Java reflection, proxies, and annotations

Tapti Palit

UCDAVIS

Motivation

- Annotations and reflection is widely used in frameworks
- Microservice, dependency injection, persistence frameworks

Java annotations preview

- Annotations are "metadata" added to Java code
- By themselves, they have no direct impact on code execution at runtime
- Java provides some annotations, programmer can define more
- Syntax: `@Annotation_name`

```
@Annotation1
public class Car {
    @Annotation2
    private int speed;


    public Car(int speed) {
        this.speed = speed;
    }

    @Annotation3
    public void accelerate(int moreSpeed) {
        this.speed += moreSpeed;
    }

    @Annotation3
    public void decelerate(int lessSpeed) {
        this.speed -= lessSpeed;
    }
}
```

UCDAVIS

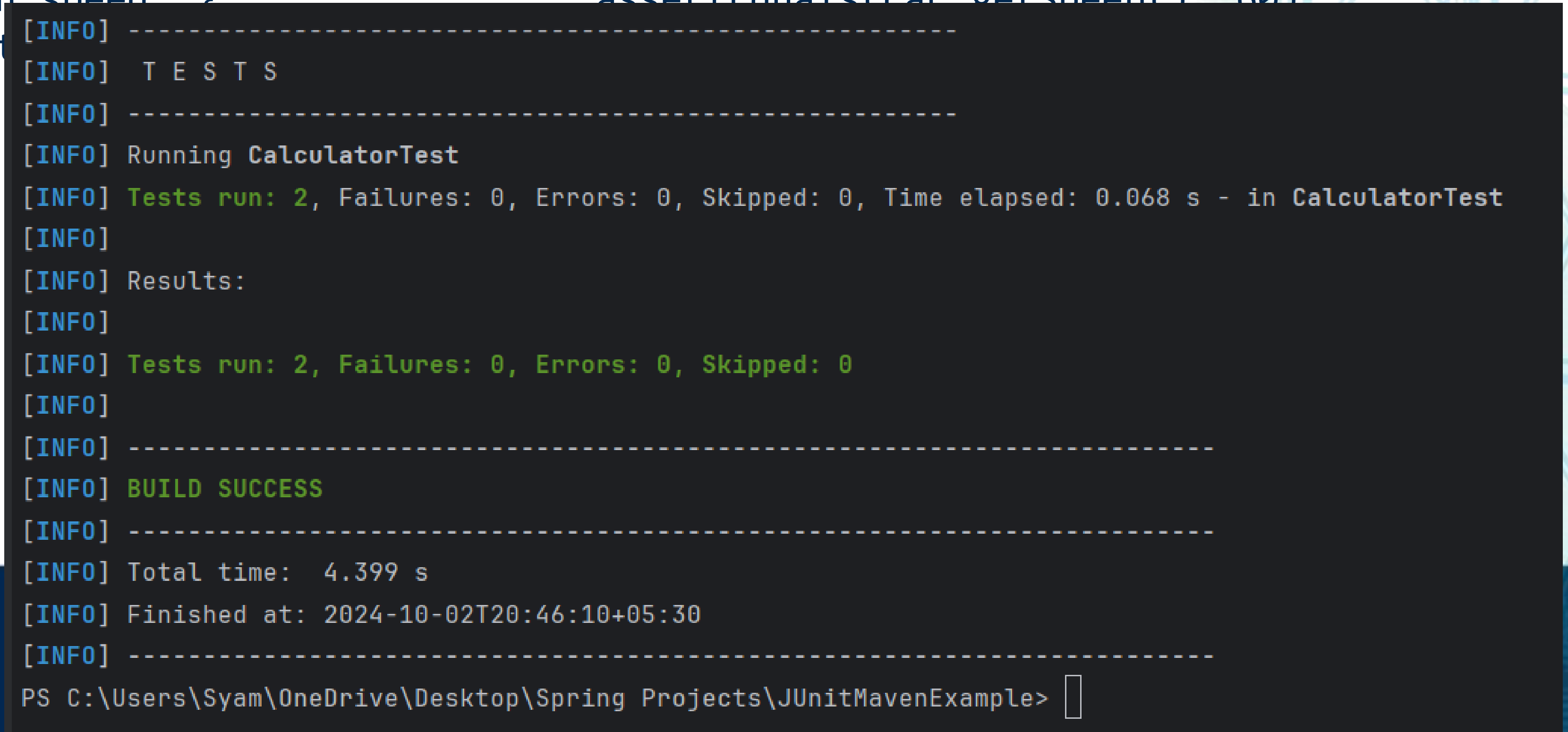
Annotation targets

- Annotations can be applied to
  - Classes
  - Fields
  - Methods
  - … many other program elements (Check https://docs.oracle.com/javase/tutorial/java/annotations/basics.html)

Motivation: Junit

```java
public class Car {
    private String model;
    private int speed;

    public Car(String model, int speed) {
        this.model = model;
        this.speed = speed;
    }

    public int getSpeed() { return speed; }
    public String getModel() { ret

    ... // setters

    public void accelerate() {
        this.speed += 10;
    }
}
```

```java
import org.junit.jupiter.api.Test;
import
org.junit.jupiter.api.Assertions.assertEquals;

public class CarTest {
    @Test
    public void testAccelerate() {
        Car car = new Car("Toyota", 50);
        car.accelerate();
        assertEquals(car.getSpeed(), 60);
```



```
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running CalculatorTest
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.068 s - in CalculatorTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  4.399 s
[INFO] Finished at: 2024-10-02T20:46:10+05:30
[INFO] ------------------------------------------------------------------------
PS C:\Users\Syam\OneDrive\Desktop\Spring Projects\JUnitMavenExample>
```

Motivation: Hibernate

- Automatically load and store Java objects from a SQL database

users

id	name

```
@Entity
@Table(name = "users")
public class User {

    @Id
    private Long id;

    private String name;

    protected User() {} // required by Hibernate

    public User(String name) {
        this.name = name;
    }

    public Long getId() { return id; }
    public String getName() { return name; }
}

 public static void main(String[] args) {
        SessionFactory factory = ...;
        Session session = factory.openSession();

        User u = new User("Alice");
        session.persist(u);

 }
```

UCDAVIS

Motivation: Hibernate

- Automatically load and store Java objects from a SQL database

users

id	name

Hibernate annotations

```
@Entity
@Table(name = "users")
public class User {

    @Id
    private Long id;

    private String name;

    protected User() {} // required by Hibernate

    public User(String name) {
        this.name = name;
    }

    public Long getId() { return id; }
    public String getName() { return name; }
}
```

Hibernate classes

```
 public static void main(String[] args) {
        SessionFactory factory = ...;
        Session session = factory.openSession();

        User u = new User("Alice");
        session.persist(u);

 }
```

UCDAVIS

Motivation: Hibernate

- Automatically load and store Java objects from a SQL database

users

id	name
102811	Alice

```
@Entity
@Table(name = "users")
public class User {

    @Id
    private Long id;

    private String name;

    protected User() {} // required by Hibernate

    public User(String name) {
        this.name = name;
    }

    public Long getId() { return id; }
    public String getName() { return name; }
}

 public static void main(String[] args) {
       SessionFactory factory = ...;
       Session session = factory.openSession();

       User u = new User("Alice");
       session.persist(u);

 }
```

Reflection and metaprogramming

- Ability of a program to inspect and manipulate its own structure and behavior at runtime **within the same language environment**
- Can (dynamically) instantiate classes given a **class name** and invoke methods and constructors on it
- Types of Reflection
  - Run time (now)
  - Compile time (later)
- Supported by Java, Go, Javascript

Java reflection

- Provides APIs to inspect and manipulate the classes, methods, fields, and so on…
- Note: reflection bypasses all encapsulation guarantees
  - You can directly access private class fields and methods from outside the class using reflection
  - But (hopefully) for greater good!!

Key classes

- java.lang.Class: Represents metadata for a class or interface
- `java.lang.reflect.Method`: Represents metadata for a class method
- `java.lang.reflect.Field`: Represents metadata for a class field
- `java.lang.reflect.Constructor`: Represents metadata for a constructor
- Automatically defined by the Java runtime

public class Student
private int id;
private String name;
public void setId(..) {..}
//... other getters and setters

public class Class
private String className;
private Class<?>[] classes;
private Constructor<?>[] constructors;
private Field[] fields;
private Method[] methods;
// getters, setters, other methods

public class Method
// ... meta-info

public class Field
// ... meta-info

public class Constructor
// ... meta-info

Key classes

- Java runtime also automatically creates a `Class` object for every class you define
- Contains objects of type `Method`, `Field`, etc

public class Student
private int id;
private String name;
public void setId(..) {..}
//... other getters and setters

public class Class
private String className;
private Class<?>[] classes;
private Constructor<?>[] constructors;
private Field[] fields;
private Method[] methods;
// getters, setters, other methods

Class studentClass$1 /* object of type Class */
className = “Student”;
classes = [];
constructors = defCons$1;
fields = [field$1, field$2];
methods = [method$1, method$2];

UCDAVIS

Key classes

- Java runtime also automatically creates a `Class` object for every class you define
- Contains objects of type `Method`, `Field`, etc

public class Student
private int id;
private String name;
public void setId(..) {..}
//... other getters and setters

public class Method
private String methodName;
private Type[] argTypes;
private Type returnType;
// getters, setters, other methods

Method method$1 /* object of type Method */
methodName = “setId”;
argTypes = [IntegerType];
returnType = VoidType;

UCDAVIS

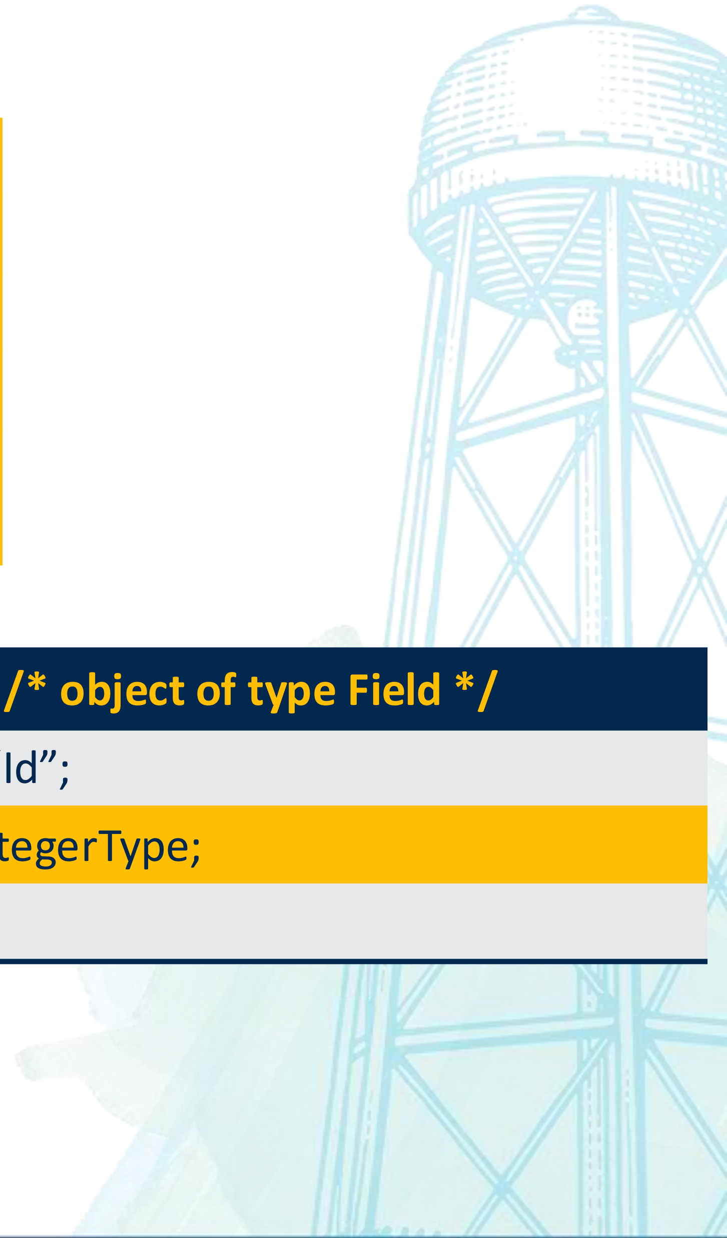
Key classes

- Java runtime also automatically creates a `Class` object for every class you define
- Contains objects of type `Method`, `Field`, etc

public class Student
private int id;
private String name;
public void setId(..) {..}
//... other getters and setters

public class Field
private String FieldName;
private Type[] fieldType
// getters, setters, other methods

Field Field$1 /* object of type Field */
fieldName = "Id";
fieldType = IntegerType;
//..



Key classes

- The programmer can manipulate these reflection objects (`Class`, `Field`, Method, etc)

public class Student
private int id;
private String name;
public void setId() {..}
//... other getters and setters

Class studentClass$1 /* object of type Class */
className = "Student";
classes = [];
constructors = defCons$1;
fields = [field$1, field$2];
methods = [method$1, method$2];

Method method$1 /* object of type Method */
methodName = "setId";
argTypes = [IntegerType];
returnType = VoidType;

Field Field$1 /* object of type Field */
fieldName = "Id";
fieldType = IntegerType;
//..

Key classes

- Reflection objects independent of class objects

public class Student
private int id;
private String name;
public void setId() {..}
//… other getters and setters

Student studentObj1
101
“ABC”

Student studentObj2
102
“DEF”

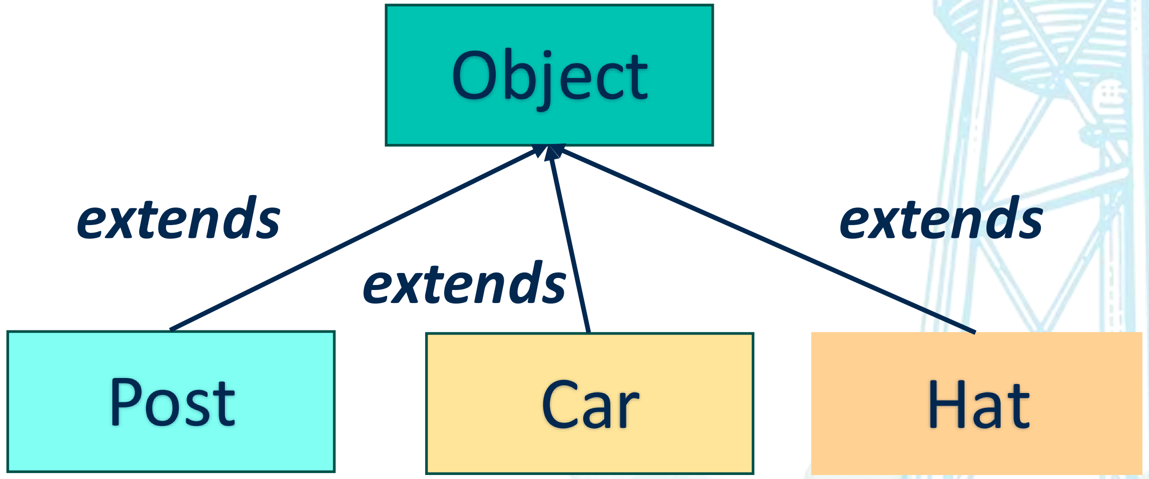
Class studentClass$1 /* object of type Class */
className = “Student”;
classes = [];
constructors = defCons$1;
fields = [field$1, field$2];
methods = [method$1, method$2];

Method method$1 /* object of type Method */
methodName = “setId”;
argTypes = [IntegerType];
returnType = VoidType;

Field Field$1 /* object of type Field */
fieldName = “Id”;
fieldType = IntegerType;
//..

Background

- In Java, all classes inherit the `Object` class implicitly



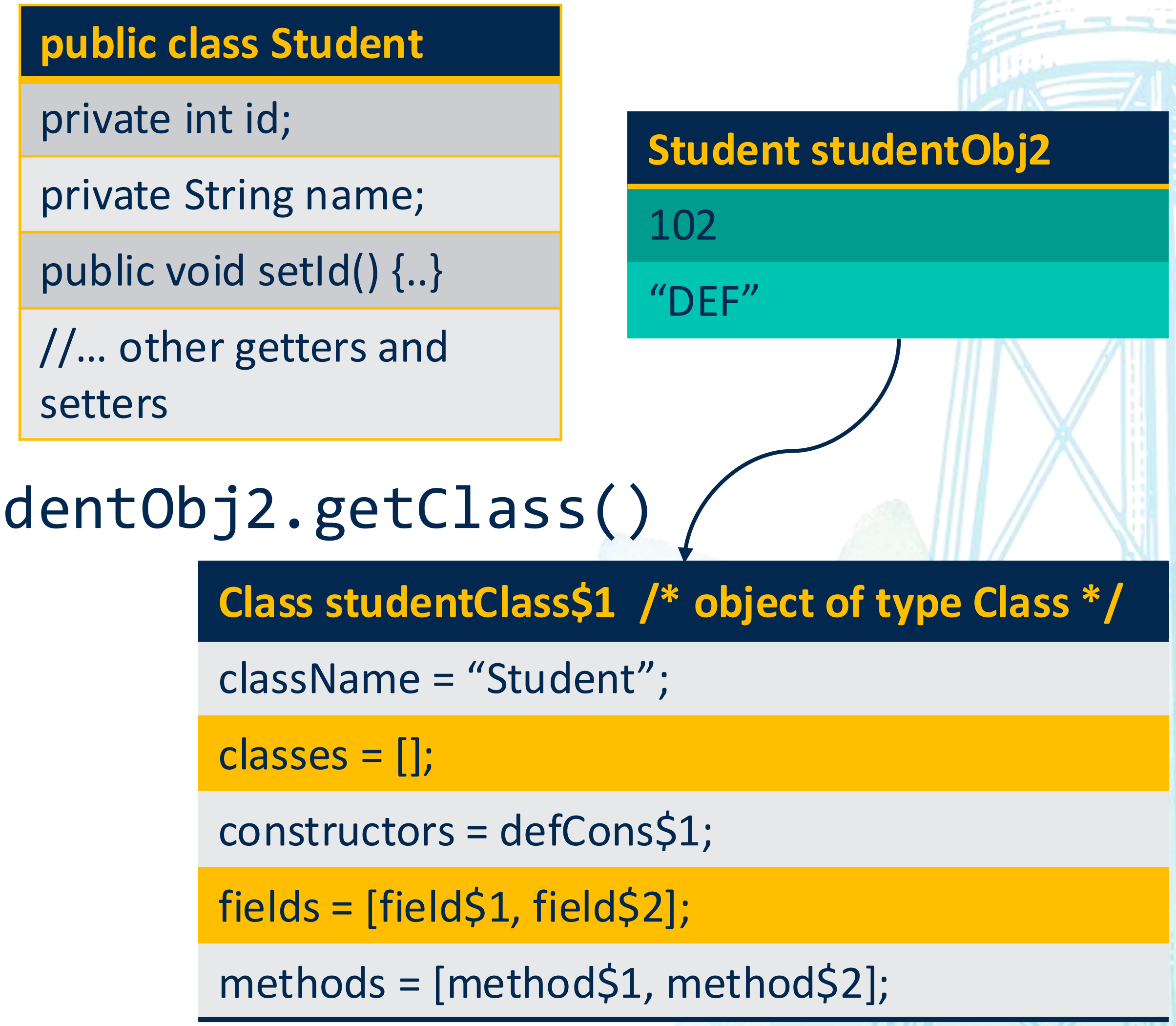
UCDAVIS

Reflection API

- `java.lang.reflect.Class`
  - An object of this type encapsulates the class metadata for a particular class
  - `Class clazz = Class.forName(“com.ecs160.MyApp”);`
  - `Class clazz = obj.getClass();`
- `clazz.getMethods():` Get an array of methods of class `clazz`
- `java.lang.reflect.Method`
  - Invoke using `m.invoke(obj, [args]);`
- `java.lang.reflect.Field`
  - Can get and set field values using `f.getValue(obj), f.setValue(obj, val)`
  - Reflection can bypass access modifiers using the `setAccessible(true)` method
- Invoke constructor to create a new Object
  - `Object o = c.getConstructor().newInstance();`

Reflection API

- `java.lang.reflect.Class`
  - An object of this type encapsulates the class metadata for a particular class
  - `Class clazz = Class.forName("com.ecs160.MyApp");`
  - `Class clazz = obj.getClass();`



```
public class Student
private int id;
private String name;
public void setId() {..}
//... other getters and setters
```

```
Student studentObj2
102
"DEF"
```

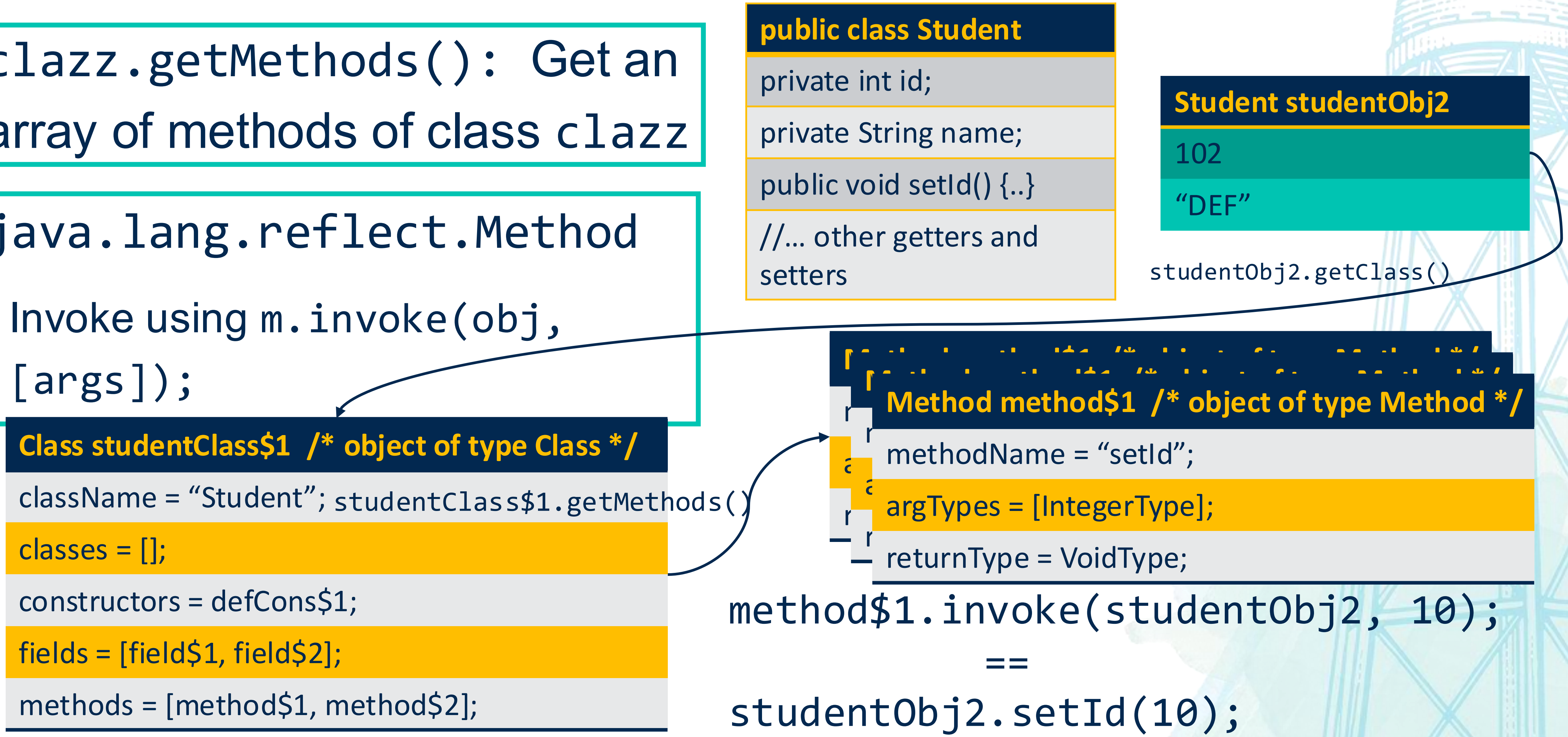
```
Class studentClass$1  /* object of type Class */
className = "Student";
classes = [];
constructors = defCons$1;
fields = [field$1, field$2];
methods = [method$1, method$2];
```

UCDAVIS

Reflection API

- `clazz.getMethods()`: Get an array of methods of class `clazz`

- `java.lang.reflect.Method`
  - Invoke using `m.invoke(obj, [args]);`



public class Student
private int id;
private String name;
public void setId() {..}
//... other getters and setters

Student studentObj2
102
"DEF"

`studentObj2.getClass()`

Class studentClass$1 /* object of type Class */
className = "Student";
classes = [];
constructors = defCons$1;
fields = [field$1, field$2];
methods = [method$1, method$2];

`studentClass$1.getMethods()`

Method method$1 /* object of type Method */
methodName = "setId";
argTypes = [IntegerType];
returnType = VoidType;

```
method$1.invoke(studentObj2, 10);
                ==
studentObj2.setId(10);
```

UCDAVIS

Reflection example

- E.g., using reflection to dynamically invoke all methods in an object

```
class Car {
    private String model;
    private int year;
    public String getModel() {return model;}
    public int getYear() { return year; }
    public Car(String model, int year) {...}
}

class MyApp {
    public static void main(String[] args) {
        Car c = new Car("Toyota", 2019);
        Class<?> clazz = c.getClass();
        for (Method m: clazz.getDeclaredMethods()) {
            Object result = m.invoke(c);
            S.o.p(result);
        }
    }
}
// Prints "Toyota" and then "2019"
```

Same reflection code works on all objects

- Using reflection on another object type
- Allows the programmer to define functionality that works on objects of ***any*** class
- Demo: https://github.com/davsec-teaching/reflection_demo/tree/master

```
class Post {
    private String authorName;
    private String content;
    private int replyCount;
    public int getAuthorName() { return authorName; }
    public int getContent() { return content; }
    public Integer getReplyCount() { return replyCount; }
    public Post(String authorName, String content,
        int replyCount) {
        this.authorName = authorName;
        this.content = content;
        this.replyCount = replyCount;
    }
}

class MyApp {
    public static void main(String[] args) {
        Post p = new Post("Tapti", "Welcome to ECS 160",
0);
        Class<?> clazz = p.getClass();
        for (Method m: clazz.getDeclaredMethods()) {
            Object result = m.invoke(p);
            S.o.p(result);
        }
    }
}
// Prints "Tapti" and then "Welcome to ECS 160" and then
"0"
```

Use case: Redis persistence framework

- Redis: REmote DIctionary Server
- In-memory data store
- Redis uses
  - As a temporary database
  - As a caching layer
  - As a message broker

$\left.\begin{array}{l}\\ \\ \\ \end{array}\right\}$ ***More in the Microservices module***

- Goal: create a persistence framework for Redis that works with ***any*** objects

UCDAVIS

Redis overview

- Redis is a key-value store
- Key: unique identifier for the record you’re storing
- Value: String or collection of values of field-value pairs (hashmap)

Key	Value	
10279811	**Field**	**Value**
	Name	ABC
	Age	22
	GPA	3.8
	Credits	45
10279812	**Field**	**Value**
	Name	DEF
	Age	21
	GPA	3.9
	Credits	60

Students

UCDAVIS

Jedis library

- Used to communicate with Redis server from local machine

```
String key = ... ;
String value = ...;
Jedis jedisSession = new Jedis("localhost", 6379);
jedisSession.set(key, value);
```

UCDAVIS

Jedis library

```
Student s = new Student(‘ABC’, 22, 3.8, 45);
Jedis jedis = new Jedis("localhost",6379);
Map<String, String> studentMap = new HashMap<>();

studentMap.put(“Name", s.getName());
studentMap.put(“Age”, s.getAge());
studentMap.put(“GPA", s.getGPA());
studentMap.put(“Credits", s.getCredits());

jedis.hset("10279811", studentMap);
```

Needs to be duplicated for each class

UCDAVIS

Why reflection?

- Goal: define `persistAll` method, independent of the object type

```
class Student { // fields of student}

class RedisDB {
    Jedis jedisSession;
    static int id;
    static {
        jedisSession = new Jedis(“localhost”, 6379);
        id = 0;
    }

    public void persistAll(Student s) {
        studentMap.put(“Name", s.getName());
        studentMap.put(“Age”, s.getAge());
        studentMap.put(“GPA", s.getGPA());
        studentMap.put(“Credits", s.getCredits());
        jedis.hset(id++, studentMap);
    }

}
```

Solution: use reflection

```
class Student { // fields of student}

class RedisDB {
    Jedis jedisSession;
    static int id;
    static {
        jedisSession = new Jedis("localhost",
6379);
        id = 0;
    }

    public void persistAll(Student s) {
        studentMap.put("Name", s.getName());
        studentMap.put("Age", s.getAge());
        studentMap.put("GPA", s.getGPA());
        studentMap.put("Credits",
s.getCredits());
        jedis.hset(id++, studentMap);
    }

}
```

```
class Student {}
class Post { }

class RedisDB {

    public void persistAll(Object obj) {
        Map<String, String> objMap;
          Class c = obj.getClass();
        for (Field f: c.getDeclaredFields()) {
           String fieldname = f.getName();
           fieldVal.setAccessible(true);
           Object fieldVal = f.get(obj);
           objMap.put(fieldname, fieldVal);
        }
        jedis.hset(id++, objMap);
    }

}
```

```
Student s = new Student("ABC", …);
Post p = new Post("Hello world", "1/23/2025");
RedisDB db = new RedisDB();
db.persistAll(s);
db.persistAll(p);
```

UCDAVIS

Loading all classes

- Can load all classes in application

-
  ```
  ClassLoader cl = ClassLoader.getSystemClassLoader();
  List<class> classes = cl.getClasses();
  ```

Design problem: testing framework

- Design a framework that invokes all methods in all classes whose names start with Test
  - Assume all such methods do not have take any arguments

```
List<class> classes = classLoader.getClasses();
for (Class clazz: classes) {
    if (clazz.getName().startsWith(“Test”) {
        Object o = clazz.constructor.invoke();
        for (Method m: clazz.getDeclaredMethods()) {
            m.invoke(o);
        }
    }
}
```

UCDAVIS

Drawback of previous approach

- All fields saved
- What if we want only a subset of all fields persisted?
- Solution: annotations

```
class RedisDB {
    //..

    public void persistAll(Object obj) {
        Map<String, String> postMap;
        Class c = obj.getClass();
        for (Field f: c.getDeclaredFields()) {
           String fieldname = f.getName();
           Object fieldVal = f.get(obj);
           fieldVal.setAccessible(true);
           postMap.put(fieldname, fieldVal);
        }
        jedis.hset(id++, postMap);
    }

}

Post p = new Post(“Hello world”, “1/23/2025”);
Car c = new Car(“BMV”);
RedisDB db = new RedisDB();
db.persistAll(p);
db.persistAll(c);
```

UCDAVIS

Java annotations

- Annotations are “metadata” added to Java code
- They have no direct impact on code execution at runtime
- Java provides some annotations, programmer can define more
- Syntax: `@Annotation_name`
- Widely used in popular frameworks

Annotation targets

- Annotations can be applied to
  - Classes
  - Fields
  - Methods
  - … many other program elements (Check https://docs.oracle.com/javase/tutorial/java/annotations/basics.html)

Annotations for compiler checks

- Additional information for the compiler
  - Detect errors
  - Suppress warnings
- Common predefined annotations
  - @Override, @Deprecated, @SuppressWarnings

```
class Vehicle {
    public void start() {
        System.out.println("Vehicle starts");
    }
}

class Bicycle extends Vehicle {
    @Override
    public void start() { ... }

    @Override // COMPILER ERROR!
    public void stop() { ... }
}
```

UCDAVIS

Few common predefined annotations

- `@Override`
  - The annotated method must override a parent method. If not, results in compiler error
- `@Deprecated`
  - The annotated method is deprecated and will show a compiler warning if used
- `@SuppressWarnings`
  - Compiler warnings for the annotated entity are suppressed

Defining new annotations

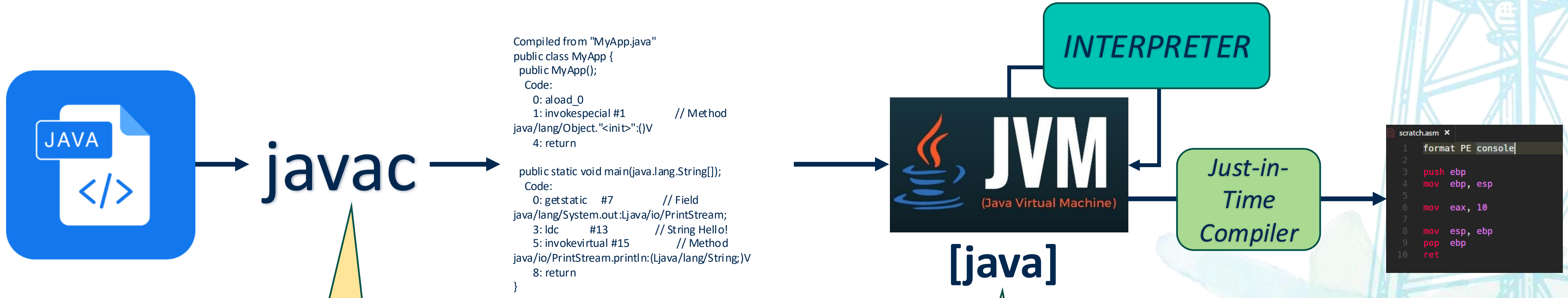
- Annotation definition must begin with @interface
- Annotations can have multiple fields
  - *Annotation type element* declarations
- Each field has a constructor
  - Constructors can have default values
- Fields can consist of arrays

```
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnnotation {
   String author();
   String date();
   int currentRevision() default 1;
   String lastModified() default "N/A";
   String lastModifiedBy() default "N/A";
}
```

```
@MyAnnotation (
    author = "John Doe",
    date = "3/17/2002",
 )
 public class Car extends Vehicle {
    // ...
 }
```

UCDAVIS

Compilation toolchain



```
Compiled from "MyApp.java"
public class MyApp {
  public MyApp();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method
java/lang/Object."<init>":()V
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: getstatic     #7                  // Field
java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #13                 // String Hello!
       5: invokevirtual #15                 // Method
java/io/PrintStream.println:(Ljava/lang/String;)V
       8: return
}
```

```
format PE console

push ebp
mov  ebp, esp

mov  eax, 10

mov  esp, ebp
pop  ebp
ret
```

MyApp.java

MyApp.class

Machine code

Some annotations retained only in the compiler

Some annotations retained at runtime

Reflection can access annotations

- Can check if annotation is present
  - `Class clazz = … ;`
    `clazz.isAnnotationPresent(MyAnnotation.class)`
  - `Field field = …;`
    `field.isAnnotationPresent(…);`
  - Same for `Method`
- Can get the annotation
  - `MyAnnotation myAnnotation = clazz.getAnnotation(MyAnnotation.class)`
- Can get the values of the annotation element fields

Drawback of previous approach

- All fields saved
- What if we want only a subset of all fields persisted?
- Solution: annotations

```
class Post { }

class RedisDB {
     Jedis jedisSession;
     static int id;
     static {
          jedisSession = new Jedis("localhost", 6379);
          id = 0;
     }

     public void persistAll(Object obj) {
          Map<String, String> postMap;
          Class c = obj.getClass();
          for (Field f: c.getDeclaredFields()) {
             String fieldname = f.getName();
             Object fieldVal = f.get(obj);
             objMap.put(fieldname, fieldVal);
          }
          jedis.hset(id++, objMap);
     }

}

Post p = new Post("Hello world", "1/23/2025");
Car c = new Car("BMV");
RedisDB db = new RedisDB();
db.persistAll(p);
db.persistAll(c);
```

Full solution

- Create an annotation @Persistable with Runtime retention policy
- Annotate only some fields
- When persisting the fields, check if the annotation is present
  - Only persist if the annotation is present

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Persistable {
}

class Post {
    @Persistable
    private String content;

    private Integer tempVal;
}

class RedisDB {
    // … set up Jedis Session
    public void persistAll(Object obj) {
        Map<String, String> objMap; Class c = obj.getClass()
        for (Field f: c.getDeclaredFields()) {
            if (f.isAnnotationPresent(Persistable.class) {
                f.setAccessible(true);
                String fieldname = f.getName();
                Object fieldVal = f.get(obj);
                objMap.put(fieldname, fieldVal);
            }
        }
        jedis.hset(id++, objMap);
    }
```

UCDAVIS

Annotations and retention policies

- The `@Retention` annotation indicates the retention policy for the annotation
- `@Retention(RetentionPolicy.SOURCE)` – only present in the source file
- `@Retention(RetentionPolicy.CLASS)` – only present in the class file
- `@Retention(RetentionPolicy.RUNTIME)` – present at runtime
- Reflection can only access annotations with `RetentionPolicy.RUNTIME`

```
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnnotation {
   String author();
   String date();
   int currentRevision() default 1;
   String lastModified() default "N/A";
   String lastModifiedBy() default "N/A";
}
```

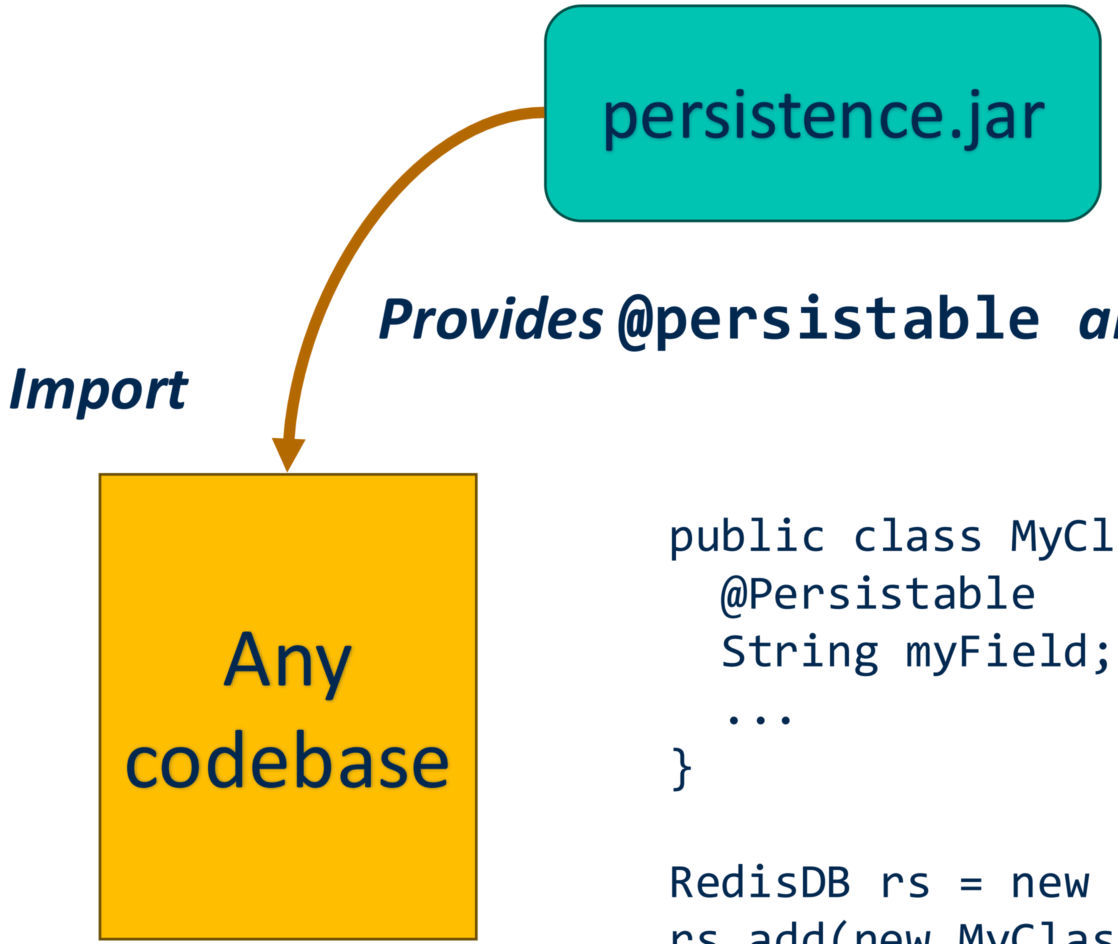
Why would we want to create our own annotations visible only at the source code / class level?

UCDAVIS

Reflection + annotations: summary

- What did we achieve?
  - The ability to persist ***any*** object, provided it is annotated
  - The persistence framework ***does not need*** to know the classes that can be persisted
  - The persistence logic is ***fully decoupled*** from the application's data model

Distribute persistence framework as a library



Compiled separately into a JAR file

Provides* `@persistable` *annotation and* `RedisDB` *class

```
public class MyClass {
  @Persistable
  String myField;
  ...
}

RedisDB rs = new RedisDB();
rs.add(new MyClass());
rs.persistAll();
```

Object relational mapping (ORM) frameworks

- Hibernate: allows the programmer to annotate a class with `@Entity`
- Provides an API `save()` that accepts an object of a class annotated with `@Entity` and saves it in the database
- The framework *abstracts* the SQL logic

```
@Entity
public class Car {
        @Id
        @GeneratedValue(strategy= GenerationType.IDENTITY)
        private Long id;

        private String name;
        private String email;
}
```

Unit testing frameworks

- JUnit: allows the programmer to annotate a class with `@Test`
- Can specify pre- and post-operations
- All methods of the class automatically executed using Reflection
- The framework *abstracts* the invocation logic of the tests

```
public class CalculatorTest {
        @Test
        public void testAdd() {
                calculator = new Calculator();
                int result = calculator.add(2, 3);
                assertEquals(5, result, "2 + 3 should equal 5");
        }
}
```

… and more

- Microservice frameworks
  - Spring Boot
- MVC frameworks
  - Spring MVC
- Logging frameworks
  - Log4j, SL4J
- … and create other design patterns such as ***Inversion of Control (IoC)***

Downsides of reflection

- Cannot *generate* new methods, fields, etc.
- Reflection causes full loss of type safety
  - If you do `s.setName(new Integer(123));` compiler will catch it at compile time
  - If you do `setNameMethod.invoke(s, new Integer(123));` it will throw an exception at runtime
- Performance is slower than directly accessing the field/method
  - Needs more method calls and memory accesses to first load the field, then the value in the field
  - In some cases, it is okay
    - Database persistence, network communications
    - Why?

Latency hierarchy

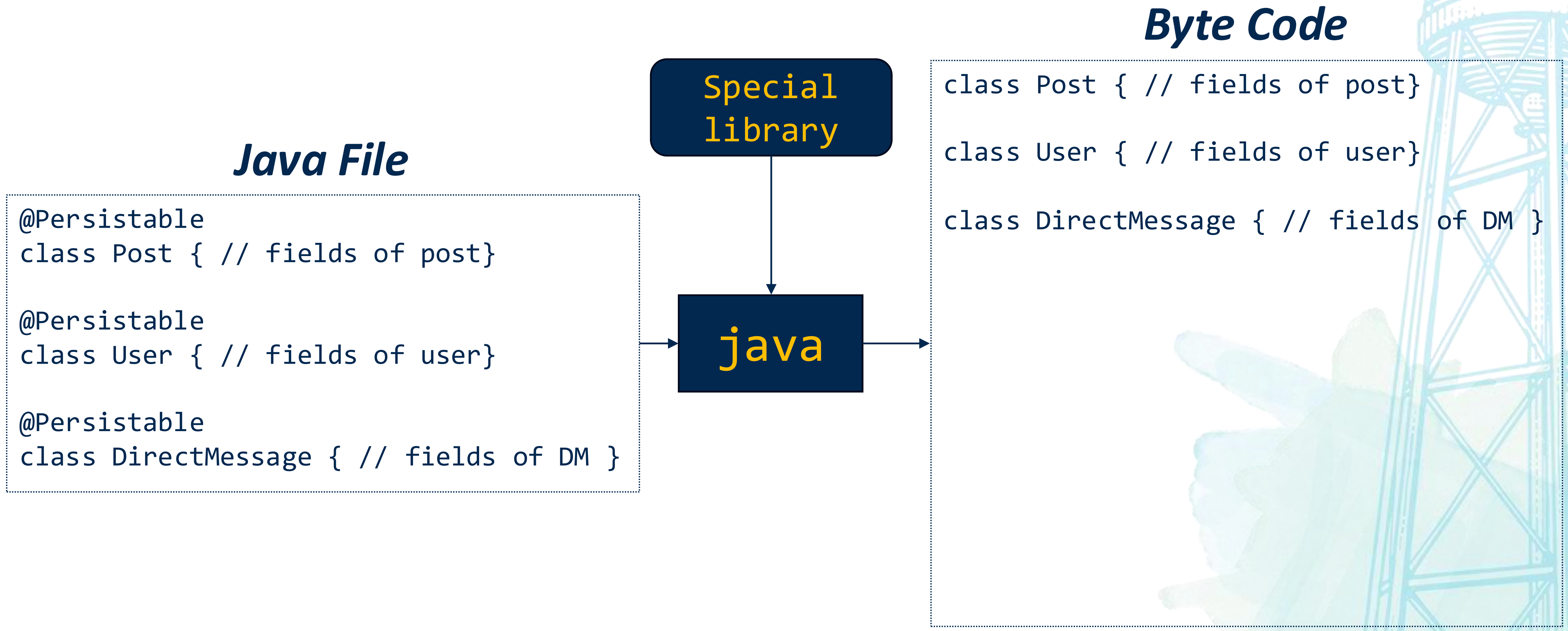
- CPU register << DRAM memory << SSDs << hard disks << network

https://static.googleusercontent.com/media/sre.google/en//static/pdf/rule-of-thumb-latency-numbers-letter.pdf

Memory accesses	***100 ns***
Secondary storage (SSDs, disks)	***20,000 ns*** (4KB random read for SSD)
Remote access (over network)	***10,000 - 500,000 ns*** (Round-trip latency within data-center)

- Adding a few extra memory accesses for reflection is not noticeable for disk and network operations

Alternate approach: bytecode instrumentation



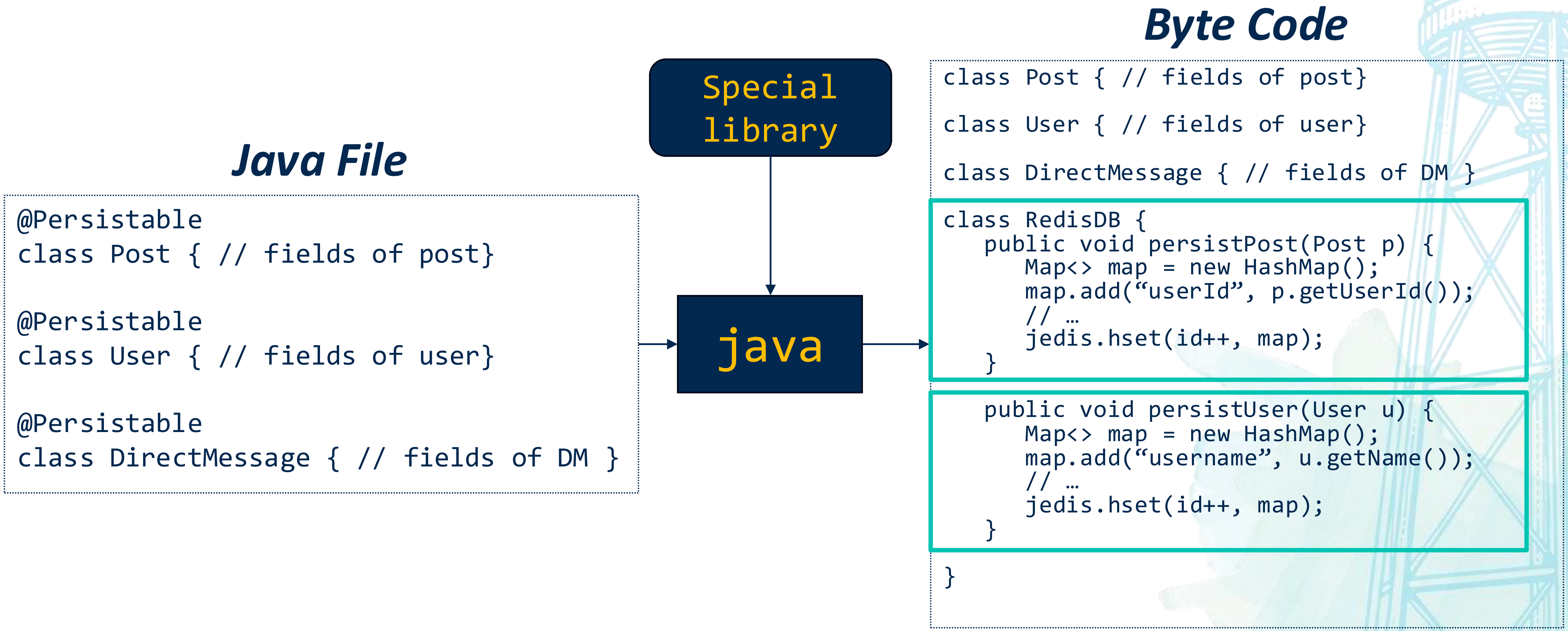
Java File

```
@Persistable
class Post { // fields of post}

@Persistable
class User { // fields of user}

@Persistable
class DirectMessage { // fields of DM }
```

Byte Code

```
class Post { // fields of post}

class User { // fields of user}

class DirectMessage { // fields of DM }
```

Alternate approach: bytecode instrumentation



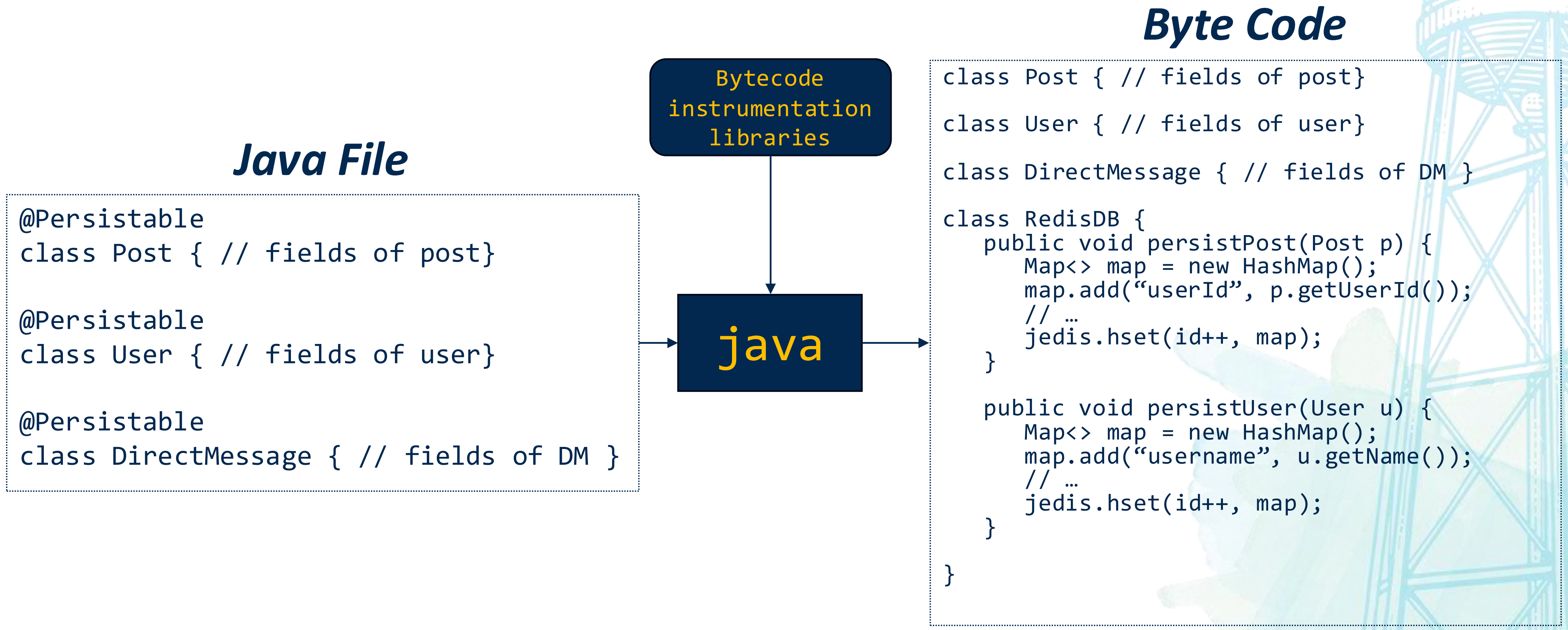
Java File

```
@Persistable
class Post { // fields of post}

@Persistable
class User { // fields of user}

@Persistable
class DirectMessage { // fields of DM }
```

Byte Code

```
class Post { // fields of post}

class User { // fields of user}

class DirectMessage { // fields of DM }

class RedisDB {
   public void persistPost(Post p) {
      Map<> map = new HashMap();
      map.add(“userId”, p.getUserId());
      // …
      jedis.hset(id++, map);
   }

   public void persistUser(User u) {
      Map<> map = new HashMap();
      map.add(“username”, u.getName());
      // …
      jedis.hset(id++, map);
   }

}
```

Bytecode instrumentation



Java File

```
@Persistable
class Post { // fields of post}

@Persistable
class User { // fields of user}

@Persistable
class DirectMessage { // fields of DM }
```

Byte Code

```
class Post { // fields of post}

class User { // fields of user}

class DirectMessage { // fields of DM }

class RedisDB {
   public void persistPost(Post p) {
      Map<> map = new HashMap();
      map.add(“userId”, p.getUserId());
      // …
      jedis.hset(id++, map);
   }

   public void persistUser(User u) {
      Map<> map = new HashMap();
      map.add(“username”, u.getName());
      // …
      jedis.hset(id++, map);
   }

}
```

Runtime bytecode generation and manipulation

Javassist

Java bytecode engineering toolkit since 1999

Used by Hibernate

Reflection in JavaScript **[Not in Syllabus]**

- Javascript `Reflect` API
- `Reflect.get()`
- `Reflect.set()`

```
const person = {
  name: 'John Doe'
};

const name = Reflect.get(person, 'name');
console.log(name); // 'John Doe’


Reflect.set(person, 'name', 'Jane Doe');
console.log(person.name); // 'Jane Doe'
```

Reflection in C++ **[Not in Syllabus]**

- RunTime Type Information (RTTI) maintains type information for each object

  -
    ```
    Person person;
    type_info personType = typeid(person);
    std::println("{}", personType.name());
    ```
- But no language-level support for dynamic invocation
- Possible to programmatically enable support for dynamic invocation
- Describe with pseudo-code how you would design reflection for C++. Assume that the programmer must manually enable reflection for a class

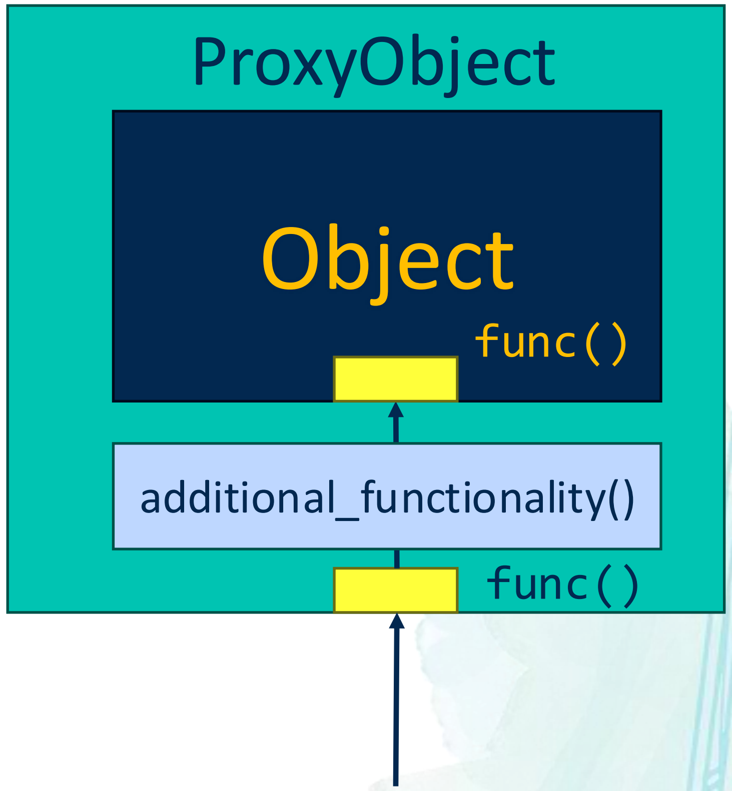
Proxies

Tapti Palit

UCDAVIS

Proxy design pattern - what is a proxy object?

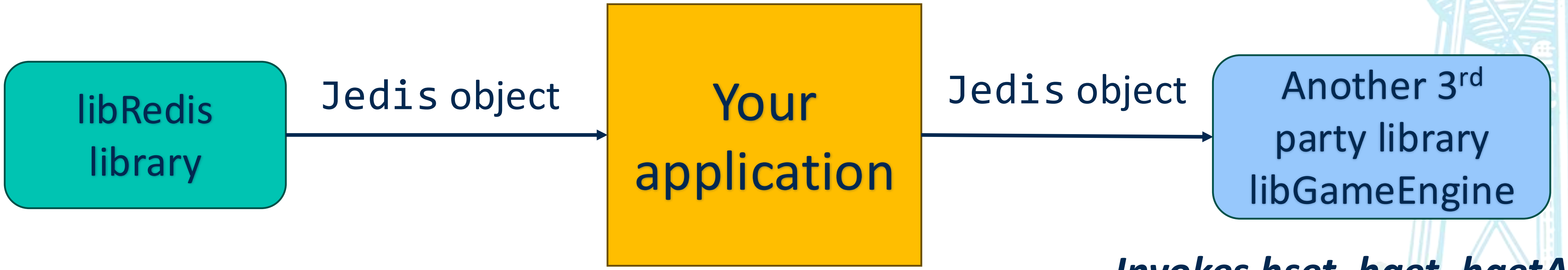
- A proxy is a wrapper around the original/target object
- The user accesses the proxy object instead of the original target object
- The proxy object typically
  - Performs some additional logic
  - Then forwards the request to the target object
    - ***Method interception***



UCDAVIS

The need for method “interception”

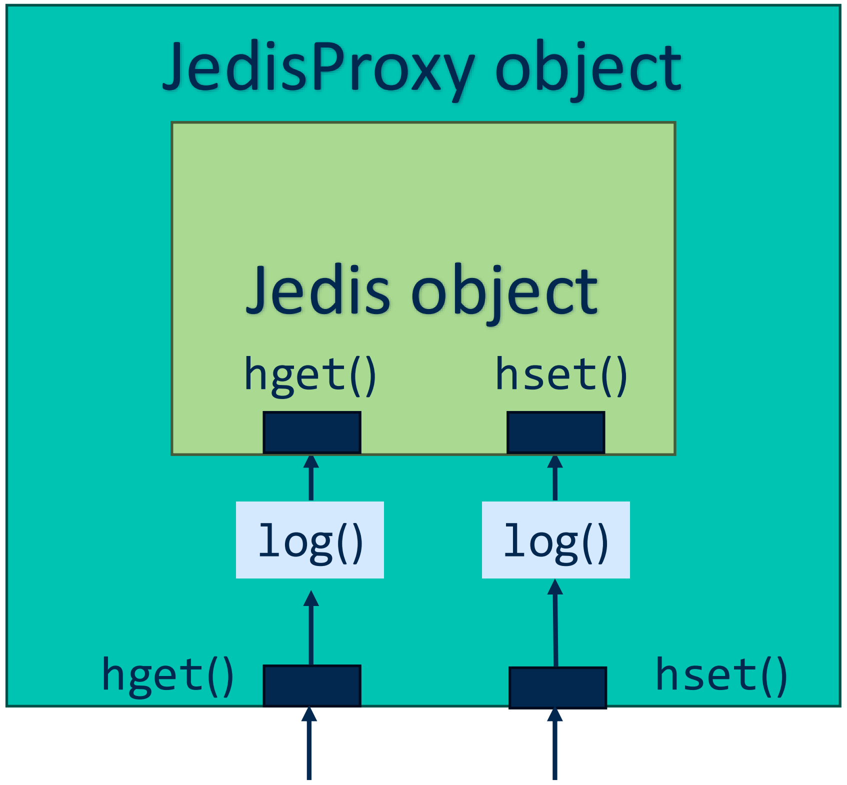
- Imagine –



Invokes hset, hget, hgetAll methods

- You want –
  - Every time `hset`, `hget`, and `hgetAll` method is invoked, it should log the access on the terminal
- Challenge: can’t change source code of Jedis library or the other 3rd party library

Proxy object wraps and extends target class

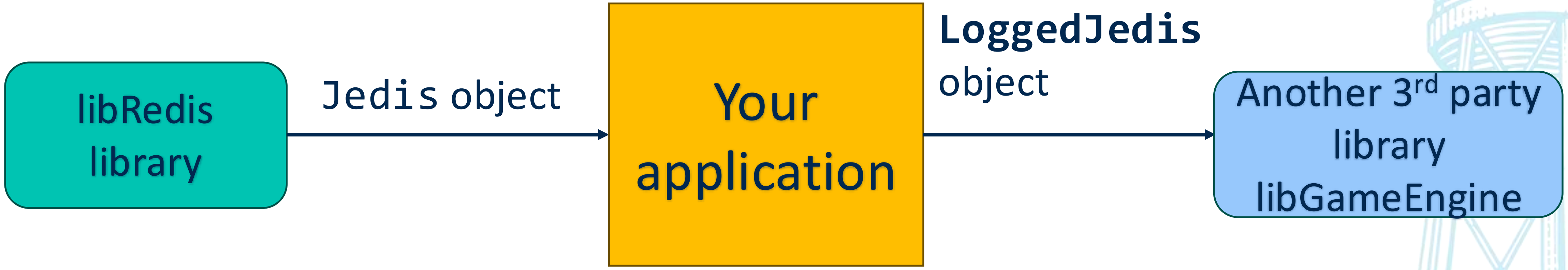


```
class Jedis {
    public String hget(String id) { … }
    public void hset(String id, String val) { … }
}

class JedisProxy extends Jedis{
    private Jedis jedis; // Wrap jedis obj

    public log() { S.o.p(…); }

    public String hget(String id) {
        log();
        return jedis.hget(id);
    }
    public void hset(String id, String val) {
        log();
        jedis.hset(id, val);
    }
}
```

Proxies intercept the method invocations

UCDAVIS

What did we gain?



Wrap Jedis object in a LoggedJedis proxy

```
LoggedJedis loggedJedis = new LoggedJedis();
loggedJedis.setJedis(jedis);
```

- Every `hget` and `hset` method invocation calls logging functionality
- No need to change `libRedis` or `libGameEngine` source code
- **No need to manually copy any fields**

What are the limitations?

- Explicitly create a class that wraps the original object type
- Explicitly create objects of this proxy class
- ***Limitation:*** proxy classes must be statically designed for each "proxyable" class
  - Duplicated logging logic

```
class SQL {
    public void executeQuery(String query) { … }
}
```

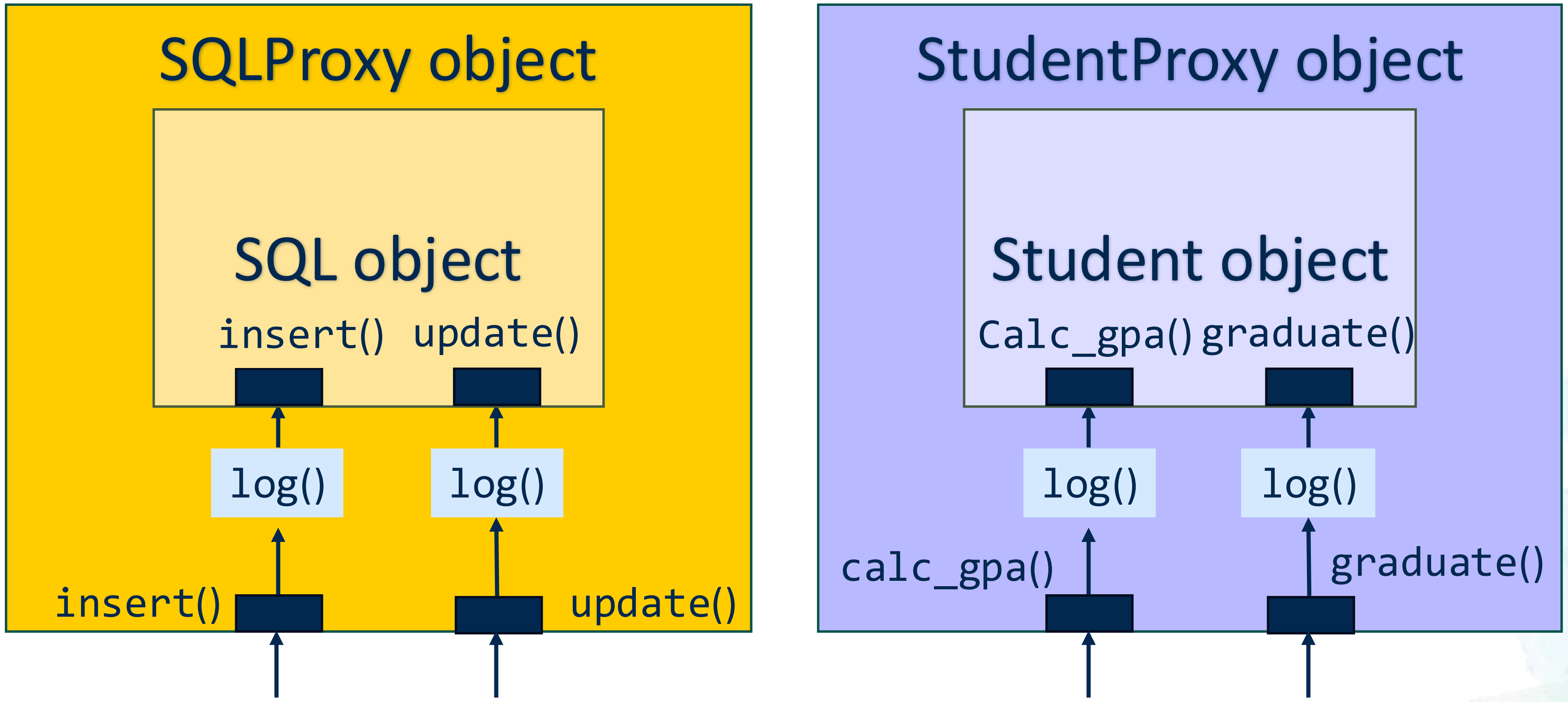
```
class SQLProxy extends Query{
    public log() { S.o.p(…); }

    public void executeQuery(String query) {
        log();
        query.executeQuery(query);
    }

}
```

```
SQL sql = …; // sql object
SQLProxy sqlProxy = new SQLProxy(sql);
```

UCDAVIS

Duplicated work

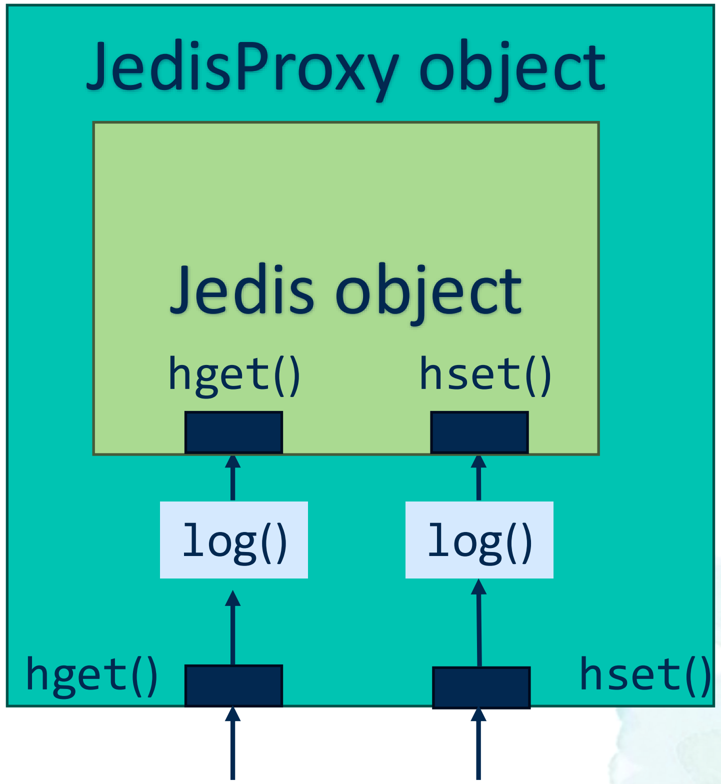
- Add logging to other classes

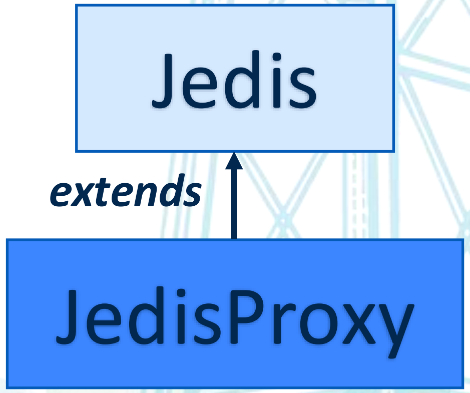


Duplicated work designing proxy classes for each class

Proxy goals

- Want to reuse the proxy functionality (logging, for example)
- Would be nice to ***dynamically*** create a ***subclass*** for the target object class that wraps any target object with the proxy functionality
- E.g. magical method which accepted the `log()` method and the `Jedis` class, and generated `JedisProxy` ***on the fly***





UCDAVIS

Proxy goals

- Want to reuse the proxy functionality (logging, for example)
- Would be nice to ***dynamically*** create a ***subclass*** for the target object class that wraps any target object with the proxy functionality
- E.g. magical method which accepted the `log()` method and generated `JedisProxy` ***on the fly***

```
Object createProxy(Object target, [FUNCTION
encapsulating the additional functionality]) {
    // 1. Create proxyClass which is a subclass
of target.getClass()
    // 2. This proxyClass will intercept all
method invocations on itself
    // 3. And invoke the additional
functionality and then retarget the method
invocation to the target object
    // 4. Create and return an object of
proxyClass
}
```

Runtime bytecode generation and manipulation

- ByteBuddy, Javassist libraries allow proxy creation via dynamic subclassing
  - ByteBuddy and Javassist hide bytecode manipulation complexities
  - Internally uses ASM library which gives complete bytecode generation/manipulation capabilities
- Java also has a dynamic proxy functionality
  - IMHO, it's unnecessarily complicated
  - Is an example of a language providing support for a design pattern!

Proxying using Javassist

- Create proxyClass which is a subclass of `target.getClass()`
- This `proxyclass` will intercept all method invocations on itself
- And invoke the additional functionality and then retarget the method invocation to the target object
- Create and return an object of `proxyClass`

```
Object createProxy(Object object) {


}
```

UCDAVIS

Proxying using Javassist

- **Create proxyClass which is a subclass of `target.getClass()`**
- This `proxyclass` will intercept all method invocations on itself
- And invoke the additional functionality and then retarget the method invocation to the target object
- Create and return an object of `proxyClass`

```
Object createProxy(Object object) {

 Class<?> clazz = object.getClass();

 ProxyFactory proxyFactory = new ProxyFactory();
 proxyFactory.setSuperclass(clazz);
 Class<?> proxyClass = proxyFactory.createClass();

 // We will see how to specify the additional
 // functionality in next slide
}
```

UCDAVIS

Proxying using Javassist

- Create proxyClass which is a subclass of `target.getClass()`
- This `proxyclass` will intercept all method invocations on itself
- And invoke the additional functionality and then retarget the method invocation to the target object
- **Create and return an object of `proxyClass`**

```
Object createProxy(Object object) {

 Class<?> clazz = object.getClass();

 ProxyFactory proxyFactory = new ProxyFactory();
 proxyFactory.setSuperclass(clazz);
 Class<?> proxyClass = proxyFactory.createClass();

 MethodHandler methodHandler = new LogHandler();
    // Additional functionality (more next slide)

 Object proxyObject =
proxyClass.getDeclaredConstructor().newInstance();

 ((javassist.util.proxy.Proxy)
proxyObject).setHandler(methodHandler);

 return proxyObject;
}
```

Proxying using Javassist

- Create proxyClass which is a subclass of `target.getClass()`
- **This proxyclass will intercept all method invocations on itself**
- **And invoke the additional functionality and then retarget the method invocation to the target object**
- Create and return an object of `proxyClass`

```
Object createProxy(Object object) {
 // -- snip
 MethodHandler methodHandler = new LogHandler();
    // Additional functionality (more next slide)
 // -- snip
}

class LogHandler extends MethodHandler {
    @Override
    public Object invoke(Object self,
        Method thisMethod,
        Method proceed,
        Object[] args) throws Throwable {
        log("accessing method" +
thisMethod.getName());
        return proceed.invoke(self, args);
    }
}
```

Javassist ensures that all method invocations of proxy object intercepted by `invoke` ***method***

UCDAVIS

Proxying using Javassist

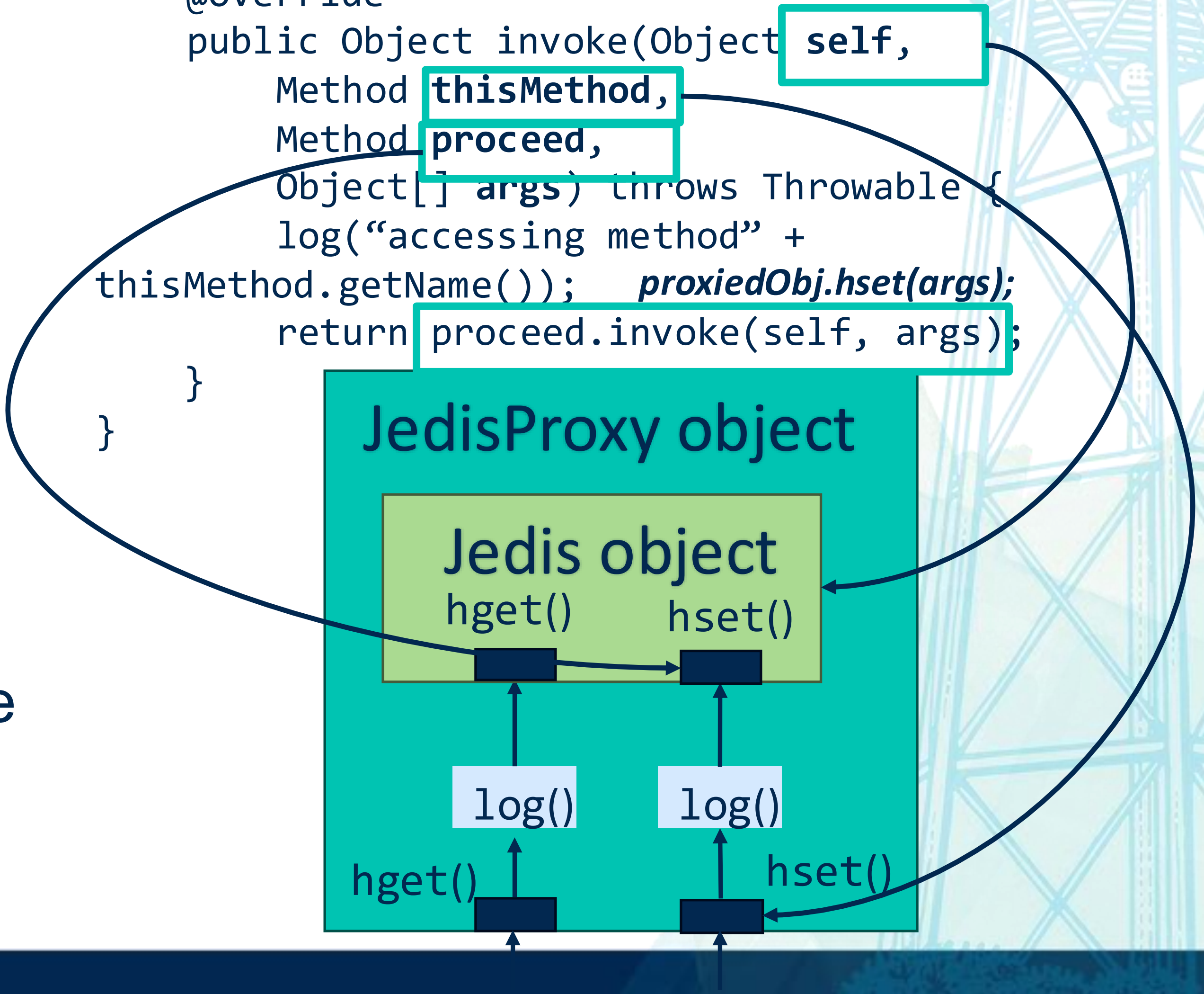
- Arguments to invoke method
  - `self` – the target object
  - `proceed` – the invoked method in the target object
  - `thisMethod` – the invoked method in the proxy object (generally not useful)
  - `args` – any arguments passed to the method invocation

```
class LogHandler extends MethodHandler {
    @Override
    public Object invoke(Object self,
        Method thisMethod,
        Method proceed,
        Object[] args) throws Throwable {
        log(“accessing method” +
thisMethod.getName());
        return proceed.invoke(self, args);
    }
}
```

Proxying using Javassist

- Arguments to invoke method
  - `self` – the target/proxied object
  - `proceed` – the invoked method in the target object
  - `thisMethod` – the invoked method in the proxy object (generally not useful)
  - `args` – any arguments passed to the method invocation

```
class LogHandler extends MethodHandler {
    @Override
    public Object invoke(Object self,
        Method thisMethod,
        Method proceed,
        Object[] args) throws Throwable {
        log("accessing method" +
thisMethod.getName());
        return proceed.invoke(self, args);
    }
}
```



Using dynamic proxies

```
Object createProxy(Object object) {
 Class<?> clazz = object.getClass();

 ProxyFactory proxyFactory = new ProxyFactory();
 proxyFactory.setSuperclass(clazz);

 MethodHandler methodHandler = new LogHandler();
 Class<?> proxyClass =
proxyFactory.createClass();
 Object proxyObject =
proxyClass.getDeclaredConstructor().newInstance()
;
 ((javassist.util.proxy.Proxy)
proxyObject).setHandler(methodHandler);
 return proxyObject;
}

class LogHandler {
  // -- snip
}
```

```
Jedis jedis = …;

Jedis jedisProxy = (Jedis) createProxy(jedis);
jedisProxy.hset(…);
```

```
Student student = …;

Student studentProxy =
             (Student)createProxy(student);
studentProxy.getName();
```

Demo

- https://github.com/davsec-teaching/javassist_demo

Real world use cases of proxies

- Sample use cases
  - Lazy loading
  - Mock testing

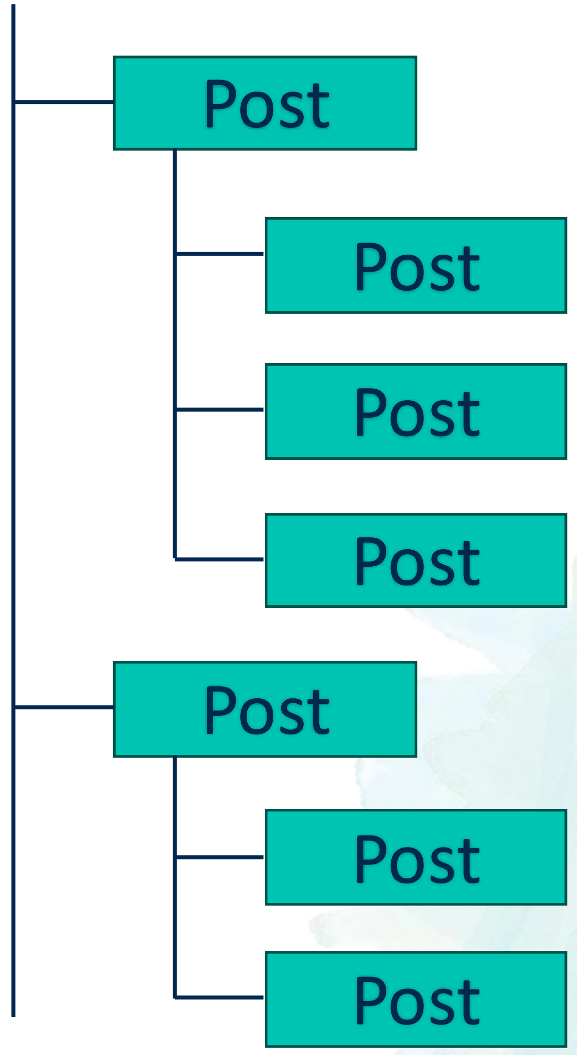
HIBERNATE

mockito

EASYMOCK

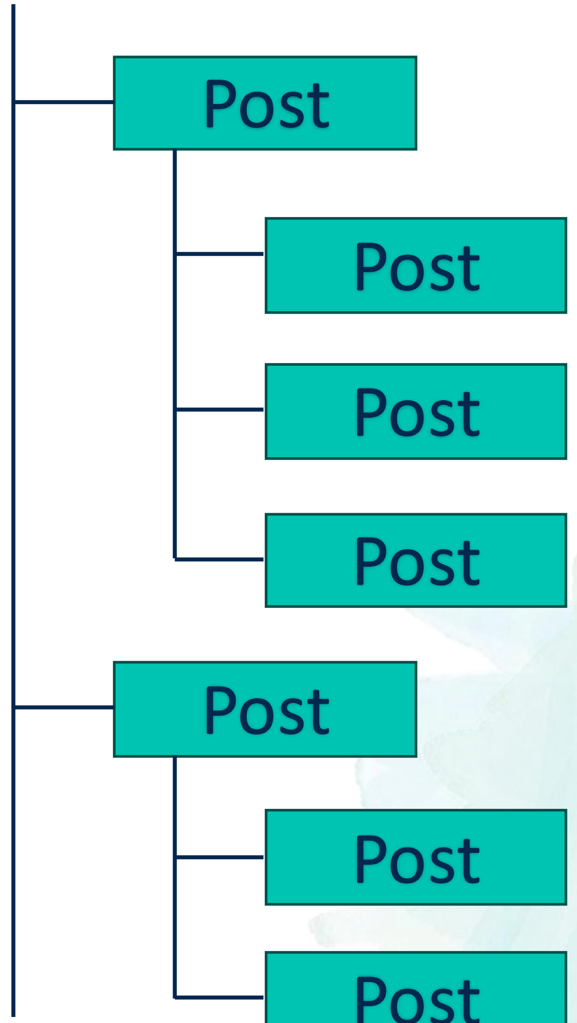
Loading posts

- Load a single post from the Redis database



Loading posts

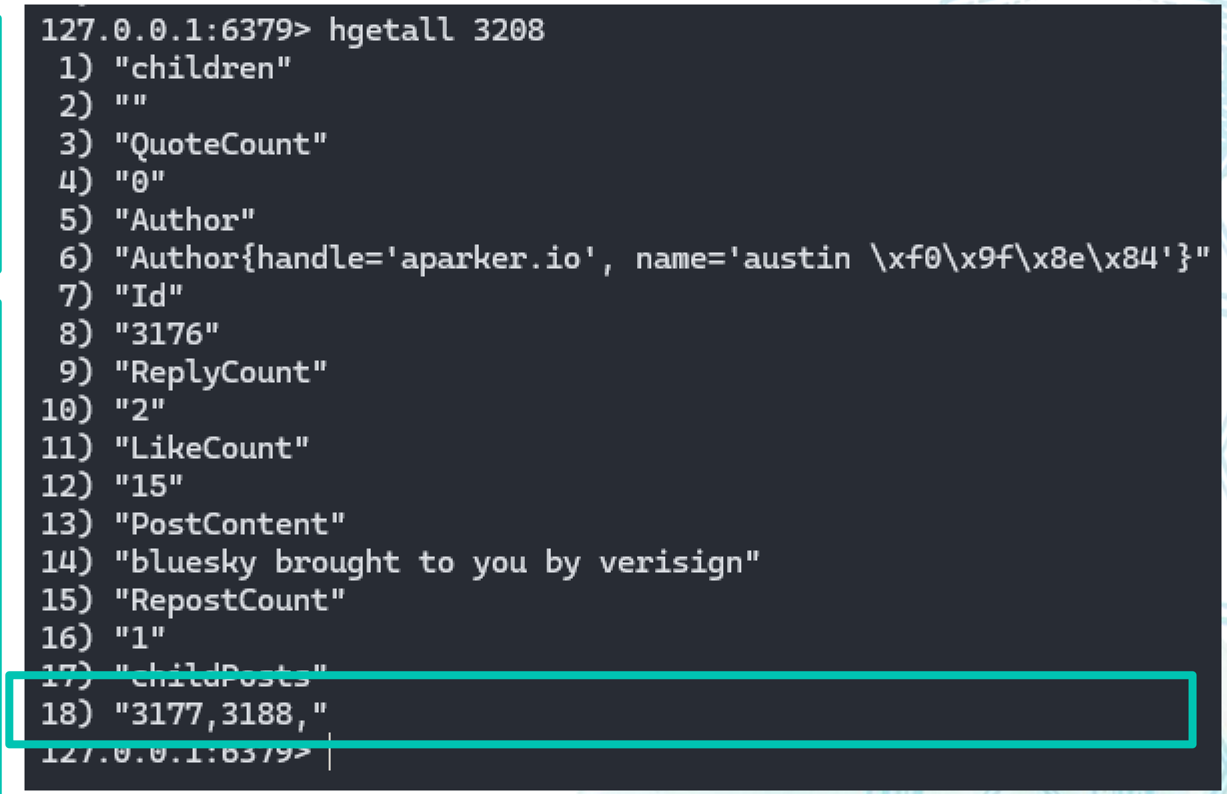
- Load a single post from the Redis database
- High level overview
  - Load a post
  - For each reply
    - Load the reply



Loading posts from Redis

```
map = jedis.hgetAll(“3208”);
Post post = new Post();
post.setId(“3208”);
post.setCreatedAt(map.get(“createdAt”));
```

```
List<String> replies =
map.get(“childPosts).split(“,”);

for (String replyId: replies) {
    replyMap = jedis.hgetAll(replyId);
    Post reply = new Post();
    reply.setId(replyId);
    reply.setCreatedAt(replyMap.get(“createdAt”);
    post.getReplies().add(reply);
}
```



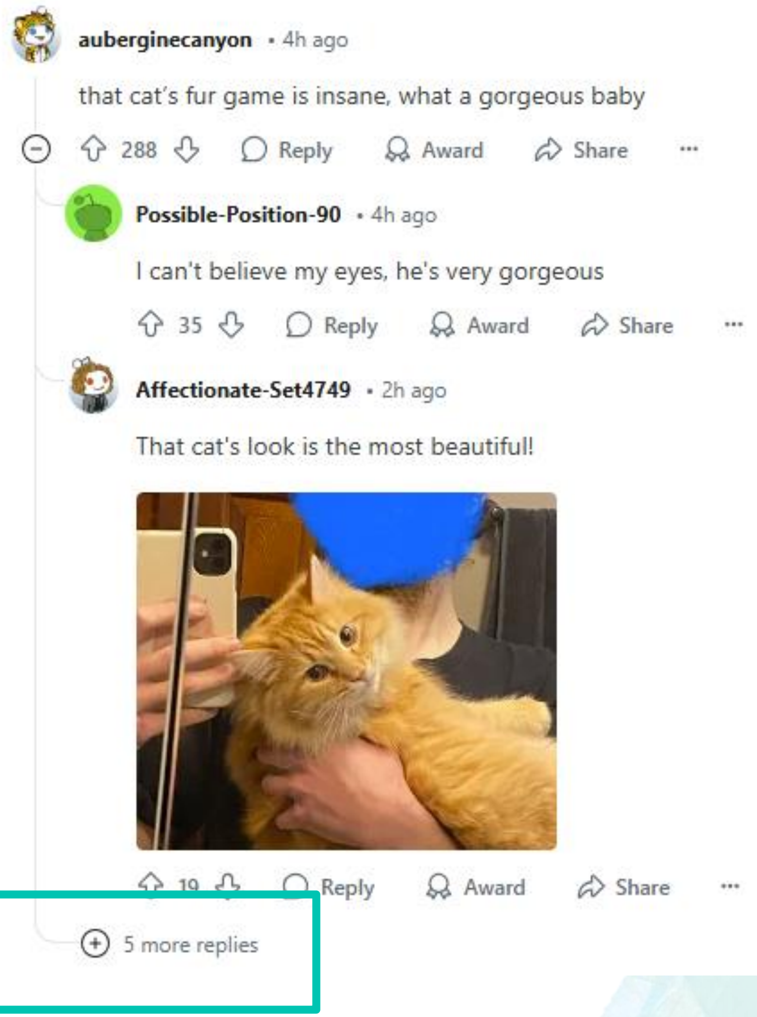
```
127.0.0.1:6379> hgetall 3208
 1) "children"
 2) ""
 3) "QuoteCount"
 4) "0"
 5) "Author"
 6) "Author{handle='aparker.io', name='austin \xf0\x9f\x8e\x84'}"
 7) "Id"
 8) "3176"
 9) "ReplyCount"
10) "2"
11) "LikeCount"
12) "15"
13) "PostContent"
14) "bluesky brought to you by verisign"
15) "RepostCount"
16) "1"
17) "childPosts"
18) "3177,3188,"
127.0.0.1:6379>
```

All replies loaded when the post is loaded

UCDAVIS

Lazy loading design pattern

- Common performance improvement
- An object’s children (replies) are lazily loaded/fetched on-demand
- Improves UI responsiveness



auberginecanyon • 4h ago

that cat’s fur game is insane, what a gorgeous baby

Possible-Position-90 • 4h ago

I can't believe my eyes, he's very gorgeous

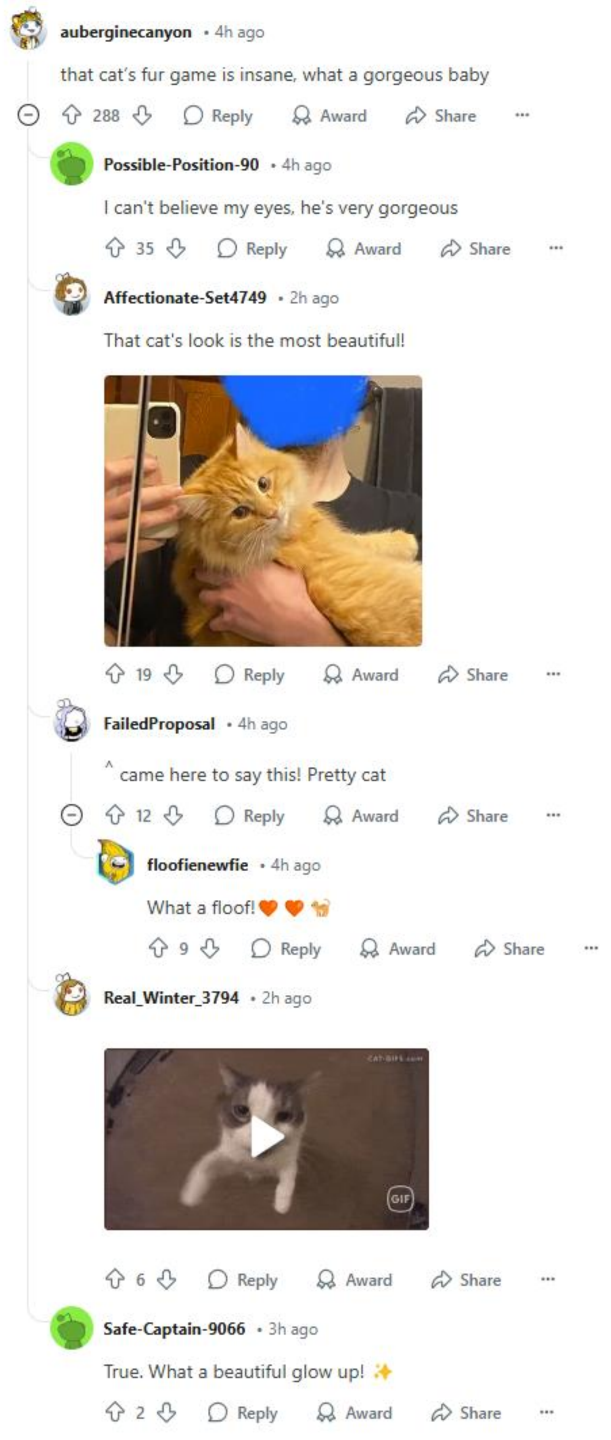
Affectionate-Set4749 • 2h ago

That cat's look is the most beautiful!

UCDAVIS

Lazy loading

- Common performance improvement
- An object's children (replies) are lazily loaded/fetched on-demand
- Clicking on '+' loads the remaining replies



auberginecanyon • 4h ago

that cat's fur game is insane, what a gorgeous baby

288 Reply Award Share ···

Possible-Position-90 • 4h ago

I can't believe my eyes, he's very gorgeous

35 Reply Award Share ···

Affectionate-Set4749 • 2h ago

That cat's look is the most beautiful!

19 Reply Award Share ···

FailedProposal • 4h ago

^ came here to say this! Pretty cat

12 Reply Award Share ···

floofienewfie • 4h ago

What a floof! 🧡 🧡 🐈

9 Reply Award Share ···

Real_Winter_3794 • 2h ago

6 Reply Award Share ···

Safe-Captain-9066 • 3h ago

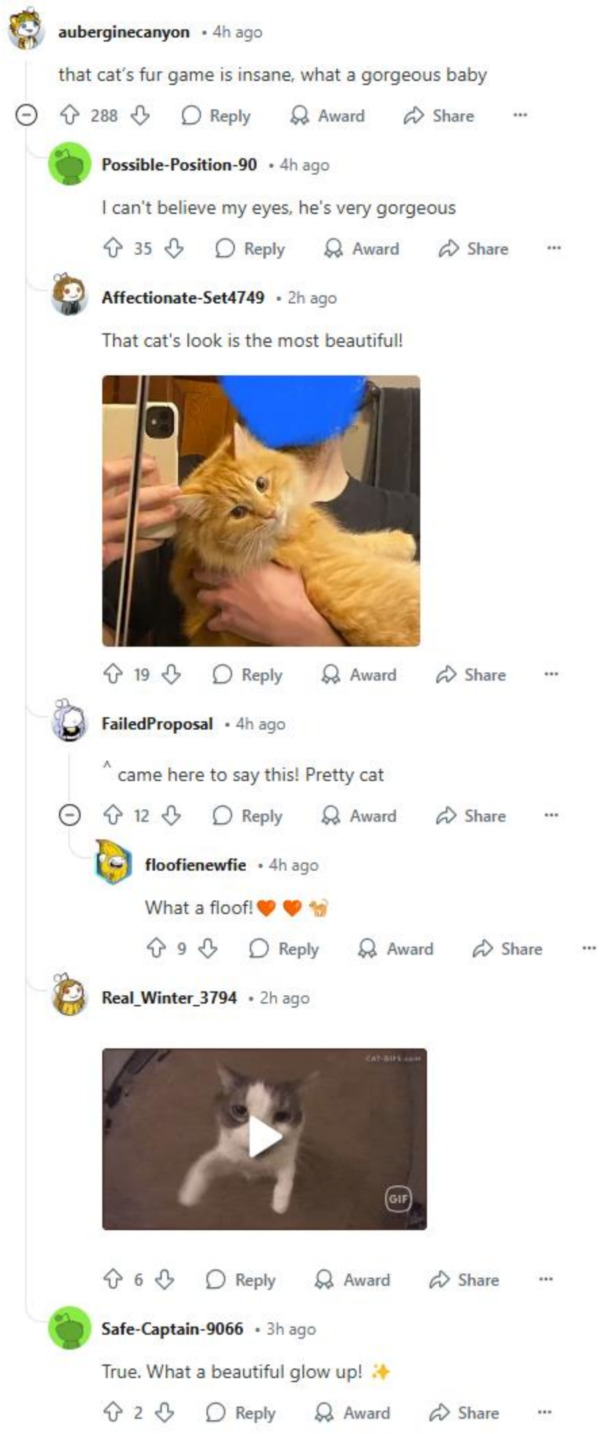
True. What a beautiful glow up! ✨

2 Reply Award Share ···

UCDAVIS

Lazy loading

- Clicking on '+' loads the remaining replies
- Dynamic proxies simplify implementing lazy loading



auberginecanyon • 4h ago

that cat's fur game is insane, what a gorgeous baby

288 Reply Award Share

Possible-Position-90 • 4h ago

I can't believe my eyes, he's very gorgeous

35 Reply Award Share

Affectionate-Set4749 • 2h ago

That cat's look is the most beautiful!

19 Reply Award Share

FailedProposal • 4h ago

^ came here to say this! Pretty cat

12 Reply Award Share

floofienewfie • 4h ago

What a floof! 🧡 🧡 🐈

9 Reply Award Share

Real_Winter_3794 • 2h ago

6 Reply Award Share

Safe-Captain-9066 • 3h ago

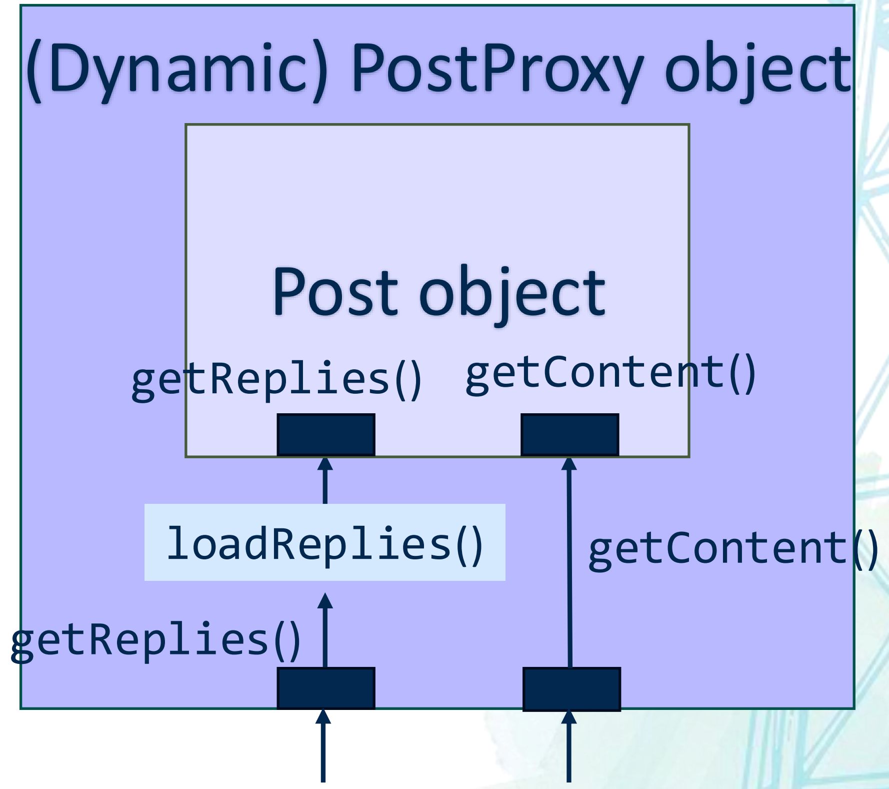
True. What a beautiful glow up! ✨

2 Reply Award Share

UCDAVIS

Proxies for lazy loading

- High level approach
  - Create reply Post objects with only the `postId` (do not load the data using `hgetall()`)
  - Create a (dynamic) proxy for the Post object and have it intercept `getReplies()` method invocation
  - Only when the `getReplies()` method is invoked, perform `loadReplies()` to load all the reply post objects

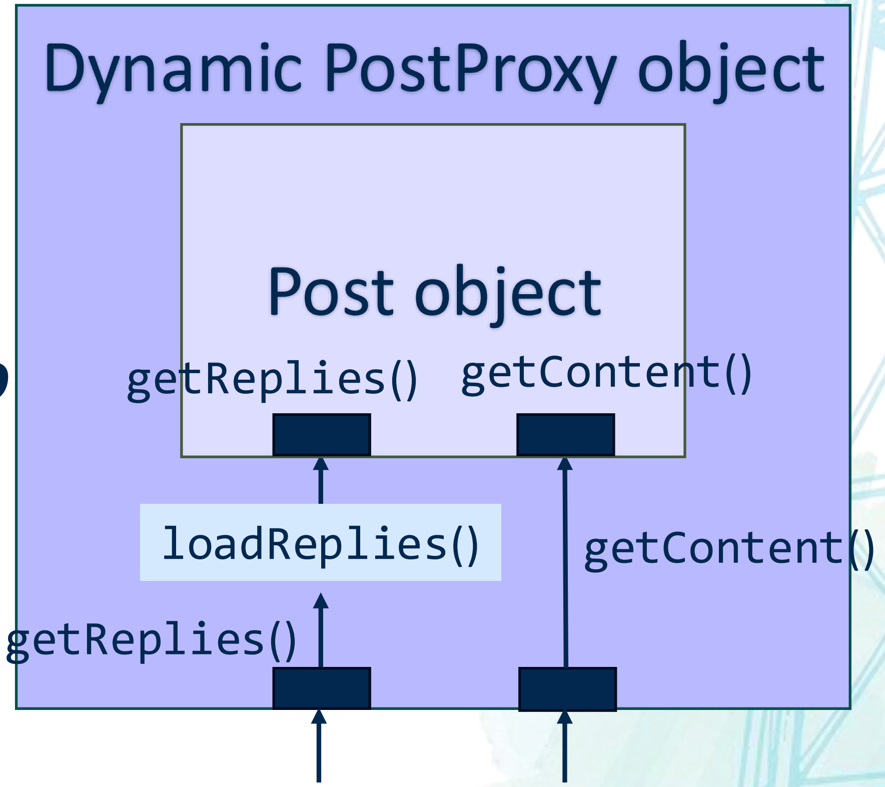


UCDAVIS

Proxies for lazy loading

```
Post loadPost(String id) {
 map = jedis.hgetAll(id);
 Post post = new Post();
 post.setId(id);
 post.setCreatedAt(map.get(“createdAt”));

 List<String> replies =
    map.get(“childPosts).split(“,”);

 for (String replyId: replies) {
    Post reply = new Post();
    // does not load the reply details
    reply.setId(replyId);
    post.getReplies().add(reply);
 }
 return createLazyLoadProxy(post);
}
```

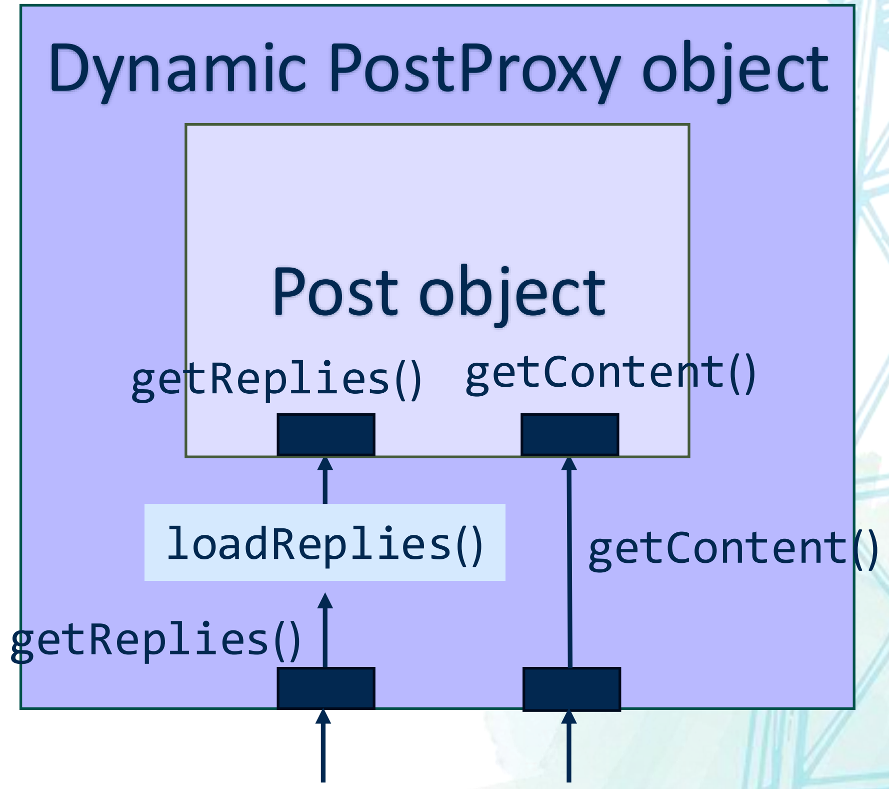
Only set the Reply ID



Proxies for lazy loading

```
List<Post> loadReplies() {
    List<Post> newReplies = …;
    for (Reply reply: this.getReplies()) {
     // only the id is populated
     Reply reply = loadPost(reply.getId());
     newReplies.add(reply);
    }
    this.setReplies(newReplies);
}
```

Proxy calls loadReplies transparently



Lazy loading code is provided by the proxy class

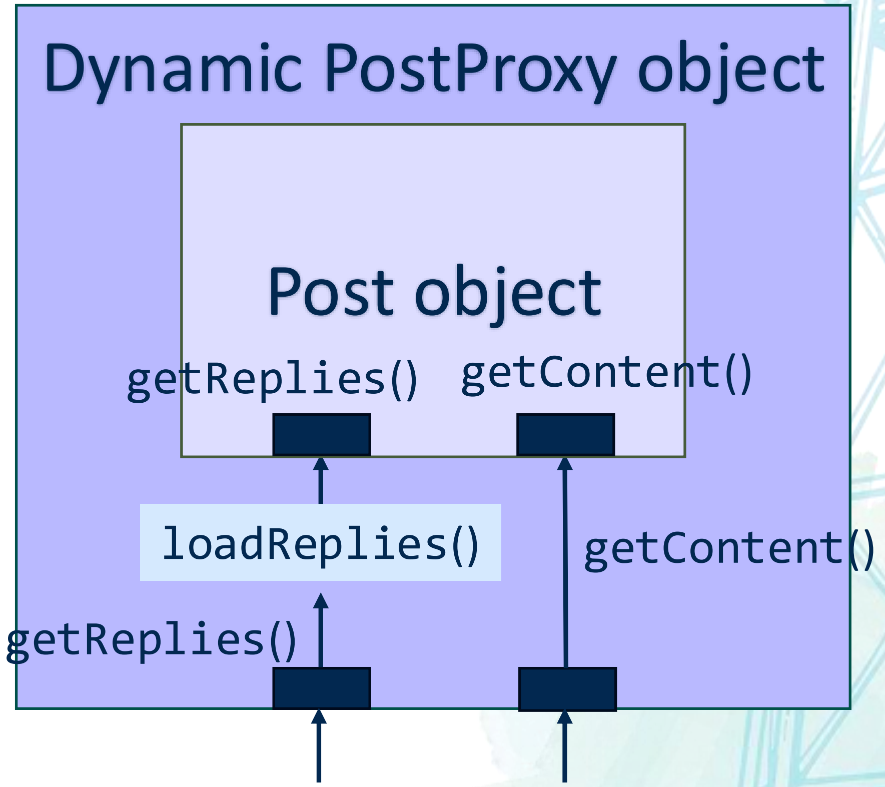
Proxies for lazy loading

```
Object createLazyLoadProxy(Object target) {
    Class<?> clazz = object.getClass();
    ProxyFactory proxyFactory = new ProxyFactory();
    proxyFactory.setSuperclass(clazz);

    MethodHandler methodHandler = new
LazyLoadMethodHandler(target);

    Class<?> proxyClass =
proxyFactory.createClass();
    Object proxyObject =
proxyClass.getDeclaredConstructor().newInstance();
    ((javassist.util.proxy.Proxy)
proxyObject).setHandler(methodHandler);
    return proxyObject;
}
```

Create PostProxy objects

```
class MyMethodHandler extends MethodHandler {
    private Object target;
    public Object invoke(…) throws Throwable {
        // Logic to invoke loadReplies();
        // if getReplies() is invoked
        return proceed.invoke(target, args);
    }
}
```

Invoke loadReplies()



createLazyLoadProxy returns the proxy object

Reflection + annotations + dynamic proxies

- Can combine proxies with reflection and annotation
- Lazy loading only for annotated fields
  - Load the @LazyLoad annotated fields only when their getter is invoked
- ***Transparent*** lazy load from the programmer's perspective

```
class Post {
    @Persistable
    Integer postId;
    @Persistable
    String postContent;
    @Persistable
    @LazyLoad
    List<Post> replies;

    // getters and setters for postId,
postContent, replies
    // All methods intercepted by dynamic proxy
    // replies lazy loaded by the dynamic proxy
}
```

UCDAVIS

Lazy loading

- Database records
  - All ORMs such as Hibernate support lazy loading using annotations
- File content
  - Lazy load 1 GB file
- Content to be fetched over the network
  - Lazy load remote content

UCDAVIS

Dependency Injection

UCDAVIS

Dependency inversion principle

- Large component (typically services) should not depend directly on smaller components (backend processors)
- Instead, should depend an ***abstract*** component

```
// Code that violates dependency
inversion principle
class OrderService {
   private PaypalProcessor p = new
PaypalProcessor();
    // -- snip
}
```

UCDAVIS

Dependency injection

- Dependency injection is a design pattern where an object's dependencies are **provided externally** rather than created internally
- Constructor or setter methods
- Promotes loose coupling, modularity, better testability
- `OrderService` depends on an interface and concrete implementation is chosen by the caller

```
// Without DI
class OrderService {
   private PaypalProcessor p = new
PaypalProcessor();
    // -- snip
}
```

👎

```
// With DI
class OrderService {
    private PaymentProcessor p;

    OrderService(PaymentProcessor p) {
        this.p = p;
    }
   // -- snip
}
```

👍

UCDAVIS

Dependency injection

- Dependency injection is a design pattern where an object's dependencies are **provided externally** rather than created internally
- Constructor or setter methods
- Promotes loose coupling, modularity, better testability
- `OrderService` depends on an interface and concrete implementation is chosen by the caller

```
interface PaymentProcessor { // --snip }
class PaypalProcessor implements
          PaymentProcessor { // --snip}

class OrderService {
    private PaymentProcessor p;

    OrderService(PaymentProcessor p) {
        this.p = p;
    }
    // -- snip
}

public static void main(..) {
    PaymentProcessor p = new PaypalProcessor();
    OrderService service = new OrderService(p);
}
```

Inversion-of-control and DI containers

- DI container is the framework responsible for instantiating and injecting the dependencies
- Inversion of Control means a class gives away control of instantiating its dependencies to the IoC framework
- Goal: design IoC framework that can automatically inject the `A`, `B`, `C` dependencies into `OrderService`

```
class OrderService {
    private A a;
    private B b;
    private C c; // … and more

    OrderService(A a, B b, C c, …) {
        this.a = a; // -- snip
    }
    // -- snip
}

public static void main(..) {
    A a = new A();
    B b = new B();
    // …
    OrderService service = new OrderService(a,
b, c, …);
}
```

Spring IoC framework

- Provides automatic dependency injection
- Provides `@Component`, `@Service`, `@Repository` annotations to register ***beans*** which are ***auto-injected***
- Question: how does Spring achieve this?

```
@Service
class OrderService {
     @Autowired
    private A a;
     @Autowired
     private B b; // … and more

    OrderService() {
        // -- no need to assign anything
    }
}

@Component
class A {}

@Component
class B {} // -- snip

public static void main(..) {
    ApplicationContext ctx = // -- snip;

     OrderService service =
        ctx.getBean(OrderService.class);
}
```

UCDAVIS

Proxies for mock testing

Mocking overview

- What is mocking?
  - Simulate the behavior of real objects in a controlled way
- Useful during unit-testing
- Why?
  - Isolate components for unit testing
  - External systems (APIs, Databases)
  - Complex or time-consuming operations

UCDAVIS

Mocking overview

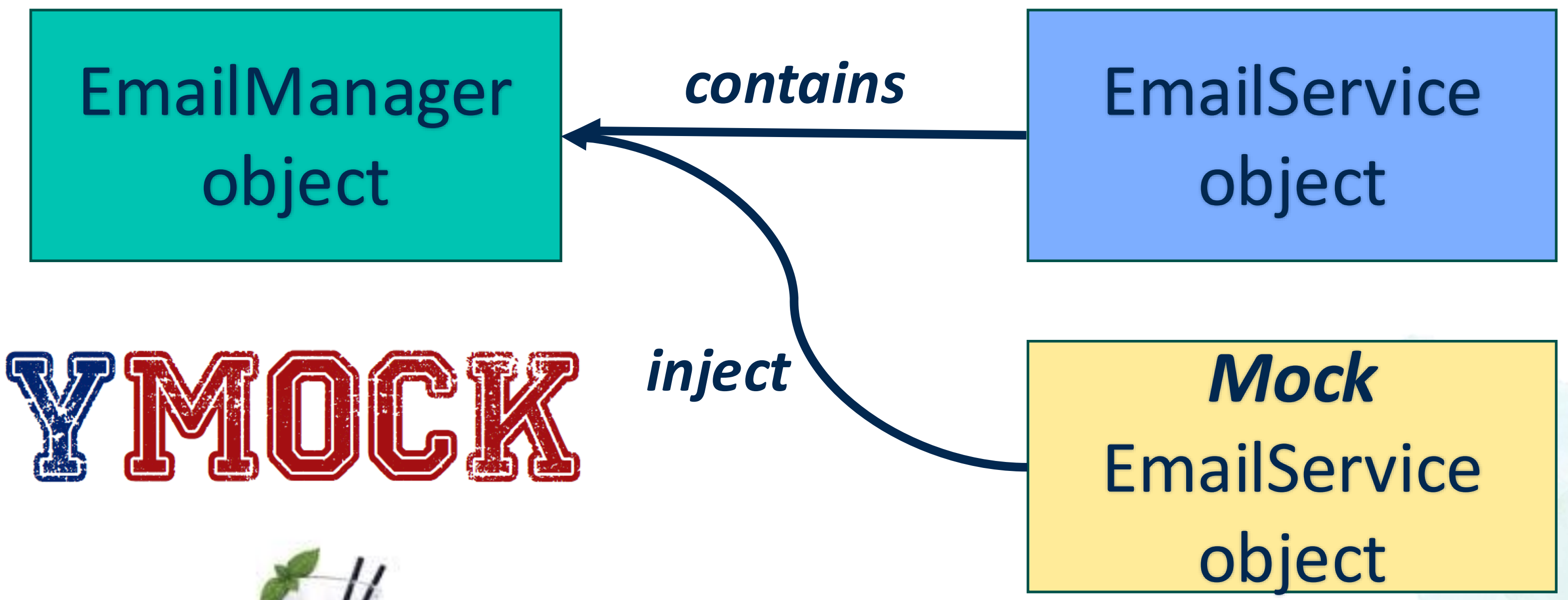
- Example: unit test `composeEmail`
  - But don't want to actually send an email to a client!!

```
public EmailService {
    public boolean sendEmail(...) {
        // send an email
        if (success) return true;
        return false;
    }
}
```

```
public class EmailManager {
    private EmailService emailService;

    private void formatEmail(String email) { … }
    private void displayError(boolean succ) { …}

    public String composeEmail(...) {
        String email = ...;
        formatEmail(email);
        boolean success = emailService.send(email);
        displayError(success);
    }
}
```

Mockito overview

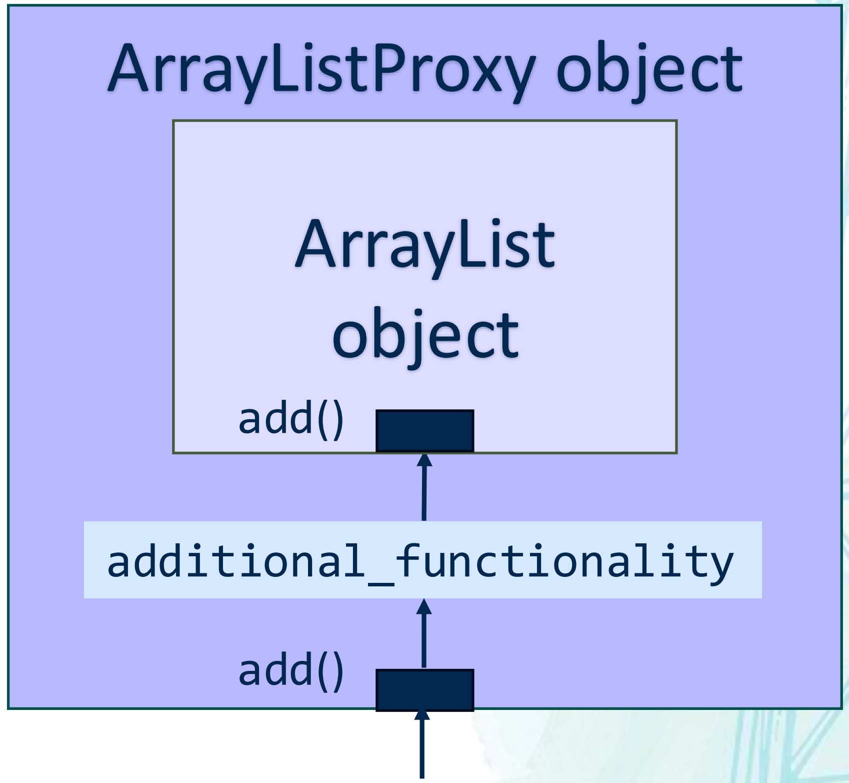
Allows programmer to inject mock objects to ***stub*** functionality



Stubbed **`sendEmail()`** ***function that does nothing***

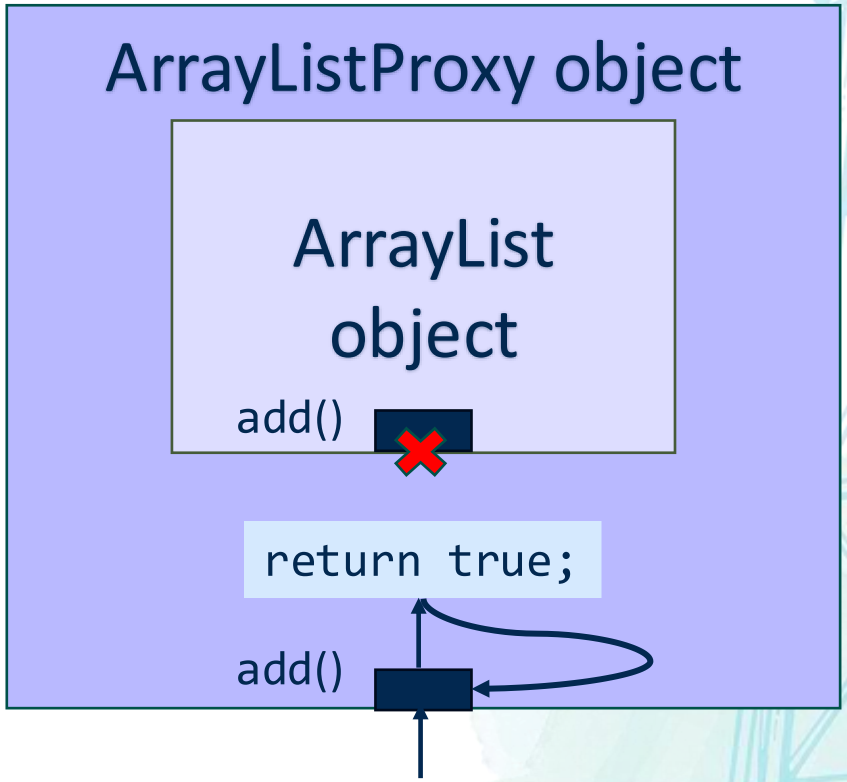
Proxy can hijack functionality

- Previous cases proxies augmented the target object functionality



Proxy can hijack functionality

- Previous cases proxies augmented the target object functionality
- It can also hijack functionality and not invoke the target object's method
- ArrayList proxy object can always return true for `add()` invocation



UCDAVIS

Proxy can hijack functionality

- Previous cases proxies augmented the target object functionality
- It can also hijack functionality and not invoke the target object's method
- ArrayList proxy object can always return true for add( ) invocation

```
Object createProxy(Object object) {
 // -- snip

 MethodHandler methodHandler = new MockHandler();

 // -- snip
 return proxyObject;
}

class MockHandler extends MethodHandler {
@Override
    public Object invoke(Object self,
        Method thisMethod,
        Method proceed,
        Object[] args) throws Throwable {
        log(“accessing method” +
thisMethod.getName());
        ~~return proceed.invoke(self, args);~~
        return new Boolean(true);
    }
}
```

UCDAVIS

Mock an ArrayList

- Mockito's `mock()` method injects a proxy
- Configure the proxy object
  - `when`
  - `thenReturn`

```
import java.util.List;

import static
org.mockito.ArgumentMatchers.anyInt;
import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.when;

public class MyApp {
    public static void main(String[] args) {
        List<Number> myList =
mock(ArrayList.class);
        when(myList.add(10)).thenReturn(true);

when(myList.add(anyInt())).thenReturn(false);

        System.out.println(myList.add(30)); //
return false
        System.out.println(myList.add(10)); //
return true

    }
}
```

Mock an ArrayList

- Mockito's `mock()` method injects a proxy
- Configure the proxy object
  - `when`
  - `thenReturn`
- Mockito proxy objects record method invocations the first time
  - Then replay the configured return value

```
import java.util.List;

import static
org.mockito.ArgumentMatchers.anyInt;
import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.when;

public class MyApp {
    public static void main(String[] args) {
        List<Number> myList =
mock(ArrayList.class);
        when(myList.add(10)).thenReturn(true);

when(myList.add(anyInt())).thenReturn(false);

        System.out.println(myList.add(30)); //
return false
        System.out.println(myList.add(10)); //
return true

    }
}
```

UCDAVIS

Writing a JUnit test with Mockito

```
public EmailService {
    public boolean sendEmail(...) {
        // send an email
        if (success) return true;
        return false;
    }
}

public class EmailManager {
    private EmailService emailService;

    private void formatEmail(String email) { … }
    private void displayError(boolean succ) { …}

    public String composeEmail(...) {
        String email = ...;
        formatEmail(email);
        boolean success = emailService.send(email);
        displayError(success);
    }
}
```

```
import ...;

public class EmailManagerTest {
    @Mock
    private EmailService emailService;

    @InjectMocks
    private EmailManager emailManager;

    @BeforeEach
    public void setUp() {
        MockitoAnnotations.openMocks(this); // Initialize
mocks
    }

    @Test
    public void testComposeEmail() {
        // Stub the sendEmail method to return true

when(emailService.sendEmail(anyString())).thenReturn(true);

        // Call the method under test
        String result = emailManager.composeEmail();
        // do any assertion checks

     }
}
```

Summary

- Reflection, annotations, and dynamic proxies are very powerful
  - Must be used judiciously
  - Typically, not used in regular application development
  - Used in framework development

UCDAVIS

Summary

- Frameworks using reflection, annotations, and dynamic proxies are widely used
  - Beneficial to know how they work under-the-hood
  - Very helpful in debugging

```
@Entity
class InsuranceClient {
    @Id
    private Integer insuranceClientId;

    @Column(name = “address”)
    private String address;

    @OneToMany(fetch = FetchType.LAZY)
    List<Policy> policies;

    // getters and setters

}

InsuranceClient client = …;
client.getPolicies(); // sometimes takes seconds
                      // Why?
```

UCDAVIS

Summary

- Reflection and proxies have application way beyond just Redis persistence and logging
- Used in microservice frameworks, MVC frameworks, dependency injection, database persistence, and so on, in many languages

spring Boot

LOG4J

spring MVC

MICRONAUT®