Inf1B
Object Design

Perdita Stevens
adapting earlier versions by Ewan Klein, Volker Seeker, et al.

School of Informatics
Cohesion describes how well a unit of code maps to a logical task or entity. A good OO design aims for high cohesion.

Coupling describes the interconnectedness of classes. A good OO design aims for loose coupling.
In this code, the method does two things at the same time. What if the coordinate format would change?
Cohesion Example

private static final int FOX_WIN_ROW = 0;

private static int getRowCoord(String boardCoords) {
    String rowCoord = boardCoords.substring(1);
    int row = Integer.parseInt(rowCoord) - 1;
    return row;
}

public static boolean isFoxWin(String foxPos) {
    // error handling code omitted ...

    int foxRow = getRowCoord(foxPos);

    boolean isWin = foxRow == FOX_WIN_ROW;
    return isWin;
}

This code is now more cohesive and coordinate format changes would only have to be addressed in one place.
Cohesion Example

Code duplication can be a sign of poor cohesion.

Demo
public static boolean isFoxWin(String foxPos) {
    if (!isBoardCoordinate(pos)) {
        throw new IllegalArgumentException("Given position must" + " be a valid board coordinate but is: " + pos);
    }

    int foxRow = getRowCoord(foxPos);

    boolean isWin = foxRow == FOX_WIN_ROW;
    return isWin;
}

Responsibility Driven Design and Encapsulation help to loosen coupling within code.
Coupling Example

```java
public static boolean isFoxWin(BoardCoordinate foxPos) {
    Objects.requireNonNull(foxPos, "...");

    boolean isWin = foxPos.getRow() == FOX_WIN_ROW;
    return isWin;
}
```

The BoardCoordinate class guarantees valid data and takes responsibility for coordinate translation. Also, it can be changed internally without affecting the game logic.
public class BoardCoordinate {
    private final int row;
    private final int column;

    public BoardCoordinate(int row, int column) {
        if (row < 0 || column < 0) {
            throw new IllegalArgumentException("Invalid ...");
        }
        this.row = row;
        this.column = column;
    }

    public int getRow() { return row; }
    public int getColumn() { return column; }

    @Override
    public String toString() {
        String rowRepr = "" + (row + 1);
        String columnRepr = "" + (char)(‘A’ + column);
        return columnRepr + rowRepr;
    }
}
Enums
Enumerated Types

A type whose legal values consist of a fixed set of constants.

For example, types of figures in a game:

```java
public static final String HOUND_FIELD = "H";
public static final String FOX_FIELD = "F";
```
Enumerated Types

A type whose legal values consist of a fixed set of constants.

For example, types of figures in a game:

code

```java
public static final String HOUND_FIELD = "H";
public static final String FOX_FIELD = "F";
```

or fruit ...

code

```java
public static final int APPLE_FUJI = 0;
public static final int APPLE_PIPPIN = 1;
public static final int APPLE_GRANNY_SITH = 2;

public static final int ORANGE_NAVAL = 0;
public static final int ORANGE_TEMPLE = 1;
public static final int ORANGE_BLOOD = 2;
```
Enumerated Types

A type whose legal values consist of a fixed set of constants.

For example, types of figures in a game:

```java
public static final String HOUND_FIELD = "H";
pUBLIC STATIC final String FOX_FIELD = "F";
```

or fruit ...

```java
public static final int APPLE_FUJI = 0;
pUBLIC STATIC final int APPLE_PIPPIN = 1;
pUBLIC STATIC final int APPLE_GRANNY_SITH = 2;

public static final int ORANGE_NAVEL = 0;
pUBLIC STATIC final int ORANGE_TEMPLE = 1;
pUBLIC STATIC final int ORANGE_BLOOD = 2;
```

Those are known as int enum pattern or String enum pattern.
Enumerated Types

```java
class Fruit {
    public static final int APPLE_FUJI = 0;
    public static final int APPLE_PIPPIN = 1;
    public static final int APPLE_GRANNY_SITH = 2;
}

class Citrus {
    public static final int ORANGE_NAVEL = 0;
    public static final int ORANGE_TEMPLE = 1;
    public static final int ORANGE_BLOOD = 2;
}
```

This type of pattern has many shortcomings:

▶ no type safety
Enumerated Types

```java
public static final int APPLE_FUJI = 0;
public static final int APPLE_PIPPIN = 1;
public static final int APPLE_GRANNY_SITH = 2;

public static final int ORANGE_NAVEL = 0;
public static final int ORANGE_TEMPLE = 1;
public static final int ORANGE_BLOOD = 2;
```

This type of pattern has many shortcomings:

- no type safety
- little expressive power
Enumerated Types

public static final int APPLE_FUJI = 0;
public static final int APPLE_PIPPIN = 1;
public static final int APPLE_GRANNY_SITH = 2;
public static final int ORANGE_NAVEL = 0;
public static final int ORANGE_TEMPLE = 1;
public static final int ORANGE_BLOOD = 2;

This type of pattern has many shortcomings:

▶ no type safety
▶ little expressive power
▶ no distinct name spaces
Enumerated Types

public static final int APPLE_FUJI = 0;
public static final int APPLE_PIPPIN = 1;
public static final int APPLE_GRANNY_SITH = 2;

public static final int ORANGE_NAVEL = 0;
public static final int ORANGE_TEMPLE = 1;
public static final int ORANGE_BLOOD = 2;

This type of pattern has many shortcomings:

- no type safety
- little expressive power
- no distinct name spaces
- no easy way to iterate over all items
- no easy way to translate into int enum constants printable strings
- string enum constants can cause performance problems due to string comparison
Enumerated Types

```
public static final int APPLE_FUJI = 0;
public static final int APPLE_PIPPIN = 1;
public static final int APPLE_GRANNY_SITH = 2;

public static final int ORANGE_NAVEL = 0;
public static final int ORANGE_TEMPLE = 1;
public static final int ORANGE_BLOOD = 2;
```

This type of pattern has many shortcomings:

- no type safety
- little expressive power
- no distinct name spaces
- no easy way to iterate over all items
- no easy way to translate into int enum constants printable strings
Enumerated Types

public static final int APPLE_FUJI = 0;
public static final int APPLE_PIPPIN = 1;
public static final int APPLE_GRANNY_SITH = 2;

public static final int ORANGE_NAVEL = 0;
public static final int ORANGE_TEMPLE = 1;
public static final int ORANGE_BLOOD = 2;

This type of pattern has many shortcomings:

- no type safety
- little expressive power
- no distinct name spaces
- no easy way to iterate over all items
- no easy way to translate into int enum constants printable strings
- string enum constants can cause performance problems due to string comparisson
Enums

Luckily, Java offers a way to overcome all of those with the enum type:

```java
public enum Figure { FOX, HOUND }
public enum Apple { FUJI, PIPPIN, GRANNY_SMITH }
public enum Orange { NAVEL, TEMPLE, BLOOD }
```
Luckily, Java offers a way to overcome all of those with the `enum` type:

```java
public enum Figure FOX, HOUND
public enum Apple FUJI, PIPPIN, GRANNY_SMITH
public enum Orange NAVEL, TEMPLE, BLOOD
```

In Java, enums are full-fledged classes that export one instance for each enumeration constant via a public static final field.
private static Figure swapPlayers(Figure currentTurn) {
    if (currentTurn == Figure.FOX) {
        return Figure.HOUND;
    } else {
        return Figure.FOX;
    }
}

All of the following would cause a compiler error.

Figure nextToMove = swapPlayers(Orange.TEMPLE);
Figure nextToMove = swapPlayers(Figure.CAT);
Figure nextToMove = swapPlayers(0);
 Enums are efficient

```java
private static Figure swapPlayers(Figure currentTurn) {
    if (currentTurn == Figure.FOX) {
        return Figure.HOUND;
    } else {
        return Figure.FOX;
    }
}
```

Comparison is fast because only references need to be compared.
 Enums can be iterated

for (Apple apple : Apple.values()) {
    // do what you want
}

Each enum class automatically comes with a `values` method which returns a collection of all available enum instances.
Enums can be printed

```java
for (Apple apple : Apple.values()) {
    System.out.println(apple.name());
    // or just use toString()
}
```

**Output**

FUJI
PIPPIN
GRANNY_SMITH

Each enum class provides a named string for all of its instances.
Enums can be parsed

String userInput = "BLOOD";
Orange myFruit = Orange.valueOf(userInput);

The **valueOf** method allows parsing string values to corresponding enum types. But beware, illegal strings will cause an exception.
Since enums are full-fledged classes, much more is possible than mentioned above:

▶ specify methods
▶ associate data with each constant
▶ ...

Advanced Enum Programming
Comparing Objects
Java rules for comparrison

For Primitives use $==$

For Object References use $==$  

For Object States use $equals$ (if it is implemented)
Custom Types in HashMaps

You can also put your own data types into a **HashMap**:

```java
HashMap<String, Circle> data = new HashMap<String, Circle>();
data.put("Small", new Circle(2));
data.put("Large", new Circle(200));
```

Using custom types as keys, is more tricky: You will have to make sure they have an `equals` method and produce the same hash code.
How do you properly implement equals and hashCode?

Demo
Design Patterns
Towards Software Engineering

First learn your basic tools and material.

Source: https://i.kinja-img.com/gawker-media/image/upload/s–Zo3E8URT–/c_scale,f_auto,fl_progressive,q_80,w_800/18muwoa3oozw6jpg.jpg
Towards Software Engineering

First learn your basic tools and material.
Then build large houses ...

Source: http://hannesdorfmann.com/images/legohouse/legohouse.jpg
Towards Software Engineering

First learn your basic tools and material.
Then build large houses ...or even cities.

Source: https://i.kinja-img.com/gawker-media/image/upload/s–uTscbBDV–/c_scale,f_auto,fl_progressive,q_80,w_800/tu3yxy86lxwmw5vw8yeu.jpg
Design Patterns

Software Design Patterns are blueprints of solutions for common software design problems.

Source: https://www.rff.com/cloverleaf.png
Classification

- Creational Patterns
- Structural Patterns
- Behavioural Patterns
Creational Example: Singleton

Problem

- access a resource in your program
Creational Example: Singleton

Problem

- access a resource in your program → database resource
Problem
▶ access a resource in your program → database resource
▶ initialising resource access is expensive
Creational Example: Singleton

Problem

- access a resource in your program → database resource
- initialising resource access is expensive → only one instance
Creational Example: Singleton

Problem

- access a resource in your program → database resource
- initialising resource access is expensive → only one instance
- multiple classes need access
Creational Example: Singleton

Problem

- access a resource in your program → database resource
- initialising resource access is expensive → only one instance
- multiple classes need access → globally available
public class Database {
    private final DBConnection connection;

    public Database() {
        connection = new DBConnection("myuser", "myhost", "mydatabase");
        connection.connect();
    }

    public List<String> query(String q) {
    }
}
public class Database {
    private final DBConnection connection;

    public Database() {
        connection = new DBConnection("myuser",
                          "myhost", "mydatabase");
        connection.connect();
    }

    public List<String> query(String q) {
    }
}

Globally available instance not guaranteed!
public class Database {
    private static Database dbase;
    private final DBConnection connection;

    public Database() {
        connection = new DBConnection("myuser", "myhost", "mydatabase");
        connection.connect();
    }

    public List<String> query(String q) { ... }
}

Add private static field for storing the singleton instance.
public class Database {
    private static Database dbase;
    private final DBConnection connection;

    public Database() {
        connection = new DBConnection("myuser",
                                      "myhost", "mydatabase");
        connection.connect();
    }

    public static Database getInstance() {
        // ?
    }

    public List<String> query(String q) {
        // ...
    }
}

Declare public static creation method to access the singleton instance.
Creational Example: Singleton Solution!

```java
class Database {
    private static Database dbase;
    private final DBConnection connection;

    public Database() {
        connection = new DBConnection("myuser",
                                   "myhost", "mydatabase");
        connection.connect();
    }

    public static Database getInstance() {
        if (dbase == null) dbase = new Database();
        return dbase;
    }

    public List<String> query(String q) {
    }
}
```

Lazily create the instance of the singleton if necessary and return it.
public class Database {
    private static Database dbase;
    private final DBConnection connection;

    private Database() {
        connection = new DBConnection("myuser", 
                                     "myhost", "mydatabase");
        connection.connect();
    }

    public static Database getInstance() {
        if(dbase == null) dbase = new Database();
        return dbase;
    }

    public List<String> query(String q) { ... }
}

Make the singleton constructor private.
Creational Example: Singleton Solution!

```java
public static void main(String[] args) {
    Database db = Database.getInstance();
    db.query(args[0]);
}
```

In a client, use the getInstance method to access the singleton.
Structural Example: Facade

Problem

- you need to integrate a complex library into your own codebase
- many interdependencies between your own code and the third party code
Problem

- you need to integrate a complex library into your own codebase
- many interdependencies between your own code and the third party code

What if a new version of this library is suddenly broken?
What if you find a better library?
Structural Example: Facade

Solution

Use a facade class which provides a simple interface to the library code.

Source: https://refactoring.guru/design-patterns/facade
Behavioural Example: Observer

Problem

How to best communicate events between classes?

Source: https://refactoring.guru/design-patterns/observer
Behavioural Example: Observer

Solution

Source: https://refactoring.guru/design-patterns/observer
A large catalog of common design patterns exists:

https://refactoring.guru/design-patterns/catalog
Reading

Books

- **Objects First** *Chapter 8*
- **Effective Java** by Joshua Bloch
- **Design Patterns: Elements of Reusable Object-Oriented Software** by Erich Gamma, Ralph Johnson, John Vlissides, Richard Helm

Web Resources

- [https://refactoring.guru/design-patterns](https://refactoring.guru/design-patterns)
Questions?

Give Feedback and Support this course:

▶ EUSA Teaching Awards
▶ Course Feedback via Learn