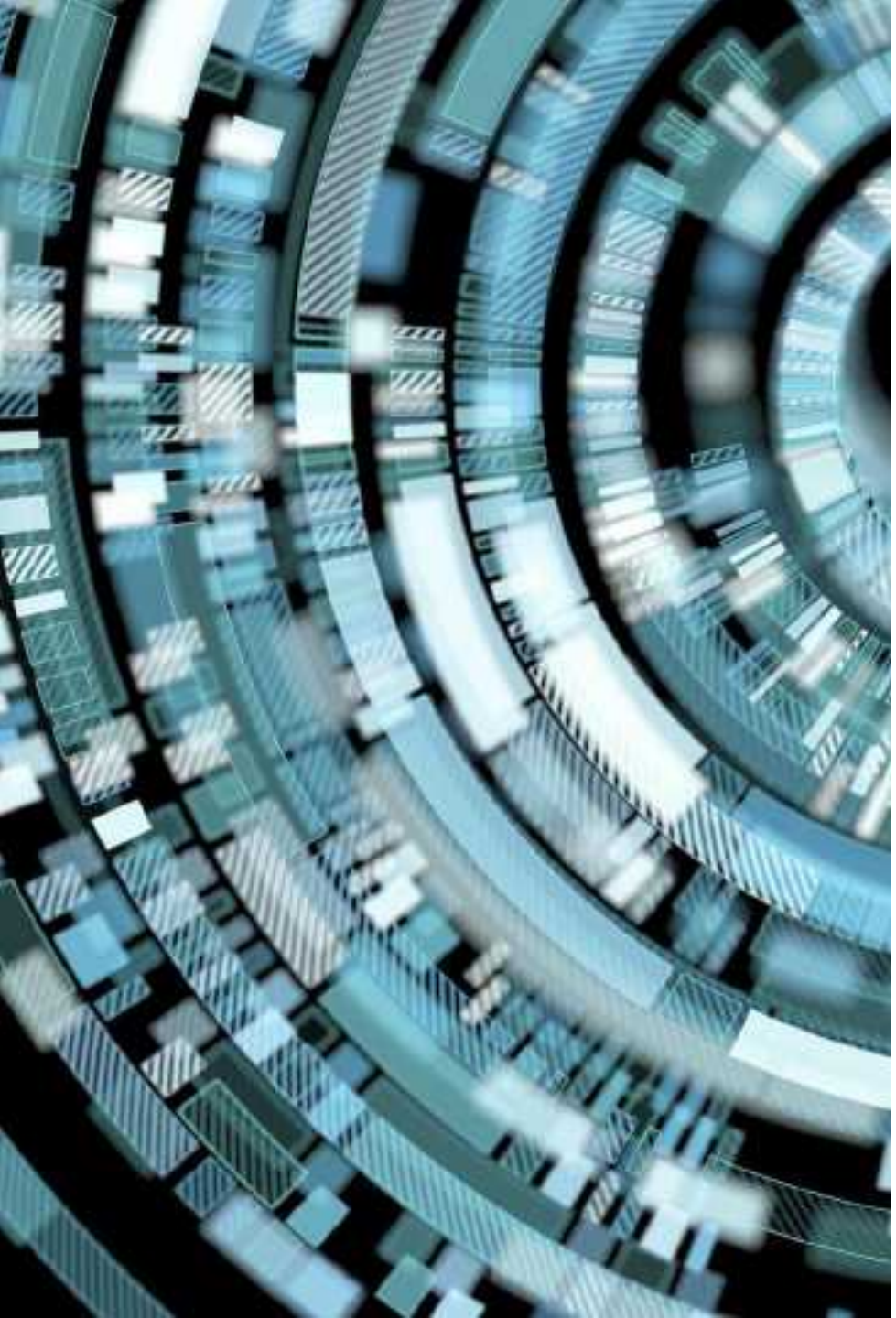
Computer System Design & Application

计算机系统设计与应用A

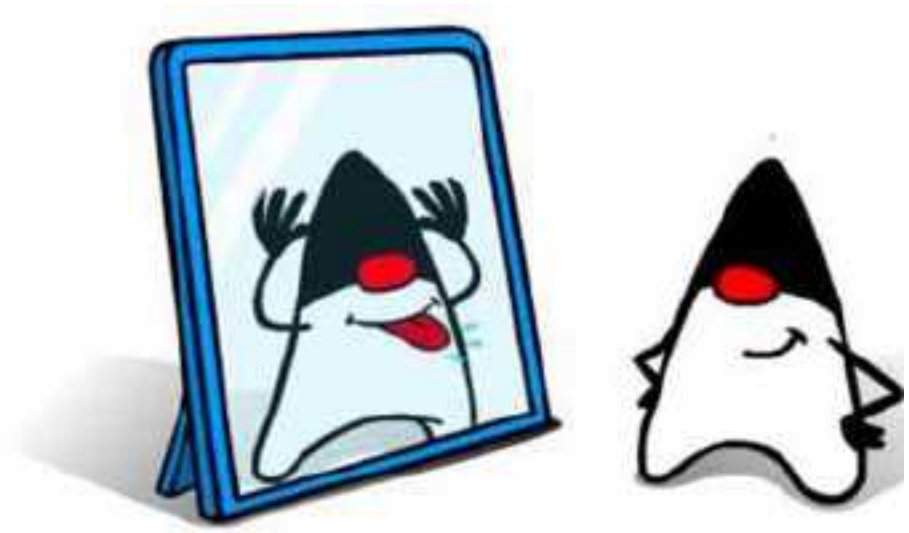
陶伊达 （TAO Yida）
taoyd@sustech.edu.cn



Lecture 10

- Reflection
- Annotation
- Unit Testing

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

Use Cases

- Executing all methods starting with “testXXX()” in a user-provided class
- Given an object, writing all its fields and respective values to a JSON/XML format
- Invoking a method at runtime
- ……

A More Concrete Example

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

- x might be obtained from users, JSON file, server response, etc.; We do not know its exact type
- Using instanceof, 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 instanceof block?

Finding the actual type

`getClass()`
Returns the runtime class of this `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()
```

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

The Class class

java.lang.Object
Java.lang.Class<T>

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

Class cls = new Class(String);

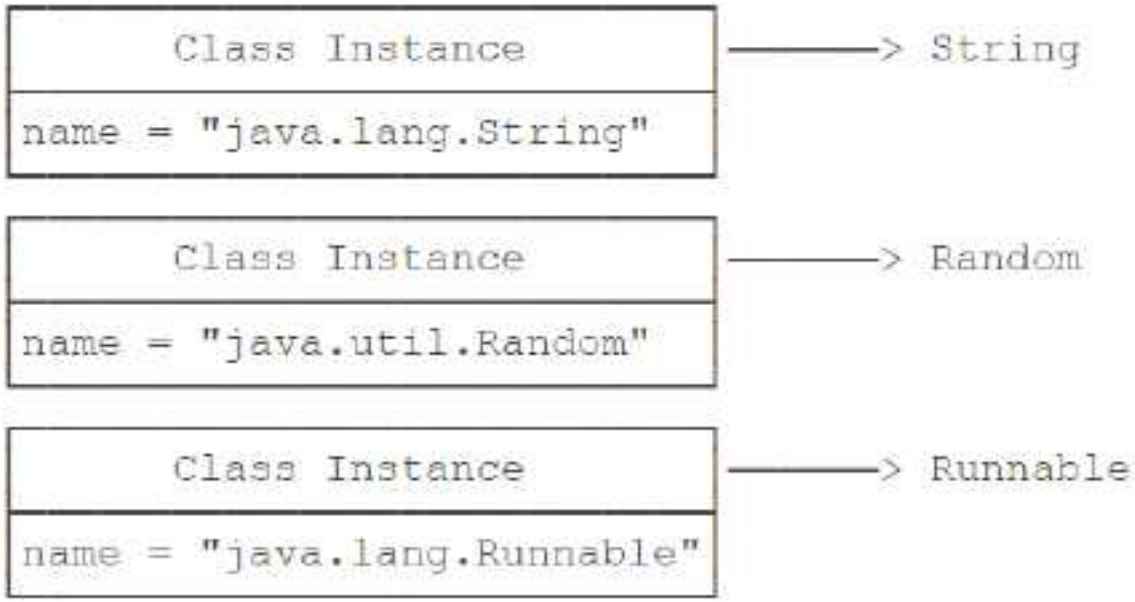
Sort of. Class has no public constructor so we cannot create Class objects, only JVM can.

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

The Class class

Reflection: getting information of a class through its Class instance

JVM creates one instance of type `Class` for every data type



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

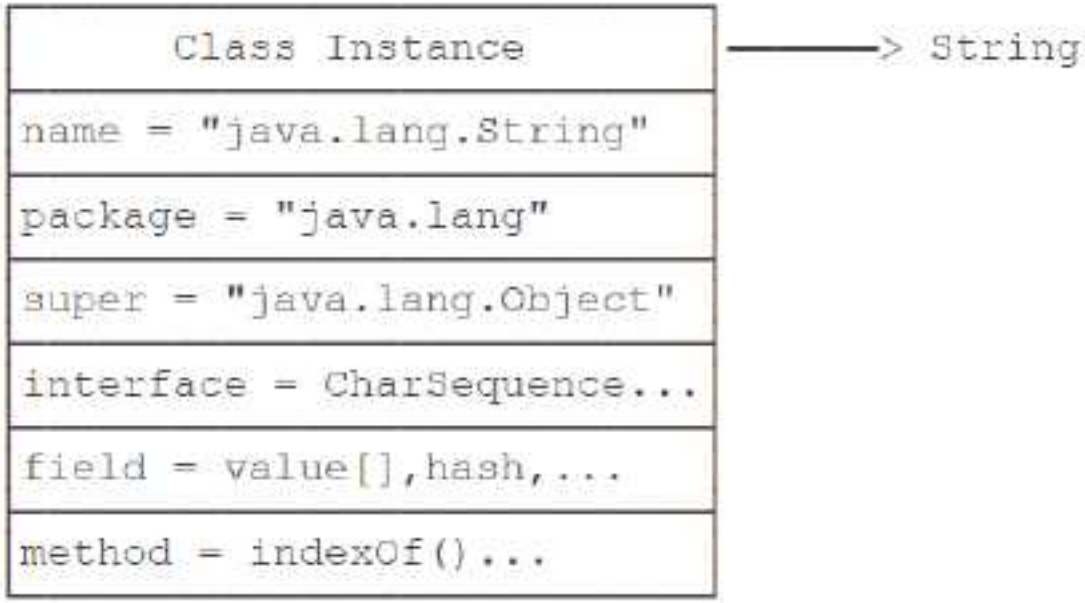


Image source: https://www.liaoxuefeng.com/wiki/1252599548343744/1264799402020448

JVM Loading Process

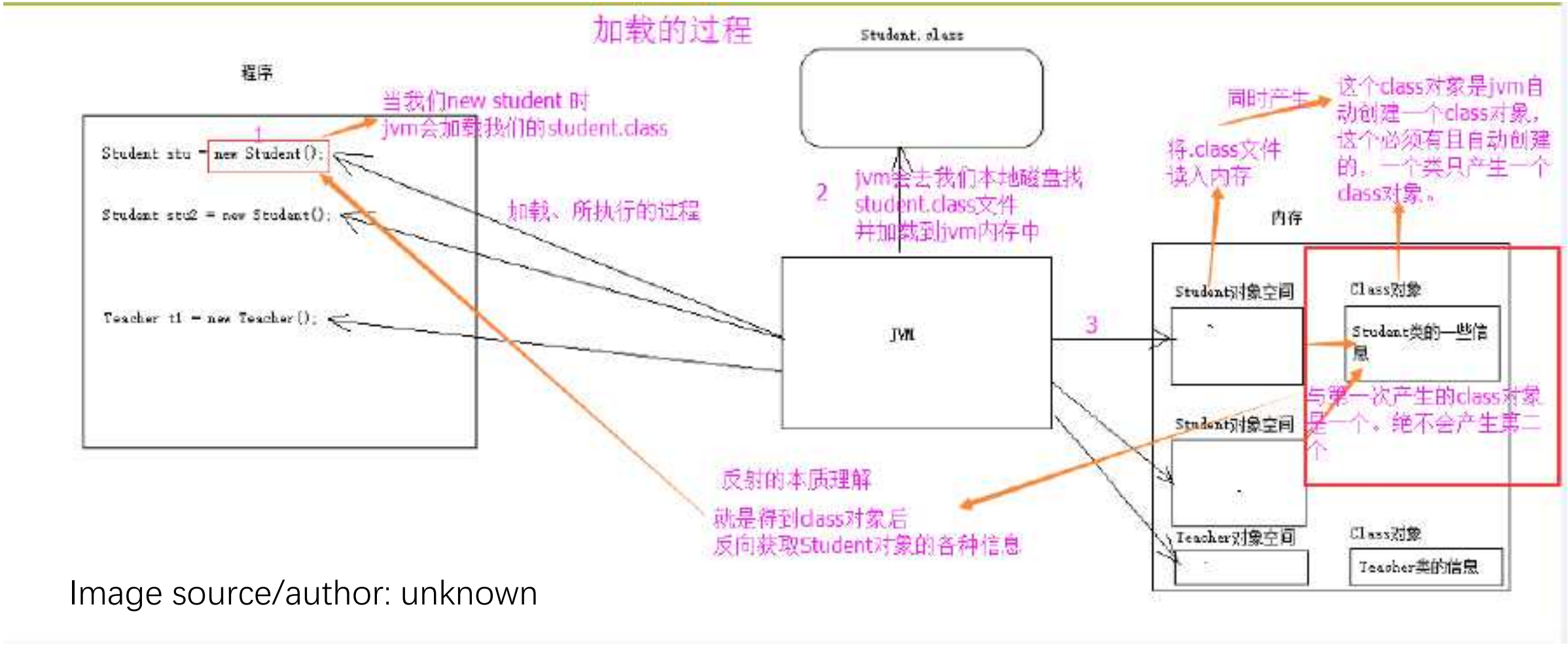


Image source/author: unknown

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?

Note

- 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 objects describe the same type

```
Class cls1 = String.class;

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

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

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

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:

[*type*	array type
B	byte
C	char
D	double
F	float
I	int
J	long
L*name*;	class or interface
S	short
Z	boolean

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

Getting Class Fields

`Field getField(name)`	get public field given the name
`Field getDeclaredField(name)`	get field given the name
`Field[] getFields()`	get all public fields
`Field[] getDeclaredFields()`	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");
f.getName(); // "value"
f.getType(); // [C
int m = f.getModifiers();
Modifier.isFinal(m); // true
Modifier.isPublic(m); // false
Modifier.isProtected(m); // false
Modifier.isPrivate(m); // true
Modifier.isStatic(m); // false
```

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( name: "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 getMethod(name, Class...)`	get public method
`Method getDeclaredMethod(name, Class...)`	get method
`Method[] getMethods()`	get all public methods
`Method[] getDeclaredMethods()`	get all methods (excludes inherited methods)

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

The Method Class

int	**getModifiers()** Returns the Java language **modifiers** for t
String	**getName()** Returns the name of the method represen
Annotation[][]	**getParameterAnnotations()** Returns an array of arrays of Annotations of the Executable represented by this obj
int	**getParameterCount()** Returns the number of formal parameters executable represented by this object.
Class<?>[]	**getParameterTypes()** Returns an array of Class objects that rep represented by this object.
Class<?>	**getReturnType()** Returns a Class object that represents th

- 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 world";
Method m = String.class.getMethod( name: "substring", int.class);
String r = (String) m.invoke(s,  ...args: 6);
System.out.println(r);
```

Example:
https://www.liaoxuefeng.com/wiki/1252599548343744/1264803678201760

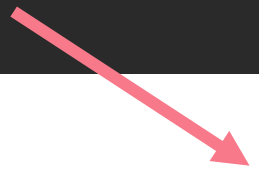
What's the output?

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 m = Integer.class.getMethod( name: "parseInt", String.class);
Integer n = (Integer) m.invoke( obj: null,  ...args: "12345");
System.out.println(n);
```



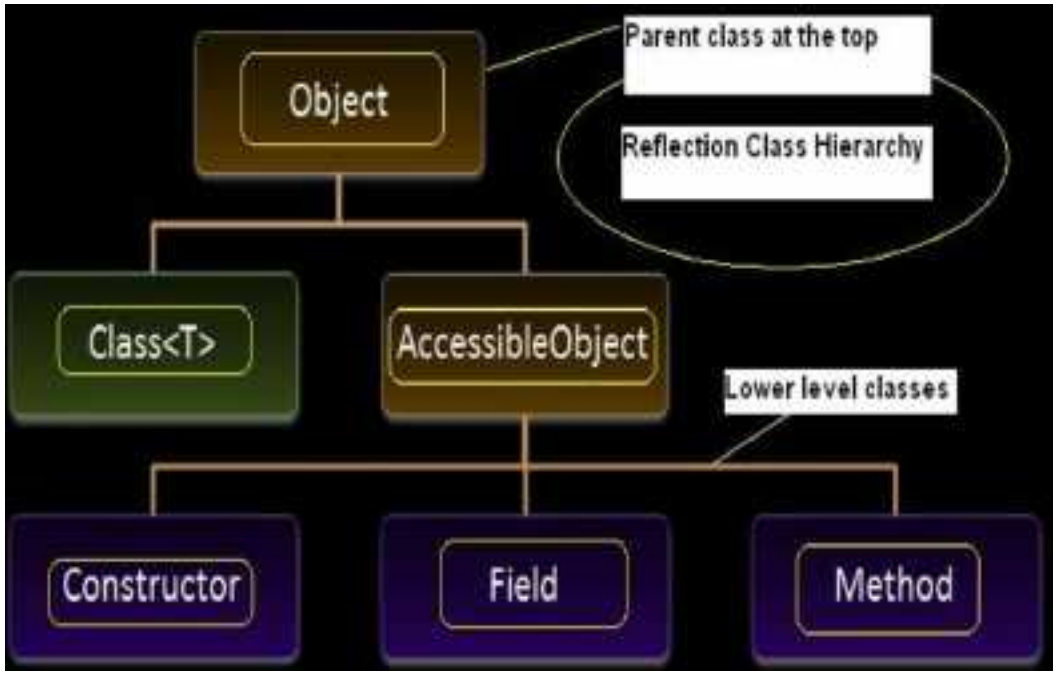
Why null?

Example:
https://www.liaoxuefeng.com/wiki/1252599548343744/1264803678201760

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 `setAccesible(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



- 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
- This permits sophisticated applications with sufficient privilege, such as Java Object Serialization or other persistence mechanisms, to manipulate objects in a manner that would normally be prohibited.

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

AccessibleObject methods

boolean	**canAccess**(**Object** obj)
void	**setAccessible**(boolean flag)
static void	**setAccessible**(**AccessibleObject**[] array, boolean flag)
boolean	**trySetAccessible**()

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

Instantiation using Reflection

- Using `newInstance()`
  - Creates a new instance of the class represented by this Class object.
  - The class is instantiated as if by a new expression with an empty argument list.

```
public T newInstance()
                throws InstantiationException,
                       IllegalAccessException
```

```
Class cls = Student.class;
Student s = (Student) cls.newInstance();
```

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);
```

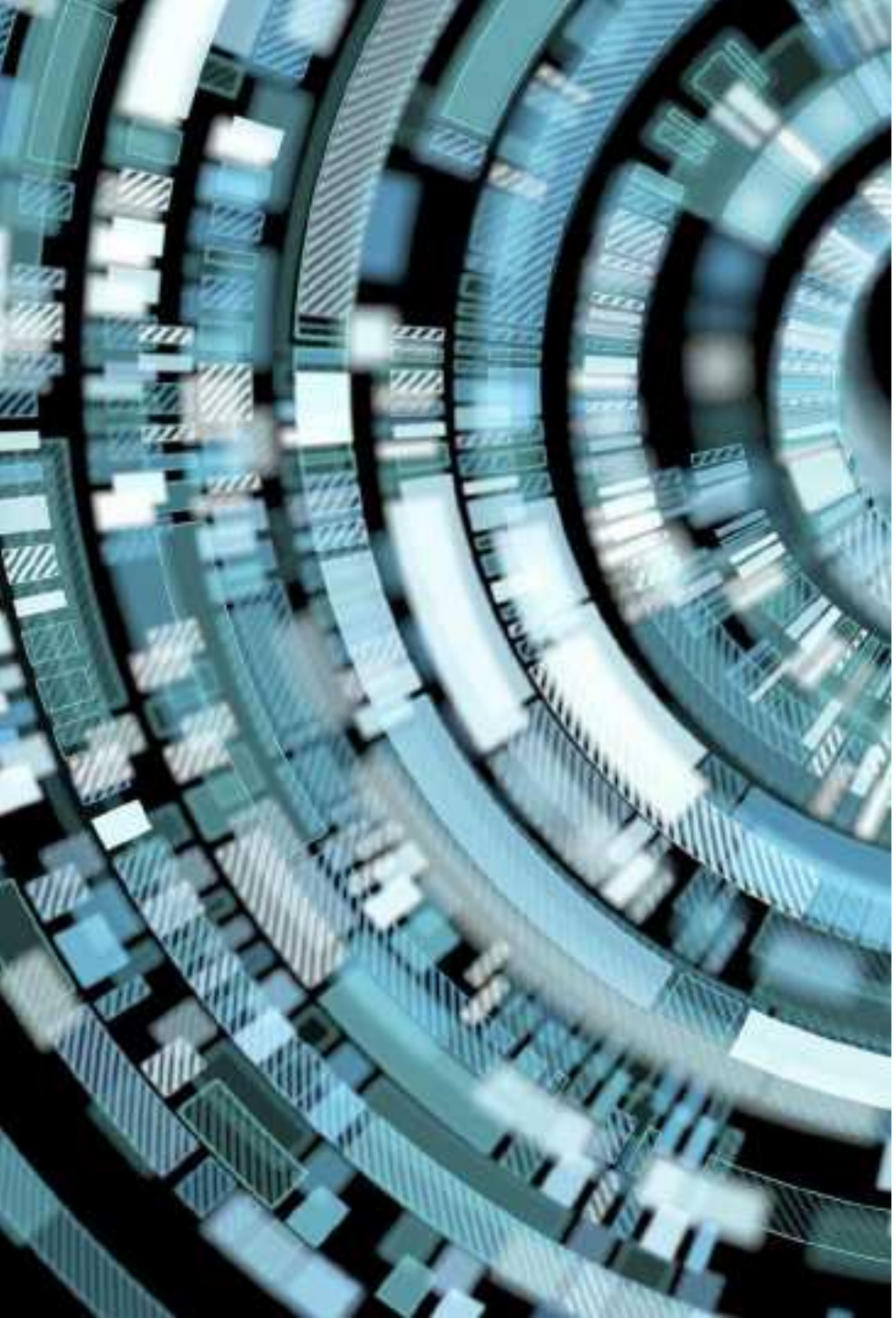
Inspecting Class Inheritance with Reflection

```
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

- **Code analyzer tools**: they do static analysis of syntax, show optimization tips and even report error conditions, and many more such things.
- **IDEs code auto completion**: method name suggestion
- **Marshalling and unmarshalling**: the process of transforming the memory representation of an object into a data format suitable for storage (JSON, XML)
- **Junit test cases**: previous Junit processor was using reflection to iterate over all methods in class, and find-out methods starting with *test* and run this as testcase.
- ……



Lecture 10

- Reflection
- Annotation
- Unit Testing

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 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

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

- 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

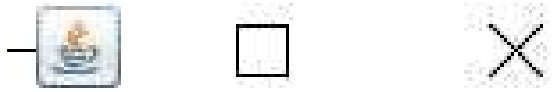
@Override

- This annotation informs the compiler that the element is meant to override an element 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(); }
}
```

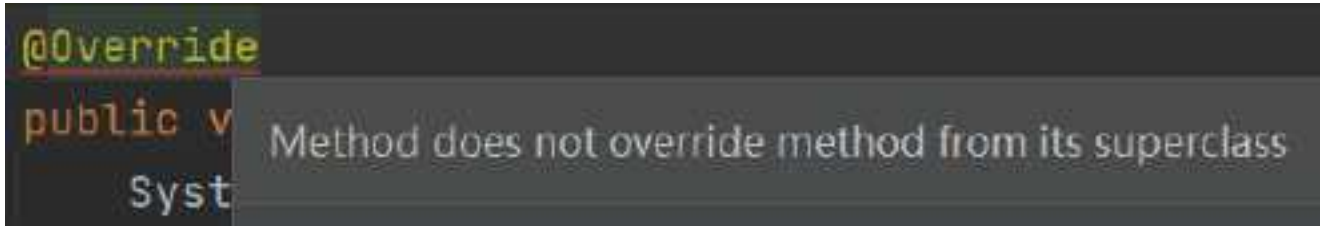


Example: https://www3.ntu.edu.sg/home/ehchua/programming/java/Annotation.html

Example of using @Override

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

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



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

```
private List list = new ArrayList();

public void addSth(String sth) {
    list.add(sth);
}
```

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

```
private List list = new ArrayList();

@SuppressWarnings("unchecked")
public void addSth(String sth) {
    list.add(sth);
}
```

@SafeVarargs

- varargs: a method has parameter(s) of variable length
- Compiler gives the warning about unsafe usage
- If we are sure that our actions are safe, we could use the @SafeVarargs annotation to suppress this warning

```java
public class SafeVarargsExample {
    public static void main(String[] args) {
        display( array: "10", 20, 30, 40.00);
    }

    @SafeVarargs
    public static <T> void display(T... array) {
        for (T arg : array) {
            System.out.println(arg.getClass().getName());
        }
    }
}
```

Custom Annotations

Meta-annotations go here

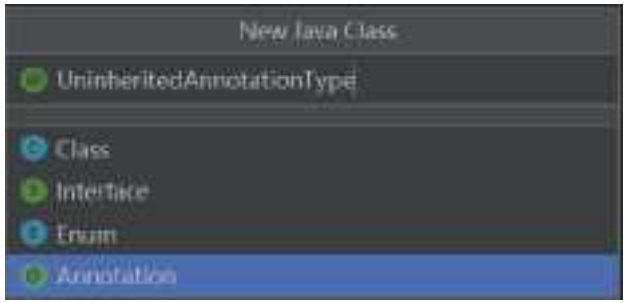
```
[Access Specifier] @interface<AnnotationName> {
  DataType <Method Name>() [default value];
}
```

It is also possible to create your own custom annotations.

- 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 parameters cannot have a null value.
- 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

Annotation Type



- An annotation type class implicitely extends the marker interface java.lang.annotation.Annotation
- Annotation type declarations are similar to normal interface declarations
- The @interface keyword is used to declare a new annotation type

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). Annotation types are a form of *interface*, which will be covered in a later lesson. For the moment, you do not need to understand interfaces.

Example

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

The body of the previous annotation definition contains *annotation type element* declarations, which look a lot like methods. default values.

After the annotation type is defined, you can use annotations of that type, with the values filled in, like this:

```
@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

}
```

What's the point?

Meta-annotations

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

@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

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

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

```java
@Target({ElementType.TYPE,ElementType.FIELD,ElementType.METHOD,ElementType.PARAMETER,ElementType.CONSTRUCTOR,
@Retention(RetentionPolicy.SOURCE)
public @interface SuppressWarnings
extends annotation.Annotation
```

@Retention

- This annotation specifies how an annotation is stored (at which level it is available)
- Syntax: `@Retention(RetentionPolicy)`
- 3 types of RententionPolicy
  - `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(value=METHOD)
 @Retention(value=SOURCE)
public @interface Override
```

```
@Target(value={TYPE,FIELD,METHOD,PARA
 @Retention(value=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 bytecode manipulation/processing tools (without interfering with runtime behaviors)
  - Code obfuscation: @KeepName
  - See https://stackoverflow.com/questions/3849593/java-annotations-looking-for-an-example-of-retentionpolicy-class

RetentionPolicy.RUNTIME

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

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

```java
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;
    }
}
```

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

RetentionPolicy.RUNTIME

```java
public static void check(Person person) throws IllegalArgumentException, ReflectiveOperationException {
    // 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());
                }
            }
        }
```

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

RetentionPolicy.RUNTIME

```
Person p1 = new Person( name: "Alice",  city: "Beijing",  age: 20);
Person p2 = new Person( name: "a",  city: "Beijing", age: 12);
Person p3 = new Person( name: "Alice",  city: "The city name is Beijing",  age: 30);
Person p4 = new Person( name: "Alice",  city: "Shenzhen",  age: 130);

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
```

@Inherited

- @Inherited annotation indicates that the annotation type can be inherited from the super class
- Subclasses of annotated classes are considered having the same annotation as their superclass.

```java
@Inherited
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface InheritedAnnotationType {
}
```

```java
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface UninheritedAnnotationType {
}
```

```java
@UninheritedAnnotationType
class A {

}

@InheritedAnnotationType
class B extends A {

}

class C extends B {

}
```

@Inherited

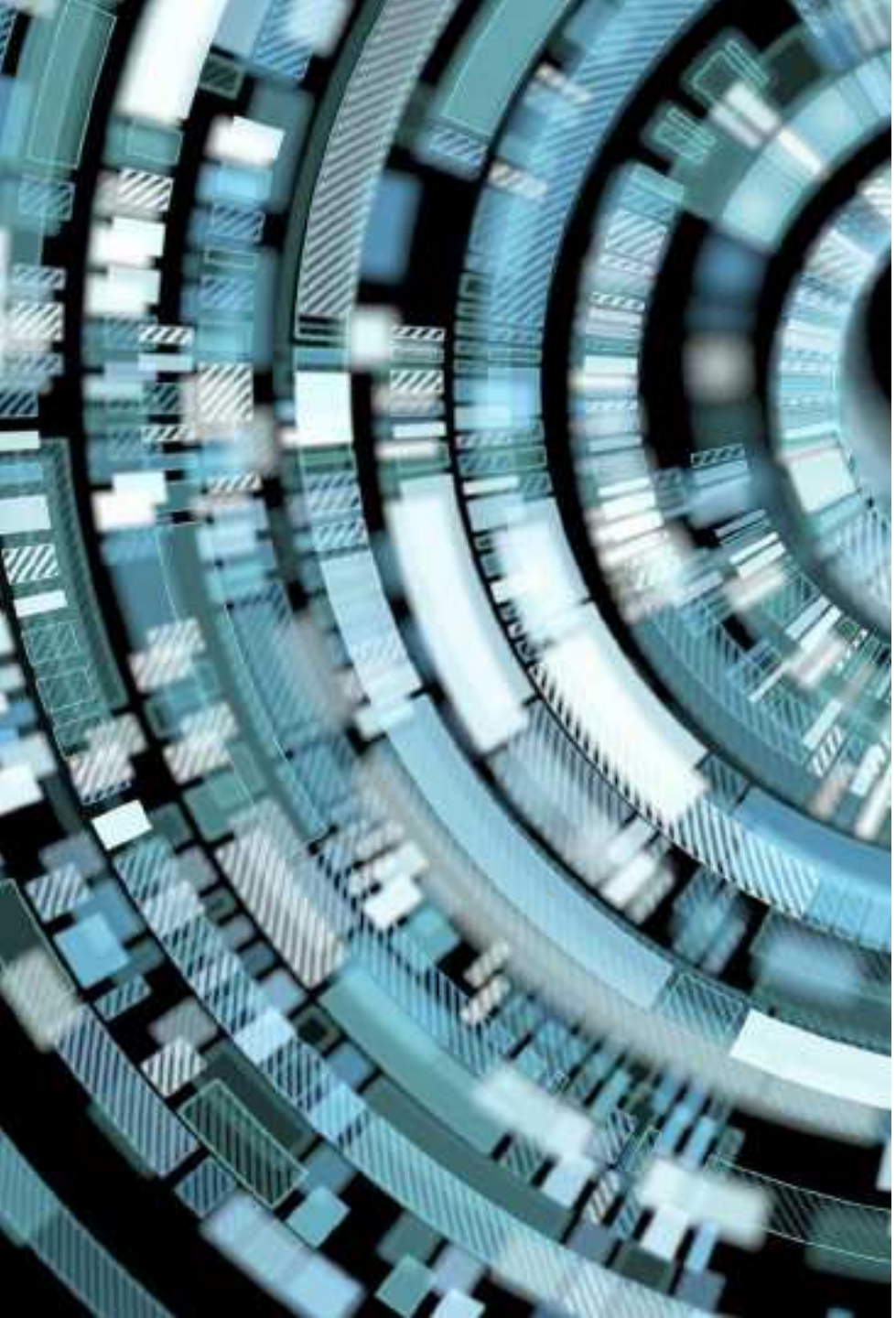
```java
@UninheritedAnnotationType
class A {

}

@InheritedAnnotationType
class B extends A {

}

class C extends B {

}
```

```java
System.out.println(new A().getClass().isAnnotationPresent(InheritedAnnotationType.class));
System.out.println(new B().getClass().isAnnotationPresent(InheritedAnnotationType.class));
System.out.println(new C().getClass().isAnnotationPresent(InheritedAnnotationType.class));
System.out.println("_____");
System.out.println(new A().getClass().isAnnotationPresent(UninheritedAnnotationType.class));
System.out.println(new B().getClass().isAnnotationPresent(UninheritedAnnotationType.class));
System.out.println(new C().getClass().isAnnotationPresent(UninheritedAnnotationType.class));
```

```
false
true
true
_____
true
false
false
```

Other meta-annotations

- @Documented annotation indicates that whenever the specified annotation is used, those elements should be documented using the Javadoc tool
- @Repeatable indicates that the marked annotation can be applied more than once to the same declaration or type use. See https://docs.oracle.com/javase/tutorial/java/annotations/repeating.html for more info.



Lecture 10

- Reflection
- Annotation
- Unit testing

Software Testing

- Software testing is the process of evaluating and verifying that a software product or application does what it is supposed to do.
- It involves execution of software/system components using manual or automated tools to evaluate one or more properties of interest.
- The benefits of testing include preventing bugs, reducing development costs and improving performance.

https://www.ibm.com/topics/software-testing

Types of Software Testing

- **Unit Test**: Test individual method/class in isolation. A unit is the smallest testable component of an application.
- **Integration Test**: Test a group of associated components/classes and ensure that they operate together.
- **Acceptance Test**: operate on a fully integrated system, testing against the user interface
- **Regression Test**: Tests to ensure that a change does not break the system or introduce new faults.
- ……. (there are more than 150 types of testing types and still adding)

https://www3.ntu.edu.sg/home/ehchua/programming/java/JavaUnitTesting.html

Code Coverage

- A measurement of how well your test set is covering your source code (i.e. to what extent is the source code covered by the set of test cases).
- It is generally accepted that 80% coverage is a good goal to aim for.
- Granularity
  - Statements/blocks/methods coverage
  - Condition/Decision/Loop coverage

JUnit 5

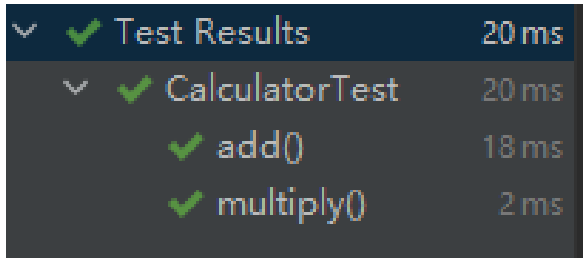
- JUnit is an open-source Unit Testing Framework for Java
- Initially designed by Erich Gamma and Kent Beck
- JUnit 5
  - JUnit 5 is the latest version and uses the new org.junit.jupiter package for its annotations and classes
  - JUnit 5 leverages features from Java 8 or later, such as lambda functions, making tests more powerful and easier to maintain.
  - JUnit 5 has added some very useful new features for describing, organizing, and executing tests

A Simple JUnit Example

```java
public class Calculator {

    public double add(double... operands) {
        return DoubleStream.of(operands)
                .sum();
    }

    public double multiply(double... operands) {
        return DoubleStream.of(operands)
                .reduce( identity: 1, (a, b) -> a * b);
    }
}
```

```java
class CalculatorTest {

    @Test
    void add() {
        Calculator c = new Calculator();
        assertEquals( expected: 4, c.add(2, 2));
    }

    @Test
    void multiply() {
        Calculator c = new Calculator();
        assertEquals( expected: 6, c.multiply( operands: 3, 2, 1));
    }
}
```

- @Test annotation denotes that this method is a test method
- Assertions is a collection of utility methods that support asserting conditions in tests.
- Run the test class CalculatorTest will execute all its test methods



Test Classes and Methods

- **Test Class**: any class that contains at least one test method. Test classes must not be abstract and must have a single constructor.
- **Test Method**: any instance method that is directly annotated or meta-annotated with @Test, @RepeatedTest, @ParameterizedTest, @TestFactory, or @TestTemplate.
- **Lifecycle Method:** any method that is directly annotated or meta-annotated with @BeforeAll, @AfterAll, @BeforeEach, or @AfterEach

https://junit.org/junit5/docs/current/user-guide/#writing-tests-classes-and-methods

Test Classes and Methods

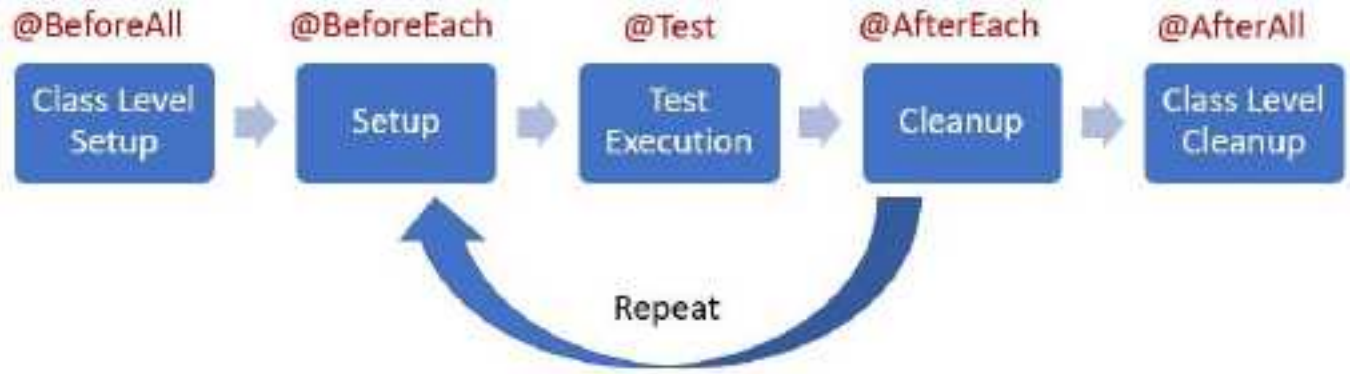
- Test methods and lifecycle methods may be declared locally within the current test class, inherited from superclasses, or inherited from interfaces
- Test methods and lifecycle methods must **not** be abstract and must **not** return a value (except @TestFactory methods which are required to return a value).
- Test classes, test methods, and lifecycle methods are **not** required to be public, but they **must not** be private

https://junit.org/junit5/docs/current/user-guide/#writing-tests-classes-and-methods

Test Lifecycle

The complete lifecycle of a test case can be seen in 3 phases

1. **Setup**: This phase puts the test infrastructure in place. JUnit provides class level setup (`@BeforeAll`) and method level setup (`@BeforeEach`). Generally, heavy objects like database connections are created in class level setup while lightweight objects like test objects are reset in the method level setup.
2. **Test Execution**: In this phase, the test execution and assertion happen, and results signify a success or failure.
3. **Cleanup**: This phase is used to cleanup the test infrastructure setup in the first phase. Just like setup, teardown also happen at class level (`@AfterAll`) and method level (`@AfterEach`).



Reference: https://howtodoinjava.com/junit5/junit-5-test-lifecycle/

@BeforeEach & @AfterEach

```
class CalculatorTest {

    @Test
    void add() {
        Calculator c = new Calculator();
        assertEquals( expected: 4, c.add(2, 2));
    }

    @Test
    void multiply() {
        Calculator c = new Calculator();
        assertEquals( expected: 6, c.multiply( ...operands: 3, 2, 1));
    }
}
```

- @BeforeEach is used to signal that the annotated method should be executed before each @Test method in the current test class.
- @BeforeEach methods must have a void return type, must not be private, and must not be static

```
class CalculatorTest {

    Calculator c;

    @BeforeEach
    public void setUp() {
        this.c = new Calculator();
    }

    @AfterEach
    public void tearDown() {
        this.c = null;
    }

    @Test
    void add() {
        assertEquals( expected: 4, c.add(2, 2));
    }

    @Test
    void multiply() {
        assertEquals( expected: 6, c.multiply( ...operands: 3, 2));
    }
}
```

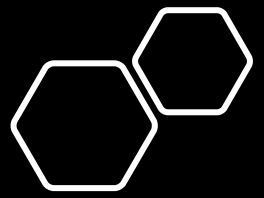
@BeforeAll & @AfterAll

- Generally, heavy objects like database connections are created in class level setup

```
public class DatabaseTest {
    static Database db;

    @BeforeAll
    public static void initDatabase() {
        db = createDb(...);
    }

    @AfterAll
    public static void dropDatabase() {
        ...
    }
}
```

- @BeforeAll is used to signal that the annotated method should be executed before all tests in the current test class.
- In contrast to @BeforeEach methods, @BeforeAll methods are only executed once for a given test class.
- @BeforeAll methods must have a void return type, must not be private, and must be static by default

Image: https://www.liaoxuefeng.com/wiki/1252599548343744/1304049490067490



Test Instance Lifecycle

- In order to allow individual test methods to be executed in isolation and to avoid unexpected side effects due to mutable test instance state, JUnit creates a new instance of each test class before executing each test method (default)

- If you would prefer that JUnit Jupiter execute all test methods on the same test instance, annotate your test class with @TestInstance(Lifecycle.PER_CLASS)
  - A new test instance will be created once per test class.
  - If your test methods rely on state stored in instance variables, you may need to reset that state in @BeforeEach or @AfterEach methods.

Further reading:
https://junit.org/junit5/docs/current/user-guide/#writing-tests-test-instance-lifecycle

Assertions

```
@Test
void standardAssertions() {
    assertEquals( expected: 2, Calculator.add(1, 1));

    assertEquals( expected: 4, Calculator.multiply( ...operands: 2, 2),
            message: "The optional failure message");

    assertTrue( condition: Calculator.add(1, 1) == 2);

    assertArrayEquals(new int[]{1,2,3},new int[]{1,2,3});

    assertNull( actual: null);

}
```

java.lang.Object

org.junit.jupiter.api.Assertions

Assertions is a class/collection of utility methods that support asserting conditions in tests.

If one assert fails, the test will stop and you won't see the results of the rest asserts

assertAll

```
public static void assertAll(String heading,
                             Executable... executables)
                      throws MultipleFailuresError
```

Asserts that *all* supplied `executables` do not throw exceptions.

```
Address address = unitUnderTest.methodUnderTest();
assertAll("Should return address of Oracle's headquarter",
    () -> assertEquals("Redwood Shores", address.getCity()),
    () -> assertEquals("Oracle Parkway", address.getStreet()),
    () -> assertEquals("500", address.getNumber())
);
```

```
org.opentest4j.MultipleFailuresError:
    Should return address of Oracle's headquarter (3 failures)
    expected: <Redwood Shores> but was: <Walldorf>
    expected: <Oracle Parkway> but was: <Dietmar-Hopp-Allee>
    expected: <500> but was: <16>
```

If any supplied Executable throws an AssertionError, all remaining executables will still be executed, and all failures will be aggregated and reported in a MultipleFailuresError.

Example: https://stackoverflow.com/questions/40796756/assertall-vs-multiple-assertions-in-junit5

Next Lecture

- Web Applications