Introduction to Computer Science II

CSE1030A

Academic Support Programs: Bethune

- having trouble with your FSC and LSE courses?
  - consider using the Academic Support Programs at Bethune College
  - PASS
    - free, informal, structured, facilitated study groups: http://bethune.yorku.ca/pass/
  - For the Summer 2014 term, the PASS Leader for CSE 1030 is Mr. Arinze Anozie, beckyngomakaya@hotmail.com
  - peer tutoring
    - free, one-on-one, drop-in tutoring: http://bethune.yorku.ca/tutoring/

Who Am I?

- Dr. Andriy Pavlovych
- Office
  - Lassonde 2001
  - Office hours : check with syllabus on course web page
    - Most probably – 1 hour right before class
- email
  - **andriyp@cse.yorku.ca**
    - **(use "CSE 1030" in the subject line)**

Course Format

- everything you need to know will be on course website
  - `http://www.eecs.yorku.ca/course/1030`

- labs start on Thursday* (June 26)
  - * – Subject to change

- BUT you should do Lab 0 this week if you have not taken an EECS course before OR if you have not used eclipse before

CSE1030 Overview

- in CSE1020, you learned how to use objects to write Java programs
  - a Java program is made up of one or more interacting objects
  - each object is an instance of a class
- where do the classes come from?
- in CSE1030, you will learn how to design and implement classes
  - introduction to concepts in software engineering and computer science

What You Should Know from CSE1020

- how to read an API
  - determine what package a class is located in
  - determine what the class/interface/field/method is supposed to do
  - determine the name of a method
  - determine what types a method requires for its parameters
  - determine what type a method returns
  - determine what exceptions might be thrown

What You Should Know from CSE1020

- create and use primitive type variables and their associated operators
  - **`int`**, **`double`**, **`boolean`**, **`char`**

What You Should Know from CSE1020

- create (using a constructor) and use reference variables
  - e.g., **`type.lib.Fraction`**, **`java.util.Date`**
  - **`Random`**, **`String`**, **`List`**, **`Set`**, **`Map`**

What You Should Know from CSE1020

- understand the difference between primitive and reference types
  - memory diagrams

What You Should Know from CSE1020

- understand the difference between **`==`** and **`equals`**

What You Should Know from CSE1020

- use class methods (and fields)
  - e.g.,

```
double value = Math.sqrt(2.0);
```

- use instance methods (and fields)
  - e.g.,

```
String s = "hello";
String t = s.toUpperCase();
```

What You Should Know from CSE1020

- **`if`** statements
- e.g.

```
if (grade >= 65) {
    System.out.println("Go to second year");
}
else {
    System.out.println("Try again");
}
```

What You Should Know from CSE1020

- **`for`** loops
- e.g., for some **`String`** reference **`s`**

```
for (int i = 0; i < s.length(); i++) {
  char c = s.charAt(i);
  if (c == 'a') {
    System.out.println(s + " contains an \'a\'");
    break;
  }
}
```

What You Should Know from CSE1020

- **`for`** each loops
- e.g., for some **`List<String>`** reference **`t`**

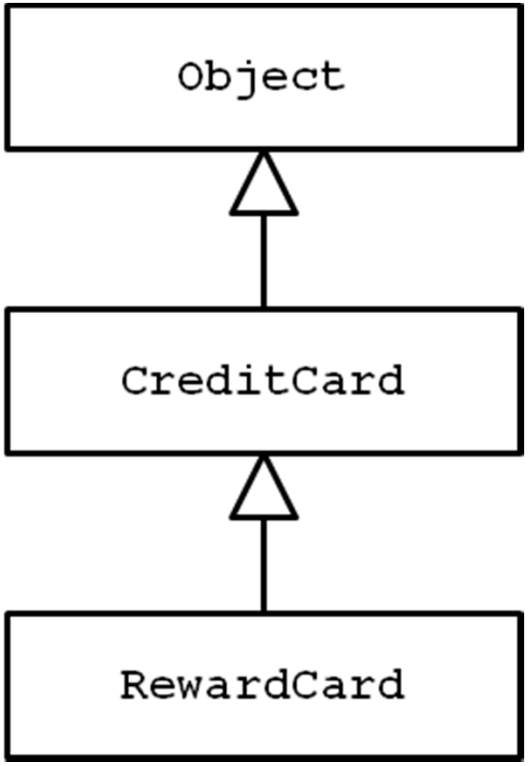
```
for (String s : t) {
  for (int i = 0; i < s.length(); i++) {
    char c = s.charAt(i);
    if (c == 'a') {
      System.out.println(s + " contains an \'a\'");
      break;
    }
  }
}
```

What You Should Know from CSE1020

- the difference between aggregation and composition
- the differences between aliasing, shallow copying, and deep copying

What You Should Know from CSE1020

- inheritance and substitutability



What You Should Know from CSE1020

- what an exception is
- the difference between a checked and unchecked exception
- how to handle exceptions (**`try`** and **`catch`**)

What You Should Know from CSE1020

- style

```
public class hairsOnHead
{
  public static void main(String[] args) {
    int Diameter = 17;
    double f = 0.5;
    double areaCovered=f*Math.PI*Diameter*Diameter;
    int d = 200;
    double numberofhairs = areaCovered * d;
    System.out.print("The number of hairs on a human head is ");
  System.out.println(numberofhairs);
  }
}
```

class names should start with a capital letter

inconsistent brace alignment

variable names should start with a lowercase letter; magic number

variable names should be informative; magic number

1 space around operators

variable names should be informative; magic number

variable names should use camelcase

inconsistent indenting

Organization of a Java Program

Packages, classes, fields, and methods

In This Lecture

1. demonstrate the use of eclipse by solving a CSE1020 eCheck problem
2. review the organization of a typical CSE1020 Java program
3. improve the organization of the program by writing a method
4. explain the organization of a typical Java program that uses packages and multiple classes

eCheck04A

- in a nutshell:
  - write a program that computes the fraction

$$A = \frac{x + y}{z + t}$$

where x, y, z, and t are proper fractions entered by a user from the command line

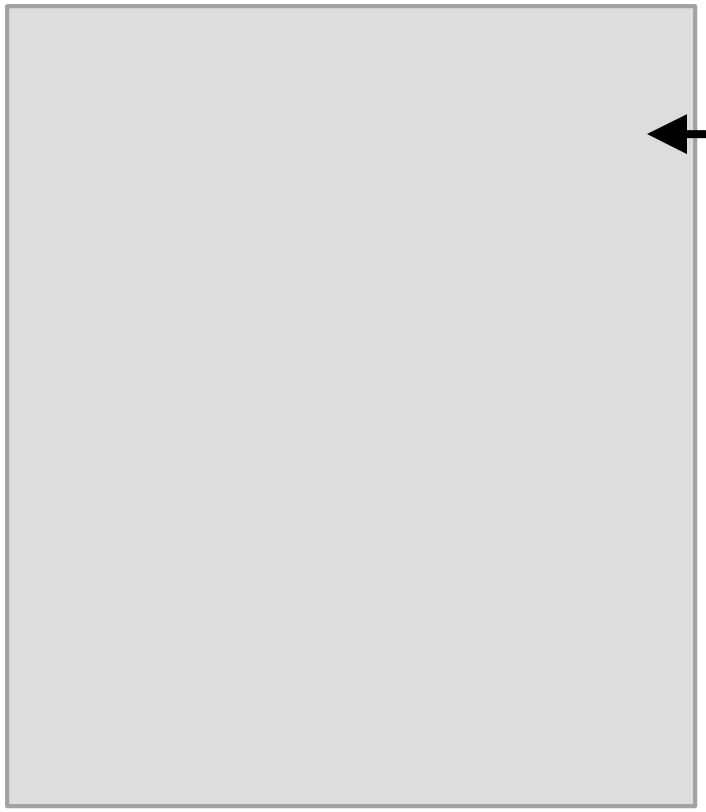
eCheck04A Sample Output

```
For each fraction enter its numerator/denominator,
pressing ENTER after each
Enter x
83
100
Enter y
5
9
Enter z
667
1000
Enter t
-2
3
A = 12470/3 = 4156 2/3 = 4156.666666666667
```

eclipse Demo Here

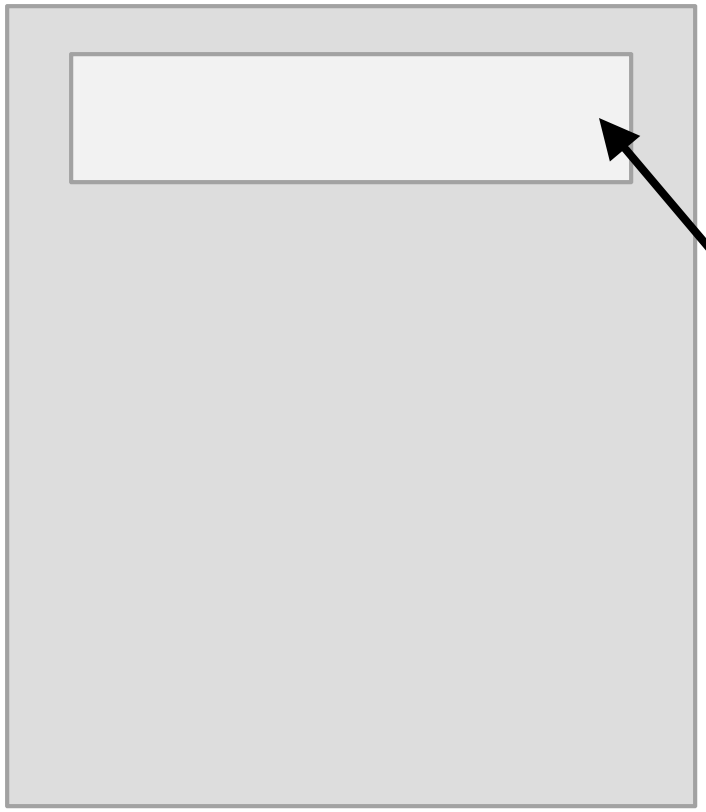
- if you missed this class then you missed this demo

Organization of a CSE1020 Program



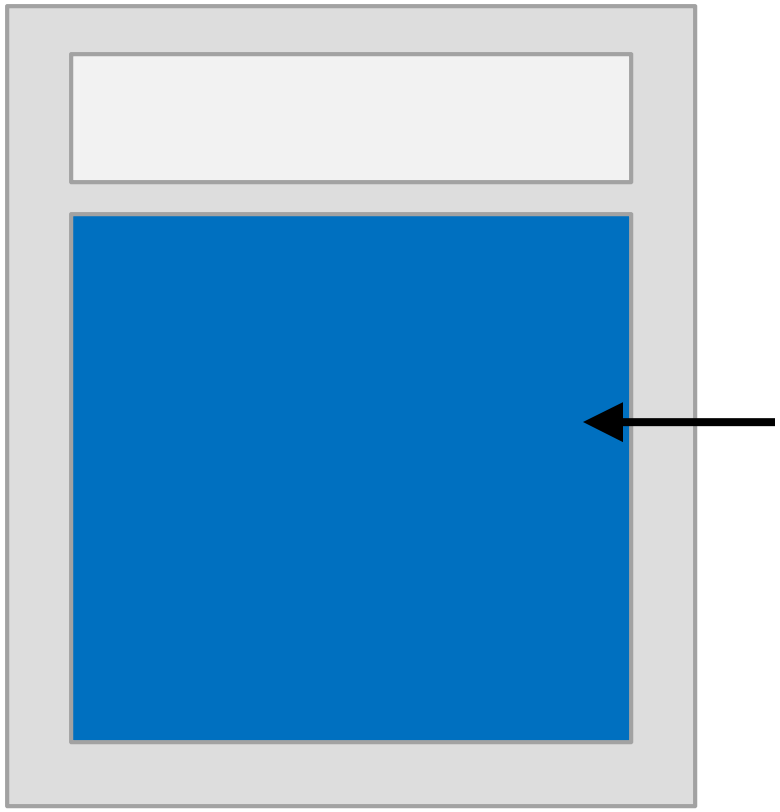
- one file
  - **`Check04A.java`**

Organization of a CSE1020 Program



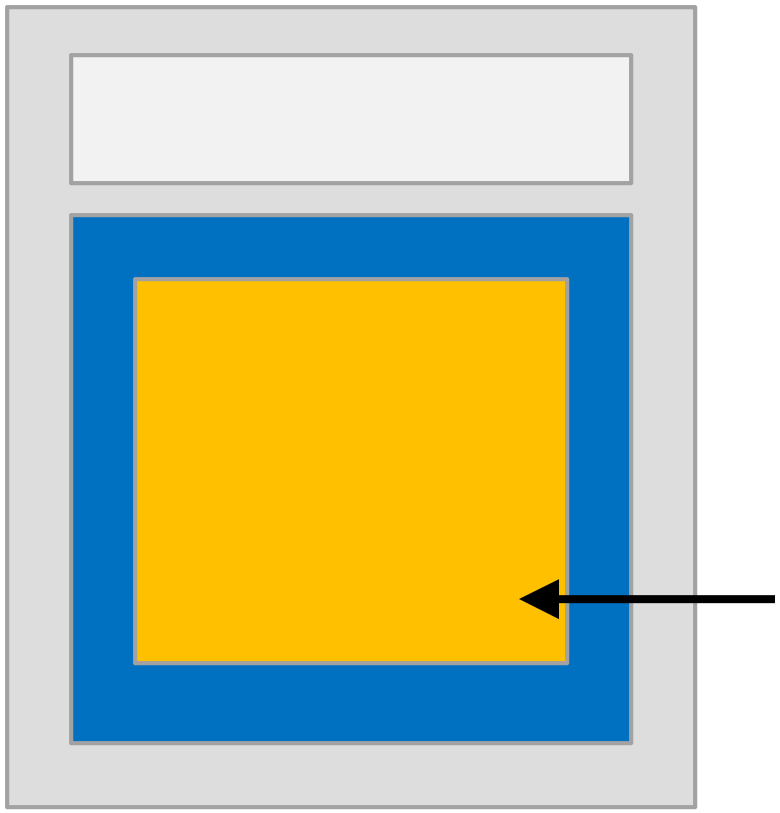
- one file
  - `Check04A.java`
- zero or more import statements

Organization of a CSE1020 Program



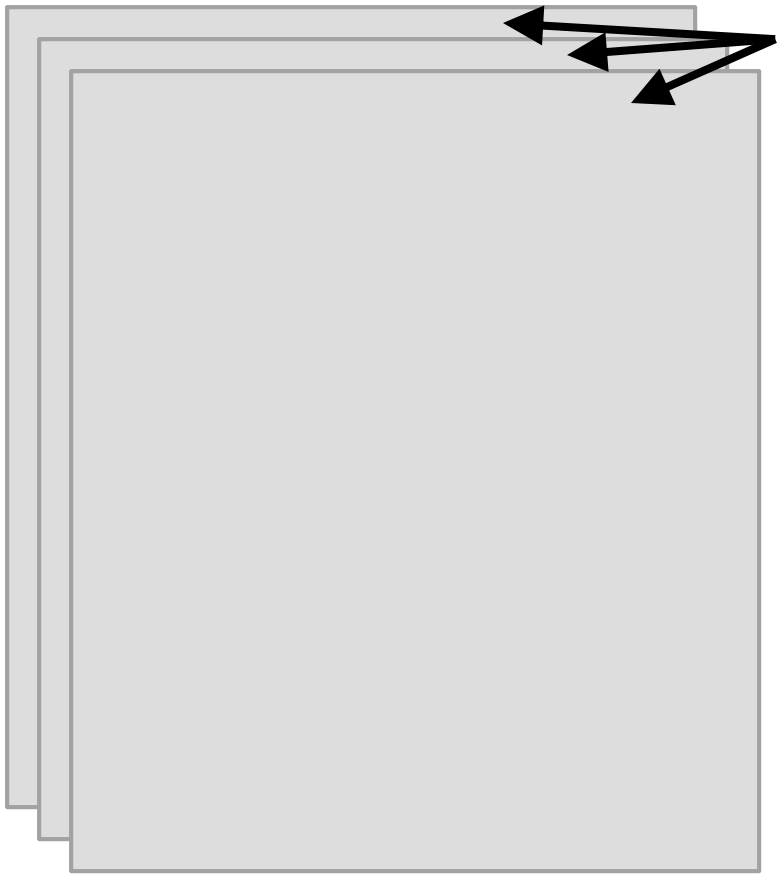
- file
  - `Check04A.java`
- zero or more import statements
- one class
  - **`Check04A`**

Organization of a CSE1020 Program



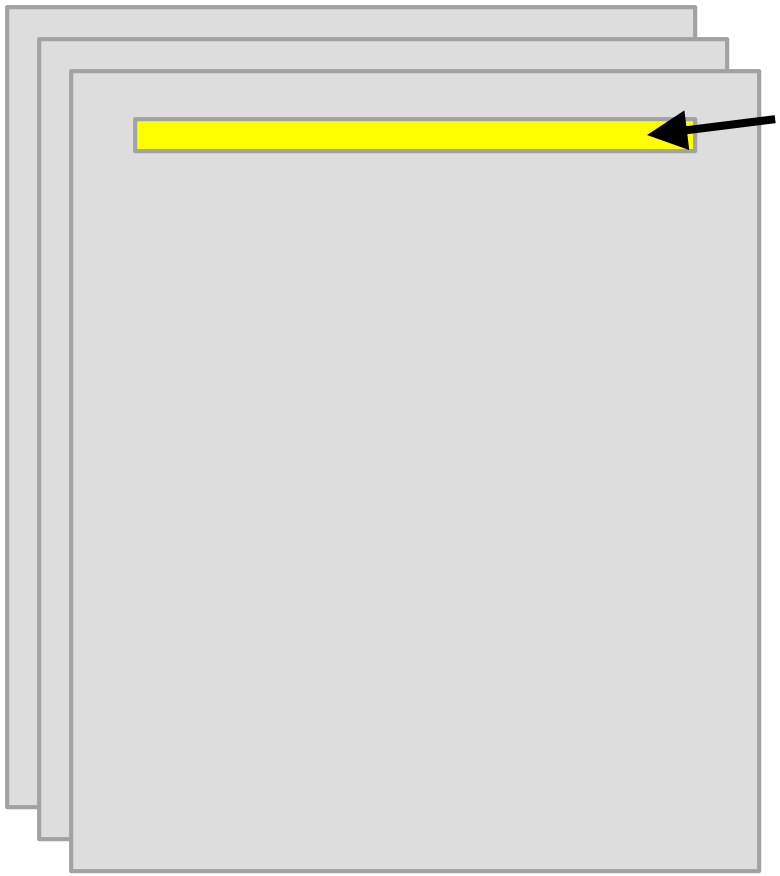
- file
  - Checko4A.java
- zero or more import statements
- one class
  - **`Check04A`**
- one static method
  - **`main`**

Organization of a Typical Java Program



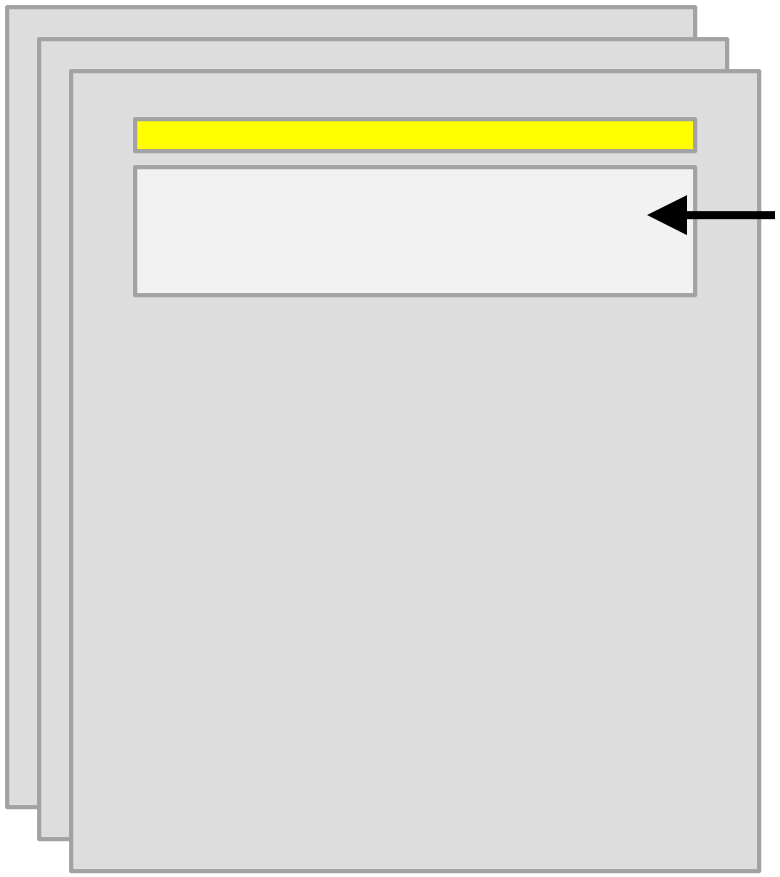
- one or more files

Organization of a Typical Java Program



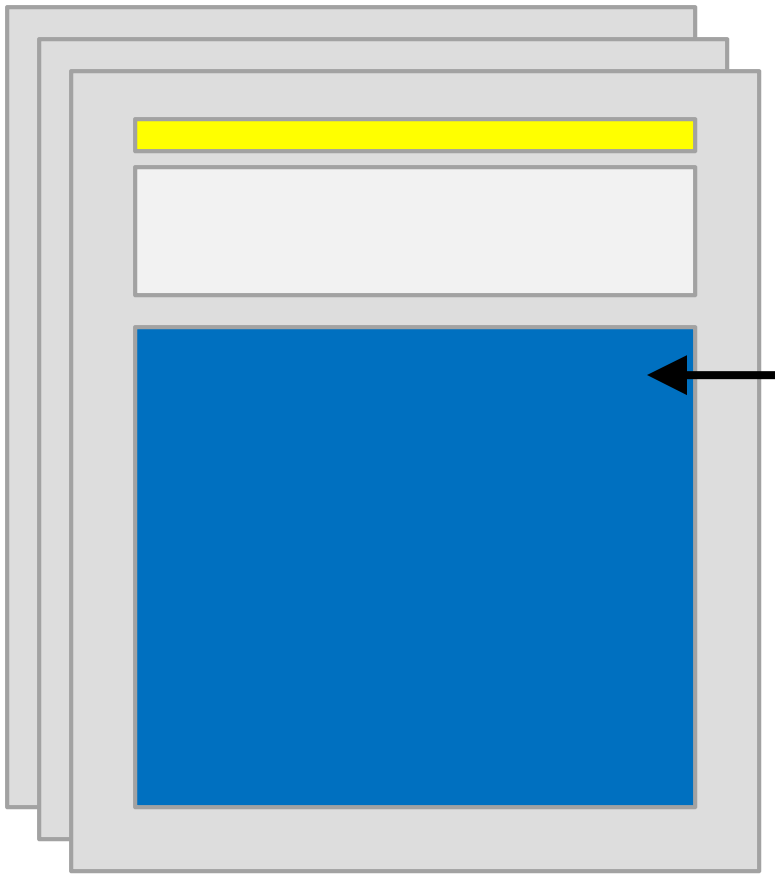
- one or more files
- one package name

Organization of a Typical Java Program



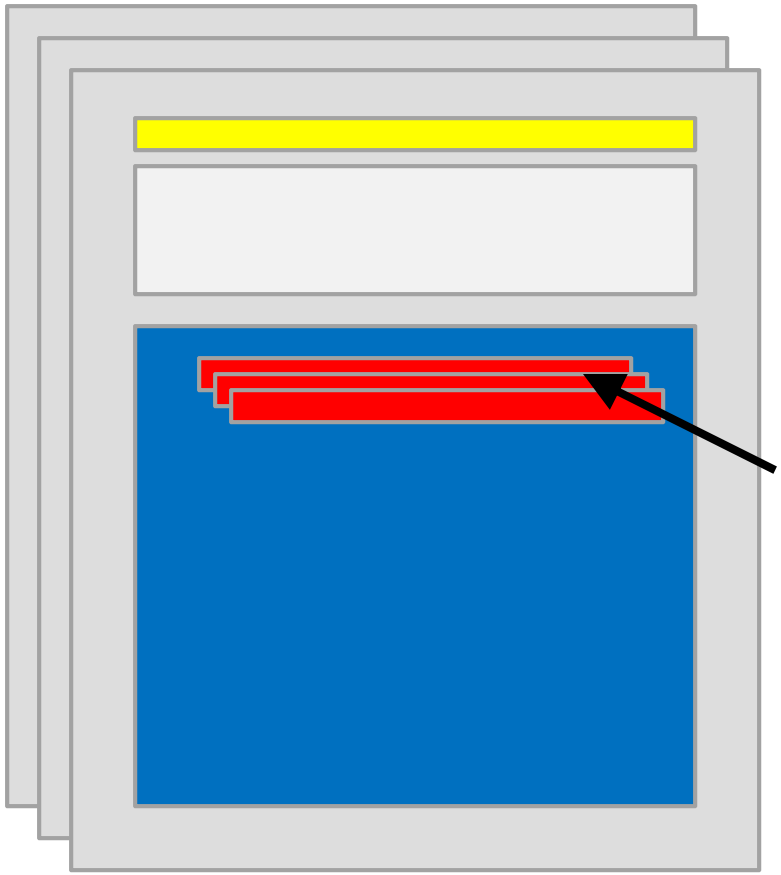
- one or more files
- zero or one package name
- zero or more import statements

Organization of a Typical Java Program



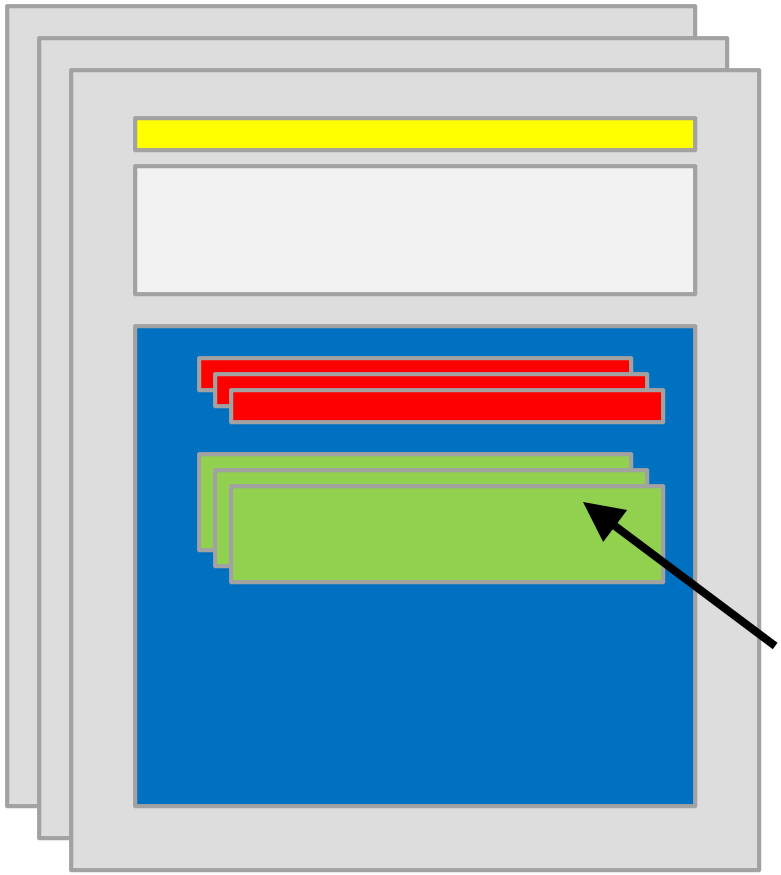
- one or more files
- zero or one package name
- zero or more import statements
- one class

Organization of a Typical Java Program



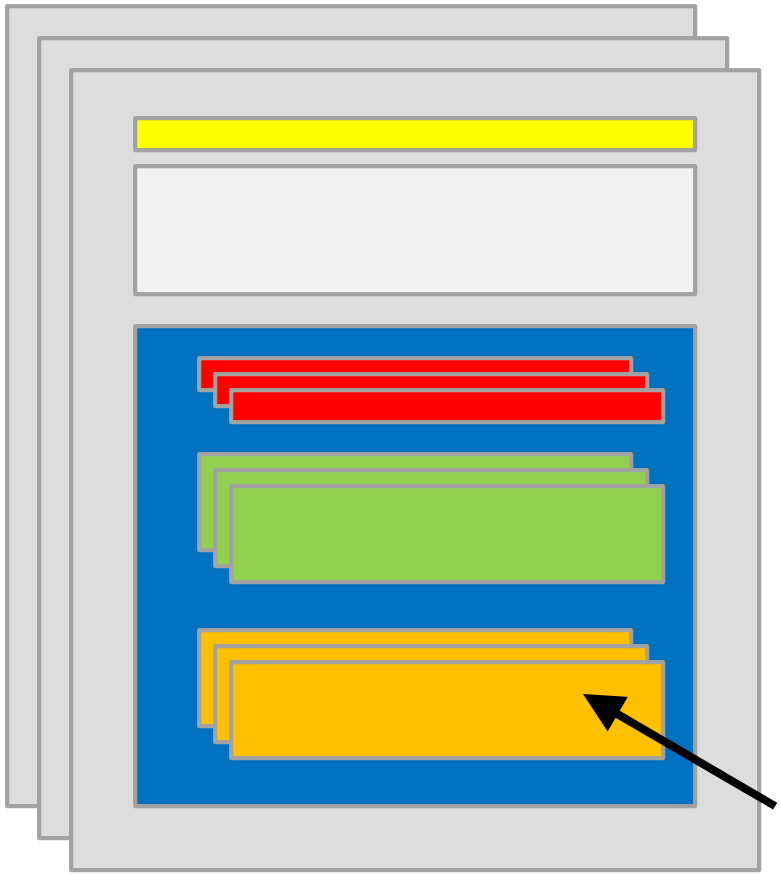
- one or more files
- zero or one package name
- zero or more import statements
- one class
- one or more fields (class variables)

Organization of a Typical Java Program



- one or more files
- zero or one package name
- zero or more import statements
- one class
- zero or more fields (class variables)
- zero or more more constructors

Organization of a Typical Java Program



- one or more files
- zero or one package name
- zero or more import statements
- one class
- zero or more fields (class variables)
- zero or more more constructors
- zero or more methods

Organization of a Typical Java Program

- it's actually more complicated than this
  - static initialization blocks
  - non-static initialization blocks
  - classes inside of classes (inside of classes ...)
  - classes inside of methods

- see http://docs.oracle.com/javase/tutorial/java/javaOO/index.html

Packages

- packages are used to organize Java classes into namespaces
- a namespace is a container for names
  - the namespace also has a name

Packages

- packages are use to organize related classes and interfaces
  - e.g., all of the Java API classes are in the package named **`java`**

Packages

- packages can contain subpackages
  - e.g., the package **`java`** contains packages named **`lang`**, **`util`**, **`io`**, etc.

- the fully qualified name of the subpackage is the fully qualified name of the parent package followed by a period followed by the subpackage name
  - e.g., **`java.lang`**, **`java.util`**, **`java.io`**

Packages

- packages can contain classes and interfaces
  - e.g., the package **`java.lang`** contains the classes **`Object`**, **`String`**, **`Math`**, etc.

- the fully qualified name of the class is the fully qualified name of the containing package followed by a period followed by the class name
  - e.g., **`java.lang.Object`**, **`java.lang.String`**, **`java.lang.Math`**

Packages

- packages are supposed to ensure that fully qualified names are unique
- this allows the compiler to disambiguate classes with the same unqualified name, e.g.,

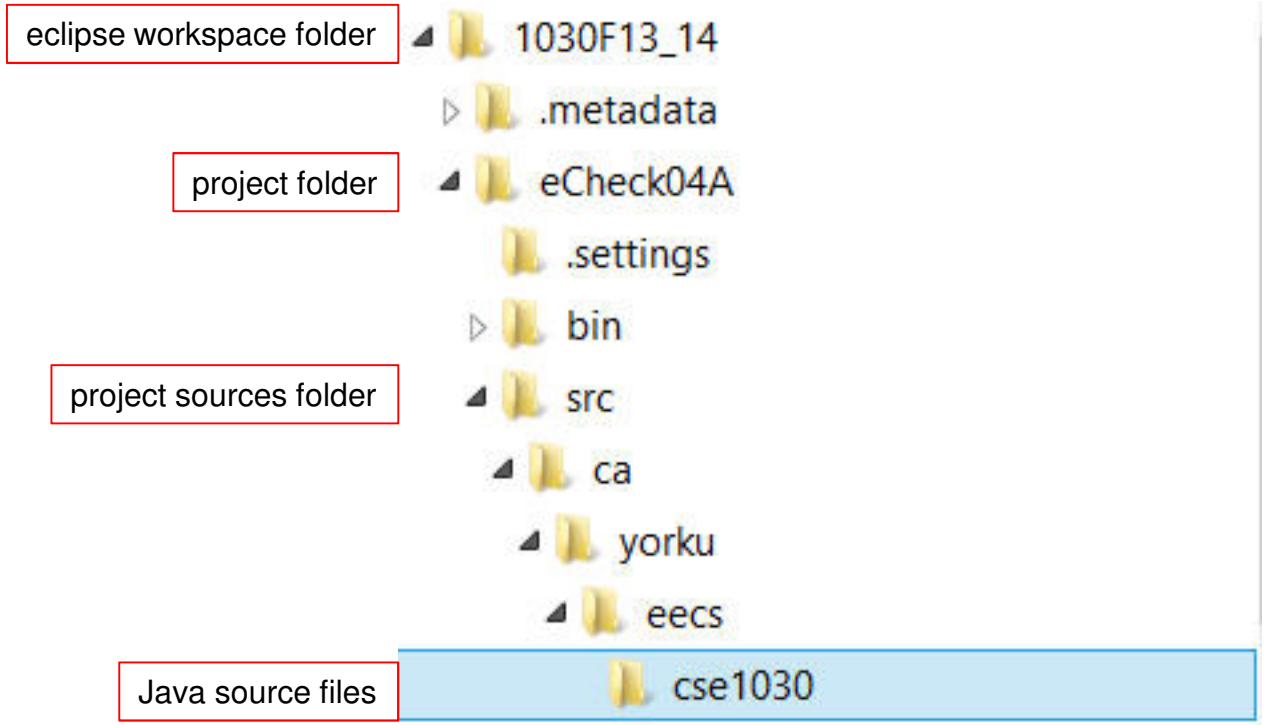
```
your.Fraction f = new your.Fraction(1, 3);
type.lib.Fraction g = new type.lib.Fraction(1, 3);
```

Packages

- how do we ensure that fully qualified names are unique?
- package naming convention
  - packages should be organized using your domain name in reverse, e.g.,
    - EECS domain name **`eecs.yorku.ca`**
    - package name **`ca.yorku.eecs`**
- we might consider putting everything for this course under the following package
  - **`ca.yorku.eecs.cse1030`**

Packages

- most Java implementations assume that your directory structure matches the package structure, e.g.,
  - there is a sequence of folders **`ca\yorku\eecs\cse1030`** inside the project **`src`** folder



Things For You to do this Week

- get a CSE account if you do not already have one
- do Lab 00 to get (re)acquainted with eclipse and the CSE labs
  - available tomorrow
- review CSE1020

CSE1020 Review Questions

CSE1020 Review

- what does the following program print?

```
public class Puzzle01
{
  public static void main(String[] args)
  {
       System.out.print("C" + "S" + "E");
       System.out.println('1' + '0' + '3' + '0' + 'z');
  }
}
```

CSE1020 Review

- which of the following methods are associated with a class?

```
static boolean disjoint(Collection<?> c1, Collection<?> c2)

void setIcon(Icon newIcon)

String toString()

static int round(double a)

static void showMessageDialog(Component parent, Object message)
```

CSE1020 Review

- what is the return type for each of the following methods?

```
static boolean disjoint(Collection<?> c1, Collection<?> c2)

void setIcon(Icon newIcon)

String toString()

static int round(double a)

static void showMessageDialog(Component parent, Object message)
```

CSE1020 Review

- how many parameters do each of the following methods have, and what are their types?

```
static boolean disjoint(Collection<?> c1, Collection<?> c2)

void setIcon(Icon newIcon)

String toString()

static int round(double a)

PrintStream printf(String format, Object... args)
```

CSE1020 Review

- what is a method precondition
- what is a method postcondition?

- what happens if a precondition is violated?
- who is responsible if a postcondition is false?

CSE1020 Review

- a **`type.lib.Fraction`** object has two attributes: a numerator and a denominator
- draw the memory diagram for the following program
  - after line 1 completes
  - after line 2 completes

```
import type.lib.Fraction;

public class Fraction1 {
  public static void main(String[] args) {
    Fraction f = new Fraction(1, 2);   // 1
    f.add(new Fraction(3, 4));         // 2
  }
}
```

CSE1020 Review

- class **`X`** is an aggregation of one **`Y`**; it has a method **`getY`** that returns a reference to its **`Y`** object
- what are the values of **`sameState`** and **`sameObject`**?

```
Y y = new Y();
X x = new X(y);   // x has a reference to y
boolean sameState = y.equals(x.getY());
boolean sameObject = y == x.getY();
```

CSE1020 Review

- class **`X`** is an composition of one **`Y`**; it has a method **`getY`** that returns a reference to its **`Y`** object
- what are the likely values of **`sameState`** and **`sameObject`**?

```
Y y = new Y();
X x = new X(y);   // x uses composition with y
boolean sameState = y.equals(x.getY());
boolean sameObject = y == x.getY();
```

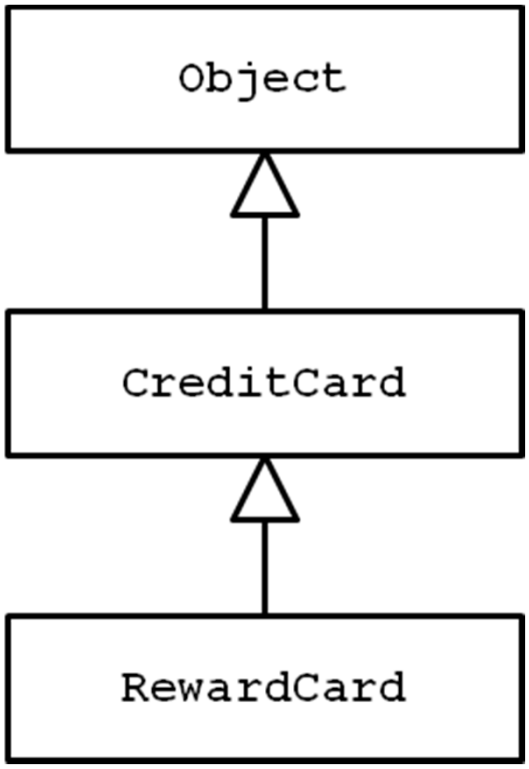
CSE1020 Review

- class **`X`** is an composition of one **`Y`**; it has a method **`getY`** that returns a reference to its **`Y`** object
  - furthermore, **`Y`** is immutable
- what are the likely values of **`sameState`** and **`sameObject`**?

```
Y y = new Y();
X x = new X(y);   // x uses composition with y
boolean sameState = y.equals(x.getY());
boolean sameObject = y == x.getY();
```

CSE1020 Review

- consider the following UML diagram



- which statements are true?
  1. **`Object`** is a **`CreditCard`**
  2. **`CreditCard`** is an **`Object`**
  3. **`RewardCard`** is an **`Object`**
  4. **`RewardCard`** is a **`CreditCard`**
  5. a **`CreditCard`** is usable anywhere a **`RewardCard`** is required
  6. a **`RewardCard`** is usable anywhere a **`CreditCard`** is required

CSE1020 Review

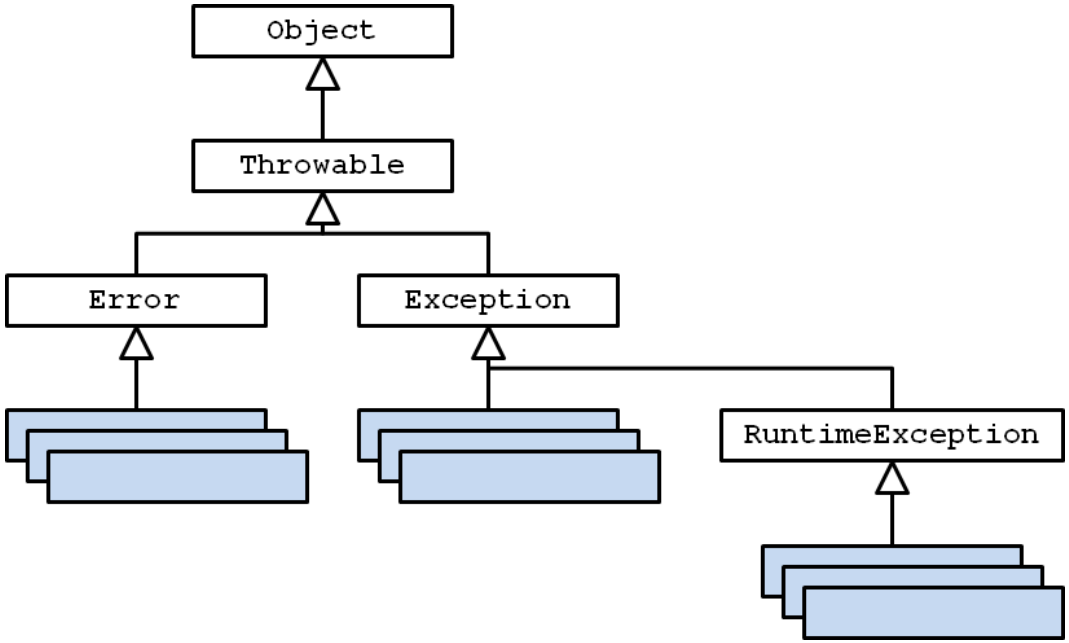
- **`t`** is a reference to a **`List<String>`** object
- write some code that prints out each element of **`t`**

CSE1020 Review

- **`p`** is a reference to a **`Map<String, Integer>`** object
- write some code that prints out each key-value pair of **`p`**

CSE1020 Review

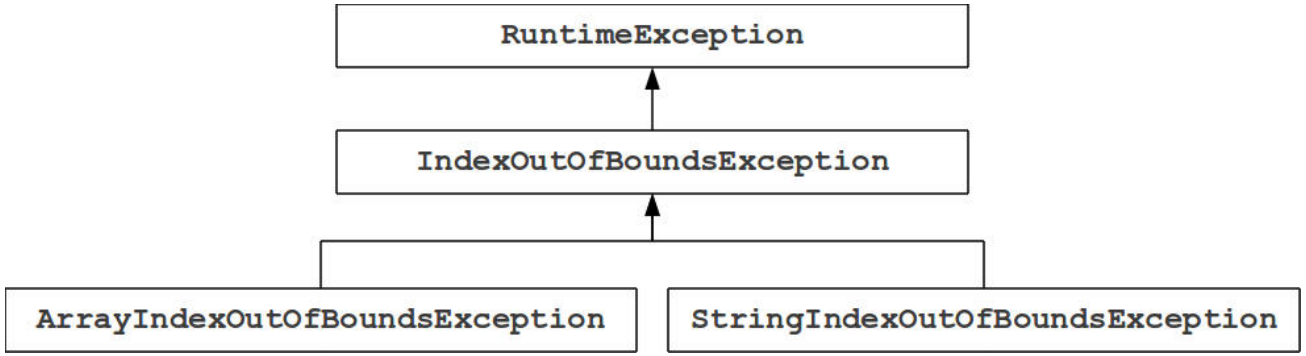
- consider the UML diagram for Java exceptions:



- checked exceptions are subclasses of … ?
- unchecked exceptions are subclasses of … ?

CSE1020 Review

- consider the UML diagram for some common exceptions:



- will the following code fragment compile?

```
try { // some legal code not shown here }
catch (IndexOutOfBoundsException e) { // not shown }
catch (StringIndexOutOfBoundsException e) { // not shown }
```

CSE1020 Review

- more questions can be found here:
  - http://www.eecs.yorku.ca/course_archive/2011-12/F/1020/practice.shtml

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

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

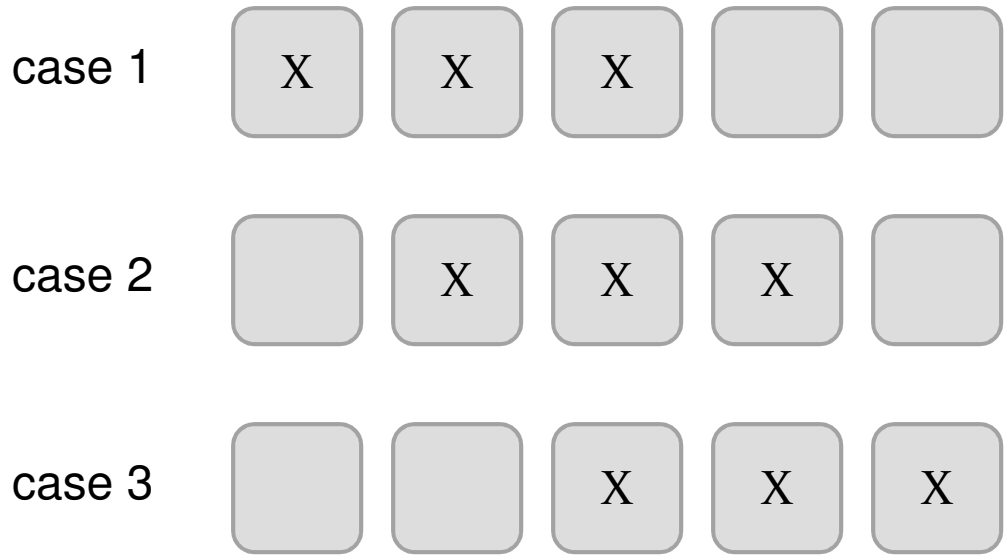
Sorting a List

- 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();
```

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
}
```

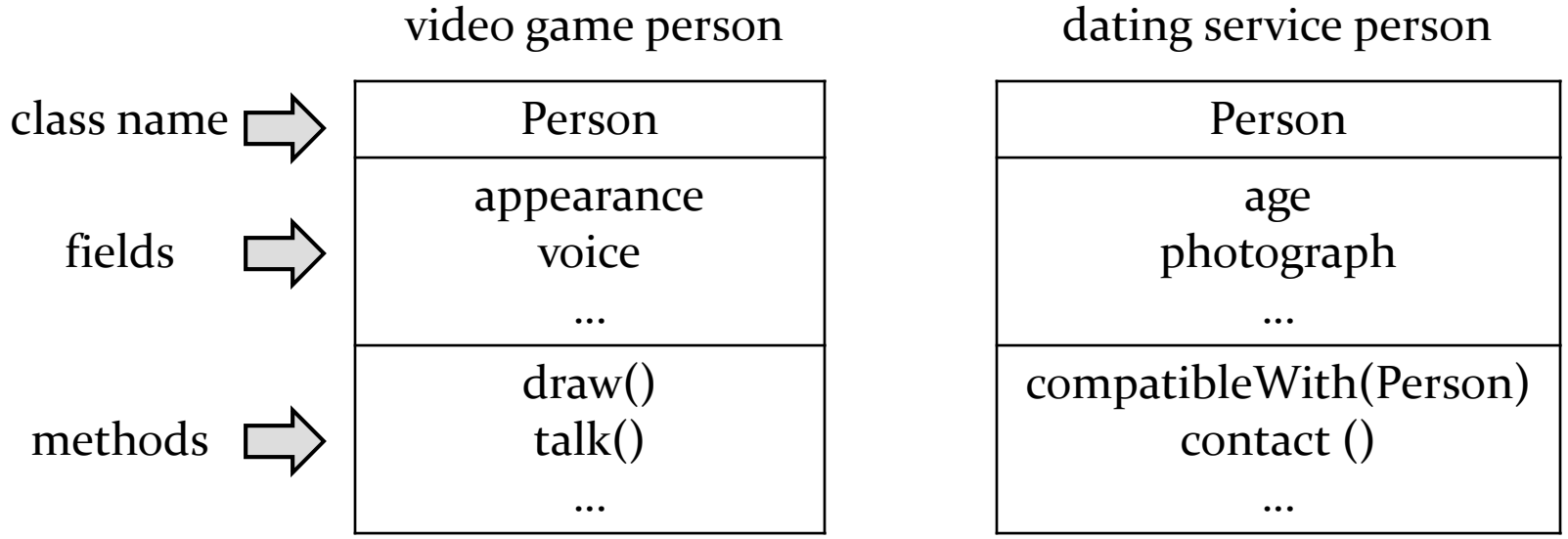
Review: Java Class

- 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

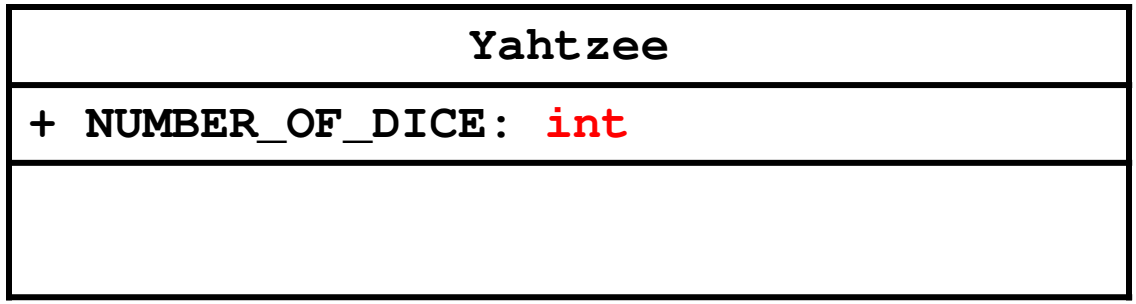
Designing a Class

- 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



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 o f **`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

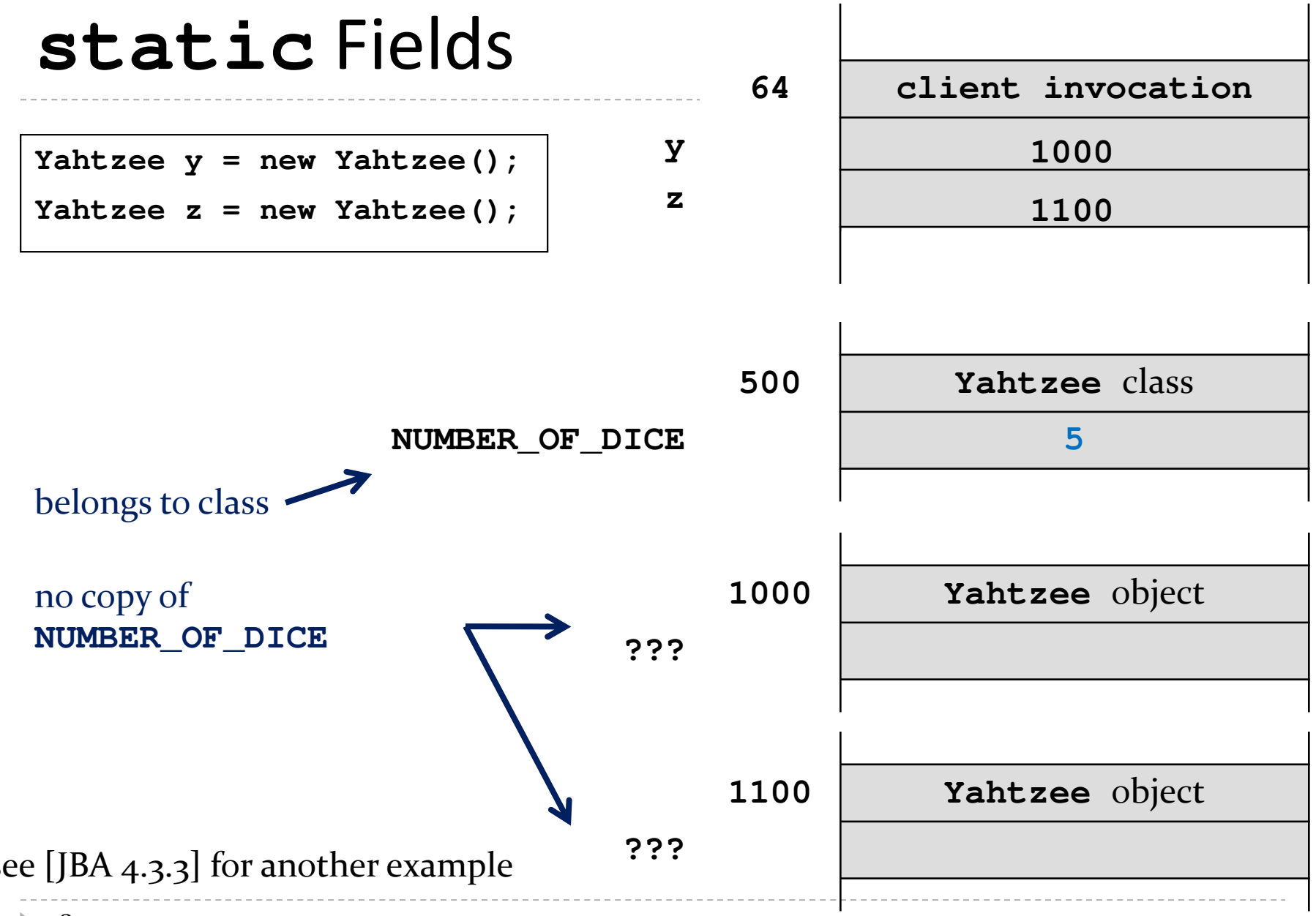
- avoid `public` fields in production code
  - except when you want to expose constant value types

`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;
                           // 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

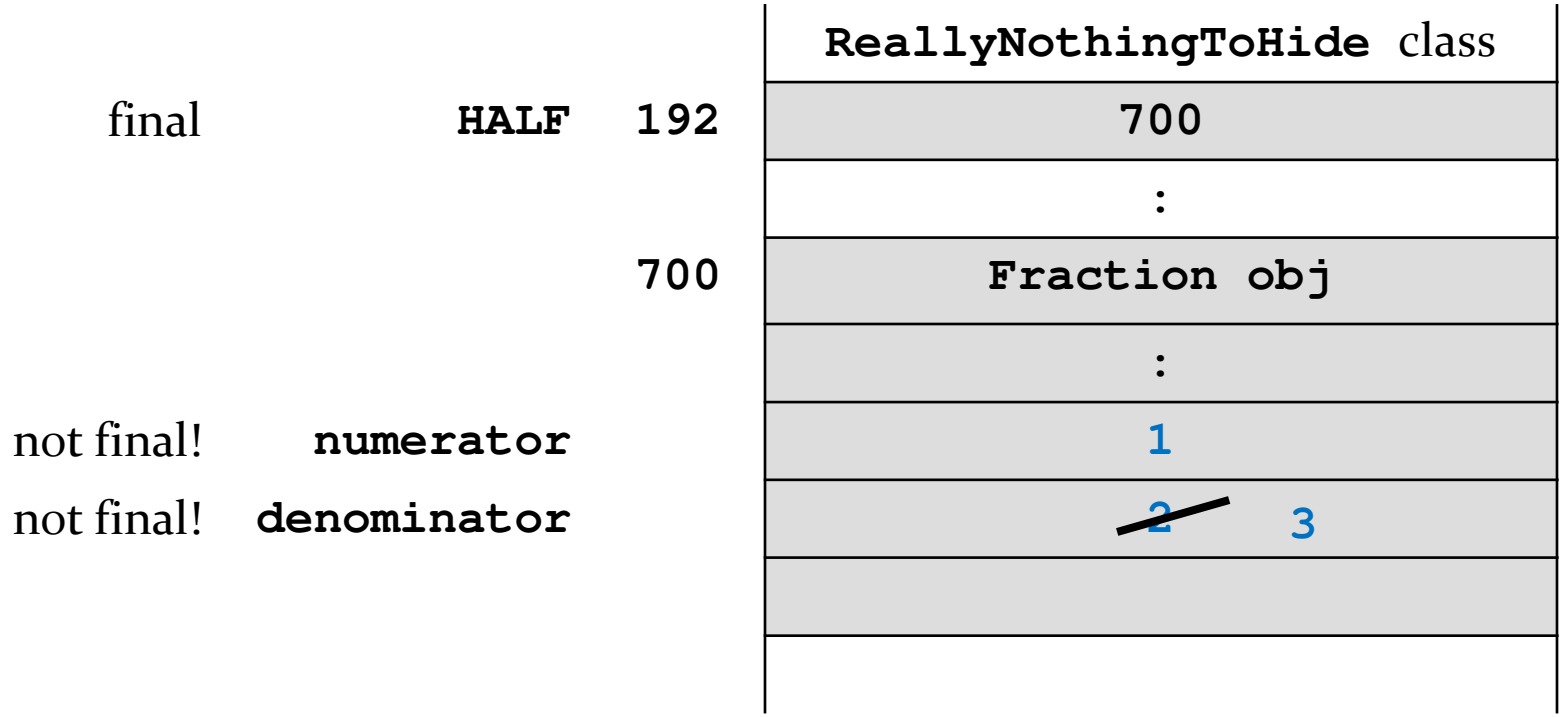
- `final` fields of mutable types are not logically constant; their state can be changed

```
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);
  }
}
```