

Computer System Design & Application

计算机系统设计与应用A

陶伊达 (TAO Yida)

taoyd@sustech.edu.cn



Lecture 10

- Reflection
- Annotation

What is Reflection (反射) ?



- Reflection is a feature in Java, an API in `java.lang.reflect` package
- Reflection is used to examine or modify the behavior of methods, classes, interfaces **at runtime**
 - Examining all fields and methods of a class
 - Invoking a method of an object
 - Accessing a private field from another class

A Motivating Example

- x might be obtained from users, JSON file, server response, etc.; We do not know its exact type
- Using isinstance, we still do not know the exact type of x (it might be a subclass, e.g., Rectangle)
- What if we need to perform different actions for different types of x? Should we write a dozen if isinstance block?

```
Object x = . . . ;  
if (x instanceof Shape)  
{  
    Shape s = (Shape) x;  
    g2.draw(s);  
}
```

Finding the actual type

```
public final class Class<T>  
    extends Object
```

- If you have any object reference, you can find the actual type of the object to which it refers with the `getClass()` method
- The `getClass()` method returns an object of type `Class` that describes the object's class.

```
Class c = x.getClass()
```

```
getClass()
```

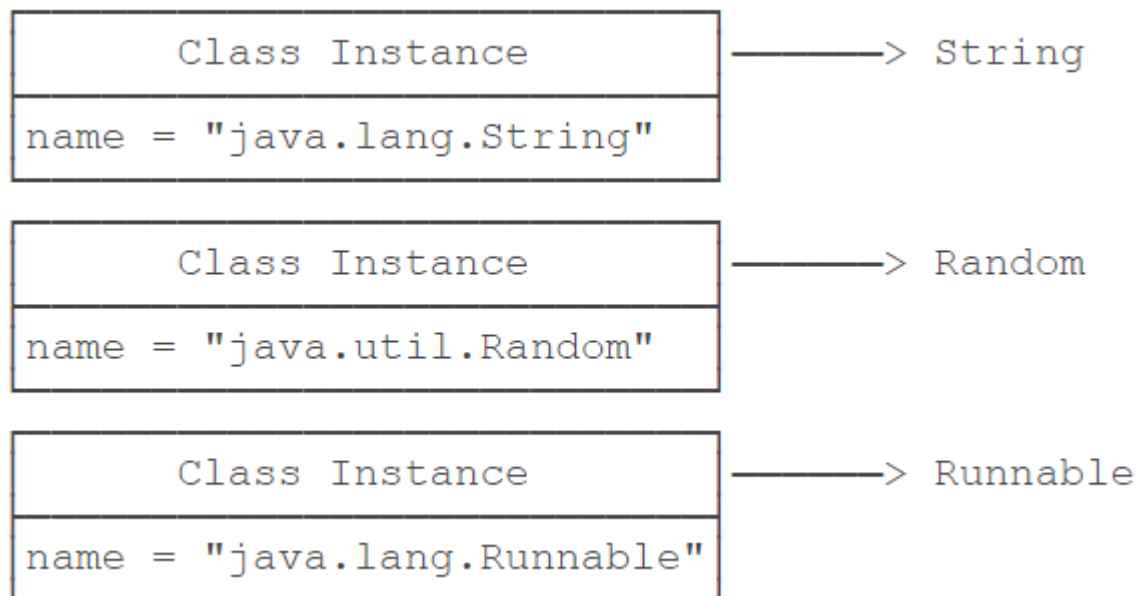
Returns the runtime class of this Object.

After you have a `Class` instance, you can obtain a large amount of information about the class (e.g., name, fields, constructors, methods)

The `Class` class

Reflection: getting information of a class through its `Class` instance

JVM creates one instance of type `Class` for every data type (including primitive types, classes and interfaces)



A `Class` instance has detailed information for the corresponding class

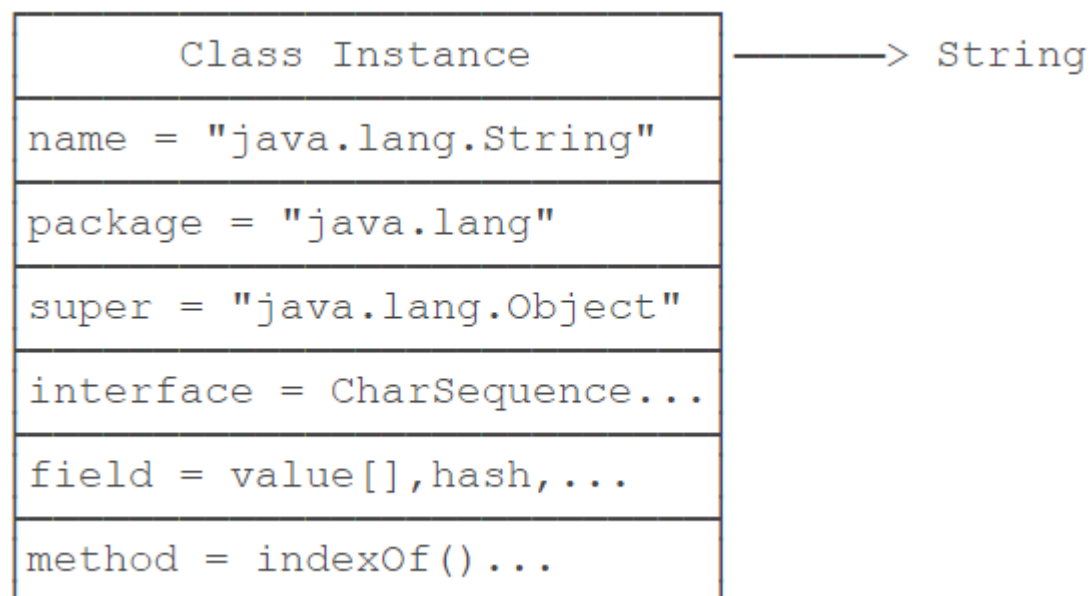


Image source: <https://www.liaoxuefeng.com/wiki/1252599548343744/1264799402020448>

JVM Loading Process

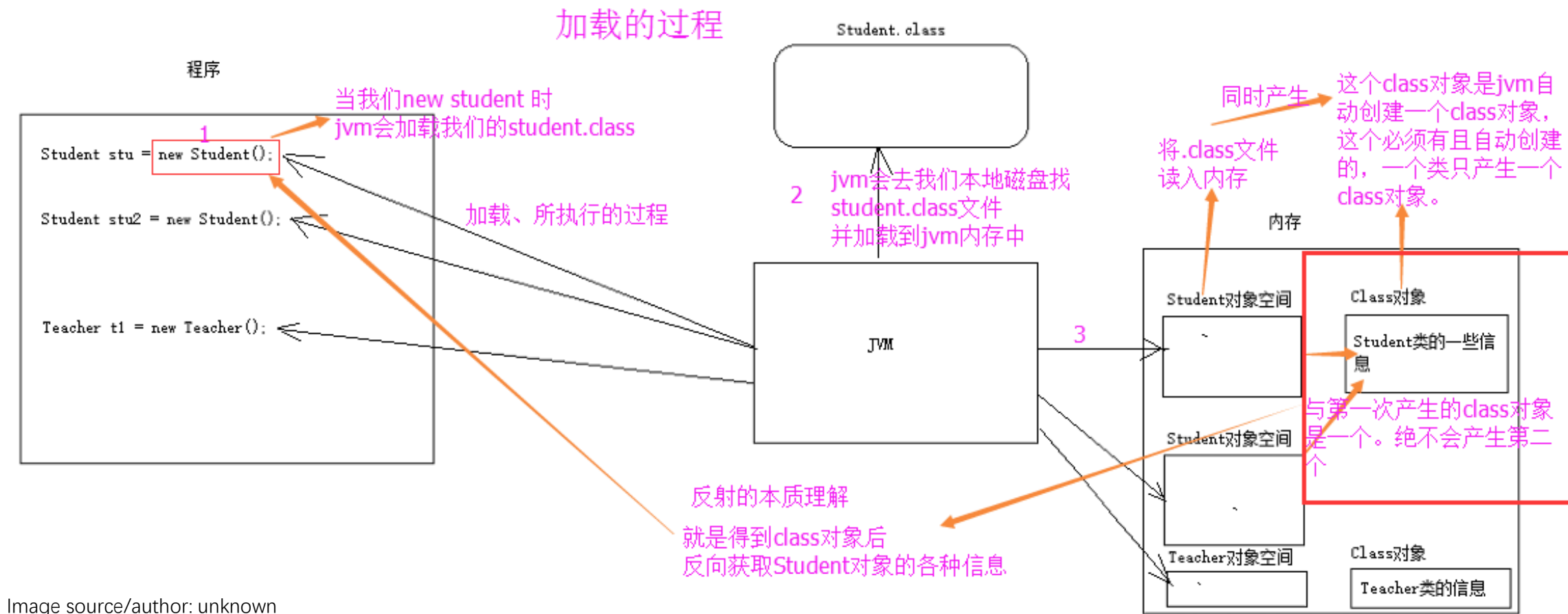
- Class instances are constructed automatically by JVM when it loads classes (.class files)
- Whenever JVM loads a class/type (e.g., String), it creates an instance of type Class for it

```
Class cls = new Class(String);
```

Sort of. Class has no public constructor so we cannot create Class objects, only JVM can.

```
/*  
 * Constructor. Only the Java Virtual Machine creates Class  
 * objects.  
 */  
private Class() {}
```

JVM Loading Process



Getting Class Objects

- Approach 1: using `.class` (for known classes)

```
Class cls1 = String.class;
```

- Approach 2: using `Class.forName()` (using full package name)

```
Class cls2 = Class.forName("java.lang.String");
```

- Approach 3: using `getClass()` on class instances

```
String s = "Hello";  
Class cls3 = s.getClass();
```

What is the relationship
between `cls1`, `cls2`, and `cls3`?

Getting Class Objects

- There is only one `Class` object for every type that has been loaded into JVM
- We can use the `==` operator to test whether two class instances describe the same type

```
Class cls1 = String.class;  
Class cls2 = Class.forName("java.lang.String");  
Class cls3 = "hello".getClass();
```

```
System.out.println(cls1 == cls2);  
System.out.println(cls1 == cls3);
```

Getting Class Names

- To get the exact class name of a Java object, get its Class object and invoke `getName()` on it

```
String s = "Hello";  
System.out.println(s.getClass().getName());
```

`java.lang.String`

Getting Class Names

NOTE For historical reasons, the `getName` method produces strange-looking names for array types. For example, `double[].class.getName()` is

`"[D"`

and `String[][] .class.getName()` is

`"[[Ljava.lang.String;"`

In general, an array type name is made up according to the following rules:

<i>[type</i>	array type
B	byte
C	char
D	double
F	float
I	int
J	long
<i>Lname;</i>	class or interface
S	short
Z	boolean

Reference: Object-Oriented Design & Patterns.
Cay S. Horstmann. Chapter 7.

Getting Class Fields

<code>Field getField(name)</code>	get public field given the name
<code>Field getDeclaredField(name)</code>	get field given the name
<code>Field[] getFields()</code>	get all public fields
<code>Field[] getDeclaredFields()</code>	get all fields (excludes inherited fields)

This includes public, protected, default (package) access, and private fields, but excludes inherited fields.

The Field Class

- A `Field` provides information about, and dynamic access to, a single field of a class or an interface

```
Field f = String.class.getDeclaredField(name: "value");
System.out.println(f.getName()); // value
System.out.println(f.getType()); // [B

int m = f.getModifiers();
System.out.println(Modifier.isFinal(m)); // true
System.out.println(Modifier.isPrivate(m)); // true
```

```
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence,
               Constable, ConstantDesc {

    The value is used for character storage.
    Implementation This field is trusted by the VM, and is a subject to constant folding i
    Note:           is constant. Overwriting this field after construction will cause probl
    Additionally, it is marked with Stable to trust the contents of the ar
    facility in JDK provides this functionality (yet). Stable is safe here, b
    never null.

    @Stable
    private final byte[] value;
```

`getModifiers()` returns the Java language modifiers for the field represented by this `Field` object, as an integer. The `Modifier` class should be used to decode the modifiers.

The Field Class

Using Reflection, we could update a field value (even for private fields)

```
BankAccount bc = new BankAccount();
System.out.println(bc.getBalance()); // 0.0

Class clz = bc.getClass();
// get the private field
Field f = clz.getDeclaredField("balance");
// make the private field accessible
f.setAccessible(true);
// set the balance
f.set(bc, 1000);
System.out.println(bc.getBalance()); // 1000.0
```

Getting Class Methods

An array of Class objects that identify the method's formal parameter types, in declared order (e.g., `int.class`)

Method <code>getMethod(name, Class...)</code>	get public method
Method <code>getDeclaredMethod(name, Class...)</code>	get method
Method[] <code>getMethods()</code>	get all public methods
Method[] <code>getDeclaredMethods()</code>	get all methods (excludes inherited methods)

This includes public, protected, default access, and even private methods, but excludes inherited ones.

The Method Class

<code>int</code>	<code>getModifiers()</code> Returns the Java language modifiers for the method represented by this object.
<code>String</code>	<code>getName()</code> Returns the name of the method represented by this object.
<code>Annotation[][]</code>	<code>getParameterAnnotations()</code> Returns an array of arrays of <code>Annotations</code> of the <code>Executable</code> represented by this object.
<code>int</code>	<code>getParameterCount()</code> Returns the number of formal parameters executable represented by this object.
<code>Class<?>[]</code>	<code>getParameterTypes()</code> Returns an array of <code>Class</code> objects that represent the parameter types of the method represented by this object.
<code>Class<?></code>	<code>getReturnType()</code> Returns a <code>Class</code> object that represents the return type of the method represented by this object.

- A Method provides information about, and access to, a single method on a class or interface.
- The reflected method may be a class method or an instance method (including an abstract method).

Invoking Methods using Reflection

- Invokes the underlying method represented by this Method object, on the specified object with the specified parameters.

```
public Object invoke(Object obj,  
                    Object... args)  
    throws IllegalAccessException,  
        IllegalArgumentException,  
        InvocationTargetException
```

```
String s = "Hello Java";  
Method m = String.class.getMethod("substring", int.class);  
System.out.println(m.invoke(s, 6));
```

What's the output?

Invoking Methods using Reflection

- Invokes the underlying method represented by this Method object, on the specified object with the specified parameters.

```
public Object invoke(Object obj,  
                    Object... args)  
    throws IllegalAccessException,  
        IllegalArgumentException,  
        InvocationTargetException
```

```
Method m2 = Integer.class.getMethod("parseInt", String.class);  
System.out.println(m2.invoke(null, "12345").getClass().getName());
```

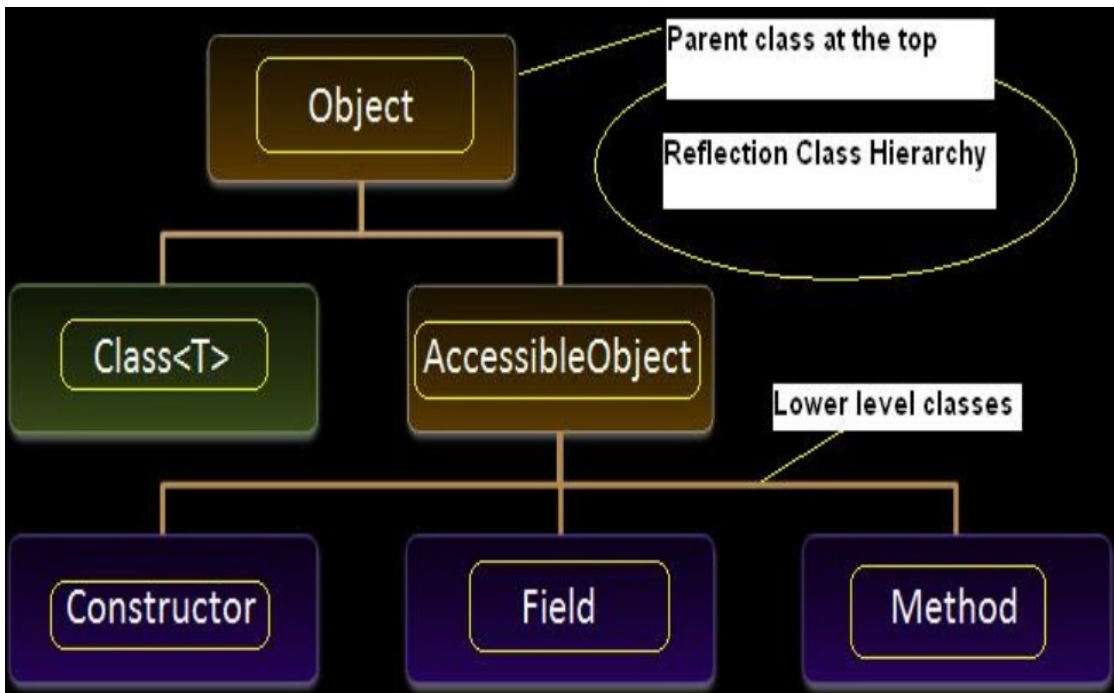


Why null?

Method Accessibility

- By default, not all reflected methods are accessible. This means that the JVM enforces access control checks when invoking them.
- For instance, if we try to call a private method outside its defining class or a protected method from outside a subclass or its class's package, we'll get an `IllegalAccessException`
- By calling `setAccessible(true)` on a reflected method object, the JVM suppresses the access control checks and allows us to invoke the method without throwing an exception

AccessibleObject



<https://docs.oracle.com/javase/9/docs/api/java/lang/reflect/AccessibleObject.html>

- The AccessibleObject class is the base class for Field, Method, and Constructor objects (known as reflected objects)
- It provides features to check access and suppress access checks

boolean	<code>canAccess(Object obj)</code>
void	<code>setAccessible(boolean flag)</code>
static void	<code>setAccessible(AccessibleObject[] array, boolean flag)</code>
boolean	<code>trySetAccessible()</code>

Object Instantiation using Reflection

- Use `getConstructor(Class<?>...)`
 - Returns a `Constructor` object that reflects the specified public constructor of the class represented by this `Class` object
 - Could specify **formal parameters** of constructors

```
Class clz = Student.class;  
Constructor constructor = clz.getConstructor(String.class, int.class);  
Student std = (Student) constructor.newInstance("Alice", 15);
```

Inspecting Class Inheritance with Reflection

```
Class ac = ArrayList.class;
Class sc = ac.getSuperclass();
System.out.println(sc);
System.out.println("-----");
Class[] ai = ac.getInterfaces();
for (Class i : ai) {
    System.out.println(i);
}
```

```
class java.util.AbstractList
-----
interface java.util.List
interface java.util.RandomAccess
interface java.lang.Cloneable
interface java.io.Serializable
```

Real Usages of Reflection

- **JUnit testing:**

- Previous Junit processor was using reflection to iterate over all methods in test classes, and find-out methods starting with *test* and run them as testcases.
- We may need to **set private fields** or **invoke private methods** for testing purpose

- **Spring framework:**

- **Dependency injection** (DI) heavily uses reflection
- Identify dependencies (e.g., parameter types for constructors)
- Create instances of classes at runtime and inject the instances to other objects

Problems of Reflection

- **Risky:** you could set private final fields and invoke private methods
- **You lose compile-time type safety** - it's helpful to have the compiler verify that a method is available at compile time. If you are using reflection, you'll get an error at runtime which might affect end users if you don't test well enough. Even if you do catch the error, it will be more difficult to debug.
- **It causes bugs when refactoring** - if you are accessing a member based on its name (e.g. using a hard-coded string) then this won't get changed by most code refactoring tools and you'll instantly have a bug, which might be quite hard to track down.
- **Performance is slower** - reflection at runtime is going to be slower than statically compiled method calls/variable lookups. If you're only doing reflection occasionally then it won't matter, but this can become a performance bottleneck in cases where you are making calls via reflection thousands or millions of times per second.

<https://softwareengineering.stackexchange.com/a/101217/21021>

Using Reflection?

If you could do something without reflection, stick to that.

Normal code is simpler, cleaner, and more readable, with compiler type safety and optimization



WITH
GREAT POWER
COMES
GREAT
RESPONSIBILITY



Lecture 10

- Reflection
- Annotation



Java Annotation Overview

- Java annotations start with '@'
- Java annotations are **metadata** (data about data) attached to program entities such as classes, interfaces, fields and methods
- Java annotations leave the semantics of a program **unchanged** (i.e., annotations do not change the action or execution of a compiled program)
- We cannot consider annotations (注解) as pure comments (注释) as they can change the way a compiler or runtime treats a program



Java Annotation Overview

- Java annotations are typically used for providing the following extra information:
 - **Compiler instructions:** The compiler can use annotations to catch errors or suppress warnings.
 - **Build-time instructions:** Build tools may scan Java code for specific annotations and generate source code or other files (e.g., XML) based on these annotations
 - **Runtime instructions:** Some annotations are available to be examined (by reflection) at runtime.

Compiling vs Building

The "Build" is a process that covers all the steps required to create a "deliverable" of your software. In the Java world, this typically includes:

1. Generating sources (sometimes).
2. Compiling sources.
3. Compiling test sources.
4. Executing tests (unit tests, integration tests, etc).
5. Packaging (into jar, war, ejb-jar, ear).
6. Running health checks (static analyzers like Checkstyle, Findbugs, PMD, test coverage, etc).
7. Generating reports.

So as you can see, compiling is only a (small) part of the build (and the best practice is to fully automate all the steps with tools like Maven or Ant and to run the build continuously which is known as [Continuous Integration](https://stackoverflow.com/a/2650423/636398)).

<https://stackoverflow.com/a/2650423/636398>

Annotation Categories

- **Predefined annotations**

- ① • Built-in annotation: annotation types used by the Java language
 - Meta-annotation: annotations that apply to other annotations

③

- ② • **Custom annotations**



Built-in Annotations

Annotation types defined in `java.lang`

- `@Deprecated`
- `@Override`
- `@SuppressWarnings`
- `@FunctionalInterface`
- `@SafeVarargs`

@Deprecated

```
Date dt = new Date( year: 2002, month: 12, date: 20 );
```

'Date(int, int, int)' is deprecated

```
@Deprecated
@Contract(pure = true)
public Date(
    int year,
    @MagicConstant(intValues = {Cal
    int date
)
```

- This annotation indicates the element (class, method, field, etc.) 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

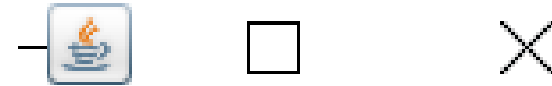
@Override

- This annotation informs the compiler that a method is meant to override a method declared in a superclass
- While it is not mandatory to use this annotation when overriding a method, it helps to prevent errors.
- If a method marked with @Override fails to correctly override a method of its superclass (e.g., wrong parameter type), the compiler generates an error.

Example of using @Override

- The code compiles and runs, but the close button won't work

```
import java.awt.*;
import java.awt.event.*;
public class AnnotationOverrideDemo extends Frame {
    public AnnotationOverrideDemo() {
        this.addWindowListener(new WindowAdapter() {
            public void windowclosing(WindowEvent e) {
                System.exit(0);
            }
        });
        setSize(200, 100);
        setTitle("Annotation Override Demo");
        setVisible(true);
    }
    public static void main(String[] args) { new AnnotationOverrideDemo(); }
}
```



Example: <https://www3.ntu.edu.sg/home/ehchua/programming/java/Annotation.html>

Example of using @Override

- Add annotation @Override to the windowClosing(), which signals your intention, serves as documentation, and allows the compiler to catch this error.

```
@Override  
public void windowclosing(WindowEvent e) {  
    System.exit(0);  
}
```

Should be windowClosing

```
@Override  
public v  
Syst  
Method does not override method from its superclass
```

Example: <https://www3.ntu.edu.sg/home/ehchua/programming/java/Annotation.html>



@SuppressWarnings

- This annotation tells the compiler to suppress specific warnings that it would otherwise generate
- Every compiler warning belongs to a category. The Java Language Specification lists two categories: deprecation and unchecked. The unchecked warning can occur when interfacing with legacy code written before the advent of generics.
- To suppress multiple categories of warnings, use:

```
@SuppressWarnings({"unchecked", "deprecation"})
```

@SuppressWarnings

```
public class SuppressWarningsTest {
```

```
    public static void addSth(List list){
```

```
        list.add("Test");
```

```
    }
```

```
    public static
```

```
        List li
```

```
    }
```

```
}
```

Unchecked call to 'add(E)' as a member of raw type 'java.util.List'

Try to generify 'SuppressWarningsTest.java' Alt+Shift+Enter

```
@Contract(value = "_->true", mutates = "this")
```

```
public abstract boolean add(
```

```
    E e
```

```
,
```

```
public class SuppressWarningsTest {
```

```
    @SuppressWarnings("unchecked")
```

```
    public static void addSth(List list){
```

```
        list.add("Test");
```

```
    }
```

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

```
        List list = new ArrayList<>();
```

```
    }
```

```
}
```

@SafeVarargs

- varargs: a method has parameter(s) of variable length
- When we use varargs with generic types, as there is a potential risk of a fatal runtime exception, the compiler warns us about a possible unsafe varargs usage

```
public class SafeVarargsDemo {  
    public static void main(String[] args) {  
        display( ...array: "10", 20, 30.00);  
    }  
}
```

```
public static <T> void display(T... array){  
    for(T arg: array){  
        System.out.print  
    }  
}
```

Possible heap pollution from parameterized
Annotate as '@SafeVarargs' Alt+Shift+Enter

```
public static <T> void display(  
    @NotNull T... array
```

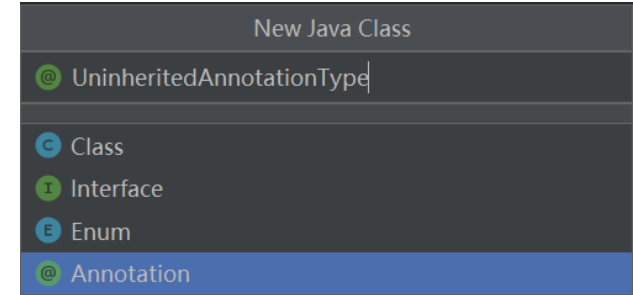


@SafeVarargs

If we are sure that our actions are safe, we could use the @SafeVarargs annotation to suppress this warning

```
public class SafeVarargsDemo {  
    public static void main(String[] args) {  
        display( ...array: "10", 20, 30.00);  
    }  
  
    @SafeVarargs  
    public static <T> void display(T... array){  
        for(T arg: array){  
            System.out.println(arg.getClass().getName());  
        }  
    }  
}
```


Custom Annotations



- It is also possible to create your own custom annotations.
- An annotation type class implicitly extends the marker interface `java.lang.annotation.Annotation`
- The `@interface` keyword is used to declare a new annotation type
- Annotation type declarations are similar to normal interface declarations



Custom Annotations

Meta-annotations go here

```
[Access Specifier] @interface<AnnotationName> {  
    DataType <Method Name>() [default value];  
}
```

- Annotations can be created by using **@interface** followed by the annotation name.
- The annotation can have elements that look like methods but they do not have an implementation.
- The default value is optional.
- The return type (DataType) of the method can be primitive, enum, string, class name or array of these types.

<https://www.programiz.com/java-programming/annotation-types>

Example

<https://docs.oracle.com/javase/tutorial/java/annotations/declaring.html>

Many annotations replace comments in code.

Suppose that a software group traditionally starts the body of every class with comments providing important information:

```
public class Generation3List extends Generation2List {  
  
    // Author: John Doe  
    // Date: 3/17/2002  
    // Current revision: 6  
    // Last modified: 4/12/2004  
    // By: Jane Doe  
    // Reviewers: Alice, Bill, Cindy  
  
    // class code goes here  
  
}
```

Example

<https://docs.oracle.com/javase/tutorial/java/annotations/declaring.html>

To add this same metadata with an annotation, you must first define the *annotation type*. The syntax for doing this is:

```
@interface ClassPreamble {  
    String author();  
    String date();  
    int currentRevision() default 1;  
    String lastModified() default "N/A";  
    String lastModifiedBy() default "N/A";  
    // Note use of array  
    String[] reviewers();  
}
```

```
[Access Specifier] @interface<AnnotationName> {  
    DataType <Method Name>() [default value];  
}
```

The annotation type definition looks similar to an interface definition where the keyword interface is preceded by the at sign (@) (@ = AT, as in annotation type).

Example

<https://docs.oracle.com/javase/tutorial/java/annotations/declaring.html>

After defining the **ClassPreamble** annotation, you can use it with the values filled in, which is:

- More organized and a fixed, consistent format
- Easy to be included in automatically generated Javadoc

```
public class Generation3List extends Generation2List {  
  
    // Author: John Doe  
    // Date: 3/17/2002  
    // Current revision: 6  
    // Last modified: 4/12/2004  
    // By: Jane Doe  
    // Reviewers: Alice, Bill, Cindy  
  
    // class code goes here  
  
}
```



```
@ClassPreamble (  
    author = "John Doe",  
    date = "3/17/2002",  
    currentRevision = 6,  
    lastModified = "4/12/2004",  
    lastModifiedBy = "Jane Doe",  
    // Note array notation  
    reviewers = {"Alice", "Bob", "Cindy"}  
)  
public class Generation3List extends Generation2List {  
  
    // class code goes here  
  
}
```



Meta-annotations

- Annotations that apply to other annotations are called meta-annotations.
- There are several meta-annotation types defined in `java.lang.annotation`.
 - `@Documented`
 - `@Target`
 - `@Retention`
 - `@Inherited`, `@Repeatable`

@Documented

- @Documented indicates that annotations with a type are to be documented by Javadoc by default.

```
@Documented
@interface ClassPreamble {

    // Annotation element definitions

}
```

The @ClassPreamble now is available in the Javadoc of **Generation3List**

```
@ClassPreamble (
    author = "John Doe",
    date = "3/17/2002",
    currentRevision = 6,
    lastModified = "4/12/2004",
    lastModifiedBy = "Jane Doe",
    // Note array notation
    reviewers = {"Alice", "Bob", "Cindy"}
)
public class Generation3List extends Generation2List {

    // class code goes here

}
```



@Target

- This annotation marks another annotation to restrict what kind of Java elements the annotation can be applied to.
 - `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

- This annotation marks another annotation to restrict what kind of Java elements the annotation can be applied to.

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.SOURCE)
public @interface Override {
}
```

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface FunctionalInterface {}
```

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.CONSTRUCTOR, ElementType.METHOD})
public @interface SafeVarargs {}
```



@Retention

- This annotation specifies how an annotation is stored (at which level it is available)
- Syntax: `@Retention(RetentionPolicy)`
- 3 types of RetentionPolicy
 - `RetentionPolicy.SOURCE` – The marked annotation is retained only in the source level and is ignored by the compiler (do not exist in .class files).
 - `RetentionPolicy.CLASS` – The marked annotation is retained by the compiler at compile time, but is ignored by JVM (recorded in the .class file but are discarded during runtime)
 - `RetentionPolicy.RUNTIME` – The marked annotation is retained by the JVM so it can be accessed by the runtime environment.



RetentionPolicy.SOURCE

The marked annotation is retained only in the source level and is ignored by the compiler (do not exist in .class files).

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.SOURCE)
public @interface Override {
}
```

```
@Target({TYPE, FIELD, METHOD, PARAMETER,
})
@Retention(RetentionPolicy.SOURCE)
public @interface SuppressWarnings {
```



RetentionPolicy.CLASS

- The marked annotation is retained by the compiler at compile time, but is ignored by JVM
- The compiler keeps the annotations in the `.class` files, however they are not loaded by the `ClassLoader` when running a program.
- Useful for tools that process `.class` files, such as bytecode analyzers, code obfuscators, or static analysis tools. These tools can read annotations in the `.class` file without needing the annotation at runtime.

Motivating Example

The validity checks are boring but necessary. Can we simplify the code?

```
public class Person {  
  
    public String name;  
    public String city;  
    public int age;  
  
    public Person(String name, String city, int age) {  
  
        if (name == null || name.length() < 2 || name.length() > 50) {  
            throw new IllegalArgumentException("Name must be between 2  
                                             and 50 chars.");  
        }  
  
        if (city == null || city.length() < 2 || city.length() > 50) {  
            throw new IllegalArgumentException("City must be between 2  
                                             and 50 chars.");  
        }  
  
        if (age < 0 || age > 120) {  
            throw new IllegalArgumentException("Age must be between 0  
                                             and 120.");  
        }  
  
        this.name = name;  
        this.city = city;  
        this.age = age;  
    }  
}
```

RetentionPolicy.RUNTIME

```
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Range {
    int min() default 0;
    int max() default 255;
}
```

RUNTIME policy signals to the Java compiler and JVM that the annotation should be **available via reflection at runtime**.

Example adapted from
<https://www.liaoxuefeng.com/wiki/1252599548343744/1265102026065728>

```
public class Person {
    @Range(min=3, max=20)
    public String name;

    @Range(max=10)
    public String city;

    @Range(min=1, max=100)
    public int age;

    public Person(String name, String city, int age) {
        this.name = name;
        this.city = city;
        this.age = age;
    }
}
```

RetentionPolicy.RUNTIME

Since @Range is available at runtime, we can check it using reflection.

If we use SOURCE or CLASS retention policy for @Range, it will not be available at runtime (range == null in the code)

```
public static void check(Person person) throws IllegalAccessException {  
    // go through each field  
    for(Field field: person.getClass().getFields()){  
        // get the @Range annotation of the field  
        Range range = field.getAnnotation(Range.class);  
        // if there is a @Range annotation  
        if (range != null){  
            // get the value of the field  
            Object value = field.get(person);  
            if (value instanceof String){  
                String s = (String) value;  
                if (s.length() < range.min() || s.length() > range.max()){  
                    throw new IllegalArgumentException("Invalid range " +  
                        "of string field: " + field.getName());  
                }  
            }  
            else{  
                int i = (int) value;  
                if (i < range.min() || i > range.max()){  
                    throw new IllegalArgumentException("Invalid range of " +  
                        "int field: " + field.getName());  
                }  
            }  
        }  
    }  
}
```

RetentionPolicy.RUNTIME

```
Person p1 = new Person( name: "Alice", city: "Beijing", age: 20);  
Person p2 = new Person( name: "a", city: "Beijing", age: 20);  
Person p3 = new Person( name: "Alice", city: "The city name is Beijing", age: 20);  
Person p4 = new Person( name: "Alice", city: "Shenzhen", age: 200);
```

```
check(p1); OK  
check(p2); java.lang.IllegalArgumentException: Invalid range of string field: name  
check(p3); java.lang.IllegalArgumentException: Invalid range of string field: city  
check(p4); java.lang.IllegalArgumentException: Invalid range of int field: age
```


Wait, why do I want to write all those extra code?

```
public class Person {  
    @Range(min=3, max=20)  
    public String name;  
  
    @Range(max=10)  
    public String city;  
  
    @Range(min=1, max=100)  
    public int age;  
  
    public Person(String name, String city, int age) {  
        this.name = name;  
        this.city = city;  
        this.age = age;  
    }  
}
```

Dev

```
@Target(ElementType.FIELD)  
@Retention(RetentionPolicy.RUNTIME)  
public @interface Range {  
    int min() default 0;  
    int max() default 255;  
}  
  
public static void check(Person person) throws IllegalAccessException {  
    // go through each field  
    for(Field field: person.getClass().getFields()){  
        // get the @Range annotation of the field  
        Range range = field.getAnnotation(Range.class);  
        // if there is a @Range annotation  
        if (range != null){  
            // get the value of the field  
            Object value = field.get(person);  
            if (value instanceof String){  
                String s = (String) value;  
                if (s.length() < range.min() || s.length() > range.max()){  
                    throw new IllegalArgumentException("Invalid range " +  
                        "of string field: " + field.getName());  
                }  
            }  
            else{  
                int i = (int) value;  
                if (i < range.min() || i > range.max()){  
                    throw new IllegalArgumentException("Invalid range of " +  
                        "int field: " + field.getName());  
                }  
            }  
        }  
    }  
}
```

Framework

Next Lecture

- Java EE
- The Spring Framework