Lecture 7
`switch`, `enum` and jshell

Remainder of LP2

- Friday: final lecture 13-15
  - No exercises this week
- Exam preparation
  - Send me your requests

    (or else I'll force you to sit through more pony examples)
- Monday: exam time!
  - There will be **no** graphical programming on the exam
- Exercises next week: final exercise session
  - We will go through Monday's exam

Mid-course evaluation

- Summary available on course website
- Generally very positive!
- But let’s have a closer look at how the course will improve for LP3!

Exercises

- **Problem**: exercise sessions are unstructured.
- **Problem**: book exercises too easy compared to labs/exam.
- **Action**: a larger/harder exercise will be handed out as homework for the exercise session.
  - This exercise is **not mandatory**.

Lectures

- **Problem**: lectures are too fast-paced for students with no prior exposure to programming.
- **Action**: quiz checkpoints during lectures.
- If you don’t understand, **interrupt me**!
  - If you didn’t understand something, there are at least another ten students who also didn’t!

Lectures

- **Problem**: it’s hard to write everything down during lectures.
- **Action**: please don’t!
  - Slides + code are available from the course website
  - Slides are designed to work as study material on their own
  - If you want to take notes, focus on key points!

`final` and constructors

- `final` fields may be assigned either where they are declared or in the object's constructor

```
public class HangWord {
    private final String word = "blah";

    public HangWord() {
    }
}
```

vs

```
public class HangWord {
    private final String word;

    public HangWord(String word) {
        this.word = word;
    }
}
```

However, in this case you should probably just make the field `static` as well, since it will have the same value in every object.

Assigning `final` fields in the constructor is generally a lot more useful.

Recap: references

- Strings and other classes are *reference types*
- Variables of reference types store references to their actual content
- Many reference variables can point to the same object:

```
Pony ponyA = new Pony("Bruce Dickinson");
Pony ponyB = ponyA;
ponyB.train("heavy metal");
```

- `ponyA` and `ponyB` point to the same object, so *both* will learn the “heavy metal” skill!

Recap: mutability

- Modifying an object or variable = *mutating* it
- Variables are mutable by default…
- ...but become immutable if declared `final`
- Mutating a variable and mutating an object are not the same thing!
- Prefer *immutable* types when possible
- Primitive types are immutable
- An immutable class:
  - Has only `final` fields, and
  - Has only fields of other immutable types

Recap: class variables

- Just like methods, variables can also be `static`
- Such variables are called “class variables”
- Belong to *the class itself*, not objects of the class
- Usually `final` (at least when `public`)

Recap: Make Hangman Great Again!

(Using the things we learned last week)

Important exam notice

Nothing new from this point forward will appear on the first exam!

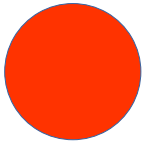
(Material which also appeared in previous lectures may of course be on the exam.)

jshell

- A “Read-Eval-Print-Loop” (REPL for short) for Java
- Run snippets of Java code without a main method
- Useful for testing your classes as you write them

Watch out for “Hadouken Code”!

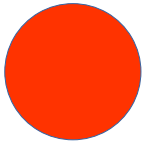
```
public static void registerUser(User user) {
    if(user.getName() != null) {
        if(user.getName() != "") {
            if(user.getPassword() != null) {
                if(user.getPassword().length() > 5) {
                    if(user.getConfirmPassword() != null) {
                        if(user.getConfirmPassword() == user.getPassword()) {
                            if(user.getName() != null) {
                                if(isValidName(user.getName())) {
                                    if(user.getEmail() != null) {
                                        if(isValidEmail(user.getEmail())) {
                                            if(user.getConfirmEmail() != null) {
                                                if(user.getEmail() == user.getConfirmEmail()) {
                                                    createUser(user);
                                                    setResponse(200, "User registration successful!");
                                                    return;
                                                }
                                            }
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
    setResponse(500, "Registration failed!");
}
```



`switch` statements

- Selects between multiple branches
- Equivalent to `if … else if … else if … else`

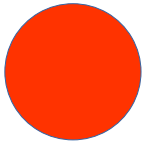
```
switch(expression) {
case 0:
    // do something if expression == 0
    break;
case 1:
    // do something else if expression == 1
    break;
…
default:
    // we end up here if expression was not equal to any case
}
```



`switch` statements

- `case` branches will “fall through” if `break` is omitted
- Sometimes handy, often dangerous!

```
switch(expression) {
case 0:
case 1:
case 29:
    // do something else if expression == 0, 1 or 29
    break;
case 2:
    // do something else if expression == 2
    break;
…
default:
    // we end up here if expression was not equal to any case
}
```

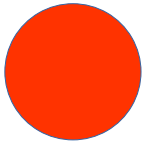


`enum` types

- Declares a new type which can have one of a finite amount of values
- Often declared inside a class that uses it, but can be declared in its own file
- ```
  public class Fika {
      public enum Pastry { CAKE, CINNAMON_ROLL, LUSSEKATT }
      public enum Drink { COFFEE, TEA, MILK }

      public static void serve(Pastry eat, Drink drink) {
          ...
      }
  }
  ```
- Can be used like any other type

```
public class FikaMain {
    public static void main(String[] args) {
        Fika.Pastry toEat = Fika.Pastry.CAKE;
        Fika.Drink toDrink = Fika.Drink.MILK;
        Fika.serve(toEat, toDrink);
    }
}
```
```



`enum` types

- Declares a new type which can have one of a finite amount of values
- Often declared inside a class that uses it, but can be declared in its own file

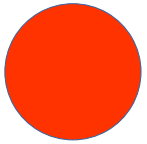
```
public class Fika {
    public enum Pastry { CAKE, CINNAMON_ROLL, LUSSEKATT }
    public enum Drink { COFFEE, TEA, MIL

    public static void serve(Pastry ea
        ...
    }
}
```

A variable of type `Fika.Drink` can only be either `Fika.Drink.COFFEE`, `Fika.Drink.TEA`, or `Fika.Drink.MILK`.

- Can be used like any other type

```
public class FikaMain {
    public static void main(String[] args)
        Fika.Pastry toEat = Fika.Pastry.CAKE;
        Fika.Drink toDrink = Fika.Drink.MILK;
        Fika.serve(toEat, toDrink);
    }
}
```



`enum` types

- Declares a new type which can have one of a finite amount of values
- Often declared inside a class that uses it, but can be declared in its own file

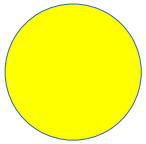
```
public class Fika {
    public enum Pastry { CAKE, CINNAMON_ROLL, LUSSEKATT }
    public enum Drink { COFFEE, TEA, MILK }

    public static void serve(Pastry eat, Drink drink) {
        ...
    }
}
```

- Can be used like any other ty

This method is in the same class as `Pastry` and `Drink`, so we can shorten `Fika.Pastry` to just `Pastry`.

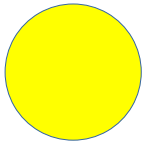
```
public class FikaMain {
    public static void main(String[] args) {
        Fika.Pastry toEat = Fika.Pastry.CAKE;
        Fika.Drink toDrink = Fika.Drink.MILK;
        Fika.serve(toEat, toDrink);
    }
}
```



`enum` goes well with `switch`

```
public static void serve(Pastry eat, Drink drink)
    switch(eat) {
    case CAKE:
        System.out.println("You have some cake.");
        System.out.println("It's so delicious and moist.");
        break;
    case CINNAMON_ROLL:
        System.out.println("You have a cinnamon roll.");
        System.out.println("It's tasty.");
        break;
    case LUSSEKATT:
        System.out.println("You're overcome with Christmas spirit.");
        break;
    default:
        throw new IllegalArgumentException("unsupported value for"+eat);
    }
    ...
}
```

- Cover all possible values of the type
- Use a `default` branch to throw an appropriate exception
- Easy to forget updating when adding new values to `enum`!



`enum` goes well with `switch`

```
public static void serve(Pastry eat, Drink drink)
    switch(eat) {
    case CAKE:
        System.out.println("You have some cake.");
        System.out.println("                    ");
        break;
    case CINNAMON_RO
        System.out.pr
        System.out.pr
        break;
    case LUSSEKATT:
        System.out.println("You're overcome with Christmas spirit.");
        break;
    default:
        throw new IllegalArgumentException("unsupported value for"+eat);
    }
    ...
}
```

No need to say `Pastry.CAKE` here, Java knows that `CAKE` is a `Pastry` because `eat` (i.e. the expression we're `switch`ing on) has type `Pastry`.

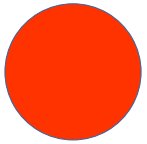
- Cover all possible values of the type
- Use a `default` branch to throw an appropriate exception
- Easy to forget updating when adding new values to `enum`!

Example: Pony Types

`null`

- Called “my billion dollar mistake” by its inventor
- Has been around since 1965(!)
- Now considered a “code smell”
- Avoid using it whenever possible
- We’ll see more safe/modern alternatives later

Tony Hoare,
Inventor of `null`

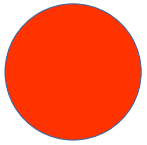


Optional<T>

- A “box” which may or may not contain a value
- `T` is the type of the thing in the box
- Important methods
  - `T get()` returns the value inside the box, or throws a `NoSuchElementException` if the box is empty
  - `boolean isEmpty()` returns `true` if the box is empty, otherwise `false`
  - `static Optional<T> empty()` creates an empty box
  - `static Optional<T> of(T value)` creates a box which contains `value`

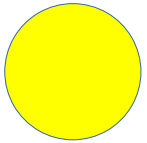
```
Optional<String> maybeName = Optional.empty();
Optional<String> maybeName2 = Optional.of("Gösta");

System.out.println(maybeName.isEmpty());  // true
System.out.println(maybeName2.isEmpty()); // false
```



Optional<T>

- Optional<T> only works when T is a reference type
  - String, your own classes, etc.
- There are also OptionalInt, OptionalDouble, etc.
  - These classes don't have the get method
  - Instead, they have getAsInt, getAsDouble, etc.



Optional<T>

- Use `Optional` instead of `null` for methods which might not return a value
- Or for optional instance variables
- If I had created `showInputDialog`:
  - `public static Optional<String> showInputDialog(String text)`
- Other imaginary examples:
  - `public static OptionalDouble safeDiv(double d, double divisor)`
    - Returns `OptionalDouble.empty()` if `divisor` is 0
  - `public Optional<String> getAddress()`
    - Returns `Optional.empty()` if person is homeless
  - `public OptionalInt getWinner()`
    - Returns `Optional.empty()` if game still doesn't have a winner

Example: Optional Pony Skills

Recommended Reading

- *Java Direkt med Swing* 3.11, 19.3.1
- `Optional<T>` tutorial
  - https://dzone.com/articles/java-8-optional-usage-and-best-practices
- *Code Complete* Chapter 5