



Intermediate Java

Course Overview



The Intermediate Java course examines common language features and APIs required to develop complex stand-alone Java applications. The course builds on the Introduction to Java course and examines topics such as lambda expressions, streams, generic programming, and new features in Java 8.

Part the First



- Structure of Enterprise Applications.
- We are going to concentrate on Java that you would use on the back end, in your business layer Service objects.
- After a brief discussion, you will do a quick lab to create some baseline code that you will work with for the rest of the class.
- Post lab discussion will lead us into discussions of what to look for when trying to designing software for change.
- Design patterns will appear as if by magic – dependency injection, Factory, Builder, Strategy etc.
- We will start to see glimmers of Functional programming techniques and get introduced to Lambdas

Structure of Enterprise Applications



- Layered Architectures. e.g.
 - Presentation Layer
 - Controllers
 - Business Layer
 - Services
 - Integration Layer
 - To access outside resources
 - DAOs
 - Repositories
 - Web clients

Lab



- Setting up a baseline. You are going to write a simple application with just two or three moving parts. Instructions are in the **JavaLabPrequel.pdf** document.
- We will have an extensive post lab discussion about your solutions and look at different implementation choices that we can make.

Structure of Enterprise Applications



- The only sure thing you know about your application is that it will **have** to change for various reasons:
 - You got it wrong
 - The requirements were wrong or incomplete
 - The world changed around you has changed
- So you need to design for change. And you want to make sure your design allows you to change and/or replace individual elements of the various moving parts
 - Maybe a controller needs to use a different service.
 - Or a service needs use a different repository

Structure of Enterprise Applications



- Your design should allow for flexible ways of wiring up your objects.
 - Dependency injection / IOC
- It also very useful if you have some knowledge of how other people have addressed the same issues that you may be facing. In other words, **Design Patterns**. Which are simply codifications of solutions that people have found useful for addressing specific concerns.

Structure of Enterprise Applications



- Different types of design patterns.
 - Structural – ways of composing objects and classes.
 - Adapter – adapts one interface to a different one.
 - Facade – acts as a single interface to a subsystem.
 - Creational – ways of creating objects.
 - Factory
 - Builder
 - Behavioral – ways of getting objects to talk to each other.
 - Strategy
 - Observer
 - We will encounter a few of these in the code.

What is Functional?



- Functional programming can mean many things. We are going to stick to the basics.
- Fundamental to the idea of functional programming is the ability to be able to create references to objects that encapsulate “functions”.
- You can think of such references as pointers to executable code which can be passed in to other functions or returned as results.
- In many Functional languages, functions are a type in the language, just like int and char and MyClass.
- Java does not have a real function type. It fakes it with the notion of a **functional interface**.

Functional Interfaces



- A new name for an old thing.
- A Functional Interface is an interface with just **one abstract** method.
- **Default** and **Static** methods do not count. We will talk about those later in the course.
- You can use the **@FunctionalInterface** annotation on your own interfaces to assert that they obey that rule
 - This is **not** necessary for it to work correctly as a functional interface.
 - But it does make the compiler enforce the rule. Like `@Overrides`
 - And it also works as documentation.
 - A “Good habit”. Again, just like `@Overrides`.

Functional Interfaces



- We are talking about them here because they are fundamental to how Java “fakes” the notion of a Function type.
- In Java, objects that represent Functions are **always** an implementation of a Functional interface.
- This does imply that we have always had the ability to pass functions around in Java. Any object that implements an interface with one abstract method is a “function” in Java.
- The rub, prior to Java 8 was that the syntax to do this was clunky.
- Two features of Java 8 come to the rescue
 - Lambdas
 - Method References

Lambdas



- A simplified syntax for implementing Functional Interfaces.
- Removes syntactic noise.
- Along with greatly enhanced type inference capabilities in the compiler, we now have a much more convenient way to create function objects to pass around in our code.

```
Predicate<String> pred1 = new Predicate<>() {  
    @Override  
    public boolean test(String s) {  
        return s.length() > 5;  
    }  
};
```

//vs

```
Predicate<String> pred2 = s -> s.length() > 5;
```

Basic Lambda Syntax



- (Arg1, Arg2, ...) → code for the implementation
- Let's implement a `Function<String, Integer>` to return the length of the argument.

- Function takes one argument of type `String`
- It needs to return something of type `Integer`
- So we get

Function<String, Integer> func = (String arg1) → {
 return arg1.length;
};

Argument(s) Arrow Code

- Further variations and cleanups are possible, shown on the next page

Basic Lambda Syntax



@FunctionalInterface

```
interface MyInterface {  
    public String doWork(String str, int num);  
}
```

```
class YYY {
```

```
    public void fun() {
```

```
        //For just a single statement, no curly braces required,  
        //return statement NOT allowed. The result of the expression  
        //is the return value.
```

```
        MyInterface mi = (String s, int i) -> s.length() > i ? "Yes" : "No";
```

```
        //Arguments types can often be inferred by the compiler
```

```
        MyInterface mi2 = (s, i) -> s.length() > i ? "Yes" : "No";
```

```
        //For single argument functions, you can leave out the parentheses  
        //around the argument.
```

```
        Consumer<String> cons = s -> System.out.println(s);
```

```
        //The above can be written like this. Lot's of inferencing  
        //and compiler help going on here.
```

```
        Consumer<String> cons2 = System.out::println;
```

```
        //For elaborate work, use curly braces, and a return  
        //is REQUIRED (for non void functions)
```

```
        MyInterface mi3 = (s, i) -> {
```

```
            //other work
```

```
            return s.length() > i ? "Yes" : "No";
```

```
        };
```

```
    }
```


```
}
```

Method References



- Very cool new feature in Java 8.
- Allow you to create references that point at methods.
- So you can create references to existing methods that satisfy the signature of a Functional Interface without having to actually implement the interface.

```
public void foo() {  
    Predicate<String> pred2 = s -> {  
        //lot's of code here  
        return s.length() > 5;  
    };  
  
    //vs  
    Predicate<String> pred3 = this::doAComplicatedTest;  
}  
  
//Existing method  
public boolean doAComplicatedTest(String s) {  
    //lot's of code here  
    return s.length() > 5;  
}
```



java.util.function



- New package in Java 8
- Has a bucket full of new Functional Interfaces which are generally useful in many situations.
- You can create your own Functional Interfaces, but chances are that you may find one in **java.util.function** that will fit your needs.
 - Good practice to use official interfaces when possible
 - But there is no extra magic in the official interfaces, so if none of those shoes fit, then go ahead and make your own.

java.util.function



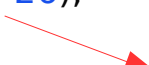
- Some Interfaces that will become your friends
 - Predicate<T> - perform some test on the argument of type T and return true or false
 - Function<T, R> - take in an argument of type T, perform an operation on it, and return some (possibly different) type R
 - Consumer<T> - take in an argument of type T and return void. i.e., *consume* the argument
 - Supplier<T> - take no arguments and return something of type T
- There are also two argument variations of some of the above.
- And interfaces dealing with primitive types, to avoid the expense of auto boxing.

Functions as arguments



- Remember that when we talk about Functions in general, we are not talking about the `java.util.function.Function` interface, but rather about objects that implement **any** Functional Interface
- To pass a function into a method or return it as a result, the corresponding type **has** to be a Functional Interface.

```
public void doit() {  
    List<Integer> input = List.of(1, 20, 50, 33, 60404);  
    List<Integer> result = filter(input, s -> s > 26);  
}  
  
public List<Integer> filter(List<Integer> input, Predicate<Integer> pred) {  
    List<Integer> result = new ArrayList<>();  
    for(Integer s : input) {  
        if(pred.test(s)) {  
            result.add(s);  
        }  
    }  
    return result;  
}
```



Default Interface methods



- Java 8 allows you to create **default** methods in an Interface.
- Useful and necessary when you want to add methods to an interface
- A method declared as default **has** to have an implementation defined in the interface.
- A class implementing the interface can choose to use the default implementation, or override it in the standard way.
- If an actual implementation is found, it is always wins.
- In case of ambiguity, e.g. implementing two interfaces with the same default method, the implementing class **has** to override the method.
- Interfaces still cannot have an state, i.e. **no** variables

Default Interface methods



- Below is the new **forEach** method defined in the `java.util.Iterable` interface.
- It takes a `Consumer` as an argument and calls its `accept` method for each element in the `Iterable`.
- So a `forEach` method is now available for **all** `Iterable` classes.

```
default void forEach(Consumer<? super T> action) {  
    Objects.requireNonNull(action);  
    for (T t : this) {  
        action.accept(t);  
    }  
}
```

Our consumer

passed to

called by

```
//call forEach  
List<String> ls = ...  
ls.forEach(s → System.out.println(s))
```

Static methods and properties



- Java 8 also allows **static** methods in an interface
- They work exactly the same as static methods in a class.
- Since Java 9 you can also have **private** methods in an interface. These are mostly useful as helpers for default methods.

Lab



- Lambda Labs

Part the Second



- Review of Day 1
- We continue our journey into Functional programming.
- We get introduced to the Stream API and spend a good amount of time exploring it's various features, including:
 - Intermediate and Terminal Operations
 - reducing computations to results.
 - Streams of primitives
 - Optional

More Part the Second



- At some point we will swing to talking about Generics in Java.
- Why? How? Pitfalls.
- Wildcards (? super T) and (? extends T)
- Co/Contra variance
- Creating Generic classes and Methods

Lambda issues



- All local variables used inside a Lambda have to be **effectively final**.
 - Declare them as final
 - Or not, you still can't change them.
- Lambdas can only throw those checked exceptions (if any) that are declared in the functional interface they implement.
- Any local variables visible to the code in a Lambda when it is declared get “baked” into the code, and will be available to the code later when that Lambda is run even if the local variables no longer exist at that point.
 - This is known as a **closure**

Streams



- You often need to iterate through a collection of objects and perform some operations on its elements to produce a final result.
- The Stream API addresses this need by allowing you to convert some source of data into a stream of elements that flow through a pipeline of operations to produce the final result.
- You can think of it as another way to do a for loop, where the library controls the actual loop, and you tell it what you want done with each element.
- This approach often makes it much easier to chain operations together to compose a processing pipeline.
- And it makes it much easier to parallelize certain types of workloads

Streams



- A Stream pipeline has 3 parts
 - A source of some sort, often a collection, but streams can be created in various ways.
 - Zero or more **intermediate** operations.
 - One **terminal** operation at the end of the pipeline.
- The terminal operation is **required**. No data flows down the stream until a terminal operation is attached to it.
- A Stream can be used only once. To run the same pipeline again, you have to create a new Stream.

Streams



- An Example:

```
public void foo() {  
    List<Integer> ints = List.of(20, 49, 484, 2085, 48, 12);  
  
    List<Integer> squaresForGT25 = ints.stream()  
        .filter(i -> i > 25)  
        .map(i -> i * i)  
        .collect(Collectors.toList());  
}
```

stream source

intermediate operations

Required terminal operation.

Streams



- Each element in the Stream flows all the way from the source to the terminal operation, or gets thrown away along the way.
- If the intermediate operations are **stateless**, such as filter or map on the previous page, then no temporary collections need to be created in processing the pipeline.
- In the previous example, when the filter operation finds an element less than or equal to 25, it throws it away and goes back to ask the source for the next element.
- Only elements that pass the filter make it to the next operation in the pipeline.

Streams



- **Intermediate** operations all return a Stream type – an easy way to identify them in documentation.
- Some common **intermediate** operations

`filter(Predicate<T>)`

Removes non-matching elements

`map(Function<T, R>)`

Produces a stream of transformed elements

`flatMap(Function<T, Stream<R>)`

Flattens a stream of streams into one stream

`limit(long)`

Truncates the stream's length

`peek(Consumer<T>)`

Calls Consumer for each element

`skip(long)`

Removes elements from start of stream

`sorted`

Orders elements in the stream. **Stateful**

`distinct`

Removes duplicate elements. **Stateful**

Streams



- **Terminal** operations sit at the end of the Stream, and specify what kind of result to produce. These operations do **not** return Streams
- A common name for the act of creating the final result is known as **reduction**.
- The Streams API has two main ways to do a reduction
 - **Mutable Reduction** - using the **collect** method
 - **Immutable Reduction** - using the **reduce** method
- Mutable reductions are often used to produce a collection result. The thing that is being mutated is the result collection.
- Conversely, immutable reductions are used to create a single result like a sum or an average.

Streams



- Some common **terminal** operations

<code>allMatch(Predicate)</code>	Returns whether all elements of this stream match the provided predicate.
<code>anyMatch(Predicate)</code>	Returns whether any elements of this stream match the provided predicate.
<code>collect</code>	Performs a mutable reduction operation on the elements of this stream.
<code>count</code>	Returns the count of elements in this stream.
<code>findFirst</code>	Returns an Optional describing the first element of this stream, or an empty Optional if the stream is empty.
<code>forEach(Consumer)</code>	Performs an action for each element of this stream.
<code>reduce</code>	Performs an immutable reduction on the elements of the Stream.
<code>toArray()</code>	Collect the elements into an array

Streams Collectors



- Collectors are used to do mutable collections.
- Several predefined collectors available through the **Collectors** class.
- If you need to further process collected elements you can add additional **downstream** collectors.

Streams Collectors



- Some common predefined collectors:

<code>counting</code>	Counts the elements in the stream
<code>groupingBy</code>	Various forms of classification operations
<code>joining</code>	Concatenates CharSequence elements to String
<code>maxBy(Comparator)</code>	Find the “maximum” element
<code>minBy(Comparator)</code>	Find the “minumim” element
<code>partitioningBy(Predicate<T>)</code>	Partition into two groups based on the test
<code>toList</code>	Collect the elements into a List
<code>toMap</code>	Map elements into key+value in a Map
<code>toSet</code>	Collect elements into a set
<code>toCollection(Supplier<C>)</code>	Collect to supplied Collection

Optional<T>



- Some terminal operations return an **Optional** type
 - `findFirst`, or `findAny` for example.
- These are operations which might or might not have a result to return.
- In Java you might return a null to indicate no value.
- In Streams (and many other APIs like it) you prefer to return an object that either has a value or is empty.
- You can then further process the result using methods in the `Optional` to:
 - Get the value if it exists or else return a default object.
 - Get the value if it exists or else return a null.
 - Get the value if it exists or else throw an Exception.

Lab



- Streams labs

Generics



- In the language since Java 5
- Used to provide the compiler with extra information to perform type checks at **compile** time.
- Implemented in Java using a mechanism know as **type erasure**.
 - Compile time type information does not make it to the class file
 - For example `List<String>` becomes a standard old pre Java 5 `List` when it gets compiled
- Very important to always use a type argument then using Generic types.
 - Never ignore the “raw type” warning from the compiler

Generics



- Terminology
 - `public class MyClass<T> {...}`
 - **T** is a type **parameter**
 - `MyClass<String> mc = ...`
 - **String** is the type **argument**
- **Parameter** in the declaration, **argument** at the point of use where an actual type is associated with the parameter.

Generics – Raw types



- “Raw type” warning

```
public void fun() {  
    List<String> goodReference = new ArrayList<>();  
  
    goodReference.add("one");  
    //lstr.add(1);    //Compiler error when using Generic reference  
  
    //Using a Generic type in it's  
    // Raw form, i.e. without the type argument.  
    //This will compile but the compiler will  
    // shriek out a warning about Raw Types.  
    // Bad Bad Bad, DON'T do this  
    List badReference = goodReference;  
  
    badReference.add(1); //No compile error  
  
    //ClassCastException here if you use the raw reference  
    for(String s : goodReference) {  
        System.out.println(s);  
    }  
}
```

Custom Generic Types



- Most often you will be using Generic Types supplied to you by a library
 - `List<String>` Is
- You can create your own Generic Types if the need arises.

Custom Generic Types



• An Example

```
public interface BaseDAO<T> {
```

Type parameter specified as **T** in the declaration

```
    void update(T updateObject);
```

T used in the declaration as a place holder

```
    void delete(T oldObject);
```

```
    T create(T newObject);
```

```
    T get(int id);
```

Actual type argument specified when using the Generic type. The compiler uses the argument to make sure you are using the type consistently.

```
    List<T> getAll();
```

```
}
```

```
public class InMemoryStudentDAO implements BaseDAO<Student> {
```

```
    public void update(Student updateObject) {
        if(students.containsKey(updateObject.getId())) {
            students.put(updateObject.getId(), updateObject);
        }
    }
```

```
    public void delete(Student student) {
        students.remove(student.getId());
    }
```

```
    ...
```

```
}
```

Custom Generic Methods



- All methods in a Generic Type that use the type parameters are, by definition, generic methods.
- But you can also create methods which have their own type parameters.
- These can be in a Generic Type, but maybe introduce type parameters that are different from, or in addition to, the parameters of the enclosing class.
- Or they can be methods in a non Generic Type.

Custom Generic Methods



Declaration of parameter **T**.
Without this, the compiler would
see the symbol **T** as a syntax error

```
public <T> void add(List<T> input, T[] arr) {  
    for (T t: arr) {  
        input.add(t);  
    }  
}
```

Use of parameter **T** as
a placeholder for type.

Generic Wildcards



- Using simple type parameters like T or R can often be restrictive, specially when creating generic methods.
- The compiler is very strict when it comes to comparing types, and often will not allow you to do things which should be permissable.
- Generic wild cards to the rescue.

Generic Wildcards



- `List<? extends Number>` is a list which will have *at least* Numbers in it, i.e. a List of Number or any sub type of Number.
- You use this when you want to take elements out of the list and do Number types of things with them. The compiler guarantees that the elements are *at least* of type Number.
- The list is being used as a **Producer** of elements

Generic Wildcards



- `List<? super Integer>` is list of either Integer or some super type of Integer.
 - In other words, the List is a list of some type into which you can *add* Integers.
 - You use this when you want to add things into a List and you want to ensure that the List will be of a type into which you can add the type of thing you are dealing with (in this case Integer).
 - The List in this case is being used as a **consumer** of elements.

Generic Wildcards



- One way to figure out whether you should use **extends** or **super** is to reason it out based on what you are trying to do.
- But, as a memory aid, there is **PECS** – courtesy of Joshua Bloch
- **P**roducer **E**xtends, **C**onsumer **S**uper

Generic Wildcards



- **Producer Extends**
 - A Producer is something which provides you with objects that you want to perform an operation on.
 - You perform operations on specific types, e.g. **Number.doubleValue()**
 - The **<? extends Number>** ensures that you are going to get *at least* a Number
 - In exchange you give up the ability to assign values to the object in question.

Generic Wildcards



- **Consumer Super**

- A Consumer is something that will be accepting objects of some type
- You have to ensure that it's type is *wide enough* to accept the type in question
- Which, by definition, means either that type or any of it's super types
- The `<? super Number>` ensures that you are going to get a object to which you can assign a Number.
- But now you loose the ability to assume anything about the actual type of object. You can assign to it, but not perform any operations on it (except for Object operations).

Co/Contra Variance



- Inheritance relationships in Java determine what objects are assignable to a particular.
- You can assign to a reference either it's own type, or any of it's sub types

```
integer it = 10;  
Number n = it;
```

- Java arrays have the same property

```
integer [] itarr = new Integer[10];  
Number [] narr = itarr;
```

- Integer [] is a sub type of Number []
- We say that arrays are **Covariant**, because the type relationship between them varies in the same direction as the underlying types (Number and Integer in this case)

Co/Contra Variance



- Generic types in Java are **invariant**

```
List<Integer> lint = new ArrayList<>();  
List<Number> lnumBad = lint;    //will NOT compile
```

- List<Integer> **is not** a subtype of List<Number>
- The **extends** wild card makes generic types **Covariant**

```
List<? extends Number> lnumOkay = lint;
```

- List<Integer> **is** a subtype of List<? extends Number>
- You loose the ability to add things to lnumOkay

Co/Contra Variance



- Going the other way, **super** makes types **Contravariant**

```
List<Number> boringNum = new ArrayList<>();  
List<? super Number> ISuper = lint;    // Will NOT compile
```

```
List<? super Number> ISuper = boringNum;  
//A plain old List<Object>  
List<Object> lobj = new ArrayList<>();  
ISuper = lobj;
```

- In the last assignment, a List<Object> is being assigned to List<? super Number>
- So the List<Object> is a sub type of List<? super Number>
- The type relationship between the Lists varies **contrary** to their underlying types - **Contravariance**
- You loose the ability to assume anything about type of elements in ISuper.

Lab



- Generics Labs

Part the Third



- Review of Day 2.
- We start the day talking about Effective Exception handling in Java.
 - The nuts and bolts of Exceptions.
 - Creating custom Exceptions
 - How lambdas and exceptions work together.
 - Short Answer – not well.
 - Exception handling in the presence of lambdas
 - What on earth is a Monad?

More Part the Third



- We wind up the course with a discussion about Reflection and Annotations
 - Why? How?
 - Creating custom Annotations.
 - Make Annotations and Reflection play with each other.

Reflection and Annotations



- Two different things that play well together.
- The Reflection API allows you work with objects and classes that you *did not know about at compile time*.
 - Many of our bread and butter frameworks would be impossible without it.
 - Unit testing frameworks
 - Spring
 - Web servers
 - Validation frameworks
- Annotations give you the ability to provide **metadata** in your code that tools like compilers or frameworks (or your own code) can use to figure out what you want them to do. Often using Reflection.

Reflection



- API to invoke operations on objects and classes that are unknown at compile time.
- **Class** object – For every class the JVM loads, it creates a corresponding object of type **Class**. This is the magic object we need if we are going to use reflection.
- Several ways to get a reference to it. For example, for a class called `ttl.domain.MyClass`
 - **Class.forName**("ttl.domain.MyClass")
 - `MyClass.class`
 - `new MyClass().getClass()`

Reflection



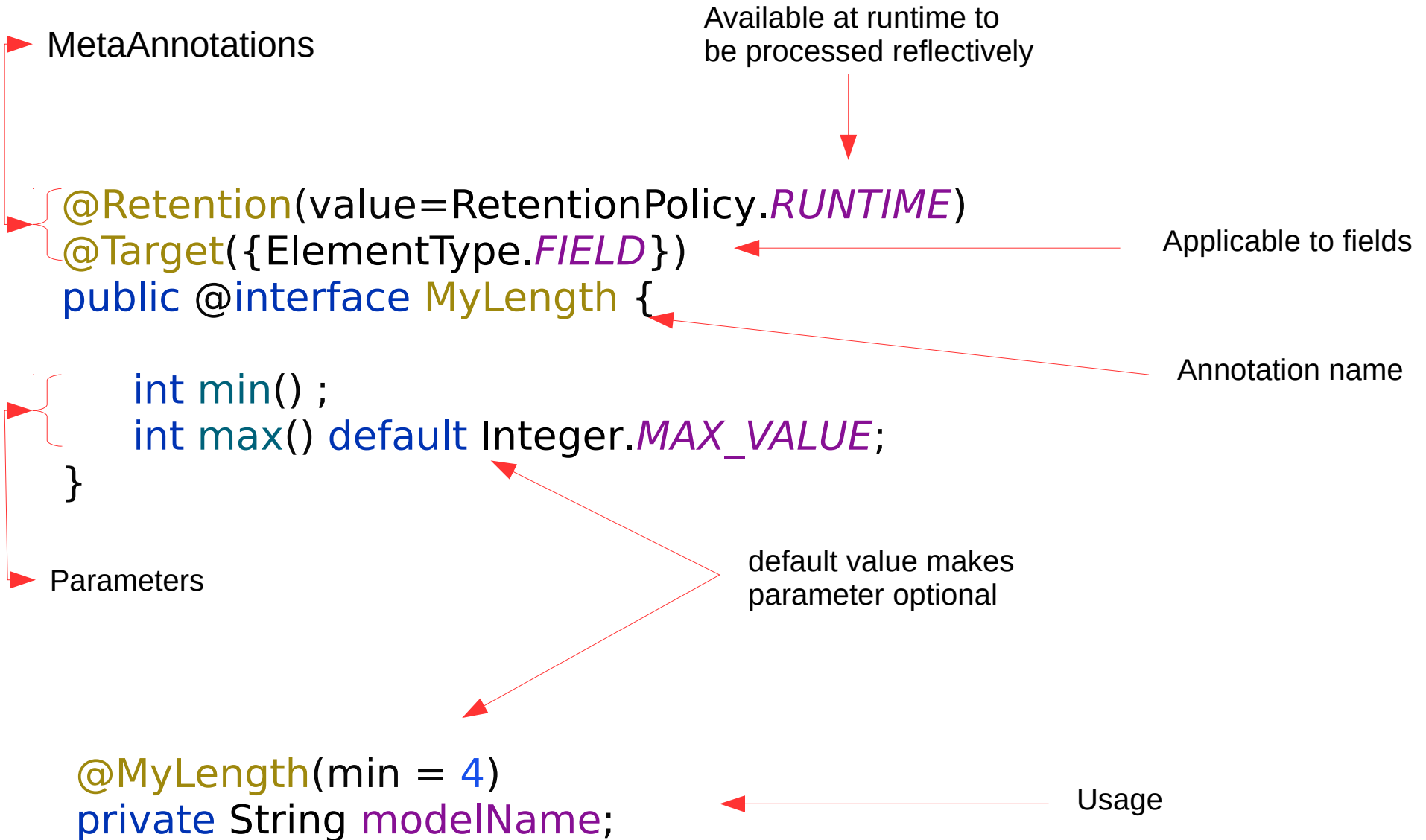
- **Class** objects can give you references to other useful objects
 - **Method** objects which you can use to invoke methods
 - **Constructor** objects to create new objects reflectively
 - **Field** objects to interact with fields
- **setAccessible()** – mechanism to allow you to access private fields of classes
- Several code examples in **`ttl.larku.reflect`**

Annotations



- In the language since Java 5
- Think of them as “sticky notes”.
- Annotations have **no** executable semantics on their own.
- Used to provide metadata that other tools can use as instructions.
- Often read and acted upon using Reflection.

Defining an Annotation



Annotations



- Examples of using Annotations and reflection in **`ttl.larku.reflect`**

Lab



- Reflection Lab