1-core-class-design

Contents

  - Inheritance
  - Polymorphism
  - Overloading
  - Overriding
    * toString
    * hashCode & equals
  - Invocation
    * Constructor
    * Methods
  - Composition
  - Singleton
  - Immutability
  - Static
    * Variable
    * Block
    * Rules
- Summary
  - Implement encapsulation
  - Implement inheritance
  - Implement polymorphism
  - Override hashCode, equals, and toString
  - Singleton and immutable classes
  - Static initialize blocks, variables, methods, and classes

Class design

Encapsulation

The term encapsulation refers to combining data and associated functions as a single unit. In object oriented programming data and associated behavior forms a single unit, which is referred to as a class. The access modifiers determine the level of visibility for a Java entity (a class, method or a field). Access modifiers enable you to enforce effective encapsulation. If all member variables of a class can be accessed from anywhere then there is no point putting these variables in a class and no purpose in encapsulating data and behavior together in a class or unit.

There are four types of access modifiers which are - `public, private, protected and default`

Public

The most liberal one, If a class or its members are declared as public, they can be accessed from any other class regardless of the package boundary. It is comparable to a public place in the real world.

`A public method in a class is accessible to the outside world only if the class is declared as public, if the class does not specify any access modifier itself, meaning it is using default access, then the public method is accessible only within the containing package`

```
// com/java/core/subpackage/DefaultAccessClass/java

package com.java.core.subpackage;
```

```
// the DefaultAccessClass, is declared with `default` access, meaning only
   classes within the `subpackage` can access it
// and its methods.
class DefaultAccessClass {

    // this constructor even if defined public, is only public for classes
       which are within the same package as this class
    // meaning that only those classes can access this constructor and create
        an instance of this class
    public DefaultAccessClass() {
    }
}

// com/java/core/TopLevelClass.java

package com.java.core;
package com.java.core.subpackage;

public class TopLevelClass {

    public static void main(String[] args) throws IOException {
        // this is a compile time error, the DefaultAccessClass is not
           visible, neither is its constructor to the
        // TopLevelClass, since it is located in a different package, at a
           higher level than the DefaultAccessClass
        TestJava2 two = new TestJava2();
    }
}
```

The example above shows how two different files, located in different packages within the same project can guard against global public access when using the default package protected access modifier

Private

As the name implies the most stringent access modifier. A private class member cannot be accessed from outside the class, only members of the same class can access these private members.

`A class or interface cannot be declared as private or protected. Furthermore, member methods or fields of an interface cannot be declared as private or protected.`

```
// this is invalid, will produce a compile time error, there is no way to
   declare a private or protected class, the
// access modifiers that can be put on a class are either public or default (
   no access modifier specified, implies default
// access, see above)
private class PrivateClassType {
   // such a class declaration can never exist, it not allowed by the
      language itself
}
```

Protected

The protected one is a modifier which allows the member which is declared as protected to be accessed from the containing class where it is defined (just as private), and every class which inherits from it.

Default

As already discussed the default or package protected modifier, is the implicit one, when no modifier is specified in front of a class or member declaration, the default one takes effect, it is only accessible to other members within the same package where the protected type / member is defined.

Relationship

The table below represents the Relationship between the different types of modifiers and how are they accessible based on the location they are declared in. Takes into account classes, subclasses - inside or outside the same package.

Modifier	Within the same class	Subclass inside the package	Subclass outside the package	Other class inside the package	Other class outside the package
Public	Yes	Yes	Yes	Yes	Yes
Private	Yes	No	No	No	No
Protected	Yes	Yes	Yes	Yes	No
Default	Yes	Yes	No	Yes	No

Inheritance

A re-usability mechanism in object-oriented programming. With inheritance, the common properties of various objects are exploited to form relationships with each other. The abstract and common properties are provided in the super class, which is available to the more specialized sub class. Inheritance represents a is-a relationship between different class type declarations

```
// take an array of numbers and sum them up
public static double sum(Number []nums) {
    double sum = 0.0;
    for(Number num : nums) {
        sum += num.doubleValue();
    }
    return sum;
}
public static void main(String []s) {
    // create a Number array
    Number []nums = new Number[4];
    // assign derived class objects
    nums[0] = new Byte((byte)10);
    nums[1] = new Integer(10);
    nums[2] = new Float(10.0f);
    nums[3] = new Double(10.0f);
    // pass the Number array to sum and print the result
    System.out.println("The sum of numbers is: " + sum(nums));
}
```

The example above demonstrates how using the top level superclass Number, which is a superclass of types such as Float, Double, Integer etc, can be used to sum up different types, and also collect them into an array of the same type, which are referencing instances of the child classes. Each of them implements the method `doubleValue`, meaning that they can all be summed up together and produce a result even though they are concretely different types

Polymorphism

Generally speaking most languages have two different types of polymorphism, static and dynamic. Each is evaluated during different stage of the `lifecycle` of the program and its execution.

- `Static` - when different forms of a single entity are resolved at compile time (early binding). Function overloading is an example of static polymorphism

```
class Square {
    private int aside;
    private int bside;
    public int area(int a) { return a * a; }
    public int area(int a, int b) { return a * b; }
}

// the area method is overloaded, one version of it will calculate the area
   of an actual square, the other will
// calculate the area of special type of a square - a rectangle, there are
   two versions of the same method - area
Shape shape1 = new Square();
System.out.println(shape1.area(10));
Shape shape2 = new Square();
System.out.println(shape2.area(10, 5));
```

- `Dynamic` - when different forms of a single entity are resolved at run time (late binding). Function overriding is an example of dynamic polymorphism

```
class Shape {
    public double area() { return 0; } // default implementation
    // other members
}
class Circle extends Shape {
    private int radius;
    public Circle(int r) { radius = r; }
    // other constructors
    public double area() {return Math.PI * radius * radius; }
    // other declarations
}
class Square extends Shape {
    private int side;
    public Square(int a) { side = a; }
    public double area() { return side * side; }
    // other declarations
}

Shape shape1 = new Circle(10);
System.out.println(shape1.area());
```

```
Shape shape2 = new Square(10);
System.out.println(shape2.area());
```

Overloading

The process of defining a method with the same name, but with different input arguments/parameters is what method overloading is. This feature works at compile time, and it is the compiler that would actually resolve which call to make, to which relevant version of the overloaded method. The method overloading allows for one to provides a way to define multiple 'versions' of the same method, each of which can do different things.

```
The rule of thumb for overloading is that overloading can not be done entirely based
on return types, it is based on the name of the method along side the order and types
 of arguments this method is declared with, the return type has no control over the
overloading of a given method
```

```
class Circle {
    // other members
    public void fillColor (int red, int green, int blue) {
        /* color the circle using RGB color values - actual code elided */
    }
    public void fillColor (float hue, float saturation, float brightness) {
        /* color the circle using HSB values - actual code elided */
    }
}
```

Overloaded methods can call each other, as long as the same version of the overloaded method does not call itself, otherwise recursion will occur, and that should handled differently with extra care.

Constructors can also be overloaded, they are not much different than regular methods, with the only difference is that every version overloaded method can return different result, however constructors always have one predefined return type, obviously. Another possibility is to call different version of the overloaded constructor, from a constructor, that reduces code duplication and similarly to overloaded methods, constructors can call overloaded constructors for the same type

```
public Circle(int x, int y, int r) {
    xPos = x;
    yPos = y;
    radius = r;
}
public Circle(int x, int y) {
    this(x, y, 10); // passing default radius 10
}
public Circle() {
    this(20, 20, 10);
    // assume some default values for xPos, yPos and radius
}
```

Overloading resolution - the way the compiler resolves overloaded methods is that it follows a procedure - first it looks for the exact match - the method definition with exactly the same number of parameters and types of parameters, If match is not found, it looks for the closest match by using implicit upcasts (autocast), If the compiler can not find a match then you will get a compiler error. What is upcasting, the compiler allows what is called a safe autocast, which is always upcasting action, never downcasting, since that can produce all types of overflow (for integer types) errors or in some cases not even possible to do.

Class type rules

- `Integer` -> can be upcasted to Number (Integer is a child class of Number)
- `Integer` -> can be upcasted to Object (All classes are child classes of Object)
- `String` -> can be upcasted to CharSequence (String is a child class of CharSequence)
- `String` -> can be upcasted to Object (All classes are child classes of Object)

Primitive type rules

- `byte` -> `short` -> `int` -> `long` (Primitive upcasting rules and procedures)

Cross type rules

- `autoboxing` - primitive types will be autoboxed to their Class equivalents or upcasted Class equivalents

In the example below the overloaded methods are ordered in a way which aims to represent the order in which the overloaded methods can be resolved, starting from the narrowest (byte) type, to the widest (Object)

```
public static void aMethod(byte val) {
    System.out.println("byte");
}

public static void aMethod(short val) {
    System.out.println("short");
}

public static void aMethod(int val) {
    System.out.println("int");
}

public static void aMethod(long val) {
    System.out.println("long");
}

public static void aMethod(Integer val) {
    System.out.println("integer");
}

public static void aMethod(Number val) {
    System.out.println("number");
}

public static void aMethod(Object val) {
    System.out.println("object");
}

public static void main(String[] args) {
    byte b = 9;
    // A explanation of how the resolution works
    // in case no byte overload existed -> short
    // in case no short overload existed -> int
    // in case no int overload existed -> long
    // in case no long overload existed -> Integer
    // in case no Integer overload existed -> Number
```

```
    // in case no Number overload existed -> Object
    aMethod(b);
}
```

Something VERY VERY important to remember is that by default all numeric literals defined in java code are defined as implicit int/Integer

```
// a demonstration of method overloading with two major caveats from the java
    spec - downcast combined with code literals
public static void aMethod (byte val ) { System.out.println ("byte"); }
public static void aMethod (short val ) { System.out.println ("short"); }

public static void main(String[] args) {
    // this will produce a compile time error, the literal 9 is implicitly
       defined as int/Integer by the Java spec, meaning
    // that there is truly no overload that can be used to accept this call,
       remember that downcasts are not allowed, they
    // have to be explicitly defined - like so (byte) 9

    aMethod(9); // compiler error
    aMethod((byte) 9); // that is going to resolve fine, to the byte overload
    aMethod((short) 9); // that is going to resolve fine, to the short
       overload
}
```

Ambiguous overloading, can occur when the compiler can not truly resolve the method to use for overloading, consider a case where the arguments defined/declared for a method call are such that the implicit upcasting provided by the compiler can interfere with the overloading resolution. Even though the method is called with two integer types, there is no direct overload that takes in two integers as arguments, however there are two overloads which have the same type of arguments but in different order, since the compiler will try to upcast the integer to match the `long`, there is no way for it to resolve which one of these two overloads should take precedence, since the order of arguments does matter, so does their type, even more so.

```
public static void aMethod (long val1, int val2) {
    System.out.println ("long, int");
}
public static void aMethod (int val1, long val2) {
    System.out.println ("int, long");
}

// this is ambiguous call, and will produce a compile time error, the
   compiler can upcast both 9 or 10 to long, but
// which overload to call, both of the
overloaded methods are valid, and can be called with these arguments
aMethod(9, 10);

// now the call below is valid, since it explicitly casts the types into the
   respective types, which means the compiler
// now finds an exact overloaded method match, and knows which method to call
    - int, long
aMethod((int) 9, (long) 10);
```

Overriding

By default all classes and types defined by the user or the standard library, extend from the implicit Object type, which provides several important methods which usually should be overridden when a new type is defined, those are:

- `String toString()` - defines how a class should be expressed when printed out, in a human readable format
- `int hashCode()` - creates an integer hash of the class instance, using the instance fields
- `boolean equals(Object)` - compares the current instance with another one

There are several others which are not mandatory, or required to be overridden, but can also be useful in certain situations

- `Object clone()` - clone the state of the current instance into a new object
- `void finalize()` - is the inverse version of the constructor, used to free resources which the instance might use to avoid leaking them

Several others which are not allowed to be overridden, because they are defined as final, are the methods like

- `void wait()and its overloaded counterparts` -
- `void notify()` -
- `void notifyAll()` -
- `Class<?> getClass()` -

```
    // this will produce compile time error, since toString is declared as
       public, in the Object class, it is not possible
    // to reduce the visibility of this method during overriding process,
       neither protected or private will work in this case
    @Override
    protected String toString() {
    }

    // another example, which changes the signature of the declared method,
       in this case the declaration of the toString
    // is explicitly defined that the method returns String, therefore the
       overridden version must comply with that and not
    // change the declaration
    @Override
    public Object toString() {
    }
```

`While overriding a method you can not specify a class privilege which reduces the visibility of the initial method declaration, for example a method declared public, can not be overridden as protected or private, however the other way around it is allowed, a private method, can be overridden as protected or even public. Further more it is not allowed to change the signature of the method, when overriding both the return type and the type and number of arguments, as well as the name (obviously)must match exactly as the ones defined in the parent class`

There is a way to make overriding more user friendly, while still keeping intact the signature of the method being overridden, this is the so called `covariant types`, they allow one to specify a narrower type for the return type the narrower type however `has to be part of the hierarchy of the original type.`

```
public abstract class Shape {
    public abstract Shape copy();
```

```
}

public class Circle extends Shape {
    @Override
    public Circle copy() { /* return a copy of this object */ }
}
```

The example above is using a covariant type for the copy method, overrides the copy method with a different return type, however that return type is very much part of the hierarchy of the Shape type, and is also a narrower type than the Shape which is the original return type of the copy method defined in the Shape abstract class above, therefore that is allowed

Covariant type inference is ONLY allowed for return types, changing the method arguments /parameters from the original declaration is not allowed, even using covariant types, at this point what is happening is method overloading not overriding, a method can not be overriden and overloaded both at the same time, it exists only in one or the other state

```
// imagine the following type Point, which aims to override the equals method
   , however instead of that it actually
// overloads it, leading to hidden mistakes in the code, which are hard to
   spot
public boolean equals(Point other){
    if(other == null)
        return false;
    if((xPos == other.xPos) && (yPos == other.yPos))
        return true;
    else
        return false;
}

Point p1 = new Point(10, 20);
Point p2 = new Point(50, 100);
Point p3 = new Point(10, 20);
// this seemingly works, since the types are of Point, and the method being
   invoked is the actual overloaded one (not
// the overridden) equals specified above, however it is a mistake
System.out.println("p1 equals p2 is " + p1.equals(p2));
System.out.println("p1 equals p3 is " + p1.equals(p3));

Object p1 = new Point(10, 20);
Object p2 = new Point(50, 100);
Object p3 = new Point(10, 20);
// the types are objects, so the .equals being called here is the default
   equals which the Object type implements, that
// usually compares the objects by reference, and naturally the return here
   is `false`
System.out.println("p1 equals p2 is " + p1.equals(p2));
System.out.println("p1 equals p3 is " + p1.equals(p3));
```

The example above shows how `covariant` types are not applicable for argument of a function which needs to be overridden, it effectively overloads it instead, that is why it is advisable to use the `@Override` annotation, that annotation serves as a note to the compiler that one would like to override the method, and changing the

method arguments to covariant types would have immediately resulted in a compiler error, since the compiler would have seen that instead of overriding the method we are overloading it.

`Use the @Override annotation where possible to avoid such hidden mistakes, which can be detrimental to resolving issues in ones code`

toString

As already discussed overriding toString method is not mandatory however it is recommended, this helps identify different objects when they are printed out during debugging or general logging actions. The toString method has a string method signature which has to be followed in order to correctly override the method.

hashCode & equals

These are the two methods which most classes should strive to correctly override, always. They are important for the correct functioning of a lot of internal containers provided by the java core libraries and containers. They are often used to compare and distinguish different instances of a given class. They represent a consistent, constant way to generate a unique identifier for a given state of a given class instance, and also a consistent way to compare different instances, and their state.

`The methods hashCode()and equals()need to be conssitent for a class. For parctical purposes, ensure that you follow this one rule: the hashCode method MUST return the same hash value for two objects if the equals method returns true for them. This is a general rule of thumb that has to be followed to ensure correctness across the program`

Usually both `hashCode` and `equals` use the members of a given class to generate the respective hashCode or check the equality of an object.

```
class Point {

    public int hashCode() {
        // use bit-manipulation operators such as ^ to generate close to
           unique
        // hash codes here we are using the magic numbers 7, 11 and 53,
        // but you can use any numbers, preferably primes
        return (7 * xPos) ^ (11 * yPos) ^ (53 * yPos);
    }

    public boolean equals(Object arg) {
        // perform some basic checks between the two instances, which can
           early exit with a clear answer
        if(arg == null) return false;
        if(this == arg) return true;

        // only in case the target is of the same instance is it actually
           possible to do any meaningful comparison
        if(arg instanceof Point) {
            Point that = (Point) arg;
            // note that both the equals and hashCode use the same members to
                perform the comparison or hash generation
            if((this.xPos == that.xPos) && (this.yPos == that.yPos)) {
                return true;
            }
```

```
        }
        // otherwise return false for equality
        return false;
    }
}
```

In the example above, both `hashCode` and `equals` use the same members and their state to compute the result of the methods, which ensures that executing the `hashCode` and `equals` on two objects which are the same will produce the same result - same `hash code` and also the equals method will return `true`

Invocation

Method invocation from another context (i.e super class) can be tricky, explore the different options one has when having to invoke a method outside the scope of the declaring class type

Constructor

It is often useful to call the base class method inside the overridden method. To do that one can use the special keyword called `super`. The `super` keyword refers to the super class, in this case the parent of the current class, every class type has a super type, at the very least even if a class that has no parent, extends by default from the Object class which is still its super class. When calling the super constructor, from the constructor that has to be the first statement in the constructor.

```
public class InvokeSuperClass {

    // Example 1
    public InvokeSuperClass() {
        // this super call in the constructor has to be the very first call,
           otherwise a compile time error will occur
        super();
        // set more properties here, the super or this keyword can be used to
            reference super or this members, as many
        // times as one would like
    }

    // Example 2
    public InvokeSuperClass() {
        // this super call in the constructor has to be the very first call,
           otherwise a compile time error will occur
        this();
        // set more properties here, the super or this keyword can be used to
            reference super or this members, as many
        // times as one would like
    }

    // Example 3
    public InvokeSuperClass() {
        // this is not possible, one can not invoke the super and then call
           one of the current class/type constructors
        // in the same constructor, this will produce a compile time error
        super();
        this();
```

```
    }

    // Example 4
    public InvokeSuperClass() {
        // this is not possible, the first statement has to be either no call
            to super() or this() constructors, or the
        // call has to be the very first statement in the constructor, this
           will produce a compile time error
        this.member = 0;
        this();
    }
}
```

The examples above show a few options for invoking a `super` and `this` constructor from the declaring type, however note that it is not possible to combine both statements in the same constructor, usually to do this, one has to have two versions of the constructor, one having the `super()` call only, and another which invokes it with `this()`. Also there is a clear example that if a call to this() or super() is made in the constructor it has to be the very first statement in the constructor otherwise a compile time error will be issued.

Methods

As a continuation of the above, calling super methods can be done in any place in the child method, it does not have to be the very first statement of the method, neither does it have to be calling a super method only once, there are no obvious restrictions to calling super methods from a regular member method of a declaring class, besides the obvious recursion problem which might occur, but the compiler will not complain

```
public class MemberMethodSuper {

    public int method() {
        return 0;
    }
}

public class MemberMethodThis extends MemberMethodSuper {

    @Override
    public int method() {
        // the super method is being overridden by a new one, however the
           original one can still be called, the actual
        // implementation is not lost and can still be referenced
        int f = this.computeSomeValue();
        int k = super.method();
        return f + k;
    }

    public int method2() {
        // a completely different method which is not present in the super
           class, also refers to the `original`
        // implementation of the `method`
        return super.method() + super.method();
    }
}
```

Composition

Composition is the companion class design along side Inheritance, unlike Inheritance which represents is-a relationship the Composition represents has-a relationship, meaning that a given set of classes can be combined together co build a more complex data structure, unlike inheritance there is no limit to how many classes one can combine with has-a relationship.

```
public class Circle {
    private Point center;
    private double radius;

    public Circle(int x, int y, int r) {
        center = new Point(x, y);
        radius = r;
    }
}
```

The general rule of thumb is to always prefer composition over inheritance, there are very many cases where composition is the right choice, and inheritance is not. When designing a system, start by considering composition first, then fallback to inheritance where composition might make things less re-usable or harder to use / interface with.

Use inheritance when a subclass specifies a base class, so that you can exploit dynamic polymorphism. In other cases use composition, to get code that is easy to change and loosley coupled. In summary, as mentioned favor composition over inheritance.

Singleton

Special type of pattern in class design which ensures that a given class is only instantiated only once, usually on demand, the first time a method for the class is accessed, but often times that could also be done during a static block, then the class loader loads the classes in the run time for the first time. In either case the idea is that only one single instance would ever need to exist of a given class or type. These usually have no visible to the outside world constructors.

A singleton class usually has a public static final method - getSingleton(), which provides a single point of access for the singleton, in a global context, this method is usually in charge of creating the singleton the first time it is called, and using the very same instance on every subsequent call to the getSingleton method. One such example might be a Logging claass, or in general a class which provides common stateless, utility actions which do not need to be stateful, rarely would one persist a static state in a singleton class.

```
// make sure to define the class as final, usually singleton classes are
   never extended, since they provide a well
// encapsulated functionality
public final class Singleton {

    // declare but do not initialize or construct the member which would
       store a reference to the singleton instance
    private static Singleton instance;

    // the first rule of thumb, to make sure it is indeed going to be
       guaranteed to be singleton, by completely hiding
```

```
    // the default constructor away, now only members of the singleton class
       can access and work with that constructor
    private Singleton() {
    }

    // the second rule, to obtain or create the instance correctly, unless
       that is done in a static block, then the
    // method should be declared as synchronized, this would effectively lock
        the state of the class, that would guarantee
    // that if multiple threads happen to call getSingleton at the same time,
        all but one thread would be locked/wait for
    // the method to execute and complete
    public static synchronized final Singleton getSingleton() {
        if(instance == null) {
            instance = createSingletonInstance();
        }
        return instance;
    }
}
```

The use of synchronized above, while correct is an overkill, since the lock will be put every time the method is called however, it is only needed just once, when the instance is created, there is a more performant solution which will not be such a bottleneck for all subsequent calls where the lock is not necessary.

```
public static Singleton getSingleton() {
    if (instance == null) {
        synchronized (Singleton.class) {
            if (instance == null) {
                instance = new Singleton();
            }
        }
    }
    return instance;
}
```

The slightly modified example above shows how to use a conditional locking, it does effectively the same thing, however only when the instance is not initialized. This saves on extra performance in every other scenario, which is like 99% of the cases - the singleton is created, and only used, where locking is not required.

Immutability

An immutable class is such that once created, its state can never be altered, none of the methods or interface the class exposes would ever mutate the instance, it might produce a copy of the original with a modified state, which copy would also be immutable, but the state of the source instance will never be changed. Such an eample is the String class from `java.lang`. While the String class exposes certain methods, which seem to be mutating the String, actually they are producing a brand new instance of the original, representing the mutated state - e.g. the method trim(), would not trim out the String instance that it is called on, it would instead produce a new instance with the trim applied

Such classes are quite powerful, because they ensure a functional type approach to state transition, implying that they are more robust, easy to use and reason about, however they can also be the source or root of performance and other problems when misused or abused.

- immutable objects are safer to use, once the value is checked or known, it can not be changed, and remains constant
- immutable objects are thread safe, they do not require any locking since no state can be changed, no matter how many threads use it
- immutable objects that have the same state or share some of the state internally can be cached or optimized out by sharing

`There is a good rule of thumb, which states that all classes should be immutable, unless there is a very good reason to make them mutable, and in practice that is usually the case`

To define an immutable class, and make sure it stays the same there are really two main points to consider,

- make the fields final and initialize them in the constructor - this can be taken to extreme levels by making the class itself final, and the methods as well, that would ensure that the class is very much closed to being extended, or the methods overridden, completely never allowing any external party to produce a child class which silently mutates the instance state in the interface implementation
- ensure that methods never mutate the state of the members - extra care should be taken in case the Immutable type itself includes through composition mutable reference types, such types which through their interface can or would mutate the internal state of the instance/object.

```
public final class ImmutableValue {

    // declare them final, to ensure that it can not be re-assigned

    private final Value value;

    // make sure the members are initialized correctly

    public ImmutableValue(Value value) {
        this.value = value;
    }

    // provide the rest of the immutable interface

    public final ImmutableValue increment() {
        // take care of making sure that if the members themselves are
           mutable (e.g. Value, is mutable) the methods do
        // not make any mutable calls on the object, or restore it to the
           original state if that would ever happen, or is
        // required.
    }
}
```

As mentioned immutable classes have the drawback of possibly creating many new instances of the Immutable instance, instead of modifying the original one, which could prove to be a problem, that is usually solved by having a mutable version of the type, or by caching the immutable state, so it can be re-used since it is immutable - the default `java.lang.String` implementation does that

Static

The static context in java is usually bound to the type itself, when the `ClassLoader` loads a given class, the static context is initialized and bound to the loaded Class instance itself, in a way. This gives the user the ability to reference a state which is bound not to an instance of the class, but to the class itself, meaning

that different instances of the class have access to this shared state. It is not really shared, it simply lives in a higher scope, so it is visible between the different instances.

Variable

```
public class StaticVariable {

    private static int k = 0;

    public void increment() {
        k++;
    }
}

// the initial value is going to be 0, at the very moment the class loader
   loads the class into the run-time, the static
// variables and initializers are called
StaticVariable s1 = new StaticVariable();
StaticVariable s2 = new StaticVariable();

s1.increment(); // after this call the value of k will be 1
s2.increment(); // after this call the value of k will be 2

// different instances access the same state through a higher scope or level
   bound to the class type itself, not the instance
```

Block

There is also a static block which extends the meaning and abilities of the static initializer, the application of the static block is to allow more complex code to be executed before a given static variable is initialized

```
public class StaticBlock {

    private static Map<String, String> map;

    static {
        map = new HashMap<String, String>();
        map.put("1", "a");
        map.put("2", "b");
        map.put("3", "c");
        map.put("4", "d");
        map.put("5", "e");
        map.put("6", "f");
    }
}
```

The static block above not only creates an instance of the map, but also makes sure the map is populated with some sort of values. It is executed right after the JVM loads the class into memory. The static block is like a default constructor for the class type itself, in a way, semantically they have a very similar effects.

Rules

There are several rules which apply to static context and static methods in particular, listed below, these are:

- Static methods can not use the this keyword, since that would imply referencing an instance, however in a static context, there is no class instance to speak of
- Static methods can not use the super keyword, for the very same reason mentioned above, it invokes a base class method in an instance context
- Static methods can not be overridden, since overriding is a run time, late binding process, which is only applicable to instance methods
- Static methods are mostly suited for utility purposes and actions, since they can not access member variables of an instance, they should not be used to mutate static state
- Main method must always be defined static, otherwise there is no way for the run-time to locate the entry point for the program, and execute it

Summary

Let us briefly review the key points from each objective in this chapter

Implement encapsulation

- Encapsulation: Combining data and the functions operating on it as a single unit.
- You cannot access the private methods of the base class in the derived class.
- You can access the protected method either from a class in the same package (just like package private or default) as well as from a derived class.
- You can also access a method with a default access modifier if it is in the same package.
- You can access public methods of a class from any other class.

Implement inheritance

- Inheritance: Creating hierarchical relationships between related classes. Inheritance is also called an "IS-A" relationship.
- You use the super keyword to call base class methods.
- Inheritance implies IS-A and composition implies HAS-A relationship.
- Favor composition over inheritance.

Implement polymorphism

- Polymorphism: Interpreting the same message (i.e., method call) with different meanings depending on the context.
- Resolving a method call based on the dynamic type of the object is referred to as runtime polymorphism.
- Overloading is an example of static polymorphism (early binding) while overriding is an example of dynamic polymorphism (late binding).
- Method overloading: Creating methods with same name but different types and/or numbers of parameters.
- You can have overloaded constructors. You can call a constructor of the same class in another constructor using the this keyword.
- Overload resolution is the process by which the compiler looks to resolve a call when overloaded definitions of a method are available.
- In overriding, the name of the method, number of arguments, types of arguments, and return type should match exactly.
- In covariant return types, you can provide the derived class of the return type in the overriding method.

Override hashCode, equals, and toString

- You can override clone(), equals(), hashCode(), toString() and finalize() methods in your classes. Since getClass(), notify(), notifyAll(), and the overloaded versions of wait() method are declared final, you cannot override these methods.
- If you're using an object in containers like HashSet or HashMap, make sure you override the hashCode() and equals() methods correctly. For instance, ensure that the hashCode() method returns the same hash value for two objects if the equals() method returns true for them.

Singleton and immutable classes

- A singleton ensures that only one object of its class is created.
- Making sure that an intended singleton implementation is indeed singleton is a nontrivial task, especially in a multi-threaded environment.
- Once an immutable object is created and initialized, it cannot be modified.
- Immutable objects are safer to use than mutable objects; further, immutable objects are thread safe; further, immutable objects that have same state can save space by sharing the state internally.
- To define an immutable class, make it final. Make all its fields private and final. Provide only accessor methods (i.e., getter methods) but don't provide mutator methods. For fields that are mutable reference types, or methods that need to mutate the state, create a deep copy of the object if needed.

Static initialize blocks, variables, methods, and classes

- There are two types of member variables: class variables and instance variables. All variables that require an instance (object) of the class to access them are known as instance variables. All variables that are shared among all instances and are associated with a class rather than an object are referred to as class variables (declared using the static keyword).
- All static members do not require an instance to call/access them. You can directly call/access them using the class name.
- A static member can call/access only a static member of the same class.