Utilities (Part 1)

Implementing static features

Goals for Today

- definition of a utility class
- initiate the design of a utility class
- learn about class attributes
  - public
  - static
  - final

Review: Java Class

- a class is a model of a thing or concept

- in Java, a class is usually a 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

Utility classes

- sometimes, it is useful to create a class called a *utility class* that is not used to create objects
  - such classes have no constructors for a client to use to create objects
- in a utility class, all features are marked as being **`static`**
  - you use the class name to access these features
- examples of utility classes:
  - **`java.lang.Math`**
  - **`java.util.Arrays`**
  - **`java.util.Collections`**

Utility classes

- the purpose of a utility class is to group together related fields and methods where creating an object is not necessary
- **`java.lang.Math`**
  - groups mathematical constants and functions
  - do not need a **`Math`** object to compute the cosine of a number
- **`java.util.Collections`**
  - groups methods that operate on Java collections
  - do not need a **`Collections`** object to sort an existing **`List`**

A simple utility class

- implement a utility class that helps you calculate Einstein's famous mass-energy equivalence equation $E = mc^2$ where
  - m is mass (in kilograms)
  - c is the speed of light (in metres per second)
  - E is energy (in joules)

Start by giving the class a name and creating the class body block.

```
public class Relativity {




}
```

Add a field that represents the speed of light.

```
public class Relativity {

  public static final double C = 299792458;




}
```

Add a method to compute $E = mc^2$.

```
public class Relativity {

  public static final double C = 299792458;

  public static double massEnergy(double mass) {
    return mass * Relativity.C * Relativity.C;
  }

}
```

Utility class for a game

- 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

http://barefootliam-stock.deviantart.com/art/five-ivory-dice-97476774

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

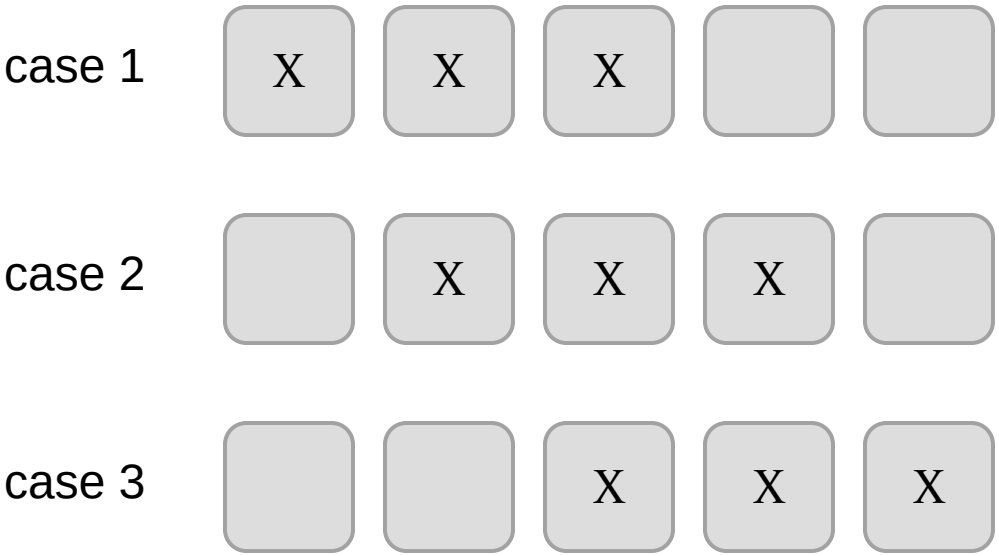
Sorting a List

- 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

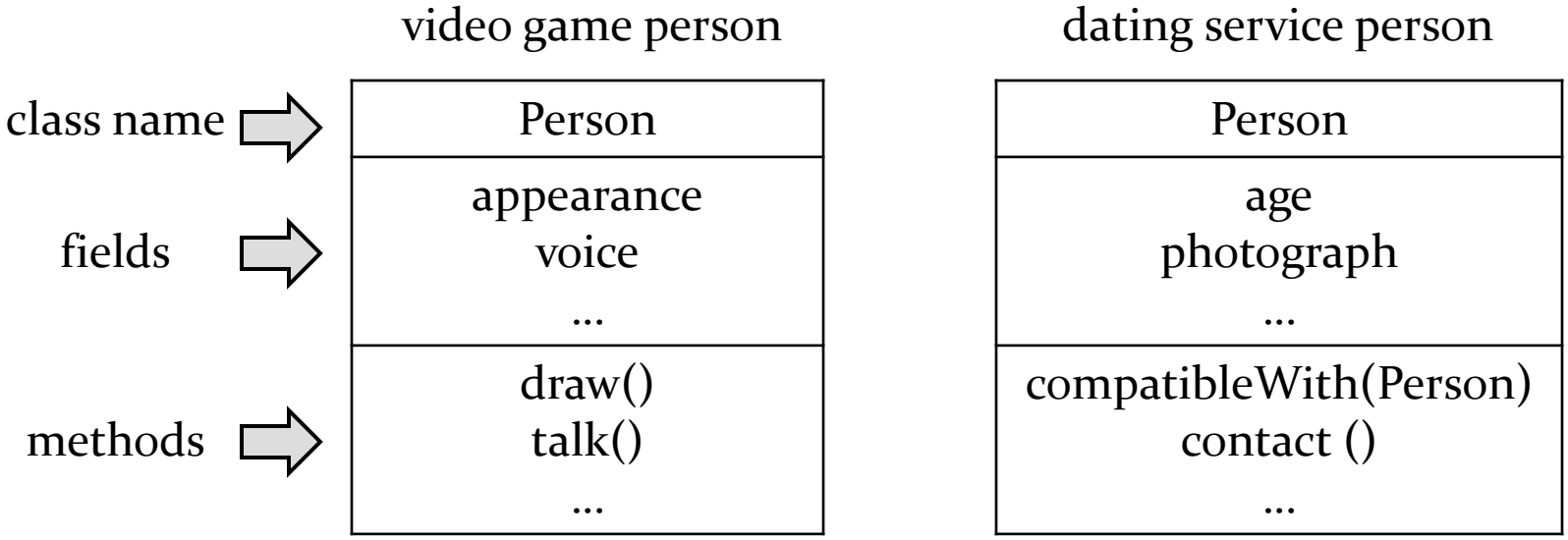


Three-of-a-kind?

```
// 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();
```

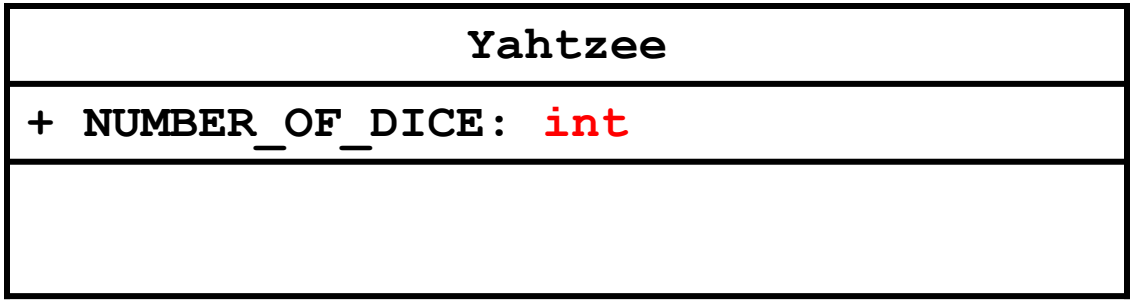
Designing a Class

- 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
    - fields that are constant have all uppercase names



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 exisiting client code

`public` Fields

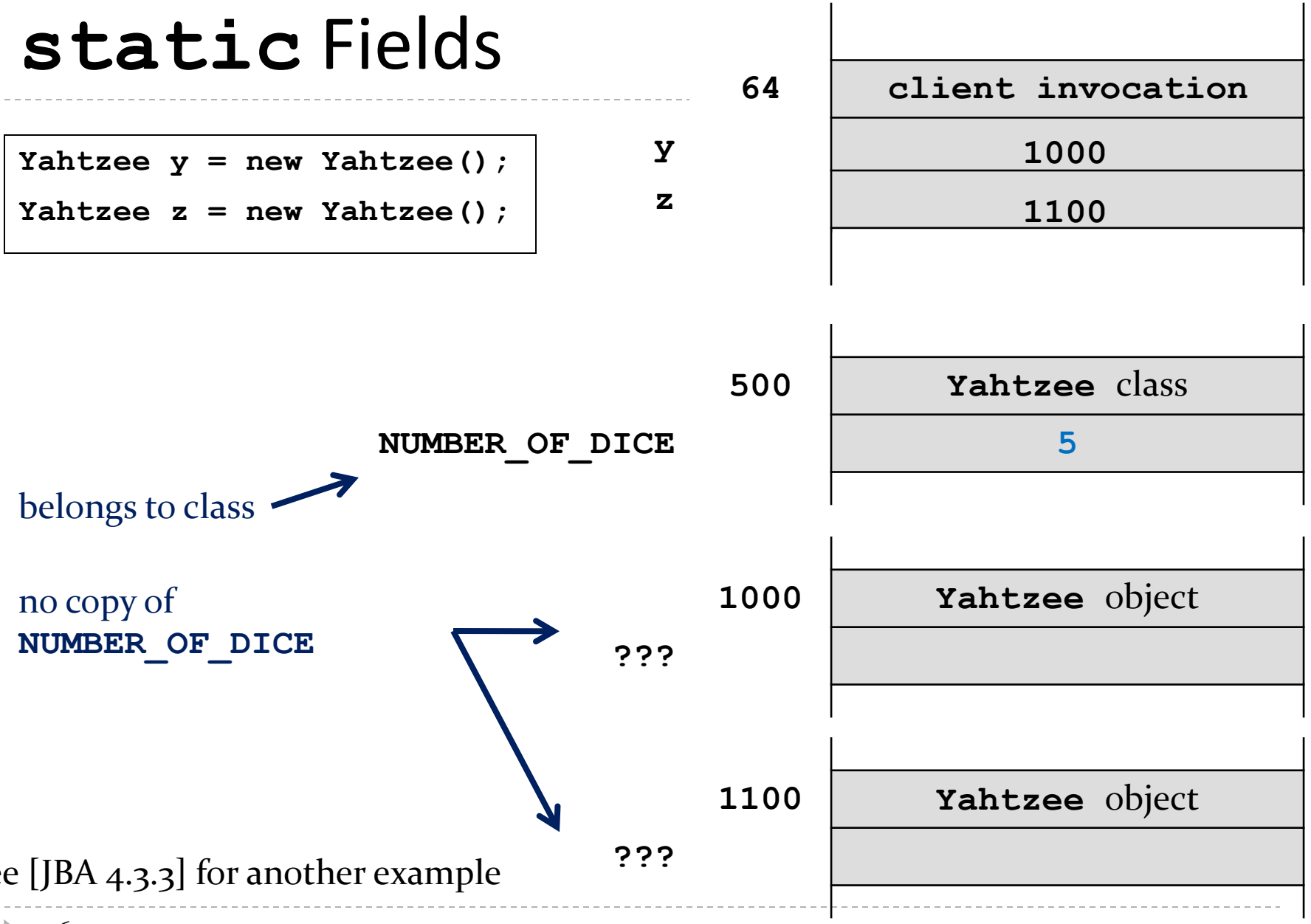
- avoid **`public`** fields in production code
  - except when you want to expose constant value types

`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();
```



see [JBA 4.3.3] for another example

`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;
                           // field X 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 X is final and
                              // previously assigned
```

- **`String`** is immutable
  - it has no methods to change its contents

`final` Fields of Mutable Types

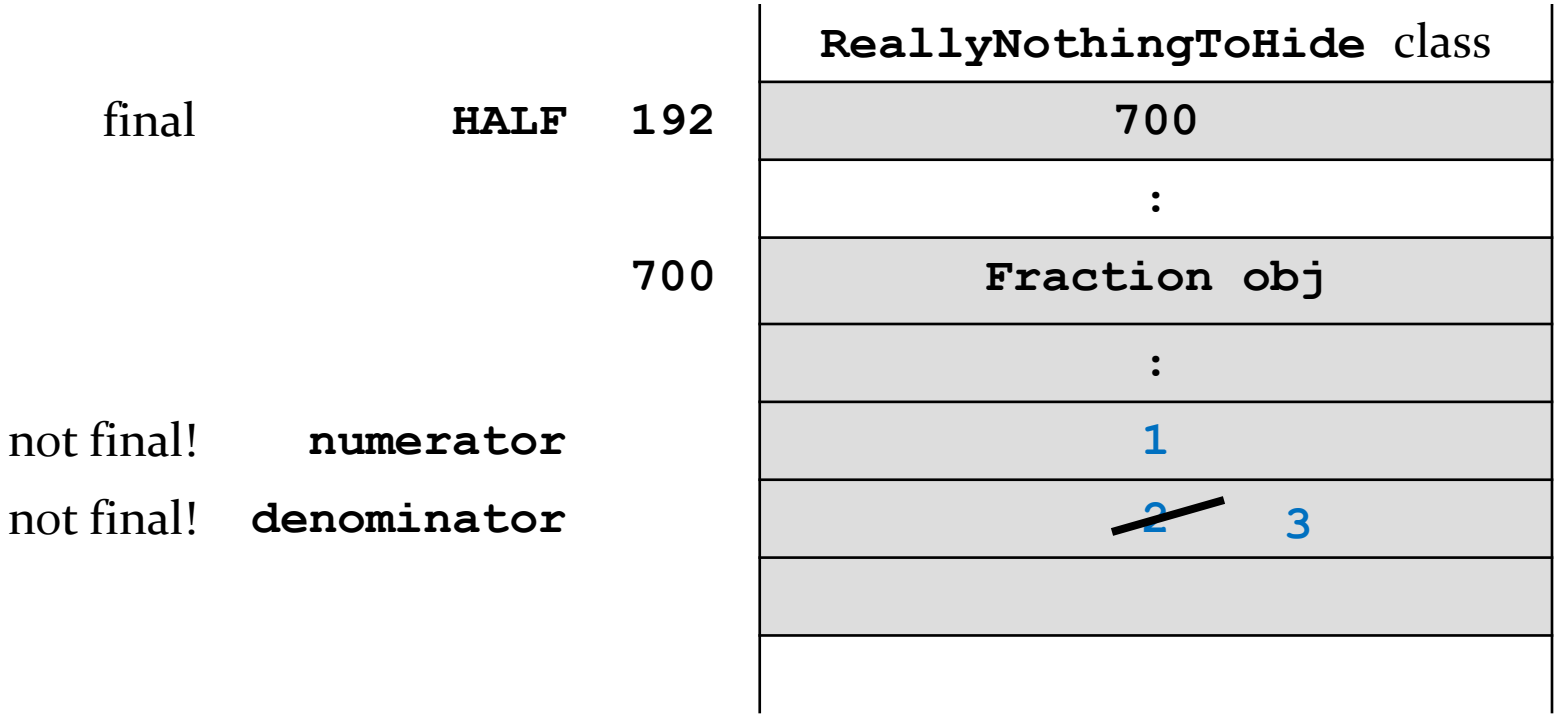
- **`final`** fields of mutable types are not logically constant; their state can be changed

```
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!!
                                   // HALF is now 1/3
```

`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!
                           // X is no longer empty
```

`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);
  }
}
```