**Lambda:**

A Java lambda expression is a function which can be created without belonging to any class.

It can be passed around as if it was an object and executed on demand.

Lambda expressions basically express instances of functional interfaces (An interface with single abstract method is called functional interface. Eg: java.lang.Runnable).

Lambda expressions implement the only abstract function and therefore implement functional interfaces

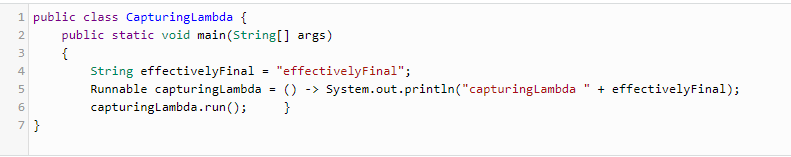
**How does Lambda work?**

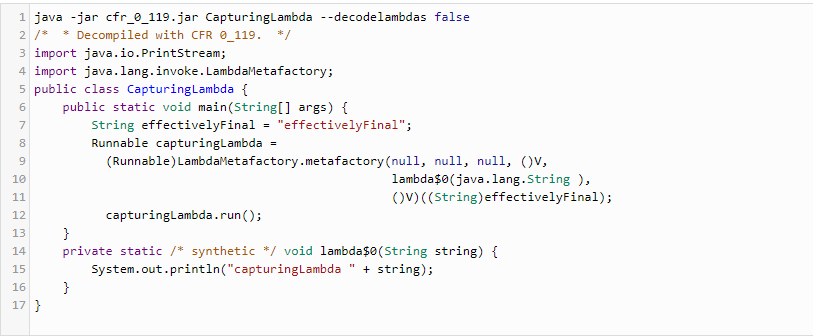
The key to lambda implementation is the InvokeDynamic instruction, introduced in Java 7. This allows dynamic languages to bind to symbols at runtime.

1. Generates invokedynamic call site and uses a lambdafactory to return the functional implementation.

2. Lambda converted to a method to be invoked by invokedynamic.

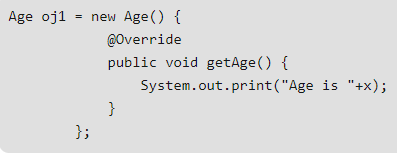
3. The method is stored in a class as a private static method.

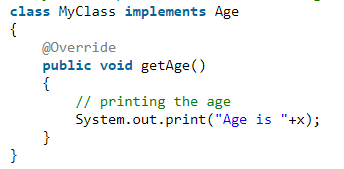




**Anonymous inner class:**

It is an inner class without a name and for which only a single object is created. An anonymous inner class can be useful when making an instance of an object with certain “extras” such as overloading methods of a class or interface, without having to actually be subclass of a class.





Here, an object to MyClass is not created but an object of Age is created and copied in the entire class code as shown above.

So, MyClass is anonymous inner class. It is hidden inner class of Age interface whose name is not written but an object to it is created.

**Diff between wait() and sleep():**

wait() is an instance method that's used for thread synchronization.

It can be called on any object, but it can only be called from a synchronized block. It releases the lock on the object so that another thread can jump in and acquire a lock.

Thread.sleep() is a static method that can be called from any context. Thread.sleep() pauses the current thread and does not release any locks.

**Difference between Array and LinkedList:**

Array - contiguous memory locations time from stack, static memory allocation at compile time, random access

LinkedList - random memory locations from heap, dynamic memory allocation at run time, sequential access

ArrayList in JAVA is implemented as a resizable array. As more elements are added to ArrayList, its size is increased dynamically. So, ArrayList is basically an array.

**Generics:**

They allow types(not primitives) to be a parameter to method, classes and interfaces.

Generics make errors to appear compile time than at run time by means of type safety

List<?> indicates list of any type. ? is called wild card

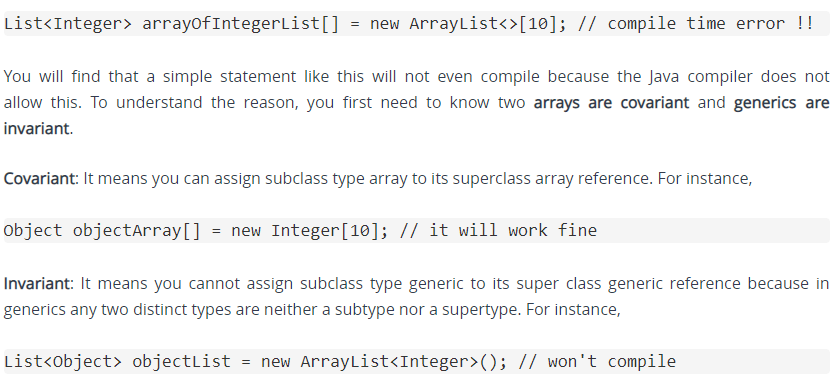
It's important to realize that generic type information is only available to the compiler, not the JVM. In other words, **type erasure** means that generic type information is not available to the JVM at runtime, only compile time.

Bounded Type parameters

Eg: Cage<T extends Animal>. We are forcing T to be a subclass of animal

We cannot create a static field of the type parameter T.

Array doesn’t support Generics and that’s why prefer List over Array because List can provide compile time type-safety over Array



**Static Binding:** The binding which can be resolved at compile time by compiler is known as static or early binding. Binding of all the static, private and final methods is done at compile-time .

If there is any private, final or static method in a class, there is static binding.

**Dynamic Binding:** In Dynamic binding compiler doesn’t decide the method to be called. Overriding is a perfect example of dynamic binding.

private, final and static members (methods and variables) use static binding while for virtual methods (In Java methods are virtual by default) binding is done during run time based upon run time object.

**static keyword:**

* used for memory management mainly, belongs to the class than an instance of the class.
* The static variable gets memory only once in the class area at the time of class loading.
* main() method is static because the object is not required to call a static method.
* "this" can't be referenced in a static method

If it were a non-static method, JVM creates an object first then call main() method that will lead the problem of extra memory allocation.

It makes it very clear for the JVM to call it for launching the Java Application. Otherwise, it would be required to specify the entry function for each Java application build, for the JVM to launch the application.

Thus, the main() method in java is declared as static so that the JVM can access it directly using its class name before object creation.

Static methods can't be overridden but can be overloaded because a static method is resolved at compile time cannot be overridden by a subclass.

An instance method which is resolved at runtime can be overridden.

Overriding means that the particular method would be called based on the run time type of the object and not on the compile time type of it

Hiding: Parent class methods that are static are not part of a child class (although they are accessible), so there is no question of overriding it.

Even if you add another static method in a subclass, identical to the one in its parent class, this subclass static method is unique and distinct from the static method in its parent class.

**Inheritance:**

Java instance variables cannot be overridden in a subclass

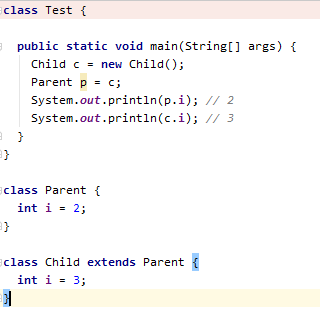
We cannot reduce the visibility of the inherited method from super class. If the overridden or hidden method is public, then the overriding or hiding method must be public; otherwise, a compile-time error occurs

Parent p = new Child();

Child c = new Parent(); // compilation error as We can't store super class object in subclass reference but we can store subclass object in super class reference.

Java instance variables cannot be overridden in a subclass because inheritance is intended to modify behavior which is exposed through methods but not state(variables).

However, they are hidden and are shown based on the reference class (type)



**How to make classes not extendible?**

1. Declare them as final
2. Add a private constructor which can never be invoked from outside of the class, but it prevents the automatic insertion of the default constructor.

**Difference between Aggregation & Composition:**

Aggregation relation is “has-a” and composition is “part-of” relation.

* Aggregation -> child can exist independent of the parent (Bank and Employee, delete the Bank and the Employee still exist).
* whereas Composition -> cannot exist independent of the parent. (Library and book)

Method overloading is the example of compile time polymorphism and can't be performed by changing return type of the method only.

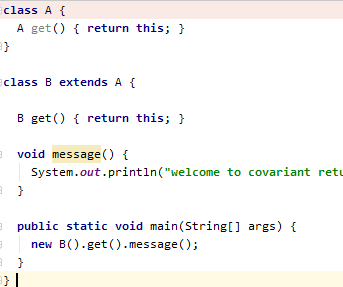
Method overriding is the example of run time polymorphism. (same return type or covariant return type)

final method can be inherited but cannot be overridden.

blank final variable can be initialized only in the constructor

**Covariant Return Type:**

The covariant return type specifies that the return type may vary in the same direction as the subclass.



The return type of the get() method of A class is A but the return type of the get() method of B class is B. Both methods have different return type but it is method overriding. This is known as covariant return type.

**Super Keyword:**

* The super keyword in Java is a reference variable which is used to refer immediate parent class (constructor, method, instance vars).
* Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

**Which is invoked first constructor or initializer block?**

Instance initializer block is not firstly invoked. It is invoked at the time of object creation.

The java compiler copies the instance initializer block in the constructor after the first statement super(). So firstly, constructor is invoked.

**Runtime polymorphism:**

It is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

In this process, an overridden method is called through the reference variable of a superclass.

The determination of the method to be called is based on the object being referred to by the reference variable.

It can't be achieved by data members.

Method overriding is the example of run time polymorphism.

**The Boy Scout Rule:** Leave the codebase cleaner than you found it