Java Programming 1

Java Collections & Generics. Reflection. Exceptions handling. Lambdas and Stream API.

Course Schedule

- Block 1: 9:00 – 10:30
- Pause: 10:30 – 10:45
- Block 2: 10:45 – 12:15
- Lunch: 12:15 – 13:00
- Block 3: 13:00 – 14:30
- Pause: 14:30 – 14:45
- Block 4: 14:45 – 16:15

Where to Find The Code and Materials?

Java Academy projects and examples are available @GitHub:

https://github.com/iproduct/course-java-web-2021

Generics and Collections

Practical Exercises

Parameterizied Types: Generics (1)

- Collections and their methods before Java 5 were limited to handle a single type of elements.
- If we want to create typed containers we had to implement different container types for each entity type.
- *Example:* In a e-Bookstore we want to sell **Books** and want the container to contain only **Books** (being strongly typed) --> we should implement separate class **BookList**, as well as for each **Book** we want to keep a list of **Authors** --> we should implement **AuthorList** too, and so on.

Parameterizied Types: Generics (2)

❖ *Solution:* We can skip writing multiple similar classes (e.g. typed containers for each type of elements) using **Generic types**

❖ *Generic type invocation:*

```
List<Book> books = new ArrayList<Book>()
List<Author> authors = new ArrayList<Author>()
```

❖ **<> – *Diamond*** *operator* – new in Java™ 7, allows automatic inference of the generic type:

```
List<Book> books = new ArrayList<>()
List<Author> authors = new ArrayList<>()
```

Parameterizied Types: Generics (3)

- *Generic type declaration:*

```
public class Position<T extends Product> {
        private T product;
        ...
        public Position(T product, double quantity) {
                this.product = product;
                this.quantity = quantity;
                price = product.getPrice();
        }
        public T getProduct() {
                return product;
        }
...
```

Generic data type

Conventions Naming Generic Parameters

- **Generic parameters naming conventions:**

T – type parameter (if there are more – S, U, V, W ...)

E – element of a collection – e.g.: List<E>

K – key in associative pair – e.g.: Map<K,V>

V – value in associative pair – e.g.: Map<K,V>

N – number value

- *Example:*

```
public class Invoice <T extends Product> {
            ...
            private List<Position<T>> positions = new ArrayList<>();
     ...
}
```

Generic Methods (1)

- We can implement generic methods and constructors too:

```
public static <U extends Product> String
getPositionsAsString (List<Position<U>> positions) {
     StringBuilder posStr = new StringBuilder();
     int n = 0;
     for(Position<U> p: positions){
         posStr.append( String.format(
             "\n| %1$3s | %2$30s | %3$6s | %4$4s | %5$6s |%6$8s |",
             ++n, p.getProduct().getName(), p.getQuantity(),
             p.getProduct().getMeasure(),p.getPrice(), p.getTotal()
        ));
     }
     return posStr.toString();
} ...
```

Generic Methods (2)

- Invoking generic method / constructor:

```
result += Invoice.<T> getPositionsAsString(positions);
```

- OR we can let Java to automatically infer the generic type:

```
result += Invoice.getPositionsAsString(positions);
```

Bounded Type Parameters

- We can define upper bound constraint for the possible types that can be allowed as actual generic type parameters of the class / method /constructor:

public static <U **extends Product**> String

getPositionsAsString (List<Position<U>> positions) { ... }

- OR

```
public static <U extends Product & Printable> String getPositionsAsString
(List<Position<U>> positions) {
    ...
    p.getProduct().print();
    ...
}
```

Generics Sub-typing

- If the class Product extends class Item, can we say that List<Product> extends List<Item> too? Can we substitute the first with the second?
- The answer is „**NOT**“, because the basic generic type is not designed to reflect the specifics of of the Products.
- Dos and donts when using generics inheritance:

```
interface Service extends Item; Service s = new Service( ...);
Collection<Service> services =  ...; services.add(s); // OK
interface Product extends Item; Product p = new Product( ...);
Collection<Product> products =  ...; products.add(p); // OK
Collection<Item> items = ...; items.add(s); items.add(p); // OK
items = products; // NOT OK
items = services; // NOT OK
```

Using ? as Type Specifier (Wildcards)

- If we want to declare that we expect specific, but not pre-determined type, which for example extends the class **Item**, we could use **?** To designate this:

```
Collection<? extends Item> items; // Upper bound is Item
items = products; // OK
items = services; // OK
Items.add(p); // NOT OK – Can not write into it – it is not safe!
Items.add(s); // NOT OK – Can not write into it – it is not safe!
for(Item i: items) { // OK – Can read it – it is known to be at least Item.
    System.out.println( i.getName() + „:“ + i.getPrice() );
}
List<? super Product> products; // Lower bound is Product
products.add(p); // OK – Can write into it – it is now safe.
Product p = products.get(0); //NOT OK may be superclass of Product
```

- Producer extends and Consumer super (PECS) principle

Type Erasure & Reification

- **Type Erasure** – chosen in java as backward-compatibility alternative – information about generic type parameters is erased during compilation, and is NOT available in runtime – the generic type becomes compiled to its basic raw type:

  Collection<Product> products; --(runtime)--> Collection products;

  This design decision creates problems if we want to create generic type instance with **new**, or to convert to the generic type, or to check the generic type using **instanceof**.

- **Reification** – better alternative strategy, implemented in languages such as C++, Ada и Eiffel, using which the generic type information is accessible in runtime.

Generic Containers

- ❖Allow compile time type checking – earlier error detection
- ❖Remove unnecessary typecasting to more specific types – less ClassCastExceptions
- ❖Examples:

```
Collection <String> s = new ArrayList <String>();
Map <Integer, String> table = new HashMap <Integer, String>()
```

- ❖New for loop – for each element of a Collection :

```
for(String i: s) { System.out.println(i) }
```

Main Implementing Classes. Examples

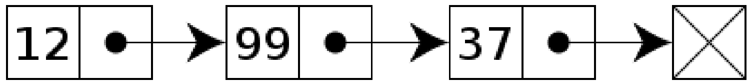
- Associative lists (dictionaries) – interface **Map**
- Comparing different implementations:
  - **HashMap**
  - **TreeMap**
  - **LinkedHashMap**
  - **WeakHashMap**
- Hashing.
- Cash implementations – **Reference, SoftReference, WeakReference и PhantomReference**
- Choosing a container implementation

Контейнерни класове и интерфейси в Java. Итератори.

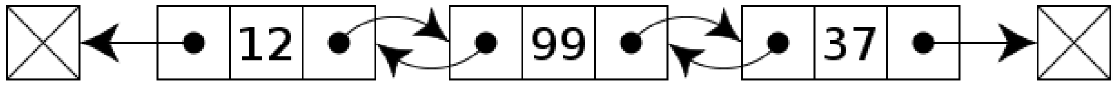
- Колекции – интерфейс **Collection**
- Списъци – интерфейс **List**, реализации – **ArrayList, LinkedList, ...**
- Множества – интерфейс **Set**, реализации – **HashSet, TreeSet, …**
- Асоциативни списъци – интерфейс **Map**, реализации – **HashMap, TreeMap, LinkedHashMap, WeakHashMap, …**
- Обхождане на колекция с итератор.
- Реализиране на структури от данни стек, опашка, дек – интерфейси **Queue** и **Dequeue**. Реализации.

Структури от данни

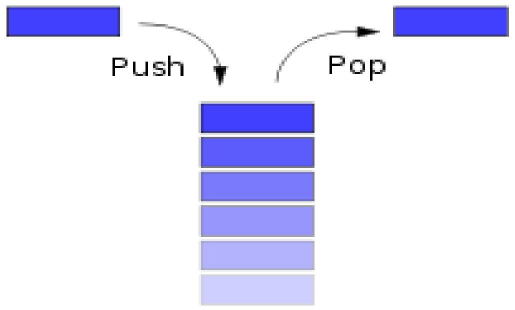
- Едно-свързан списък:



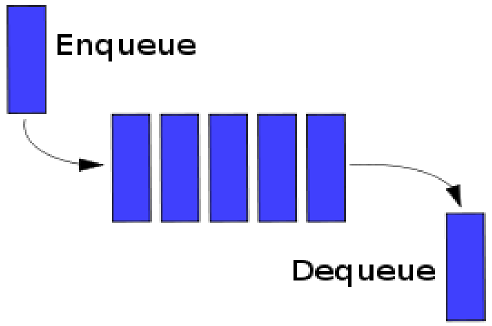
- Дву-свързан списък:



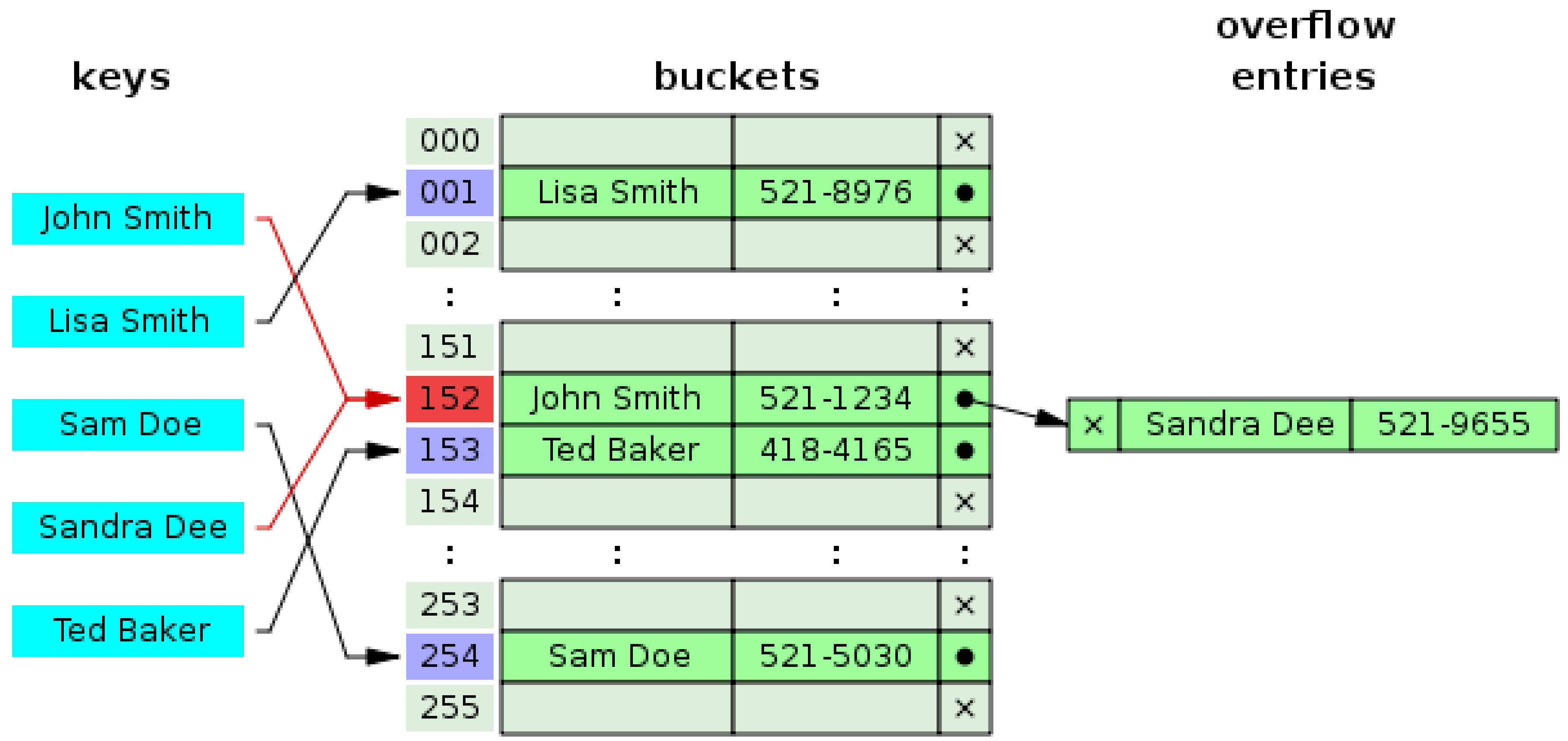
- Стек:



- Опашка:



Хеширане, хеш функции, хеш таблици



Resources

- Oracle Generics tutorial – https://docs.oracle.com/javase/tutorial/extra/generics/index.html

Annotations

Why Annotations?

- Java APIs often require boilerplate code for bootstrapping & maintenance.
- For example, in order to write SOAP-based web service with JAX-WS, you must provide paired interface and implementation. If we decorate java class methods using annotations, we can indicate which methods are meant to be exposed as web methods for remote access. This boilerplate code could be generated automatically by a tool based on annotations.
- REST services with JAX-RS can use annotations for bootstrapping and routing.
- Other APIs require configuration files to be maintained in parallel with code – e.g. JavaBeans require BeanInfo classes, Servlet-based web applications require web.xml deployment descriptor. It would less error-prone and more convenient to maintain this information as annotations in the program code, or combination of both approaches could be employed.

History of Declarative Programming in Java

- The Java platform always had various ad hoc annotation mechanisms:
  - **transient** modifier is an ad hoc annotation indicating that a field should be ignored during the serialization;
  - **@deprecated** javadoc tag is an ad hoc annotation suggesting that the method should no longer be used.
- Java 5 introduced a general purpose mechanism called metadata annotations, permitting to extend these declarative programming mechanisms by defining custom annotations and applying them in code.
- Since then a numerous java frameworks and libraries have been redesigned to take advantage of the flexibility and ease of maintenance that annotations in the code offer – e.g. JavaEE Servlet API, JAX-RS, JAXB, EJB, CDI, JPA, Bean Validation, Hibernate, Spring, and many more.

Java Annotations

- The annotations mechanism consists of:
  - syntax for declaring annotation types;
  - syntax for annotating declarations and (since java 1.8) usages of a type;
  - APIs for reading annotations,
  - class file representation for annotations accessible using reflection in runtime
  - an annotation processing tool (APT) that was integrated in javac since v1.6.
- Annotations do not directly affect program semantics, but they do affect the way programs are treated by tools and frameworks, which can generate new code and in turn affect the semantics of the program to be executed by JVM. Annotations can be read from source files, class files, or reflectively at run time.

Declarative Programming Using Annotations

- Java annotation is a form of syntactic metadata that can be added to the source code.
- Classes, methods, variables, parameters and Java packages may be annotated. Since Java 1.8 annotations can also be applied to any type use - new, casts, implements and throws clauses, etc.
- Like Javadoc tags, Java annotations are read from source files. They can complement javadoc tags. If the markup is intended to produce documentation, it should be javadoc tag; otherwise, annotation.
- Unlike Javadoc tags, annotations can also be embedded in and read from Java class files generated by the Java compiler. This allows annotations to be retained by the JVM at run-time and read using reflection.
- Some annotations are "meta"-annotations allowing to create new ones.

What the Java Annotations Can Be Used For?

- Annotations, as a form of metadata, provide data about a program that is not part of the program itself. Annotations do not directly effect the operation of the program code they annotate.
- Annotations have a number of usages – they provide:
  - Information for the compiler — Annotations can be used by the compiler to detect errors or suppress warnings.
  - Compile-time and deployment-time processing - tools and frameworks can process annotation information to generate code, XML files, and other types of files.
  - Runtime processing — some annotations are available to be examined at runtime using reflection mechanisms discussed in previous lessons.

Annotation Types Used by the Java Language

- **@Deprecated** – indicates that the marked element is deprecated and should no longer be used.
- **@Override** – informs the compiler that the element is meant to override an element declared in a superclass.
- **@SuppressWarnings({"unchecked", "deprecation"})** – tells the compiler to suppress specific warnings that it would otherwise generate.
- **@SafeVarargs** – when applied to a method or constructor, asserts that the code does not perform potentially unsafe operations on its varargs parameter.
- **@FunctionalInterface** – since Java 1.8, indicates that the type declaration is intended to be a functional interface, as defined by JLS

Example: Using Annotation Types in Java Code

-
  ```
  @Override
  @SuppressWarnings("unchecked")
  public <T> T getBean(Class<T> cls) {
      List<BeanDescriptor> beanDescriptors =
                beansByType.get(cls.getCanonicalName());
      if (beanDescriptors.size() != 1) {
            throw new BeanLookupException("There should be
                exactly one bean of type '" + cls + "', but " +
                beanDescriptors.size() + " found.");
      }
      return (T) getProxyInstance(cls, beanDescriptors.get(0));
  }
  ```

Annotations That Apply to Other Annotations (1)

- Annotations that apply to other annotations are called meta-annotations. There are several meta-annotation types defined in java.lang.annotation:
- **@Retention** – specifies how the marked annotation is stored:
  - RetentionPolicy.SOURCE – The marked annotation is retained only in the source level and is ignored by the compiler.
  - RetentionPolicy.CLASS – The marked annotation is retained by the compiler at compile time, but is ignored by the Java Virtual Machine (JVM).
  - RetentionPolicy.RUNTIME – The marked annotation is retained by the JVM so it can be used by the runtime environment.

Annotations That Apply to Other Annotations (2)

- **@Documented** – indicates that whenever the specified annotation is used those elements should be documented using the Javadoc tool.
- **@Target** – marks another annotation to restrict what kind of Java elements the annotation can be applied to:
  - ElementType.ANNOTATION_TYPE
  - ElementType.CONSTRUCTOR
  - ElementType.FIELD – field or property
  - ElementType.LOCAL_VARIABLE
  - ElementType.METHOD
  - ElementType.PACKAGE
  - ElementType.PARAMETER – method parameters
  - ElementType.TYPE – can be applied to any element of a class.

Annotations That Apply to Other Annotations (2)

- **@Inherited** – indicates that the annotation type can be inherited from the super class. (This is not true by default.) When the user queries the annotation type and the class has no annotation for this type, the class' superclass is queried for the annotation type. This annotation applies to class declarations only.

- @Repeatable – since Java 1.8 indicates that the marked annotation can be applied more than once to the same declaration or type use. E.g.:

```
@Schedules({@Schedule(dayOfMonth="last"),
           @Schedule(dayOfWeek="Fri", hour=23)})
public void doPeriodicCleanup() {
    //...
}
```

Example: Defining Repeatable Annotations

```
@Retention(RUNTIME)
@Target(METHOD)
@Repeatable(Schedules.class)
public @interface Schedule {
   String dayOfMonth() default "first";
   String dayOfWeek() default "Mon";
   int hour() default 12;
}
```

```
@Retention(RUNTIME)
@Target(METHOD)
public @interface Schedules {
     Schedule[] value();
}
```

Defining Class, Method, Parameter and Field annot.

(JSR 330: Dependency Injection for Java)

-
  ```
  @Target({ METHOD, CONSTRUCTOR, FIELD })
  @Retention(RUNTIME)
  @Documented
  public @interface Inject {}
  ```

-
  ```
  @Target(ANNOTATION_TYPE)
  @Retention(RUNTIME)
  @Documented
  public @interface Qualifier {}
  ```

-
  ```
  @Qualifier
  @Retention(RetentionPolicy.RUNTIME)
  @Target({FIELD, METHOD, TYPE, PARAMETER})
  public @interface JaxbRepository {}
  ```

Using @Inject with Custom Qualifier Annotation

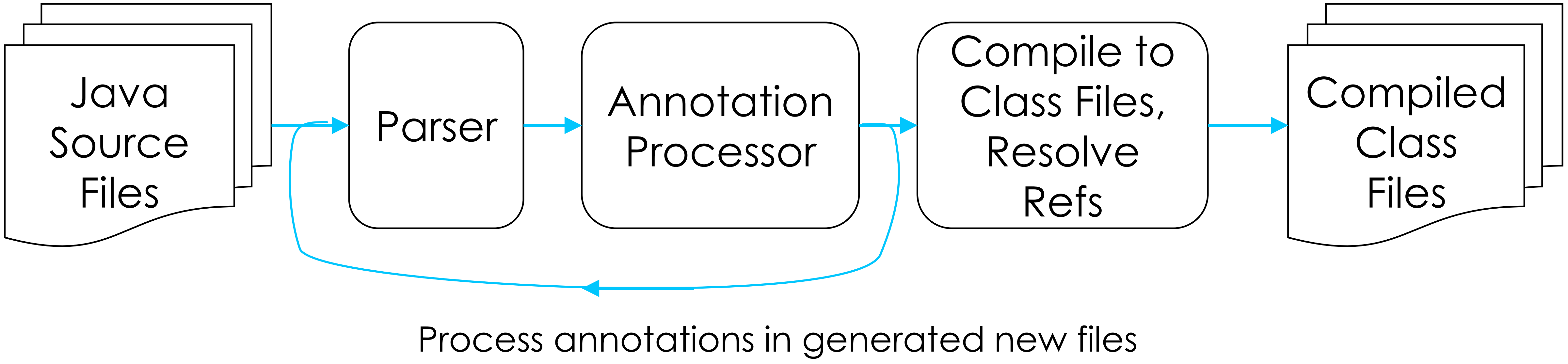
(JSR 330: Dependency Injection for Java)

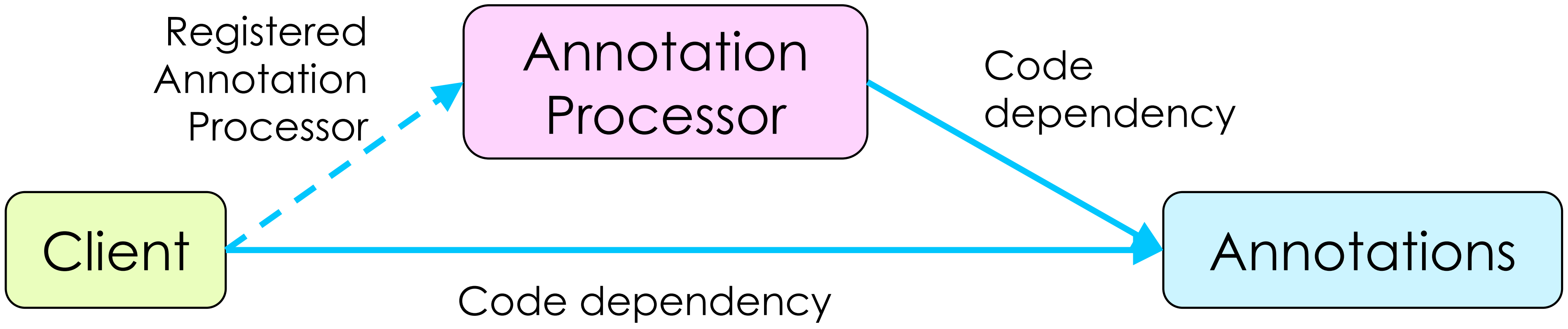
-
  ```
  @Component
  public class UserControllerImpl implements UserController {
      @Inject @JaxbRepository
      private UserRepository repo;

      public UserRepository getRepo() {
          return repo;
      }

      public void setRepo(UserRepository repo) {
          this.repo = repo;
      }
      //...
  }
  ```

Compile-time Annotation Processing (1)

- Tools and frameworks can process annotation information to generate additional source files (code), XML files, metadata, documentation, resources, and other types of files, based on annotations in the source code.
- Tutorial: https://www.baeldung.com/java-annotation-processing-builder
- It can only be used to generate new files, not to change existing ones.
- Lombok is a noteworthy exception – it uses annotation processing as a bootstrapping mechanism to include itself into the compilation process and modify the AST via some internal compiler APIs (not the intended purpose of annotations processing).
- Actively used by many Java libraries, to generate metaclasses in QueryDSL and JPA, to augment classes with boilerplate code in Lombok, etc.

Compile-time Annotation Processing (2)


Java Source Files
Parser
Annotation Processor
Compile to Class Files, Resolve Refs
Compiled Class Files
Process annotations in generated new files



Compile-time Annotation Processing (3)

- Package javax.annotation.processing - Facilities for declaring annotation processors and for allowing annotation processors to communicate with an annotation processing tool environment:
- Completion - A suggested completion for an annotation.
- Filer - This interface supports the creation of new files by an annotation processor.
- Messager - A Messager provides the way for an annotation processor to report error messages, warnings, and other notices.

Compile-time Annotation Processing (4)

- ProcessingEnvironment - An annotation processing tool framework will provide an annotation processor with an object implementing this interface so the processor can use facilities provided by the framework to write new files, report error messages, and find other utilities.
- Processor- The interface for an annotation processor.
- RoundEnvironment - An annotation processing tool framework will provide an annotation processor with an object implementing this interface so that the processor can query for information about a round of annotation processing.

Defining Custom Annotations

-
  ```java
  @Retention(RUNTIME)
  @Target(TYPE)
  @Inherited
  public @interface Repository {
      String value() default "";
  }
  ```

-
  ```java
  @Retention(RUNTIME)
  @Target(TYPE)
  @Inherited
  public @interface Scope {
      BeanScope value() default BeanScope.SINGLETON;
  }
  ```

-
  ```java
  public enum BeanScope { SINGLETON, PROTOTYPE }
  ```

Processing Annotations at Runtime (1)

- An annotation A is **directly present** on an element E if E has a RuntimeVisibleAnnotations or RuntimeVisibleParameterAnnotations or RuntimeVisibleTypeAnnotations attribute, and the attribute contains A.
- An annotation A is **indirectly present** on an element E if E has a RuntimeVisibleAnnotations or RuntimeVisibleParameterAnnotations or RuntimeVisibleTypeAnnotations attribute, and A 's type is repeatable, and the attribute contains exactly one annotation whose value element contains A and whose type is the containing annotation type of A 's type.

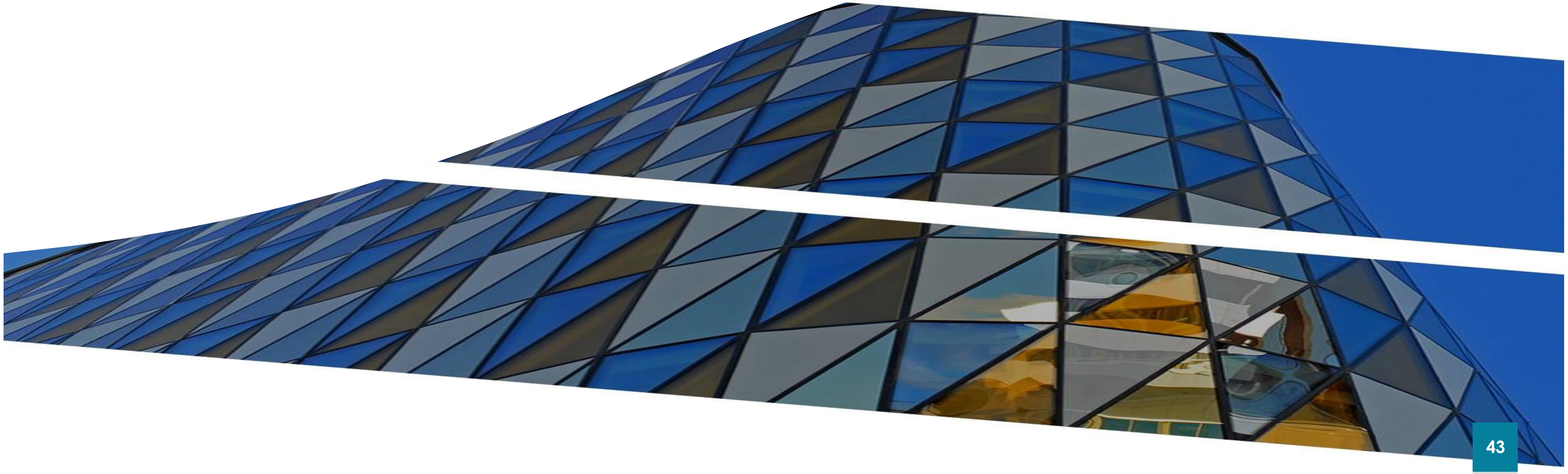
Processing Annotations at Runtime (2)

- An annotation A is **present** on an element E if either:
  - A is directly present on E; or
  - No annotation of A 's type is directly present on E, and E is a class, and A 's type is inheritable, and A is present on the superclass of E.
- An annotation A is **associated** with an element **E** if either:
  - A is directly or indirectly present on E; or
  - No annotation of A 's type is directly or indirectly present on E, and E is a class, and A's type is inheritable, and A is associated with the superclass of E.

Processing Annotations at Runtime (3)

Overview of kind of presence detected by different AnnotatedElement methods					
		Kind of Presence			
Method		Directly Present	Indirectly Present	Present	Associated
T	getAnnotation(Class<T>)			X	
Annotation[]	getAnnotations()			X	
T[]	getAnnotationsByType(Class<T>)				X
T	getDeclaredAnnotation(Class<T>)	X			
Annotation[]	getDeclaredAnnotations()	X			
T[]	getDeclaredAnnotationsByType(Class<T>)	X	X		

Java Reflection API



RTTI & Reflection

- **Runtime Type Information (RTTI)** allows you to discover and employ type information in runtime.
- The difference between **RTTI** and **reflection** is that with **RTTI**, the compiler **opens and examines the .class file at compile time**, while with **reflection**, the .class file is unavailable at compile time; it is **opened and examined by the runtime** environment. Examples:

– **JavaBeans** - Rapid Application Development (RAD) in an Application Builder Integrated Development Environment (IDE)

– Object serialization – Serializable interface

– **Remote Method Invocation (RMI)** - discovering class information at run time provides ability to create and execute objects on remote platforms, across the network.

– **Dynamic Proxies** – DI, Spring, AOP

Runtime Type Information (RTTI)

- The **Class** object, **Class.forName()**, **Type.class**, and **object.getClass( )**
- **Class** models all types in java – class, interface, array, primitive type, void (e.g. int.class, long.class, double.class, void.class, etc.)
- It allows writing flexible and generic utility methods and tools capable of processing (field, method, type variable, annotation, superclass and interface metadata reflection, instantiation (constructor invocation), method invocation, field data access, etc.) of different types of objects given as parameters to those methods and tools, and generally not known at the time of their writing.
- **T newInstance()** - creates a new instance of the class
- **T cast(Object obj)** - casts an object to the class or interface

Class: Names, Loader, Annotations

- **String getName()** - name of the type (class, interface, array, primitive type, void)
- **String getSimpleName()** - the simple name of the underlying class
- **String getCanonicalName()** - canonical name of class
- **ClassLoader getClassLoader()** - returns the class loader for the class.
- **<A extends Annotation>A getAnnotation(Class<A> annotationClass)** - annotation if present
- **Annotation[] getAnnotations()** - all annotations present on this type
- **<A extends Annotation> A[] getAnnotationsByType(Class<A> annotationClass)** – if its argument is a repeatable annotation type (JLS 9.6), it attempts to find one or more annotations of that type by "looking through" a container annotation.
- **Annotation[] getDeclaredAnnotations()** - directly present annotations
- **<A extends Annotation> A getDeclaredAnnotation(Class<A> annotationClass)**
- **<A extends Annotation> A[] getDeclaredAnnotationsByType(Class<A>)**
- **boolean isAnnotation()** - true if an annotation type.
- **boolean isAnnotationPresent(Class<? extends Annotation> annotatClass)**

Class: Constructors, Fields, Methods

- **Constructor<?>[] getConstructors()** - array of all public Constructors
- **Constructor<T> getConstructor(Class<?>... parameterTypes)** – public
- **Constructor<?>[] getDeclaredConstructors()** - all class constructors
- **Constructor<T> getDeclaredConstructor(Class<?>... parameterTypes)**
- **Field getField(String name)** - public member field
- **Field[] getFields()** - public member fields
- **Field getDeclaredField(String name)** - a field:public, private, package, protected
- **Field[] getDeclaredFields()** - all class fields: public, private, package, protected
- **Method getMethod(String name, Class<?>... parameterTypes)** - reflects the specified public member method
- **Method[] getMethods()** - reflects all public member method
- **Method getDeclaredMethod(String name, Class<?>... parameterTypes)**
- **Method[] getDeclaredMethods()** - all methods

Class: Members, Inner Classes,Types

- **Class<?>[] getClasses()** - public class and interface members of this Class
- **Class<?>[] getDeclaredClasses()** - all the class and interface members
- **Class<?> getDeclaringClass()** - if a member of another class
- **Class<?> getEnclosingClass()** - enclosing class of the underlying class
- **Constructor<?> getEnclosingConstructor()** - for local/anonymous class
- **Method getEnclosingMethod()** - for local/anonymous class
- **T[] getEnumConstants()** - elements of this enum class
- **Type[] getGenericInterfaces()** - Types representing the interfaces impl.
- **Type getGenericSuperclass()** - Type representing the superclass impl.
- **AnnotatedType[] getAnnotatedInterfaces()** - annotated superinterfaces
- **AnnotatedType getAnnotatedSuperclass()** - annotated superclass
- **Class<?> getComponentType()** - the component type of an array

Class: SuperClasses, Interfaces, Resources

- **Class<? super T> getSuperclass()** - the superclass of the entity
- **Class<?>[] getInterfaces()** - the interfaces implemented
- **int getModifiers()** - Java language modifiers for entity (public, package) - as int
- **Package getPackage()** - gets the package for this class.
- **URL getResource(String name**) - finds a resource with a given name
- **InputStream getResourceAsStream(String name)** - finds a resource with a given name.
- **ProtectionDomain getProtectionDomain()** - returns the ProtectionDomain of this class.
- **Object[] getSigners()** - gets the signers of this class
- **String toGenericString()** - Returns a string describing this Class, including information about modifiers and type parameters

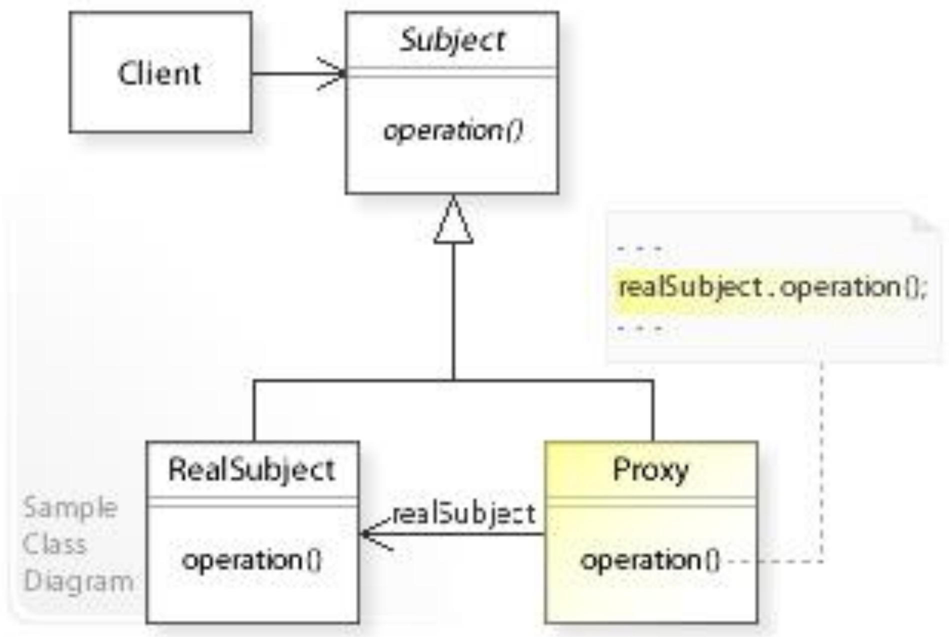
Class: Superclass, Type Vars, Is*

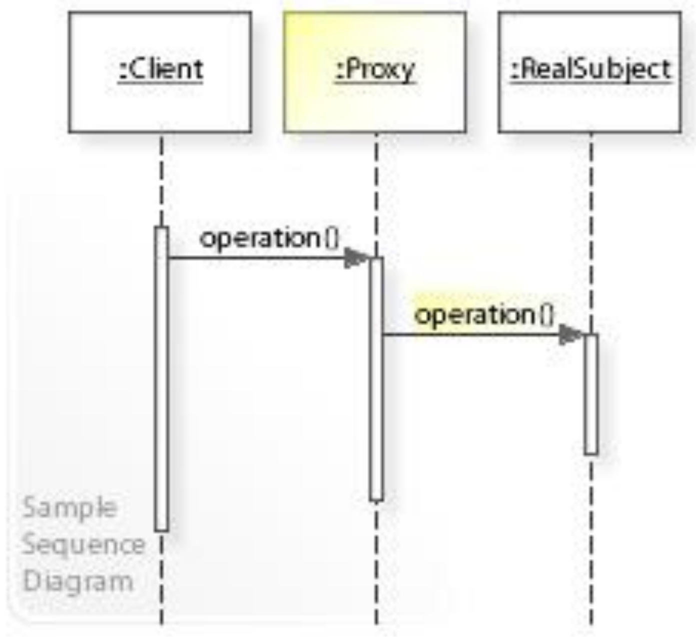
- **String getTypeName()** - an informative string for the name of this type.
- **TypeVariable<Class<T>>[] getTypeParameters()** - generic declaration GenericDeclaration TypeVariable objects
- **boolean isAnonymousClass()** - true if anonymous class.
- **boolean isArray()** - if an array class.
- **boolean isAssignableFrom(Class<?> cls)** -same/ superclass/ superinterface
- **boolean isEnum()** - if enum
- **boolean isInstance(Object obj)** – dynamic instanceof
- **boolean isInterface()** - if interface
- **boolean isLocalClass()** - if local class
- **boolean isMemberClass()** - if the underlying class is a member of this class
- **boolean isPrimitive()** - if the specified Class object represents a primitive type
- **boolean isSynthetic()** - if this class is a synthetic class

Example Using Reflection: Dynamic Proxy Design Pattern

- A proxy, in its most general form, is a class functioning as an **interface to something else**. A proxy is a wrapper or agent object that is being called by the client to access the real serving object behind the scenes. Use of the proxy can simply be forwarding to the real object, or can provide **additional logic**.
- In the proxy, **extra functionality can be provided**, for example caching when operations on the real object are resource intensive, or checking preconditions before operations on the real object are invoked.
- For the client, usage of a proxy object is similar to using the real object, because **both implement the same interface**.
- **Dynamic proxies** can be generated through reflection of bean methods, providing additional functionality

Dynamic Proxy Design Pattern





Inversion of Control (IoC) & Dependency Injection (DI) (Recap.)

- Components provide necessary metadata (specification) about their supported interfaces (contracts) which allows their dynamic discovery and assembly at runtime.
- There is no dependency on concrete component implementations (only on supported interfaces), which allows to be developed in parallel with their clients, and to be swapped with improved implementations without changing the existing code of client components.
- This is usually done using the principle of Inversion of Control (IoC) implemented either as Dependency Injection (DI - e.g. CDI) or a programmatic lookup (e.g. JNDI).

Exercise: Building Custom DI Framework

- **Goal:** To build an initial prototype of custom Dependency Injection (DI) Framework using JSR 330: Dependency injection for Java, and custom annotations:
  - for bean (component) stereotypes: @Component, @Repository, @Service, following the principles of Domain Driven Design (DDD)
  - for bean scope declaration – initially only two bean scopes will be supported: SINGLETON and PROTOTYPE
- The framework should be implemented using Java Dynamic Proxy implementation, and using reflection mechanisms at runtime to discover annotations and inject dependencies.

Resources

- Jakob Jenkov’s Java Reflection Tutorial – http://tutorials.jenkov.com/java-reflection/index.html

Exception handling

Where to Find the Code?

Java Web Development projects and examples are available @ GitHub:

https://github.com/iproduct/course-java-fd

Exception Handling in Java

- Obligatory exception handling in Java → secure and reliable code
- Separation of concerns: business logic from exception handling code
- Class **Throwable** → classes Error и Exception
- Generating Exceptions – keyword **throw**
- Exception handling:
  - **try – catch – finally** block
  - Delegating the handling to the caller method - **throws**

Try-Catch-Finally Block

- Operator **try** for demarcation of un-reliable code, multiple **catch** blocks for catching exceptions, and **finally** clause for garateed clean-up in the end:
- **try {**
-     // code that can throw Exceptions: Ex1, Ex2, …
- **} catch(Ex1 ex) {** // excuted only when Ex1 is thrown
-     // handling problem 1 (Ex1)
- **} catch(Ex2 ex) {** // excuted only when Ex2 is thrown
-     // handling problem 2 (Ex2)
- **} finally {**
-     // executed always independently of whether an exception was thrown or not
- **}**
-

Exception Handling in Java - II

- Implementing custom exceptions
- Using stack trace
- Unchecked exceptions extending **RuntimeException**
- Guaranteed completion using **finally**

Novelties in Exception Handling since Java 7

- Multi-**catch** clause:

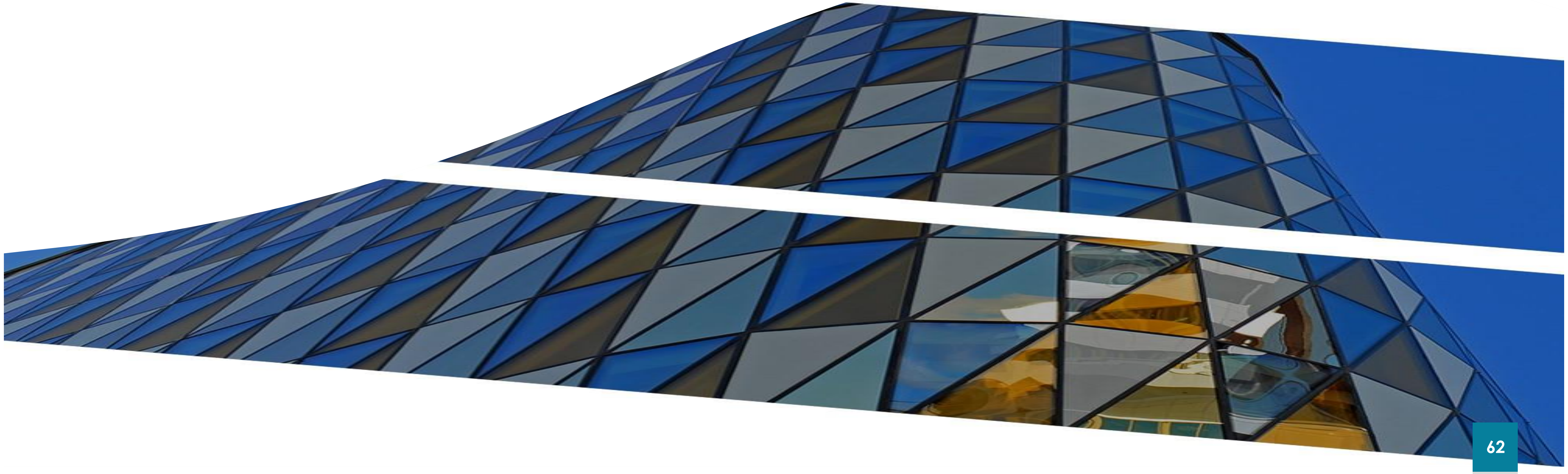
```
• catch (Exception1|Exception2 ex) {
•     ex.printStackTrace();
• }
```

- Program block **try-with-resources**

```
• String readInvoiceNumber(String myfile) throws IOException
  {
•     try (BufferedReader input = new
•                   BufferedReader(new
•                       FileReader(myfile))) {
•         return input.readLine();
•     }
• }
```

Java 8 Stream API

Practical Exercises – Functional Programming Koans



Agenda for This Session

- Fundamentals
- Functional interfaces
- Method references
- Constructor references

Новости в Java™ 8

- Ламбда изрази и поточно програмиране – пакети **java.util.function** и **java.util.stream**)
- Референции към методи
- Методи по подразбиране и статични методи в интерфейси – множествено наследяване на поведение в Java 8
- Функционално програмиране в Java 8 с използване на монади (напр. Optional, Stream) – предимства, начин на реализация, основни езикови идиоми, примери

Функционални интерфейси в Java™ 8

- Функционален интерфейс = интерфейс с един абстрактен метод
  SAM (Single Abstract Method) – @FunctionalInterface

- Примери за функционални интерфейси в Java 8:

```
public interface Comparator<T> {
    int compare(T o1, T o2);
}
public interface ActionListener extends EventListener {
    public void actionPerformed(ActionEvent e);
}
public interface Runnable {
    public void run();
}
public interface Callable<V> {
    V call() throws Exception;
}
```

Ламбда изрази – пакет java.util.function

Примери:

```
(int x, int y) -> x + y

() -> 42

(a, b) -> a * a + b * b;

(String s) -> { System.out.println(s); }

book -> book.getAuthor().fullName()

voter -> voter.getAge() >= legalAgeOfVoting

(person1, person2) -> person1.getAge() - person2.getAge()

(song1, song2) -> song1.getArtist().compareTo(song2.getArtist())
```

Правила за форматирне на ламбда изрази

- **Ламбда изразите (функциите)** могат да имат произволен брой **параметри**, които се ограждат в скоби, разделят се със запетаи и могат да имат или не деклариран тип (ако нямат - типът им се извежда от **контекста на използване = target typing**). Ако са само с един параметър, то скобите не са задължителни.

- **Тялото на ламбда изразите** се състои от произволен езикови конструкции (statements), разделени с **;** и заградени във фигурни скоби. Ако имаме само една езикова конструкция – израз то използването на фигурни скоби не е необходимо – в този случай стойността на израза автоматично се връща като стойност на функцията.

Пакет java.util.function

- **Predicate<T>** – предикат = булев израз представящ свойство на обекта подавано като аргумент
- **Function<A,R>**: функция която приема като аргумент **A** и го трансформира в резултат **R**
- **Supplier<T>** – с помощта на **get()** метод всеки път връща инстанция (обект) – фабрика за обекти
- **Consumer<T>** – приема аргумент (метод **accept()**) и изпълнява действие върху него
- **UnaryOperator<T>** – оператор с един аргумент **T -> T**
- **BinaryOperator<T>** – бинарен оператор **(T, T) -> T**

Поточно програмиране (1)

Примери:

```
books.stream().map(book ->
    book.getTitle()).collect(Collectors.toList());
books.stream()
       .filter(w -> w.getDomain() == PROGRAMMING)
       .mapToDouble(w -> w.getPrice()) .sum();
document.getPages().stream()
     .map(doc -> Documents.characterCount(doc))
     .collect(Collectors.toList());
document.getPages().stream()
     .map(p -> pagePrinter.printPage(p))
     .forEach(s -> output.append(s));
```

Източник: AdoptOpenJDK/lambda-tutorial (https://github.com/AdoptOpenJDK/lambda-tutorial) Автор: Graham Allan, Лиценз: GNU GENERAL PUBLIC LICENSE v2

Поточно програмиране (2)

Примери:

```
document.getPages().stream()
    .map(page -> page.getContent())
    .map(content -> translator.translate(content))
    .map(translated -> new Page(translated))
    .collect(Collectors.collectingAndThen(
        Collectors.toList(),
        pages -> new
    Document(translator.translate(document.getTitle()), pages)));
```

Референции към методи

- Статични методи на клас – Class::staticMethod
- Методи на конкретни обектни инстанции – object::instanceMethod
- Методи на инстанции реферирани чрез класа – Class::instanceMethod
- Конструктори на обекти от даден клас – Class::new

```
Comparator<Person> namecomp = Comparator.comparing(Person::getName);

Arrays.stream(pageNumbers).map(doc::getPageContent).forEach(Printers::print);

pages.stream().map(Page::getContent).forEach(Printers::print);
```

Статични и Default методи в интерфейси

- Методите с реализация по подразбиране в интерфейс са известни още като virtual extension methods или defender methods, защото дават възможност интерфейсите да бъдат разширявани, без това да води до невъзможност за компилация на вече съществуващи реализации на тези интерфейси (което би се получило ако старите реализации не имплементират новите абстрактни методи).
- Статичните методи дават възможност за добавяне на помощни (utility) методи – например factory методи директно в интерфейсите които ги ползват, вместо в отделни помощни класове (напр. Arrays, Collections).

Пример за default и static методи в интерфейс

- @FunctionalInterface

```
interface Event {
    Date getDate();
    default String getDateFormatted() {
            return String.format("%1$td.%1$tm.%1$tY", getDate());
    }
    public static <T, U extends Comparable<? super U>>
    Comparator<T> comparing(Function<T, U> getKey) {
        return (c1, c2) -> getKey.apply(c1).compareTo(getKey.apply(c2));
    }
}
Event current = () -> new Date();
System.out.println(current.getDateFormatted());
```

Функционално програмиране и монади

- Понятие за **монада** във функционалното програмиране (теория на категориите) – **Монадата** е множество от три елемента:

– Параметризиран тип **M<T>**

– "**unit**" функция: **T -> M<T>**

– "**bind**" операция: **bind(M<T>, f:T -> M<U>) -> M<U>**

- В Java 8 пример за монада е класът **java.util.Optional<T>**

Параметризиран тип: **Optional<T>**

– "**unit**" функции: **Optional<T> of(T value) , Optional<T> ofNullable(T value)**

– "**bind**" операция: **Optional<U> flatMap(Function<? super T,Optional<U>> mapper)**

Exercise: Java 8 Functional Programming Koans

Available @GitHub: https://github.com/iproduct/course-java-web-development/tree/master/lambda-tutorial-master

1. Read carefully the JavaDoc for the unit tests stating the problem to solve: Exercise_1_Test.java, Exercise_2_Test.java, Exercise_3_Test.java, Exercise_4_Test.java and Exercise_5_Test.java
2. Fill the code in place of comments like: // [your code here]
3. Run the unit tests to check if your proposed solution is correct. If not return to step 1.

Resources

- Oracle tutorial – lambda expressions - http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html

- Java SE 8: Lambda Quick Start - http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/Lambda-QuickStart/index.html

- OpenJDK Lambda Tutorial - https://github.com/AdoptOpenJDK/lambda-tutorial

Thank’s for Your Attention!

Trayan Iliev

IPT – Intellectual Products & Technologies

http://iproduct.org/

http://robolearn.org/

https://github.com/iproduct

https://twitter.com/trayaniliev

https://www.facebook.com/IPT.EACAD