Object-Oriented Programming

Types, Objects & Classes

Declarative vs Imperative

Declarative languages specify what should be done but not necessarily how it should be done. In a functional language, you specify what you want to happen by providing an example of how it can be achieved. For example, the ML compiler can do exactly that or something equivalent (it can do something else but it must give the same output or result for a given output). For example, in SQL, you specify what you want to achieve, but not how it should be done.

Imperative languages specify exactly how something should be done. An imperative compiler acts very robotically – it does exactly what you tell it to and you can easily map your code to what happens at a machine code level.

Procedural Programming involves the use of procedures to group statements together into pieces of code that can be called in order to manipulate the state.

Java as a purely Procedural Programming Language:

- Since Java is designed as an Object-Oriented Language, in order to force it to act procedurally requires us to do annoying things.
    - All functions have to be static
    - Placeholder classes are required
    - Lots of boilerplate code is required

Object-Oriented Programming is an extension to procedural programming where we recognise that the functions can be usefully grouped together and that it also makes sense to group the state they affect with them.

Considering the way ML is a Declarative Language

In ML, you appeared to specify how a function did something. However, you are actually specifying the desired result by giving an example of how it might be obtained. The compiler may or may not be the same as the example (it would generally optimise how it works) – so long as it gives the same outputs for all inputs, it is irrelevant.

Control Flow

Decision Making

Decision making is making use of:

```
if(...) { ... }
else { ... }
```

Looping

Two types of loops, for and while.

```
for (initialisation; termination; increment) { ... }
//eg for (int i = 0; i < 10; i++) { ... }
while(boolean_expression) { ... }
```

Branching

- Branching statements consist of: **return, break** and **continue**
- **return**
    - Return is used to return from a function at any point
    - However, better idea is to only have one return point in a function, therefore making it simple.
- **break**
    - It is used to jump out of a loop
    - It returns out of a single loop (therefore you may need to have multiple breaks.
- **continue**
    - Continue is used to skip the current iteration in a loop

```
for (int i = 0; i < 10; i++)
{
    if (i == 5) continue;
    System.out.println(i);
}
```

Procedures, Functions & Methods

Proper Functions

- Proper functions always return a (non-void) result
- The result is only dependent on the inputs (arguments)
- No state outside of the functions can be modified (ie no side effects)

Procedures

- Procedures have similarities to proper functions but permit side effects
- State can affect the result of a function
- (Given only the procedure name and its arguments, we cannot predict what the state of the system will be after calling it without knowing the current state of the machine).
- *Could be thought that a procedure is a proper function that takes as input the arguments you give and **all the system state***
- You can have void procedures (and no argument procedures).

Function Prototypes

- Functions are made up of a prototype and a body
    - Prototype specifies the function name, arguments and return type.
    - Body is the actual code

Mutability

In ML, all state is immutable, therefore every time you changed the value of a variable, you were in fact creating a new chunk of memory and giving it a value and dereferencing the old reference.

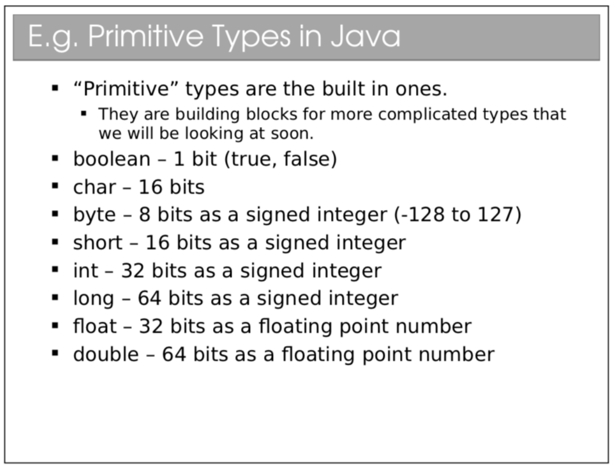
Java has variables that can be updated (mutable state), so when you change the value of a variable, rather than creating a new object and referencing the new value from that new object, Java will literally change the value of the object in memory.

Explicit Types & Type Inference

In certain declarative languages, such as ML, the compiler infers types – therefore allowing us to use polymorphism to avoid writing separate functions for integers, reals, etc.

Java (and OOP) are statically typed. This means that every variable name must be bound to a type, which is given when it is declared. For a function, you must define the return type of a function and the type of the arguments.

Java Primitives



E.g. Primitive Types in Java

- "Primitive" types are the built in ones.
  - They are building blocks for more complicated types that we will be looking at soon.
- boolean – 1 bit (true, false)
- char – 16 bits
- byte – 8 bits as a signed integer (-128 to 127)
- short – 16 bits as a signed integer
- int – 32 bits as a signed integer
- long – 64 bits as a signed integer
- float – 32 bits as a floating point number
- double – 64 bits as a floating point number

Similar to C and C++, but apart from for characters – in C a char is 1 byte (ASCII) but in Java, a char is 2 bytes (Unicode). Also, bytes are signed!

Polymorphism and Overloading

Since Java demands that all types are explicit, polymorphism becomes harder, therefore we make use of procedure overloading. In this way, we can have multiple functions with the same function name but different arguments. (They can have either the same or different return types). When you call the function, the compiler selects which procedure is the right procedure to run.

Classes and Objects

In ML, you defined your own datatypes, in order to utilise systems better. In OOP, this is the crux, utilising everything as an object, with classes being the way of defining things. However, it goes further than simply grouping of variables (as it is in ML), by writing procedures that manipulate the state (variables) of the object. **OOP classes glue together the state and the behaviour to create a complete unit of the type.**

State	**Behaviour**
Fields	Functions
Instance Variables	Methods
Properties	Procedures
Variables	
Members	

NB: When you have a procedure inside a class, it is called a method!

Instantiation classes: Objects

Since a class is a grouping of state and functions, this can be instantiated and we can create an object defined by that class, thus:

```
MyClass m = new MyClass();
MyClass m2 = new MyClass();
```

For reference: **Classes define what properties and procedures every object of the type should have, while each object is a specific implementation with particular values.**

Programs are therefore made out of lots and lots of Objects which we manipulate.

Defining Classes:

- **Constructors**
    - **a.** To define a class we need to declare the state and define the methods it contains. However, it also needs to define what should happen when the object gets constructed. When you call the keyword new …(), the …() is actual a function call, calling the constructor method of the class.
    - **b.** A constructor has the same name as the class and has no return type (already has the *new* type, which returns a reference to the new object).
    - **c.** However, there can be multiple constructors with different arguments.
    - **d.** There is also a default constructor – when you provide no constructor, the default constructor is created which is empty, simply calling super() if there is a parent class.
    - **e.** *In C++, you can instantiate an object in two different ways.*
        - **i.** *MyClass x = new MyClass()*
        - **ii.** *MyClass *y = new MyClass()*
    - **f.** *The difference is subtle. X is associated with an object – when x goes out of scope, so does the object.*
    - **g.** *Y is actually a pointer to a newly created object, so when y goes out of scope,*
- **Static**
    - **a.** If there is state which is more logically associated with a class than an object, it is made to be static – therefore it is only created once in the program's execution despite being declared as part of a class and therefore being a part of every object of that class.
    - **b.** A static variable is only instantiated once per class not per object. **You don't even need to create an object to access a static variable, simply MyClass.MyVariable would work.**
    - **c.** Methods can also be static. **In this case they must not use anything other than local or static variables, ie cannot use anything which is instance-specific.**
    - **d.** **Why use static methods?**
        - **i.** Easier to debug – only depends on static state
        - **ii.** It is self documenting
        - **iii.** Can call methods without requiring to have an object of that class.t
        - **iv.** Compiler can produce more efficient code since no specific object involved.

Designing Classes

Identifying Classes

We always aim to split things into coherent classes, each with a specific item in it, each embodying a specific well-defined concept.

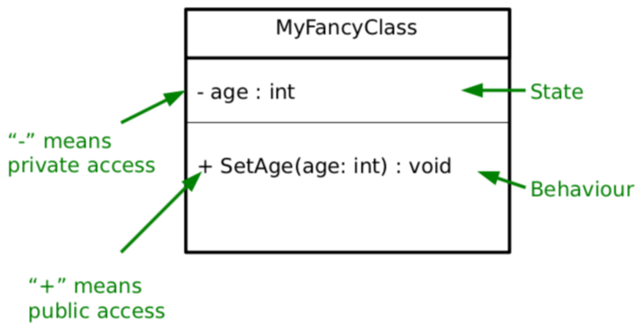
We want it to be a **grouping of conceptually related state and behaviour.**

Often describe the problem in words – the nouns refer to the state and the verbs refer to the behaviour.

Generally, classes follow directly from the problem – it's more of an art than a science. However, usually straightforward to develop sensible classes and then keep on refining them until we have something better.

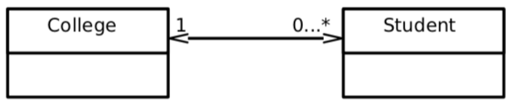
UML

UML is a method of representing a class graphically; it is used to describe a piece of software independent of the programming language that is used to create the software.

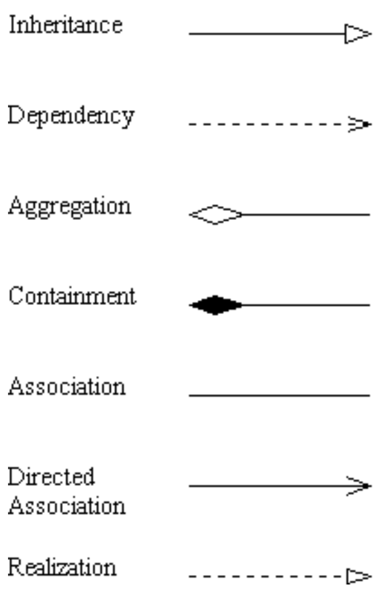


Association

An association describes a relationship between two entities, a 'has a' relationship. You also describe the number of items in the relationship hence:



- Arrow going left to right says "a College has zero or more students"
- Arrow going right to left says "a Student has exactly 1 College"
- What it means in real terms is that the College class will contain a variable that somehow links to a set of Student objects, and a Student will have a variable that references a College object.



N.B. If you have a relationship with a 0 on one side, you can simply leave out the arrowhead instead of writing an arrowhead and writing 0.

OOP Concepts

Modularity

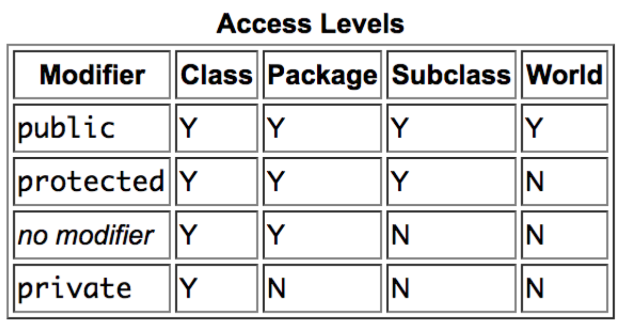
Modularity is breaking complex problems into more tractable sub-problems. Each class represents a sub-unit of code that can be **developed, tested and updated independently** from the rest of the code. This allows two classes (that achieve the same thing but working in different ways) to be easily swapped. This also means that a class can be easily lifted and used in other programs.

Code Re-use

Code re-use through modularity allows us to take objects from one piece of code and put it into other programs. So, once you've developed and tested a class that embodies everything about an item, you can easily use it in lots of other programs with minimal effort. Java also has the method of using **packages** to group together classes which are conceptually linked together – uk.ac.cam.aa2001…

Encapsulation and Information Hiding

The idea that a class should expose a clean interface that allows full interaction with the class without exposing anything about its internal state or how it manipulates it. We do this be setting the access modifiers such that the scope of variables and methods are such that only things which are required to see something can see it.



Access Levels

Modifier	Class	Package	Subclass	World
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N

Coupling refers to how much one class depends on one another – high coupling is bad since it means changing one class will require you to fix up lots of others. **Cohesion** is a measure of how strongly related everything in a class is – high cohesion is good, Encapsulation helps to minimise coupling and maximise cohesion.

Python Access Modifiers: Python has no access modifiers, however there is a convention that any function starting with an underscore should be treated as private – ie you shouldn't touch them directly.

Immutability

In concurrency, it is often helpful to make things immutable:

1. We know that nothing can change a particular chunk of memory – easier to **construct, test and use.**
    a. It means that we can happily share it between threads without worry of contention issues.
2. It also has a tendency to make code less ambiguous and more readable.
3. It is however more efficient to manipulate previously allocated memory chunks rather than allocate new chunks.
4. Also allows lazy instantiation

In Java, we can make a class immutable in the following way:

1. Make all state private
2. Consider making state final (this just tells the compiler that the value never changes once set).
3. Make sure no method tries to change internal state.

It is generally good practise to make a class immutable unless there is a good reason to make it mutable. And if it cannot be immutable, the mutability should be minimised as much as possible.

Parametrization

Parametrization allows us to use a type of polymorphism using Generics. Classes are defined with placeholders <T> and we fill them in when declaring an object, thus:

```
LinkedList<Integer> = new LinkedList<Integer>();
```

It is important to note that we can’t use primitive types with generics, instead we must use an object. All primitives have an object type associated with them which can be used for this purpose. The classes also often offer useful methods, such as parse_int for Integer.

N.B. You can also have parametrized types as parameters, for example, as: LinkedList<Vector3D<Integer>>

Pointers, References and Memory

Pointers and References

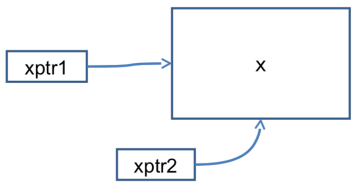
Compiler stores a mapping from a variable name to a specific memory address, along with the type so it knows how to interpret the memory (knows how long it should be and how to deal with it).

Some variables however simply store memory addresses, these are pointers or references. Manipulating the memory directly allows us to write fast, efficient code, but also exposes us to greater risks, since the program could crash.

Pointer

A pointer is just the memory address of the first memory slot used by the variable. The type of the pointer object tells us how many memory-slots the object takes up. Therefore, they cannot be tested for validity, as you can alter the memory location stored within the pointer easily and go and search other parts of the memory in other areas. Additionally, we can arbitrarily modify them either accidentally or intentionally and this can lead to all sorts of problems.

```
int x = 72;
int *xptr1 = &x;
int *xptr2 = xptr1;
```

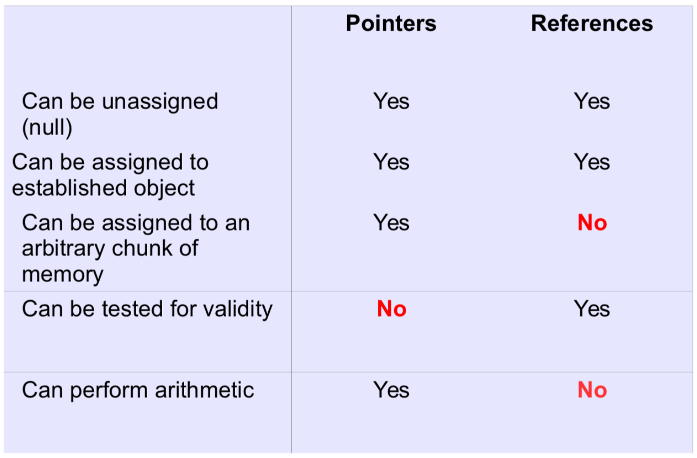


Representing Strings

We represent strings by storing a pointer to the first character, storing the next characters in subsequent slots and finish with a special character (the NULL of terminating character). It can therefore be considered as an array of characters in memory, with the string pointer pointing to the first item in the array.

References

References are aliases for other objects – they redirect to the real object. They generally use pointers in order to implement them. A reference is the same as a pointer except that the compiler restricts operations that would violate the properties of references.



	Pointers	References
Can be unassigned (null)	Yes	Yes
Can be assigned to established object	Yes	Yes
Can be assigned to an arbitrary chunk of memory	Yes	**No**
Can be tested for validity	**No**	Yes
Can perform arithmetic	Yes	**No**

Pointers are rarely available in most high-level languages, with C and C++ being ones of the few that offer pointers.

Using References in Java

- Declaring unassigned

```
SomeClass ref = null;  // explicit

SomeClass ref2;  // implicit
```

- Defining/assigning

```
// Assign
SomeClass ref = new ClassRef();

// Reassign to alias something else
ref = new ClassRef();

// Reference the same thing as another reference
SomeClass ref2 = ref;
```

Java has two classes of types: *primitive* and *reference*. A primitive type is a built-in type. Everything else is a reference type, including arrays and objects.

Call Stack

Whenever a procedure is called (**when a function is called from another, this is called nesting**), we jump to the machine code for the procedure, execute it and then jump back to where it was before and continue on. This means that, before it jumps to the procedure code, it must save where it is.

We do this using a call stack, making use of a stack. Here the items in the stack are:

1. The function parameters
2. Local variables the function creates
3. A return address that tells the CPU where to jump to when the function is done

When we finish a procedure, we delete the associated stack frame and continue executing from the return address it saved.

It is important to note that after completing a recursive function or nested functions, you have to return all the way down the stack frame, which takes time (Java is not optimised for recursion). Additionally, while tail-recursive functions are better in ML, they're only better if the compiler knows it can optimise them to use O(1) space. Java compilers don't so, so normal iteration is simply better.

Heap
The heap is a large pool of free memory, which is available to be used, and dynamically allocated rather than statically allocated as per the stack. When we allocate the memory we need from the heap, we store a pointer to the chunk we allocated in the stack. Pointers are of known size, so won't even increase. For example, if we want to resize our array, we create a new, bigger array, we copy the contents across and update the pointer within the stack. The reference for the array is updated to refer to the array.

In fact, the heap gets fragmented, as we create and delete stuff, we leave holes in memory. Occasionally we have to spend time compacting the holes, so it can be used more efficiently.

Pass-by-value vs Pass-by-reference
Pass-by-value is when an object is copied into a new value in the stack when it is passed as an argument to a procedure, whereas pass-by-reference when a reference is created to the object and this is passed to the procedure. Here you can alter the variable and the change will prevail when you return to the original procedure.

Arguments are always passed-by-value in Java, however, you need to be careful about what this means, for the two types of things, primitives and references. In the case of primitives, a copy of the primitive is created and so when the primitive is altered in the procedure, it is irrelevant to the original procedure. However, in the case of objects passed as arguments to the procedure, the reference is what is used to define the object. Therefore, the reference is passed to the procedure (simply a memory address), where an identical copy of the reference is added to the stack frame.

Therefore, when the object is altered in the procedure, the value of the changes is maintained when we return to the original procedure.

In C, you can define whether an argument is passed by value or by reference, by putting an ampersand in front of the argument. While powerful, this has the potential to lead to silly errors making programs not work.

Inheritance
Inheritance is an extremely powerful concept that is used extensively in good OOP. It is an 'is-a' relationship. It means that all state (non-private) and behaviours are taken by the subclass.

If a class B extends another class A:

- A is the superclass of B
- A is the parent of B
- A is the base class of B

- B is the child of A
- B derives from A
- B extends A
- B inherits from A
- B subclasses A

Casting
Casting creates a new reference with a different type and points it to the same chunk of memory as the object is in. Everything we need will be in the chunk if we cast to a parent class (plus extra things).

If we try to cast to a child class, there won't be all the necessary info in the memory so it will fail. However, the compiler will still be fine with the case, just returning a runtime error if anything goes wrong.

N.B. Casting primitive numeric types '(int)2.0' for example is different, since a new variable of the primitive type is created and assigned the relevant value.

Shadowing
When using subclasses and superclasses with state, you can do something called shadowing when you create a variable with the same name as one which exists in a subclass. You can deal with both variables separately, using the 'super.x' and 'this.x' keywords (when referring to variable x).

Overriding
When a method is written and has the same name as a method in a parent class, there are a couple of things that could happen. It could be treated the same as for fields and the base method could be shadowed and still be accessible. *This is the default in C++, but not for Java.* In Java, instead the parent class method is overridden, that is it is no longer accessible.

You should generally write @Override before the method that is overriding a parent class method, for two reasons. It makes it clear to a programmer that a method has been overridden, also allowing the compiler to check that a method has been overridden, checking for a misspelling or things like that.

Abstract Methods and Classes
An abstract method is used in a base class to force a class that extends it to implement a method, but you don't know what the implementation will be in the parent class.

Therefore an abstract method can be used for this – it only contains the function definition (the **method prototype** or **method stub)** of the function, rather than having the implementation of the function at all.

It works in a contractual way, any non-abstract class that inherits from this class must have that method implemented.

If a class has an abstract method, it must also be abstract. This means that the class is not able to be instantiated, therefore you can't make an object from it. This is useful when we have an abstract concept that is used a base class – for example a car (when we then define models of cars are then defined); we don't want to be able therefore to just make a generic car object.

A class is shown as being abstract in UML by having the class or method in italics.

Polymorphism

There is the general problem of when we refer to an object via a parent type and both types implement a particular method, which method should it run, such as in this:

```
Student s = new Student(); // Person and Student both have a dance method
Person p = (Person)s;
p.dance();
```

Static Polymorphism

In static polymorphism, we decide which method to run at compile-time (this is also called **early binding**). Since we don't know what the true type of the object will be, we just run the parent method. Therefore, if there are any type errors, they appear at compile-time.

Dynamic Polymorphism

In dynamic polymorphism, the method is run in the child. Therefore, it must be done at run-time since that is when we know the child's type. Therefore, any errors will cause run-time faults… (the program will entirely crash). This form of polymorphism is OOP-specific and is called **sub-type or ad-hoc polymorphism.** Because it must check types at runtime (**late binding**), there is a performance overhead associated with dynamic polymorphism. However, it gives us more flexibility and makes code more readable.

Implementations of Polymorphism

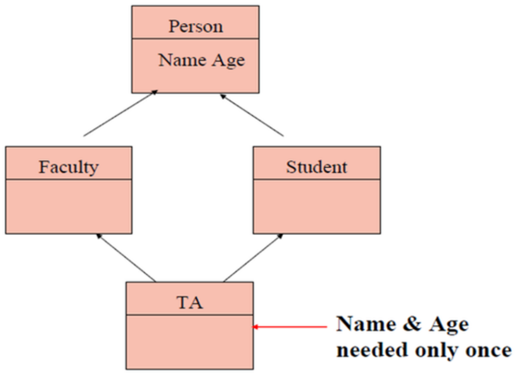
- Java
  - All methods are dynamic polymorphic.
- Python
  - All methods are dynamic polymorphic.
- C++
  - Only functions marked *virtual* are dynamic polymorphic

Though some people would consider Java's use of the word final as a way of using static polymorphism, in fact it just means the method cannot be overridden in a subclass, which is entirely different. The compiler isn't really choosing between multiple implementations but rather enforcing that there can only be one implementation.

Multiple Inheritance and Interfaces

The theory of multiple inheritance is that we can create a class which is made of up multiple things, for example DrawableFish is a DrawableEntity and a Fish.

If we multiple inherit, we capture the concept we want, but there is a big problem of defining which methods we inherit in the case of both things we are inheriting from having methods of the same name. This always occurs in the case of the 'dreaded diamond':



In C++, we can deal with this, by simply referring to the class's method we are calling when we call a method.

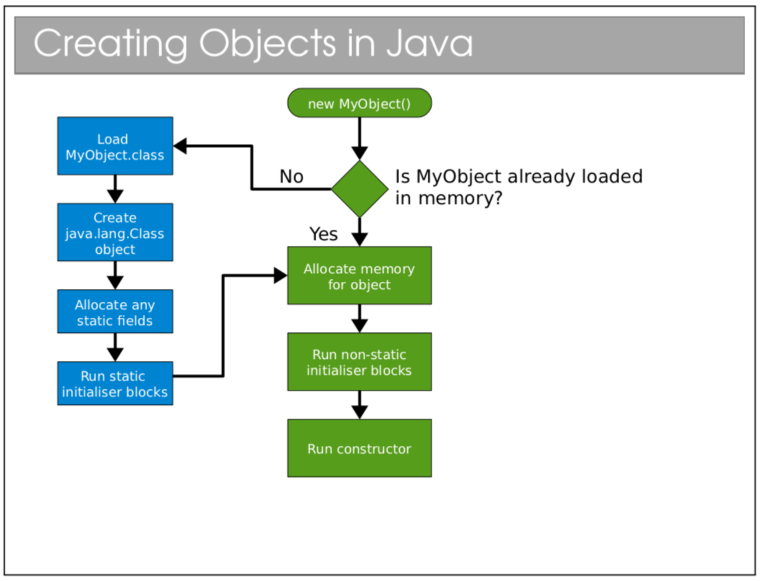
In Java, however, we fix it with abstraction: It is trivial that the problem goes away if one of the methods is completely abstract. **We call these totally abstract methods interfaces.** A class can **implement as many interfaces as desired (but can only inherit from one class).**

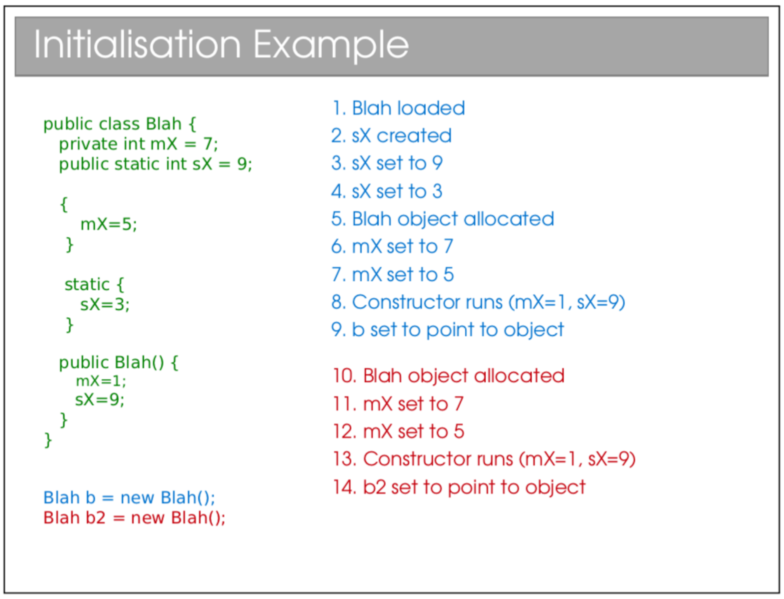
A Java interface is a class that has:
1. No state whatsoever
2. All methods abstract

Interfaces are represented in UML class diagrams by a preceding <<interface>> label and inheritance occurs using the 'implements' keyword. Interfaces are considered so important as to be the third reference type, in addition to classes and arrays.

Object Lifecycle

Process of creating an object


Creating Objects in Java
new MyObject()
Is MyObject already loaded in memory?
No
Yes
Load MyObject.class
Create java.lang.Class object
Allocate any static fields
Run static initialiser blocks
Allocate memory for object
Run non-static initialiser blocks
Run constructor



Initialisation Example

```
public class Blah {
  private int mX = 7;
  public static int sX = 9;

   {
     mX=5;
   }

   static {
     sX=3;
   }

  public Blah() {
     mX=1;
     sX=9;
   }
}

Blah b = new Blah();
Blah b2 = new Blah();
```

1. Blah loaded
2. sX created
3. sX set to 9
4. sX set to 3
5. Blah object allocated
6. mX set to 7
7. mX set to 5
8. Constructor runs (mX=1, sX=9)
9. b set to point to object

10. Blah object allocated
11. mX set to 7
12. mX set to 5
13. Constructor runs (mX=1, sX=9)
14. b2 set to point to object

When you construct an object of a type with parent classes, we call the constructors of all the parents in sequence – moving up the classes in hierarchy and calling the constructors. In reality, Java adds the line super() for all classes as the first line of the constructor. However, if you need to add an argument when calling the constructor of the super class, then you must make the call yourself.

In a language like C++, which supports multiple inheritance, the process of which order to call the super constructors must be defined by you.

Destruction

Deterministic Destruction: Objects are created, used and eventually destroyed (as expected). They are auto-deleted when they go out of scope or when they are manually deleted.

Destructor: Most OO languages have a notion of a destructor which is run when the object gets destroyed – allowing us to release any resources that we might have created especially for the object.

Non-Deterministic Destruction

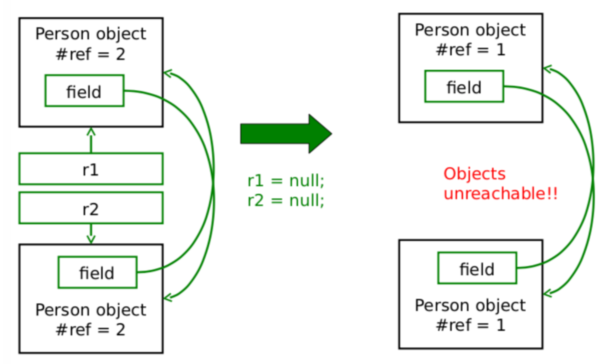
Since humans are really, truly terrible at keeping track of what needs to get deleted and when (we either forget to delete – **memory leak** – or delete multiple times – **crash**) we leave it to the system to decide when to delete.

Garbage Collection (Java System)
This system figures out when to delete an object and does it for us. In reality, it needs to be cautious and deletes later than it could. This leads to us not being able to predict exactly when something will be deleted,

Finalisers: Conventional destructors don't make sense in non-deterministic systems as we don't know when they will be run (if at all). Instead in garbage collection, we use finalisers, which are effectively the same thing.

The garbage collection is a separate process that is constantly monitoring your program, looking for things to delete – running this is not free and it may take time and give noticeable pauses to the program. The garbage collection works with a number of algorithms, including reference counting and tracing.

Reference Counting: This is the original method. It keeps track of how many references point to a given object. If there are none, the programmer can't access that object ever again, so it can be deleted. (This means every object needs more memory to store the reference count). Also, circular references are painful to deal with – since they can often continue to exist with reference count of greater than 0.



Tracing: Starts with a list of all references you can get to, following each reference recursively, marking each object. Then delete all objects that were not marked.

Java Collections

Since Java isn't just a programming language, but also a platform, shipping with a large class library, containing a whole range of things including:

- Complex Data Structures and Algorithms
- Networking
- GUIs
- I/O
- Security and Encryption

Because Java code runs on a virtual machine, the underlying code is abstracted away. For C++ on the other hand, while the backend can be the same, things like I/O and graphical interfaces are different for each platform (Windows, OSX, Linux).

Collections and Generics

Collections framework is a set of interfaces and classes that handles groupings of objects and allow us to implement various algorithms invisibly to the user.

1. **Sets: <<interface>> Set**
    - **a.** A collection of elements with no duplicates that represents the mathematical notion of a set.
    - **b.** TreeSet: Objects stored in order
    - **c.** HashSet: Objects in unpredictable order, but fast to operate on.
    - **d.** Eg.

```
TreeSet<Integer> ts = new TreeSet<Integer>();
ts.add(15);
ts.add(12);
ts.contains(7); // false
ts.contains(12); // true
ts.first; // 12 since it is sorted
```

2. **Lists: <<interface>> List**
    a. An ordered collection of elements that may contain duplicates
    b. LinkedList: linked list of elements
    c. ArrayList: array of elements (efficient access)
    d. Vector: Legacy, as ArrayList but threadsafe
    e. eg.

```
LinkedList<Double> ll = new LinkedList<Double>();
ll.add(1.0);
ll.add(0.5);
ll.add(3.7);
ll.add(0.5);
ll.get(1); // get the second element (0 indexed) (== 0.5)
```

3. **Queues: <<interface>> Queue**
    a. An ordered collection of elements that may contain duplicates and supports removal of elements from the head of the queue.
    b. Offer() to add to back and poll() to take from the front.
    c. LinkedList: supports the necessary functionality
    d. PriorityQueue: adds a notion of priority to the queue so more important things go to the top.
4. **Maps: <<interface>> Map**
    a. Like dictionaries in ML
    b. Maps keys to values
    c. Keys must be unique
    d. Values can be duplicated and null
    e. Can be:
        i. TreeMap: keys kept in order
        ii. HashMap: keys not in order, efficient.

```
TreeMap<String, Integer> tm = new TreeMap<String, Integer>();
tm.put("A", 1);
tm.put("B", 2);
tm.get("A"); // 1
tm.get("C"); // null
tm.contains("G"); // false
```

5. **Iteration using foreach**

```
LinkedList<Integer> list = new LinkedList<Integer>();
...
for(Integer i : list) { ... }
```

6. **Iterators**
    a. Still works if the loop changes the structure of the list through which is being iterated.

```
Iterator<Integer> it = list.iterator();
while(it.hasNext()) { Integer i = it.next(); }
for (; it.hasNext(); ) {
    Integer i = it.next();
    it.remove(); // It is safe to modify the structure of the list
}
```

Object Comparisons

Imposing ordering on data collections:

If you use a TreeSet, TreeMap, etc, the ordering is automatic. For other collections, you may need to explicitly sort (Collections.sort(x)). While the ordering is obvious for numeric types, it is unobvious how to do this for custom objects.

Object Equality

Reference Equality (r1 == r2, r1 != r2): These test whether the two references point at the same chunk of memory.

Value Equality: Uses the equals() method in Object. The default implementation just uses reference equality so we have to override the method. You should check whether a class overrides equals() to do anything other than just use '==' (reference equality).

```
public EqualsTest {
   public int x = 8;

   @Override
   public boolean equals(Object o) {
      EqualsTest e = (EqualsTest)o;
      return (this.x==e.x);
   }

   public static void main(String args[]) {
      EqualsTest t1 = new EqualsTest();
      EqualsTest t2 = new EqualsTest();
      System.out.println(t1==t2);
      System.out.println(t1.equals(t2));
   }
}
```

N.B. Object also gives classes hashCode(). It assumes that if equals(a,b) returns true then a.hashCode() is the same as b.hashCode(). Therefore, you should override hashCode() at the same time as equals().

Comparator and Comparable

In order to sort objects using built in classes, you need to write something that allows two objects to be ordered. Often, our classes have a natural ordering.

Comparable<T> Interface

You must implement 'int compareTo(T obj);' and return an integer:

- r < 0 – object is less than obj
- r == 0 – object equal to obj
- r > 0 – object greater than obj

However, if you want to sort in a number of ways, there are other ways you can do the ordering, using a Comparator.

Comparator<T> Interface

You must implement 'int compareTo(T obj1, T obj2)'

Eg.

```
public class Person  implements Comparable<Person> {
    private String mSurname;
    private int mAge;
    public int compareTo(Person p) {
        return mSurname.compareTo(p.mSurname);
    }
}

public class AgeComparator implements Comparator<Person> {
   public int compare(Person p1, Person p2) {
      return (p1.mAge-p2.mAge);
   }
}

...
ArrayList<Person> plist = ...;
...
Collections.sort(plist);   // sorts by surname
Collections.sort(plist, new AgeComparator());   // sorts by age
```

Some languages (such as C++) allow you to overload the comparison operators (however, Java does not have this):

```
class Person {
    public:
     int mAge;
     bool operator== (Person &p) (
         return (p.mAge == mAge)
     );
}
Person a, b;
b == a; // tests for equality
```

Error Handling

Types of errors

Syntactic Errors: They are generally reasonably easy to spot because you get a nice error from the compiler.

Logical Errors: These are more problematic because comprehensive testing (checking the output for every possible input and system state) is usually infeasible.

External Errors: This occurs for processes code relies upon but we don't control fails – such as a failing hard disk or overheating CPU causing the parts to not behave as expected.

In order to deal with these errors, you should attempt to do two things. The first thing would be to attempt to minimise the number of bugs which happen and the second is that you should anticipate errors anyway and you should use techniques to handle the errors.

Minimising bugs

1. **Modular (Unit) Testing**

   a. OOP encourages you to develop uncoupled chunks of code in classes. Each class should be testable independent of the others. It is much easier to comprehensively test lots of small bits of code and then put them together.
2. **Using Assertions**
   a. Assertions are a form of error checking designed for debugging only.
   b. They are a simple statement that evaluates a Boolean: If it is true nothing happens, it if is false, the program ends.
   c. Assert(Boolean_condition) : "Some error message"
   d. They aren't for production code – they are simply there to help you check the logic of code is correct – they should be switched off when code gets released.
   e. Very useful to put at the end of an algorithm to check it is functioning correctly. (Postconditions). Also, useful for preconditions to check at the start of an algorithm…
      i. But, shouldn't use assertions to check for public preconditions
3. **Defensive Programming Styles**
   a. Learn useful habits for each language that can reduce errors
      i.

In C++, if(exp) is true if x > 0

      ii. To fix issue in forgetting a second '=' sign in a Boolean expression, you can always do the other side first ie instead of 'x==5' (which can easily be 'x=5' by mistake) use 5 == x (since 5 = x will return an error).
4. **Pair Programming**
   a. Programming in pairs insofar as to make one person write code while the other person watches over their shoulder, looking for errors or bugs. (Writer and watcher switch roles regularly).

Dealing with errors

Return Codes

The traditional way of handling errors is to return a value from a method which indicates success / failure / errors, with a list of such return codes for each function.

However, (1) it ignores the return value (can't return something as well), (2) we have to keep checking what the return values are meant to mean.

In order to also return a value from a function which was previously returning something, we look to use return codes which outside of the range of output of the function (ie a square root function might use -1 as a return error state since you'd never get negative numbers).

If the return type isn't something we can repurpose (like a custom class), then we can instead pass output by reference and have the function return an integer to indicate the error state, eg:

```
int func (float a, SomeCustomClass result ) {
    if (a < 0.0) return -1.0;
    else result.set(...);
    return 0;
}
```

Some people would return a null or a null object if there is an error, but this has the issue of the programmer having to check for this. If they don't and try to dereference null, they will receive an error. This is a larger issue of the programmer having to test the return value, leading to annoying code.

Deferred Error Handling

A similar idea (with the same issues) is to set some state in the system which needs to be checked for errors. C++ does this for stream:

```
ifstream file( "test.txt" );
if ( file.good() )
{
      cout << "An error occurred opening the file" << endl;
}
```

Exceptions

An exception is an object that can be thrown or raised by a method when an error occurs and caught or handles by the calling code.
When an execution is thrown, any code left to run in the try block of code is skipped and the code in the catch part is done (we test for the exception conditions in sequence in the order they are written). After code in the try and catch (irrelevant of what it is – even if it is a return) is done, the **finally** part is done (this is guaranteed to be attempted to be run). This makes finally very useful for dealing with resources that need to be closed.

In order to throw an exception (an object that has Exception as an ancestor) you effectively need to create a new version of that object, therefore doing 'throw new randomException();'

Creating Exceptions: In order to create an exception, you must simply create a class which extends Exception or RuntimeException. It is generally good form to add a detail message in the constructor but it isn't required.

```
public class DivideByZero extends Exception {}

public class ComputationFailed extends Exception {
   public ComputationFailed(String msg) {
      super(msg);
   }
}
```

RuntimeExceptions inherit from Exception but are different from all other exceptions in that they are generally unchecked. This means they are not expected to be handled and in order to fix the system you would have to change the code.

Checked Exceptions (other Exceptions) on the other hand must be handled or passed up. They are used for recoverable errors and Java requires you to declare the checked exceptions that a method throws, catching that exception when the method is called.

What errors and which exception:

1. **Machine Fault**
    a. Completely unrecoverable, machine broken

   - **b.** Error
2. **Program Fault**
   - **a.** Need to change the code
   - **b.** Extend Exception
3. **Recoverable Fault**
   - **a.** Expected faults
   - **b.** Extend RuntimeException

Advantages of Exceptions

1. Class name can be descriptive
2. Doesn't interrupt flow of code
3. Exception object can contain state that gives errors on why errors occurred.
4. Can't be ignored, only handled.

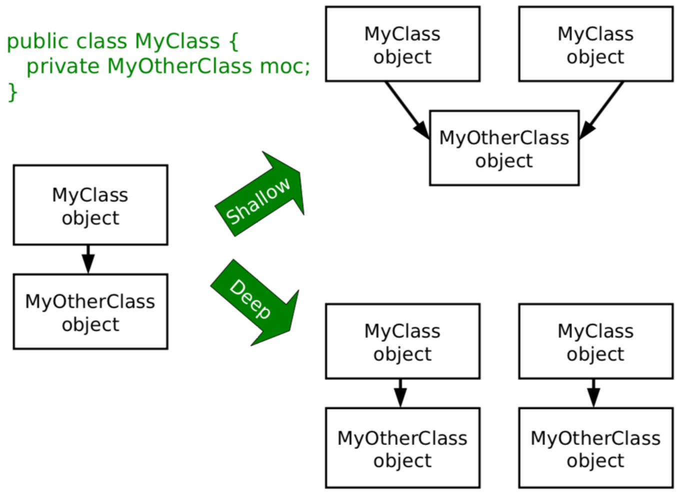
Things to never do

1. **Exceptions for Flow Control**
   - a. In some senses, throwing an exception is like a GOTO and it is tempting to use this to get out of a lot of stacks (when using recursion) quickly
   - b. However, this is not good!!
     - i. Exceptions are for exceptional circumstances only. Using it like this makes things harder to read and may prevent optimisations
   - c. It is in fact slower than just getting rid of stacks. This is because exception handling is very slow – it has never been bothered to be optimised since it is only for exceptional circumstances.
2. **Blank Exception Handlers**
   - a. Checked handlers must be handled, and it is tempting to just put an empty catch (to be dealt with later).
   - b. This can lead to problems
   - c. At least print the stack trace (e.printStackTrace()) so there's some record of the problem being printed to the screen.
3. **Circumventing Exception Handlers**
   - a. Using: catch (Exception e) {}
   - b. Terrible idea – catches unchecked exceptions and generally terrible practise.

Copying Objects

Shallow Copy vs Deep Copy

A shallow copy is when references to objects in the item being are copied (so the copy and the original both point to the same objects), whereas in a shallow copy, the object being referenced is also copied and the new object has a reference to the new copy of the item inside it, as per this diagram:



Cloning

In cloning, we are doing a byte-for-byte copy of an object in memory. Any primitive types, will therefore be copied, while references will also be copied, but not the objects they point to. Therefore, this gets us a shallow copy.

Clone() is a function provided by the Object class, but it will return an error if you call clone on anything which doesn't implement the Cloneable interface (empty interface – **MARKER INTERFACE**).

Recipe for cloning (with deep copy)

1. Implement cloneable
2. Make clone() public
3. Call super.clone() casting to the correct item
4. Clone any references to mutable states

This could look like this:

```
public class Velocity implement Cloneable {
     ....
     public Object clone() {
        return super.clone();
     }
};

public class Vehicle implements Cloneable {
  private int age;
  private Velocity v;
  public Student(int a, float vx, float vy) {
      age=a;
      vel = new Velocity(vx,vy);
  }

  public Object clone() {
      Vehicle cloned = (Vehicle) super.clone();
      cloned.vel = (Velocity)vel.clone();
      return cloned;
  }
};
```

Cloning Arrays: Arrays have built in cloning (Array.clone()) but the contents are only cloned shallowly. Therefore, it may be necessary to create a new array and clone each item in term and assign them to the new array.

COVARIENT RETURN TYPES

Recent versions of Java allow you to override a method in a subclass and change its return type to a subclass of the original's class

Copy Constructors

You can define a copy constructor that takes in an object of the same type and manually copies the data, ie:

```
public class Vehicle {
   private int age;
   private Velocity vel;
   public Vehicle(int a, float vx, float vy) {
      age=a;
      vel = new Velocity(vx,vy);
   }
   public Vehicle(Vehicle v) {
        age=v.age;
        vel = v.vel.clone();
  }
}
```

However, it makes it hard to create a copy of a list of items of different types. When you call the creation of a new x() based on a previous y() (where y may be a child of x), only an x is produced. the new list contains only objects of the type of the parent. (Otherwise, reflection must be used to define what method to call in every different possibility – this goes against the entire point of polymorphism). T

However, other than this, it is very useful.

Copy Constructor Recipe

1. Create a constructor that that takes an object of the same type to be copied
2. Call super copy constructor
3. Copy all primitives and references
4. Deep copy any mutable states (using copy constructor or clone)

Language Evolution

Many languages start out by a programmer fixing a small issue, creating something that is suitable for a particular niche. If the language is to be widely used, then it has to evolve to incorporate new paradigms while also introducing old paradigms that were rejected but turned out to be necessary.

The challenge is maintaining backwards compatibility, for example in Java:

- Vectors were originally part of Java, choosing to make it synchronised.
- When they introduced collections, they decided that things shouldn't be synchronised – therefore they created an ArrayList which is the same as Vector

    - But is unsynchronised and more efficient.
        - *If you need a synchronised ArrayList, it can be done using ArrayLists as well.*
- But they had to retain Vector for backwards compatibility.

Generics

As previously discussed, Generics allow us to use the same class, or method with multiple different types of variables – offers a means of parametrization.

Java implements type erasure, such that the compiler checks through your code to make sure you only used a single type with a given Generics object. It then deletes all knowledge of the parameter, converting it to the old style invisibly.

```
LinkedList<Integer> ll =
   new LinkedList<Integer>();

...

for (Integer i : ll) {
   do_sthing(i);
}
```

```
LinkedList ll =
   new LinkedList();

...

for (Object i : ll) {
   do_sthing( (Integer)i );
}
```

(This explains why you can't use primitives as parameters: whatever is put there must be castable to Object)

C++ Templates Solution

C++ Compilers first generates the class definitions from the template, producing a special class for each instance you request.

```
class MyClass<T> {
  T membervar;
};
```

```
class MyClass_float {
  float membervar;
};
class MyClass_int {
  int membervar;
};
class MyClass_double {
  double membervar;
};
...
```

Java 8

Java 8 adds a lot of features.

1. **Lambda Functions**
    a. It supports anonymous functions, that is functions without a name
    b. () -> System.out.println("Hello World");
    c. (x, y) -> x + y
2. **Functions as Values**
    a. You can store functions in certain variables, such as these:

```
// No arguments
Runnable r = ()->System.out.println("It's nearly over...");
r.run();

// No arguments, non-void return
Callable<Double> pi = ()->3.141;
pi.call();

// One argument, non-void return
Function<String,Integer> f = s->s.length();
f.apply("Seriously, you can go soon")
```

3. **Method References**
    a. You can make references to methods which have already been created, such as:
    b. System.out::println can be used instead of the function s -> System.out.println(s)
    c. Therefore can then be used like a lambda function
4. **New Foreach**
    a. List<String> list = new LinkedList<>();
    b. list.add("Hello World");
    c. list.forEach(s->System.out.println(s))
    d. (This is equivalent to:)
    e. list.forEach(System.out::println)
    f. **This is effectively the map function from ML!**
5. **Sorting**
    a. Sorting functions can also be defined using lambda function to make them easier to read:
        i. Collections.sort(list, (item1, item2) -> item1.length() – item2.length())
6. **Streams**
    a. Collections can be made into streams (sequences)
    b. These can be mapped or filtered.

```
List<Integer> list = ...
list.stream().map(x->x+10).collect(Collectors.toList());
list.stream().filter(x -> (x > 5)).collect(Collectors.toList());
```

Design Patterns

A design pattern is a general reusable solution to a commonly occurring problem in software design. They are about intelligent solutions to a series of generalised problems that you may be able to apply when designing code

Why study them

1. They encourage us to identify the fundamental aims of given pieces of code.
2. They save use time and give us confidence that our solution is sensible.
3. They demonstrate the power of OOP.
4. They demonstrate that naïve solutions are bad.
5. They give us a common vocab to describe our code.
    a. When you work in a team, one quickly realises that it is important to succinctly describe what your code is trying to do.

Open-Closed Principle
'Classes should be open for extension but closed for modification'.

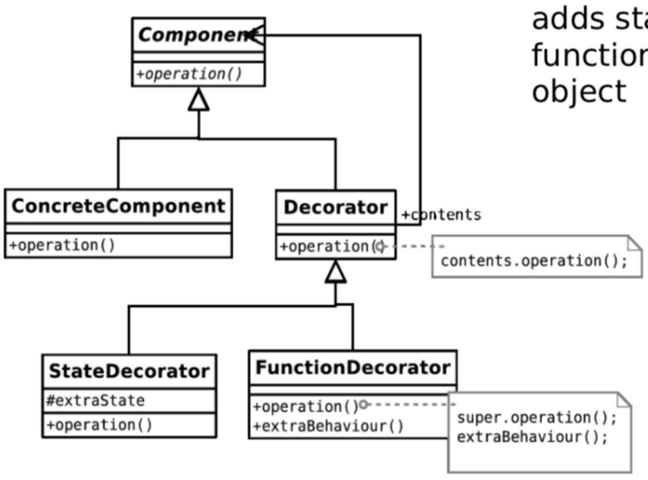
In order to alter a class, people could edit the source code. However, this would mean that they had a customised version of the library that they wouldn't be able to update when new (bug-reduced) versions appeared. A better solution is to use the library class as a base class and implement the minor changes that are desired in the custom child.

If you are writing code that others will use, you should make it easy for them to extend your classes and discourage direct editing of them.

The Decorator Pattern

Abstract Problem: How can we add state or methods at runtime.
Example Problem: How can we efficiently support gift-wrapped books in an online bookstore?

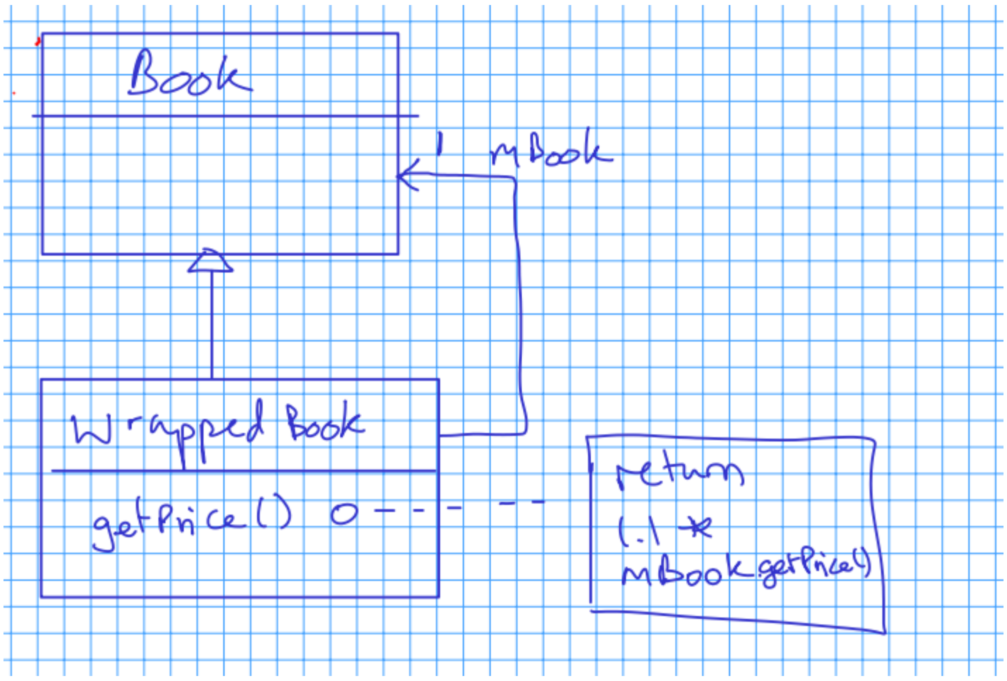
Solution 1: Add variables to the established Book class that describes whether or not the product is to be gift wrapped. It violates open-closed and is wasteful – however is easy to unwrap.
Solution 2: Extend Book to create a class for WrappedBook. It passes Open-Closed but we cant swap book -> Wrapped book (there is no object conversion).
Solution 3: (Decorator) Extend Book to create WrappedBook and also add a member reference to a Book object. Just pass through any method calls to the internal reference, intercepting any that are to do with shipping or price to amount for the extra wrapping behaviour.



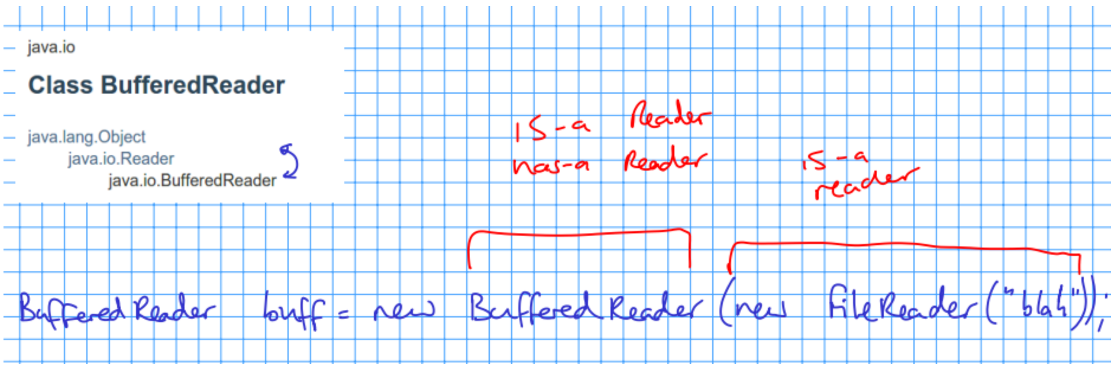
```
contents.operation();
```

```
super.operation();
extraBehaviour();
```

- The decorator pattern adds state and/or functionality to an object *dynamically*



An example of where this is used is in **Buffered Reader** where the Buffered Reader is defined by giving a new FileReader, as below:

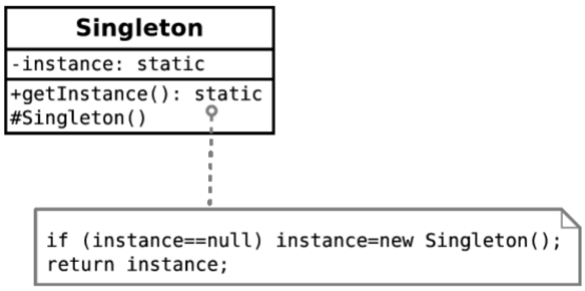


Abstract
Create a decorator which has a component and also inherits from the component. The pattern can be used to add state (variables) or functionality (methods), or both if we want.

The Singleton Pattern

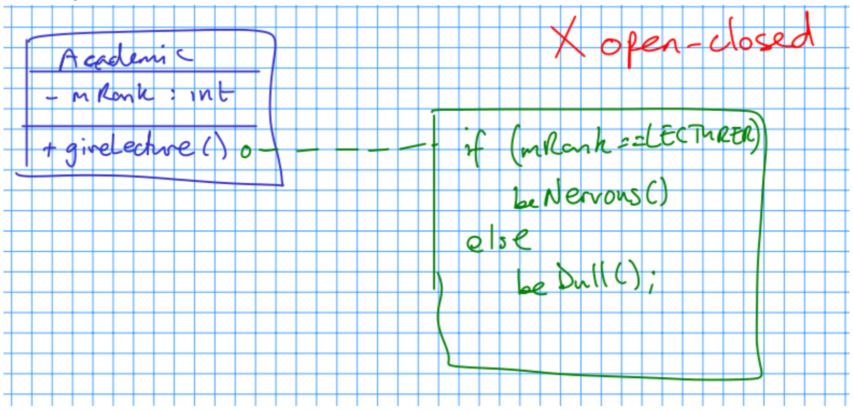
Abstract Problem: How can we ensure only one instance of an object is created by developers using our code.
Example Problem: You have a class that encapsulates accessing a database over a network. When instantiated, the object will create a connection and send the query. Unfortunately, you are only allowed one connection at a time.

Singleton Recipe
- Mark the constructor private
- Create a single static instance
- Create a static getter for the instance



The State Pattern

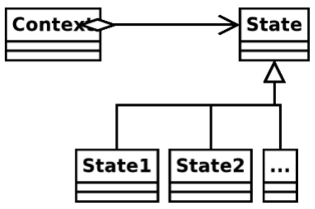
Abstract Problem: How can we let an object alter its behaviour when its internal state changes?
Example Problem: Representing academics if they progress through the ranks

1. **Solution 1:** Have an abstract academic class which acts as a base class for Lecturer, Professor, etc.
    a. Can't convert objects
    b. Can't replace objects
2. **Solution 2:** Make Academic a concrete class with a member variable that indicates rank. To get rank specific behaviour, check this variable within the relevant methods.
    a. 

3. **Solution 3:** Make Academic a concrete class that has-a academic rank as a member. Use academic rank as a base for Lecturer, Professor, etc, implementing the rank specific behaviour in each.
    a. Items can be swapped easily since there is only one reference to AcademicRank.

Abstract Methodology

- The state pattern allows an object to cleanly alter its behaviour when internal state changes

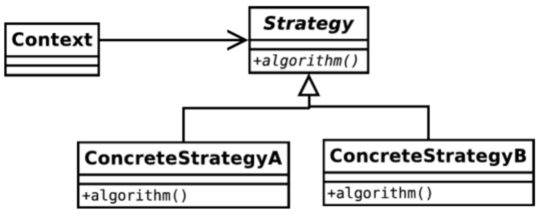


The Strategy Pattern

Abstract Problem: How can we select an algorithm implementation at runtime?
Example Problem: We have many possible change-making implementations. How do we cleanly change between them.

Solutions

1. Use a lot of if…else… statements in the getChange(…) method.
    a. Violates open-closed
2. Create an abstract Change-Finder class. Derive a new class for each of our algorithms.
    a. Definitely meets open-closed.

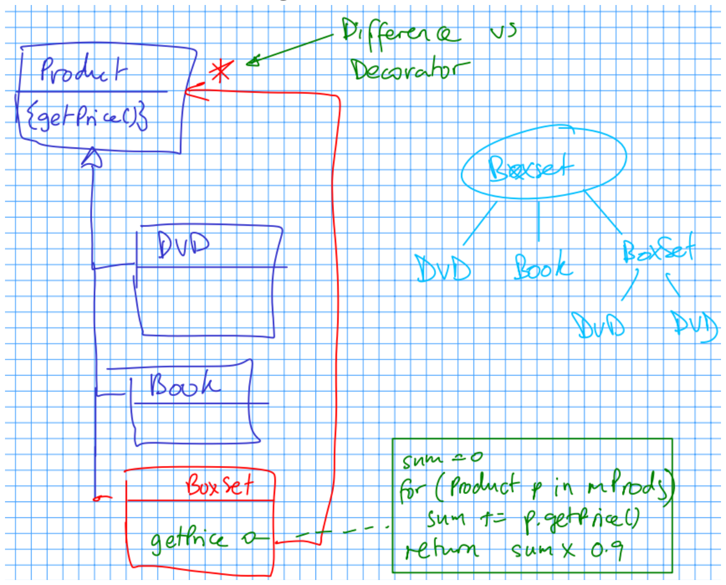


State Pattern vs Strategy Pattern (looks like the same solution but different intent)
State: Different behaviour depending on current context – hiding the classes
Strategy: Same behaviour but achieved in different way – explicit / exposed classes

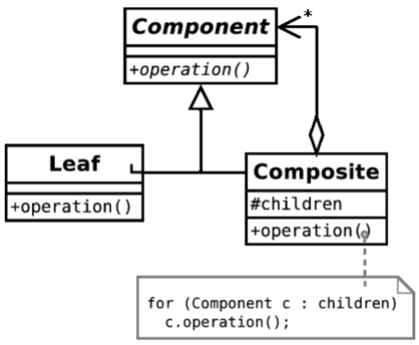
The Composite Pattern

Abstract Problem: How can we treat a group of objects as a single object?
Example Problem: Representing a DVD box-set as well as the individual films without duplicating info and with a 10% discount.

Solution is to have a BoxSet which inherits from DVD class and has a number of DVD items (it can therefore have a box set in a box set). Additionally, if you wanted to have a box set (more generally) which could have more than just DVDs in it, for example having books, the UML would look something like this:



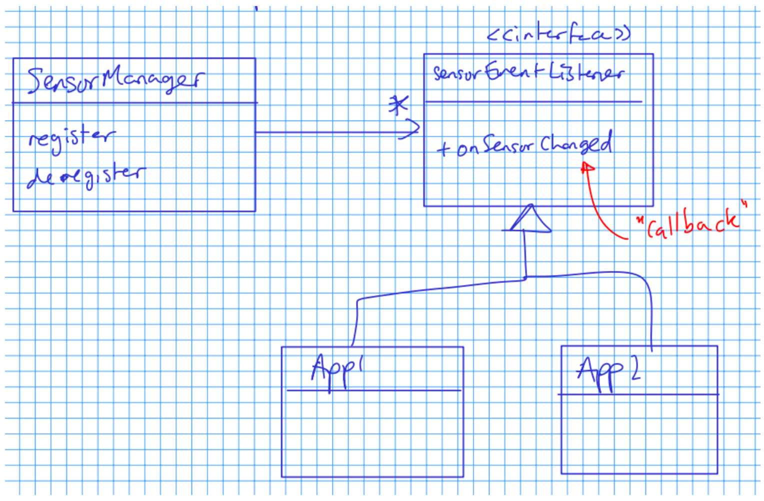
The difference in comparison to the decorator pattern is that a box set has multiple association with the product class (it can have multiple products). Additionally, the intent is different – we are not adding additional functionality to objects, but rather supporting the same functionality for groups of objects.



The Observer Pattern

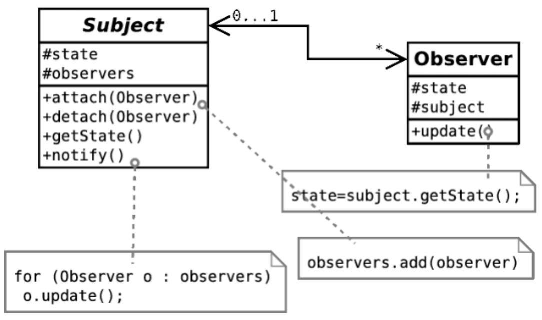
Abstract Problem: When an object changes state, how can any interested parties know?
Example Problem: How can we write phone apps that react to accelerator events?

The process of working a system like this is analogous to a magazine subscription, with you subscribing with the magazine publisher to get publish events when they are available.



In an Android smartphone, the system provides a subject in the form of a SensorManager object, which is a singleton (only one manager at a time). We can acquire the manager and register the listener. Our class must implement SensorEventListener, which forces us to specify an onSensorEvent() method. Whenever the system gets a new accelerometer reading, it cycles over all the objects that have registered with it, feeding them the new information.

▪ The observer pattern allows an object to have multiple dependents and propagates updates to the dependents automatically.



Classifying and Analysing Patterns

Patterns can be classified according to what their intent is or what they achieve:

1. **Creational Patterns**
    a. Patterns concerned with the creation of objects
    b. Singleton
2. **Structural Patterns**
    a. Patterns concerned with the composition of classes or objects
    b. Composite, Decorator
3. **Behavioural Patterns**
    a. Patterns concerned with how classes or objects interact and distribute responsibility
    b. Observer, State, Strategy

It is also important to note that solutions have been concentrated on structuring code to be more readable and maintainable, and to incorporate constraints structurally where possible. The performance of the solutions is never discussed. Many, for example, exploit runtime polymorphism which is expensive.

Java, the JVM and Bytecode

Java is known for having cross-platform abilities, which has given it strong internet credentials. It shouldn't really work (being able to send files compiled on one platform and running it on another) since machine code should be different.

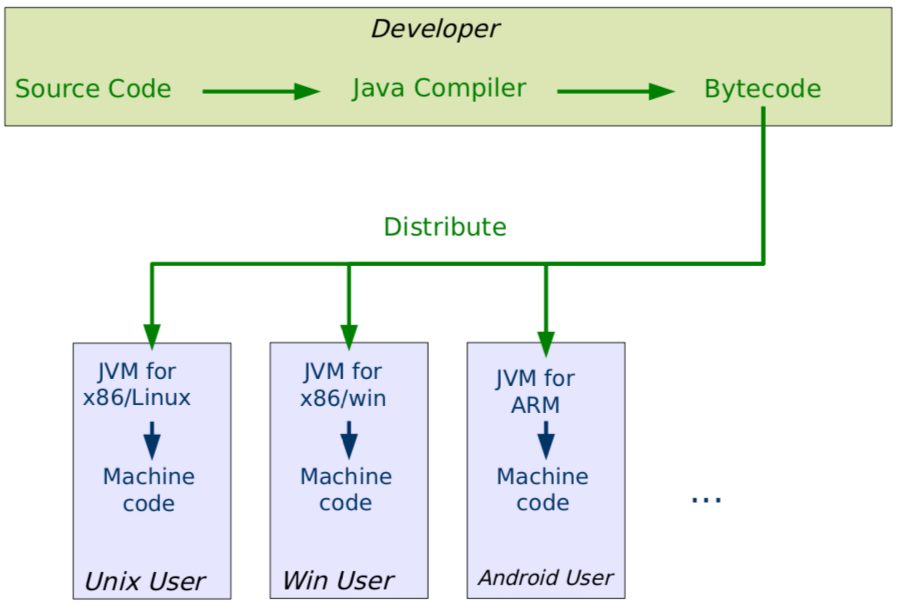
You could send source code and use an interpreter (like Javascript) but this is not very space-efficient and the source code would implicitly be there for everyone to see, which hinders commercial viability.

SUN envisioned a JVM (Java Virtual Machine). Java is compiled into machine code (called bytecode) for that imaginary machine. The bytecode is then distributed. In order to use the bytecode, the user must have a JVM which converts the correct machine code for the local computer.

Advantages

1. Since Bytecode is compiled, it is not easy to reverse-engineer.

2. The JVM ships with lots of libraries which makes the bytecode very small.
3. The toughest part of the compile (from source to bytecode – human readable to computer readable) is done by the compiler, leaving the JVM to complete the easier, faster job.
4. However, there is still a performance hit compared to fully compiled (native) code.


Developer
Source Code
Java Compiler
Bytecode
Distribute
JVM for x86/Linux
Machine code
Unix User
JVM for x86/win
Machine code
Win User
JVM for ARM
Machine code
Android User
...