Part IA Object Oriented Programming Notes

1 Polymorphism

1.1 Overloading

- Multiple function with the same name but different arguments

1.2 Abstract Classes

- An class containing an abstract method must be made abstract
- To be instantiated an abstract class must be inherited and all abstract methods implemented

1.3 Shadowing

- All non-private state is inherited, meaning that you can have multiple variables of the same name

```
1    class A { public int x; }
2
3    class B extends A { public int x; }
4
5    class C extends B { public int x; }
```

- For example in the case of C there will be 3 integers allocated

1.4 Casting

- Java will always round down when loosing precision
- Can always cast up to a parent class
- Can only cast to child class if the underlying object is really that child
- This will throw a run-time error, not a compiler error

1.5 Static Polymorphism

- Decided at compile time
- Since we don't know what the true type of the object will be, we just run the parent method
- Type error give compiler errors

1.6 Dynamic Polymorphism

- **All methods are dynamic polymorphic**
- Run the method of the child
- Must be done at run-time sine that is when we know the child's type
- Type error cause run-time errors
- Slower than if only static polymorphism is used

1.7 Covariant Return Types

- When overriding a function, Java will allow the return type to be a subclass of the parent's implementation's return type

- For example, the following is allowed:

```
1   public class A {
2       public Object work(Object o) {...}
3   }
4
5   public class B extends A {
6       @Overrude
7       public Person work(Object o) {...}
8   }
```

- This is not allowed in arguments (Covariant Parameter Types)

1.8 Generics and Subtyping

- "In general, if Foo is a subtype (subclass or subinterface) of Bar, and G is some generic type declaration, it is not the case that G<Foo>is a subtype of G<Bar>."

- The following will fail (compliation error):

```
1   List<Person> pList = new LinkedList<>();
2   List<Animal> aList = (List<Animal>)pList;
```

2 Garbage Collection

2.1 Finalises

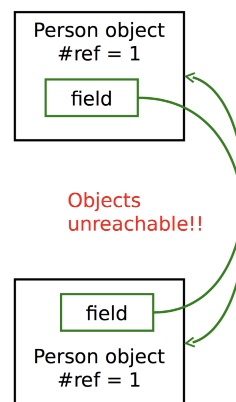
- Methods that will be run when garbage collector deletes an object

2.2 Reference Counting

- Can write your own garbage collector

- Holds a count for the number of references that point to an object

- Falls apart, eg.

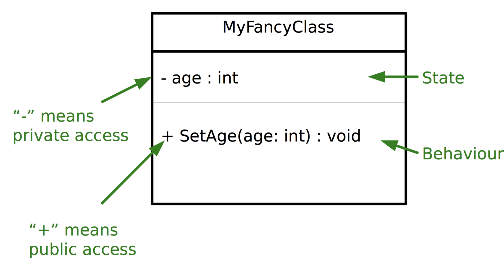


2.3 Tracing

- Start with the list of all references you can get to
- Recursively follow all references, marking each object
- Delete all unmarked objects

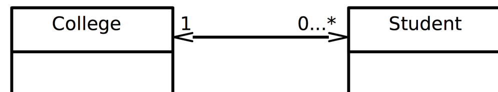
3 UML Class Diagrams

3.1 Class



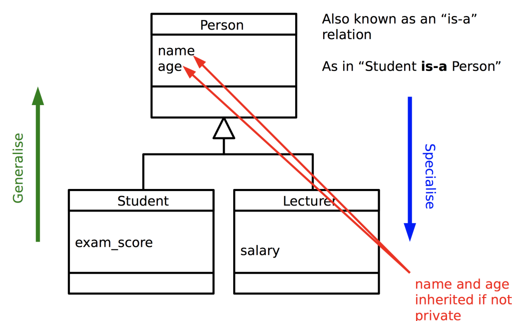
- Abstract methods and classes are indicated by surrounding their name in braces

3.2 Has-a Association



- Left to right indicates a college has 0 or more students
- Right to left indicates a student has exactly one college

3.3 Inheritance (Is-a)



4 References

4.1 References vs. Pointers

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

- If a reference is non-null, it will be valid
- Can not treat references as numbers, eg. incrementing to move throw array

4.2 Iterators

- Allows iteration over a collection while altering its structure

5 Comparisons

5.1 equalto Method

- Use the equalto method, overridden from Object
- Unless specifically overridden this will just use reference equality

```
1        boolean equalto(Object o){ ... }
```

5.2 Reference Equality

- Tests if the references point to the same place in memory

5.3 Comparable Interface

- Implement the `Comparable<T>` interface and override the `compareTo` method
- Returns an integer, (eg. in `this.compareTo(obj)`

$$\begin{array}{rl} r < 0 & \text{this} < \text{obj} \\ r == 0 & \text{this equal to obj} \\ r > 0 & \text{this} > \text{obj} \end{array}$$

- Gives a natural order

5.4 Comparator Interface

- Make a separate class which implements `Comparator<T>`
- Override `compare(T obj1, T obj2)`

6 Copying

6.1 Clone Method

- Implement `Cloneable`
- Override `clone()` method
- Call `super.clone()` and then add further state
- Does a byte-for-byte copy and is therefore a shallow copy, unless you clone all referenced objects in the clone method
- Arrays have built in shallow cloning

6.2 Cloneable Interface

- The cloneable interface is empty, it is a marker interface
- If clone is called on an object that does not implement this interface then exception will be thrown

6.3 Copy Constructor

- Constructor takes an object of the same type
- Manually copies over all state

7 Errors

7.1 Types of Error

- **Syntactic Errors** - Errors in the program's syntax, for example missing a bracket
- **Logical Errors** - Errors in the logic of the program, for example incorrectly bracketing an expression
- **External Errors** - Errors caused by processes code relies on but can not control, for example a failing hard disk

7.2 Returned Codes

- Return value can be ignored
- Have to keep checking what the return values might signify
- The actual result often can't be returned in the same way
- Forces one value in the result range to be used to denote an error
- Results in mixing of code and error handling

7.3 Deferred Error Handling

- Set some state in the system that needs to be checked for errors
- Used in C++ streams, with the good() method

7.4 Exceptions

- An object that can be thrown and caught
- Has the class Exception as an ancestor
- Used with a try ... catch .. finally block
- The finally block will always run, no matter what happens
- Once any catch block is matched the rest will be skipped, start with the most specific exception and generalise
- The exception object can contain state that gives lots of detail on the error that causes the exception

7.4.1 Checked Exceptions

- Must be handled or passed up
- Used for recoverable errors
- Must declare any that a method throws
- Required to catch such errors

7.4.2 Unchecked Exceptions

- Not expected to be handled
- Extends RunTimeException

7.5 Assertions

- Form of error checking for debugging only
- Assertions require the program is consistent with itself, not that the user is consistent with the program
- Great for preconditions, OK for preconditions
- For example

```
assert(x > 0);

//or

assert(x < 0) : "An error message";
```

8 Patterns

8.1 The open-closed principle

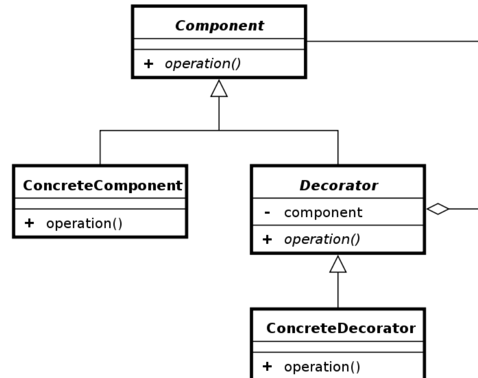
- Classes should be open for extension but closed for modification

8.2 3 Types of Pattern

- **Creational Patterns -** Patterns concerned with the creation of objects (eg. singleton)
- **Structural Patterns -** Patterns concerned with the composition of classes or objects (eg. composite, decorator)
- **Behavioural Patterns -** Patterns concerned with how classes or objects interact with and distribute resources (eg. observer, state, strategy)

8.3 The Decorator Pattern

- How do we add state at runtime?
- Wrap object in another object
- Call the parent's method for all methods unless they add new behaviour
- UML:

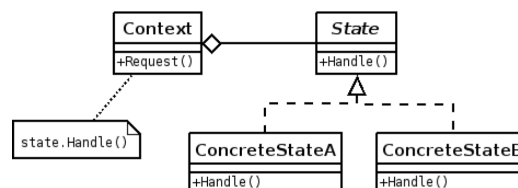


8.4 The Singleton Pattern

- How to ensure only instance of an object is created by developers using your code
- Recipe:
  1. Make the constructor private
  2. Create a single static private instance of the class inside the class
  3. Create a public static getter method that returns the instance, first creating one if required

8.5 The State Pattern

- How can we let an object alter its behaviour when its internal state changes
- Have a member variable of type 'state'
- The state then inherits from this and is held in the state variable, all state specific calls are send to this
- Internal variable NEVER released
- UML:



8.6 The Strategy Pattern

- How can we select an algorithm implementation at runtime?
- Have a member variable that is an object that performs the algorithm
- Each different implementation of the algorithm inherits from the same class

8.7 The Composite Pattern

- How can you treat a group of objects as a single object
- Have a component class that defines some universal methods
- Leaves and composites inherit this
  - Leaves implement the operation
  - Composites implement it by looping through it's children and then optionally performing extra behaviour

8.8 The Observer Pattern

- When an object changes state, how can any interested parties know
- The subject has a list of interested parties
- When an event occurs it cycles through this list and calls a notify method
- The notifty method will perform some action eg. retrieve the new data