Contents

Advanced design

A significant portion of the questions is usually related to changes introduced in the Java language and the library in java 8. This chapter covers lambda expression, which form the foundation for understanding Streams and the new facilities available in the Java 8 release

Abstract

They provide a way to specify an abstraction, without prodviding implementation details, an abstract class can be more suitable in certain cases instead of interfaces, since they can provide not only behavior (like interfaces) but also the abstraction semantics as well. An abstract class defines a common functionality and also a more stringent definition, for the type.

```
abstract class Shape {
    abstract int area();
}
```

Abstract classes can not be initialized, and they are defined by the keyword abstract, in the class declaration as well as at least one abstract method, which must not have an implementation, if an implementation for that method marked as abstract is provided, the compiler will issue an error.

```
If a class is defined as abstract but has not a single abstract non-implemented method,
it is compile time error, if a class is not defined abstract and has an abstract non-implemente
method, this is also a compile time error, also an abstract class can not be declared
both abstract & final
```

An abstract class can extend off of another abstract class as long as the rules above remain met. Meaning that the extending abstract class must at least leave out one abstract method not implemented

Final

The final keyword, can be applied to classes, method and variables. As already mentioned final classes can not be extended off of. Final methods can not be overridden and final variables can not be re-assigned.

Classes

Declaring a class as final has several benefits, which are usually overlooked, but are very well quite important

- Discussed in the immutability chapter, it is a good idea to define a class `final` to make sure that it is never sub-classes, which would in turn make sure that the behavior of the class is never changed, in a way sealing the class from any external 3rd party interference, which might misuse the interface.
- Added performance - since the compiler sees that the class is final, meaning that it can not be sub-classed, that can be used to optimize the calls to the member methods of the class. This is because for a final class there is no way to have dynamic `polymorphism` (late binding), therefore the calls to the methods can be optimized out during byte code generation.

Methods

A final method has a similar behavior to a final class, it prevents the method from being overridden, which can be used to guarantee that a certain behavior from a super class is never changed, to ensure certain functionality is not mutated by mistake.

Variables

The final variables are mostly used when dealing with immutable classes, they are very useful way to make sure that a member variable is never re-assigned. A final variable can be assgined only once. This is usually done in two places, either in the constructor (all constructors, that are defined for a given class) or along side the variable declaration. If a final variable is not assigned, it will lead to compile time error.

Nested classes

A nested class is such a class that is defined in an enclosing class. There are four types of nested classes

Static inner classes

These classes are defined in a static context bound to the class type itself. There are several rules on how they can be used and what they can access from the enclosing class or type.

- Every static inner nested class is associated `with the enclosing class type itself`.
- The accessibility of the static class is defined by the outer class - meaning that even if the static inner class is defined as public, if the outer class is package protected / default access, then the static inner class will not be publicly accessible, even though it has a public modifier
- The name of the inner static class is expressed inthe context of the wrapping outer class name - meaning that to use the nested static inner class one has to use the following syntax - `uOuterClassName.InnerClassName` it is also possible to staticly import the inner class, to avoid this long expression of class name chaining
- When a inner static class or interface is defined inside an enclosing interface they are defined as implicitly static, there is no way to define a local inner class or interface inside an interface, for the obvious reasons, the same is NOT true for abstract classes however.

- Static nested classes can be declared abstract or final, they can also be used as base classes for other clases, or they can also extend off of another class type, even the enclosing one they are in
- Static nested classes can have static members, which is not true for all types of nested classes.
- Static nested classes can ONLY access static members of the enclosing class, however if the static inner class has a reference to an instance object of the enclosing class it can access its members, no matter the access modifier - private or otherwise.
- Enclosing classes can access the static members of the inner static class, and also if they have a reference to an instance of the inner static class, their member variables, irrespective of the access modifiers - private or otherwise
- To instantiate a static inner class one has to use the following syntax, referencing the outer class name first then the static member class - `OuterClassName.InnerClassName instance = new OuterClassName.InnerClassName()`

An example of the relationship between the Outer and Inner static classes

```
public class Outer {

    private static final int outerStaticMember = 5;

    private int outerMember;

    private Inner outerInstance;

    public Outer() {
        // create the instance of the inner static class passing current
           instance as reference
        outerInstance = new Inner(this);
        // this is also valid access to a private member of the inner class
        int k = outerInstance.innerMember;
    }

    public static class Inner {

        private static final int innerStaticMember = 5;

        private int innerMember;

        private Outer innerInstance;

        public Inner(Outer outer) {
            // assign the inner instance the reference to the outer one
            this.innerInstance = outer;
            // this is valid access of the member of the outer class as
               long as there is an instance to use reference
            int k = this.innerInstance.outerMember;
            // this is also a valid reference to the outer static member
               which requires no instance reference
            this.innerMember = Outer.outerStaticMember;
        }
    }
```

```
}

// creating an instance of the inner class can be done by simply the
   Outer.Inner to create the instance. Note that the
// InnerClassName has to be visible, if the OuterClassName is defined as
   package private (default access) the
// InnerClassName will not be visible if the OuterClassName is not visible
Outer.Inner innerInstance = new Outer.Inner();
```

Inner classes

These are a specialized case of the static inner class, where the inner class is defined as non-static member inside another class. The most important difference for the non-static inner class, compared to the static one is that the instance in this case is not bound to the Class type itself, but to every instance of that class. Each instance of the class has its own instance of the non-static inner class

- Every non-static inner nested class is associated with `an instance of the enclosing class type`. Which is different than the static inner class, which is bound to the class type definition itself, see above.
- The inner class can access the members (non static ones) of the enclosing class without needing a reference to an instance of the enclosing class, this is because the inner non-static nested class is bound to the instance of a class not the class itself, see above.
- The outer class can not access member variables of the nested inner class without declaring an instance of it, obviously.
- The inner non-static class can not have static members defined, this is due to the fact that the class type definition itself is tied to an instance of the enclosing outer class, and since static members are bound to the class type not an instance, `the inner class type definition does not exist independently of the outer class instance`.

```
public class Outer {

    private static final int outerStaticMember = 5;

    private int outerMember;

    private Inner outerInstance;

    public Outer() {
        // create the instance of the inner static class passing current
           instance as reference
        outerInstance = this.new Inner(); // equivalent to calling the
           ctor like new Inner();
        // this is also valid access to a private member of the inner class
        int k = outerInstance.innerMember;
    }

    public class Inner {

        private static final int innerStaticMember = 5;

        private int innerMember;
```

```
        public Inner() {
            // this is valid access of the instance member of the
               enclosing class, note the call to Outer.this
            int k = Outer.this.outerMember;
            // this is also a valid reference to the outer static member
               which requires no instance reference
            this.innerMember = Outer.outerStaticMember;
        }
    }
}

// creating an instance of the inner class outside the context of the
   enclosing class, to do so one has to use a
// reference to a valid `instance` of the Outer class to reference the new
   operator on the outerInstance itself
Outer outerInstance = new Outer();
Outer.Inner innerInstance = outerInstance.new Inner();
```

Local classes

The local classes are named class definitions which exist only in the local scope or definition of the current code block, they are only local and are not bound to the class type or class instance within which they are located.

- Local interfaces can not be created inside methods, constructors or initialization blocks, however this restriction was removed in later versions of Java
- Local inner classes
- It is not possible to return an instance of a locally defined class type directly,

```
    // this is a compile time error, since the new class type only exists
       in the local scope of the method and not
    // outside of it, it is possible to return a super type, in case
       LocalType was a child of another class which
    // was defined in a higher scope, for example if LocalType implements
       Closeable, the method signature could be
    // `Closeable returnLocalType`, which would then be a valid return
       type which is visible to the outside world
    public LocalType returnLocalType() {
        // defines a local type , which only exists for the lifetime of
           this function execution, and not outside of it,
        // this prevents one from actually returning the class type
           directly, however that can be changed by
        // modifying the function to return a super type from which the
           local class type can extend / implement
        class LocalType {
        }
        return new LocalType();
    }
```

- Every local variable accessed in the scope of a local class type, is effectively final, for the whole scope of the function, even outside the local class, meaning that it can not be assigned to, the reference can not

be changed, or in case of primitives they can not be mutated (incremented, decremented)

```
static Color getDescriptiveColor(Color color) {
    Color color2 = color;
    // the color variable will be effectively final for the entire scope
       of the function not just within the local
    // class definition, this is because the variable is accessed in the
       local class.
    class DescriptiveColor extends Color {
        public String toString() {
            // access the color argument, will cause it to automatically
               be treated as final by the compiler even
            // though there is no qualifier on the variable/argument,
               assginments to the `color` variable are not
            // allowed in the local class scope, and outside of it
            return "You selected a color with RGB values" + color;
        }
    }
    // note this assignment - that will NOT compile, and is invalid, even
       outside the scope of the local class,
    // treated as effectively final by the compiler
    color = null;

    // this however is allowed, the color2 variable is NOT used within the
       scope of the inner local class meaning
    // that it can be treated just as any other local variable, it is not
       going to be effectively final
    color2 = null;
    return new DescriptiveColor();
}
```

The general rule of thumb is that one can pass only final variables to a local class, if a variable is not explicitly defined as final, it will become effectively final at the time of compilation, and any re-assignments will be caught by the compiler

Anonymous classes

These classes, as the name implies do not have a name. The declaration of the class automatically derives from the instance-creation expression. They are also referred to as simply anonymous. The anonymous class is useful in pretty much the same cases where one would use local class. They are the same construct semantically as the local class, with the only difference being that they have no name.

- Anonymous classes can not have explicit constructors, since the constructor is named after the name of the class, and they have no name, therefore there is no way to create a constructor for an anonymous class
- The anonymous is defined in the new expression itself
- Anonymous classes can not extend other classes or implement interfaces
-

```
public void method() {
    // this is an example of defining a local unnamed/anonymous class, in
       this case it is meant to provide an
```

```
    // implementation of the Closeable interfaces
    Closeable c = new Closeable() {
        // create a member instance variable
        int customLocalMember f = 0;

        @Override
        public void close() throws IOException {
            // add custom implementation
        }
    };

    // this demonstrates how a local class can be used to override certain
       methods from another concrete class,
    // effectively producing a new type, the actual semantics of this is
       equivalent to having a type, which extends from
    // the ConcreteClassType and overrides the desired methods, it is the
       same syntax, with the only difference being that
    // no explicit constructor can be provided since the anonymous class
       has no name.
    ConcreteClassType concrete = new ConcreteClassType() {
        // it is possible to define local members, which work just as any
           other instance member
        int localAnonymousMember = 0;

        @Override
        public void concreteMethod() throws IOException {
            // override a method from the ConcreteClassType's interface
        }
        @Override
        public void toString() throws IOException {
            // override the to string method
        }
        @Override
        public boolean equals(Object o) throws IOException {
            // override the to equals method
        }
    };

    // use the concrete anonymously created instance implementation of
       Closeable
    c.close();

    // use the overridden local anonymously created instance version of
       the ConcreteClassType
    concrete.concreteMethod();
}
```

Local anonymous classes were for the most part superseded by the introduction of lambda expressions, since the use of an anonymous was almost always expansively done to implement @FunctionalInterface interfaces, in practice.

Enums

Enums or enumeration types provide a way to define a semantic enumeration type, unlike in other languages where the enumeration types are simple integers, in Java the enumeration is a fully fledged class type, it has most all features of a normal class type, and is instantiated as well.

```
// define the most basic form of enum, which evaluates to compile time
   ordinal/integer constants, equivalent to what one
// might find in other languages such as C or C++
enum PrinterType {
    DOTMATRIX, INKJET, LASER
}
```

Just as regular classes, enums can have constructors, however those constructors are always defined as `private`. This is because the `instantiation` of the enum is taken care by the run-time, and not by the client of the API, meaning that creating instances of enums is not allowed (it can be done through reflection, but is never a good idea, and defeats the purpose of enums - which is that each enum entry must only exist once as an instance in the runtime, they are effectively singleton instances)

```
// define an enum, note that each entry is invoked with an argument, which
   matches the constructor type
enum PrinterType {
    // note the semicolon - ; at the end, this is part of the syntax, it
       is mandatory if the enum has other members -
    // methods, variables etc
    DOTMATRIX(5), INKJET(10), LASER(50);

    // an instance member of the enumeration
    private int pagePrintCapacity;

    // the default, and only constructor for this enumeration, enums can
       have as many overloaded constructors as one
    // would like, they also have a default constructor when none is
       provided, just as regular classes
    private PrinterType(int pagePrintCapacity) {
        this.pagePrintCapacity = pagePrintCapacity;
    }

    // enums can have custom methods, which extend their behavior,
    public int getPrintPageCapacity() {
        return pagePrintCapacity;
    }
}
```

`Enums are explicitly defined as public static final, meaning that they are always visible, can not be extended and can not be bound to class instance, i.e they can not be defined as non-static inner classes inside another class type`

There are several rules which apply to enumerations in Java, here are some important points to keep in mind, which differentiate enumerations and their constants from class types and class instances.

- An enum can not be created with the `new` keyword, they are created by the run-time based on their definition in the enum type itself

- Enumerations are comparable with the == operator, this is due to the fact that every enum instance of a given enum type exists and is created only once, by the run-time (see point one above).
- Enumerations from the same type can also be compared using the equals operator, which is equivalent to the == unless overridden
- By default the `toString()` of an enum type is printing the name of the enumeration entry itself, however the `toString()` method can be overridden just like for any other class type
- Using the values() method on the enumeration type, is used to obtain an array of enumeration constants for this specific type.
- Enumeration constants / entries can not be cloned, the clone() method by default is throwing a `CloneNotSupportedException`

Interfaces

Is a special type in the Java language, such that it provides no member variables, and only has member methods, which by default are not implemented, and usually represent some sort of behavior. Unlike abstract classes, interfaces are not implemented using the `extends` keyword, but the `implmements` keyword, on top of that, one class type can `implement` as many interfaces, while as it is well known in java a class can only `extend` from a single parent class

In Java 8 the interface structure was improved with the addition of the so called `default` methods, which provide a way to attach a type of utility methods which are very similar to `static methods`, however they are not static, since they can be overridden by the class which implements the interface, if the class implementor chooses to do so, those methods usually provide a wrapper of some sort around the API of the interface they are within.

```
// here is example from the iterator interface, which provides a default
   method, that loops over the remaining entries
// in the iterator calling hasNext and next, those methods hasNext and
   next are actually not implementedby default however
// the default method is wrapping their usage in a general purpose
   implementation which is fairly straightforward
default void forEachRemaining(Consumer<? super E> action) {
    Objects.requireNonNull(action);
    // note that hasNext is actually an `instance` like call, meaning that
       in the end this will end up in the late
    // binding stage, calling the actual instance of the class
       implementing the Iterator interface, same is true
    // for the call to next
    while (hasNext()) {
        action.accept(next());
    }
}
```

Several key points to remember about interfaces and how they are different from regular classes

- Interfaces can extend from other interfaces, one or more, using the `extends` keyword, however they can not extend from abstract or concrete classes
- Interfaces can not be instantiated they can not have constructors declared or defined, however a reference variable to an interface can refer to an object which implements it => `Iterator it = new IteratorImpl();`

- Interfaces do not contain non-static instance variables, due to the reason above, if a data member is defined in an interface it will actually be implicitly declared as `public static final`
- An interface can have `abstract (implicit), default and static methods` - all methods in an interface are by default abstract, there is no need to explicitly provide that keyword, it is used in abstract classes usually to make sure the method without implementation is marked accordingly, however in interfaces the default is that methods do not have implementation.
- An interface can be declared within another interface or class, and it is always by default `static`, known as nested interface, there is no notion of non-static nested interface, it would not make sense, since non-static nested types are linked to an instance
- An interface `can have an empty body`, usually those interfaces are called marker interfaces, left over for future extension, or just used during reflection stages to identify certain behavior
- If an abstract class implements an interface, it does not have to implement the methods immediately, however the concrete implementations of that class have to.
- Only public member methods of an interface are allowed, after all the interface would be useless if it has a hidden state that is not accessible from the outside, the idea of an interface is to be as open as possible exposing an action or behavior, it should not be used, and can not be used to hide actions or behavior

```
// both the interface and the method are by default implicitly defined
   public, there is no reason to provide a
// qualifier to the methods or to the interface itself
interface Public {

    static int VARIABLE = 1; // is also final by default, has to be
       initialized,

    protected void method2(); // not possible, and will produce a compiler
       error

    private void method1(); // not possible, and will produce a compiler
       error

    void method0(); // is public by default, no need to qualify the method
}
```

- Default methods can not be qualified as synchronized, this is because the qualifier synchronized on the method, requires the runtime to lock the current object/instance, however interfaces are not instantiateable, however it is possible to use a synchronized block inside the default method, which locks around `this`

```
// synchronizing around an interface method, can be done within the body,
   or if the method is overridden in the
// concrete class instead
public interface Locked {

    // this is completely fine, since when the method is invoked, within
       the body we can refer to `this` which
    // refers to the instance or object which is of the ClassType which
       has implemented the interface, however
    // `this` is of type of the interface of course - `Locked`
    public default void method() {
```

```
        synchronized(this) {
            System.out.println();
        }
    }

    // this is not allowed, the method can not be marked as synchronized,
       since the run-time does not know around
    // what/which instance to create the lock
    public default synchronized void method() {
        System.out.println();
    }
}
```

Diamond pattern

There is one well known issue when declaring multiple interfaces, and when a given class implements multiple interfaces, it is possible to have two interfaces being implemented by a single class, where the interfaces provide two default methods with the same signature, this issue can happen only with default methods, for interfaces, since the non-default ones have no implementation, therefore there is no problem with them

```
interface Interface1 {
    default public void foo() { System.out.println("Interface1's foo"); }
}
interface Interface2 {
    default public void foo() { System.out.println("Interface2's foo"); }
}

// even at this point the compiler will produce a compile time error since
   it sees to methods with the same signature
// coming from two different type hierarchies, to avoid this the method
   HAS to be overridden in the child class to
// explicitly say which method is to be used.
class Diamond implements Interface1, Interface2 {

    @Override
    public void foo() {
        // the call below would produce a compiler error, since there is a
           conflict in the resolution of the method
        // call, the compiler does not know which default method it has to
           call, and since the method is not overridden in
        // the implementation, there is no way to resolve the target
           method to be called
        // using foo call directly will not work like so => foo();

        // to resolve the issue above, and avoid the compiler error, one
           has to qualify the method using the `super`
        // keyword, similarly to `this` which refers to the current
           instance, the `super` keyword refers to the super class
        // or interface of a given class, both calls can be used, or
           referenced in here, or in any other method in `Diamond`
        if (condition) {
```

```
            Interface1.super.foo(); // use the default method from the
               first interface
        } else {
            Interface2.super.foo(); // use the default method from the
               second interface
        }
    }
}

// this is fine now, after the method in Diamond correctly resolves the
   conflict between the two `foo` methods, the
// compiler will now know that the method is explicitly overridden in the
   implementation and how to call it
new Diamond().foo();
```

The other scenario is when an interface and base class have the same signature, in that case for backwards compatibility the Java run-time would pick the method from the base class, in this case the class wins rule is applied, in a way to make sure that this is compatible with older versions and behavior of the language.

```
class BaseClass {
    public void foo() { System.out.println("BaseClass's foo"); }
}

interface BaseInterface {
    default public void foo() { System.out.println("BaseInterface's foo");
       }
}

class Diamond extends BaseClass implements BaseInterface {
}

// in this case the BaseClass's foo will be called, even though there are
   two methods with the same signature, the one
// coming from the interface will be totally ignored by the compiler
new Diamond().foo();
```

Now a small caveat, what will happen if the method in BaseClass was marked as `final`, since it means that it can not be overridden, the compiler will be confused, and it will still try to pick the class first rule, however what if one wants to change that, well it CAN not

```
public static class BaseClass {
    // note that the method is marked as final, on purpose
    public final void foo() {
        System.out.println("BaseClass's foo");
    }
}

public interface BaseInterface {
    // a normal default method implementation in the interface
    default void foo() {
        System.out.println("BaseInterface's foo");
    }
```

```
}

public static class Diamond extends BaseClass implements BaseInterface {

    // this will produce a compile time error, even though one might think
       that the compiler will assume that the method
    // to be overridden is the one from the interface, since the one
       coming from the class is final, that is not the case,
    // the class still wins, the compiler still wants to override the
       method from the class, and since it is final, it will
    // throw and error
    @Override
    public final void foo() {
    }
}
```

Functional interfaces

The functional interface is a new notion introduced in the Java 8 release, it is introduced along side the lambda expressions in the language, the idea behind functional interfaces, is that they provide only one single method, which has to be implemented, an interface with no method, or only default methods, is not considered a functional interface, the Java 8 release provides a new annotation which is called `@FunctionalInterface` which can be used to mark interfaces at compile time, which is used to validate if a given interface is truly a functional interface

```
// in java.lang package
interface Runnable { void run(); }
// in java.util package
interface Comparator<T> { boolean compare(T x, T y); }
// java.awt.event package:
interface ActionListener { void actionPerformed(ActionEvent e); }
// java.io package
interface FileFilter { boolean accept(File pathName); }
```

A functional interface has to have at least one abstract, non implemented, non default method, to be considered a functional interface, the functional interface is tightly used and related to lambda expressions, and a huge amount of the new additions to the java.lang library uses them

```
// that is perfectly fine, the interface has one abstract non-implemented
   and non-default method
@FunctionalInterface
public abstract class AnnotationTest {
    abstract int foo();
}

// that results in compiler error, the interface is not a functional
   interface, it contains to methods at all, same
// would happen if the interface had two abstract methods as well, when
   only one is allowed at most
@FunctionalInterface
public abstract class AnnotationTest {
```

```
}

// It results in a compiler error "no abstract method found in interface"
   because it only has a default
// method provided but does not have any abstract methods. How about this
   one?
@FunctionalInterface
public interface AnnotationTest {
    default int foo() {};
}
```

There are a few interesting caveats related to functional interfaces and interfaces in general, interfaces `do not inherit from Object`, but rather implicitly declare many of the same methods as Object. If you provide an abstract method from Object class in the interface it still remains a functional interface.

```
@FunctionalInterface
public interface Comparator<T> {
    int compare(T o1, T o2);
    boolean equals(Object obj);
}
```

The example above is actually a valid functional interface, this is because the method `equals` matches the ones by default defined in the Object class, and even though interfaces `do not` inherit by default implicitly from the Object class, they have the same methods, and in this case the signature of the equals matches the one in Object, therefore there is no compiler error. However had it only had the equals method in the declaration, then a compiler error would occur, since the interface has not a one abstract method (remember equals, and the other ones from Object are not considered abstract, in a way, they are provided with a default implementation by the runtime if one is not)

`Declaring methods which match the default ones from Object, in a functional or regular interface, do not count as abstract methods, they are more like default methods, which are implemented, by the run-time if an implementation is not provided`

Lambda expression

One of the most important features not just of the Java 8 release, but in general, in the language, are the new lambda functions, which are one of the most powerful features, that the language exposes. The lambda expression are like closures, they are an easy way for one to provide a stateful callback or action behavior

There are several coordinated changes in the language, the virtual machine and the libraries that were made to create the new feature of lambda reference functions and make that possible.

- First a new operator was added to introduce the creation of the lambda expression which is the `->`, this operator is used to define and declare lambda references in user space (code)
- The function reference operator was also introduced, which directly related to lambda expressions and allows functions to be converted to lambda expressions, which in tern are simple function interfaces
- Then the default keyword was introduced, used in interfaces and most importantly used to make sure that one can easily convert certain interfaces into functional interfaces while still retaining backwards compatiblity
- The streams library and the integration of the collections library with streams, this change made the streams library interop with collections library very smooth and without any significant friction.

The introduction of lambda expression, introduced a slight paradigm shift in the way programs are developed, allowed by passing behavior or actions in the form function arguments, the lambda expressions represent some sort of stateless transformation on a piece of data which can be passed around just as a simple variable

`The functional programming style is a style of programming that focuses on writing functions to perform tasks, rather than changing the state or data of a program, in a step-by-step approach, functional design comes from languages like lisp or scheme. It promotes functions and actions into user accessible and writeable variables in user space - code`

Key concepts of functional programming

- `Pure functions` - each function should always give the same output if given the same input, without relying on or changing the state of internal or external data the function has access to.
- `State immutability` - data is not changed directly. Instead of modifying existing data in-place, functions create and return new data, based on the original data, This approach avoids side effects, where one part of the program changes something that affects another part in unexpected way.
- `First class functions` - the functions are first class objects, and are treated like any other variables, you can pass them as arguments to other functiosn, return them from functions or assign them to variables. This flexibility lets you write more modular and expressive code
- `High order functions` - those functions are functions which take other functions as input or argument or return functions as output. Common examples are functions like map, filter or reduce which apply other functions to a collection of data in useful ways.

Lambda syntax and declaration. The lambda expression is defined by 2 main elements, the argument list, and the body, they are separated by the special arrow operator ->.

- (type variable) = () -> 5; - simple one line return statements, do not need to explicitly wrap the statement in curly block
- (type variable) = x -> x * x; - single argument lambda expressions where type is not provided for the argument can omit the brackets
- (type variable) = (arg1, arg2) -> { return 5; } - arguments are specified in the list, and the type can often be omitted, due to type deduction based on the left hand side
- (type variable) = (String arg1, String arg2) -> { return arg1 + arg2; } - when types need to be provided, all types have to be provided in the list, not just some of them
- (type variable) = (String arg1, String arg2) -> { action(); } - lambda expression do not have to always return a value, they can simply execute another action and exit

`Lambda expression are always linked to some sort of a type - that can be either a user defined FunctionalInterface, or some of the existing FunctionalInterfaces which the java.lang standard library has created over the years - Function, Predicate, Consumer etc. In other words, the left hand side of a lambda is always a concrete type, it is not any different than any other variable`

```
// this is a custom user defined FunctionalInterface
interface LambdaFunction {
    void call();
}

class FirstLambda {

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

```
        // as described above, the left hand side of the expression of a
           lambda expression, always refers to some type, it
        // is that binding to a type that determines the right hand side
           of a lambda expression, including the body of
        // arguments and the body of the lambda itself.
        LambdaFunction lambdaFunction = () -> System.out.println("Hello
           world");

        // lambda expressions are very much like local anonymous classes,
           they simply provide an easier way to create
        // one, instead of defining a local anonymous class which
           overrides the only method in the interfaces, however.
        lambdaFunction.call();
    }
}
```

`Calling the lambda expression is no different than accessing the respective method of the interface and passing the necessary arguments (if any), that is why lambda expression are only an extension of already existing sematnic and lexical objects from the Java language, they do not introduce a completely new type of lexical scoped object, instead they take advantage of other features of the language`

Lambda Design It is good to make a comparison to other languages, where closures also exist, and functions can be passed as arguments, for example in `javascript` one can assign the function itself to a variable, and then use the call operator `()` to invoke the function referenced in the variable, this is NOT how java wanted to introduce lambdas since that would require a completely new type of syntax and language changes at a much deeper level, instead the java spec is re-using existing functionality to replicate the same.

```
// reference used by foo
var outside = "test";

// declare foo as a closure
var foo = () => {
    console.log(outside);
};

// somewhere else, use foo
foo();
```

The example above illustrates how the closures in other languages, are different than lambda expressions in java, on a fundamental level. That should also make it pretty obvious, why local variables referenced in lambda expression are always `effectively` final, The variables must not be changed, re-assigned in the lambda expression's body or anywhere else in the local scope where the variable is used/visible, due to the way local variables are stored in java, unlike other languages (see below)

Effectively Final Here is a simple example of what would happen when either a local variable of a function argument or parameter would be re-assigned somewhere in the lambda body or in the function body. Note that this is only relevant if the referenced local variable is used in both the lambda body and the function body. For function arguments they are always considered effectively final within a lambda expression body

```
interface Functional {
    void call();
```

```
}

class FunctionalTest {
    public static void main(String []args) {
        // this local variable is not defined as final, but becomes one,
           and is treated as such by the compiler, any
        // attempt to re-assign to this variable will yield compiler
           error, either in the function body outside the lambda
        // body, or in the lambda body itself
        String word = "hello";
        Functional suffixFunc = () -> System.out.println(word + "ay");
        suffixFunc.call();

        // this is not possible, it will yield compiler error, even though
           it happens outside the lambda body, or after
        // the lambda is even created, and called.
        word = null;
    }
}
```

The example below shows how a lambda expression is actually expanded by the compiler. Fundamentally that is not true, the actual run-time creates the lambda in a different way, but conceptually this is what happens, it gets transformed to a named or anonymous class, which captures the captured scope inside member variables, initialized in the constructor

```
// here is a simple example of a lambda expression
public static void main(String[] args) {
    String foo = "Hello, World!";
    Runnable r = () -> System.out.println(foo);
    r.run();
}

// here is how it might be expanded by the compiler
public static void main(String[] args) {
    String foo = "Hello, World!";

    // expanded lambda expression into a inner local class
    final class GeneratedClass implements Runnable {

        // the outside scope captured in as member variables
        private final String generatedField;

        // initialized at the moment of lambda creation
        private GeneratedClass(String generatedParam) {
            generatedField = generatedParam;
        }

        @Override
        public void run() {
            System.out.println(generatedField);
        }
    }
```

```
    Runnable r = new GeneratedClass(foo);
    r.run();
}
```

First it is important to mention that local variables are not shared between threads, in java they are stored on the stack, and are not affected by the memory model, only instance members, static fields, array elements and so on are stored on the heap. Local variables, exception handler parameters are never stored on the heap and not shared between threads. Therefore since they can not be shared, a copy of the value has to be captured in an object (like instance member field) that can be shared between threads.

When the lambda is created what really happens is that local capture, becomes an instance variable of the unnamed or anonymous class (lambda expression), it holds the same value like the value that is held by the local variable, the reference is the same the variables are different, one of them is the local one from the function, the other is a member of the anonymous class, however both are assigned the same starting value.

Now imagine if the language provided a way to change and re-assign this, in user land, the code, it would seem that the same variable - the one from the function scope and the one from the lambda scope are the same (even though they are not, they share the same name only, for ease of use, and share the same initial value), allowing re-assignment separately and independently in the outer function scope and lambda body, would be quite confusing. Member variables are not subject to such restrictions, the lambda automatically captures `this` and an access the members without referencing `this` explicitly, however `this` can be used to be more verbose of course, so can `super`, since the lambda captures the immediate instance which is enclosing it

`Only local variables and function arguments are required to be treated as effectively final, due to the reasons above, if a lambda expression references a member variable, it is not subject to these restrictions, due to the reasons mentioned above, related to the way the memory model works in java, which means that member variables will be mutable, and re-assignable outside and inside the lambda body, this is important to remember`

Summary

Abstract classes and methods

- An abstraction specifying functionality supported without disclosing finer level details.
- You cannot create instances of an abstract class.
- Abstract classes enable run-time polymorphism, and run-time polymorphism in turn enables loose coupling.

Final variables & methods

- A final class is a non-inheritable class (i.e., you cannot inherit from a final class).
- A final method is a non-overridable method (i.e., subclasses cannot override a final method).
- All methods of a final class are implicitly final (i.e., non-overridable).
- A final variable can be assigned only once.

Inner, local and anonymous classes

- Java supports four types of nested classes: static nested classes, inner classes, local inner classes, and anonymous inner classes.
- Static nested classes may have static members, whereas the other flavors of nested classes can't.

- Static nested classes and inner classes can access members of an outer class (even private members). However, static nested classes can access only static members of outer classes.
- Local classes (both local inner classes and anonymous inner classes) can access all variables declared in the outer scope (whether a method, constructor, or a statement block). Use enumerated types including methods, and constructors in an enum type
- Enums are a typesafe way to achieve restricted input from users.
- You cannot use new with enums, even inside the enum definition.
- Enum classes are by default final classes.
- All enum classes are implicitly derived from java.lang.Enum.

Interfaces and Overriding

- An interface can have three kinds of methods: abstract methods, default methods, and static methods.
- The "diamond problem" occurs when a derived type inherits two method definitions in the base types that have the same signature.
- If two super interfaces have the same method name and one of them has a definition, the compiler will issue an error; this conflict has to be resolved manually.
- If a base class and a base interface define methods with the same signature, the method definition in the class is used and the interface definition is ignored.
- A functional interface consists of exactly one abstract method but can contain any number of default or static methods.
- A declaration of a functional interface results in a "functional interface type" that can be used with lambda expressions.
- For a functional interface, declaring methods from Object class in an interface does not count as an abstract method.

Lambda expressions

- In a lambda expression, the left side of the -> provides the parameters; the right side, the body. The arrow operator ("->") helps in concise expressions of lambda functions.
- You can create a reference to a functional interface and assign a lambda expression to it. If you invoke the abstract method from that interface, it will call the assigned lambda expression.
- Compiler can perform type inferences of lambda parameters if omitted. When declared, parameters can have modifiers such as final.
- Variables accessed by a lambda function are considered to be effectively final.