

Utilities (Part 1)

Implementing static features

Goals for Today

- ▶ initiate the design of simple class
- ▶ learn about class attributes
 - ▶ public
 - ▶ static
 - ▶ final

Motivation

- ▶ the game Yahtzee
 - ▶ use the link above to see the rules of the game



- ▶ why?
 - ▶ opportunity to solve small computational problems that are related to much harder problems

Yahtzee Roll Categories

Category	Description	Example
Three of a kind	at least three dice having the same value	6-2-3-2-2
Four of a kind	at least four dice having the same value	5-5-5-1-5
Full house	three-of-a-kind and a pair	2-3-3-2-3
Small straight	at least four sequential dice	3-1-3-4-2
Large straight	five sequential dice	5-1-3-4-2
Yahtzee	all five dice having the same value	4-4-4-4-4

- ▶ if I gave you a `List<Die>` containing 5 dice can you write a Java program that determines if the roll belongs to a particular category?

- ▶ http://www.eecs.yorku.ca/course_archive/2012-13/W/1030/Z/labs/01/doc/

Yahtzee Roll Categories

- ▶ there are several different approaches that you can use to determine if a roll belongs to a particular category
 - ▶ try to find a few different approaches for each category
- ▶ however, starting by sorting the list of dice simplifies the problem

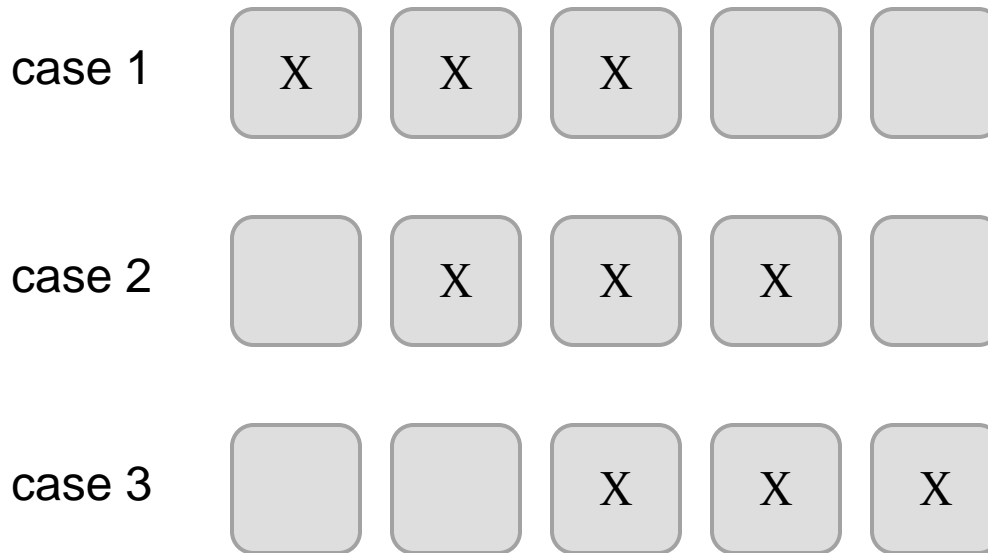
Sorting a List

- ▶ you can sort a `List<Die>` by using the `sort` method in the utility class `java.util.Collections`

```
// dice is a List<Die> reference  
Collections.sort(dice);
```

Why Does Sorting Help?

- ▶ sorting reduces the number of cases that you have to check; consider the category three-of-a-kind
 - ▶ after sorting the dice you only have to check if one of three cases are true



don't care
about the
values of the
blank dice

Three-of-a-kind?

```
// dice is a List<Die> reference
Collections.sort(dice);
boolean isThreeOfAKind =
    dice.get(0).getValue() == dice.get(2).getValue() ||
    dice.get(1).getValue() == dice.get(3).getValue() ||
    dice.get(2).getValue() == dice.get(4).getValue();
```


Sorting in General

- ▶ sorting seems useful
 - ▶ what other examples can you think of?
- ▶ how would you implement `Collections.sort`?
 - ▶ in-class sorting contest here

Sorting Strategies Tried by Students



Bad Ways to Sort

- ▶ bogosort is a very slow algorithm for sorting a list

```
while the list is not sorted {  
    randomly shuffle the elements in the list  
}
```

- ▶ bozosort is another very slow algorithm

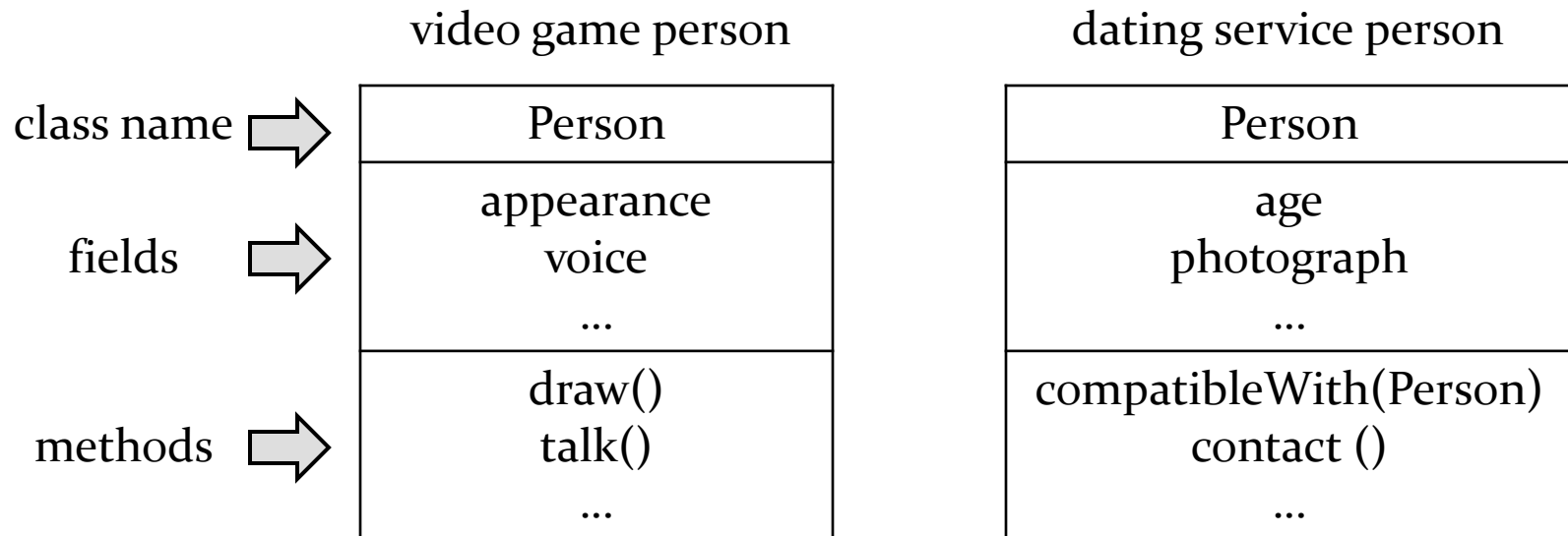
```
while the list is not sorted {  
    pick two elements at random and swap them  
}
```

Review: Java Class

- ▶ a class is a model of a thing or concept
- ▶ in Java, a class is the blueprint for creating objects
 - ▶ fields (or attributes)
 - ▶ the structure of an object; its components and the information (data) contained by the object
 - ▶ methods
 - ▶ the behaviour of an object; what an object can do

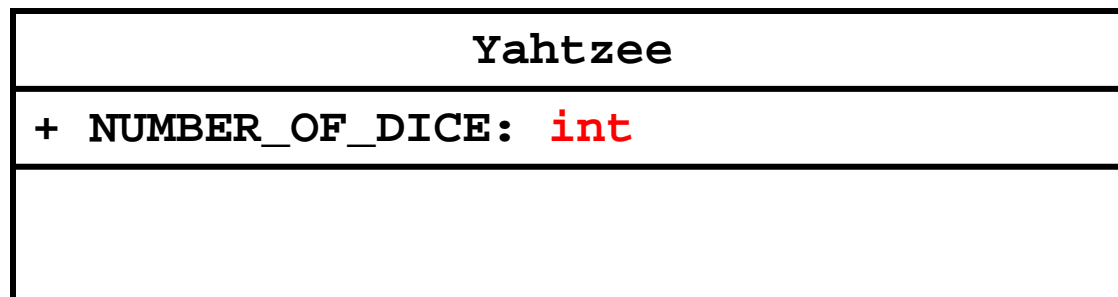
Designing a Class

- ▶ to decide what fields and methods a class must provide, you need to understand the problem you are trying to solve
 - ▶ the fields and methods you provide (the abstraction you provide) depends entirely on the requirements of the problem



A Class for Yahtzee

- ▶ design a class to encapsulate features of Yahtzee
- ▶ what fields are needed?
 - ▶ number of dice
 - ▶ note: the number of dice never changes; it is genuinely a constant value for the game called Yahtzee
 - ▶ attributes that are constant have all uppercase names



field type

Version 1

```
public class Yahtzee {  
  
    public static final int NUMBER_OF_DICE = 5;  
}
```



Fields

```
public static final int NUMBER_OF_DICE = 5;
```

- ▶ a field is a member that holds data
- ▶ a constant field is usually declared by specifying

1. modifiers

1. access modifier **public**
2. static modifier **static**
3. final modifier **final**

2. type **int**

3. name **NUMBER_OF_DICE**

4. value **5**

Fields

- ▶ field names must be unique in a class
- ▶ the scope of a field is the entire class
- ▶ [JBA] and [notes] use the term "field" only for **public** fields

public Fields

- ▶ a **public** field is visible to all clients

```
public class NothingToHide {  
    public int x; // always positive  
}
```

```
// client of NothingToHide  
NothingToHide h = new NothingToHide();  
h.x = 100;
```

public Fields

- ▶ **public** fields break encapsulation
 - ▶ a `NothingToHide` object has no control over the value of `x`
 - ▶ a client can put a `NothingToHide` object into an invalid state because the client has direct access to a **public** field

```
public class NothingToHide {  
    public int x; // always positive  
}
```

```
// client of NothingToHide  
NothingToHide h = new NothingToHide();  
h.x = 100;  
h.x = -5; // not positive
```

public Fields

- ▶ a **public** field makes a class brittle in the face of change

```
public class NothingToHide {  
    private int x; // always positive  
}
```

```
// existing client of NothingToHide  
NothingToHide h = new NothingToHide();  
h.x = 100; // no longer compiles
```

- ▶ **public** fields are hard to change
 - ▶ they are part of the class API
 - ▶ changing access or type will break existing client code

public Fields

- ▶ avoid **public** fields in production code
 - ▶ except when you want to expose constant value types

static Fields

- ▶ a field that is **static** is a per-class member
 - ▶ only one copy of the field, and the field is associated with the class
 - ▶ every object created from a class declaring a static field shares the same copy of the field
 - ▶ textbook uses the term *static variable*
 - ▶ also commonly called *class variable*

static Fields

```
Yahtzee y = new Yahtzee();  
Yahtzee z = new Yahtzee();
```

y
z

NUMBER_OF_DICE

belongs to class



no copy of
NUMBER_OF_DICE



???

???

see [JBA 4.3.3] for another example

64

client invocation

1000

1100

500

Yahtzee class

5

1000

Yahtzee object

1100

Yahtzee object

static Field Client Access

- ▶ a client should access a **public static** field without using an object
 - ▶ use the class name followed by a period followed by the attribute name

```
// client of Yahtzee
List<Die> dice = new List<Die>();
for(int i = 0; i < Yahtzee.NUMBER_OF_DICE; i++) {
    dice.add(new Die(6));
}
```


static Attribute Client Access

- ▶ it is legal, *but considered bad form*, to access a **public static** attribute using an object

```
// client of Yahtzee; avoid doing this
Yahtzee y = new Yahtzee();
List<Die> dice = new List<Die>();
for(int i = 0; i < y.NUMBER_OF_DICE; i++) {
    dice.add(new Die(6));
}
```

final Fields

- ▶ an field that is **final** can only be assigned to once
 - ▶ **public static final** attributes are typically assigned when they are declared

```
public static final int NUMBER_OF_DICE = 5;
```

- ▶ **public static final** attributes are intended to be constant values that are a meaningful part of the abstraction provided by the class

`final` Fields of Primitive Types

- ▶ `final` fields of primitive types are constant

```
public class AlsoNothingToHide {  
    public static final int X = 100;  
}
```

```
// client of AlsoNothingToHide  
AlsoNothingToHide.X = 88; // will not compile;  
                          // attribute is final and  
                          // previously assigned
```

final Fields of Immutable Types

- ▶ **final** fields of immutable types are constant

```
public class StillNothingToHide {  
    public static final String X = "peek-a-boo";  
}
```

```
// client of StillNothingToHide  
StillNothingToHide.X = "i-see-you";  
                        // will not compile;  
                        // field is final and  
                        // previously assigned
```

- ▶ also, **String** is immutable
 - ▶ it has no methods to change its contents

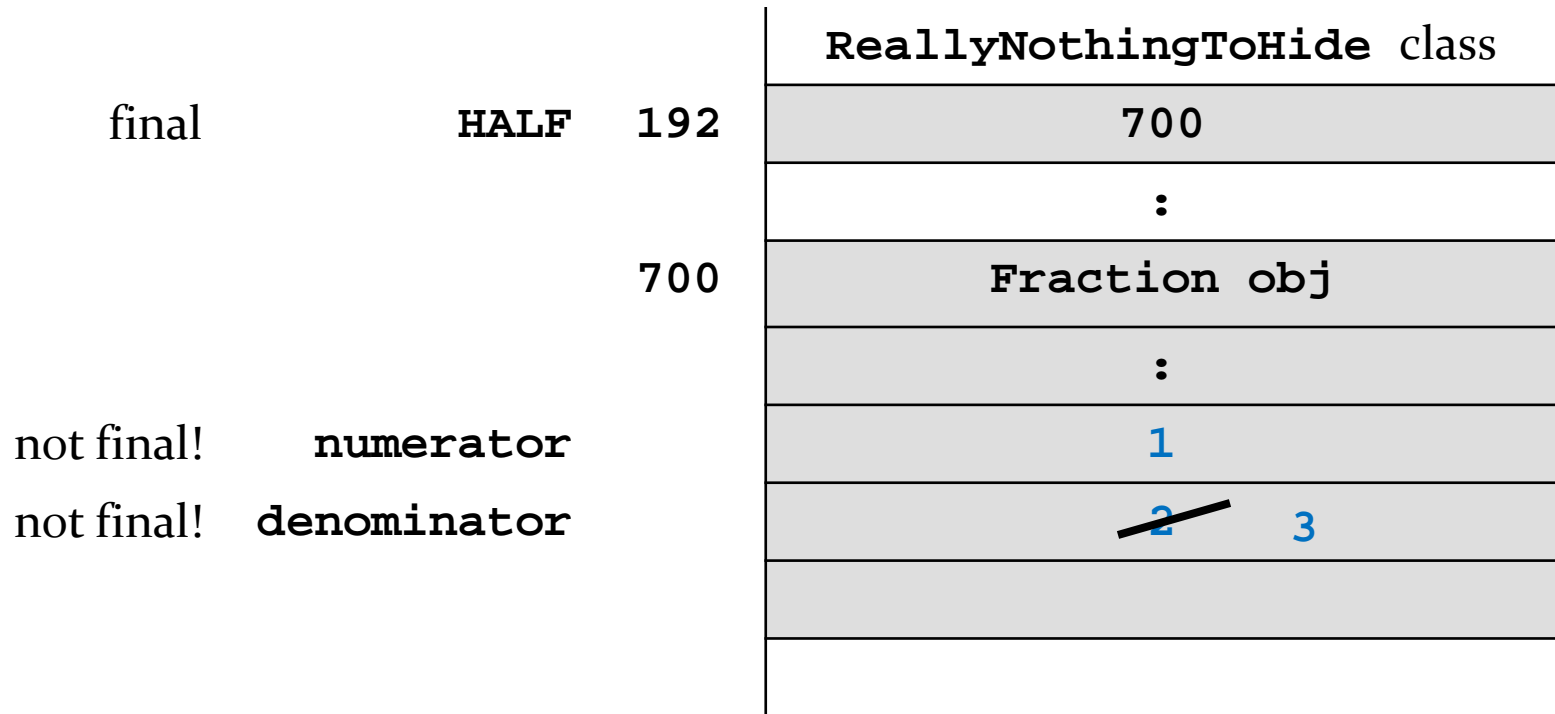
`final` Fields of Mutable Types

- ▶ `final` fields of mutable types are not logically constant; their state can be changed

```
public class ReallyNothingToHide {  
    public static final Fraction HALF =  
                                   new Fraction(1, 2);  
}
```

```
// client of ReallyNothingToHide  
Fraction third = new Fraction(1, 3);  
ReallyNothingToHide.HALF = third; // will not compile;  
                                   // HALF is final and  
                                   // already assigned  
  
ReallyNothingToHide.HALF.setDenominator(3); // works!!
```

final Fields of Mutable Types



```
ReallyNothingToHide.HALF.setDenominator(3);
```

final Fields of Mutable Types

- ▶ **final** fields of mutable types are not logically constant; their state can be changed

```
public class LastNothingToHide {  
    public static final ArrayList<Integer> X =  
        new ArrayList<Integer>();  
}
```

```
// client of LastNothingToHide  
ArrayList<Integer> y = new ArrayList<Integer>();  
LastNothingToHide.X = y;    // will not compile;  
                             // attribute is final and  
                             // previously assigned  
  
LastNothingToHide.X.add( 10000 );  
                             // works!
```

`final` Attributes

- ▶ avoid using mutable types as `public` constants
 - ▶ they are not logically constant

Puzzle

- ▶ what does the following program print?

```
public class What
{
    public static void main(String[] args)
    {
        final long
            MICROS_PER_DAY = 24 * 60 * 60 * 1000 * 1000;
        final long
            MILLIS_PER_DAY = 24 * 60 * 60 * 1000;
        System.out.println(MICROS_PER_DAY / MILLIS_PER_DAY);
    }
}
```