempting, but very bad, to:
  - Store authoritative information in the user interface.
  - Input/output information from within the model.
- This is the old "cohesion" thing.
  - Methods and classes should perform one well-defined task.
  - So too should large-scale parts of the system.
- You might need to work on one part only (e.g. just the UI).
  - Much simpler (and cheaper!) if it's highly cohesive.

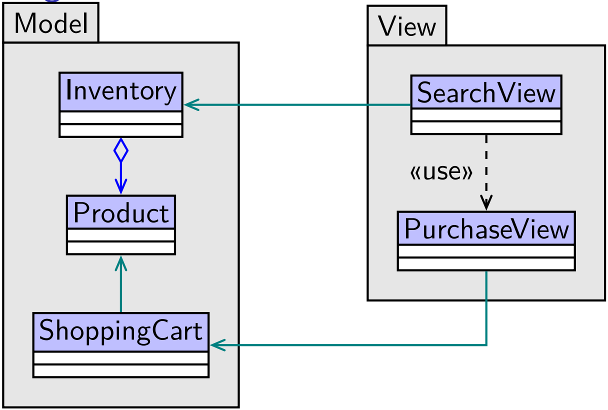
View-Model Interaction

- The View and the Model must exchange information.
  - The view must have something to output.
  - The model's data must ultimately comes from the user.
- Each architectural pattern does this differently.
- The View and the Model should have *very low* coupling.
  - Later lectures will cover some techniques for this.

Packages and Namespaces

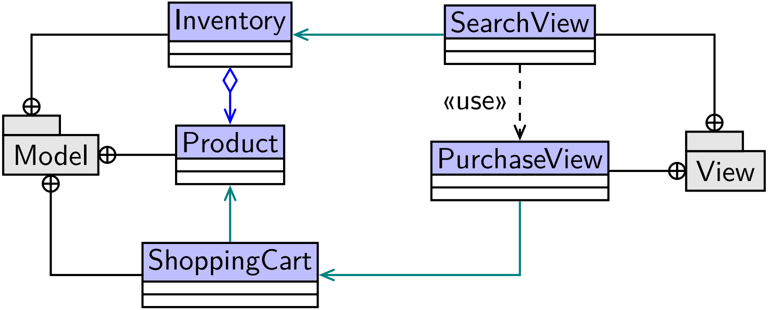
- UML, Java, C++ and Python all allow grouping of classes.
- UML, Java and Python support “packages”.
- C++ supports namespaces.
- The concepts have substantial overlap.

Packages in UML



- Classes/interfaces drawn inside a package box belong to that package.
- You can have packages-inside-packages too.
- Packages don't affect relationships between classes/interfaces.

Packages in UML (Alternate Notation)



- Alternatively, use "⊕——" (the circle-plus notation) to indicate things inside packages.
- This is probably worse! More lines and less visual grouping.
- *But* we'll discuss another use for this symbol soon.

C++ Namespaces

- Classes can be declared inside a namespace like this:

```
namespace Model
{
    class ShoppingCart
    {
        ...
    };
};
```

- The full name is now **Model::ShoppingCart**.
- In the .cpp file, write **Model::ShoppingCart::addProduct(...)** for each method.

C++ Namespace Members

- One namespaces can contain multiple classes, as well as functions and even *other namespaces*.
- However, everything is public.
- You can define parts of a namespace in different places:

```
namespace Model
{
    class ShoppingCart { ... };
};
...
namespace Model // Perhaps in another file
{
    class Product { ... };
    Product* findProduct(std::string name);
};
```

Model contains **ShoppingCart**, **Product** and **findProduct**.

C++: Importing from Namespaces

- You can always write **Model::Product** to access a class inside a namespace.

```
Model::Product product("cup");  // Create an object.
```

- But you can also write:

```
using Model::Product;
...
Product product("cup"); // Same as above, but nicer.
```

  - If you use `Product` a lot, this can make things neater.
  - But *only* do this in .cpp files, not header files.
- You may also sometimes see this:

```
using namespace std;                // Lazy programmer.
```

  - A *bad idea*. Lots of strangely-named stuff in `std`, and naming conflicts may result in very strange bugs.

Java Packages

- Java packages are *sort-of* hierarchical.
  - They *look* hierarchical.
  - They're *stored* hierarchically in directories.
  - But, at the language level, packages do not contain other packages – Java treats them all separately.
- Standard API packages include `java.io`, `java.util`, etc.
- There is a naming convention for all other packages:
  - They start with your website domain name, in reverse order.
  - e.g. if you own `http://mysite.org`, your package names should start with **`org.mysite`**.
  - Then, for a given application, you might have a series of packages as follows:
    - **`org.mysite.retailapp`** – the main package;
    - **`org.mysite.retailapp.ui`**;
    - **`org.mysite.retailapp.controller`**;
    - etc.
  - This is just an example! The latter parts of the package name are up to you.

Java Packages: Files and Directories

- Each Java class is stored as a .class file.
- Each Java package is stored as a directory containing .class files.
  - Each part of the package name maps onto a subdirectory.
  - e.g. for the package `org.mysite.retailapp`, class files will be located in the directory `org/mysite/retailapp/`.
  - But where is the top-level "org" directory located?
- Conventionally, .java and .class files are located in separate places.
- The `CLASSPATH` indicates where to find the *top-level* classes and packages:

$$\underbrace{\texttt{/home/user/projects/}}_{\text{Part of CLASSPATH}}\overbrace{\texttt{org/mysite/retailapp/}}^{\text{Package directory}}\texttt{Xyz.class}$$

Java Packages in Code

- Inside your source code, you also need a package declaration.
- Say class `MyWindow` is inside package **`org.mysite.myapp.ui`**.
- You must have:

```
package org.mysite.myapp.ui; // Before imports

import somepackage.SomeClass;

public class MyWindow
{
    ...
}
```

- (This is on top of keeping `MyWindow.java` inside the directory `org/mysite/myapp/ui/`.)

Using Java Packages

- Java's "`import`" is comparable to C++'s "`using`".
- We *could* write this:

```
org.mysite.myapp.App obj = new org.mysite.myapp.App();
```

- But we prefer to write this:

```
import org.mysite.myapp.App;
...
App obj = new App();        // Same as above but nicer.
```

- We can also write `import org.mysite.myapp.*;`
  - Imports everything from the package.
  - Similar to C++'s "`using namespace`", but not as bad in Java, because...
  - Java is simpler – harder for spooky naming-conflicts to happen.
  - Java's API is divided up into much more fine-grained packages.
- nb. `import` is not recursive. "`import java.*;`" has no effect.

Java's `import` Statement

- Java's `import` can be used several ways:
  - Import all classes/interfaces in a package:

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

  - Import a specific class/interface:

```
import java.util.Collection;
```

  - Import all methods in a class/interface:

```
import static java.util.Collections.*;
```

  - Import methods with a specific name (possibly several due to overloading):

```
import static java.util.Collections.max;
```

- The package `java.lang` (containing `System`, `String`, `Object`, `Integer`, `Math`, etc.) is imported automatically.

Package Encapsulation

- Java can enforce encapsulation on a package level; e.g.

```
public class Product { ... }
```

```
class Product { ... }            // Package-private class
```

  - A public class can be accessed outside its package.
  - A non-public class cannot; it is "package-private".
- Contents of classes can also be made package-private.
  - This is the default if you omit `public`, `protected` or `private`.

```
public class MyClass
{
    private int x;
    MyClass() {...}             // Package-private constructor
    void setX(int newX) {...} // Package-private method
    public int getX() {...}
}
```

Python Packages

- Python packages are quite similar to Java, but not exactly the same.
- Like Java:
  - Python packages are arranged in directory trees.
  - Top-level package directories must be in `PYTHONPATH` (which works like `CLASSPATH`).
  - There's an `import` statement for accessing packages.
- Unlike Java:
  - Each package directory must contain a file called "`__init__.py`".
    - This is the package constructor.
    - Often an empty file will do, but it must exist.
    - It can contain code for initialising the package, if necessary.
  - Python packages contain *modules*, which contain classes (and functions).
    - This is an extra layer that doesn't exist in Java.

Python Modules

- Every .py file represents a module.
- Everything at the topmost scope of a .py file is a member of that module.
  - Except for things starting with "_".
- Module members can be imported into other modules.
- e.g. Say the file `myPkg/otherPkg/myModule.py` contains:

```
class MyClass:
    ...

def myFunction(s):
    ...
```

  - `MyClass` and `myFunction` are both members of `myModule`.
  - `myModule` itself is a member of `myPkg.otherPkg`.

Python's import Statement

- Importing a module:

```
import myPkg.otherPkg.myModule
...
myModule.myFunction("Hello")
```

- Importing a member of a module:

```
from myPkg.otherPkg.myModule import myFunction
...
myFunction("Hello")
```

- Python's API is mostly made up of global modules; e.g.

```
import sys    # Standard Python module; not in a package.
```

- Python supports "`from ... import *`", similar to Java.
  - Not widely used in Python.
  - Discouraged for similar reasons to C++'s "`using namespace`".

Higher-level Encapsulation

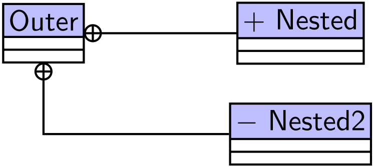
- Like individual classes, larger-scale parts of the software have a public interface and private implementation.
- The public interface consists of all the classes intended to be used by the outside world.
- The implementation is everything else – all the private or package-private classes, constructors, methods and fields.
- This applies to the model, the view and the controller, and to any other ways of breaking up the software.
- Encapsulation works at several levels at once:
  - Package-private classes still have an interface and implementation.
  - Their internal workings are still protected from the rest of the package.
- The very largest components – e.g. the view – may consist of several entire packages. Some may be purely implementation.

Nested Classes

- Supported by Java, C++, Python and others, but use sparingly.
- Very occasionally, one class may form a natural part of another.
  - We're *not* talking about "is-a" or "has-a" relationships.
- When this happens, you can define one class *inside* another.
- The nested class may:
  - Participate in a complex part of the parent class's implementation, and/or
  - Help the parent class interact with other classes.
- Overuse may violate encapsulation, and lead to excessively large files.

Nested Classes: UML

The circle-plus notation again:



- Here, Nested and Nested2 are defined within the Outer class.
  - Nested is public, while Nested2 is private.
  - They can still do anything a normal class does.
- Bending the UML specification a bit...
  - Technically, this notation is defined for things-in-*packages*.
  - But it seems logical to use it for things-in-classes too.
- Relatively rare in practice.
  - It *is* useful to describe certain OOSE concepts.
  - But nested classes can be a fairly low-level detail (once you understand them!).

Nested Classes: C++

- In C++, nested classes behave a bit like classes inside namespaces, except:
  - The nested class can be private or protected.
  - You can't avoid using the parent class's name ("using namespace" won't work on class names).

```
class Outer
{
    public:
        class Nested {...};
    private:
        class Nested2 {...};
    ...
};
```

- Instead of a private nested class, you could define a separate class inside the .cpp file.

Nested Classes: C++

- The C++ containers (**vector**, **list**, **map**, **set**, etc.) each have a nested class called **iterator**.
- This declares an iterator for iterating over a vector of MyClass pointers:

```
std::vector<MyClass*>::iterator it;
```

Nested Classes: Java

```
public class Outer
{
    public static class Nested {...};
    private static class Nested2 {...};
    ...
}
```

- This works more-or-less like in C++.
- Interfaces and classes can also be nested inside each other.
- Use `static` (as above) to get a normal nested class.
- Remove `static` to get an *inner class*.
  - Outside of Java, "inner class" = "nested class".
  - In Java, inner classes are a special kind of nested class.
  - Each inner class *instance* (not just the class itself) is linked to an outer class instance.
  - An implicit association between the inner and outer classes.

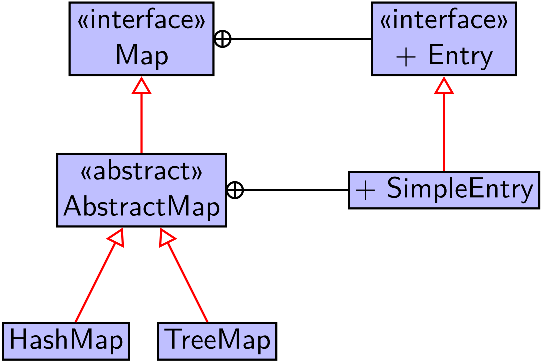
Nested Classes: Java – Example from the API

- **java.util.Map** (an interface) contains a *nested interface* called **Map.Entry**.
- Used to iterate over both keys and values in the map at the same time:

```
for(Map.Entry entry : theMap.entrySet())
{
    System.out.println(
        entry.getKey() + "->" + entry.getValue());
}
```

- The nested class **AbstractMap.SimpleEntry** implements the nested interface **Map.Entry**.

Nested Classes: Java – Example from the API



Nested Classes: Python

- Python nested classes are always public, of course:

```
class Outer:
    class Nested:
        ...

    ... # Rest of "Outer"
```

Another Interesting(ish) Example

```
public class Outer
{
    // Cannot be seen by the outside
    private static class Nested { ... }

    public static Object getObj()
    {
        // Return an instance of a private class:
        return new Nested();
    }
}
```

- Right? Wrong? Will it compile?
- Again, same behaviour in Java and C++.

Instances of Nested Classes

- On the previous slide, the class `Nested` is private.
- But, once created, all objects are equal, including instances of `Nested`.
- Such objects can be passed around like any other objects.
- Their exact class will be unknown to the outside world.
- That's where polymorphism comes in.

Exceptions

- Exceptions are interruption to the normal flow of control.
  - Designed mostly for error handling.
  - Exceptions can break out of an algorithm when something goes wrong.
  - Why not `if` statements? Exceptions are often simpler – fewer boolean conditions and scopes.
- Instances of the **`Exception`** class represent exceptions.
  - Java, C++ and Python all have this standard class.
  - However, in each language there are different subclasses.
  - Different kinds of exceptions can occur.

Creating an Exception

Exceptions occur when:

- You execute a `throw` statement (or "`raise`" in Python):

```
throw new MyExceptionType();
```

- You execute an expression with invalid operands. For example, in Java:

```
obj = null;
obj.method();  // NullPointerException
```

```
int[] array = new int[5];
array[25] = 0; // ArrayIndexOutOfBoundsException
```

```
num = 1/0;      // ArithmeticException
```

- You call a method that throws (doesn't handle) an exception.

Throwing Exceptions

- Without a "`try-catch`" statement, exceptions immediately end a method.
- This *should* happen when that particular method is *not responsible* for resolving the issue.
- Say you have a chain of method calls:

```
public void a() { b(); ... }
public void b() { c(); ... }
public void c() { int i = 1/0; ... }
```

  - **c**() triggers an `ArithmeticException`.
  - **c**() throws the exception to its caller, **b**(), which throws it to its caller, **a**(), which throws it to its caller.
  - When each method receives the exception, it immediately ends.
  - The rest of the code in **c**(), **b**() and **a**() is skipped over.
- An exception is thrown up the call tree until one method "catches" (handles) it.

Declaring Thrown Exceptions

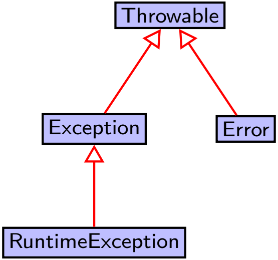
- In Java (but not Python or C++), you must state what types of exceptions your method might throw.

```
public void myMethod() throws MyExceptionType
{                              Can be a comma-separated list
    if(...)
    {
        throw new MyExceptionType("Some message");
    }
}                                                    // Java
```

- This is part of the method's signature, along with its parameters and return type.
  - If you override the method in a subclass, it cannot throw any extra exception types.
- This applies to "normal" exceptions, but there are some special cases. . .

Exception Hierarchy

- In Java (and Python), there are Exceptions and "Errors".
  - Errors include OutOfMemoryError, LinkageError, AssertionError and others.
  - Errors are often unresolvable – a bug, or a VM-level problem.
- Java also has "Runtime Exceptions".
  - These include NullPointerException, ArithmeticException, etc.
  - Often these indicate a bug, which you can't usually handle; other times user errors, which you can.



- Since you often can't handle them, you don't need to declare Errors or RuntimeExceptions.
- However, they all essentially work the same way.

Catching Exceptions

- One method must be responsible for resolving an exception.
- It uses a try-catch statement:

```
try {
    // Do something that may produce an exception.

    // Then (if needed) do something else that
    // assumes no exception has occurred.
}
catch(MyException e) {
    // Handle a particular kind of exception.
}
catch(MyOtherException e) {
    // Handle a different kind of exception.
}                                          // Java
```

(1 or more catch blocks are permitted.)

Catching Exceptions

- If an exception occurs, the rest of the `try` block is skipped.
  - This is why the `try` block should also contain any code that assumes no exception has occurred.
- If the exception type matches one of the `catch` blocks, that block is executed.
  - Each `catch` block handles one type of exception only.
- Inside the `catch` block, you can examine the exception object.

```
try {...}
catch(MyException e) // e is the exception object
{
    System.out.println("Oops: " + e.getMessage());
    ...
}                                        // Java
```

- If no `catch` block matches, the method ends as if there was no `try-catch` at all.

Exceptions in C++

- To generate an exception (notice the lack of new):

```
throw MyExceptionType("Some message");          // C++
```

- To catch an exception (notice the & symbol):

```
try {
    ... // Do something that might go wrong
}
catch(MyExceptionType& e) {
    ... // Handle one type of exception
}
catch(MyOtherExceptionType& e) {
    ... // Handle another type of exception
}                                               // C++
```

- Declaring exceptions thrown is broken in C++.
  - You can do it, but it's practically useless.

Exceptions in Python

- To generate an exception:

```
raise MyExceptionType("Some message")        # Python
```

- To catch an exception:

```
try:
    # Do something that might go wrong
    ...
except MyExceptionType as e:
    # Handle one type of exception
    ...
except MyOtherExceptionType as e:
    # Handle another type of exception
    ...                                      # Python
```

- You can't declare exceptions thrown from a Python method.
  - A method may throw any exception.

How to Handle Exceptions

- What exactly should you do inside a `catch` block?
- Depends on the situation and the type of exception.

Can you try the same thing again?

- If the exception was caused by faulty input, perhaps you can ask the user again. (e.g. `try-catch` inside a loop.)

Do you have a backup plan?

- Perhaps you failed to read a data file, but there are hardcoded default values you can use instead.

Were you trying to do something "nice but not esssential"?

- Maybe you can get away without it.

Is it a localised problem in one feature of the software?

- Keep the program running. Report the failure to the user.

Is it a catastrophic error that throws everything into disarray?

- Save the user's data! (And explain what happened.)

Exceptions and Responsibilities

- Throwing exceptions is part of a method's "contract" with the calling method.
  - If a method can throw an exception, that must be documented.
  - If you call that method, you must handle the exception.
  - In Java, exception throwing is part of a method's signature, like the parameter list.
- Don't throw confusing exception types.
  - Each method has its own area of responsibility.
  - Any exceptions it throws should reflect that responsibility.
- Often you need to `catch` one exception type and re-`throw` a different one.
  - The original exception may not reflect the method's purpose, and may be confusing for a calling method to deal with.

Catching and Rethrowing Exceptions (Example)

- Consider a method for displaying an image.
  - As an implementation detail, it reads image data from a file.
  - File I/O can trigger a **IOException**.
  - If this happens, the method must throw an exception – it has no sensible way to fix the problem.
  - *However*, the calling code *should not know* about the file IO (separation of concerns).
- The solution: `catch` the **IOException**, and re-`throw` a `GraphicsException` (for instance).

```
public void drawImage(...) throws GraphicsException
{
    try { ... /* IOException may happen here. */ }
    catch(IOException e) {
        throw new GraphicsException("Bad img file", e);
    }
} // GraphicsException is more meaningful.      (Java)
```

Catching and Rethrowing Exceptions (Discussion)

In the previous example:

- We made up `GraphicsException` – see next slide.
- We wrote `new GraphicsException("Bad img file", e);`
  - `e` is the original `IOException`.
  - We're saying that `e` is the "cause" of the `GraphicsException`.
  - This is important for debugging. A stack trace will now show the whole history of both exceptions.
- Code that calls `Graphics.drawImage()` now only has to deal with `GraphicsException`.
  - Simplifies the usage of that method, particularly if a `GraphicsException` can occur for *other* reasons too.
  - Helps achieve separation of concerns.

Defining Exception Types

- If `GraphicsException` doesn't exist, we can create it:

```java
public class GraphicsException extends Exception
{
    public GraphicsException(String message,
                             Throwable cause)
    {
        super(message, cause);
    }
} // That's the entire class!                (Java)
```

- The constructor takes an existing exception and sets it as the "cause" of this new exception.
- In Java, you may also have constructors that omit the message, the cause, or both.

Defining Exception Types: Python and C++

- Python makes it trivial:

```
class GraphicsException(Exception):
    pass                                    # Python
```

  - "pass" does absolutely nothing. It's only needed because, syntactically, you can't have an empty block in Python.
  - The constructor `__init__()` is inherited, so you don't need to write another one.

- In C++:

```
#include <stdexcept>

class GraphicsException: public std::exception
{
    ...
}                                           // C++
```

Final Actions

- Often you need to guarantee certain “clean up” actions.
  - Saving data, closing files, closing GUI windows, etc.

```
try { // A try-finally statement
    ...
}
finally {
    // This code *always* runs.
    ...
}
// This code runs only if there was no exception.
...                                              // Java
```

- The `finally` block runs after the `try` block, *regardless* of whether an exception has occurred.
  - And not just exceptions, but `return`, `break` and `continue` too.

Finally in Python and C++

- In Python:

```
try:
    ... # Something that might go wrong
finally:
    ... # This code always runs.            (Python)
```

- C++ has no “try-finally” statement.
- You can achieve a similar effect using destructors.
  - Declare/construct an object on the stack.
  - When the function ends, it’s destructor is guaranteed to be called.