Objects and Classes

- Classes are types of data
- Objects are instances of classes
- Classes define objects
- Constructors create objects

• Encapsulation

- Classes encapsulate state and behaviour
  - Hides implementation detail
  - Implementation can be changed if interface remains constant

- The "Law of Demeter" talks about only talking to "friends", or not trying to break apart internal functionality where useful methods already exist.

- Classes are organised into packages.

- Encapsulation has several levels:
  - Public (keyword: public)
    - Visible to any class in the system
  - Protected (keyword: protected)
    - Visible to the class and its sub-classes (any package)
  - Package Protected (no keyword, default)
    - Visible in any class in the same package
  - Private (keyword: private)
    - Visible only in the same class.

• APIs

- A style of encapsulation to make code reuseable.
- Users only need to understand the public halfs, the Application Programming Interface (API)
- Implementation can be ignored
- APIs are documented with JavaDoc

• Inheritance

- Super-classes capture commonalities
  - Attributes, methods and responsibilities
- Sub-classes...
  - inherit attributes and methods of their super-class
  - can add new ones.
  - can change values and override methods
  - can invoke super-class methods with super.m()
  - MUST respect Liskov's Substitution Principle:
    - Any method of a super-class must not be overwritten in a way that breaks expetation

REMEMBER THIS!

• Polymorphism

- Means invoking different functionality using the same method name.
- Concrete functionality is decided based on the run-time type of the object.

Eg.

class Shape {...}

- class Square extends Shape {...}
- class Circle extends Shape {...}

• Abstract Classes

Eg. public abstract class Shape {...}

- These classes can have defined and abstract methods
- They cannot be instantiated
  - Instantiation requires extension by a non-abstract class, which must implement all abstract methods.

• Interfaces

- The "ultimate" abstract classes.
- They can contain only abstract methods
- Classes may implement any number of interfaces

• Basis of OOP

- Nouns → potential classes
- Verbs/verb phrases → potential methods

- An interface should use the throws keyword to indicate the error cases that may occur.

Model / View / Controller (MVC) Structure

- Basic 3-Tier Structure:

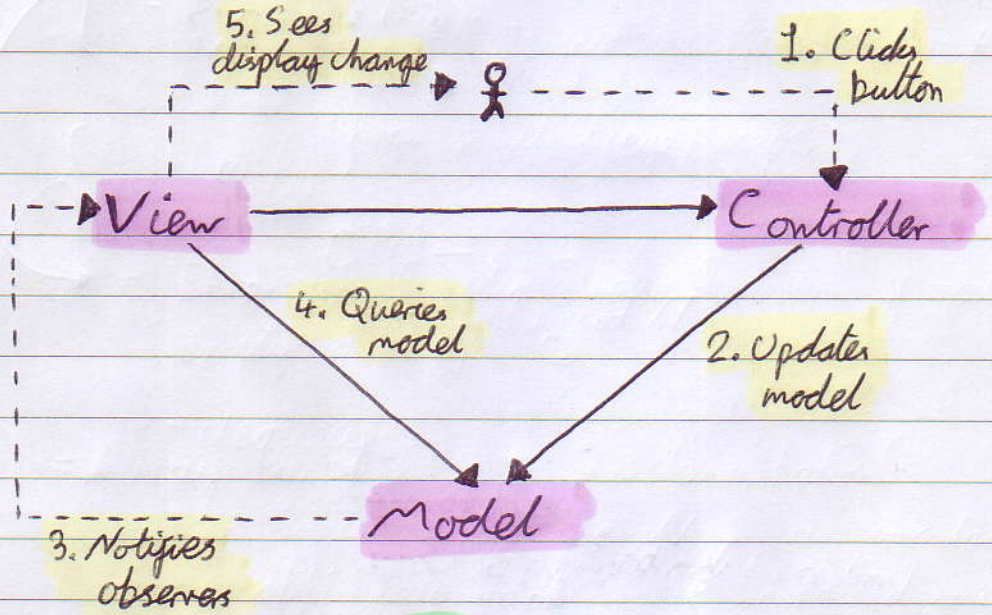
- Updating Views w/ an Observer

```
public class MainApp extends Observable {
    ....
    public void changeAValue() {
        value = value + 3;
        setChanged();
        notifyObservers();
    }
}

public class Alarm implements Observer {
    ....
    public Alarm(MainApp m) {
        ....
        m.addObserver(this);
    }

    public void update() { ~~~ }
}
```

TRIGGERS

• MVC Structure:

- **Model**: maintains data
  notifies view of any changes
- **View**: display current model state
  register as observers with the model
- **Controller**: manipulate model state
  react to user input



Note: Many Java Swing widgets are views and controllers in one object

Coding Practises

- Choose informative names
- Use naming conventions - getX, setX, isX.
- Encapsulation and polymorphism are your friends
- Good commenting
- DRY - Don't Repeat Yourself
- YAGNI - You Ain't Gonna Need It.

Basic GUI Components

- Key Elements:
  - Widgets
  - Top-level containers
  - Layouts
  - Events and actions → Next topic

• Widgets

Purpose	Widgets
Information	Labels
Action	Buttons, menus, etc.
Choice	Radio/Check boxes, spinners, combos, etc.
Free Text Input	Text field/box, password field, etc.

Note: some widgets must be encased in a JScrollPane in Java to enable scroll bars.

• Positioning + Layouts

- One option is absolute positioning
  - Uses a cartesian coordinate grid
  - Simple
  - Very, very inflexible

- A better option is to use layout managers.
  - Intentions are declared (minimise width, spread evenly, etc.) and the actual control is given to a layout manager.
  - More complex
  - Flexible, as redraw is automatic on resize or other changes

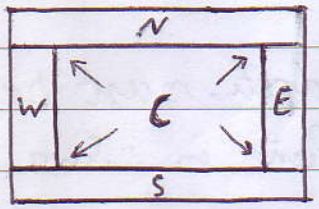
- In a layout manager, content is arranged in slots. Each slot can only have one child, but that child may be a container with its own layout.

• Common Layouts in Java Swing

- **Flow Layout**
  - Simple layout in horizontal rows

- **Grid Layout**
  - Simple grid-based layout

- **Border Layout**
  - North, East, South, West and Centre:



- **Grid Bag Layout**
  - Sent by the devil.

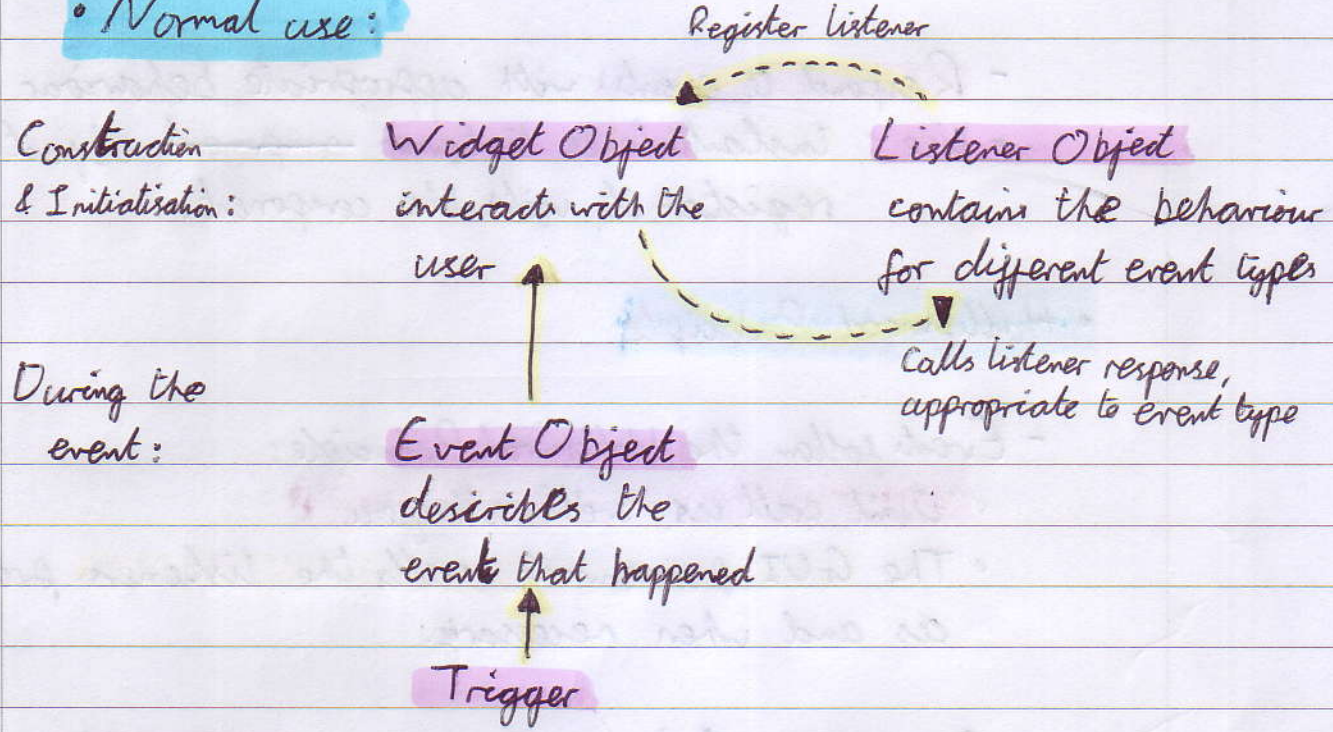
• Using Layout Managers

- Construct a container
- Construct a layout manager
- Set the layout of the container to the manager
- Construct and add components to the container
- Call .pack() on the container before making it visible.

Events and Actions

- Basic event handling involves three parts:
  - Widget, like a JButton
  - Event, like a button press
  - Event handler, which is a method (eg: ActionPerformed() in a widget implementing ActionListener).

- Normal use:



- Interfaces

  - Event handlers are objects that implement an interface
    - For ActionEvents, they implement ActionListener.
  - These define what methods can be invoked
  - Event handlers are registered using a widgets
    - addXYZListener() methods
      - XYZ = Click, DoubleClick, etc.

• Event Objects

- Capture details of the event that occured.
- When an event is triggered, an event object is automatically constructed containing details about the source component and the event type.

• Listeners

- Respond to events with appropriate behaviour
- Use: instantiate a listener ~~component~~ object and register it with the component.

• Hollywood Principle

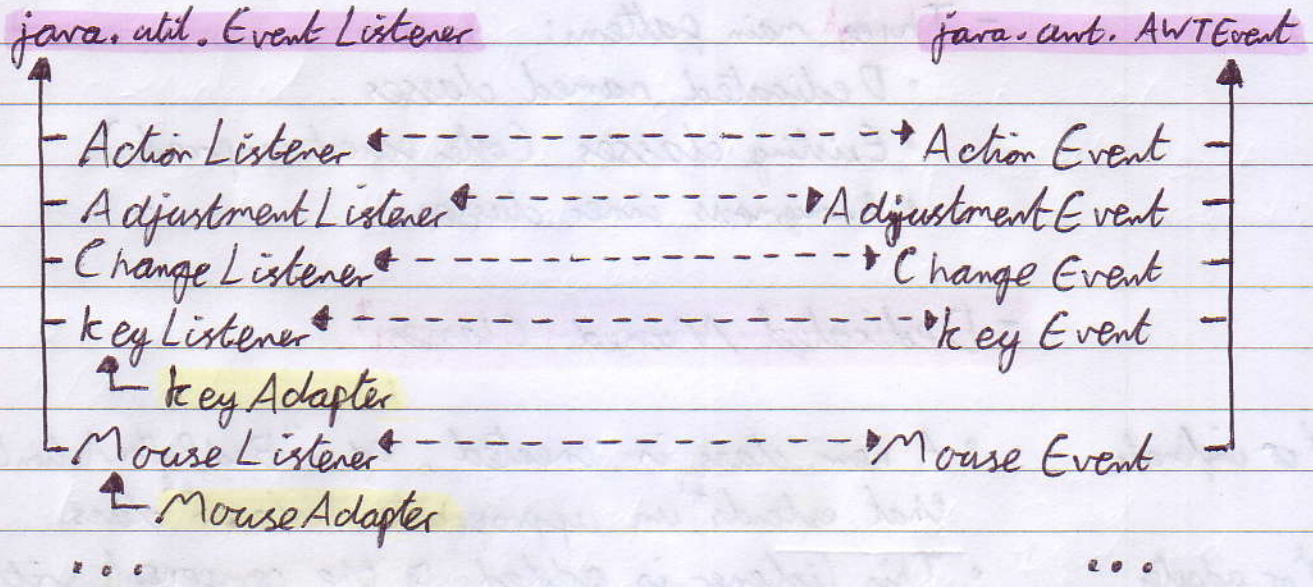
- Events follow the Hollywood Principle:
  - "Don't call us, we'll call you"
  - The GUI framework calls the listeners provided as and when necessary.

- The framework is customised by implementing framework interfaces
  - White-box reuse

- This is different from building a GUI by combining instances of framework classes
  - Black-box reuse

NB: Black-box: internals are unknown, just a component interface
White-box: internals are known, usually reuse means inheritance/interfaces.

• Types of Event



Why Adapter?

- KeyListener offers multiple methods:
- Usually only one is wanted, but an interface cannot be instantiated

```
KeyListener
void keyPressed(...);
void keyReleased(...);
void keyTyped(...);
```

- KeyAdapter contains dummy methods that can be overridden one-by-one:

```
KeyAdapter
void keyPressed(...) { }
void keyReleased(...) { }
void keyTyped(...) { }
```

• Listener Implementation Patterns

- Three main patterns:
  - Dedicated named classes
  - Existing classes (often parent component)
  - Anonymous inner classes

- Dedicated Named Classes

*1 or implements

*2 or adapter

- A new class is created, eg. "BackButtonActionListener" that extends*1 an appropriate Listener*2 class.
- This listener is added to the component with the widget's appropriate .add__Listener() method

- Existing Classes

- An existing class implements an appropriate listener interface
  - This is often the parent component, so a class extending JFrame may implement the listener for a JButton child.

- This existing class is added as a listener in the same manner described above.

- Anonymous Inner Classes

- Nested classes: classes can contain classes

```
public class Outer {          <-- Top-level class
    public class ~~Inner~~ Nested {   <-- Outer.Nested
        ...
    }
}
```

- Usage of Nested/Inner Classes

Case 1:

```
public class Outer {
    public [static] class Nested { }
}

Outer.Nested n = new Outer.Nested();
```

Case 2:

```
public class Outer {
    public class Inner { }
}

Outer o = new Outer();
Outer.Inner i = o.new Outer.Inner();
```

- Outer.Nested is almost the same as a top-level class.
- In Case 1, n cannot access members of Outer.

- In Case 2, i is created in the context of o.
- i can access members of o, an instance of Outer

- Inner classes must be instantiated in the context of an outer class object:

```
Outer o = new Outer();
Outer.Inner i = o.new Outer.Inner();
```

← like this

```
public class Outer {
    Inner i = this.new Inner();
}
```

← or like this

- Anonymous Inner Classes

  - Combines definition and instantiation of an inner class
    - When it is only needed in one place
    - When we extend/implement another class/interface

```
public class ~~Interface~~ Outer {
    ...
    SomeInterface si = new SomeInterface() {
        public void someMethod() {
            ...
        }
    };
    ...
}
```

Use "new" on an interface

Define methods in-place

- Inner Classes and Scope

NB: Inner != Nested

  - Inner classes can access fields and methods of the containing class.

  - Inner classes can access final local variables.*

  - Inner classes can see anything the outer class can see

* ie. declared in the same method as the IC.

Regular Expressions

- Regular expressions define sets of strings in a pattern format.

- Pre-defined ranges:

\d	a digit
\D	a non-digit
\w	an alphanumeric character (inc. underscore)
\W	a non-alphanumeric character
\s	a whitespace character
\S	a non-whitespace character
\n	a new line
.	any character except NL

- Defining Ranges

[abc]	"a", "b" or "c"
[a-e]	"a", "b", "c", "d" or "e"
[a|-e]	"a", "|" or "e"
[^a-c]	NOT "a", "b" or "c"

$(regex_1 | regex_2)$ means "matched by $regex_1$ OR $regex_2$"

- Quantifiers

?	0 or 1
*	any number
+	1 or more
{n}	exactly n
{n,}	at least n
{n,m}	between n and m inclusive

• Using RegEx

Method One

```
String testString = "test me";
boolean okay = testString.matches("regex");
```

Method Two

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

String testString = "test me";
boolean okay = Pattern.matches("regex", testString);
```

Method Three

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

private static Pattern dateRegex = Pattern.compile("regex");

public boolean verifyDate(String input) {
    return dateRegex.matcher(input).matches();
}
```

Which to Use?

If the same match is to be repeated many times, use #3 as the pattern is only compiled once.

Otherwise, use any.

The pattern is compiled implicitly every time #1 or #2 is ran.

Exceptions

- Exceptions allow separation of core code and error handling

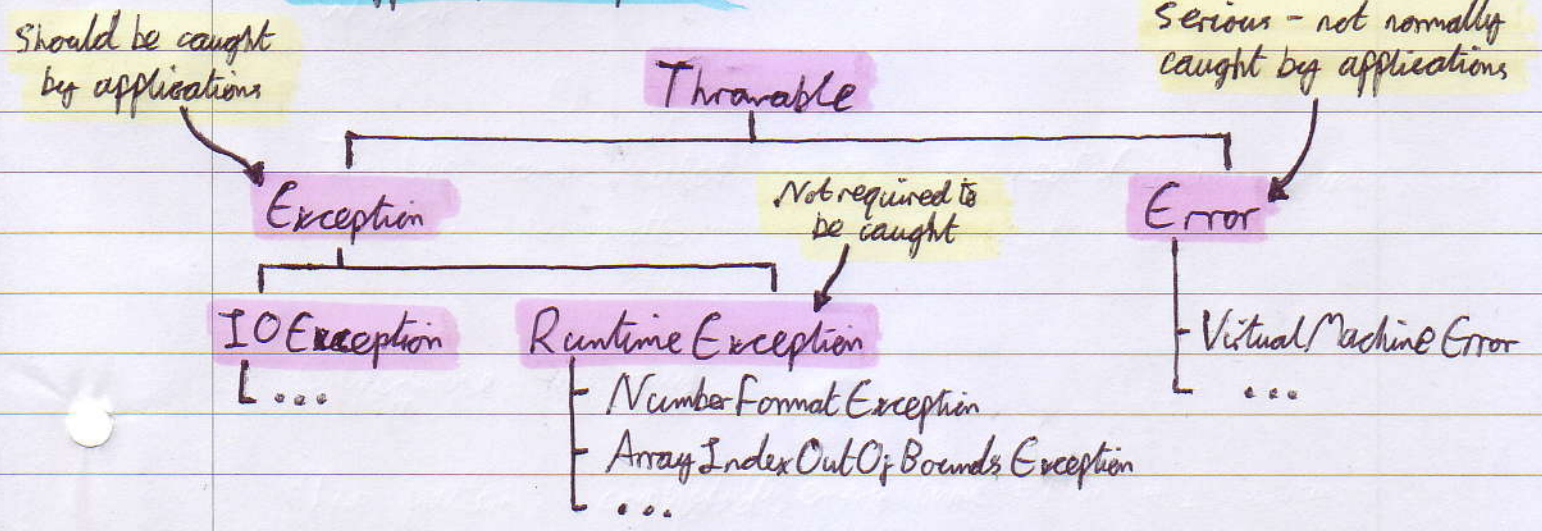
- Using Code w/ Exceptions

```
try {
  // potentially problematic code
} catch (Exception1 ex) {
  // executed if an Exception1 occurs
} catch (Exception2 ex) {
  // executed if an Exception2 occurs
} finally {
  // executed whether or not an exception occurred
}
```

- Any exception will be caught by the first matching block, even if it is a superclass of the actual exception

- Each block is separate in terms of scope - variables created inside a block only exist in that block.

- Types of Exception



• When to Catch an Exception?

- Only catch an Exception you can deal with!
  - Sometimes the application can try again with a different method / data
  - Rarely, an exception can be ignored
- Hand on unhandled exceptions

• Handing on an Exception

- A method can declare that it may throw a certain exception:

```
public static double parseString (String in) throws SomeException {
  ...
}
```

- Callers of parseString are responsible for handling an exception it may throw.

- If an exception occurs in parseString, the method is exited immediately.