Object Oriented Prorgramming Revision Notes

Kevalee Shah

April 20, 2019

Contents

1 Types of Languages

- Functional - computation is the evaluation of mathematical functions and avoids changing state and mutable data. Output of a function only depends on arguements that are passed into the functions. Type of declerative language
- Imperative - uses statements that change a programs state. Focuses on how something should be executed.
- Declarative - expresses that logic of a computation without describing its control flow. Specifies what should be done but not how. This means the compiler can decide how best to implement the function. You can give an example of what to do, but ultimately compiler can do something else provided it gets the same result. This is for optimization.
- Source Code - any collection of code, written using a human readable programming language. It has to be converted into machine code by a compiler to be understood by a CPU¿
- Machine Code - high level programs cannot be understood by the CPU and therefore they need to be converted by the compiler into machine code. It consists of binary digits.
- Assembly Code - The intermediate language between high level and machine code. It has syntax similar to English, but is still more difficult that higher level languages. An assembler converts the assembly code into machine code. It is important to understand computer architecture and

regsiter structure to write assembly code. It is good for building real-time embedded systems. There is a strong correpondence between program's statements and the architecture's machine code instructions.

- Procedural Programming - based on the concept of procedural call. Procedures contain a series of computational steps that need to be carry out.
- OOP - based on the concept of objects, which contain data in the form of fields and code in the form of procedures. Programs are designed by making them out of objects that interact with each other. Most OOP languages are class based, meaning objects are instances of classes, which determine the type.
- Functions - return non-void result, result only dependent on inputs and no state outside the function can be modified
- Procedures - can use or alter larger system state, meaning that subsequent calls to the same function can different results. can also have procedures that have no arguments and can return nothing.
- Statically Typed - every variable name must be bound to a type
- Dynamically Typed - each variable name is not assigned a type
- Platform vs Language - Languages offer style and syntax, whereas platforms are an execution environment that allow the programming language to run. Java is a platform, not just a programming language. However since Java runs on a virtual machine, the platform is abstracted away.
- Class Library - collection of prewritten classes or prewritten code templates that can be specified and used by a programmer when developing an application. Includes data structures, algorithms, IO, networking, graphical interfaces.

1.1 Primitive Types in Java

- boolean
- char
- byte
- short
- int
- long
- float
- double

1.2 Polymorphism vs Overloading

- Overloading functions - multiple methods can be written with the same name but with different argument types/number and possibly return types.

However cannot have a method with the same type and number of arguments and just a different type.

- This is not very elegant as still need to write out the function for every different type - in ML polymorphism allows one function to be written for many types.

1.3 Classes and Objects

- Class - represent the type and the helper functions that manipulate it. They glue together the state (variables) and the behaviour (functions).
- Object - an instance of a class.
- Classes determine what properties and procedures every onject of the type should have. An object is a specific implementation with specific values.

1.3.1 Constructors

The method called when the object is first made. It has no return type and has the same name as the class. Can have multiple contructors in a class by overloading the method. It can take arguments - e.g. to assign attributes of the object. The defualt constructor has no arguments, no return type and does nothing except allow you to instantiate objects.

2 Static

2.1 Static Variables

A static variable is instantiated once per class, and not once per object. An object doesn't even need to be created in order to access the static variable.
An example of this is `Math.pi` - dont need to instantiate the Math class in order to access *pi*
This can help with memory as well. Say there was a class for students of a college. We can make the college attribute for a student static, as it is going to be the same for all students. Therefore each time a student object is made, the field doesnt need to be given a piece of memory, as it already has one.

2.2 Static Methods

The same applies for static methods - they can be called without having to instantiate an object of that class. E.g. e can just use `Math.sqrt()` without having to instantiate a Math object.
Static methods must not use anything other that local or static variables. The methods are easier to debug, they are self documenting and can group related mathods without needed an object.
If you want to count the number of times a class has been called, the counter variable needs to be static in the class, and then the contructor can increment the static counter variable. If this was not static, it could not be shared between all instances of the class.

3 UML Class Diagrams

UML - unified modelling language.
These diagrams can be used to show the structure of the class - attributes, methods and relationships between classes.
There is a section at the top for the name, then a section for the attributes of the class. The lower section displays the class methods. Before every attribute or class there is an access modifier - indicating whether `public`, `private` etc.
There are arrows (open arrow heads) between classes. They often have multiplicity on them (how many of these something has)
e.g. a college has $0 \cdots$ number of students, but a student has exactly 1 college.
Arrows to show inheritance must be an empty triangle.
Italics for the title indicate it is abstract

4 OOP Concepts

Modularity - break down big problems into smaller ones. Therefore classes are developed to work on a small part of a big piece of code. Therefore each class can be developed and testest separately. It also leads to code re-use as snippets of code from other software can be taken and put in.
Packages are a way to group together conceptually similar classes.

5 Encapsulation

Another word for encapsulation is information hiding - there should be full interaction with a class but nothing about its internal state should be exposed. This means that `access modifiers` can be introduced, to limit what can access certain variables.
Usually everything is `private` meaning only the class itself can use it.
Coupling - how much one class depends on another. High coupling is not good as if you make a change to one class, then many others need to be changed.
Cohesion - how strongly related everything in the class is. We want high cohesion
Encapsulation helps with low coupling and high cohesion

5.1 Access Modifiers

- `public` - everyone, subclass, same package, same class
- `protected` - subclass, same package, same class

  variable exposed for read and write within the class and all subclasses of that class (and in Java also the classes in the same package)
- `package` - same package, same class
- `public` - same class

5.2 Immutable

Advantages:

- means we know that nothing can change a chunk of memory
- can be shared between threads without contention issues
- code more readable and less ambigious

In order to make a class immutable:

- all states are private (or even final)
- final - value never changes after constructions
- methods are not allowed to change internal state

6 Pointers and References

Variables that store memory addresses are called pointers or references in different situations. Manipulating memory directly is faster but riskier.

Pointers Memroy address of the first slot using by the variable. The type of the pointer tells the compiler how many slots the whole object uses. Thefore pointers are just variables that have memory addresses as values.

References Referebces are an alias for another thing. When using a reference, you are redirected to the underlying thing - e.g. object, array etc. In an imperative language, the implementation of a reference is by using pointers. However the difference is that the compielr restricts the operations that violate the properties of reference.

Differences

- Can be `null` - both
- Can be assigned to object - both
- Can be assigned to arbitary chunk of memory - only pointers
- Can be tested for validity - only references

  references only point to something valid or null, whereas pointers can also point to something invalid
- Can perform arthimetic - only pointers

Java has two classes of types: *primitive* and *reference* - primitive types are built-in types, whereas everything else is a reference type.

7 Call Stack

In order to keep track of function calls, we use a data structure called the call stack. New entries are pushed onto the stack, and popped from the top. Stack frames - each entry in the stack - contain the functions parameters, local variables and return address that tells the CPU where to jump to when the function is done. When a function is done we delete the stack fram and continue exectuing from the return address.
Stack overflow occurs when we run out of memory as there are too many stack frames in the stack (e.g. due to nesting)

8 Heaps

—

9 Argument Passing

Pass by reference - create a reference to the object and pass that along. Therefore the function can access the original and change it.
Pass by value - the value of the argument is copies and passed on to the new variable.
Java is strictly pass by value - even with arrays. Arrays are technically references to arrays and therefore by having a function call to an array and changing an element of the array, the value of the array is copied into a new reference. The two references point to the same thing, and therefore changing one also changes the other.

10 Inheritance

If one class (A) is a more specific version of another class (B), we can say that A is the base class, and that B inherits properties from A. This includes both state and functionality. Attributes of a superclass can be inherited if they are not private. Every non-private field is inherited by a subclass
Can think of it like a tree, as you go down the tree you get more sepcific

10.1 Casting

With inheritance it is possible to cast an object to any of the types above it in the inheritance tree. For example, `student` is a `person` and therefore we can do:

```
// this is allowed - widening
Student s = new Student();
Person p = (Person) s;
// this is not allowed - narrowing
Person p = new Person();
Student s = (Student) p;
```

Casting creates a new reference with a different type and points it to the same chunk of memory.

10.2 Shadowing and Hiding

Shadowing - define a variable in a closure scope with a variable name that is the same as one for a variable already defined in the scope
Hiding - define a variable in a child class with the same name as one we defined in the parent class
`this` - can be used in any class method to provide a reference to itself
`super` - gives access to the direct parent of the class (one step up the tree)

```java
class Parent {

    // Declaring instance variable with name ‘x‘

    String x = "Parent‘s Instance Variable";

    public void printInstanceVariable() {

        System.out.println(x);

    }

    public void printLocalVariable() {

        // Shadowing instance variable ‘x‘ with a local variable with the same name

        String x = "Local Variable";

        System.out.println(x);

        // If we still want to access the instance variable, we do that by using ‘this.x‘

        System.out.println(this.x);

    }

}


class Child extends Parent {

    // Hiding the Parent class's variable ‘x‘ by defining a variable in the child class with the same name.

    String x = "Child‘s Instance Variable";

    @Override

    public void printInstanceVariable() {

        System.out.print(x);

        // If we still want to access the variable from the
    super class, we do that by using ‘super.x‘

        System.out.print(", " + super.x + "\n");

    }

}
```

Another example of hiding:

```
public class Circle {
    public float r = 100;
    public float getR() {
        return r;
    }
}

public class GraphicCircle extends Circle {
    public float r = 10;
    public float getR() {
        return r;
    }

    // Main method
    public static void main(String[] args) {
        GraphicCircle gc = new GraphicCircle();
        Circle c = gc;
        System.out.println("Radius = " + gc.r);
        System.out.println("Radius = " + gc.getR());
        System.out.println("Radius = " + c.r);
        System.out.println("Radius = " + c.getR());
    }

}

// This has the output
Radius = 10.0
Radius = 10.0 // circle's method was overriden by
    graphiccircle.
Radius = 100.0
Radius = 10.0
// Fields are statically bound
```

10.3 Overriding

Hiding and shadwoing are for variable, for methods we have overriding. This is when classes inherit methods from their parents, but then want the method to do something different. Signpost this using `@Override`.

```
class Parent {
```

```java
    void show() {
      System.out.println("Parent's show()");
    }
}

// Inherited class
class Child extends Parent {
    // This method overrides show() of Parent
    @Override
    void show() { System.out.println("Child's show()"); }
}
```

10.4 Abstract

If we want to force a class to implement a method, but there is no obvious default, then we can create *abstract classes* and *methods*. The *abstract class* is a base class that cannot be instantiated, but it can be subclassed.
All subclasses must implement all the methods of the superclass. If the subclass does not, then it must also be declared *abstract*.
Abstract methods are methods that are declared without an implementation
If a class has *abstract methods*, the class itself must be declared *abstract*

11 Polymorphism

When a subclass is cast to a parent class, and then an overridden method is called, the question arises which method should be called? - The parent class or the child class

```java
Student s = new Student();
Person p = (Person) s
p.dance(); // supposing both Studen and Person have
    difference implementations of dance
```

There are two types of polymorphism: static and dynamic

- Java - all methods are dynamically polymorphic
- Python - Java - all methods are dynamically polymorphic
- C++ - only virtual functions are dynamically polymorphic

11.1 Static

This is when the the decision is made at compile time, also known as early-binding. Since the true type of the cast object is not known, it will just run the parent method. e.g. in the example above `Parent.dance()` would be the method that is called as the compiler just knows that p is of type `Parent`.
Static polymorphism tends to be faster as it does not rely on pointers.
Static polymorphism allows for different method implementations by casting the same object to different types.

11.2 Dynamic

This is when the decision is made at run-time and is also known as late-binding. The same implementation of the method is achieved regardless of what it has been cast to; regardless of the reference. This means that when the program is run and a decision has to be made as to which method to call, the system has to look at the exact object in memory, and see what its actual type is.
In the example, `Student.dance()` would be called, as s is actually a `Student` object.
Java follows this, and it is very OOP-specific.
There is a performance overhead associated with dynamic polymorphism

12 Multiple Inheritance

Java does not support multiple inheritance due to the following problem:

```java

// A Grand parent class in diamond
class GrandParent
{
    void fun()
    {
        System.out.println("Grandparent");
    }
}

// First Parent class
class Parent1 extends GrandParent
{
    void fun()
    {
        System.out.println("Parent1");
    }
}

// Second Parent Class
class Parent2 extends GrandParent
{
    void fun()
    {
        System.out.println("Parent2");
    }
}


// Error : Test is inheriting from multiple
// classes
class Test extends Parent1, Parent2
{
   public static void main(String args[])
   {
       Test t = new Test();
       t.fun();
    }
}
```

This causes an error on runtime as the system doesnt know which `fun()` method to call. This is also known as the diamond problem, as the classes form a diamond in a UML diagram.

12.1 Interfaces

In order to overcome this, Java supports default methods where interfaces can provide default implementations of methods. One class can implement many interfaces, but can only extend one other class. If both interfaces that have been implemented contain default methods with the same signature, then the class shoud explicitly specify which method is to be used or should override them. Interfaces are classes that have no state and all methods are abstract.

```

// A simple Java program to demonstrate multiple
// inheritance through default methods.
interface PI1
{
    // default method
    default void show()
    {
        System.out.println("Default PI1");
    }
}

interface PI2
{
    // Default method
    default void show()
    {
        System.out.println("Default PI2");
    }
}

// Implementation class code
class TestClass implements PI1, PI2
{
    // Overriding default show method
    public void show()
    {
        // use super keyword to call the show
        // method of PI1 interface
        PI1.super.show();

        // use super keyword to call the show
        // method of PI2 interface
        PI2.super.show();
    }

    public static void main(String args[])
    {
        TestClass d = new TestClass();
        d.show();
    }
}

// Ouput:
Default PI1
Default PI2
```

In order to solve the diamond problem we do the following:

```

// A simple Java program to demonstrate how diamond
// problem is handled in case of default methods

interface GPI
{
    // default method
    default void show()
    {
        System.out.println("Default GPI");
    }
}

interface PI1 extends GPI { }

interface PI2 extends GPI { }

// Implementation class code
class TestClass implements PI1, PI2
{
    public static void main(String args[])
    {
        TestClass d = new TestClass();
        d.show();
    }
}
// Output:
Default GPI
```

13 Lifecylce of an Object

13.1 Initialisation

- Java maintains a `java.lang.Class` object for every class it loads into memory.
- Allows you to query the class - name/ methods it has
- To create a new object:
  - If the class has already been loaded into memory:

    allocate memory for object

    run non static initialiser blocks

    run contructor

  – If the class has not been loaded into memory:

    Load class

    Create `java.lang.Class` object

    Allocate static fields

    run static initaliser blocks

    allocate memory for object

    run non static initialiser blocks

    run contructor

If a subclass is created, then constructors of all parents in sequence are called. If there isnt already, the compiler adds `super()` to the constructor of a child class - this is a call to the parent class' constructor. If the constructor isnt a default constructor and has arguments, then you need to explicitly chain.

Note: consider the following example.

```
class B {
  int x;
  B (int n) {
    x=n;
  }
  int returnMe () {
    return x;
  }
}

class C extends B {
}

public class Inh3 {
  public static void main(String[] args) {
  }
}
```

This code does not compile as there is no default constructor available in B, and therefore when instantiating C, `super()` is called, but that does not exists in B, only `super(int n)` exists. In order to resolve this,

```
// One way
class B {
    int x;
    B() { } // a constructor
    B( int n ) { x = n; } // a constructor
    int returnMe() { return x; }
}

```

```java
class C extends B {
}

// Another way
class B {
    int x;
    B( int n ) { x = n; } // a constructor
    int returnMe() { return x; }
}

class C extends B {
    C () { super(0); } // a constructor
    C (int n) { super(n); } // a constructor
}
```

13.2 Destruction

- Deterministic - This is when objects are deleted at predictable time - manually or auto deleted once they are out of scope.
- Destructors - in OO languages a destructor is a method that is automatically called when an object is destroyed. It releases any resources and memory that were created or being used for that object
- Non Deterministic - Humans are bad at the above. Either forget to delete which leads to memory leak, or delete too much which causes a crash
- Garbage collection is a system Java uses to figure out when to delete things and it is done automatically.
- Finalizers - conventional destructors do not make sense in a non-deterministic system as we do not know if they will run at all or when they will run. Finalizers are only called when the system deletes the object, which may be never. (Basically interchangeable)

13.2.1 Methods for Garbage Collection

Reference Counting Keeps track of how many references point to a given object. When the number is 0, the programmer cannot access that object and therefore it is deleted. The catch is cycles which are a pain to deal with. Reference counting also means that every object needs more memory in order to store the reference count.

Tracing An alternative way is to have a list of references, and recursively following them marking each object visited. All objects that are not marked are then deleted.

14 Collections

Collection - set of interfaces and classes that handle groupings of objects and allow us to implement algorithms, invisible to the user.

14.1 Iterators

Part of the Java Collections Framework. They differ from enumerations as they allow the caller to remove elements from the underlying collection during the interation.

```java
Iterator<Integer> it = list.iterator();
while(it.hasNext()) {
  it.remove();
}
```

14.2 Object Equality

When comparing objects to see if they are equal, there is a distinction to be made between reference equality and value equality.
Reference equality checks whether the two reference actually point to the same chunk of memory - the same object. Value equality checks if two separate objects happen to have the same value. If two objects are reference equal, they must be value equal as well.

```java
Person p1 = new Person("Bob")
Person p2 = new Person("Bob")

p1 == p2 // false
p1 != p2 // true
p1 == p1 // true
```

The == operator is used to test reference equality

In order to test value equality, to compare two Java objects of the same class, the `boolean equals(Object obj)` method must be overriden and implemented by the class.

```java
public class Customer {
    private String name;
    private String address;
    private String description;
    // ...

    // customers are considered equal is the name and
   address are the same

    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        } else if (obj == null) {
            return false;
        } else if (obj instanceof Customer) {
            Customer cust = (Customer) obj;
            if ((cust.getName() == null && name == null) ||
                (cust.getName().equals(name) && ((cust.
   getAddress() == null && address == null)
```

```
                || cust.getAddress().equals(address))) {
                return true;
            }
        }
        return false;
    }

}

Customer cust1 = new Customer();
Csutomer cust2 = new Customer();

if(cust1.equals(cust2)) {
  // ...
}
```

An important mistake NOT to make:

```
public boolean equals(\textbf{Customer} obj) {
  // ...
}
```

This mistake doesn't override the equals method, but instead overloads it. This means that both methods are now kept. By using the annotation `@Override` the compiler would catch the error, otherwise it would not.

Classes also have hashCodes(). As we are overriding the equals method, the hashCode method also needs to be overriden, as if `equals(a,b)` returns `true`, then `a.hashCode()` must be the same as `b.hashCode`

14.3 Comparable

This is an interface that is part of the Collections Framework. It allows two objects to be ordered. The natural ordering is the automatic ordering of a class - the object itself must know how to be ordered. It has one method, `compareTo(T obj)` which returns a number less than zero, zero, or greater than zero depending on if this is less than obj, equal to obj, or greater than obj. When calling `Collections.sort()` on the list of objects, this is the method used to compare the objects in the list.

```
public class Point implements Comparable<Point> {
  private final int mX;
  private final int mY;

  public Point (int, int y) {
    mX=x; mY=y;
  }

  // sort by y, then x
  public int compareTo(Point p) {
    if ( mY>p.mY) return 1;
    else if (mY<p.mY) return -1;
    else {
      if (mX>p.mX) return 1;
      else if (mX<p.mX) return -1;
      else return 0.
    }
  }
}
```

14.4 Comparator

This is slightly different to the comparable interface, but is another interface that is part of the Java Collections Framework. The comparable interface compares this and another object, whereas the comparator compared tow different class objects provdied. If you want to sort based on attributes, other than the natural ordering, use Comparator.
In order to do this another class needs to be created which `implements Comparator<T>` and has the method `compare`

```
public class Person implements Comparable<Person> {
  private String mSurname;
  private int mAge;

  // natural ordering
  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
```

It is also possible to do this without explicitly creating a new class, by using anonymous classes

```
public List<Pattern> getPatternsNameSorted() {
    // Gets a list of all patterns sorted by name
    List<Pattern> copy = new LinkedList<Pattern>(patterns);
    Collections.sort(copy, byName);
    return copy;
}

Comparator<Pattern> byAuthor = (Pattern p, Pattern q) -> {
    return p.getAuthor().compareToIgnoreCase(q.getAuthor());
};

Comparator<Pattern> byName = (Pattern p, Pattern q) -> {
    return p.getName().compareToIgnoreCase(q.getName());
};

Comparator<Pattern> byAuthorThenName = byAuthor.
    thenComparing(byName);
```

Another example is:

```
Collections.sort(s, new Comparator<String>() {
    @Override
    public int compare(String o1, String o2) {
        return o1.length() - o2.length();
    }
});
```

15 Errors

15.1 Return Codes

This is a traditional and not very good way of dealing with errors. The function returns a value which indicates success/failure/error. If a function naturally returns something, then need to pick cases in which there would be an error. e.g. `sqrt(a)` there will be an error if $a < 0$. Then the code can return another value in that case.
This whole approach is flawed as it depends on the programmer testing the return value - issues with this are the programmer can forget to check certain cases and the code ends up being horrible.

15.2 Exceptions

Exception - object that can be *thrown* or *raised* by a method when an error occurs and is *caught* or *handled* by the calling code.
An exception is an object that has Exception as an ancestor and therefore needs to be created before throwing.
Important:

```
private void calculateArea() throws Exception {
    ...do something
}
```

vs

```
private void calculateArea() {
  try {
      ...do something

    } catch (Exception e) {
        showException(e);
    }
}
```

There is a huge distinction between these. The first one which throws the exception makes sure that the caller handles the exception. With the second method the exception is handled with the method itself. There are different situations when either way is useful. Generally always want to catch sepcific exceptions, not just general ones.
There is a rule that *an overriden method cannot throw any extra exception other than what parent class is throwing.* In this case of course a try-catch block is required.
It is usually better practise to do:

```
performCalculation();
...
try {
    performCalculation();
catch (Exception e) {
    // handle exception
}
```

In order to create a new exception, create a new class that extends Exception

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

- Checked Exceptions - Java requires that exceptions your method throws are declared, and then when the function is called, it is caught
- Unchecked Exception - rarely encountered only if you mess up. These are exceptions that are not expected to be handled. used for programming errors such as RunTime errors, NullPointerExceptions

Exception Chaining - one exception can be realted with another exception, i.e. by describing the cause

15.3 Exception Mistakes

- Do not use it as control flow - only for exceptional circumstances, also makes program run slower as new Exception objects need to be created.
- Do not leave blank handlers

```
try {
  FileReader fr = new FileReader(filename);
}
catch (FileNotFound fnf) {
}
```

- Don't just have general exceptions in the handler

```
try{
  // whatever
}
catch(Exception e) {}
```

15.4 Advantages of Exceptions

- Name can be descriptive
- Doesn't need constant tests - so better flow
- Can give a lot of detail, as the exception is an object that can contain a state
- Can't be ignored

15.5 Final

Often we want something to be run, regardless of whether or not there are errors. The `finally` block is used and is *always* run, even after any handler. Once a catch block is matched, the rest of the catches are skipped and it goes straight to finally if there is one.

```
try {
  fr,read();
}
catch(IOException ioe) {
  // read() failed
}
finally {
  fr.close();
}
```

15.6 Assertions

(not exainable)
Designed for debugging. They are a statement that evaluate a boolean. They help you check the logic of your code. If assertions detect an error they will kill the code - help with development but then should be removed.

16 Copying

Shallow Copy - suppose B was a copy of A. Then object B points to A's location in memory. A new object B is created, and field values of A are copied over to B. If a field in A is a reference to an object (i.e. a memory address), then it copies the reference. In that way B points to the same object as A. Cheaper and simpler to implement.

Deep Copy - all things in object A's memory location get copied to object B's memory location. Objects referenced by B are distinct and independent from those referenced by A. For any referenced object of A, new copy objects are created, and references to these are placed in B. Much more expensive and complicated.

16.1 Clone

Jave has a `Cloneable` interface. If a class does not implement this interface, then it cannot call `clone()` on anything. The default implementation of `clone()` is a shallow copy. In order to make it a deep copy, need to call super.clone() and then copy every single

```
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;
  }
};
```

The cloneable interface is a *marker interface* - an empty interface that is used to label classes. They are useful but also a pain as you cant prevent it from being inherited by all subclasses.

16.2 Copy Constructors

Another way that objects are copied, are by using copy constructors which manually copy the data.

```
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();
  }
}

```

```java
Vehicle v = new Vehicle(5, 0.f, 5.f);
Vehicle vcopy = new Vehicle(v);
```

17 Generics

The original Collections just had collections of Objects. This had many issues.

- Result always had to be casted
- need to know what the return type is
- types could be mixed, and could aslo lead to errors

```java
// Make a TreeSet object
TreeSet ts = new TreeSet();

// Add integers to it
ts.add(new Integer(3));
ts.add(new Person(  Bob  ));

// Loop through
iterator it = ts.iterator();
while(it.hasNext()) {
  Object o = it.next();
  Integer i = (Integer)o;
}
```

Needed a way to restrict data structures, so that they could only be of one type. However a lot of code was already written in the old way and therefore couldn't scrap thats. Needed something that would work with both.

17.1 Type Erasure

Compiler checks code to make sure only one type of object is being used in the Generics objects. It then erases that it knows this, and goes back to the old style of code.

```java
LinkedList<Integer> a = new LinkedList<Integer>();
for (Integer i : a) {
  do(i);
}
```

becomes

```java
LinkedList a = new LinkedList();
for (Object i : a) {
  do((Integer) i)
}
```

This also explains why primitives cannot be used as paramters - whatever is in the brackets must be castable to Object, but primitives are not.

18 Lambdas and other handy things

Java has some additional functionality that makes nicer and neater code:

- Lambda functions/ Anonymous functions

```java
s-> s + "hello";
(x,y) -> x + y;
```

- Functions as values, can have arguments, can have return type or can be void

```java
// 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  s o o n )
```

- forEach lists

```java
List<String> list = new LinkedList<>();
list.forEach(s->{s=s.toupperCase();
        System.out::println(s);};
```

- Collections can be made into streams and then can be filtered or mapped

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

19 Design Patterns

General reusable solution to a commonly occuring problem in software design - dont need to force code to be written to use them

Advantages:

- Help identify aims of code
- Helps ensure solution is sensible
- Help use the power of OOP
- Show how some solutions are bad
- Common way to describe code - good for team work

Open-Closed Principle - classes should be open for extension but closed for modification

Types of Patterns:

- Creational - singleton
- Structural - composite, decorator
- Behavioural - observer, state, strategy

19.1 Decorator

This pattern attaches additional responsibility to an object dynamically. Decorators provide a flexible alternative to subclasses.
e.g. pizza topping example - need a way to determine cost of pizza after toppings have been added

- Decorators have the same supertype as object
- Can have multiple decorators
- Object can be decorated at runtime
- Subclassing adds behaviour at compile time, and affects all instances of the original class, whereas using decorators can provide behaviour at runtime for individual objects.

19.2 Singleton

This pattern restricts the instantiation of a class to one object.

```java

// Classical Java implementation of singleton
// design pattern
class Singleton
{
    private static Singleton obj;

    // private constructor to force use of
    // getInstance() to create Singleton object
    private Singleton() {}

    public static Singleton getInstance()
    {
        if (obj==null)
            obj = new Singleton();
        return obj;
    }
}
```

19.3 State Pattern

Used when an Object changes it behaviour based on internal state
e.g. When a car is being made it can change types - sports, normal, luxury.

The issue with having subclasses is cant change after instantiations, therefore the main class has a state variable, which has type IState. IState is an interface which has concrete subclasses - each of the different types, and the functionality of the individual methods are written there.

19.4 Strategy Pattern

It is a family of algorithms, encapsulates eahc one, makes them interchangeable. Similar to state pattern. We use it if we want to select an algorithm implementation at runtime. We create an interface which then has subclasses for each of our options.
E.g. have a Dog class, with methods eats and bark. But each type of dog behaves differently. Have two interfaces, EatBehabiour with subclasses normal and protien and BarkBehaviour interface with subclasses Playful and Growl. Then Dog has methods doEat and doBark, and then all types of dogs are subclasses of Dog, and can implement what type of food and bark they want

19.5 Composite Pattern

Purpose is to let us treat objects and groups of objects uniformly. Consists of 4 parts

- Component - declares interface for objects in composition, default behaviour common to all classes
- Leaf - defines behavious for primitive objects in composition
- Composite - stores child compnents and implements child related operations
- Client - manipulates objects in the composition

Makes a tree structure, shows hierarchy idek
e.g. box set and individual cds
managers (composite) and developers (leaf), want to get salary for all
Should be used when clients need to ignore the difference between compositions of objectsa and individual objects - if using multiple objects in the same way

19.6 Observer Pattern

There are subject and observer objects.
Example:

- Magazine publisher - subject
- Magazines - data
- Person who has susscribed - observer; gets fata
- Person who has not suscribed - does not get data

This patterns defines a one to many dependency between objects so that one object changes state, all of its dependents are notified and updated. One to

many is between Subject and Observer (publisher to people)
Implementation:

- Observer, Subject are interfaces
- Observers who want the data implement interface
- Observer has `notify()` method which has behaviour for when it gets data
- Subject has Observer collection - list of registered observers
- `registerObserver(observer)` and `unregisterObserver(observer)` are methods to add/ remove observers
- `notifyObservers()` called when data changed and all the observers need to be given new data