

Java reflection, proxies, and annotations

Tapti Palit

Motivation

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



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`

```
@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;  
}
```

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

```
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;  
    }  
}
```

```
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\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);  
}
```

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);  
}
```

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)

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 a class or interface
- `java.lang.reflect.Method`: Represents a class method
- `java.lang.reflect.Field`: Represents a class field
- `java.lang.reflect.Constructor` : Represents 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];
```

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;
```

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)

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

public class Student

private int id;

private String name;

public void setId() {..}

//... other getters and
setters

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

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

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

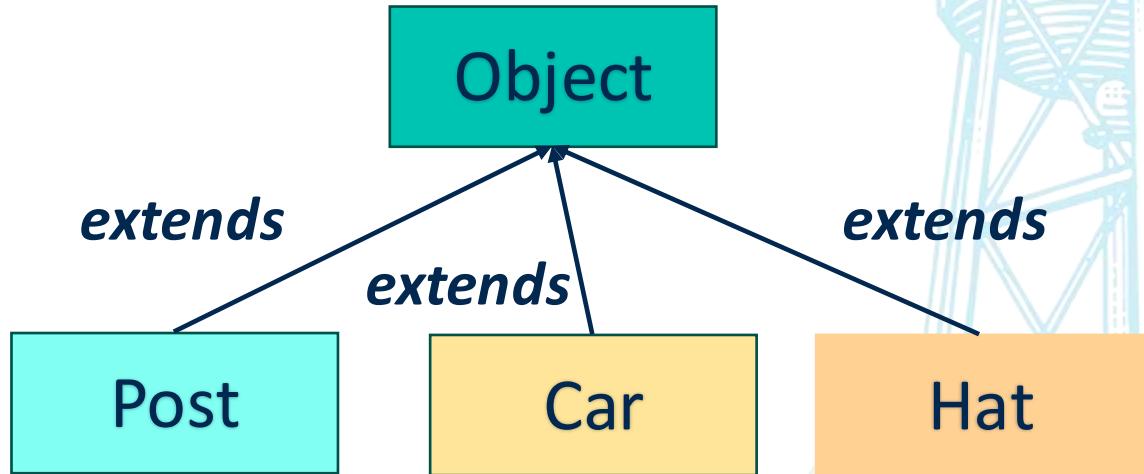
| |
|---------------------|
| Student studentObj1 |
| 101 |
| "ABC" |
| Student studentObj2 |
| 102 |
| "DEF" |

```
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



Background

- Instance method invocation
 - Every instance method call's first implicit argument is **this**
 - `Student s = new Student();
s.getName(); // implicit first argument is "this"`
 - Not applicable to static functions

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.MyAp
p");
```

- ```
Class clazz = obj.getClass();
```

`studentObj2.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];`

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"
```

```
Student.getMethods()
```

```
[...]  
Method method$1 /* object of type Method */
```

```
methodName = "setId";
```

```
argTypes = [IntegerType];
```

```
returnType = VoidType;
```

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

```
==
```

```
studentObj2.setId(10);
```

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);  
            System.out.println(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);  
            System.out.println(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
- Goal: create a persistence framework for Redis that works with *any* objects

} *More in the Microservices module*

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

Students

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);
```

# 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());
```

*Needs to be duplicated  
for each class*

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

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

 public void persistAll(Object obj) {
 Map<String, String> objMap;
 for (Field f: obj.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);
```

# 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);
 }
 }
}
```

# 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;
 for (Field f: obj.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("BMW");
RedisDB db = new RedisDB();
db.persistAll(p);
db.persistAll(c);
```

# 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() { ... }
}
```

# 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 errors for the annotated entity are suppressed

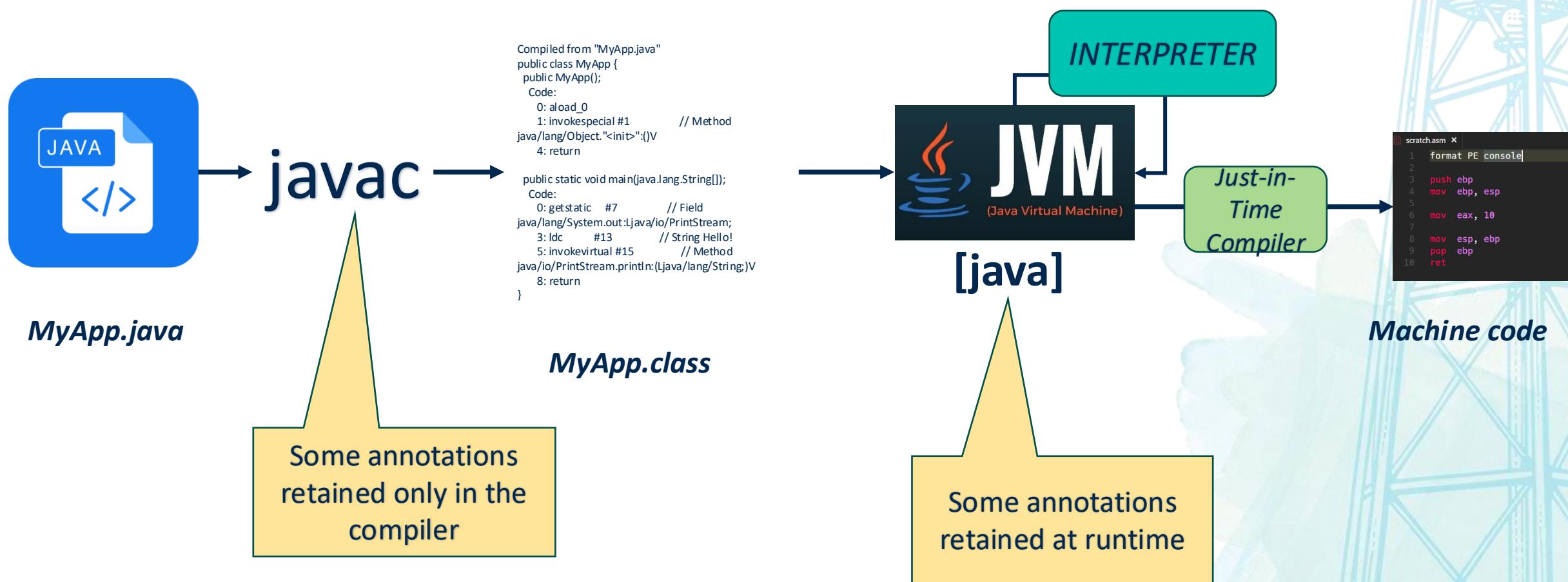
# Defining new annotations

- 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

```
@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 {
 // ...
}
```

# Recall: compilation toolchain



# 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;
 for (Field f: obj.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("BMW");
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;
 for (Field f: obj.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);
 }
}
```

# Reflection (recap)

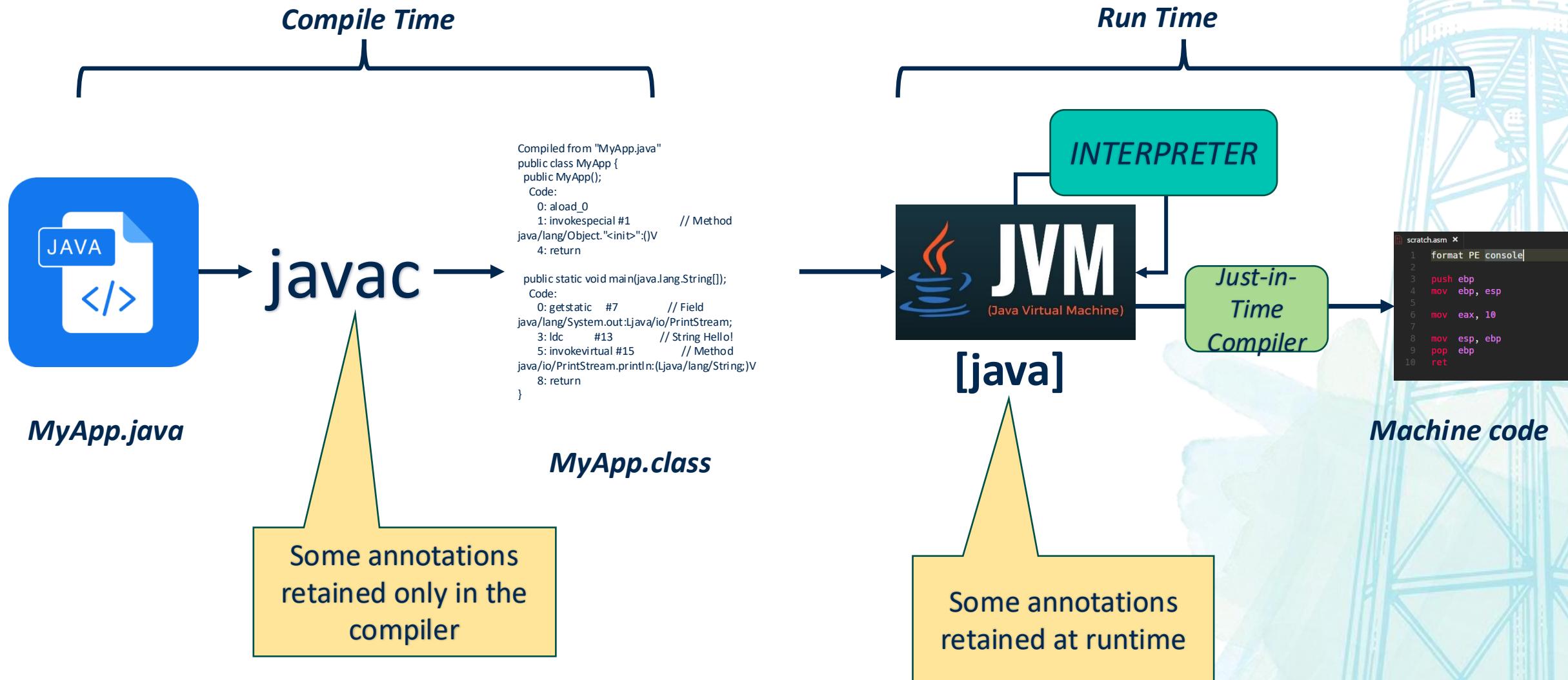
- Create an annotation  
    @Persistable with Runtime retention policy
- Annotate only some fields
- When persisting the fields, check if the annotation is present
  - Only persist if the annotation is present

```
@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;
 for (Field f: obj.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);
 }
}
```

# Recall: compilation toolchain



# 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

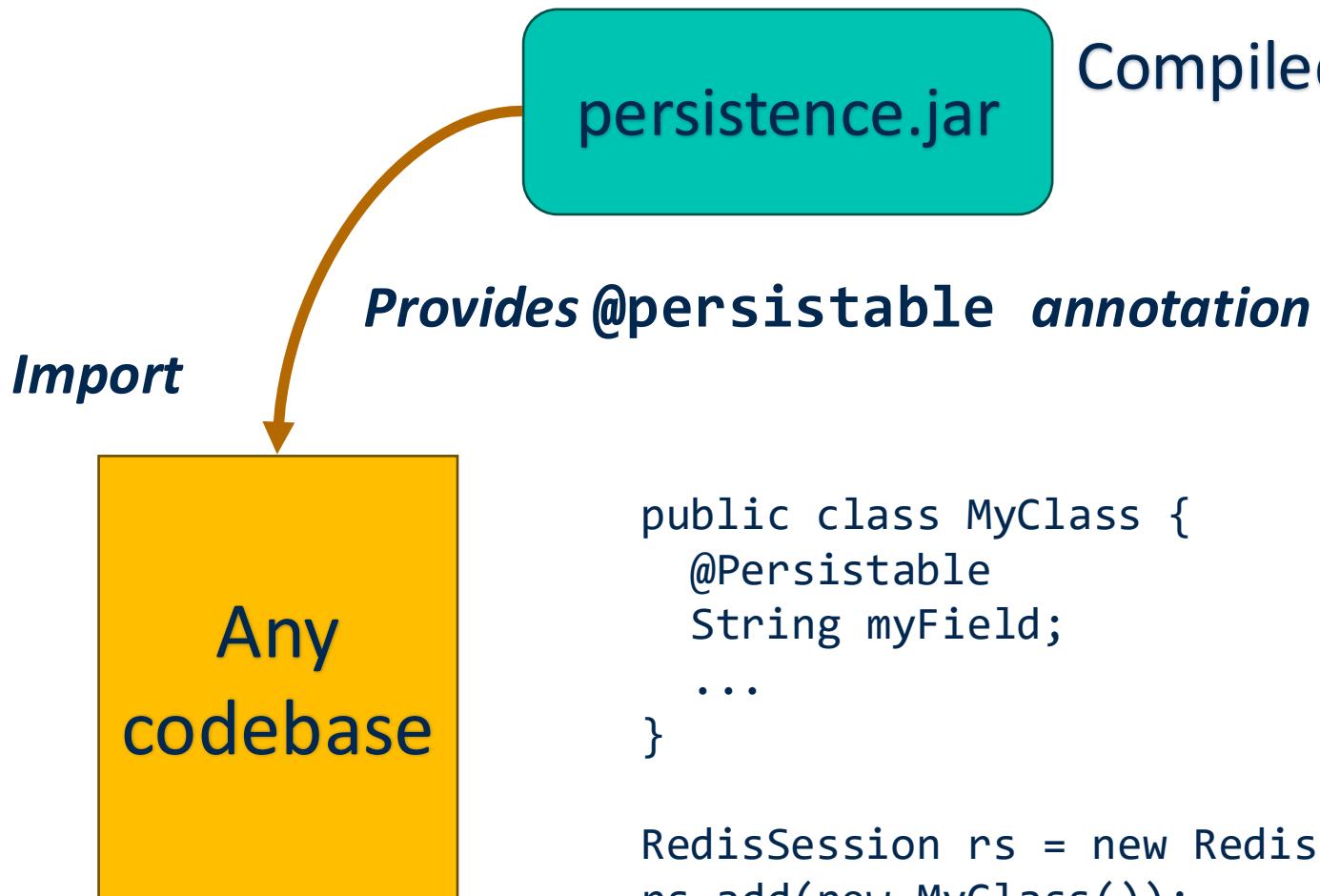
```
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnnotation {
 String author();
 String date();
 int currentRevision() default 1;
 String lastModified() default "N/A";
 String lastModifiedBy() default "N/A";
}
```

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

# 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



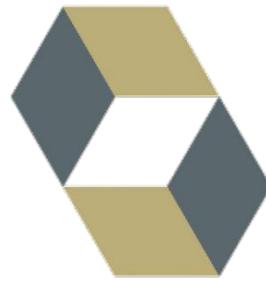
Compiled separately into a JAR file

# 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
- In-class demo

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

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



HIBERNATE

# 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)***



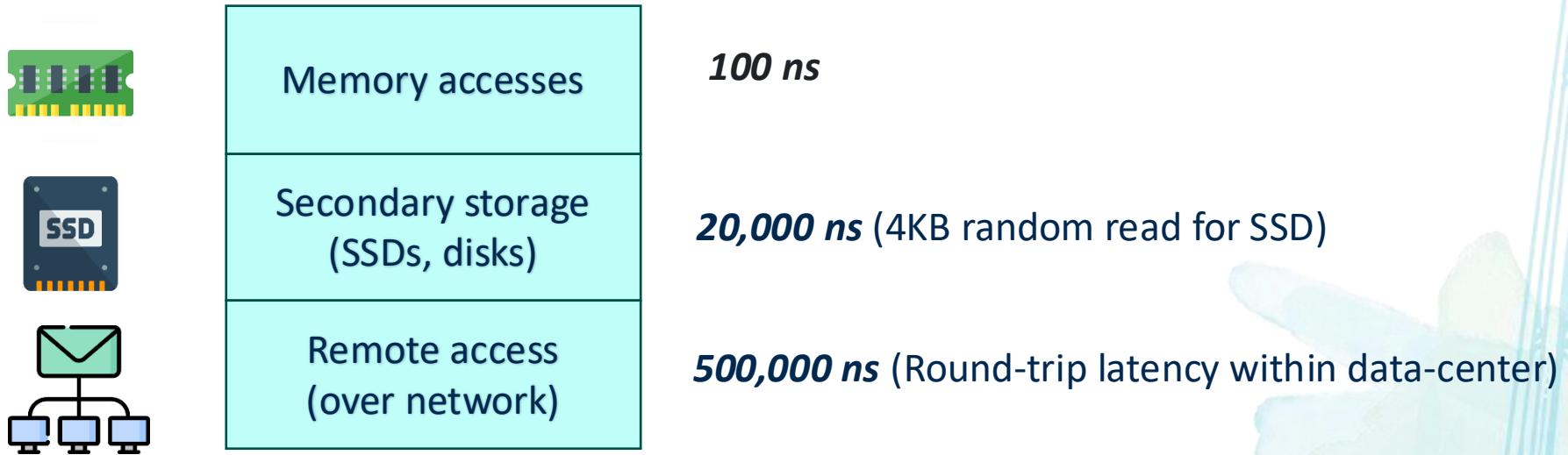
# Downsides of reflection

- 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
  - 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>



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

# Bytecode instrumentation

## Java File

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

@Persistable
class User { // fields of user}

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

Special library

java

## Byte Code

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

class User { // fields of user}

class DirectMessage { // fields of DM }

class JedisInterface {
 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

## Byte Code

```
class Post { // fields of post}
class User { // fields of user}
class DirectMessage { // fields of DM }

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

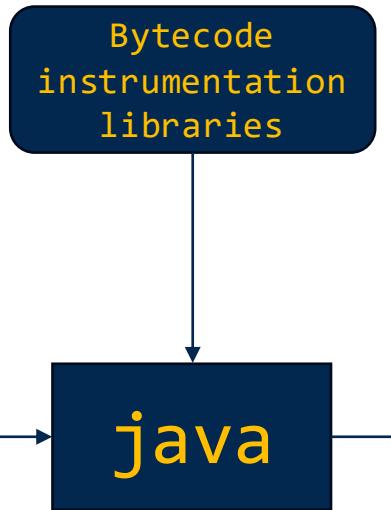
 public void persistUser(User u) {
 Map<> map = new HashMap();
 map.add("username", u.getName());
 // ...
 jedis.hset(id++, map);
 }
}
```

## Java File

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

@Persistable
class User { // fields of user}

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



# Runtime bytecode generation and manipulation

- Recap what is bytecode?



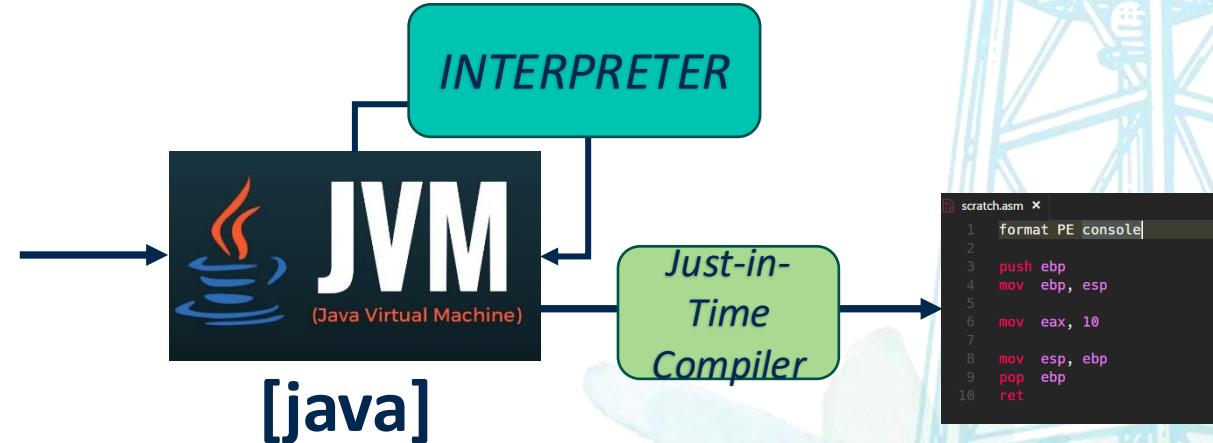
javac →

```
Compiled from "MyApp.java"
public class MyApp {
 public MyApp() {
 Code:
 0: aload_0
 1: invokespecial #1
 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
}
```

MyApp.java

MyApp.class



- Bytecode is just a format!!

# Runtime bytecode generation and manipulation



*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
- 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

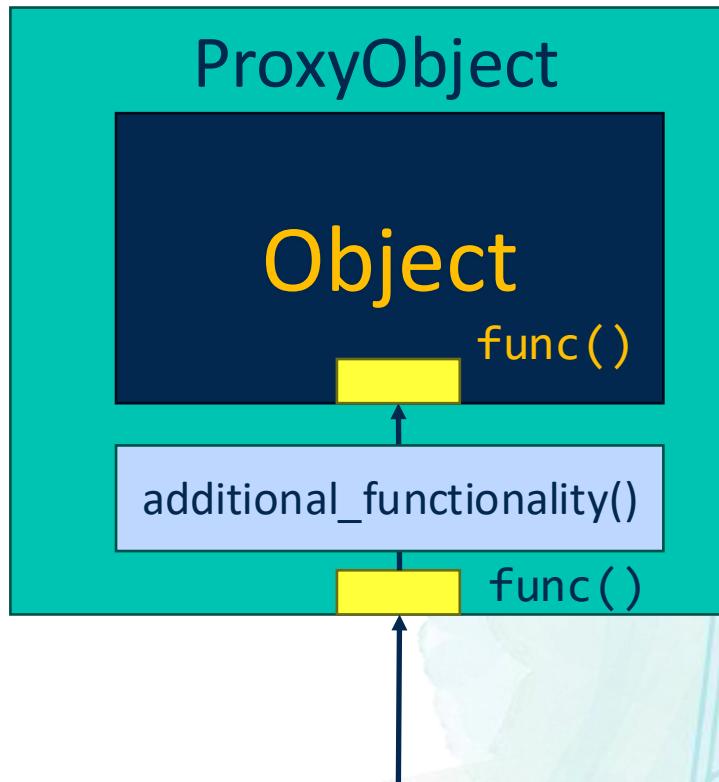
# Proxies

Tapti Palit



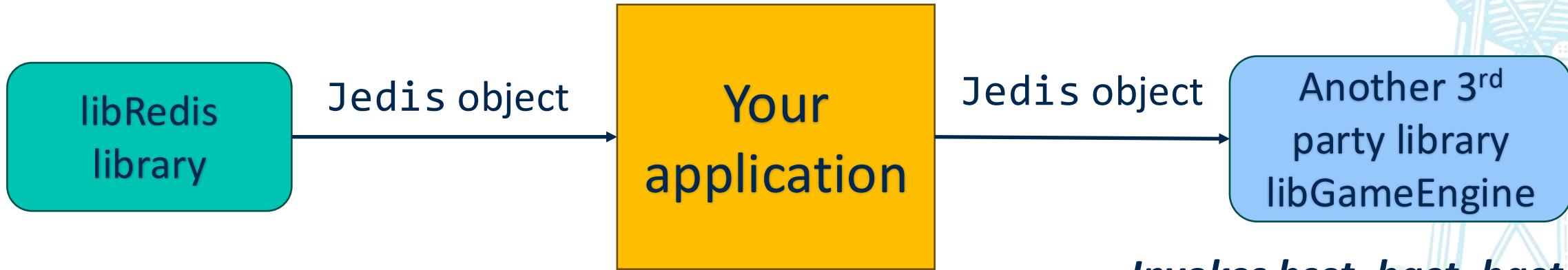
# Proxy design pattern - what is a proxy object?

- 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*



# The need for method “interception”

- Imagine –



- 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 3<sup>rd</sup> party library

*Invokes hset, hget, hgetAll methods*

# How to specialize a class?

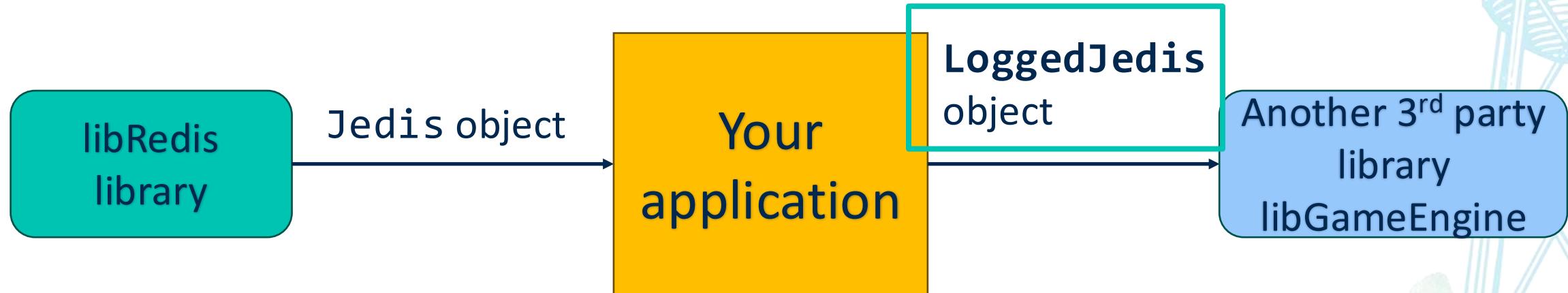
- Which OO principle to use?
- Inheritance!!

```
class Jedis {
 public void hget() { }
 public void hset() { }
}
```

```
class LoggedJedis extends Jedis {
 public void hget() {
 log(...);
 super.hget();
 }

 public void hset() {
 log(...);
 super.hset();
 }
}
```

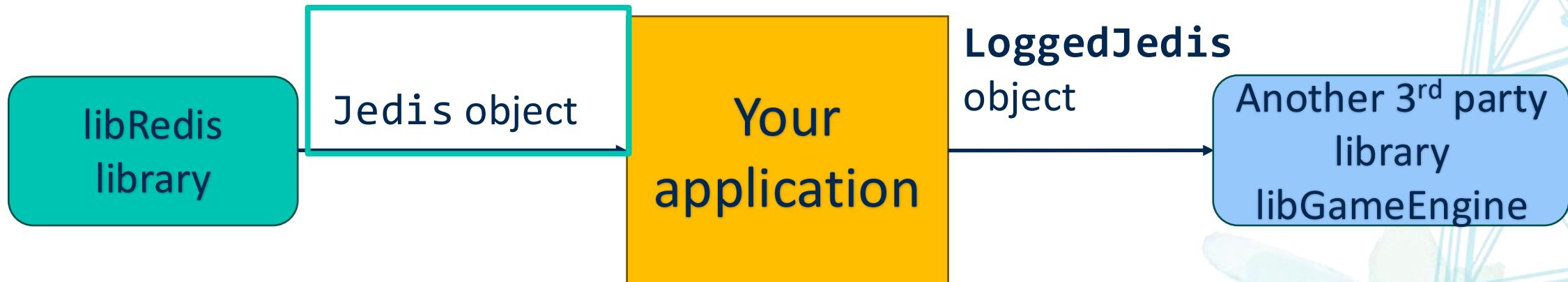
# Can pass LoggedJedis object to libGameEngine



- Why can I pass a `LoggedJedis` to `libGameEngine` when it expects a `Jedis` object?
  - Because it extends `Jedis`; `LoggedJedis` object *is a* `Jedis` object
  - Because of runtime polymorphism it will execute `LoggedJedis`'s `hget` and `hset` methods

# But can't modify Jedis library

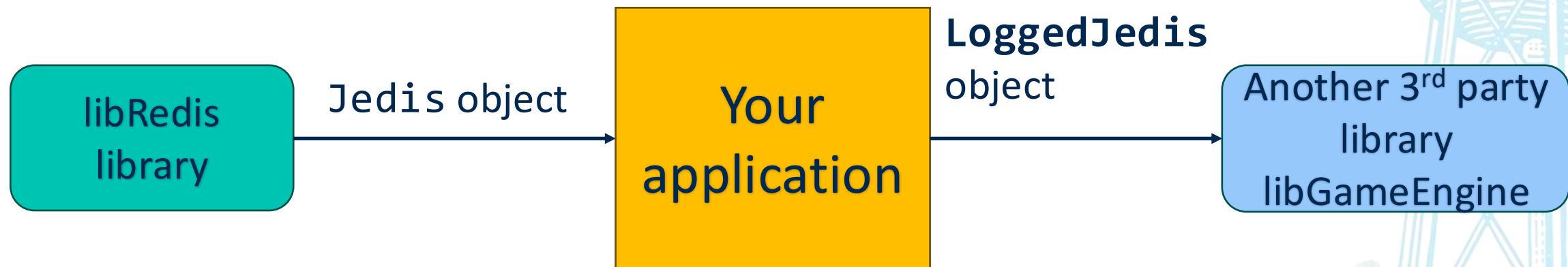
- Can't get libRedis library to return a LoggedJedis object



*How to convert Jedis object to LoggedJedis object?*

# But can't modify Jedis library

- Option 1 - Manually copy the fields

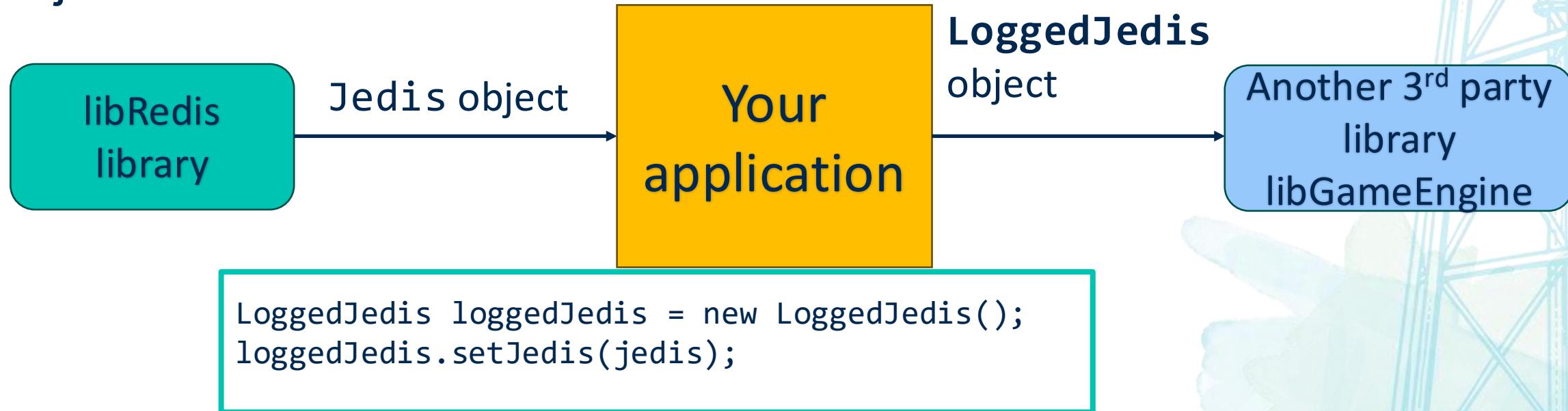


```
LoggedJedis loggedJedis = new LoggedJedis();
loggedJedis.setUrl(jedis.getUrl());
loggedJedis.setPort(jedis.getPort());
```

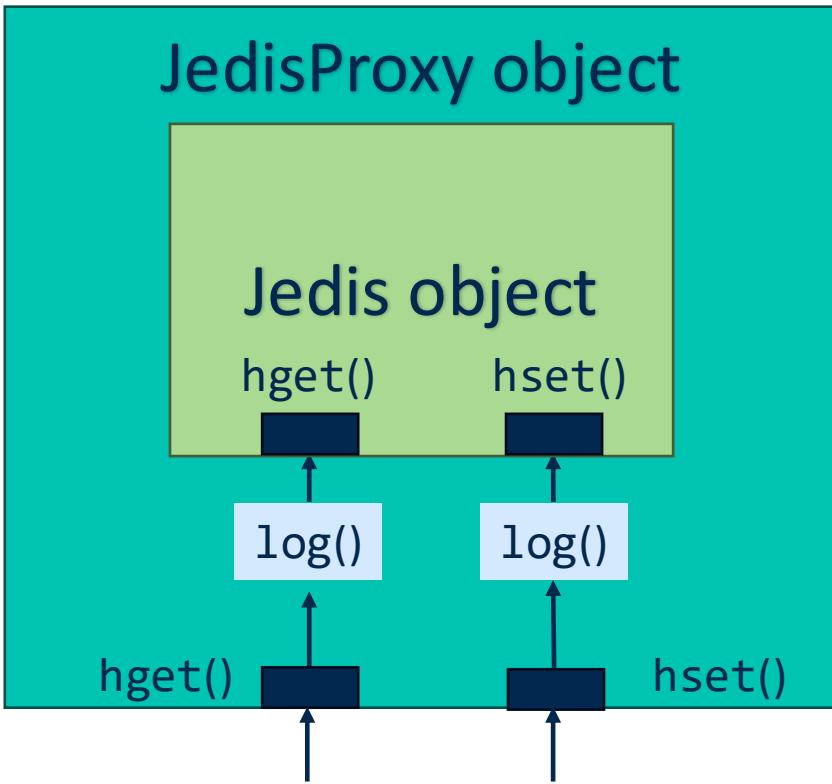
- Error-prone
- Can't access private fields
- Must deep clone any references contained by Jedis object
- Must keep updated with any changes in libRedis

# But can't modify Jediis library

- Proxy option: LoggedJedis object *contains* the original/target jedis object



# Proxy object wraps and extends target class



***Proxies intercept the method invocations***

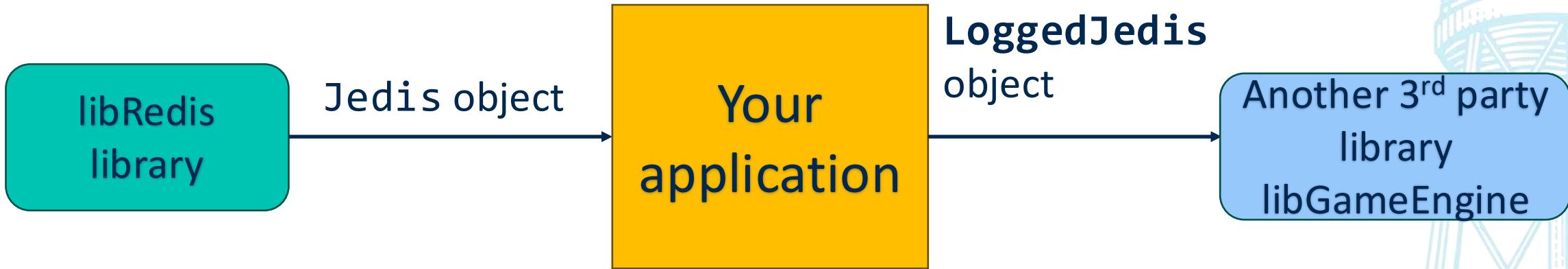
```
class Jedis {
 public String hget(String id) { ... }
 public void hset(String id, String val) { ... }
}

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);
 }
}
```

# 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
- Proxy object must *at least* manually forward all method invocations to the target object
  - Or the target object's method is not invoked at all and functionality breaks
- **Limitation:** proxy classes must be statically designed for each “proxyable” class
  - Duplicated logging logic

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

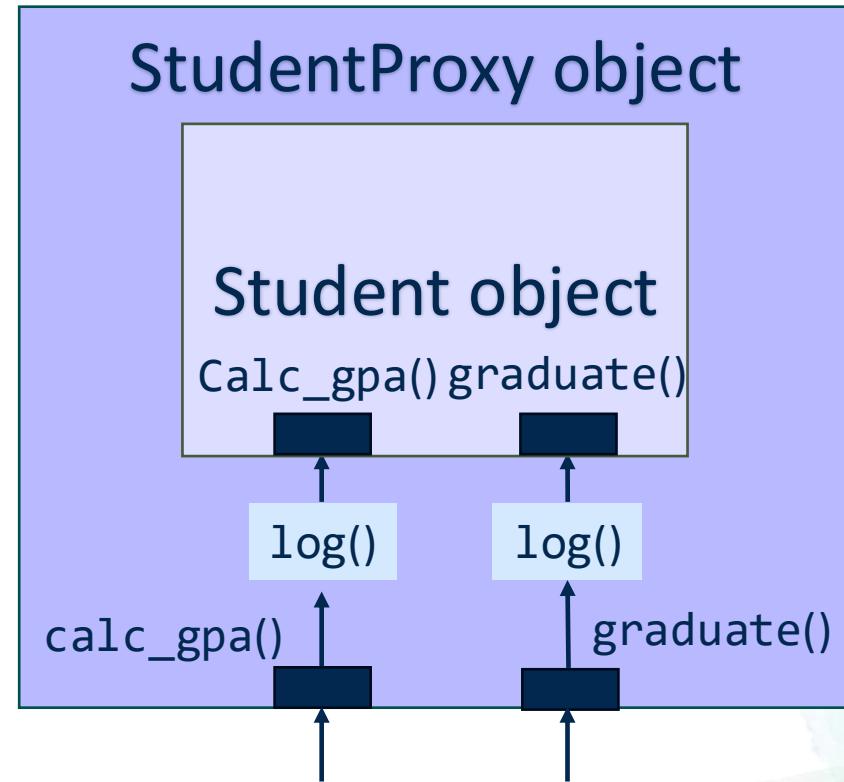
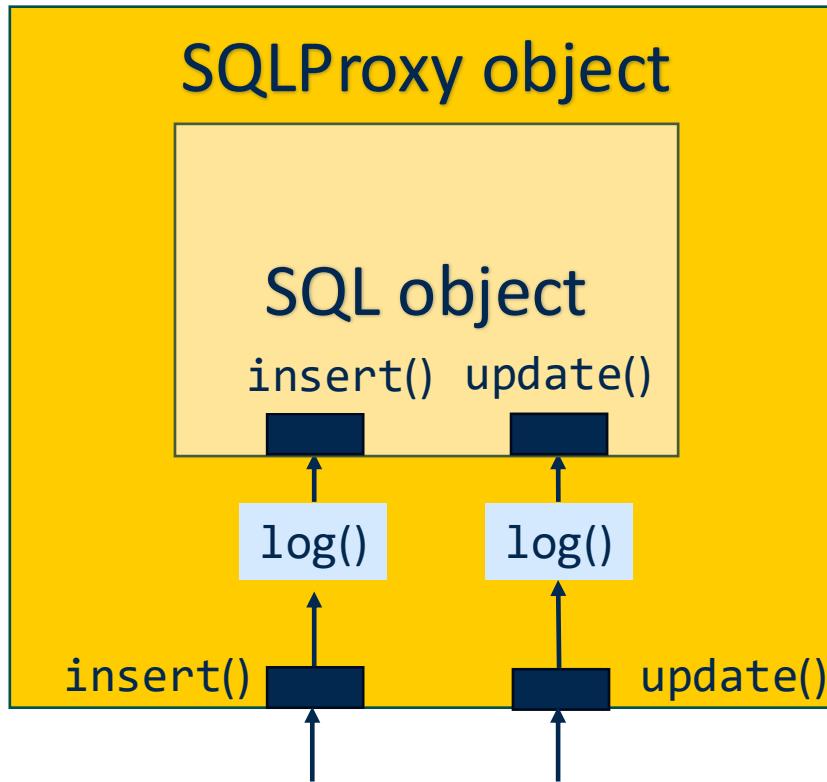
```
class SQLProxy extends Query{
 public log() { S.o.p(...); }

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

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

# Duplicated work

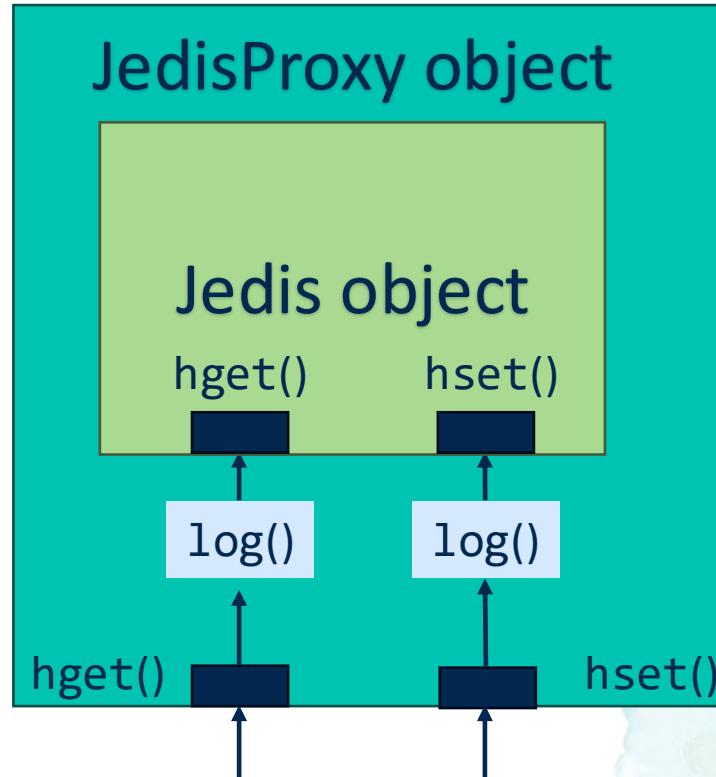
- Add logging to other classes



*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 generated `JedisProxy` ***on the fly***



# 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 JediProxy ***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 them 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

# Proxying using Javassist (extra-credit HW2)

- Create a ProxyFactory and set the parent class
- Implement the interface MethodHandler (possibly as an anonymous inner class)
- Override the invoke method to log
  - Then finally invoke the target object's method

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

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

 MethodHandler methodHandler = new 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);
 }
 }; ...
}
```

*Proxy's additional functionality*

*All methods of object intercepted by invoke method*

# Proxying using Javassist (extra-credit HW2)

- Arguments to invoke method
  - self – the target object
  - proceed – the invoked method in the target object
  - thisMethod – the invoked method in the proxy object
  - args – any arguments passed to the method invocation

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

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

 MethodHandler methodHandler = new MethodHandler() {
 @Override
 public Object invoke(Object self,
 Method proceed,
 Method thisMethod,
 Object[] args) throws Throwable {
 log("accessing method" +
 thisMethod.getName());
 return proceed.invoke(self, args);
 }
 }; ...
}
} All methods of object
intercepted by invoke method
```

*Proxy's additional functionality*

# Proxying using Javassist (extra-credit HW2)

- Create the proxy object by getting the constructor of the proxy class and invoking `newInstance` on it, using Reflection
- Set the handler
- Return the proxy object, which is a subclass of the original class type

```
Object createProxy(Object object) {
 // ... previous slide

 Class<?> proxyClass = proxyFactory.createClass();

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

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

 return proxyObject;
}
```

# Complete solution

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

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

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

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

```
Jedis jedis = ...;

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

// jedisProxy has type “subclass of Jedis” and is
// generated at runtime
// hset will first print the log and then perform
// hset operation

Student student = ...;

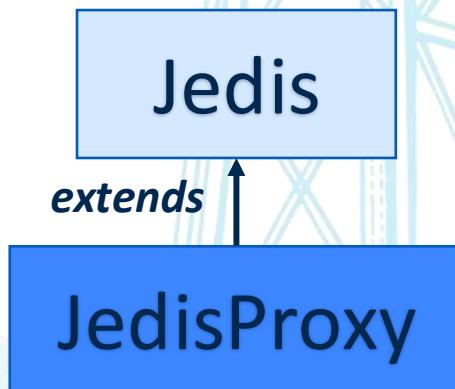
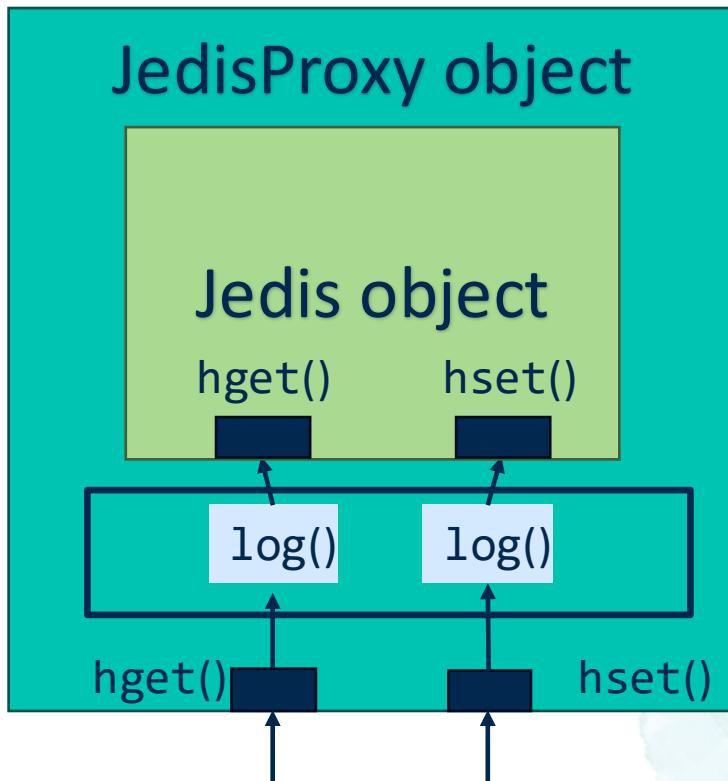
Student studentProxy = createProxy(student);
studentProxy.getName();

// studentProxy has type “subclass of Student”
// and is generated at runtime
// getName() will first print the log and then
// perform the getName operation

// ... Can dynamically create proxies of any
// object using createProxy() method
```

# Proxies: recap

- A proxy is a wrapper around a target object
- The proxy intercepts method invocations
  - Performs additional functionality
  - Dispatches the method to the target object
- Proxy class extends target class
  - Allows proxy to replace the target object for all uses



# Proxies: recap

- Dynamic proxies allow you to create Proxy classes at runtime
  - Of objects of any type
  - With any additional functionality
- Needs bytecode instrumentation (using JavaAssist)
- Imagine you have a method `Object createProxy(Object target)`
  - `Jedis jedisProxy = (JedisProxy) createProxy(jedis);`
  - `Student studentProxy = (StudentProxy) createProxy(student);`

# Proxies: recap

```
class TargetClass {
 // private fields
 // public methods
 public void doThis() { /*... */ }
 public void doThat() { /*... */ }
}
```

```
class TargetClassProxy extends TargetClass {
 private TargetClass target;
```

```
 public void doThis() {
 // additional functionality
 target.doThis();
 }

 public void doThat() {
 // additional functionality
 target.doThat();
 }
}
```

*Static Proxy*

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

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

 MethodHandler methodHandler = new
 MethodHandler(target) {};

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

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

*Dynamic Proxy*

# Proxies: recap

*Can handle any Type*

```
class TargetClass {
 // private fields
 // public methods
 public void doThis() { ... }
 public void doThat() { ... }
}
```

```
class TargetClassProxy extends TargetClass {
 private TargetClass target;

 public void doThis() {
 // additional functionality
 target.doThis();
 }

 public void doThat() {
 // additional functionality
 target.doThat();
 }
}
```

**Static Proxy**

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

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

 MethodHandler methodHandler = new
 MethodHandler(target) {};

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

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

**Dynamic Proxy**

*Set up*

*Generate proxy class,  
create object, set  
additional functionality*

*Javassist will call  
invoke() for all  
proxy methods*

*Additional Functionality*

# Proxies: recap

```
class TargetClass {
 // private fields
 // public methods
 public void doThis() { /*... */ }
 public void doThat() { /*... */ }
}
```

```
class TargetClassProxy extends TargetClass {
 private TargetClass target;

 public void doThis() {
 // additional functionality
 target.doThis();
 }

 public void doThat() {
 // additional functionality
 target.doThat();
 }
}
```

```
TargetClassProxy proxy = new TargetClassProxy();
```

**Static Proxy**

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

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

 MethodHandler methodHandler = new
 MyMethodHandler(target);

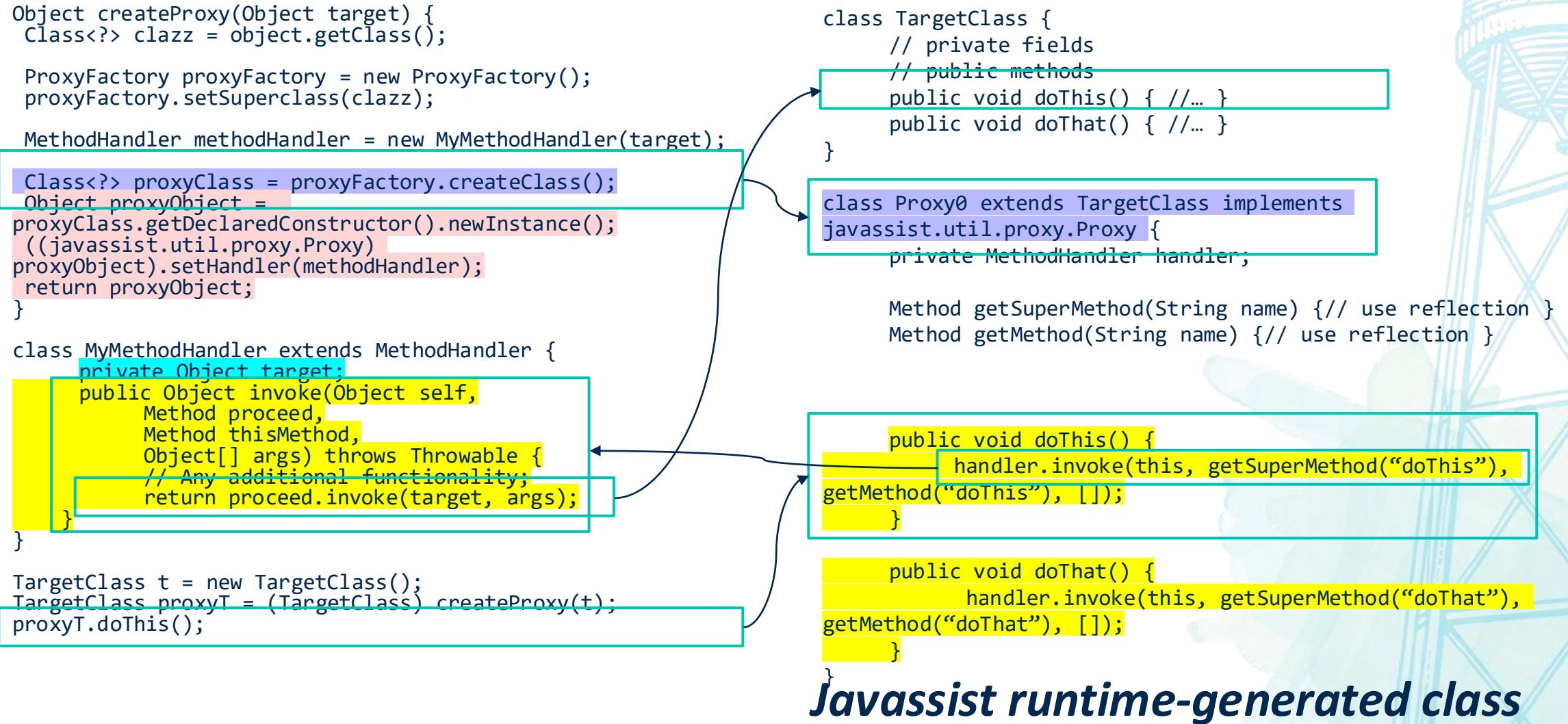
 Class<?> proxyClass = proxyFactory.createClass();
 Object proxyObject =
 proxyClass.getDeclaredConstructor().newInstance();
 ((javassist.util.proxy.Proxy)
 proxyObject).setHandler(methodHandler);
 return proxyObject;
}
```

```
class MyMethodHandler extends MethodHandler {
 private Object target;
 public Object invoke(Object self,
 Method proceed,
 Method thisMethod,
 Object[] args) throws Throwable {
 // Any additional functionality;
 return proceed.invoke(target, args);
 }
}
```

*Javassist will call  
invoke() for all proxy  
methods*

**Dynamic Proxy**

# Javassist ... behind the scenes



# Demo

- [https://github.com/davsec-teaching/javassist\\_demo](https://github.com/davsec-teaching/javassist_demo)

# Real world use cases of proxies

- Sample use cases
  - Lazy loading
  - Mock testing



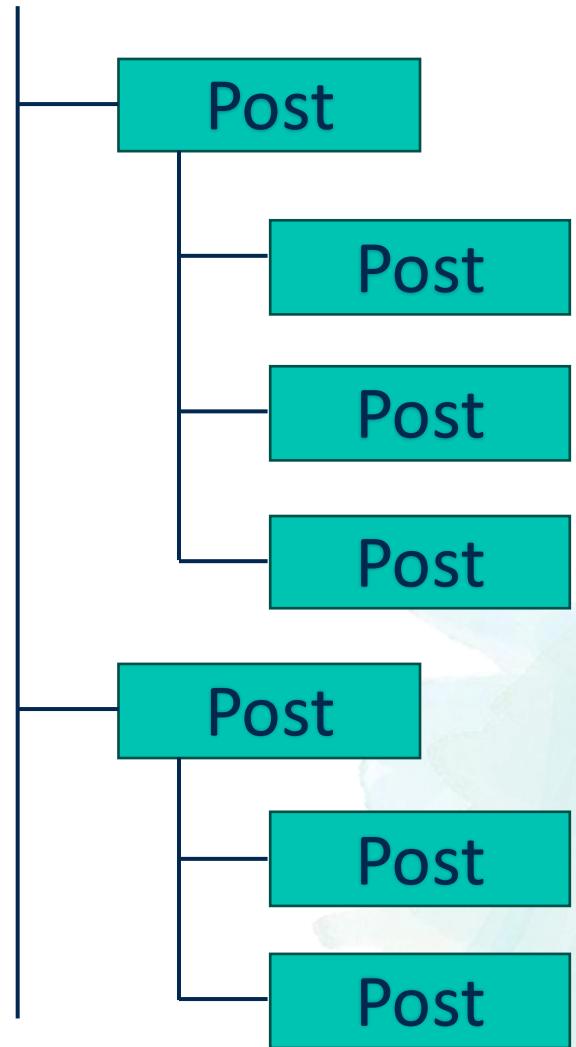
HIBERNATE



EAS<sup>Y</sup>MOCK

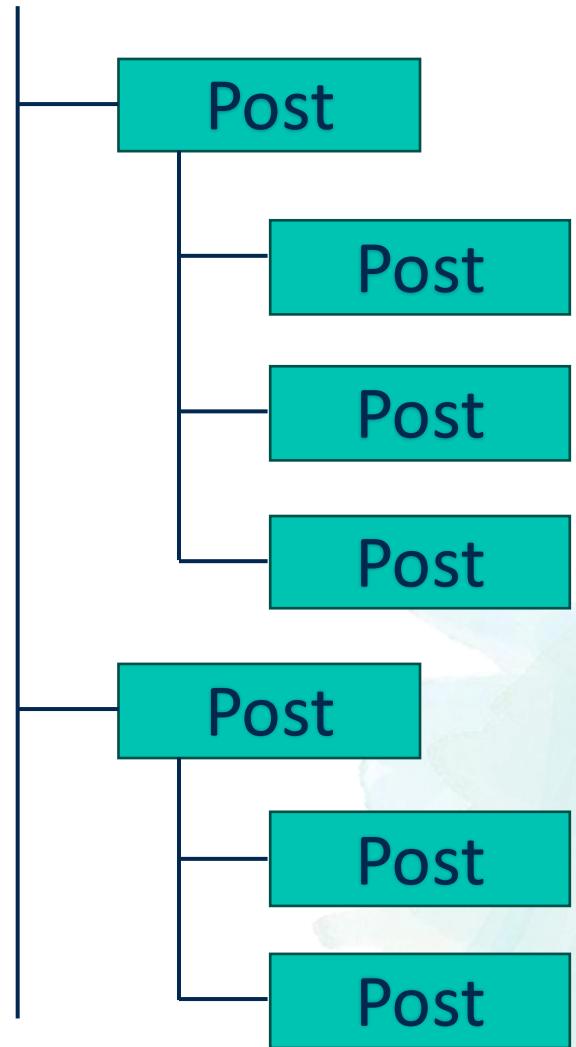
# Loading posts

- 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(",,");

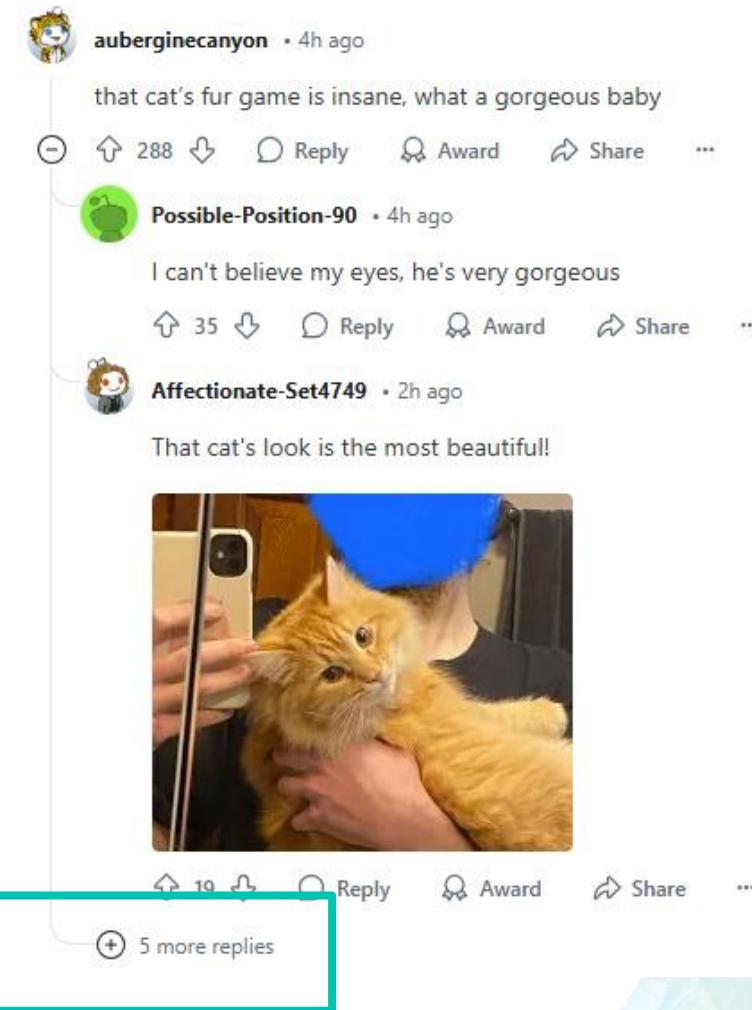
for (String replyId: replies) {
 replyMap = jedis.hgetAll(replyId);
 Post reply = new Post();
 reply.setId(replyId);
 reply.setCreatedAt(replyMap.get("createdAt"));
 post.getReplies().add(reply);
}
```

```
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*

# Lazy loading posts UI

- Common performance improvement
- Replies are loaded only when user indicates they want to see the replies
- Improves UI responsiveness



# Lazy loading

- Common performance improvement
- Replies are loaded only when user indicates they want to see the replies
- Clicking on ‘+’ loads the remaining replies

The screenshot shows a social media feed with a post from 'auberginecanyon' about a cat. The post has 288 upvotes. Below it, a reply from 'Possible-Position-90' with 35 upvotes and a photo of an orange cat wearing a blue hat. Another reply from 'Affectionate-Set4749'称赞 the cat's look. A reply from 'FailedProposal' says the cat is pretty. A reply from 'floofienewfie' says it's a floof with heart and paw emojis. A reply from 'Real\_Winter\_3794' shows a video thumbnail of a cat. A final reply from 'Safe-Captain-9066' says the cat has a beautiful glow up.

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 ...

# Mock UI code for posts

- Imagine this is a single application
  - Ignore network for a moment
- Java UI
  - Window, Button
- Java objects of type Post

```
class UIButton {
 void onClick() {
 // event handler to be triggered on click
 }
}

class UIWindow {
 // ...
}

class Post {
 private String content;
 private String createdAt;
 private List<Post> replies;

 // getters and setters
 List<Post> getReplies() {return replies;}
}

public static void main(String[] args) {
 Post post = loadPost("3208");
 UIWindow window = new UIWindow();
 window.add(new UIText(post.getContent()));
 window.add(new UIButton("+"));
}
```

# Lazy loading replies

- Idea

- loadPost only loads the post
- Replies loaded from database when the button is clicked
  - onClick event is triggered

```
class UIButton {
 void onClick() {
 // event handler to be triggered on click
 }
}

class UIWindow {
 // ...
}

class Post {
 private String content;
 private String createdAt;
 private List<Post> replies;

 // getters and setters
 List<Post> getReplies() {return replies; }
}

public static void main(String[] args) {
 Post post = loadPost("3208");
 UIWindow window = new UIWindow();
 window.add(new UIText(post.getContent()));
 window.add(new UIButton("+"));
}
```

# Lazy loading replies

- Original loadPost method
- How many database accesses?

```
// original loadPost method

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) {
 replyMap = jedis.hgetAll(replyId);
 Post reply = new Post();
 reply.setId(replyId);
 reply.setCreatedAt(replyMap.get("createdAt"));
 post.getReplies().add(reply);
 }
 return post;
}
```

# Lazy loading replies

- loadPost method with lazy loading
  - Only load the post fields and the ids of the replies
  - How many database accesses?
  - Much faster (even more so for on-disk SQL databases)

```
// original loadPost method

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) {
 replyMap = jedis.hgetAll(replyId);
 Post reply = new Post();
 reply.setId(replyId);
 reply.setCreatedAt(replyMap.get("createdAt"));
 post.getReplies().add(reply);
}

return post;
}
```

# Lazy loading replies

- `onClick` fetches the replies only if the button is clicked
  - Fetch all the replies
  - Draw them on the UI Window

```
class UIButton {
 Post post;
 void onClick() {
 for (Reply reply: post.getReplies()) {
 // only the id is populated, so must
 // invoke loadPost
 Reply reply = loadPost(reply.getId());
 window.add(new UIText(reply.getContent()));
 }
 }

 class UIWindow {
 // ...
 }
}
```

# Problem

- UI designers shouldn't have to worry about backend concerns
- Shouldn't have to know which method to invoke to load the reply
- Only `window.add(new UIText(reply.getContent()));`
- How to achieve this?

```
class UIButton {
 Post post;
 void onClick() {
 for (Reply reply: post.getReplies()) {
 // only the id is populated, so must
 // invoke loadPost
 Reply reply = loadPost(reply.getId());
 window.add(new UIText(reply.getContent()));
 }
 }

 class UIWindow {
 // ...
 }
}
```

# An option ...

- Stick the lazy loading logic in the Post class
- Terrible, terrible idea!!
- Assume that posts are always lazy loaded
- Assume that posts are always loaded from the database
- Breaks all reusability
- Breaks SRP

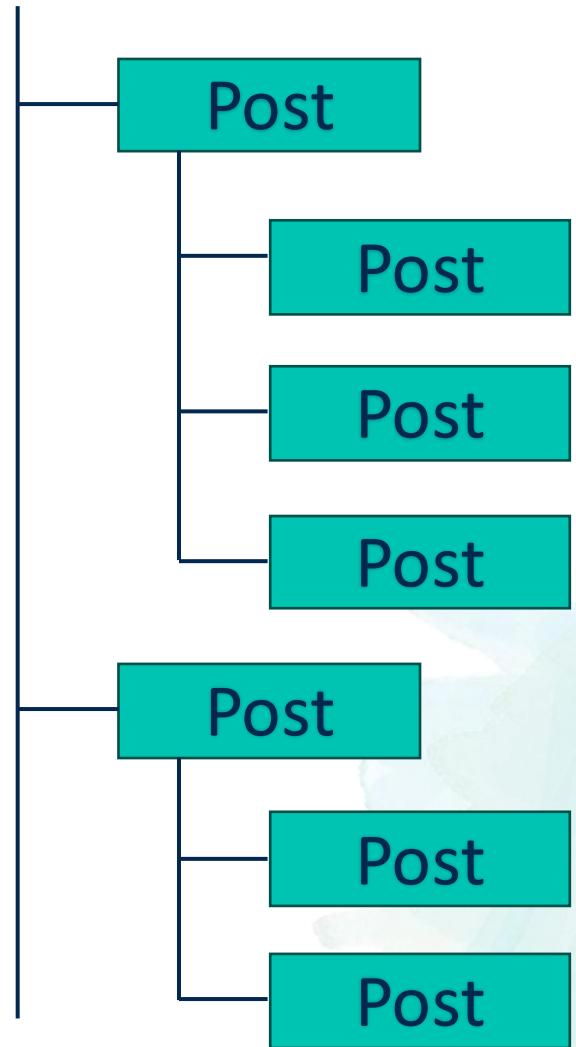
```
class Post {
 private String content;
 private String createdAt;
 private List<Post> replies;

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

# Dynamic proxy use cases

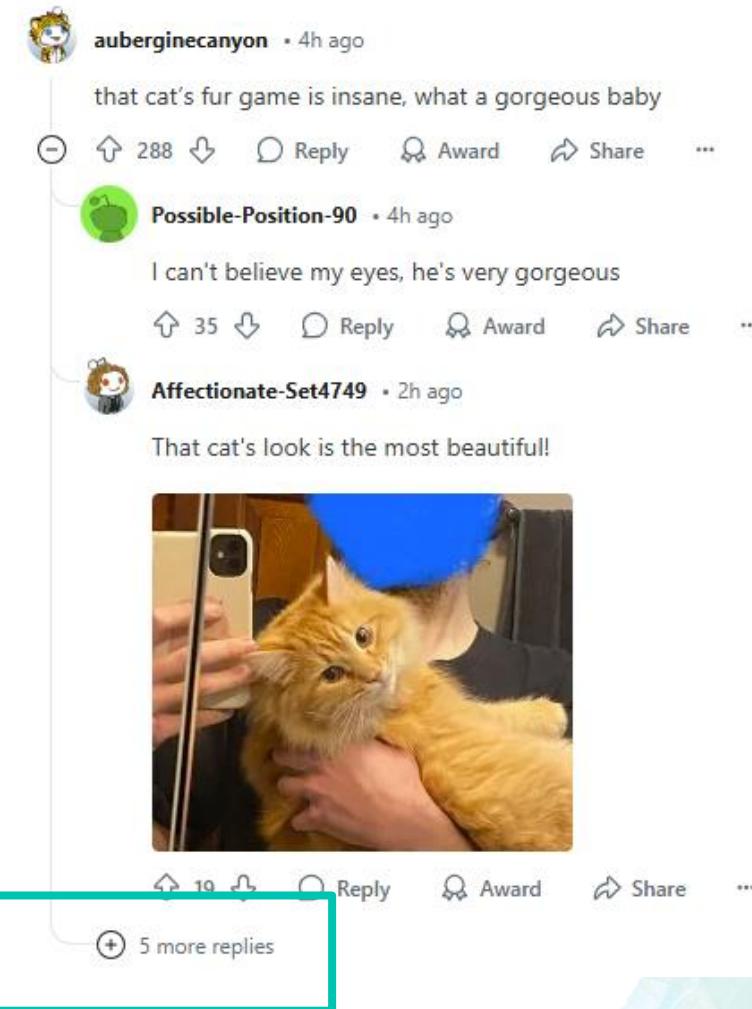
# Loading posts

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



# Lazy loading posts UI

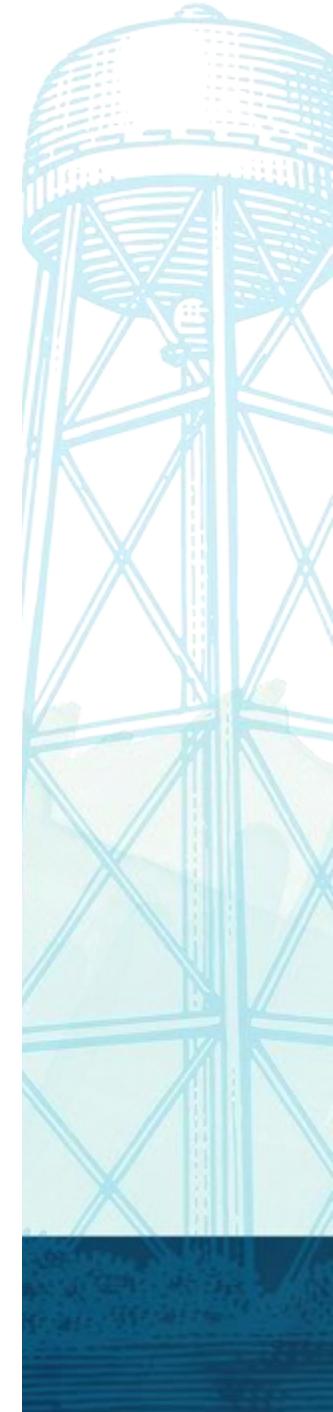
- Common performance improvement
- Replies are loaded only when user indicates they want to see the replies
- Clicking on ‘+’ loads the remaining replies



# Lazy loading

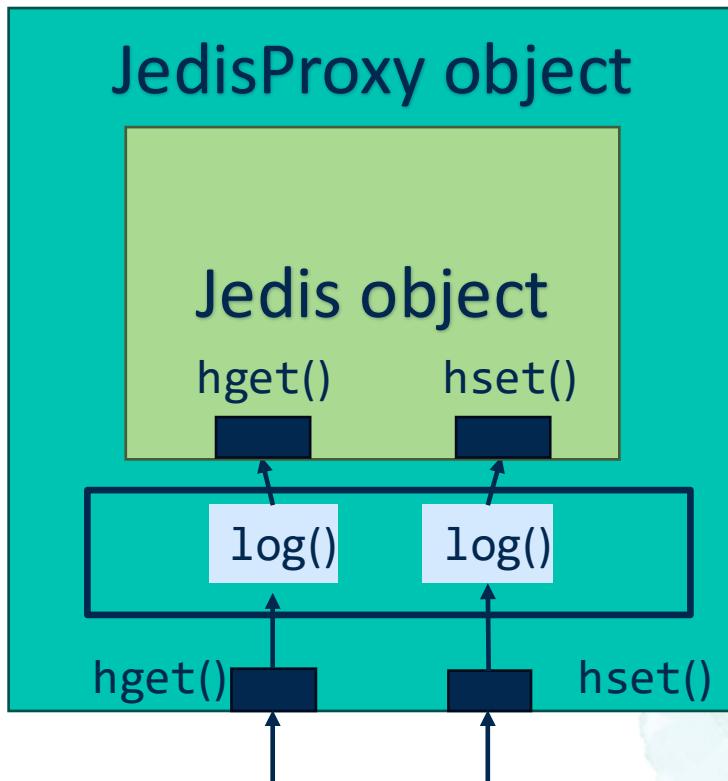
- Clicking on ‘+’ loads the remaining replies
- Dynamic proxies simplify implementing lazy loading
- This class: *sketch of a solution (not full solution)*

The image shows a social media feed with several comments and a video. At the top, a user named 'auberginecanyon' posted a comment 4 hours ago, saying 'that cat's fur game is insane, what a gorgeous baby'. Below it, 'Possible-Position-90' responded 4 hours ago with 'I can't believe my eyes, he's very gorgeous'. Another user, 'Affectionate-Set4749', commented 2 hours ago that 'That cat's look is the most beautiful!' A photo of an orange cat wearing a blue hat is shown. Further down, 'FailedProposal' said '^ came here to say this! Pretty cat' 4 hours ago. 'floofienewfie' replied 4 hours ago with 'What a floof! ❤️ ❤️ 🐱'. A video of a cat is also present. The final reply is from 'Safe-Captain-9066' 3 hours ago, saying 'True. What a beautiful glow up! ✨'.



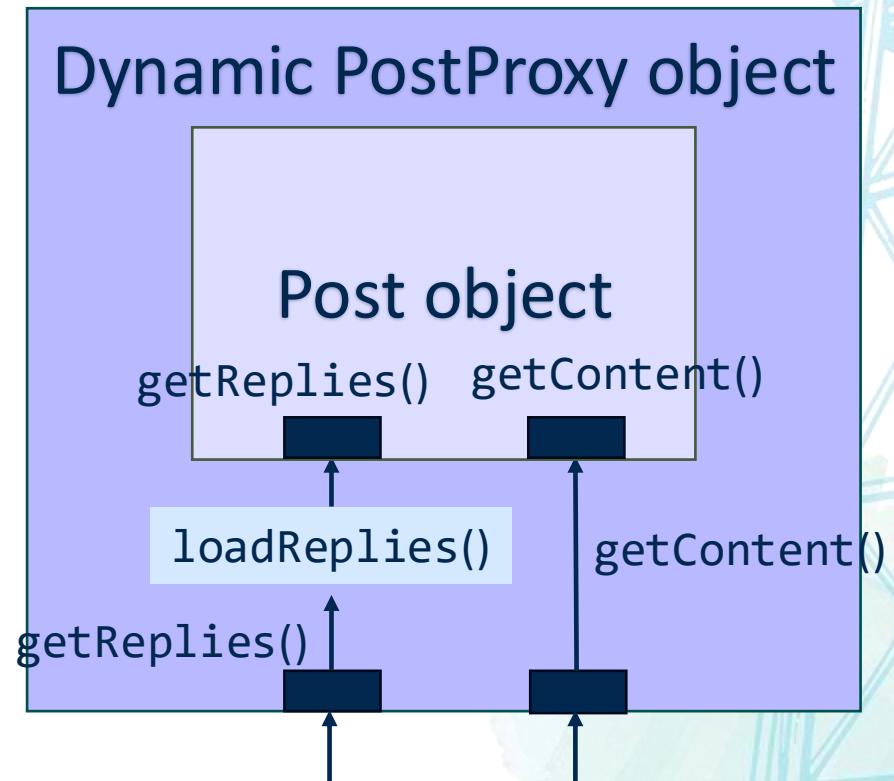
# Proxies: recap

- A proxy is a wrapper around a target object
- The proxy intercepts method invocations
  - Performs additional functionality
  - Dispatches the method to the target object
- Proxy class extends target class
  - Allows proxy to replace the target object for all uses



# 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

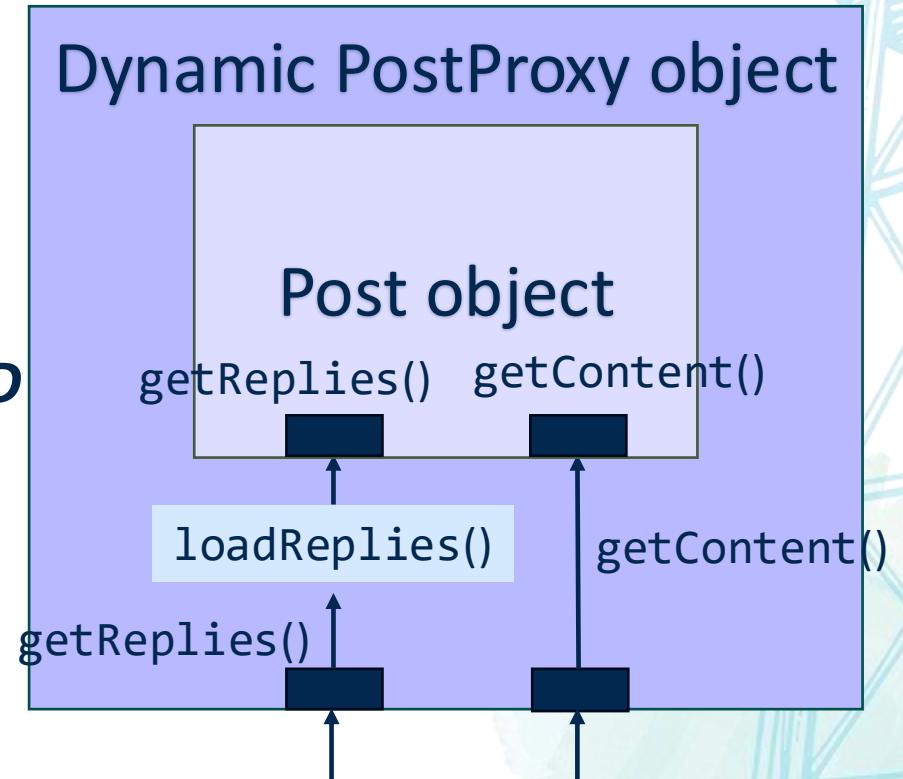


# Proxies for lazy loading

```
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) { Only set the Reply ID
 Post reply = new Post();
 // does not load the reply details
 reply.setId(replyId);
 post.getReplies().add(reply);
 }
 return createLazyLoadProxy(post);
}
```

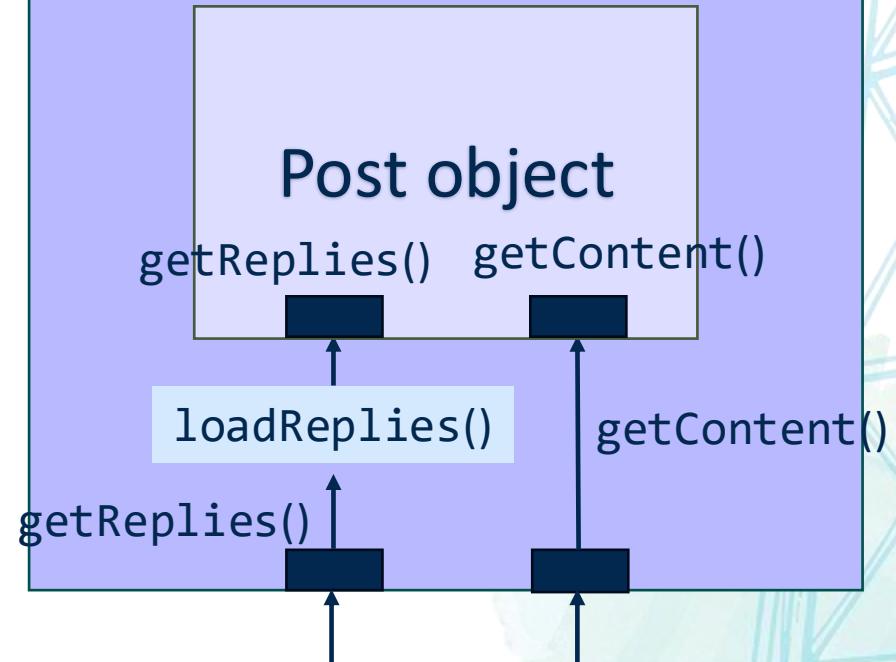


# Proxies for lazy loading

```
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*

Dynamic PostProxy object



*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);

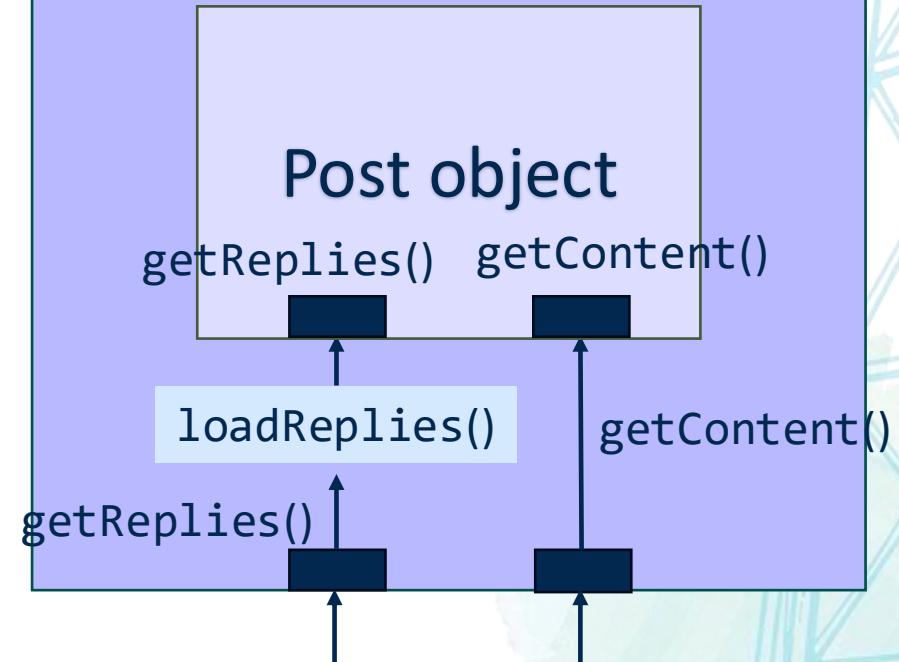
 Class<?> proxyClass =
 proxyFactory.createClass();
 Object proxyObject =
 proxyClass.getDeclaredConstructor().newInstance();
 ((javassist.util.proxy.Proxy)
 proxyObject).setHandler(methodHandler);
 return proxyObject;
}

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

**Invoke loadReplies()**

*Create PostProxy objects*

**Dynamic PostProxy object**



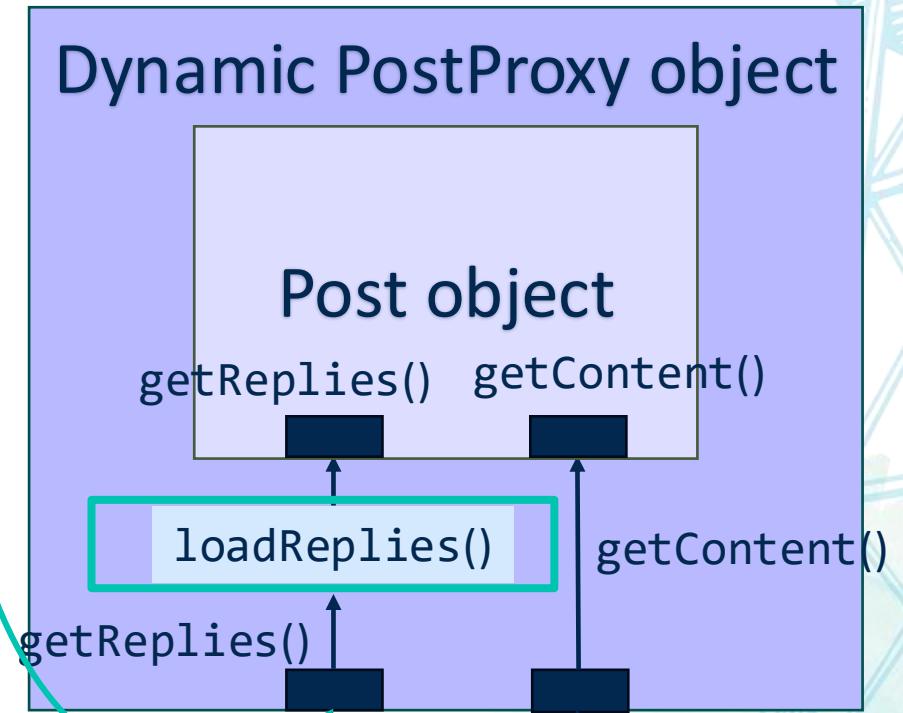
*createLazyLoadProxy returns the proxy object*

# Proxies for lazy loading

```
class UIButton {
 Post post;
 void onClick() { post is a proxy object
 for (Reply reply: post.getReplies()) {
 Reply reply = loadPost(reply.getId());
 window.add(new UIText(reply.getContent()));
 }
 }
}

class UIWindow {
 // ...
}

public static void main(String[] args) {
 Post post = loadPost("3208");
 UIWindow window = new UIWindow();
 window.add(new UIText(post.getContent()));
 window.add(new UIButton("+"));
}
```



# 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
}
```

# 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

# *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

# 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;
 }
}
```

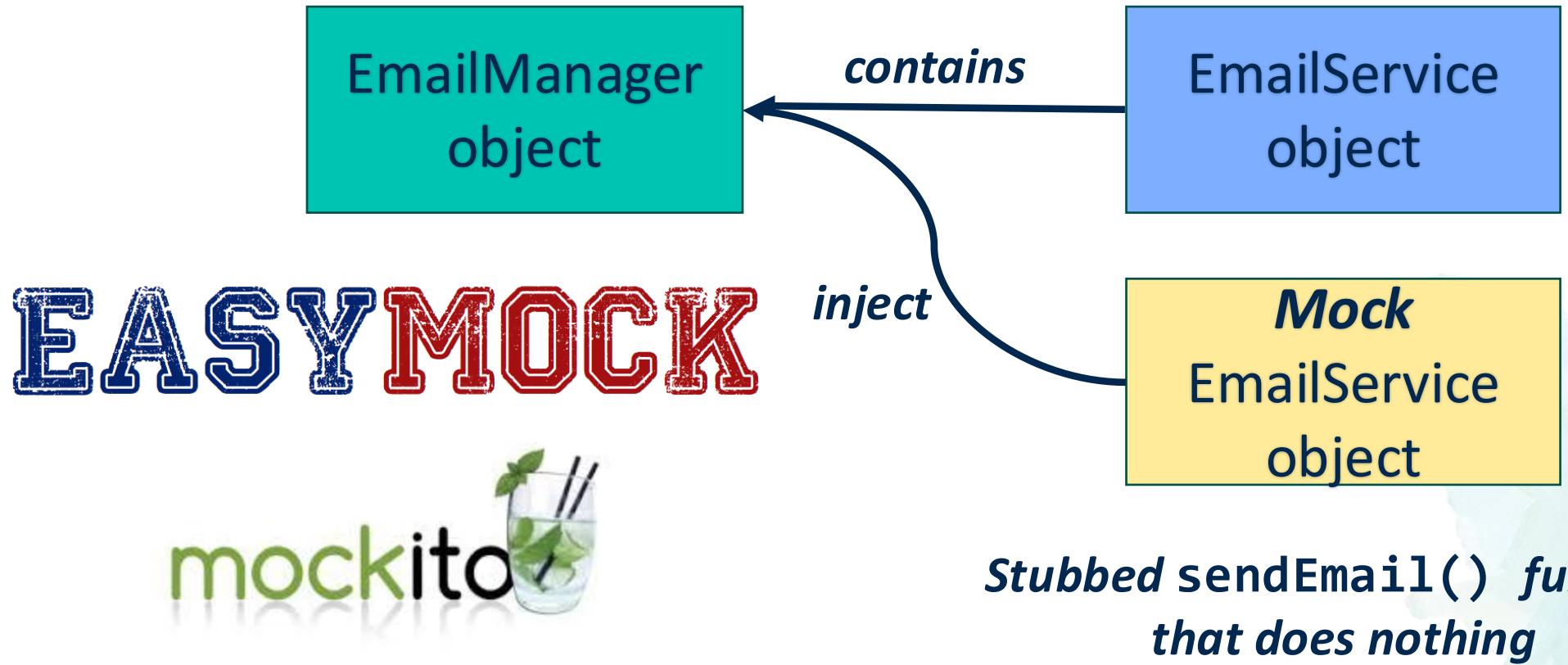
```
public class EmailManager {
 private EmailService emailService;

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

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

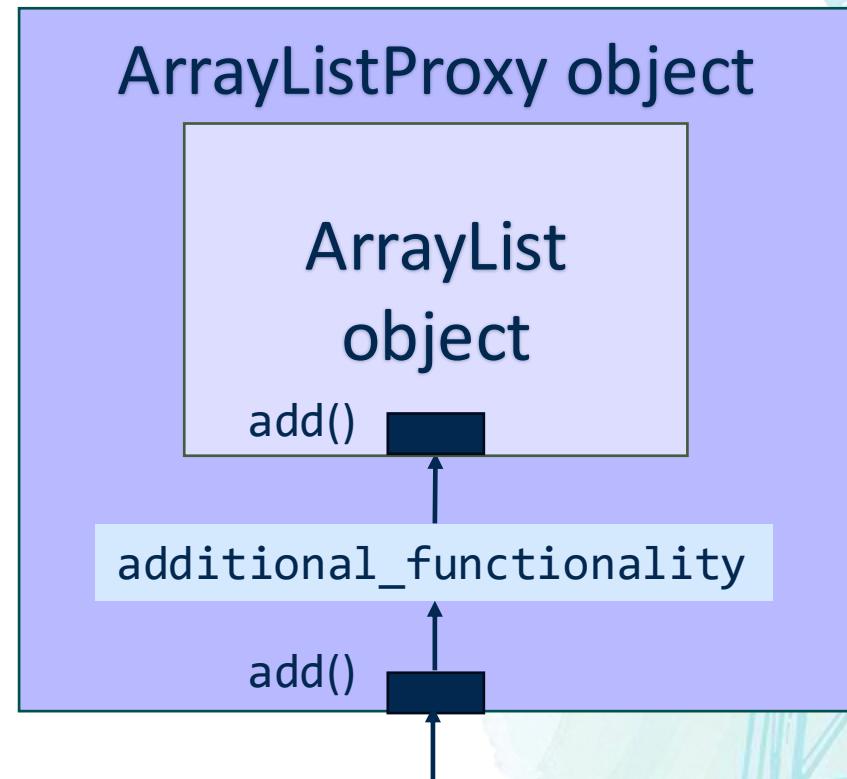
# Mockito overview

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



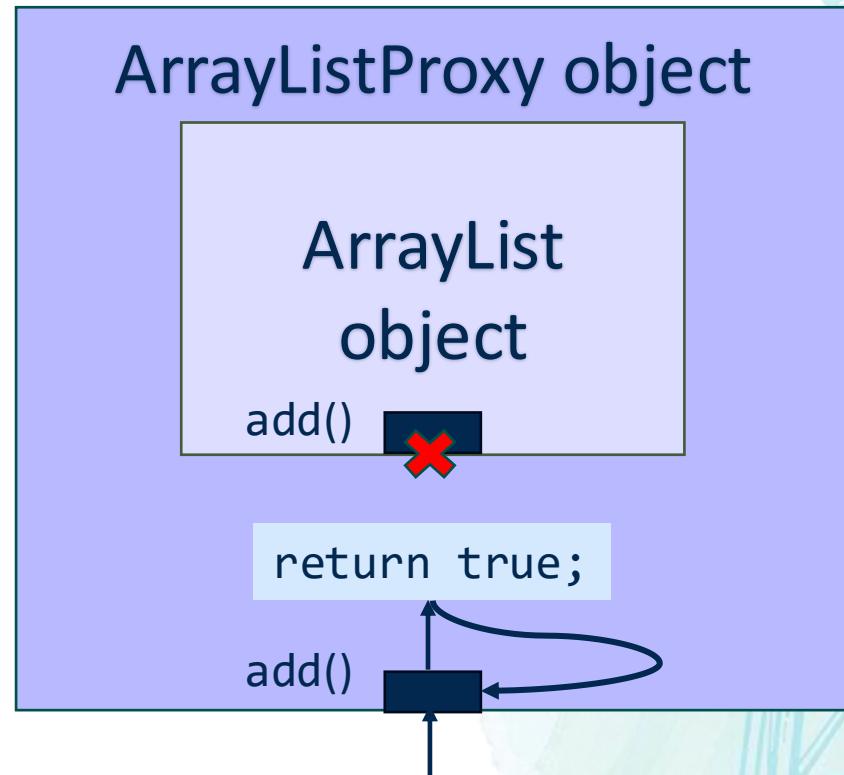
# Proxy can hijack functionality

- 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



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

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

 MethodHandler methodHandler = new 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);
 }
 };

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

# 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
 }
}
```

# 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
 }
}
```

# Concluding points

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

# Concluding points

- 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?
```

# Summary

- Problem: `persist()` method had to be duplicated for each class
  - Reflection allows for dynamic field access and method invocation
    - Partially solved the problem

```
class Post { }

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

 public void persist(Post post) {
 map.put("authorName", post.getAuthor());
 map.put(..., ...);
 // ... All relevant fields
 jedis.hset(id++, map);
 }

 public void persist(Student student) {
 map.put("studentName", student.getName());
 map.put(..., ...);
 // ... All relevant fields
 jedis.hset(id++, map);
 }
}
```

# Summary

- Problem: `persist()` method had to be duplicated for each class
- Reflection allows for dynamic field access and method invocation
  - Partially solved the problem

```
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;
 for (Field f: obj.getDeclaredFields()) {
 String fieldname = f.getName();
 Object fieldVal = f.get(obj);
 postMap.put(fieldname, fieldVal);
 }
 jedis.hset(id++, postMap);
 }
}

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

# Summary

- Problem: what if we don't want to persist all fields in an object?
  - Annotations allow adding meta-data to fields, methods, etc.
  - Annotations with `Retention.RUNTIME` are available at runtime
  - Annotating individual fields allows us to selectively persist

```
@retention(Retention.RUNTIME)
@interface Persistable {}
```

```
class Post {
```

```
 @Persistable
 Object postId;
```

```
 @Persistable
 String postContent;
```

```
 @Persistable
 String authorName;
```

```
 Integer tempField;
```

# Summary

- Problem: what if I want to *lazy-load* one or more fields
  - Dynamic proxies allow us to intercept method invocation
    - Allowing lazy-loading of fields only when the field is accessed

```
@retention(Retention.RUNTIME)
@interface Persistable {}

class Post {

 @Persistable
 Object postId;

 @Persistable
 String postContent;

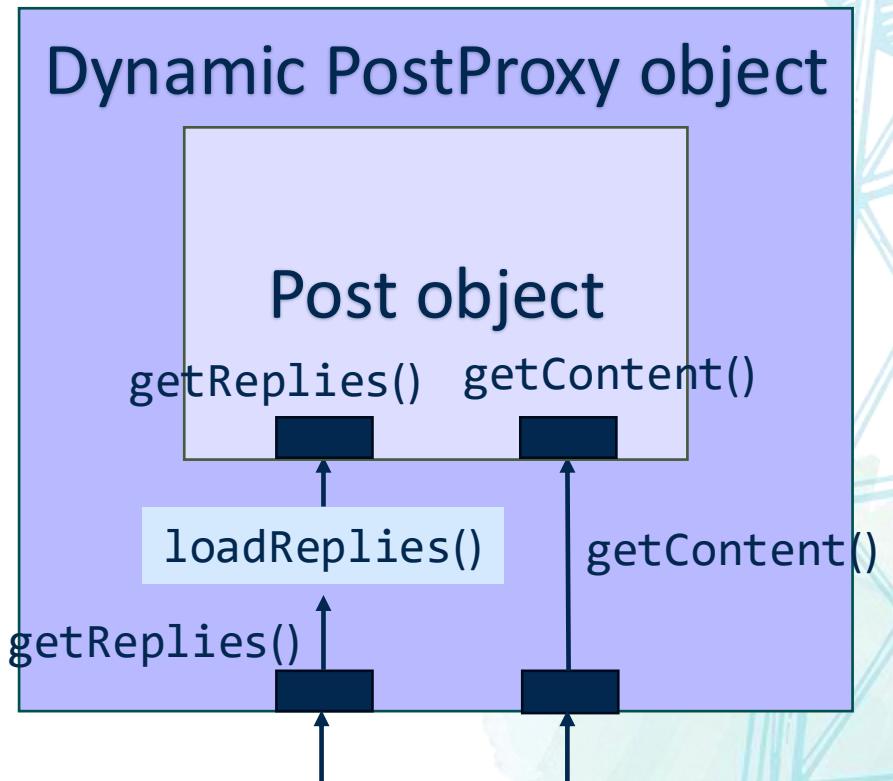
 @Persistable
 String authorName;

 @Persistable
 @LazyLoad
 List<Post> replies;

 Integer tempField;
}
```

# Summary

- Problem: what if I want to *lazy-load* one or more fields
  - Dynamic proxies allow us to intercept method invocation
    - Allowing lazy-loading of fields only when the field is accessed



# 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

