Java reflection, proxies, and annotations

Tapti Palit

Why do we need reflection?

Next few topics

- Focus shifts from individual programmer perspective
- Focus more on the larger software eco-system
- How do you design code that can be easily reused by others?
  - How do we design frameworks and libraries
    - In a way that *reduces* programmer effort?
    - In a way that *ensures* the programmer cannot *accidentally* misuse it?
      - You assume `methodA()` is invoked *before* `methodB()` but the programmer might not know this

Recall: Redis persistence from HW1

- How do you design the Redis persistence class?

Potential design

- Design a `RedisDB` class that encapsulates all Redis persistence functionality
- `RedisDB` class internally uses a `Jedis` object from the jedis library
- Expose `persistAll` and `loadAll` methods to persist and load Post

```
class Student implements Persistable{
    void serialize() { return String(id + name +
age + …);
}

class Persistable { void persist(String id, Stirng
value) { jedis.hset(id, value); }

class RedisDB {
    Jedis jedisSession;


    public void persistAll(Student student) {
        Map<String, String> studentMap;
        studentMap.put(“studentName”,
student.getName());
        studentMap.put(…, …);
        // ... All relevant fields
        jedis.hset(student.getId(), studentMap);
    }


}
```

Thought experiment

- Imagine HW2 assignment asks you to do the following -
- Design classes for social media platform
  - User, Post, Comment, Feed, Platform, Scheduler, Analytics, Message, Notification
- Add methods to the `RedisDB` class to persist all of them to a Redis database
- How to design/implement this?

Goal: reuse persistence code

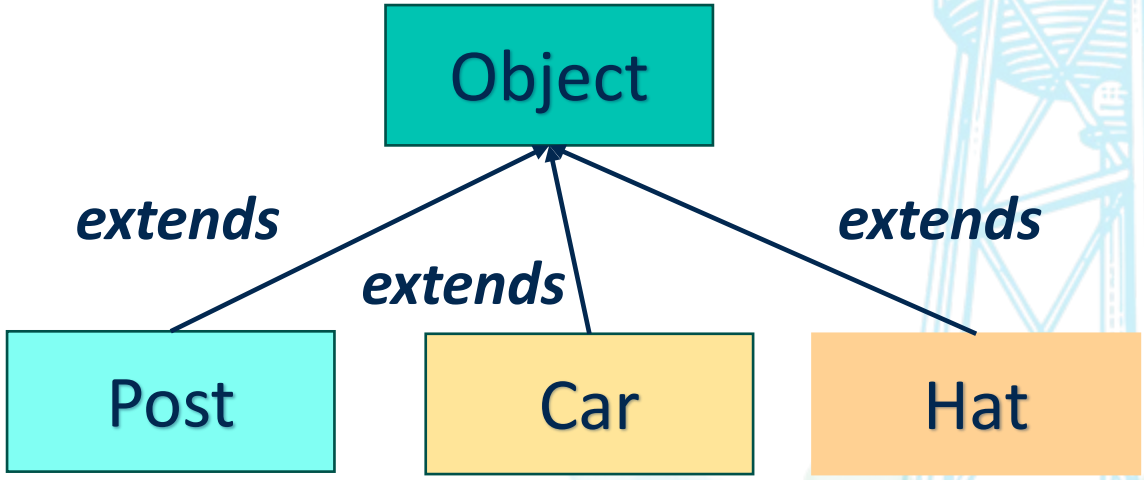
- ***I wrote the persistence code once, why should I have to write it again?***
- ***… what are the options?***

Goal: reuse persistence code

- Two pieces to this puzzle
  - Reflection
  - Annotation

Background

- In Java, all classes inherit the Object class implicitly



- Every instance method call's first implicit argument is the object it is being applied to
  - `Student s = new Student();`
    `s.getName(); // implicit first argument is s`

Background

- Instance method invocation
  - Every instance method call's first implicit argument is the object it is being applied to
    - ```
      Student s = new Student();
      s.getName(); // implicit first argument is s
      ```
  - Not applicable to `static` functions [in Java]
      ```

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
- Purpose
  - Access and modify classes, methods, fields, and constructors dynamically
  - Facilitate frameworks, libraries, and tools (e.g., Spring, Hibernate)

Reflection and metaprogramming

- Types of Reflection
  - Compile time (C++ [https://en.cppreference.com/w/cpp/keyword/reflexpr.html], Rust macros)
  - Runtime (Java, JavaScript)

Java reflection

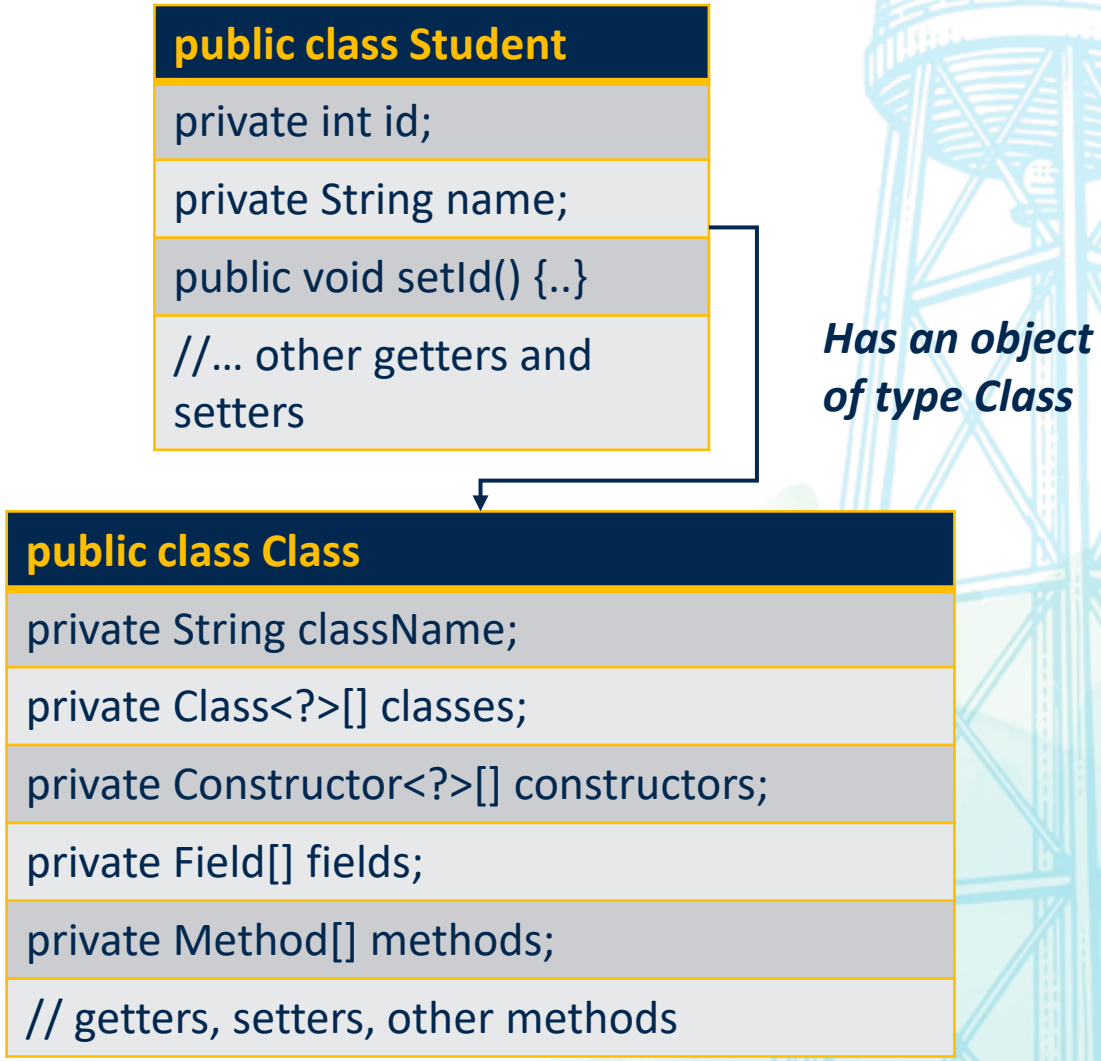
- Provides APIs to inspect and manipulate the classes, methods, fields, and so on…
- Note: reflection bypasses all encapsulation
  - But (hopefully) for greater good!!

Key classes

- `java.lang.Class`: Represents a class or interface
- Every class definition in our program would have a Class object that Java compiler automatically creates

Key classes

- java.lang.Class: Represents a class or interface
- Every class definition in our program would have a Class object that Java compiler automatically creates



Key classes

- java.lang.Class: Represents a class or interface
- Every class definition in our program would have a Class object that Java compiler automatically creates

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

Class object
private String className; **// "Student"**
private Constructor<?>[] constructors; **// [default_cons]**
private Field[] fields; **[field_obj_id, field_obj_name]**
private Method[] methods; **[//all getters and setters]**
// getters, setters, other methods

Key classes

- `java.lang.reflect.Method`: Represents a class method
- `java.lang.reflect.Field`: Represents a class field
- `java.lang.reflect.Constructor`: Represents a constructor
- ***… what can we do with these objects of type Class, Method, Field, etc?***

Reflection example

- E.g., using reflection to dynamically invoke all getters 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
- ***… what did we achieve?***

```
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"
```

Reflection API

- `Class.forName(<classname>)` – loads the Class object provided a fully qualified class name (`com.ecs160.MyClass`)
- `<obj>.getClass()` - Gets the `Class` object for the object `obj`
- `<class_obj>.getMethods()` – Get all method objects of type `Method` for the Class object `class_object`
- `<class_obj>.getMethod(<method_name>)` - Get the `Method` object for the method having `method_name`

Reflection demo

- In class demos
  - Demo1: Print all classes and methods in each class in a package
  - Demo2: Invoke all getters of an object
  - Demo3: Access private fields

Reflection demo Github link

- https://github.com/davsec-teaching/reflection_demo/tree/master

Java reflection, proxies, and annotations

Tapti Palit

Announcements

- Start working on your assignments!
- Next quiz on 10/24
  - Will cover everything covered till 10/20 (inclusive)

Agenda

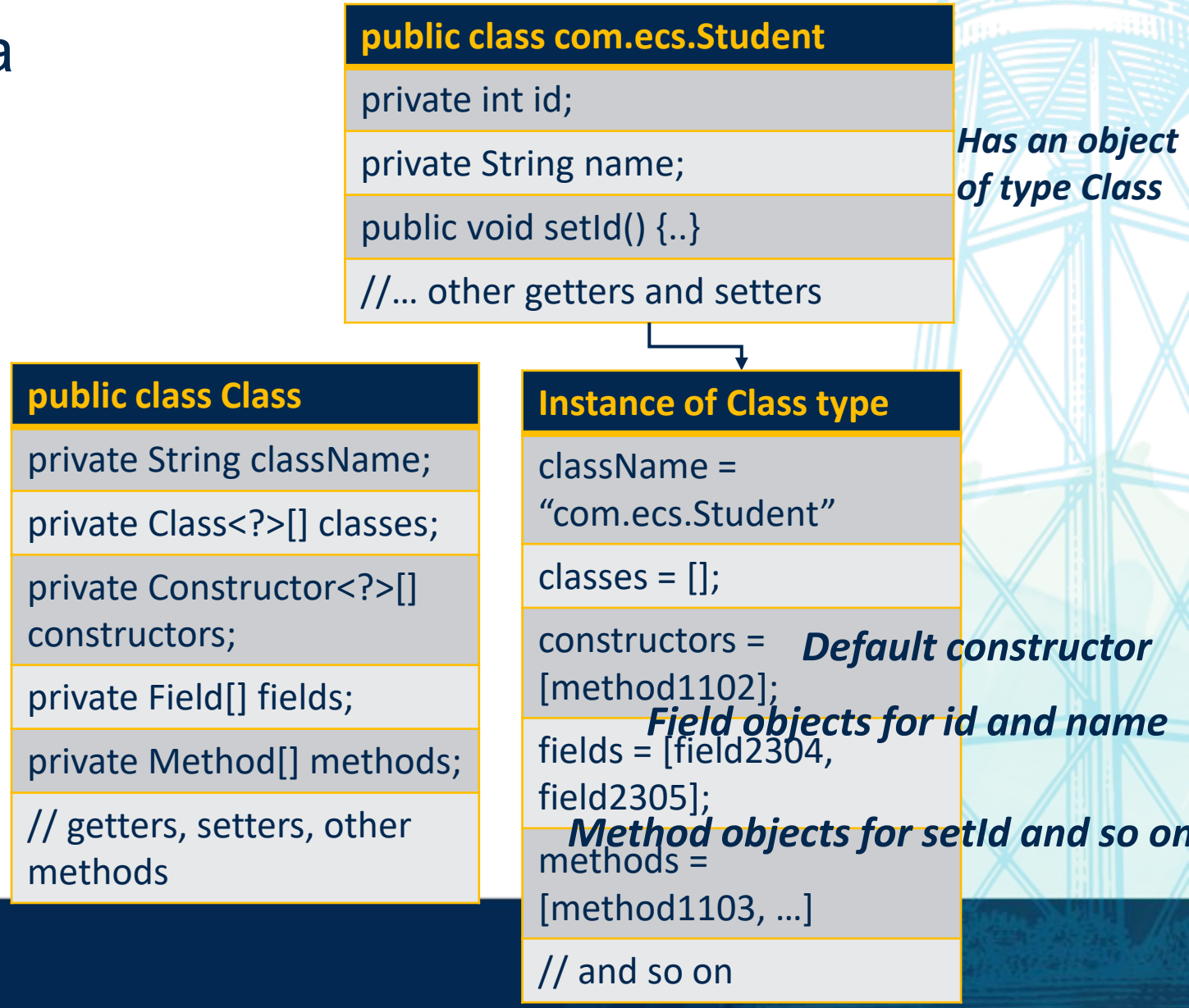
- Reflection (recap)
- Reflection demo
- Annotations (recap)
- Annotations demo

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
- Purpose
  - Access and modify classes, methods, fields, and constructors dynamically
  - Facilitate frameworks, libraries, and tools (e.g., Spring, Hibernate)

Key classes

- java.lang.Class: Represents a class or interface
- Every **class definition** in our program would have a **Class object** that Java compiler automatically creates



Reflection recap

- Java provides API to inspect class information
- Facilitates the dynamic selection of which method/field/constructor to invoke or access

Reflection recap

- `java.lang.reflect.Class`
  - An object of this type encapsulates the class information for a particular class
  - `Class clazz = Class.forName("com.ecs160.MyApp");`
  - `Class clazz = obj.getClass();`
- `java.lang.reflect.Method`
  - Invoke using `m.invoke(obj);`
- `java.lang.reflect.Field`
- Reflection can bypass access modifiers using the `setAccessible(true)` method
- Invoke constructor to create a new Object
  - `Object o = c.getConstructor().newInstance();`

Reflection and metaprogramming

- Ability of a program to inspect and manipulate its own structure and behavior at runtime **within the same language environment**
  - You can perform these operations from within Java

Why reflection?

- Goal: reuse `persistAll` logic, independent of the object type

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

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

    public void persistAll(Post post) {
        Map<String, String> postMap;
        postMap.put(“authorName”,
post.getAuthor());
        postMap.put(…, …);
        // ... All relevant fields
        jedis.hset(id++, postMap);
    }

}
```

Possible ideas

- How about an interface that other classes can implement?
  - ```
    public interface RedisPersistable {
        public persist();
    }
    public class A implements RedisPersistable { …}
    ```
    ```

Possible ideas

```
public Post implements RedisPersistable {
    private String content;
    private String createdAt;
    public persist() {
        Jedis jedis = new Jedis();
        Map map = new HashMap();
        map.put(“content”, this.content);
        map.put(“createdAt”, this.createdAt);
        jedis.hset(map);
    }
}
```

```
public Car implements RedisPersistable {
    private String model;
    private Integer year;
    public persist() {
        Jedis jedis = new Jedis();
        Map map = new HashMap();
        map.put(“model”, this.model);
        map.put(“year”, this.year);
        jedis.hset(map);
    }
}
```

Does not result in any code reuse!

Possible ideas

- How about a class?
  - ```
    public class RedisPersistable {
        public persist() {
            // …
        }
    }
    ```
  - Problem: the parent class does not have access to the child class members -> does not even work!!
    ```

Possible ideas

```
public class RedisPersistable {
    public persist() {
        // Can’t access child class members
    }
}
```

```
public Post extends RedisPersistable {
    private String content;
    private String createdAt;
    public persist() {
        Jedis jedis = new Jedis();
        Map map = new HashMap();
        map.put(“content”, this.content);
        map.put(“createdAt”, this.createdAt);
        jedis.hset(map);
    }
}
```

```
public Car extends RedisPersistable {
    private String model;
    private Integer year;
    public persist() {
        Jedis jedis = new Jedis();
        Map map = new HashMap();
        map.put(“model”, this.model);
        map.put(“year”, this.year);
        jedis.hset(map);
    }
}
```

Does not result in any code reuse!

Why reflection?

```
class Post { }

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

    public void persistAll(Post post) {
        Map<String, String> postMap;
        postMap.put(“authorName”,
post.getAuthor());
        postMap.put(…, …);
        // ... All relevant fields
        jedis.hset(id++, postMap);
    }

}
```

```
class Post { }
class Car {}

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();
           fieldVal.setAccessible(true);
           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(“BMV”);
RedisDB db = new RedisDB();
db.persistAll(p);
db.persistAll(c);
```

Reflection recap

- `java.lang.reflect.Class`
  - An object of this type encapsulates the class information for a particular class
  - `Class clazz = Class.forName("com.ecs160.MyApp");`
  - `Class clazz = obj.getClass();`
- `java.lang.reflect.Method`
  - Invoke using `m.invoke(obj);`
- `java.lang.reflect.Field`
  - Invoke using `f.getVal(obj);`
- Reflection can bypass access modifiers using the `setAccessible(true)` method
- Invoke constructor to create a new Object
  - `Object o = c.getConstructor().newInstance();`

Loading all classes

- ```
  ClassLoader cl = ClassLoader.getSystemClassLoader();
  java.lang.reflect.Field classesField =
  ClassLoader.class.getDeclaredField("classes");
  classesField.setAccessible(true);
  Vector<Class<?>> classes = (Vector<Class<?>>)
  classesField.get(cl);
  for (Class<?> c : classes) {
    System.out.println(c.getName());
  }
  ```
  ```

Reflection, annotations, proxies

Reflection recap

- `java.lang.reflect.Class`
  - An object of this type encapsulates the class information for a particular class
  - `Class clazz = Class.forName("com.ecs160.MyApp");`
  - `Class clazz = obj.getClass();`
- `java.lang.reflect.Method`
  - Invoke using `m.invoke(obj); // equivalent to calling obj.m()`
- `java.lang.reflect.Field`
  - Invoke using `f.getVal(obj); // equivalent to obj.f;`
- Reflection can bypass access modifiers using the `setAccessible(true)` method
- Invoke constructor to create a new Object
  - `Object o = c.getConstructor().newInstance();`
- Load classes using ClassLoader

Discussion: design problem

- Design a framework that invokes all methods in all classes whose names start with Test

Discussion: design problem

- Design a framework that invokes all methods in all classes whose names start with Test
- Pseudo-code
  - ```
    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);
          }
       }
    }
    ```
    ```

Design problem

- Load all classes whose name starts with `LLMService` (e.g. `LLMServiceOllama`, `LLMServiceGPT4`) These classes will contain a `runQuery(String q)` method that queries the LLM with the query `q`.
- Design a class `LLMInvoker` that has a function `invokeLLM(String q)`, that invokes the `runQuery(q)` method on all classes whose name starts with `LLMService`

Design problem

- `class Plugin { Object o; Method m; // getters setters}`

- ```
  List<Class> classes = classLoader.getClasses();
  for (Class clazz: classes) {
    if (clazz.name().startswith(“LLMService”)) {
       Object o = clazz.constructor.invoke();
       for (Method m: clazz.getDeclaredMethods()) {
           if (m.getName().startsWith(“runQuery”)) {
               plugins.push(new Plugin(o, m));
           }
       }
    }
  }
  ```

- ```
  invokeLLM(String s) {
       for (Plugin p: plugins) {
              p.getMethod().invoke(p.getObject(), s);
       }
  }
  ```

Design problem

- Auto-wiring
- Can’t express autowiring using compile time reflection

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(“BMV”);
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

MyApp.java

javac

Some annotations retained only in the compiler

```
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
}
```

MyApp.class

INTERPRETER

JVM (Java Virtual Machine)

Just-in-Time Compiler

[java]

Some annotations retained at runtime

scratch.asm

```
format PE console

push ebp
mov  ebp, esp

mov  eax, 10

mov  esp, ebp
pop  ebp
ret
```

Machine code

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
- **In class demo**

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);
             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);
```

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> postMap;
          for (Field f: obj.getDeclaredFields()) {
             if (f.isAnnotationPresent(Persistable.class) {
                f.setAccessible(true);
                String fieldname = f.getName();
                Object fieldVal = f.get(obj);
                postMap.put(fieldname, fieldVal);
             }
          }
          jedis.hset(id++, postMap);
     }

}
```

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 {
}

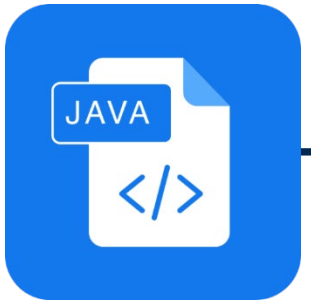
class Post {
     @Persistable
     private String content;
     private Integer tempVal;
}

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

}
```

Recall: compilation toolchain

Compile Time



MyApp.java

javac

```
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
}
```

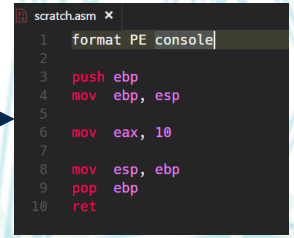
MyApp.class

Some annotations retained only in the compiler

Run Time

INTERPRETER

Just-in-Time Compiler



Machine code

[java]

Some annotations retained at runtime

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`

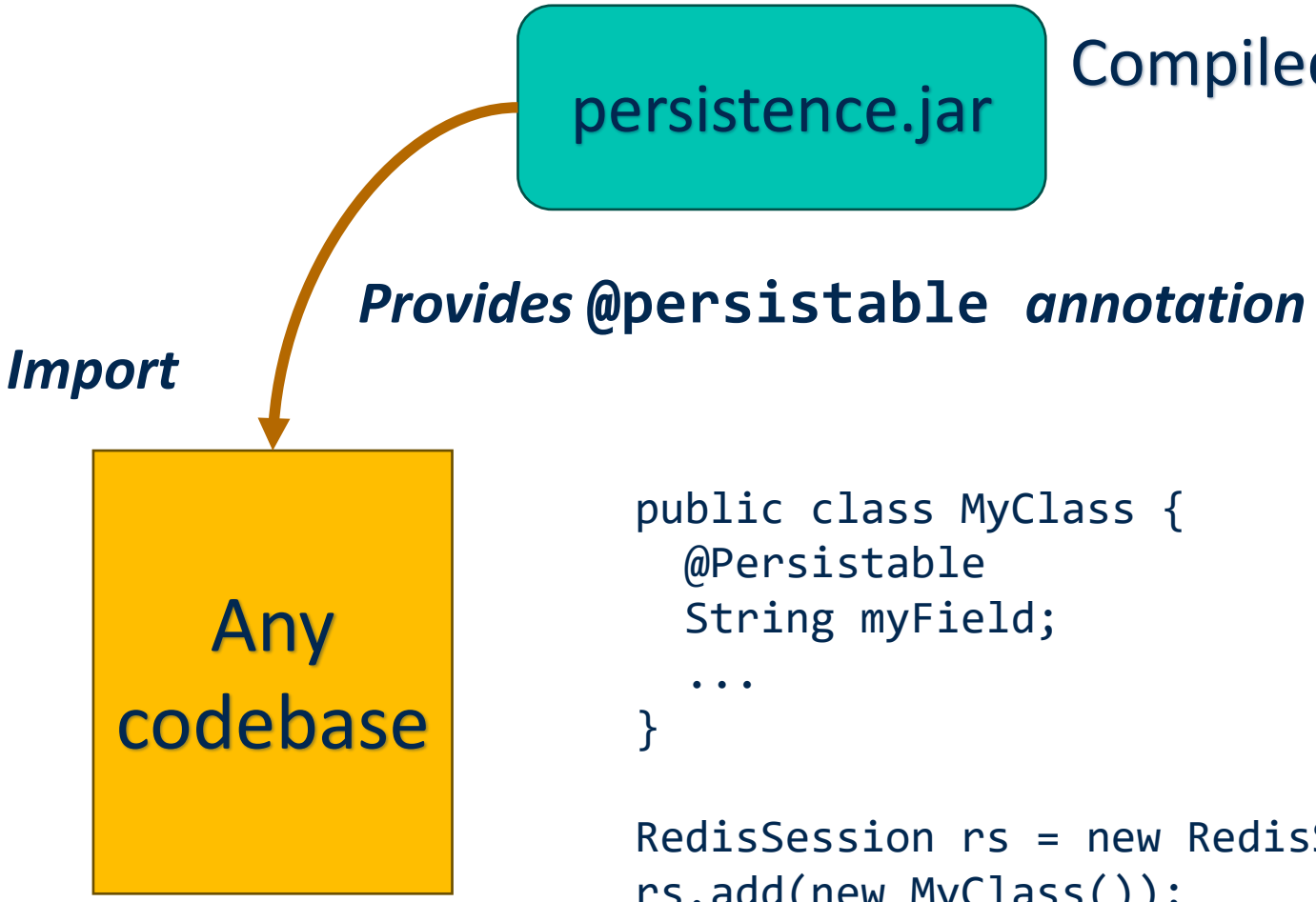
```
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnnotation {
    String author();
    String date();
    int currentRevision() default 1;
    String lastModified() default "N/A";
    String lastModifiedBy() default "N/A";
}
```

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




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

RedisSession rs = new RedisSession();
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
- **In-class demo**

HIBERNATE

```
@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
  - 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)	***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

Reflection and proxies

Agenda

- Reflection (final thoughts)
- Proxies
  - Static proxies
  - Dynamic proxies
- Proxy use-cases
  - Lazy-loading
  - Mock-testing

Reflection alternatives and optimizations

- Original persistence code
- Use reflection to generate “type-independent” persistence code

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

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

    public void persistAll(Post post) {
        Map<String, String> postMap;
        postMap.put(“authorName”,
post.getAuthor());
        postMap.put(…, …);
        // ... All relevant fields
        jedis.hset(id++, postMap);
    }

}
```

Reflection alternatives and optimizations

- Original persistence code
- Use reflection to generate “type-independent” persistence code
- What do we ***really*** want?
  - Annotate a class and the `persistAll` method for it is “magically” generated
- Can we generate code dynamically?
  - The Java language does not provide this facility

```
@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> postMap;
        for (Field f: obj.getDeclaredFields()) {
            if
(f.isAnnotationPresent(Persistable.class) {
                f.setAccessible(true);
                String fieldname = f.getName();
                Object fieldVal = f.get(obj);
                postMap.put(fieldname, fieldVal);
            }
        }
        jedis.hset(id++, postMap);
    }
}
```

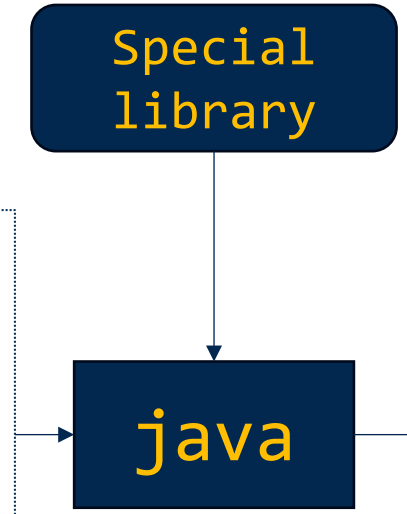
Bytecode instrumentation

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 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);
   }

}
```

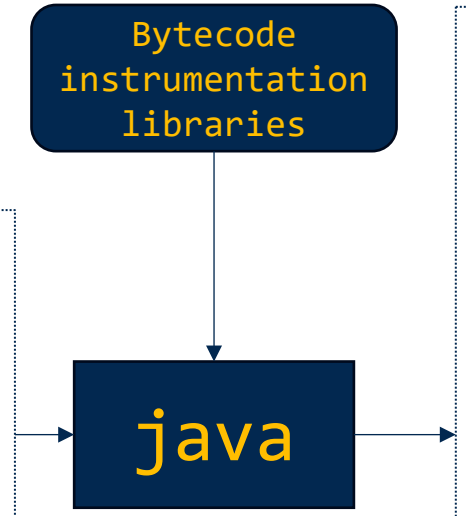
Bytecode instrumentation

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 }

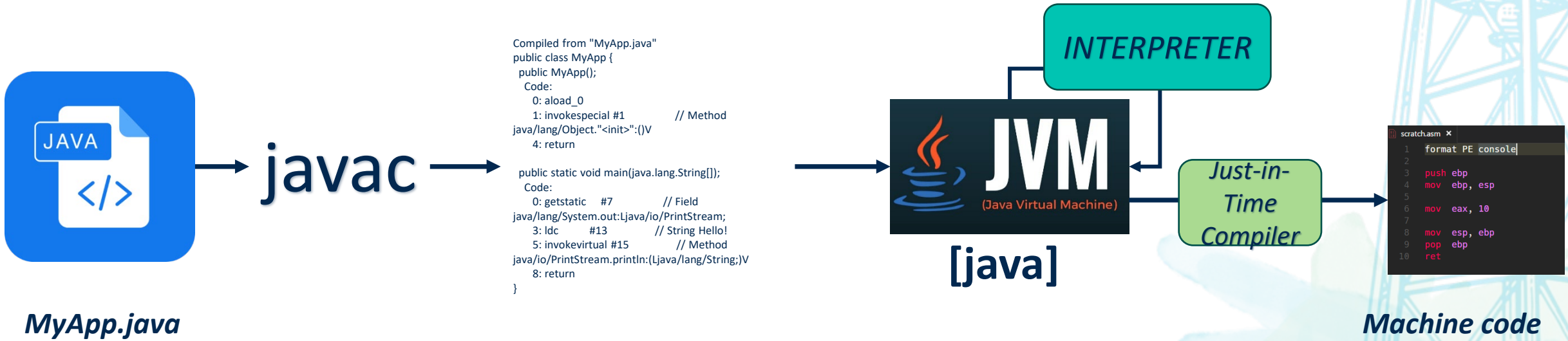
class JedisInterface {
   public void persistPost(Post p) {
      Map<> map = new HashMap();
      map.add("userId", p.getUserId());
      // …
      jedis.hset(id++, map);
   }

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

}
```

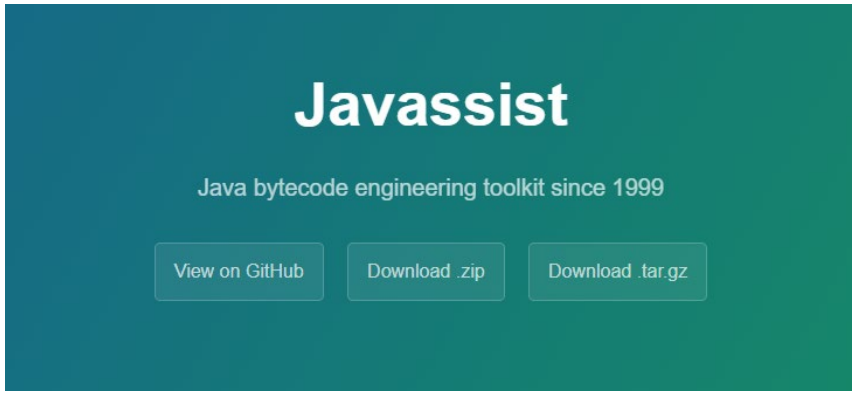
Runtime bytecode generation and manipulation

- Recap what is bytecode?



- 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

Agenda

- Proxies
  - Proxy design pattern
  - Dynamic proxies in Java
  - Use case

Background: super

- super keyword is a reference to the parent class

```
class Vehicle {
    public void accelerate() {
        S.o.p("Vehicle is accelerating");
    }
}

class Car extends Vehicle {
    public void accelerate() {
        S.o.p("Car is accelerating");
        super.accelerate();
    }
}

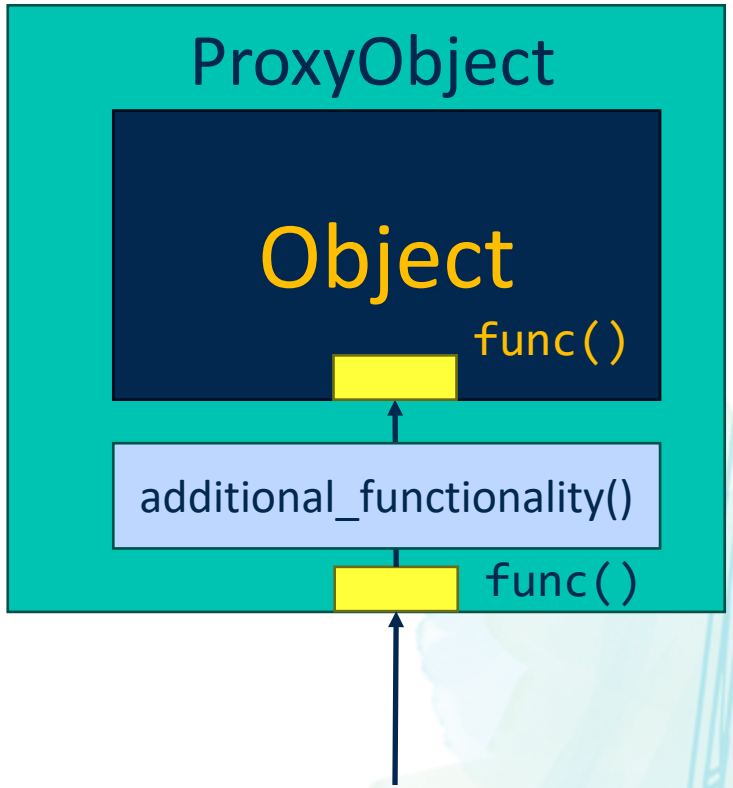
Car c = new Car();
c.accelerate(); // will print
                // Car is accelerating
                // Vehicle is accelerating
```

Design patterns – quick overview

- Classes and objects abstract object behavior
  - Focus on reusing object behavior
- Design patterns abstract object instantiation, composition, and behavior
  - Focus on abstracting the class design itself
  - Some design issues keep reoccurring
    - Allows software engineers to *reuse* previous class designs and architectures
- Design patterns used in regular applications (reflection typically used only in frameworks)

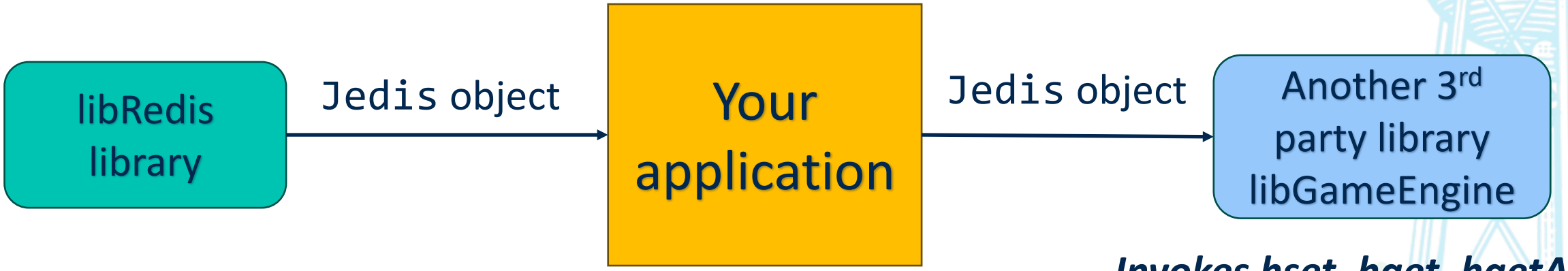
Proxy design pattern - what is a proxy object?

- 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 –



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

How to specialize a class?

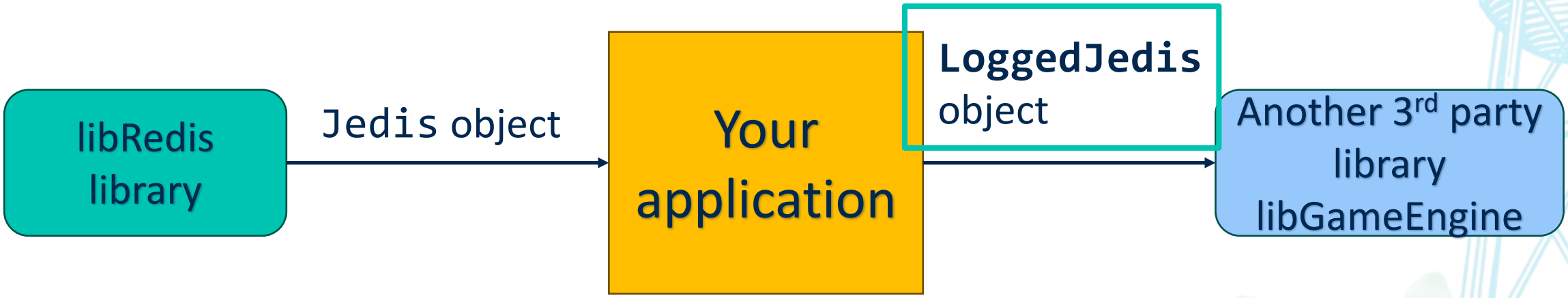
- Which OO principle to use?
- Inheritance!!

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

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

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

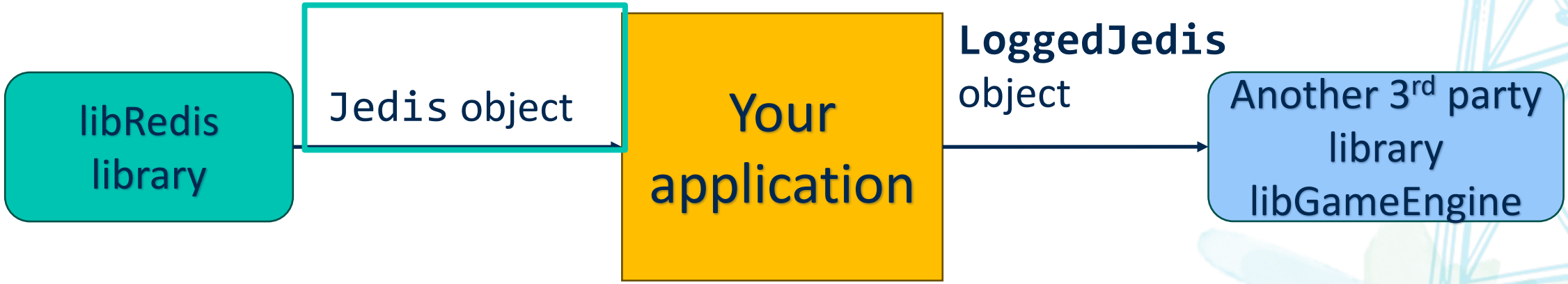
Can pass LoggedJedis object to libGameEngine



- 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

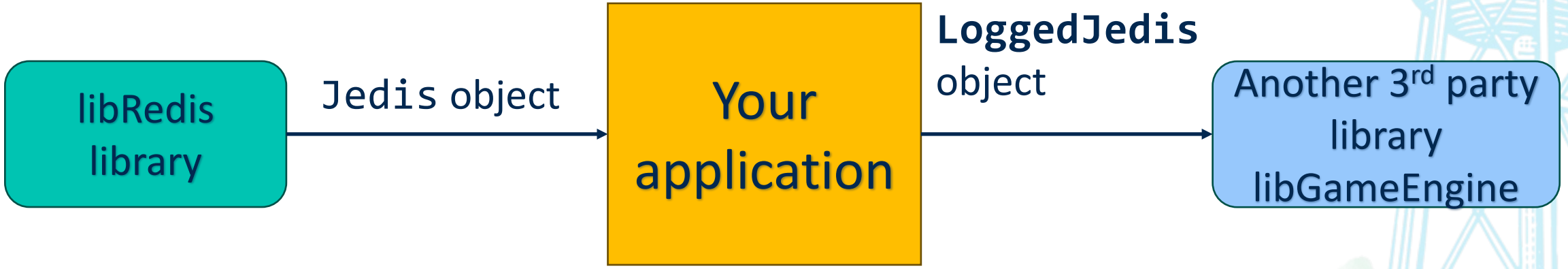
- Can’t get `libRedis` library to return a `LoggedJedis` object



How to convert Jedis object to LoggedJedis object?

But can’t modify Jedis library

- Option 1 - Manually copy the fields

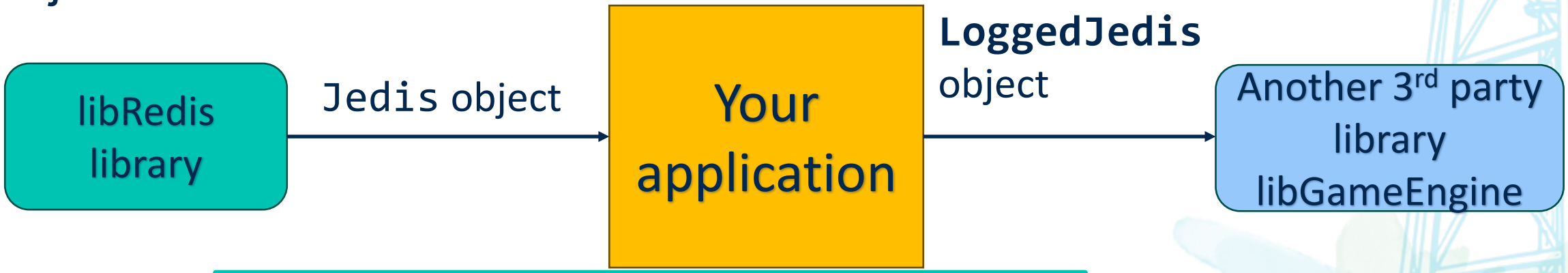



```
LoggedJedis loggedJedis = new LoggedJedis();
loggedJedis.setUrl(jedis.getUrl());
loggedJedis.setPort(jedis.getPort());
```

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

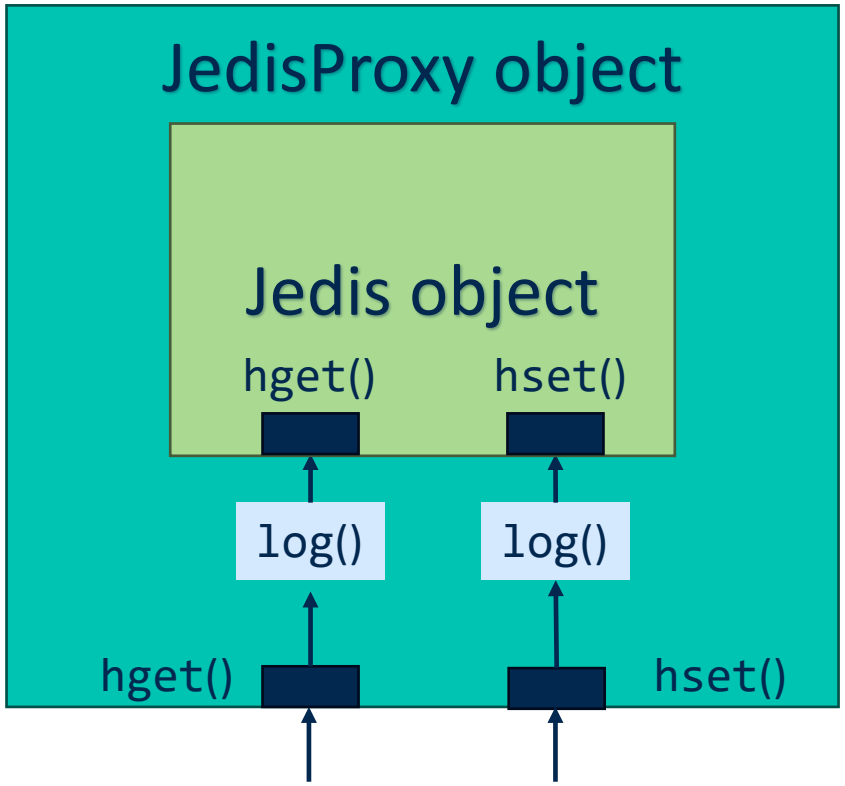
But can't modify Jedis library

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




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

Proxy object wraps and extends target class




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

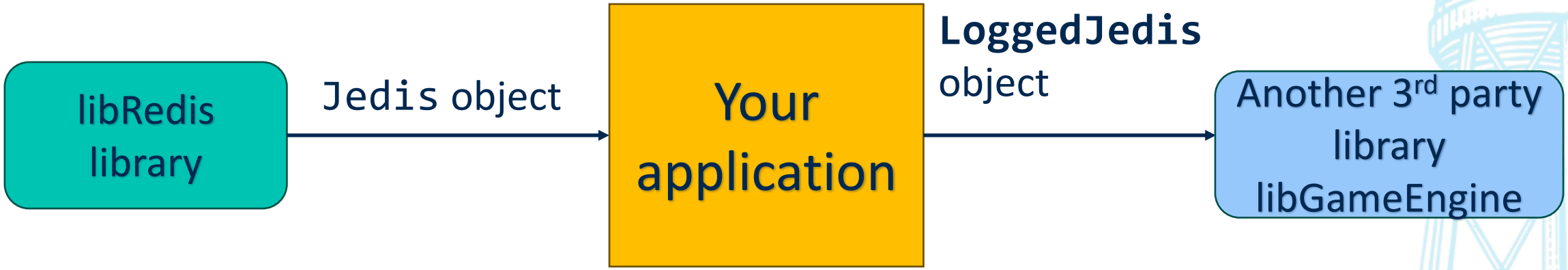
class JedisProxy extends Jedis{
    private Jedis jedis; // Wrap jedis obj

    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

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(…); }

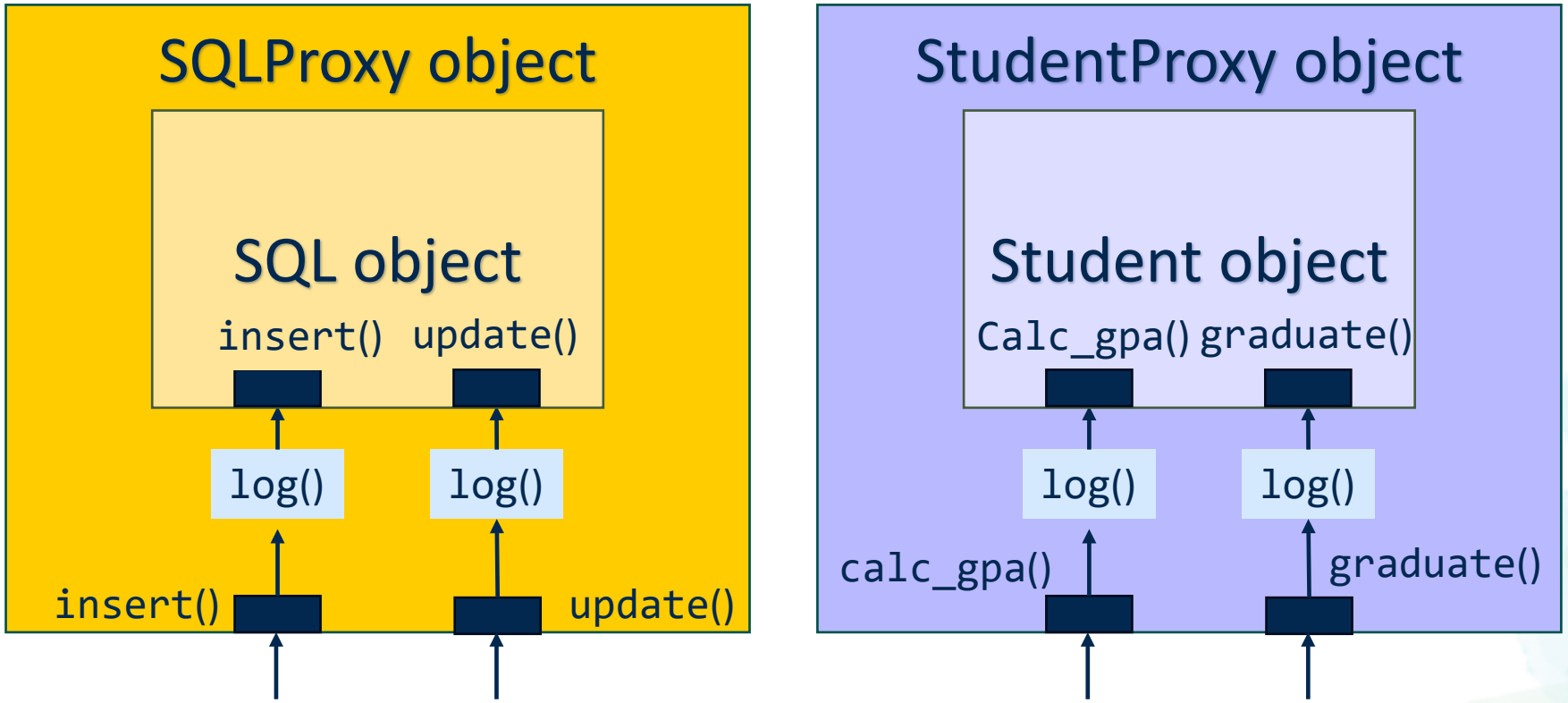
    public void executeQuery(String query) {
        log();
        query.executeQuery(query);
    }

}
```

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

Duplicated work

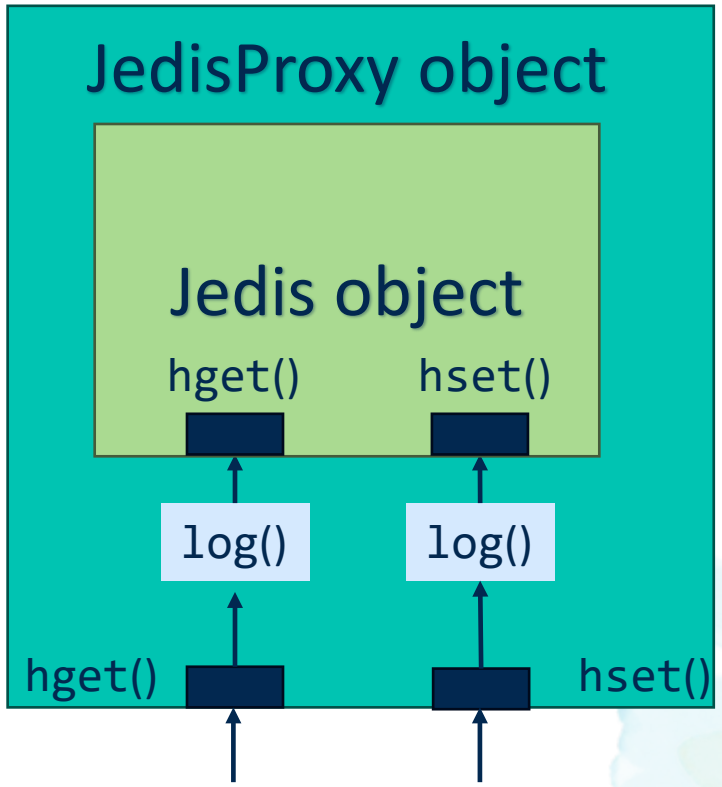
- Add logging to other classes

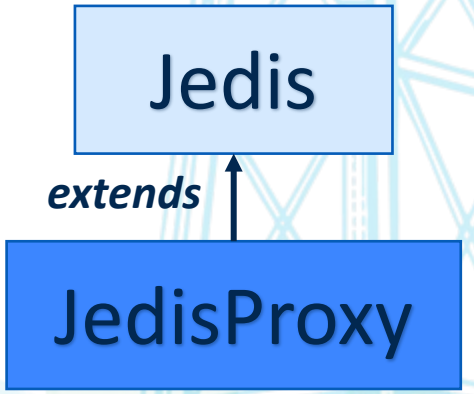


Duplicated work designing proxy classes for each class

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 `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 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);
    }
 }; …
}
```

Proxy's additional functionality

All methods of object intercepted by `invoke` ***method***

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 (contd.)

Tapti Palit

Agenda

- Dynamic proxies in Java
  - Recap
  - Demo
  - Use cases
    - Lazy loading
    - Mock testing
- Software architecture
- Software Security

Announcements

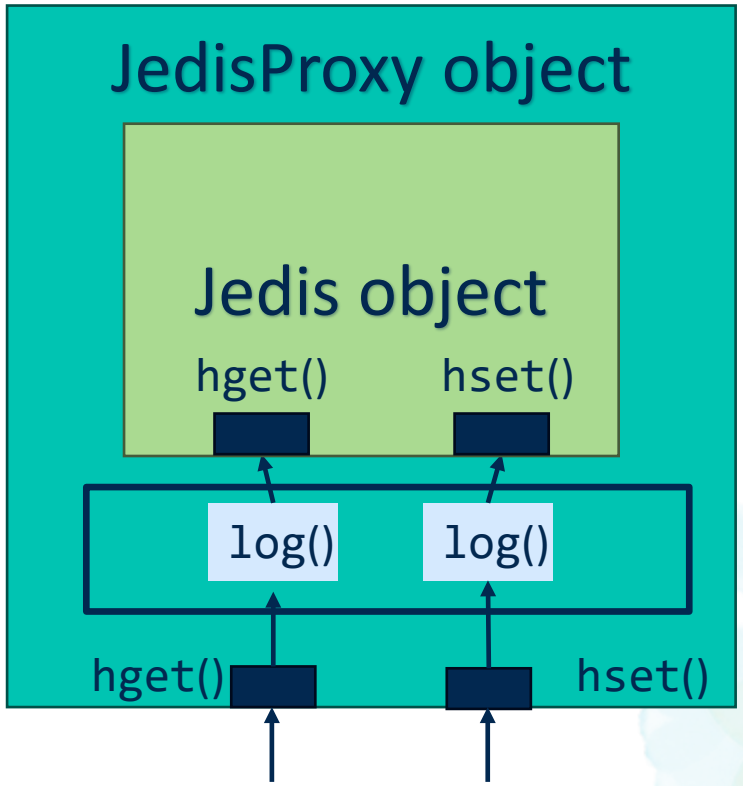
- HW1 submission
  - On Canvas – HW will be created when Canvas comes up
- Submission instructions (added to handout)
  - Make sure you don't have any private keys hardcoded in the code. You will lose points if we find that!
  - Please add a README.md file to your repository that explicitly specifies any assumptions you have made, beyond what the assignment already specifies.
    - This README.md should also explain how to set up the GITHUB API key.
  - Make sure your solution runs if we execute the command `mvn install && mvn exec:java`. (We will have Redis server running already)
  - Zip the contents of your repository. Do NOT include the repository/s you cloned as part of this homework.

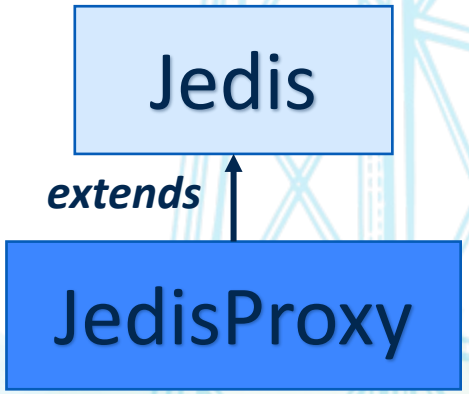
Announcements

- Emails: Add ECS 160 to subject line
  - No need for High Priority, unless really High Priority ☺
  - Only for 1-on-1 discussions
- Midterm
  - Everything including dynamic proxies + some software architecture

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 = object.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

```
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

Can handle any Type

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

Set up

Generate proxy class, create object, set additional functionality

Javassist will call invoke() for all proxy methods

Additional Functionality

Dynamic Proxy

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 = object.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

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

TargetClass t = new TargetClass();
TargetClass proxyT = (TargetClass) createProxy(t);
proxyT.doThis();
```

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

class Proxy0 extends TargetClass implements
javassist.util.proxy.Proxy {
      private MethodHandler handler;

      Method getSuperMethod(String name) {// use reflection }
      Method getMethod(String name) {// use reflection }

      public void doThis() {
            handler.invoke(this, getSuperMethod("doThis"),
getMethod("doThis"), []);
      }

      public void doThat() {
            handler.invoke(this, getSuperMethod("doThat"),
getMethod("doThat"), []);
      }
}
```

Javassist runtime-generated class

Demo

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

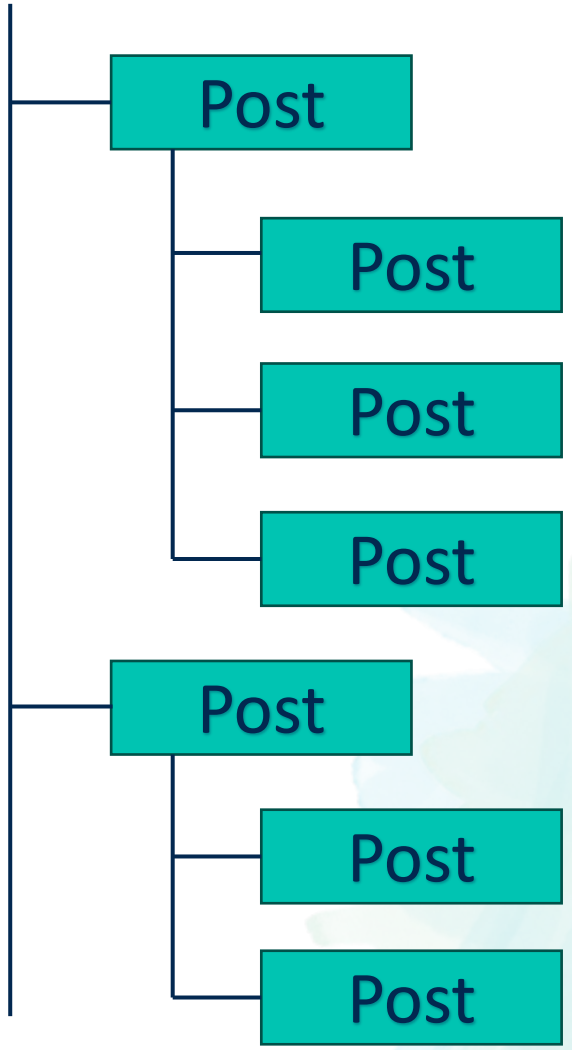
Real world use cases of proxies

- Sample use cases
  - Lazy loading
  - Mock testing

EASYMOCK

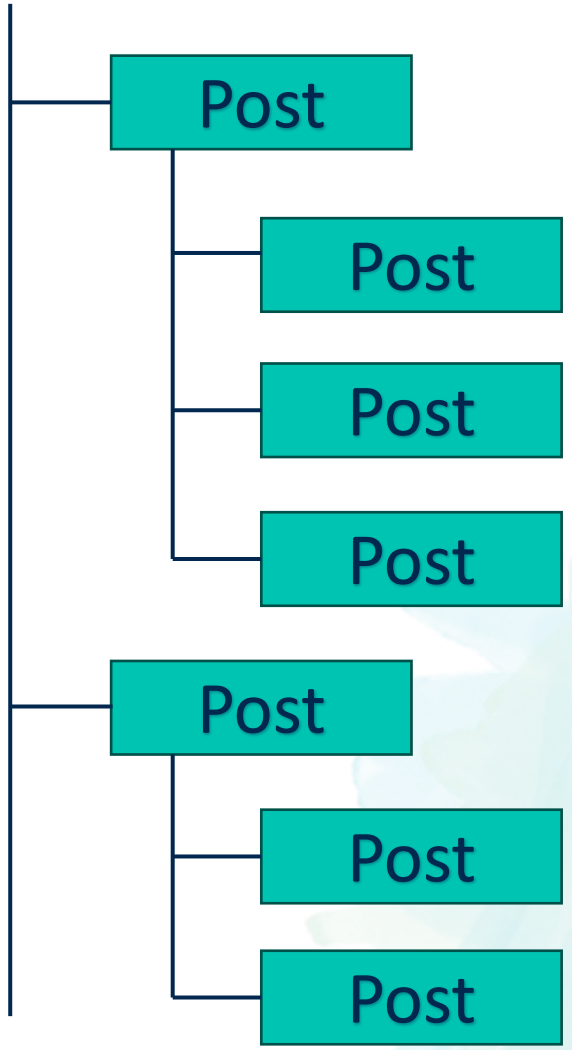
Loading posts

- 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(“,”);

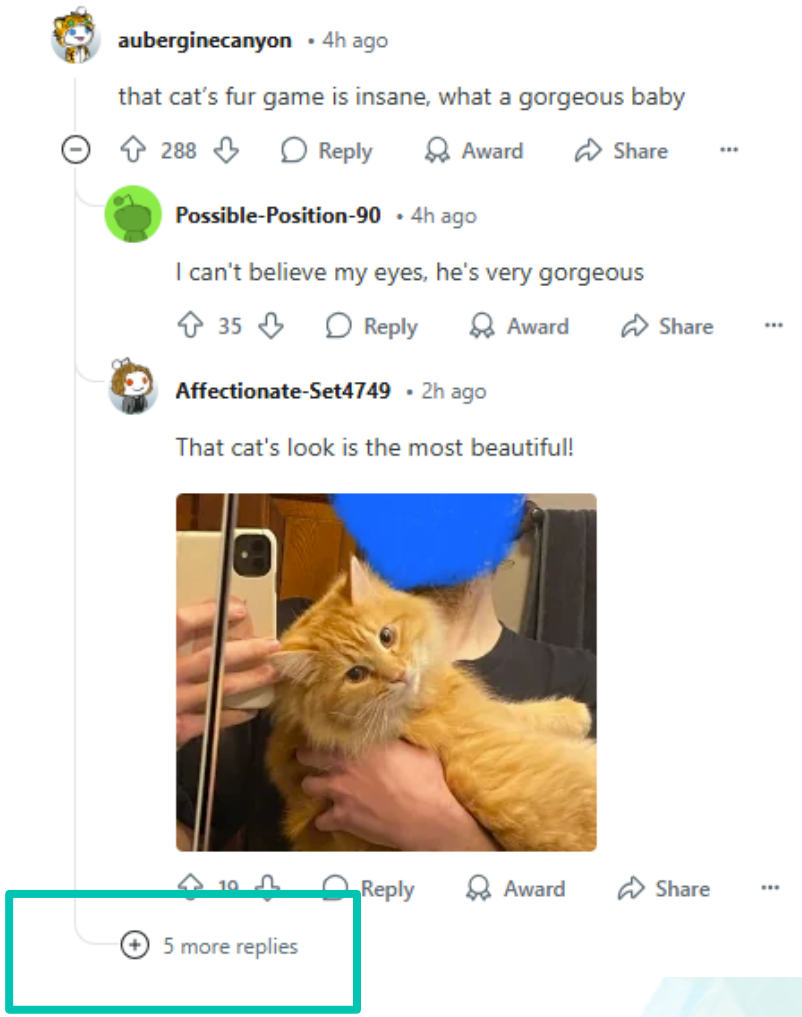
for (String replyId: replies) {
    replyMap = jedis.hgetAll(replyId);
    Post reply = new Post();
    reply.setId(replyId);
    reply.setCreatedAt(replyMap.get(“createdAt”);
    post.getReplies().add(reply);
}
```

```
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

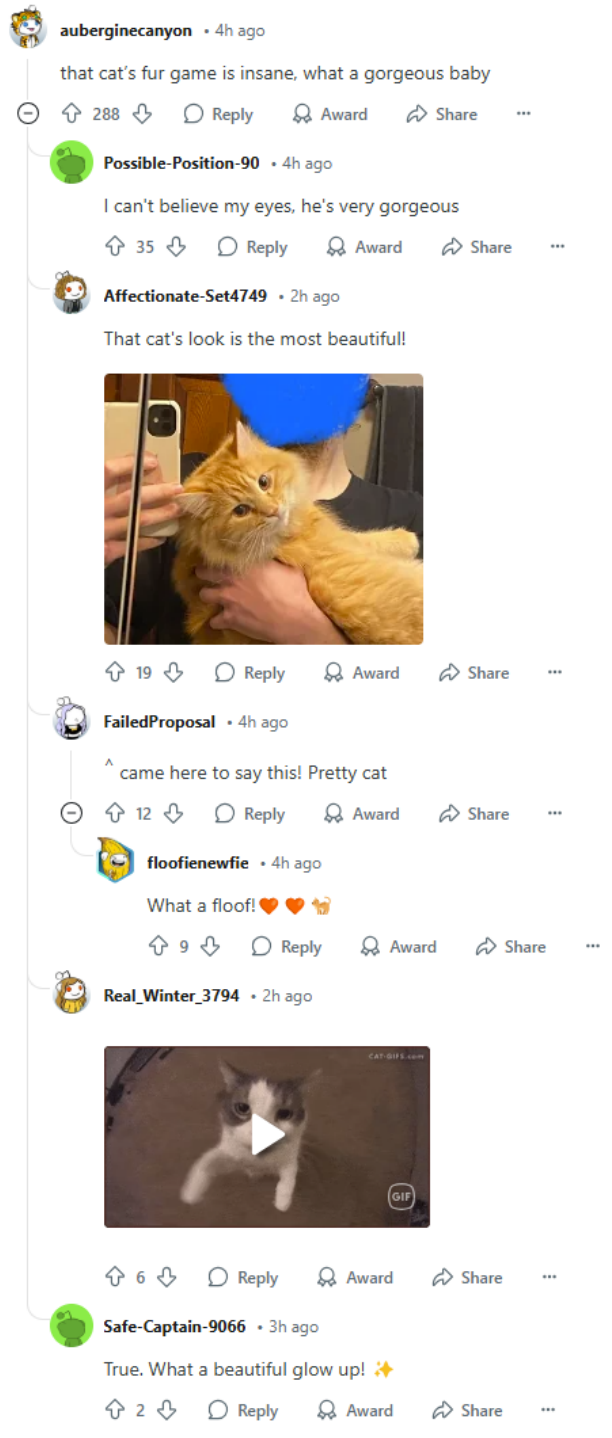
Lazy loading posts UI

- 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



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 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

Agenda

- Lazy loading with dynamic proxies
- Mock testing
- Reflection, annotations, dynamic proxies – final thoughts

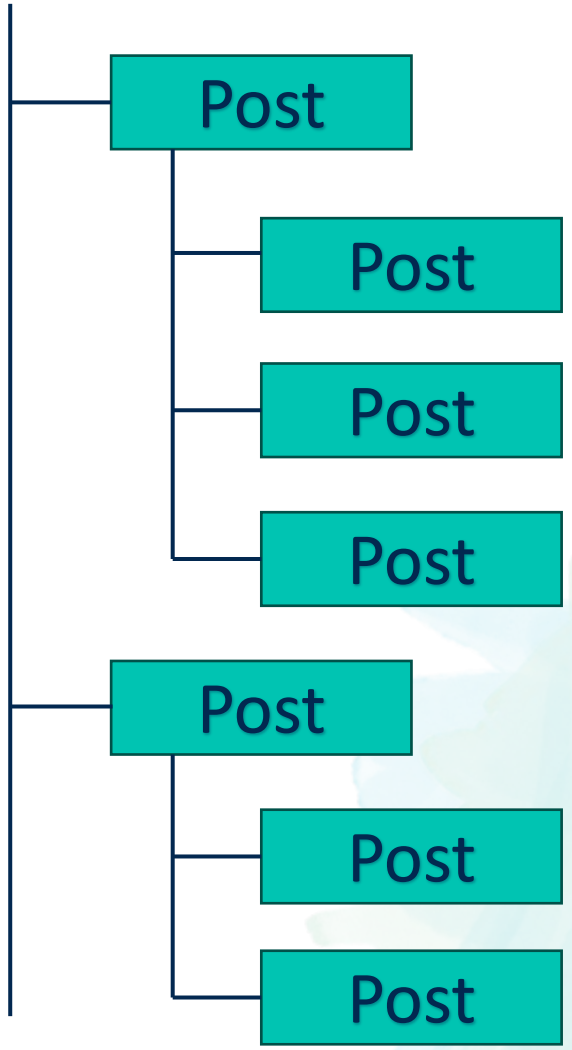
- Software architecture

Announcements

- Homework due tomorrow 11:59 PM
- Canvas HW will be created soon
- Quiz on Friday
  - 10 minutes
  - Will cover reflective programming
    - Reflection, annotations, and proxies
      - Not include lazy loading, mock testing
  - Will have MCQs and short answers

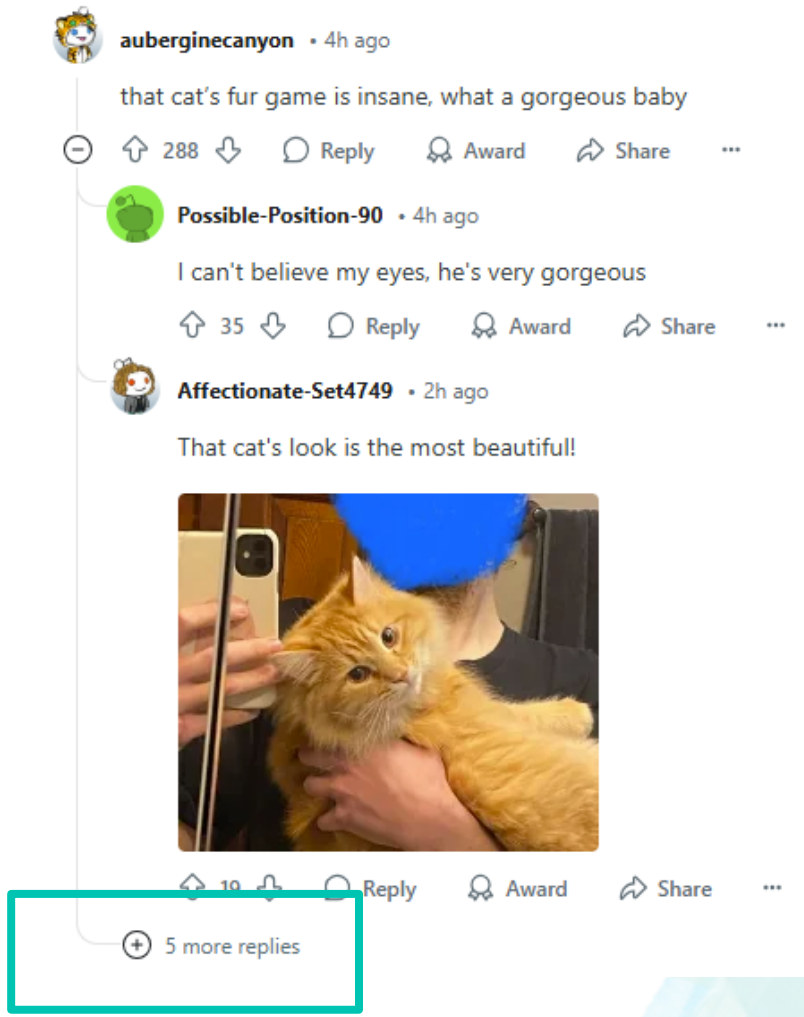
Loading posts

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



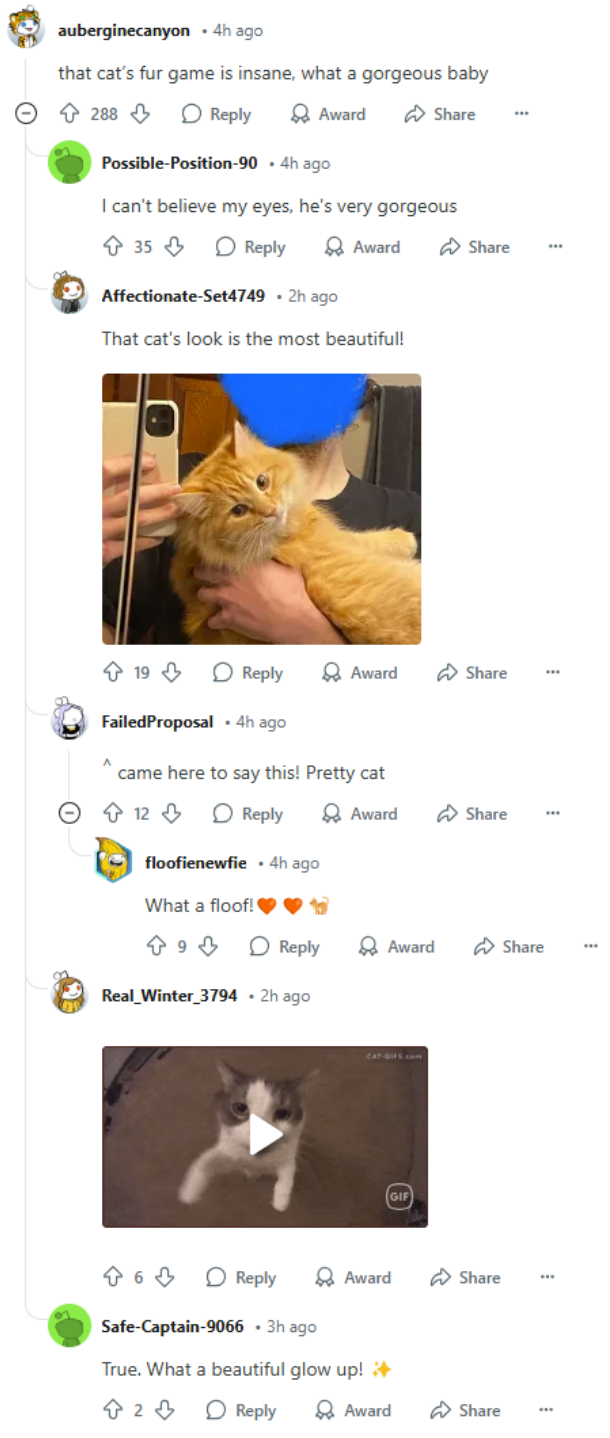
Lazy loading posts UI

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



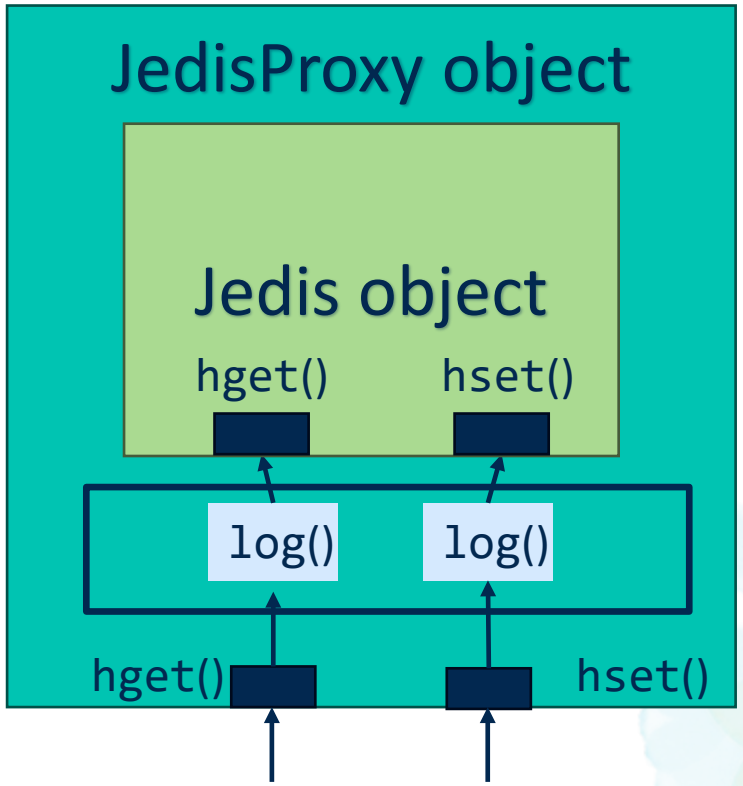
Lazy loading

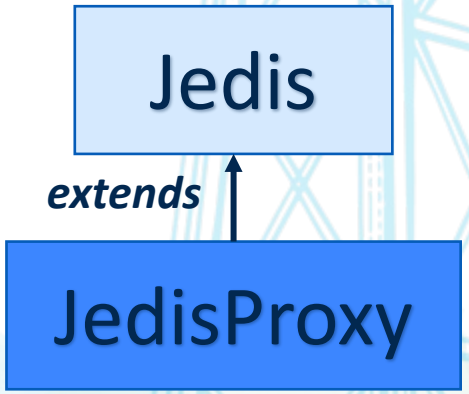
- Clicking on ‘+’ loads the remaining replies
- Dynamic proxies simplify implementing lazy loading
- This class: *sketch of a solution (not full solution)*



Proxies: recap

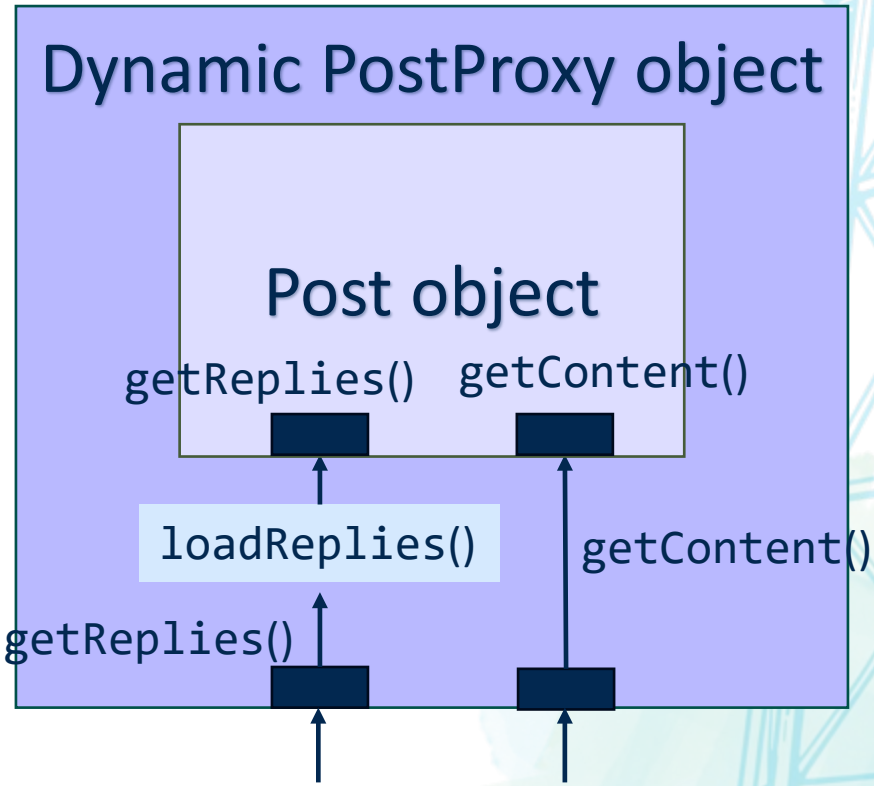
- A proxy is a wrapper around a target object
- The proxy intercepts method invocations
  - Performs additional functionality
  - Dispatches the method to the target object
- Proxy class extends target class
  - Allows proxy to replace the target object for all uses





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



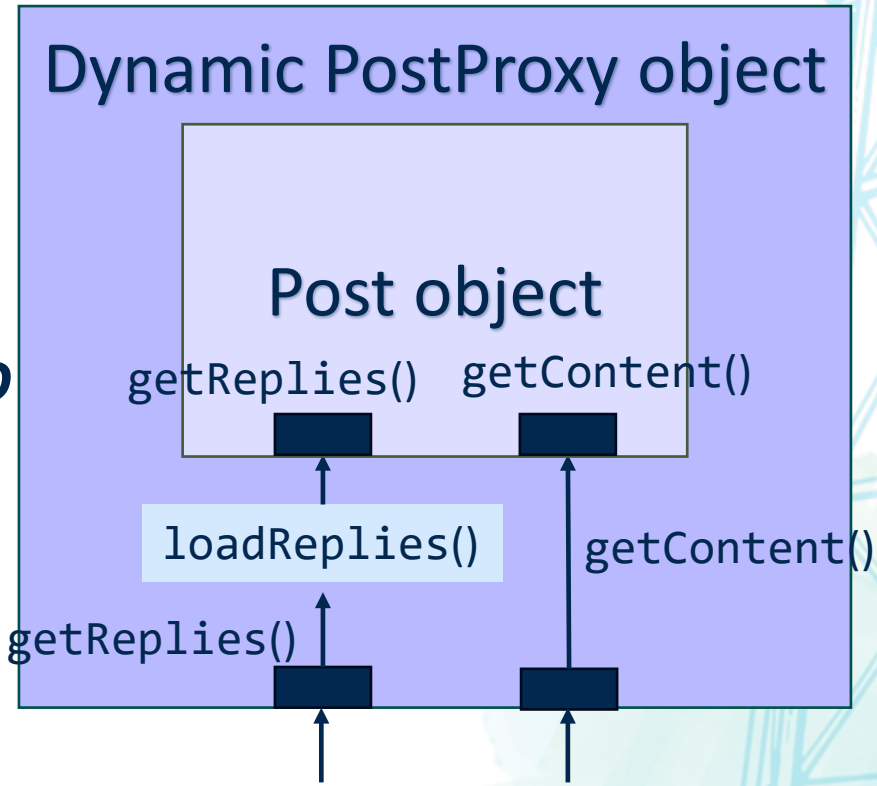
Proxies for lazy loading

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

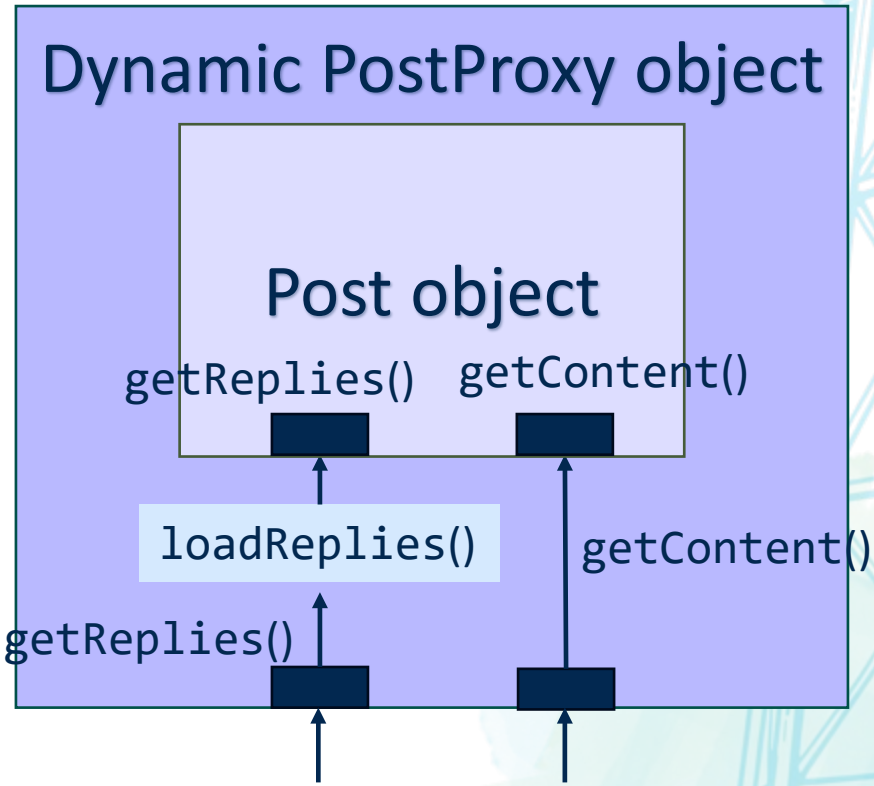
Only set the Reply ID



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

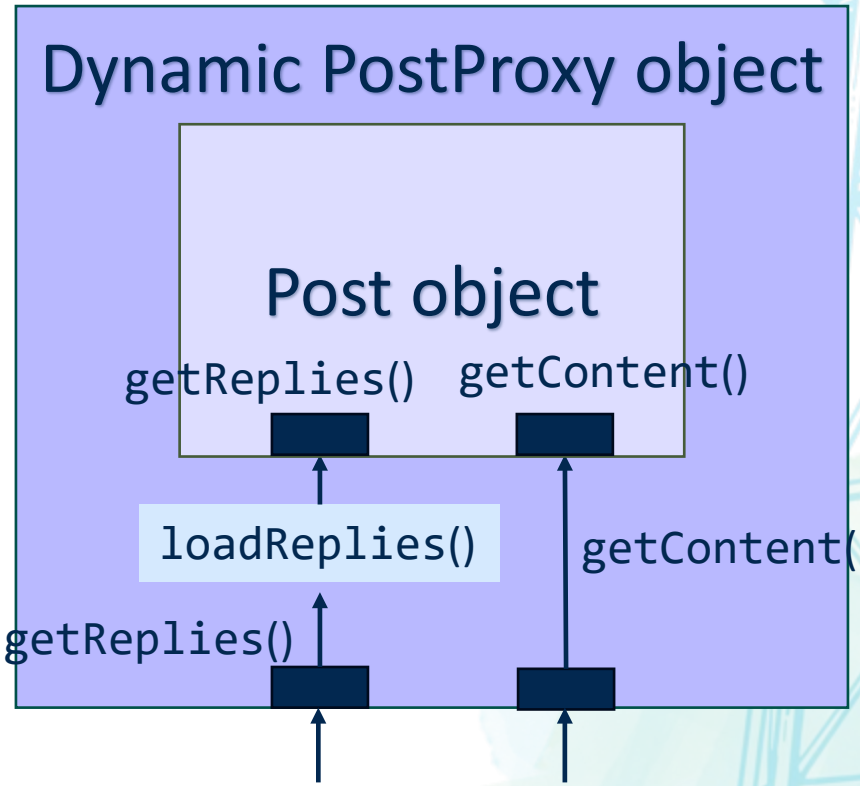
Proxies for lazy loading

```
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 {
          // Code to invoke loadReplies();
          // if getReplies() is invoked
          return proceed.invoke(target, args);
     }
}
```

Invoke loadReplies()

createLazyLoadProxy returns the proxy object

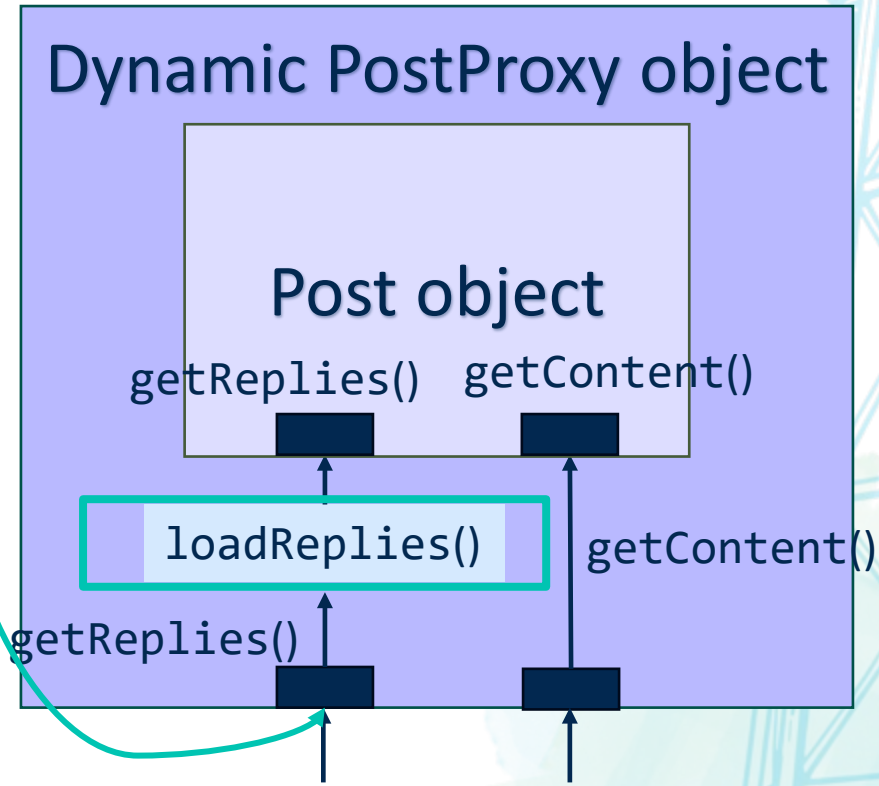
Proxies for lazy loading

```
class UIButton {
    Post post;
    void onClick() {
        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("+"));
}
```

post is a 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
}
```

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;
    }
}
```

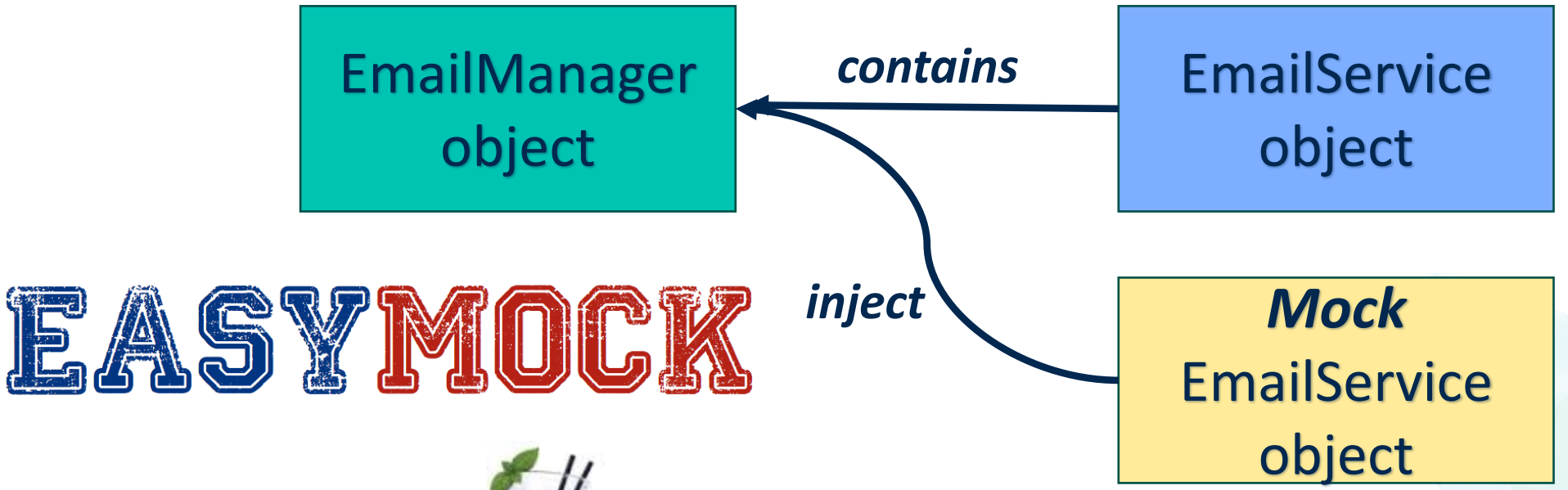
```
public class EmailManager {
    private EmailService emailService;

    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



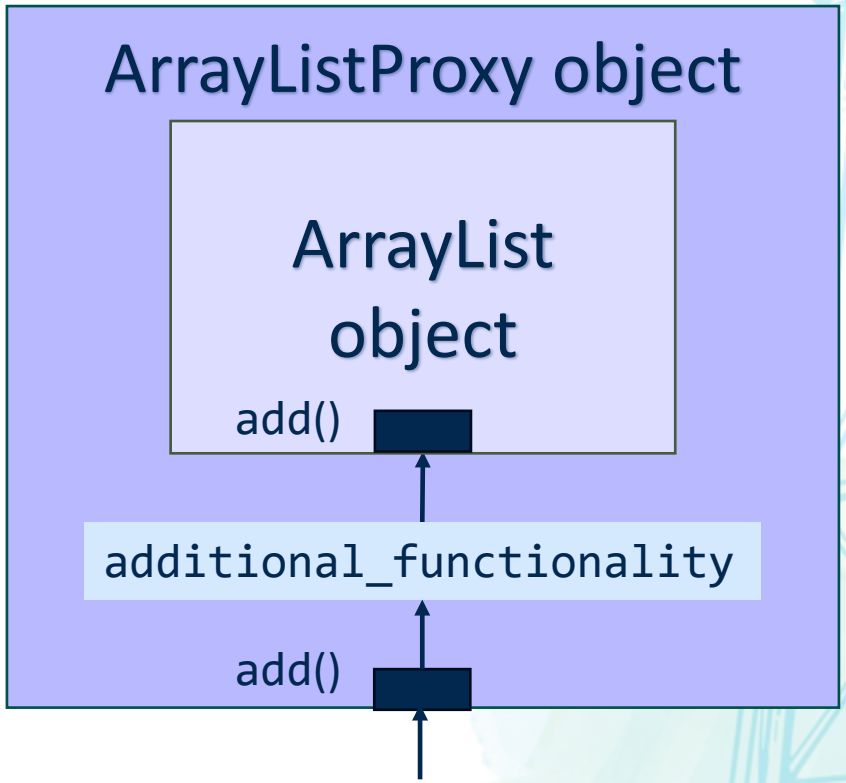
EASYMOCK

mockito

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

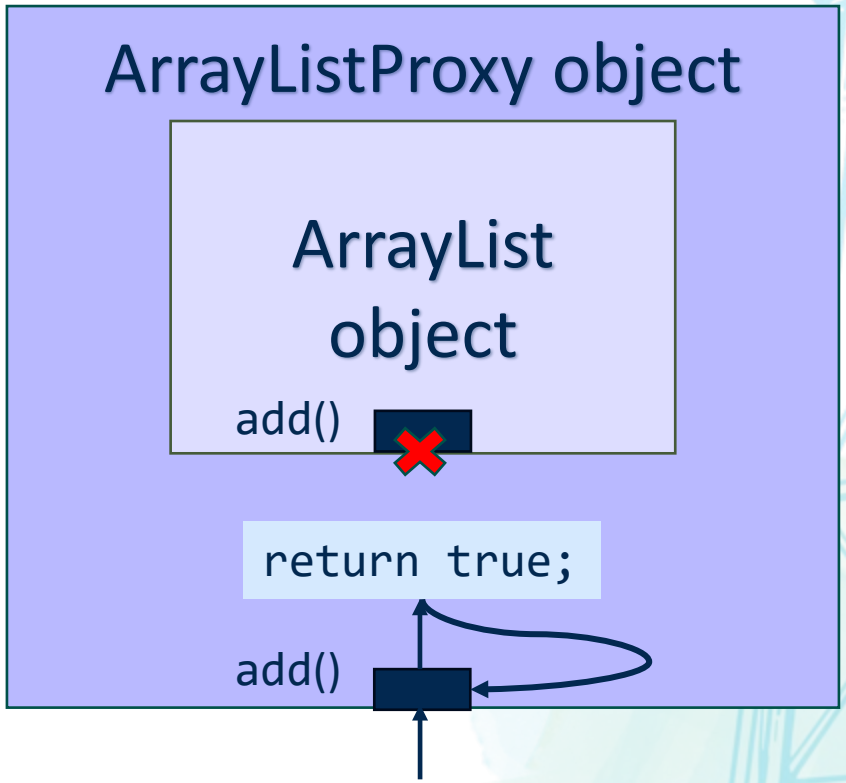
Proxy can hijack functionality

- 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



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(“BMV”);
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;

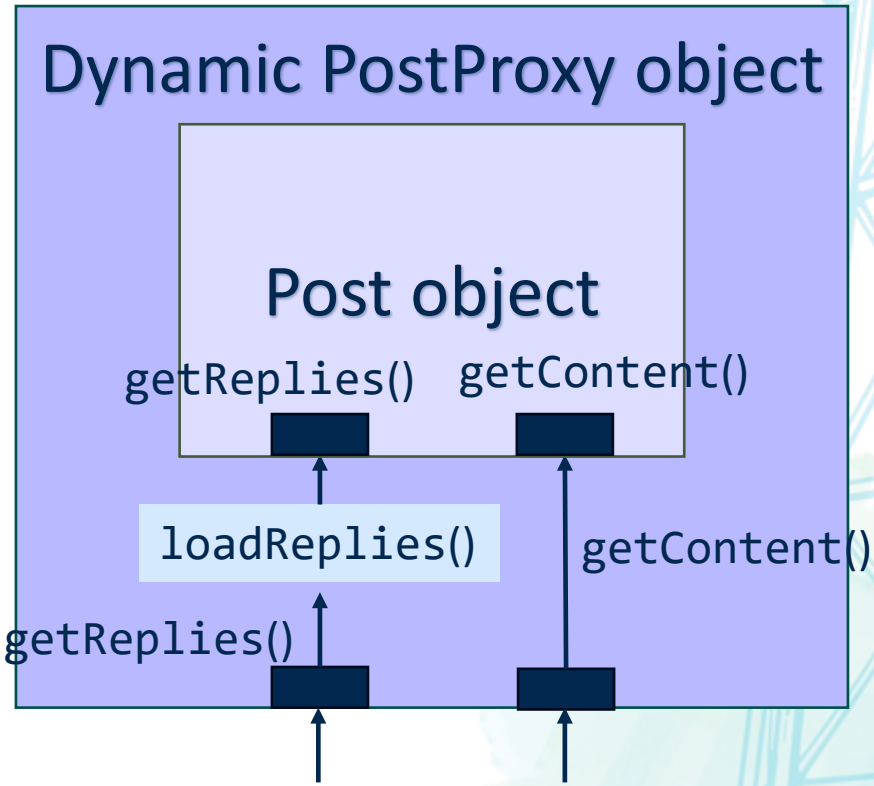
    @Persistable
    String authorName;

    @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