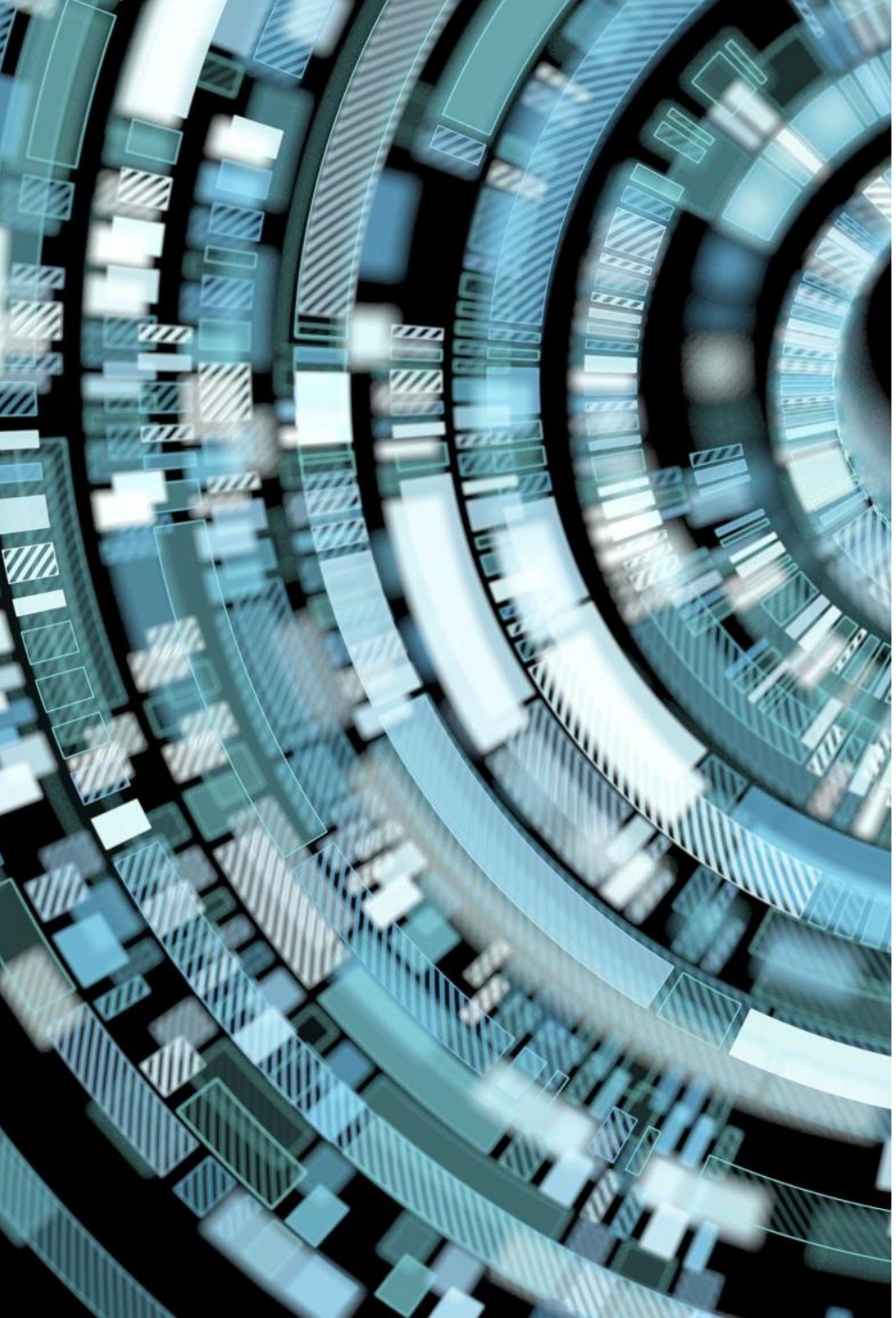
Computer System Design & Application

计算机系统设计与应用A

陶伊达 （TAO Yida）
taoyd@sustech.edu.cn



Lecture 11

- Software Testing Overview
- JUnit 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 considered (?) that 80% coverage is a good goal to aim for.
- Granularity
  - Statements/blocks/methods coverage
  - Condition/Decision/Loop coverage

JUnit 5

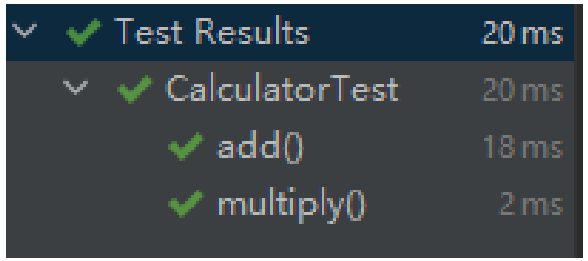
- JUnit is an open-source Unit Testing Framework for Java
- Initially designed by Erich Gamma and Kent Beck
- JUnit 5
  - JUnit 5 is the latest version and uses the new `org.junit.jupiter` package for its annotations and classes
  - JUnit 5 leverages features from Java 8 or later, such as lambda functions, 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 annotated with @Test, @RepeatedTest, @ParameterizedTest, @TestFactory, or @TestTemplate.
- **Lifecycle Method:** any method that is 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 Instance Lifecycle

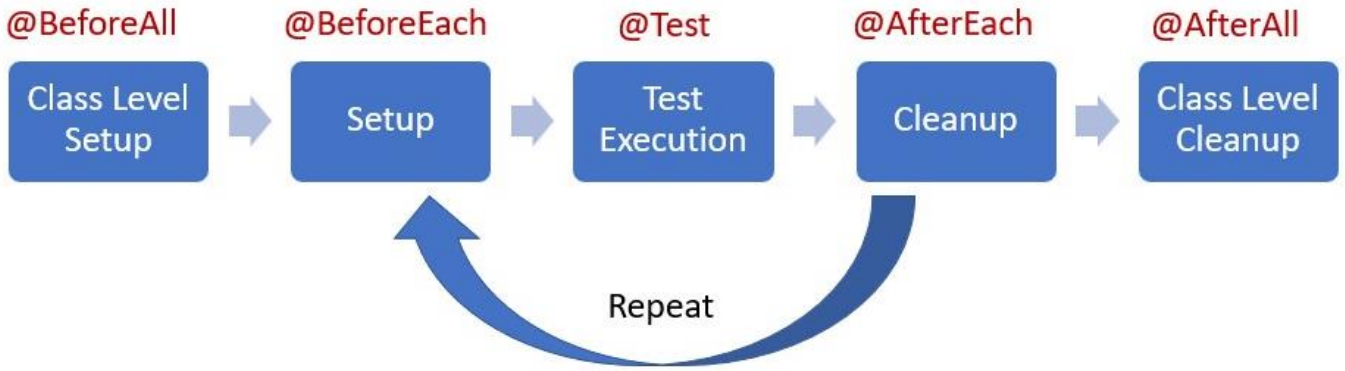
In order to allow individual test methods to be executed in isolation and to avoid unexpected side effects due to mutable test instance state, JUnit creates a new instance of each test class before executing each test method (default)

```
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 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

```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));
    }
}
```

- @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

```java
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));
    }
}
```

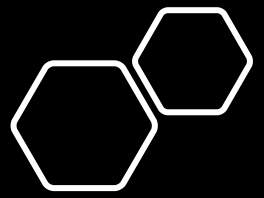
@BeforeAll & @AfterAll

- 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 (unless the PER_CLASS test instance lifecycle is used)

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

```java
@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 remaining 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

Assumptions

- Assumptions is a collection of utility methods that support conditional test execution based on assumptions.
- In contrast to failed assertions, which result in a test failure, a failed assumption results in a test being *aborted*.
- Assumptions are typically used whenever it does not make sense to continue execution of a given test method (e.g., if the test depends on something that does not exist in the current runtime environment)

https://junit.org/junit5/docs/5.0.0/api/org/junit/jupiter/api/Assumptions.html

Assumptions

```java
private final Calculator calculator = new Calculator();

@Test
void testOnlyOnCiServer() {
    assumeTrue("CI".equals(System.getenv("ENV")));
    // remainder of test
}

@Test
void testOnlyOnDeveloperWorkstation() {
    assumeTrue("DEV".equals(System.getenv("ENV")),
        () -> "Aborting test: not on developer workstation");
    // remainder of test
}

@Test
void testInAllEnvironments() {
    assumingThat("CI".equals(System.getenv("ENV")),
        () -> {
            // perform these assertions only on the CI server
            assertEquals(2, calculator.divide(4, 2));
        });

    // perform these assertions in all environments
    assertEquals(42, calculator.multiply(6, 7));
}
```

https://junit.org/junit5/docs/current/user-guide/#writing-tests-assumptions

Conditional Test Execution

- Entire test classes or individual test methods may be disabled via the `@Disabled` annotation

```java
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@API(
    status = Status.STABLE,
    since = "5.0"
)
public @interface Disabled {
    String value() default "";
}
```

- Developers could either enable or disable a test based on certain conditions programmatically

```java
@Test
@EnabledOnOs({ LINUX, MAC })
void onLinuxOrMac() {
    // ...
}

@Test
@DisabledOnOs(WINDOWS)
void notOnWindows() {
    // ...
}
```

```java
@Test
@EnabledForJreRange(min = JAVA_9, max = JAVA_11)
void fromJava9to11() {
    // ...
}

@Test
@DisabledForJreRange(min = JAVA_9)
void notFromJava9toCurrentJavaFeatureNumber() {
    // ...
}
```

https://junit.org/junit5/docs/current/user-guide/#writing-tests-conditional-execution

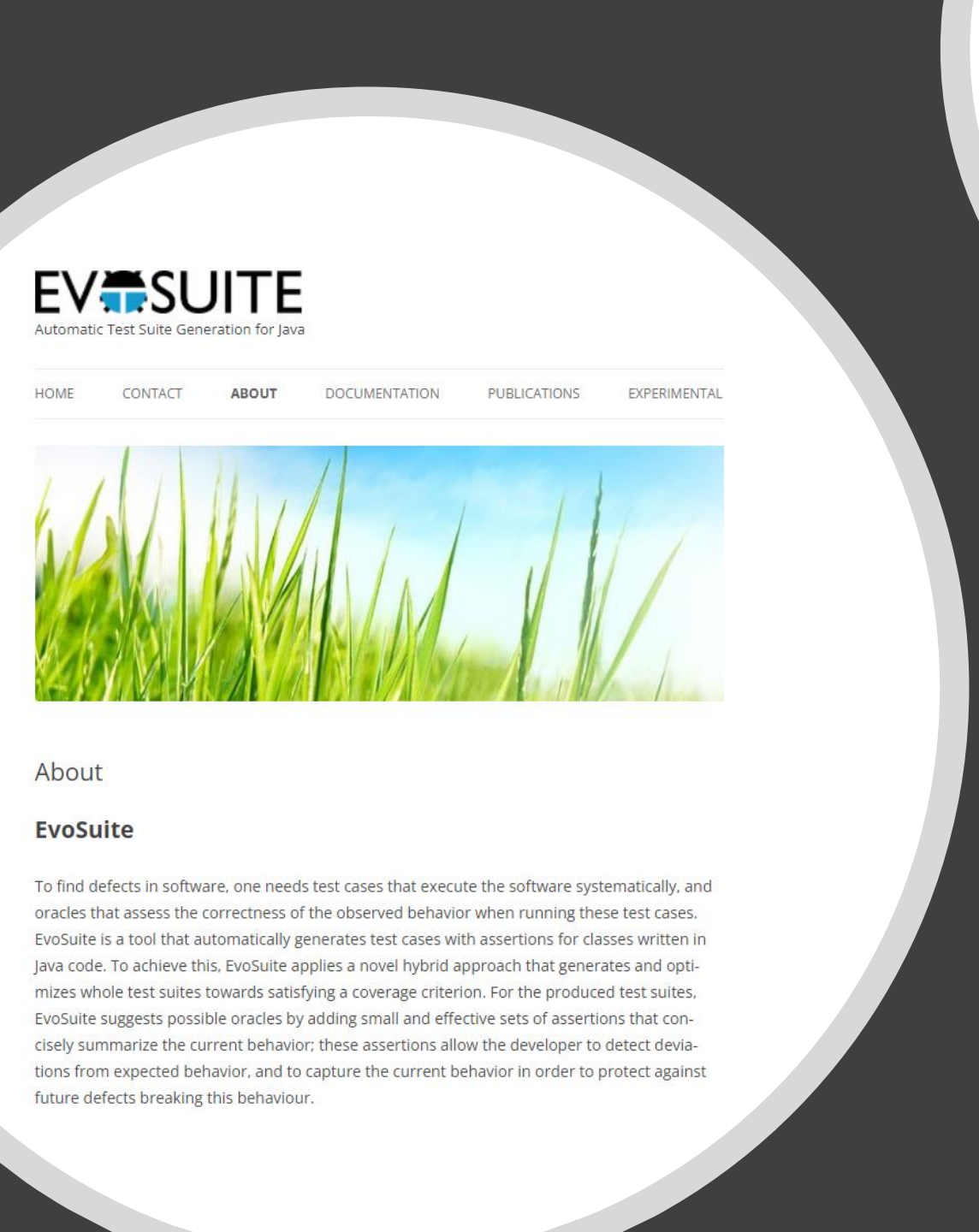
Parameterized Test

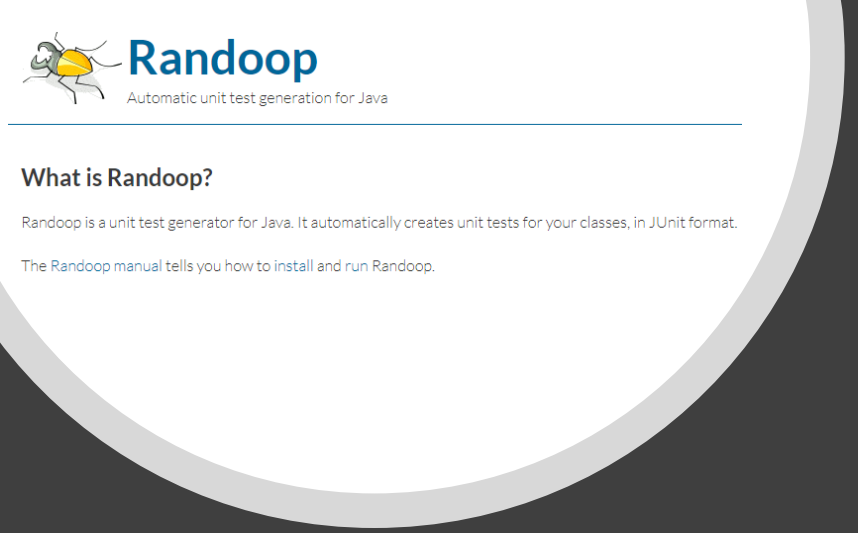
- Parameterized tests make it possible to run a test multiple times with different arguments.
- Use the @ParameterizedTest annotation
- you must declare at least one source that will provide the arguments for each invocation and then consume the arguments in the test method.

```
@ParameterizedTest
@ValueSource(strings = { "racecar", "radar", "able was I ere I saw elba" })
void palindromes(String candidate) {
    assertTrue(StringUtils.isPalindrome(candidate));
}
```

```
palindromes(String) ✓
 ├─ [1] candidate=racecar ✓
 ├─ [2] candidate=radar ✓
 └─ [3] candidate=able was I ere I saw elba ✓
```

https://junit.org/junit5/docs/current/user-guide/#writing-tests-parameterized-tests





Automatic Test Generation for Java

Build Tools

- Build tools are programs that automate the creation of executable applications from source code

- Build automation typically include:
  - Downloading dependencies.
  - Compiling source code
  - Running tests.
  - Packaging compiled code to a distributable format (e.g., .jar).
  - Deployment to production systems.

- Java build tools: Maven, Gradle, Ant

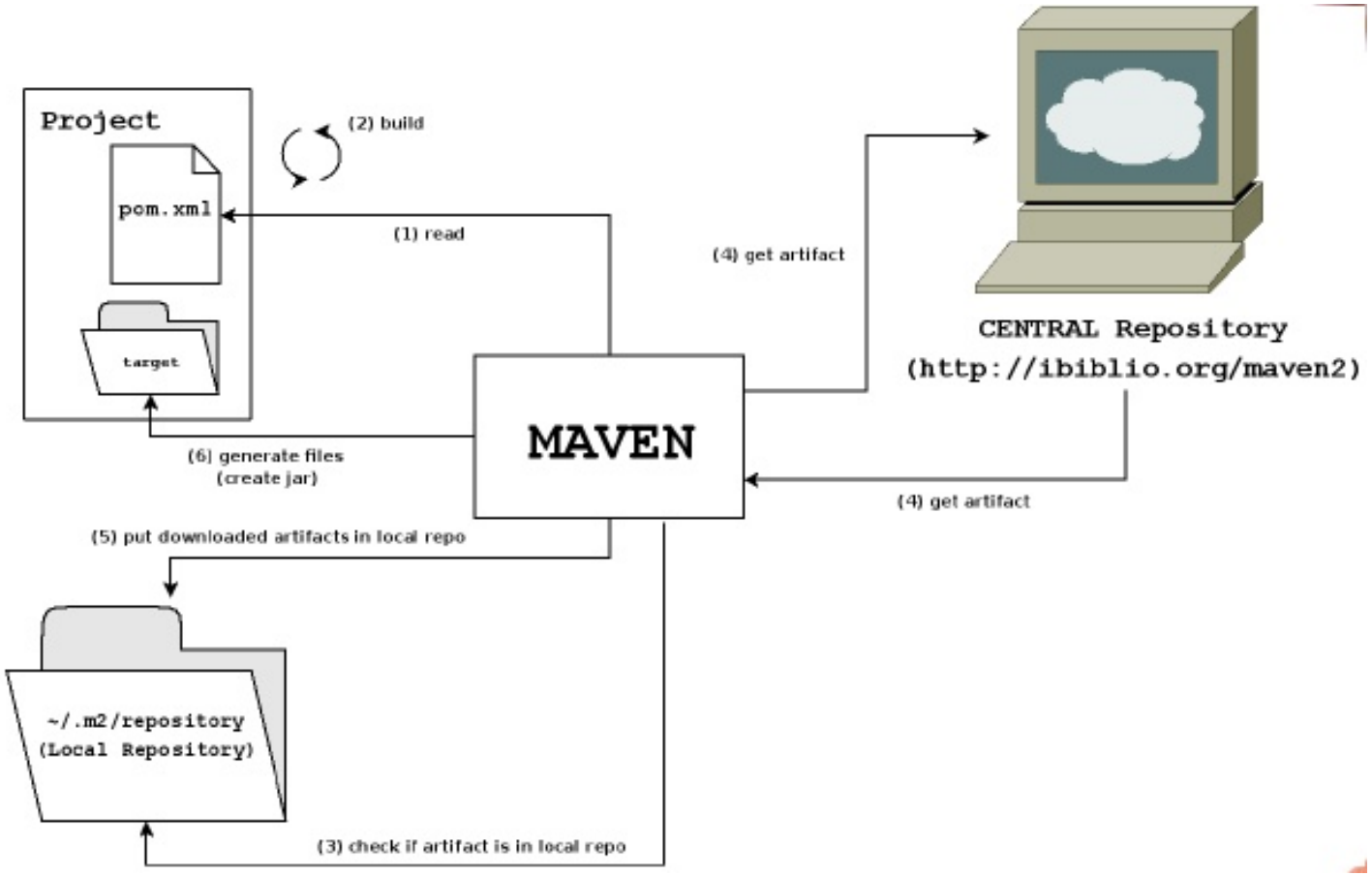
Maven POM

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1.0-SNAPSHOT</version>

  <properties>
    <maven.compiler.source>1.7</maven.compiler.source>
    <maven.compiler.target>1.7</maven.compiler.target>
  </properties>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

- POM stands for "Project Object Model". It is an XML representation of a Maven project held in a file named pom.xml.
- The pom.xml file is the core of a project's configuration in Maven.
- It is a single configuration file that contains the majority of information required to build a project in just the way you want.

https://maven.apache.org/guides/getting-started/maven-in-five-minutes.html#

Maven Workflow



https://www.slideshare.net/sandeepchawla/maven-introduction

Next Lecture

- Java EE