Contents

Classes

These are core internal component of the java language, unlike other languages where classes are not required, in JAVA the class is the cornerstone structure around which the entire language is evolving. Classes have two main components - class methods and class variables.

```
class classname {
    type instance-variable1;
    type instance-variable2;
    // ...
    type instance-variableN;
    type methodname1(parameter-list) {
    // body of method
    }
    type methodname2(parameter-list) {
    // body of method
    }
    // ...
    type methodnameN(parameter-list) {
    // body of method
    }
}
```

Constructors

These are special types of class methods. They have no explicit return type, however they have an implicit one, the return type of the constructor of a given class is the class/type itself. By default the language provides a default constructor, which takes no parameters. The default constructor can be re-defined manually/explicitly. If at least one constructor is provided for a class, the default one provided implicitly - `is not` . The constructor in all other regards is similar to class methods, the only difference is that it is the very first `method` that is invoked when a new instance of the class is created. This can be used to initialize or construct the object before it is used

```
class Box {
    double width;
```

```
    double height;
    double depth;

    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }
}
```

```
    Box b = new Box(1,2,3);
    Box b2 = new Box(); // that is compile time error, default constructor
       is superseded by the explicit one
```

The example above provides an explicit constructor, meaning that the default one would not be provided by the runtime, the object of type Box can be created by only providing the 3 parameters needed - in this case the dimensions of the box.

It is possible to invoke a constructor using the `this` keyword, however there are some restrictions. Also it is important to note that there is some `significant` performance impact in using `this()` call in a constructor due to some restrictions and design implementation details in the JVM, meaning that the use of `this()` call in constructors has to be taken with caution

- `this` call has to be the very first call in a `constructor`
- it is not possible to use any instance variable belonging to the class in the call of `this`
- `this({args})` call to a constructor has to be the very first call in a constructor
- it is not possible to have both `this` and `super` call since both have to be the same call in a constructor

```
class TestThis {
    // class member variables are defined here

    TestThis() {
    }

    TestThis(int k) {
        this(); // invoke the default constructor
        // do something else with this implementation
    }

    TestThis(double f) {
        this(); // invoke the default constructor
        // do something else with this implementation
    }
}
```

Methods

The class member methods, differ from constructors mainly by the way they are named, and that they usually have a return type, which is explicitly defined even if it is void. The class member methods can not have a name that matches the name of the class since that one is reserved for constructors, other than that, the class member methods can accept any number of arguments, it is important to note that every class member method (non static ones) receive implicitly by default as the first argument the object or instance reference of the instance of the class currently being operated one

```
class Box {
    double width;
    double height;
    double depth;

    double volume() {
        return width * height * depth;
    }
}
```

```
   Box b = new Box();
   b.volume() // is actually implicitly compiled to Box.volume(b)
```

Caveats

The `this` keyword is an important detail which is often overlooked, the keyword provides a way to explicitly reference the instance of the class from within its constructors or methods. The most common use case is to avoid ambiguity between constructor or method arguments and class members.

```
// let us have following constructor for type Box
Box(double width, double height, double depth) {
    this.width = width;
    this.height = height;
    this.depth = depth;
}
```

Note that had we skipped the this keyword usage in the method above, we would face what is called instance variable hiding, meaning that the width/height/depth would have refered to the local arguments, passed to the function, since they have a higher weight or scope, they are the closest scope, compared to the class member variables

Destructors

Like other languages Java provides a `similar` way of destroying an object, however the destruction of that object, the time and place is non-deterministic, unlike other languages, such as C++, where the destructor is called when the object goes out of scope (if created on the stack) or when we call `delete` on an object (created on the heap), the Java's finalize method is called when the garbage collector mark and sweep algorithm goes through the objects and cleans up the ones that are no longer referenced, however this is not something we can predict or that can be relied on. The garbage collection is automatic, meaning that, while, yes at some point in the future the finalize method will be called, we can not determine when exactly. The finalize method may be used to free non java resources, such as loaded textures, fonts, images etc.

```
protected void finalize( ) {
    // free non java related resources, to avoid memory or other resource
       leak
}
```

Access

This is a way to control the visibility of class members, meaning that based on the access modifier, certain restrictions apply for the outside world which is trying to access class specific members. Let's begin by defining them - default, public, private and protected.

- When a member of a class is modified by `public`, then that member can be accessed by any other code.
- When a member of a class is specified as `private`, then that member can only be accessed by other members of its class.
- When a member of a class is specified as `protected`, that is usually mostly used in Inheritance situations, where the sub-classes have to access the parent's protected methods to either re-use or override them
- When no access modifier is specified also known as `package-private` (since it lacks a specific keyword), applies when you don't explicitly specify an access modifier for a class, method, variable, or constructor - allows access only within the same package. This means classes, methods, or fields with no access modifier can be accessed by other classes in the same package, but they are inaccessible outside the package.

```java
// File: package1/ClassA.java
package package1;

class ClassA {
    void defaultMethod() {
        System.out.println("Default method in ClassA");
    }
}

// File: package1/ClassB.java
package package1;

public class ClassB {
    public void test() {
        ClassA a = new ClassA();
        a.defaultMethod();  // Accessible, since it's in the same
           package.
    }
}

// File: package2/ClassC.java
package package2;

public class ClassC {
    public void test() {
        ClassA a = new ClassA();  // Error: ClassA is not visible
           outside package1.
    }
}
```

`It is prudent to note that classes can also be defined as default, private, protected or public. However top-level classes can be defined only with default or public modifiers, defining a class as private or protected at top-level (file level)will result in compilation error`

Static

Two types of static class members fields exist - static method members, and static variable members. Both of which have the same restrictions and rules for accessing other members of the class.

- They can not access the `this` keyword, or in other words the current class instance, since static members are by definition not tied up to any instance of the given class
- They can not call or use the super constructor
- They can only manipulate other static members - other static variables or methods

To initialize static variables after their declaration, one can use the so called `static block` which is in the following form presented below. `As soon as the class is loaded it's static blocks and static members are initialized, this is quite important to note`

```
public class Type {

    public static int variable;

    static {
        variable = 1;
    }

    public static void method() {
        System.out.println("v");
    }
}
```

Final

Final members are these which are essentially considered constants, they can be given values only in one of two ways - either through the constructor or at the moment of declaration, if a final field is not given a value it is considered a compile time error, and the compiler will complain.

```
public class Type {
    public final int variable = 1;
    public final int another;

    public Test() {
        another = 2;
    }
}
```

Both are valid usages, and they depend on the general use case, normally when we create immutable but constructible objects, one would like to use the constructor instead of the declaration, since it gives you more control, however if you are building a math library, it is very much natural to initialize the number `pi` already at the moment of declaration, since it would never change.

Applying final to a method, has a different meaning, it usually implies that method will not be available for inherited classes to override, in java all methods are usually by default eligible for overriding, besides the private ones, however we can mark a protected or public method as final in the super class to avoid the inheriting classes from overriding it. Trying to override a final method will result in compile time error.

```
class A {
    final void method() {
        System.out.println("This is a final method.");
    }
}
class B extends A {
```

```
    void method() { // compile time error, this is invalid, method is
       declared final
        System.out.println("Illegal!");
    }
}
```

Methods declared as final can sometimes provide a performance enhancement: The compiler is free to inline calls to them because it "knows" they will not be overridden by a subclass. When a small final method is called, often the Java compiler can copy the bytecode for the subroutine directly inline with the compiled code of the calling method, thus eliminating the costly overhead associated with a method call. This is only possible for final methods.

`Normally, Java resolves calls to methods dynamically, at run time. This is called late binding. However, since final methods cannot be overridden, a call to one can be resolved at compile time. This is called early binding.`

Applying final to a class declaration itself means that we would like to disallow inheriting this class, this has no special benefits in general for compile or runtime optimization but it can ensure that a certain class is well enclosed and protected. Also it is not possible to define a class both `abstract` and `final` at the same time !

```
final class A {
    //...
}
class B extends A { // compile time error will occur here
    //...
}
```

Nesting

A very powerful feature of the java language is to allow us to declare nested classes within each other. Each nested class has direct access to all class members of the one within it is nested, yes even private ones. The enclosing class has no access to the private members of the classes nested within itself.

There are two ways to define a nested class, one is to define it as `static` and the other is to define it without the `static` keyword. They have `vastly different meaning and implications`

```
public class FirstLevel {

    private int member;

    public class NostaticSecondLevel {

        SecondLevel() {
        }
    }

    public static class StaticSecondLevel {

        SecondLevel() {
        }
    }
}
```

- `static` nested class is such that it can not access the members of the enclosing class without having an instance to it first. No instance of the static class is created when a new instance of the outer one is, usually this is done manually and is under the control of the programmer
- `non-static` nested class is such that it is bound to an instance of the enclosing class, meaning we can access members of the enclosing class. When a new instance of the outer is created, it effectively creates an annonymous instance of the inner non-static class as well

`If we think about it, this is very intuitive, because other types of static and non-static members work the same way, the non-static ones, are the ones bound to a particular class instance, while the static ones are not.`

This `static` nested class approach is often used to create a nested, a related `utility` class which can be used within the top level class, such as the following example below, where we define a cart, and it is only natural to have the cart items be also defined as nested class because those two are somewhat really tightly related.

```
public class Cart {

    private List<Item> items = new ArrayList<>();

    public static class Item {

        private String name;

        public Item(String name) {
            this.name = name;
        }
    }

    public List<Item> getItems() {
        return this.items;
    }
}
```

`Note that we have not defined the Item class as private, even though we can, if we expect to operate on individual items in the cart through the getItems member method, we would not be able to declare a variable of type Item, since it is private`

This `non-static` nested class approach is used when we want to provide some sort encapsulation for our outer class, as well as provide a logical means of separation between some structures. As mentioned non static inner classes have direct access to the instance members of a class, since the non-static inner classes are bound to an instance of the outer enclosing class

```
public class FirstLevel {

    private int outer;

    public class SecondLevel {

        private int inner;

        SecondLevel() {
            this.inner = outer + 5;
```

```
        }
    }
```

As we can see the inner class directly uses the `outer` member of the outer class, but that is in the `scope` of an instance of `FirstLevel` unlike the `static` class where an instance is not required. It is possible to define inner classes within any block scope. For example, you can define a nested class within the block defined by a method or even within the body of a for loop.

```
class Outer {
    private int outer_x = 100;

    void test() {
        for(int i=0; i<10; i++) {
            class Inner {
                void display() {
                    System.out.println("outer_x = " + outer_x);
                }
            }
            Inner inner = new Inner();
            inner.display();
        }
    }
}
```

Shadowing in inner non-static class can occur, this is when both the outer class and the inner class have the same members, this however can be resolved by using a special notation which is to use the name of the outer enclosing class followed by this, similarly to how we can access static members by using the name of the class, followed by the name of the member

```
public class FirstLevel {

    private int var;

    public class SecondLevel {

        private int var;

        SecondLevel() {
            this.var = FirstLevel.this.var + 5;
        }
    }
```

Note that there is an interesting caveat when we have nesting of non-static classes combined with static members. There is a special meaning in `static` members and `non-static` nested classes, which make them incompatible.

```
public class FirstLevel {

    public class SecondLevel {

        public static int VARIABLE = 1; // this is not valid we can not
           define a static member in nested non-static non-top level class
```

```
    }
```

Inheritance

The inheritance implementation in java is something that was inherited from languages like C++, however while in C++ we can have more than one super classes or base classes from which our class can inherit, that is not true for java, where one class can inherit only from one non-interface base class. This is done to avoid the diamond issue in inheritance.

```
class Box {
    double width;
    double height;
    double depth;

    Box(double w, double h, double d) {
        this.width = w;
        this.height = h;
        this.depth = d;
    }
}

class BoxWeight extends Box {
    double weight;

    BoxWeight(double w, double h, double, double v) {
        super(w, h, w); // use super to construct the base class from it's
           provided constructor
        this.weight = v;
    }
}
```

Assigning super-class variable with a sub-class - this means that a variable of a given super-class can be assigned with an implementation of the child class. There is also a related principle in programming which extends this idea, known as the Liskov's substitution principle, which states that objects of super-classes should be replaceable with objects of it's sub-classes without affecting the correctness of the program. In the example below we can see that assigning to the `box` variable the object of the created `boxWeight` sub-class which extends the Box class does not indeed change the behavior of the program,

```
    Box box = new Box(1,2,3);
    BoxWeight boxWeight = new BoxWeight(1,2,3,10);
    box = boxWeight; // this is valid we assign the super-class variable
       the sub-class instance
    boxWeight = box; // this is not valid the sub-class can not be
       assigned an object of the super-class
```

Shadowing super class members can be avoided by using the `super` keyword, similarly to the `this` keyword, we can use `super` to tell the compiler to explicitly target the super or base class instance member variable instead of the one defined for the current class. This can remove any ambiguity which may occur while we define sub-classes with similar-same member names or identifiers

```
class BaseClass {

    protected int member;
```

```
    BaseClass() {
        this.member = 0;
    }
}

class ChildClass extends BaseClass {

    protected int member;

    ChildClass() {
        super.member = 1;
        this.member = 2;
    }
}
```

Note something important the super class constructor, explicitly called or not, will be called first when constructing a child-class, meaning that in the example above, `member` from the super-class will be first set to 0, which is what the `BaseClass` constructor does, after which the `ChildClass` constructor will set it to 1

`Note in a class hierarchy, constructors complete their execution in order of derivation, from superclass to subclass. Further, since super( )must be the first statement executed in a subclass' constructor, this order is the same whether or not super( )is used. If super( )is not used, then the default or parameterless constructor of each superclass will be executed.`

Overriding

Method overriding forms the basis for one of Java's most powerful concepts: dynamic method dispatch. Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time. Dynamic method dispatch is important because this is how Java implements run-time polymorphism

In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to override the method in the superclass. When the method is called from the child class, it will always invoke the `overriden` version of the method.

`Method overriding occurs only when the names and the type signatures of the two methods are identical. If they are not, then the two methods are simply overloaded.`

```
class A {
    int i, j;
    A(int a, int b) {
        i = a;
        j = b;
    }
    void show() {
        System.out.print("i and j: " + i + " " + j);
    }
}
class B extends A {
    int k;
    B(int a, int b, int c) {
```

```
        super(a, b);
        k = c;
    }
    void show() {
        super.show();
        System.out.print(" and k: " + k);
    }
    void show(int f) {
        show();
        System.out.print(" and f: " + k);
    }
}
```

```
    A base = new A();
    A child = new B();

    base.show();
    child.show();
    child.show(5);
```

In this example above we have two instances, once which is only of the base class one of the child class, we can see that in the child class we can also reference the original implementation of the method we have overridden from the base class, in this case the first `show` call on the base object instance will print `i and j` and the second show call on the child object instance will print `i and j and k`. The third call shows that the show method can be overloaded instead, this will call/print the overridden version from the child instance of show first showing us `i and j and k and f`

`Note, do not forget to call the super class method member with super, when calling the same method from an overridden method, otherwise if we omit the super keyword, and we call the same method we will end up calling the child overridden member method, effectively entering infinite recursion.`

Polymorphism

Also known as dynamic, run-time call method dispatch is one of the most powerful mechanisms that object oriented design brings to bear on code reuse and robustness.

A superclass reference variable can refer to a subclass object. Java uses this fact to resolve calls to overridden methods at run time. Here is how. When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time.

When different types of objects are referred to, different versions of an overridden method will be called. In other words, it is the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed. Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed.

```
class Figure {
    double dim1;
    double dim2;
    Figure(double a, double b) {
        dim1 = a;
        dim2 = b;
```

```
    }
    double area() {
        System.out.println("Area for Figure is undefined.");
        return 0;
    }
}

class Rectangle extends Figure {
    Rectangle(double a, double b) {
        super(a, b);
    }
    // override area for rectangle
    double area() {
        System.out.println("Inside Area for Rectangle.");
        return dim1 * dim2;
    }
}
class Triangle extends Figure {
    Triangle(double a, double b) {
        super(a, b);
    }
    // override area for right triangle
    double area() {
        System.out.println("Inside Area for Triangle.");
        return dim1 * dim2 / 2;
    }
}
```

One of the most ubiquitous examples to really demonstrate how dynamic dispatch really works, in the real world, and how it can be a very powerful tool, when applied correctly

```
    Figure[] figures = new Figure[2];
    figures[0] = new Triangle(5, 3);
    figures[1] = new Rectangle(1, 2);

    for(Figure fig : figures) {
        System.out.println(fig.area());
    }
```

Abstract

There are situations in which we would like to define a class that must not be initialized or instantiated on it's own but it would still need to share some sort of common member methods and variables from which the sub-class hierarchy is to be build. Instead of using interfaces, which allow us to only specify method members, we can use abstract classes. These are classes which have at least one abstract method in them, this is a method without an implementation, effectively making the class unable to be instantiated on it's own. It can however have as many concrete member method implementations.

```
abstract class A {
    protected int shared;

    abstract void callme();
```

```
    void callmetoo() {
        System.out.println("This is a concrete method.");
    }
}
class B extends A {
    void callme() {
        callmetoo(); // optionally we can also call any of the base member
           methods which are implemented
        // super.callme() - not allowed, this will produce a compile time
           error, callme is not implemented
        System.out.println("B's implementation of callme.");
    }
}
```

To see how we can use them correctly, it is prudent to note again that we can not create an instance of `A` however we can certainly have a variable of type `A` which holds a reference to a child class implementing and inheriting `A`

```
    B child = new B(); // this is also allowed, the child class B
       implements all methods, not defined as abstract
    A base = new B(); // this is allowed we can certainly hold an object
       of type B in a variable referencing the base class A
    A base2 = new A(); // this however is not allowed, since the A class
       is abstract, has 1 unimplemented method in its declaration
```

Object-class

There is one special class, Object, defined by Java. All other classes are subclasses of Object. That is, Object is a superclass of all other classes. This means that a reference variable of type Object can refer to an object of any other class.

Method	Purpose
Object clone( )	Creates a new object that is the same as the object being cloned.
boolean equals(Object object)	Determines whether one object is equal to another.
void finalize( )	Called before an unused object is recycled.
Class<?> getClass( )	Obtains the class of an object at run time.
int hashCode( )	Returns the hash code associated with the invoking object.
void notify( )	Resumes execution of a thread waiting on the invoking object.
void notifyAll( )	Resumes execution of all threads waiting on the invoking object.
String toString( )	Returns a string that describes the object.
void wait( )	Waits on another thread of execution.
void wait(long milliseconds)	Waits on another thread of execution.
void wait(long milliseconds, int nanoseconds)	Waits on another thread of execution.

`The methods getClass(), notify(), notifyAll(), and wait()are declared as final. You may override the others.`