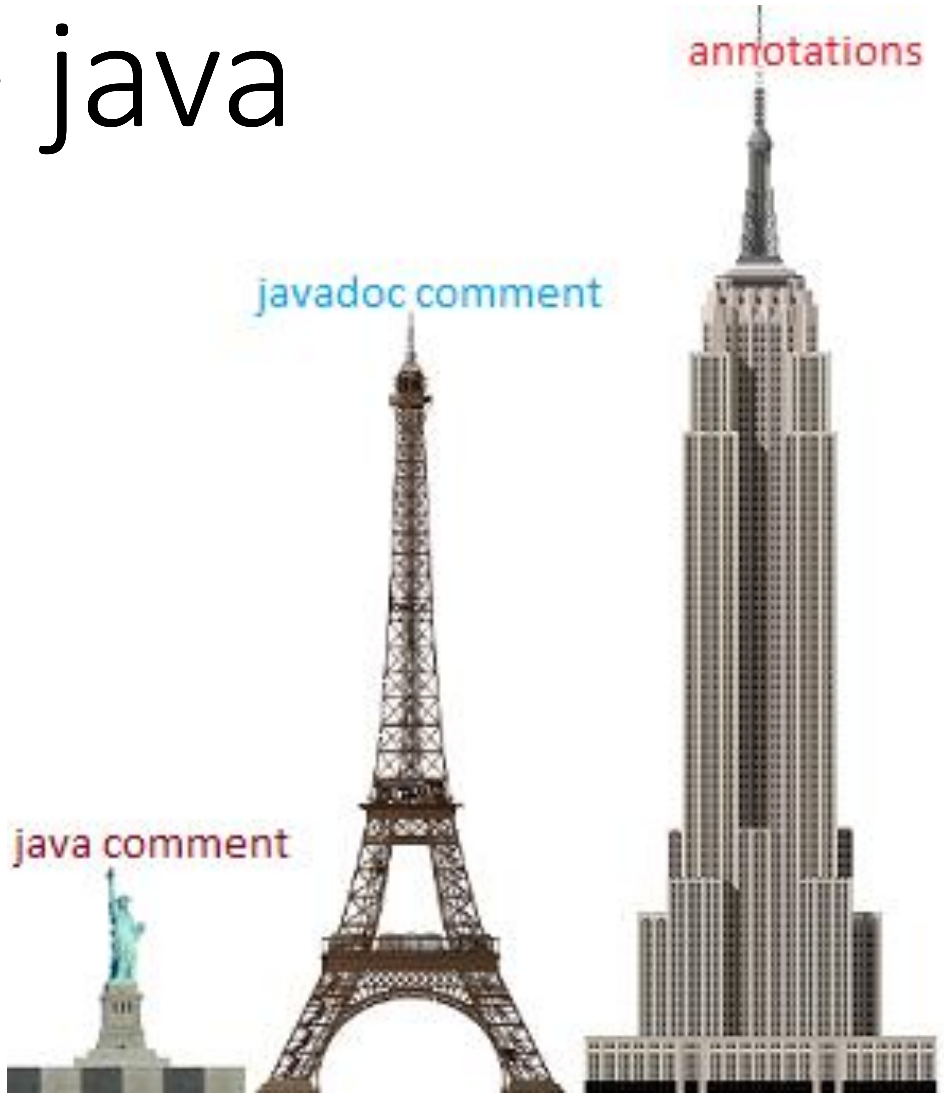
Java Annotation + java Reflection



Anotacijos

- One main difference with annotation is it can be carried over to runtime and the other two stops with compilation level. Annotations are not only comments, it brings in new possibilities in terms of automated processing.
- *Annotations*, a form of metadata, provide data about a program that is not part of the program itself. Annotations have no direct effect on the operation of the code they annotate.

Anotacijų rūšys

- **Class**

  When the annotation value is given as ‘class’ then this annotation will be compiled and included in the class file.

- **Runtime**

  The value name itself says, when the retention value is ‘Runtime’ this annotation will be available in JVM at runtime. We can write custom code using reflection package and parse the annotation. I have give an example below.

- **Source**

  This annotation will be removed at compile time and will not be available at compiled class.

Anotacijų paskirtis

Annotations have a number of uses, among them:

Information for the compiler — Annotations can be used by the compiler to detect errors or suppress warnings.

Compile-time and deployment-time processing — Software tools can process annotation information to generate code, XML files, and so forth.

Runtime processing — Some annotations are available to be examined at runtime.

Kada nepatartina naudoti Anotacijų

- Do not over use annotation as it will pollute the code.
- It is better not to try to change the behaviour of objects using annotations. There is sufficient constructs available in oops and annotation is not a better mechanism to deal with it.
- We should not what we are parsing. Do not try to over generalize as it may complicate the underlying code. Code is the real program and annotation is meta.
- Avoid using annotation to specify environment / application / database related information.

Anotacijos kurios yra naudojamos kitoms anotacijoms (meta-anotacijos).

Annotations that apply to other annotations are called *meta-annotations*. There are several meta-annotation types defined in java.lang.annotation.

@Retention @Retention annotation 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.

@Documented **@Documented** annotation indicates that whenever the specified annotation is used those elements should be documented using the Javadoc tool. (By default, annotations are not included in Javadoc.)

```
@Documented
@Retention(value=RUNTIME)
@Target(value=ANNOTATION_TYPE)
public @interface Documented
```

```
@Documented
public @interface Team {
int teamId();
String teamName();
String teamLead() default "[unassigned]";
String writeDate(); default "[unimplemented]"; }
```

```
... a java class ...
@Team( teamId = 73, teamName = "Rambo Mambo",
teamLead = "Yo Man", writeDate = "3/1/2012" )
public static void readCSV(File inputFile) { ...
} ... java class continues ...
```

@Target @Target annotation marks another annotation to restrict what kind of Java elements the annotation can be applied to. A target annotation specifies one of the following element types as its value:

- ElementType.ANNOTATION_TYPE can be applied to an annotation type.
- ElementType.CONSTRUCTOR can be applied to a constructor.
- ElementType.FIELD can be applied to a field or property.
- ElementType.LOCAL_VARIABLE can be applied to a local variable.
- ElementType.METHOD can be applied to a method-level annotation.
- ElementType.PACKAGE can be applied to a package declaration.
- ElementType.PARAMETER can be applied to the parameters of a method.
- ElementType.TYPE can be applied to any element of a class.

```
@Target({ElementType.FIELD, ElementType.METHOD, ElementType.TYPE})
  public @interface Bogus {
     ...
  }
```

Repeating Annotations Java SE 8+ versija

- There are some situations where you want to apply the same annotation to a declaration or type use. As of the Java SE 8 release, *repeating annotations* enable you to do this.
- For example, you are writing code to use a timer service that enables you to run a method at a given time or on a certain schedule, similar to the UNIX cron service. Now you want to set a timer to run a method, doPeriodicCleanup, on the last day of the month and on every Friday at 11:00 p.m. To set the timer to run, create an @Schedule annotation and apply it twice to the doPeriodicCleanup method. The first use specifies the last day of the month and the second specifies Friday at 11p.m., as shown in the following code example:

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

Step 1: Declare a Repeatable Annotation Type

The annotation type must be marked with the @Repeatable meta-annotation. The following example defines a custom @Schedule repeatable annotation type:

```
import java.lang.annotation.Repeatable;

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

The value of the @Repeatable meta-annotation, in parentheses, is the type of the container annotation that the Java compiler generates to store repeating annotations. In this example, the containing annotation type is Schedules, so repeating @Schedule annotations is stored in an @Schedules annotation.

Applying the same annotation to a declaration without first declaring it to be repeatable results in a compile-time error.

Step 2: Declare the Containing Annotation Type

The containing annotation type must have a value element with an array type. The component type of the array type must be the repeatable annotation type. The declaration for the Schedules containing annotation type is the following:

```
public @interface Schedules {
        Schedule[] value();
 }
```

Retrieving Annotations

There are several methods available in the Reflection API that can be used to retrieve annotations. The behavior of the methods that return a single annotation, such as AnnotatedElement.getAnnotationByType(Class<T>), are unchanged in that they only return a single annotation if one annotation of the requested type is present. If more than one annotation of the requested type is present, you can obtain them by first getting their container annotation. In this way, legacy code continues to work. Other methods were introduced in Java SE 8 that scan through the container annotation to return multiple annotations at once, such as AnnotatedElement.getAnnotations(Class<T>). See the AnnotatedElement class specification for information on all of the available methods.

Design Considerations

When designing an annotation type, you must consider the cardinality of annotations of that type. It is now possible to use an annotation zero times, once, or, if the annotation's type is marked as @Repeatable, more than once. It is also possible to restrict where an annotation type can be used by using the @Target meta-annotation. For example, you can create a repeatable annotation type that can only be used on methods and fields. It is important to design your annotation type carefully to ensure that the programmer using the annotation finds it to be as flexible and powerful as possible.

Annotation Types Used by the Java Language

The predefined annotation types defined
in java.lang are @Deprecated, @Override, and @SuppressWarnings.

@Deprecated @Deprecated annotation indicates that the marked element is *deprecated* and should no longer be used. The compiler generates a warning whenever a program uses a method, class, or field with the @Deprecated annotation. When an element is deprecated, it should also be documented using the Javadoc @deprecated tag, as shown in the following example. The use of the at sign (@) in both Javadoc comments and in annotations is not coincidental: they are related conceptually. Also, note that the Javadoc tag starts with a lowercase *d* and the annotation starts with an uppercase *D*.

What are Java Annotations?

Annotations is a new feature from Java 5. Annotations are a kind of comment or meta data you can insert in your Java code. These annotations can then be processed at compile time by pre-compiler tools, or at runtime via Java Reflection. Here is an example of class annotation:

```
@MyAnnotation(name="someName", value = "Hello World")
public class TheClass { }
```

The class TheClass has the annotation @MyAnnotation written ontop. Annotations are defined like interfaces. Here is the MyAnnotation definition:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface MyAnnotation {
    public String name();
    public String value();
}
```

Uses of Reflection

- Reflection is commonly used by programs which require the ability to examine or modify the runtime behavior of applications running in the Java virtual machine.
- **This is a relatively advanced feature and should be used only by developers who have a strong grasp of the fundamentals of the language.**
- With that caveat in mind, reflection is a powerful technique and can enable applications to perform operations which would otherwise be impossible.

Extensibility Features

An application may make use of external, user-defined classes by creating instances of extensibility objects using their fully-qualified names.

Class Browsers and Visual Development Environments

A class browser needs to be able to enumerate the members of classes. Visual development environments can benefit from making use of type information available in reflection to aid the developer in writing correct code.

Debuggers and Test Tools

Debuggers need to be able to examine private members on classes. Test harnesses can make use of reflection to systematically call a discoverable set APIs defined on a class, to insure a high level of code coverage in a test suite.

Drawbacks of Reflection

Reflection is powerful, but should not be used indiscriminately. If it is possible to perform an operation without using reflection, then it is preferable to avoid using it. The following concerns should be kept in mind when accessing code via reflection.

Performance Overhead

Because reflection involves types that are dynamically resolved, certain Java virtual machine optimizations can not be performed. Consequently, reflective operations have slower performance than their non-reflective counterparts, and should be avoided in sections of code which are called frequently in performance-sensitive applications.

Security Restrictions

Reflection requires a runtime permission which may not be present when running under a security manager. This is in an important consideration for code which has to run in a restricted security context, such as in an Applet.

Exposure of Internals

Since reflection allows code to perform operations that would be illegal in non-reflective code, such as accessing private fields and methods, the use of reflection can result in unexpected side-effects, which may render code dysfunctional and may destroy portability. Reflective code breaks abstractions and therefore may change behavior with upgrades of the platform.

Java klasės anotacija

- Annotations is a new feature from Java 5. Annotations are a kind of comment or meta data you can insert in your Java code. These annotations can then be processed at compile time by pre-compiler tools, or at runtime via Java Reflection. Here is an example of class annotation:
- @MyAnnotation(name="someName", value = "Hello World")
- public class TheClass {
- }
- The class TheClass has the annotation @MyAnnotation written ontop. Annotations are defined like interfaces. Here is the MyAnnotation definition:
- @Retention(RetentionPolicy.RUNTIME)
- @Target(ElementType.TYPE)
- public @interface MyAnnotation {
-     public String name();
-     public String value();
- }

@Target

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Target;
@Target({ElementType.METHOD})
public @interface MyAnnotation {
String value(); }
```

Creating Your Own Annotations

```
@interface MyAnnotation {
    String   value();
    String   name();
    int      age();
    String[] newNames();
}
```

This example defines an annotation called `MyAnnotation` which has four elements.

Notice that each element is defined similarly to a method definition in an interface. It has a data type and a name. You can use all primitive data types as element data types. You can also use arrays as data type. You cannot use complex objects as data type.

To use the above annotation, you do like this:

Creating Your Own Annotations

```
@MyAnnotation(
    value="123",
    name="Jakob",
    age=37,
    newNames={"Jenkov", "Peterson"}
)
public class MyClass {


}
```

Element Default Values

```
@interface MyAnnotation {
   String   value() default "";
   String   name();
   int      age();
   String[] newNames();
}
```

```
@MyAnnotation( name="Jakob",
age=37,
newNames={"Jenkov","Peterson}
)
public class MyClass { }
```

@FunctionalInterface

- @FunctionalInterface annotation (available since Java 8) informs the Java compiler that given interface is intended to be a functional interface. The functional interface is an interface with exactly one abstract method (not counting default methods and abstract methods overriding methods of Object class) and which can be used together with lambda expressions. If the interface annotated with @FunctionalInterface contains different number of abstract methods than one, the Java compiler raises an error. Although using this annotation is not necessary (omitting it does not cause any compilation errors), it is very useful to detect possible issues when doing bigger modifications or refactoring.

- In the example below removing existing method test() or adding a new abstract method (except the ones mentioned above) would result in a compilation error:

```java
@FunctionalInterface
public interface MyPredicate<T> {
    boolean test(T data);
}
```

How to Parse Annotation

```
package com.javapapers.annotations;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
@Retention(RetentionPolicy.RUNTIME)
public @interface Developer { String value(); }
```

```
package com.javapapers.annotations;
public class BuildHouse {
     @Developer ("Alice")
     public void aliceMethod() {
          System.out.println("This method is written by
          Alice");
     }
     @Developer ("Popeye")
     public void buildHouse() {
          System.out.println("This method is written by
          Popeye");
     }
```

```java
package com.javapapers.annotations;
import java.lang.annotation.Annotation;
import java.lang.reflect.Method;
public class TestAnnotation {
       public static void main(String args[]) throws
       SecurityException,    ClassNotFoundException {
       for (Method method :
       Class.forName("com.javapapers.annotations.BuildHouse").getMethods
())
       {
// checks if there is annotation present of the given type Developer
if (method
.isAnnotationPresent(com.javapapers.annotations.Developer.class)) {
try {
// iterates all the annotations available in the method
for (Annotation anno : method.getDeclaredAnnotations()) {
       System.out.println("Annotation in Method '" + method + "' : " +
anno); Developer a = method.getAnnotation(Developer.class);
if ("Popeye".equals(a.value())) { System.out.println("Popeye the sailor
man! " + method); } } } catch (Throwable ex) { ex.printStackTrace(); } }
} } }
```

Serialization

In computer science, in the context of data storage, **serialization** is the process of translating data structures or object state into a format that can be stored (for example, in a file or memory buffer, or transmitted across a network connection link) and reconstructed later in the same or another computer environment.[1] When the resulting series of bits is reread according to the serialization format, it can be used to create a semantically identical clone of the original object. For many complex objects, such as those that make extensive use of references, this process is not straightforward. Serialization of object-oriented objects does not include any of their associated methods with which they were previously inextricably linked. This process of serializing an object is also called marshalling an object.[2] The opposite operation, extracting a data structure from a series of bytes, is **deserialization** (which is also called **unmarshalling**).

Demonstrate serialVersionUID

- Initial class to be serialized has a serialVersionUID as 1L. (Pakeitus serialD kitu, pavyzdžiui duomenis nuskaitant sekantį kartą iš failo išmes klaidą, kad neatitinka ID...

```
import java.io.Serializable;
public class Lion implements Serializable {
    private static final long serialVersionUID = 1L;
    private String sound; public Lion(String sound) {
        this.sound = sound;
    }
    public String getSound() { return sound; } }
public class SerialVersionUIDTest {
public static void main(String args[]) throws
IOException, ClassNotFoundException {
Lion leo = new Lion("roar");
// serialize
System.out.println("Serialization done.");
FileOutputStream fos = new
FileOutputStream("serial.out");
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeObject(leo);
// deserialize FileInputStream fis = new
FileInputStream("serial.out");
ObjectInputStream ois = new ObjectInputStream(fis);
Lion deserializedObj = (Lion) ois.readObject();
System.out.println("DeSerialization done. Lion: " +
deserializedObj.getSound()); } }
```

Serialization Proxy Pattern

- So in this way, we can serialize super class state even though it’s not implementing Serializable interface. This strategy comes handy when the super class is a third-party class that we can’t change.
- Java Serialization comes with some serious pitfalls such as;
- The class structure can’t be changed a lot without breaking the serialization process. So even though we don’t need some variables later on, we need to keep them just for backward compatibility.

Serialization Proxy Pattern

- Serialization causes huge security risks, an attacker can change the stream sequence and cause harm to the system. For example, user role is serialized and an attacker change the stream value to make it admin and run malicious code.
- Serialization Proxy pattern is a way to achieve greater security with Serialization. In this pattern, an inner private static class is used as a proxy class for serialization purpose. This class is designed in the way to maintain the state of the main class. This pattern is implemented by properly implementing *readResolve()* and *writeReplace()* methods.

```java
public class Data implements Serializable{

    private static final long serialVersionUID = 2087368867376448459L;

    private String data;

    public Data(String d){
        this.data=d;
    }

    public String getData() {
        return data;
    }

    public void setData(String data) {
        this.data = data;
    }

    @Override
    public String toString(){
        return "Data{data="+data+"}";
    }
```

```java
//serialization proxy class
private static class DataProxy implements Serializable{

    private static final long serialVersionUID = 8333905273185436744L;

    private String dataProxy;
    private static final String PREFIX = "ABC";
    private static final String SUFFIX = "DEFG";

    public DataProxy(Data d){
        //obscuring data for security
        this.dataProxy = PREFIX + d.data + SUFFIX;
    }

    private Object readResolve() throws InvalidObjectException {
        if(dataProxy.startsWith(PREFIX) && dataProxy.endsWith(SUFFIX)){
        return new Data(dataProxy.substring(3, dataProxy.length() -4));
        }else throw new InvalidObjectException("data corrupted");
    }

}

//replacing serialized object to DataProxy object
private Object writeReplace(){
    return new DataProxy(this);
}

private void readObject(ObjectInputStream ois) throws InvalidObjectException{
    throw new InvalidObjectException("Proxy is not used, something fishy");
}
```

SerializationProxyTest.java

```java
package com.journaldev.serialization.proxy;

import java.io.IOException;

import com.journaldev.serialization.SerializationUtil;

public class SerializationProxyTest {

    public static void main(String[] args) {
        String fileName = "data.ser";

        Data data = new Data("Pankaj");

        try {
            SerializationUtil.serialize(data, fileName);
        } catch (IOException e) {
            e.printStackTrace();
        }

        try {
            Data newData = (Data) SerializationUtil.deserialize(fileName);
            System.out.println(newData);
        } catch (ClassNotFoundException | IOException e) {
            e.printStackTrace();
        }
    }

}
```

- Both Data and DataProxy class should implement Serializable interface.
- DataProxy should be able to maintain the state of Data object.
- DataProxy is inner private static class, so that other classes can’t access it.
- DataProxy should have a single constructor that takes Data as argument.
- Data class should provide writeReplace() method returning DataProxy instance. So when Data object is serialized, the returned stream is of DataProxy class. However DataProxy class is not visible outside, so it can’t be used directly.
- DataProxy class should implement readResolve() method returning Data object. So when Data class is deserialized, internally DataProxy is deserialized and when it’s readResolve() method is called, we get Data object.
- Finally implement readObject() method in Data class and throw InvalidObjectException to avoid hackers attack trying to fabricate Data object stream and parse it.

Singleton

- Singleton classes represent objects for which only one single instance should exist.The important question here is this: **will any actual harm come to the system** if more than 1 object is created? If the answer is "no" (and it usually is), then there's **no need whatsoever** to use a singleton. If the answer is "yes", then you will need to consider it. The main point is that a singleton should only be used if it's really necessary.
- Here's an article from Oracle describing them. This article, along with many others, reports many subtle problems with singletons. You should likely exercise care when using this pattern.

Example 1

This is the preferred style of implementing singletons. It uses a simple enumeration. It has no special needs for serialization, and is immune to clever attacks.

```
/** Preferred style for singletons. */
public enum SantaClaus {
        INSTANCE;
        /**Add some behavior to the object. */
        public void distributePresents(){ //elided } /** Demonstrate use of
        SantaClaus. */
        public static void main(String... aArgs){
                SantaClaus fatGuy = SantaClaus.INSTANCE;
                fatGuy.distributePresents();
        //doesn't compile :
        //SantaClaus fatGuy = new SantaClaus(); } }
```

Example 2

Here's an alternate style. If you decide that the class should no longer be a singleton, you may simply change the implementation of getInstance.

```
public final class Universe {

        public static Universe getInstance()
        {
                return fINSTANCE;
        }
        // PRIVATE /** * Single instance created upon class loading. */
        private static final Universe fINSTANCE = new Universe();
        /** * Private constructor prevents construction outside this class. */
        private Universe() { //..elided }
}
```

Example 3

If the above style of singleton is to be Serializable as well, then you must add a readResolve method.

```
import java.io.*;
public final class EasterBunny implements Serializable {
public static EasterBunny getInstance() { return fINSTANCE; }
// PRIVATE /** * Single instance created upon class loading. */
private static final EasterBunny fINSTANCE = new EasterBunny();
/** * Private constructor prevents construction outside this class. */
private EasterBunny() { //..elided }
/** * If the singleton implements Serializable, then this * method must
be supplied. */
private Object readResolve() throws ObjectStreamException { return
fINSTANCE; } }
```

Immutable objects

- are simple to construct, test, and use
- are automatically thread-safe and have no synchronization issues
- don't need a copy constructor
- don't need an implementation of clone
- allow hashCode to use lazy initialization, and to cache its return value
- don't need to be copied defensively when used as a field
- make good Map keys and Set elements (these objects must not change state while in the collection)
- have their class invariant established once upon construction, and it never needs to be checked again
- always have "failure atomicity" (a term used by Joshua Bloch): if an immutable object throws an exception, it's never left in an undesirable or indeterminate state

Immutable objects

- Immutable objects have a very compelling list of positive qualities. Without question, they are among the simplest and most robust kinds of classes you can possibly build. When you create immutable classes, entire categories of problems simply disappear.

Immutable objects instructions

Make a class immutable by following these guidelines:

•ensure the class cannot be overridden - make the class final, or use static factories and keep constructors private

•make fields private and final

•force callers to construct an object completely in a single step, instead of using a no-argument constructor combined with subsequent calls to setXXX methods (that is, avoid the Java Beans convention)

•do not provide any methods which can change the state of the object in any way - not just setXXX methods, but any method which can change state

•if the class has any mutable object fields, then they must be defensively copied when they pass between the class and its caller

In *Effective Java*, Joshua Bloch makes this compelling recommendation :
"Classes should be immutable unless there's a very good reason to make them mutable....If a class cannot be made immutable, limit its mutability as much as possible."
It's interesting to note that BigDecimal is technically *not* immutable, since it's not final.

```java
import java.util.Date;

/**
* Planet is an immutable class, since there is no way to change
* its state after construction.
*/
public final class Planet {

  public Planet (double aMass, String aName, Date aDateOfDiscovery) {
     fMass = aMass;
     fName = aName;
     //make a private copy of aDateOfDiscovery
     //this is the only way to keep the fDateOfDiscovery
     //field private, and shields this class from any changes that
     //the caller may make to the original aDateOfDiscovery object
     fDateOfDiscovery = new Date(aDateOfDiscovery.getTime());
  }
```

```java
/**
* Returns a primitive value.
*
* The caller can do whatever they want with the return value, without
* affecting the internals of this class. Why? Because this is a primitive
* value. The caller sees its "own" double that simply has the
* same value as fMass.
*/
public double getMass() {
  return fMass;
}

/**
* Returns an immutable object.
*
* The caller gets a direct reference to the internal field. But this is not
* dangerous, since String is immutable and cannot be changed.
*/
public String getName() {
  return fName;
}
```

```java
//  /**
//  * Returns a mutable object - likely bad style.
//  *
//  * The caller gets a direct reference to the internal field. This is usually dangerous,
//  * since the Date object state can be changed both by this class and its caller.
//  * That is, this class is no longer in complete control of fDate.
//  */
//  public Date getDateOfDiscovery() {
//    return fDateOfDiscovery;
//  }

  /**
  * Returns a mutable object - good style.
  *
  * Returns a defensive copy of the field.
  * The caller of this method can do anything they want with the
  * returned Date object, without affecting the internals of this
  * class in any way. Why? Because they do not have a reference to
  * fDate. Rather, they are playing with a second Date that initially has the
  * same data as fDate.
  */
  public Date getDateOfDiscovery() {
    return new Date(fDateOfDiscovery.getTime());
  }
  // PRIVATE

  /**
  * Final primitive data is always immutable.
  */
  private final double fMass;

  /**
  * An immutable object field. (String objects never change state.)
  */
  private final String fName;

  /**
  * A mutable object field. In this case, the state of this mutable field
  * is to be changed only by this class. (In other cases, it makes perfect
  * sense to allow the state of a field to be changed outside the native
  * class; this is the case when a field acts as a "pointer" to an object
  * created elsewhere.)
  */
  private final Date fDateOfDiscovery;
}
```

Factory methods

Factory methods are static methods that return an instance of the native class. Examples in the JDK:

- LogManager.getLogManager
- Pattern.compile
- Collections.unmodifiableCollection, Collections.synchronizeCollection , and so on
- Calendar.getInstance

Factory methods:

- have names, unlike constructors, which can clarify code.
- do not need to create a new object upon each invocation - objects can be cached and reused, if necessary.
- can return a subtype of their return type - in particular, *can return an object whose implementation class is unknown to the caller.* This is a very valuable and widely used feature in many frameworks which use interfaces as the return type of static factory methods.

Common names for factory methods include getInstance and valueOf. These names are not mandatory - choose whatever makes sense for each case.

```java
public final class ComplexNumber {

  /**
  * Static factory method returns an object of this class.
  */
  public static ComplexNumber valueOf(float aReal, float aImaginary) {
    return new ComplexNumber(aReal, aImaginary);
  }

  /**
  * Caller cannot see this private constructor.
  *
  * The only way to build a ComplexNumber is by calling the static
  * factory method.
  */
  private ComplexNumber(float aReal, float aImaginary) {
    fReal = aReal;
    fImaginary = aImaginary;
  }

  private float fReal;
  private float fImaginary;

  //..elided
}
```

Copy constructors

Copy constructors:

- provide an attractive alternative to the rather pathological clonemethod
- are easily implemented
- simply extract the argument's data, and forward to a regular constructor
- are unnecessary for immutable objects

```java
public final class Galaxy {

  /**
  * Regular constructor.
  */
  public Galaxy(double aMass, String aName) {
    fMass = aMass;
    fName = aName;
  }

  /**
  * Copy constructor.
  */
  public Galaxy(Galaxy aGalaxy) {
    this(aGalaxy.getMass(), aGalaxy.getName());
    //no defensive copies are created here, since
    //there are no mutable object fields (String is immutable)
  }

  /**
  * Alternative style for a copy constructor, using a static newInstance
  * method.
  */
  public static Galaxy newInstance(Galaxy aGalaxy) {
    return new Galaxy(aGalaxy.getMass(), aGalaxy.getName());
  }

  public double getMass() {
    return fMass;
  }
  public void setMass(double aMass){
    fMass = aMass;
  }

  public String getName() {
    return fName;
  }

  // PRIVATE
  private double fMass;
  private final String fName;

  /** Test harness. */
  public static void main (String... aArguments){
    Galaxy m101 = new Galaxy(15.0, "M101");

    Galaxy m101CopyOne = new Galaxy(m101);
    m101CopyOne.setMass(25.0);
    System.out.println("M101 mass: " + m101.getMass());
    System.out.println("M101Copy mass: " + m101CopyOne.getMass());

    Galaxy m101CopyTwo = Galaxy.newInstance(m101);
    m101CopyTwo.setMass(35.0);
    System.out.println("M101 mass: " + m101.getMass());
    System.out.println("M101CopyTwo mass: " + m101CopyTwo.getMass());
  }
}
```

- Example run of this class:
- >java -cp . Galaxy
- M101 mass: 15.0
- M101Copy mass: 25.0
- M101 mass: 15.0
- M101CopyTwo mass: 35.0

- In general, Collections are not immutable objects. As such, one must often exercise care that collection fields are not unintentionally exposed to the caller.
- One technique is to define a set of related methods which prevent the caller from directly using the underlying collection, such as:
- addThing(Thing)
- removeThing(Thing)
- getThings() - return an unmodifiable Collection

```java
import java.util.*;

public final class SoccerTeam {

  public SoccerTeam(String aTeamName, String aHeadCoachName){
    //..elided
  }

  public void addPlayer(Player aPlayer){
    fPlayers.add(aPlayer);
  }

  public void removePlayer(Player aPlayer){
    fPlayers.remove(aPlayer);
  }

  public Set<Player> getPlayers(){
    return Collections.unmodifiableSet(fPlayers);
  }

  //..elided

  // PRIVATE
  private Set<Player> fPlayers = new LinkedHashSet<>();
  private String fTeamName;
  private String fHeadCoachName;
}
```