CSE 1325: Object-Oriented Programming

Lecture 03

Encapsulation via Classes And Other Custom Types

Mr. George F. Rice

george.rice@uta.edu

Office Hours:
Prof Rice 12:30 Tuesday and Thursday in ERB 336
For TAs see this web page

What do you get when you cross a joke with a rhetorical question?

Today's Topics

- Creating Types
  - Enum
  - Class
- Classes
  - Fields
  - Constructors
  - Methods
  - Special Methods
- Unified Modeling Language

Image by ambermb from Pixabay – No attribution is required
https://pixabay.com/photos/little-boy-hiding-sad-child-fear-1635065/

A Dated Library

- Handling dates are an interesting problem
  - NOT base 10, or even base 365 – but base 12, 28-31, etc.
  - Opportunities for enum (months) and math operations
  - Complicated to validate data – is 29 Feb 2000 a valid date?
  - Various output formats (12/25, Dec 25, 25 Dec 1970, 1970.12.25, etc.)
- Dates are commonly used everyday
- Let’s first look at how we would handle dates in C++ using good *structured* programming techniques

(C++) Starting with Enum

- Since C++ allows us to define integer values for each element, let's use that feature
  - C++ enums are similar to C

```cpp
#include <iostream>

enum Month {January =  1, February =  2, March     =  3,
            April   =  4, May      =  5, June      =  6,
            July    =  7, August   =  8, September =  9,
            October = 10, November = 11, December  = 12};

 int main() {
   Month month = January;
   std::cout << "January is " << month
      << ", May is " << May
      << ", and December is " << December
      << "." << std::endl;
 }
```

```
ricegf@antares:~/dev/202108/03/code_from_slides/dateC$ g++ --std=c++17 01_enum.cpp
ricegf@antares:~/dev/202108/03/code_from_slides/dateC$ ./a.out
January is 1, May is 5, and December is 12.
ricegf@antares:~/dev/202108/03/code_from_slides/dateC$
```

(C++) Enum to String

- We need to print out the names of months, not just the numbers
- How about an array?

```cpp
#include <iostream>

enum Month {January =  1, February =  2, March     =  3,
            April   =  4, May      =  5, June      =  6,
            July    =  7, August   =  8, September =  9,
            October = 10, November = 11, December  = 12};
std::string to_string[12] = {"Jan", "Feb", "Mar", "Apr", "May", "Jun",
                             "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};
int main() {
  Month month = January;
  std::cout << "January is " << to_string[month]
     << ", May is " << to_string[May]
     << ", and December is " << to_string[December]
     << "." << std::endl;
 }
```

```
ricegf@antares:~/dev/202108/03/code_from_slides/dateC$ g++ --std=c++17 02_enum_to_string.cpp
ricegf@antares:~/dev/202108/03/code_from_slides/dateC$ ./a.out
January is Feb, May is Jun, and December is .
ricegf@antares:~/dev/202108/03/code_from_slides/dateC$
```

Wait… THAT'S not right. Let's try again!

(C++) Enum to String – Skipping 0

- We need to skip the 0th month
  - (Who gets to maintain this code?)

```cpp
#include <iostream>

enum Month {January =  1, February =  2, March     =  3,
            April   =  4, May      =  5, June      =  6,
            July    =  7, August   =  8, September =  9,
            October = 10, November = 11, December  = 12};
std::string to_string[13] = {"", "Jan", "Feb", "Mar", "Apr", "May", "Jun",
                                 "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};
int main() {
  Month month = January;
  std::cout << "January is " << to_string[month]
     << ", May is " << to_string[May]
     << ", and December is " << to_string[December]
     << "." << std::endl;
 }
```

```
ricegf@antares:~/dev/202108/03/code_from_slides/dateC$ g++ --std=c++17 03_enum_to_string_reprise.cpp
ricegf@antares:~/dev/202108/03/code_from_slides/dateC$ ./a.out
January is Jan, May is May, and December is Dec.
ricegf@antares:~/dev/202108/03/code_from_slides/dateC$
```

Looks a little odd, but it works!

(C++) Adding Day and Year

Now we need the day of the month and the year.
How about a struct?

```
// … as before

struct Date {
    int year;
    Month month;
    int day;
};

int main() {
    Date birthday{1950, December, 30};  // Not mine!
    std::cout << birthday.month << '/'
              << birthday.day   << '/'
              << birthday.year  << std::endl;
}
```

OK, we're getting there – but changing a date to a string looks like a function!

```
ricegf@antares:~/dev/202108/03/code_from_slides/dateC$ g++ --std=c++17 04_day_and_year.cpp
ricegf@antares:~/dev/202108/03/code_from_slides/dateC$ ./a.out
12/30/1950
ricegf@antares:~/dev/202108/03/code_from_slides/dateC$
```

(C++) Date to String

Now we need the day of the month and the year.
How about a struct?

```
// … as before

struct Date {
    int year;
    Month month;
    int day;
};
std::string date_to_string(Date date) {
    return std::to_string(date.year) + " "
         +      to_string[date.month] + " "
         + std::to_string(date.day);
}
int main() {
    Date birthday{1950, December, 30};  // Not mine!
    std::cout << date_to_string(birthday)  << std::endl;
}
```

Looking better, although to_string is oddly asymmetrical. Let's define a couple of additional dates!

```
ricegf@antares:~/dev/202108/03/code_from_slides/dateC$ g++ --std=c++17 05_date_to_string.cpp
ricegf@antares:~/dev/202108/03/code_from_slides/dateC$ ./a.out
1950 Dec 30
ricegf@antares:~/dev/202108/03/code_from_slides/dateC$
```

(C++) Date to String

Welcome to the Space Age!

```
// … as before

int main() {
    Date space{1961, April, 12};
    std::cout << "First human in space: "
              << date_to_string(space) << std::endl;
    ++space.day;
    std::cout << "...and the day after: "
              << date_to_string(space) << std::endl << std::endl;

    Date moon{20, July, 1969};
    std::cout << "First human on the moon: "
              << date_to_string(moon) << std::endl;
    ++moon.day;
    std::cout << "...and the day after: "
              << date_to_string(moon) << std::endl << std::endl;
}
```

Compiles great!
Runs… uh, not so well.

```
ricegf@antares:~/dev/202108/03/code_from_slides/dateC$ g++ --std=c++17 06_more_dates.cpp
ricegf@antares:~/dev/202108/03/code_from_slides/dateC$ ./a.out
First human in space: 1961 Apr 12
...and the day after: 1961 Apr 13

First human on the moon: 20 Jul 1969
...and the day after: 20 Jul 1970

ricegf@antares:~/dev/202108/03/code_from_slides/dateC$
```

(C++) Date to String

Welcome to the Space Age!

```
// … as before

int main() {
    Date space{1961, April, 12};
    std::cout << "First human in space: "
              << date_to_string(space) << std::endl;
    ++space.day;
    std::cout << "...and the day after: "
              << date_to_string(space) << std::endl << std::endl;

    Date moon{20, July, 1969};
    std::cout << "First human on the moon: "
              << date_to_string(moon) << std::endl;
    ++moon.day;
    std::cout << "...and the day after: "
              << date_to_string(moon) << std::endl << std::endl;
}
```

Bugs are easy to write and (sometimes) hard to find and fix. Help!

Data Validation – Ensuring that a program operates on clean, correct and useful data.
Validation Rules – Algorithmically enforceable constraints on the correctness, meaningfulness, and security of input data

(C++) Data Validation

Welcome to the (Validated) Space Age!

```
// … as before

Date valid_date(int year, Month month, int day) {
    if (day<1 || day>31) {
        std::cerr << "ERROR: Invalid day" << std::endl;
        return Date{0, (Month)0, 0};
    }
    return Date{year, month, day};
}
int main() {
    Date space = valid_date(1961, April, 12);
    std::cout << "First human in space: "
              << date_to_string(space) << std::endl;
    ++space.day;
    std::cout << "...and the day after: "
              << date_to_string(space) << std::endl << std::endl;

    Date moon = valid_date(20, July, 1969);
    std::cout << "First human on the moon: "
              << date_to_string(moon) << std::endl;
    ++moon.day;
    std::cout << "...and the day after: "
              << date_to_string(moon) << std::endl << std::endl;

}
```

The valid_date function detects some errors if and ONLY if we remember to use it!

(C++) Data Validation

Welcome to the (Validated) Space Age!

```
// … as before

Date valid_date(int year, Month month, int day) {
    if (day<1 || day>31) {
        std::cerr << "ERROR: Invalid day" << std::endl;
        return Date{0, (Month)0, 0};
    }
    return Date{year, month, day};
}
int main() {
    Date space = valid_date(1961, April, 12);
```

The valid_date function detects some errors if and ONLY if

```
ricegf@antares:~/dev/202108/03/code_from_slides/dateC$ g++ --std=c++17 07_data_validation.cpp
ricegf@antares:~/dev/202108/03/code_from_slides/dateC$ ./a.out
First human in space: 1961 Apr 12
...and the day after: 1961 Apr 13

ERROR: Invalid day
First human on the moon: 0  0
...and the day after: 0  1

ricegf@antares:~/dev/202108/03/code_from_slides/dateC$
```

```
    std::cout << "...and the day after: "
              << date_to_string(moon) << std::endl << std::endl;

}
```

The Problem

- Structs with public data are dangerous
  - A variable can be created with invalid data
  - The data can be changed by any code after creation
  - The data must be validated with each use, since we can never be sure if it's valid or not
- Data validation helps – *if we use it*
  - Wouldn't it be nice if we could FORCE everyone to use it?
  - Wouldn't it be nice if we could ABORT creating the struct if the data were invalid?
  - Wouldn't it be nice if we could PREVENT changes to the data unless we validate them first?

The Solution

- We prevent constructing a new struct EXCEPT through our data validator
  - We'll call it the "constructor" since it constructs
  - If the data is invalid, we'll abort by "throwing an exception", in which case no struct is created at all
- Once created, the struct fields are inaccessible by any code outside the struct
  - We call the fields "private" – for struct member eyes only
  - Only functions that are a member of the struct (we'll call them "methods") may modify or even *see* its fields
  - We can also declare some methods to be private
- Our struct has metamorphosed into a class

Definition of Encapsulation

- Object-Oriented Programming =

Encapsulation – Bundling data and code into a restricted container

We'll cover these later!

+ **Inheritance** – Reuse and extension of fields and method implementations from another class

+ **Polymorphism** – The provision of a single interface to multiple derived classes, enabling the same method call to invoke different derived methods for different results

- With encapsulation we limit access to our *fields* (data structures) via associated *methods* (functions)

NOTE: The UML calls fields "**attributes**", while C++ experts call them "**class variables**". Java and I call them "**fields**".

The UML calls methods "**operations**", while C++ experts call them "**class functions**". Java and I call them "**methods**".

C++ and Java Both Support Encapsulation

- This class was previously taught using C++ alone
  - C++ is *very* fast and *very* flexible
  - With great power comes great responsibility
  - C++ has a LOT of “sticky edges” to avoid

- Java sacrifices speed & flexibility for *safety*
  - It was designed for and (almost) *requires* proper encapsulation and good object-oriented code
  - It is among the most popular languages for the real world *because* it offers this
- We’ll switch back to Java now

Creating our Own Types

- Java gives us useful pre-defined types
  - int, double, boolean, char
- Java defines additional useful types in libraries
  - String, LinkedList, HashMap, Deque (that's a stack)
  - These encapsulate private data and also provide methods
- Java allows us to define our own general types – *classes*
  - We declare the data that will be stored (*fields*), and whether that data is visible outside the class or not (their *visibility*)
  - We declare the constructors and the methods that validate and manipulate the fields as well as the constructor and method visibilities
- Java allows us to define enumerated types – *enums*
  - Enums in Java (unlike C) are full-fledged data types, including fields and methods and even constructors that support the enumerated values
  - Enum types are "integer-like", e.g., they can iterate and switch

Defining an Enum in Java

- An enumeration consists of
  - A scope (e.g., “public”)
  - The keyword “enum” and a (required) name
  - The enumerated values
- As a type, a public enumeration MUST be in a separate file

Looks fairly similar to C… thus far!

Month.java

```
public enum Month {January,   February,   March,
                   April,     May,        June,
                   July,      August,     September,
                   October,   November,   December
}
```

Date.java

```
public class Date {
    public static void main(String[] args) {
        Month month = Month.January;
        System.out.println("January is " + month);
    }
}
```

```
ricegf@antares:~/dev/202108/03/code_from_slides/date1$ javac Date.java
ricegf@antares:~/dev/202108/03/code_from_slides/date1$ java Date
January is January
ricegf@antares:~/dev/202108/03/code_from_slides/date1$
```

And it *automatically* prints the enumerated name instead of an int, too!

Java Classes
Another, More Common, Custom Type

- Java classes are to C structs what
  Java enums are to C enums
- A class (and an enum – more later) may contain
  - Any number and any types of *fields*
    - Including enum and class types!
  - Any number of *constructors* to initialize the fields as needed
  - Any number of *methods* to manipulate the fields
  - A special *toString() method* that returns a String representation for the class

If you define the non-constant fields as private (and you should),
You have achieved encapsulation!

Date as a Java Class

- First we define the fields

```
public class Date {
    private int year;
    private Month month;
    private int day;
```

Fields (private)

- Fields are most commonly *private*
  - This ensures only class members have access
  - Final fields are
    - Similar to C's const variables
    - Immutable (cannot be changed except by a constructor)
    - May be public or private

Date as a Java Class

- Next we define one or more *constructors*

```
public class Date {
    private int year;
    private Month month;
    private int day;

    public Date(int year, Month month, int day) {
        this.year = year;
        this.month = month;
        this.day = day;

        if(1 > day || day > 31) throw new IllegalArgumentException(
                                   "Day must be between 1 and 31");
    }
```

Fields (private)

Constructor

Don't forget Data Validation!
We will discuss throwing exceptions in Lecture 05.

- Constructors always have the same name as the class and have NO return type
  - Initialize the fields, including *data validation*
  - Perform any other setup required for the class

Date as a Java Class

- Next we define one or more *constructors*

```
public class Date {
   private int year;
   private Month month;
   private int day;

   public Date(int year, Month month, int day) {
      this.year = year;
      this.month = month;
      this.day = day;

      if(1 > day || day > 31) throw new IllegalArgumentException(
                                 "Day must be between 1 and 31");
   }
```

Fields (private)

Constructor

The constructor ALWAYS has the same name as the class in Java!

"`this.year`" means "`this` object's `year` field"

"`year`" means "the closest `year` in scope", in this case, the *parameter*.

- **`this.x = x;`** is *very* common in Java constructors!

Date as a Java Class

- Next we define the toString() *method*

```
public class Date {
    private int year;
    private Month month;
    private int day;

    public Date(int year, Month month, int day) {
        this.year = year;
        this.month = month;
        this.day = day;

        if(1 > day || day > 31) throw new IllegalArgumentException(
                                   "Day must be between 1 and 31");
    }
    @Override
    public String toString() {
        return day + " " + month + ", " + year;
    }
```

Fields (private)

Constructor

Methods

toString is a “special” method, called by print and println (and everybody else!) to convert the object to a string.

The required* @override ”annotation” will be covered later.

  - This defines the default String representation for the class

* By the professor, not by Java

Date as a Java Class

For the main class, also define main() (like C/C++'s main function)

```
public class Date {
    private int year;
    private Month month;
    private int day;

    public Date(int year, Month month, int day) {
        // Java does NOT use initialization lists
        this.year = year;
        this.month = month;
        this.day = day;

        if(1 > day || day > 31) throw new IllegalArgumentException(
                                    "Day must be between 1 and 31");
    }

    @Override
    public String toString() {
        return day + " " + month + ", " + year;
    }

    public static void main(String[] args) {
        Date birthday =
            new Date(1950, Month.December, 30);
        System.out.println(birthday);
    }
}
```

```
ricegf@antares:~/dev/202108/03/code_from_slides/date3$ javac Date.java
ricegf@antares:~/dev/202108/03/code_from_slides/date3$ java Date
30 December, 1950
ricegf@antares:~/dev/202108/03/code_from_slides/date3$
```

Methods

main is a "special" method that is also the "entry point" for running the class. It is similar to the main function in C, except that it does not return an int (use System.exit(-1) to return an error code) and the (single) parameter `args` is required.

Using the Date Class

- Date may be used like any other type

```
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        int day, imonth, year;
        Month month;
        Scanner in = new Scanner(System.in);
        System.out.print("In what year were you born? ");
        year = in.nextInt();
        System.out.print("In what month were you born (1-12)? ");
        imonth = in.nextInt();
        System.out.print("On what day were you born? ");
        day = in.nextInt();

        Date birthday = new Date(year, Month.values()[imonth-1], day);
        System.out.println("Your birthday is " + birthday);
    }
}
```

An enum's values() method returns an array of the elements of that enum. We can select one with a subscript just as in C!

calls Date.toString()

```
ricegf@antares:~/dev/202108/03/code_from_slides/date3$ javac Main.java
ricegf@antares:~/dev/202108/03/code_from_slides/date3$ java Main
In what year were you born? 1992
In what month were you born (1-12)? 12
On what day were you born? 25
Your birthday is 25 December, 1992
ricegf@antares:~/dev/202108/03/code_from_slides/date3$
```

Data Validation

- (Most) invalid dates are now rejected

```
ricegf@antares:~/dev/202108/03/code_from_slides/date3$ javac Main.java
ricegf@antares:~/dev/202108/03/code_from_slides/date3$ java Main
In what year were you born? 1921
In what month were you born (1-12)? 15
On what day were you born? 15
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 14 out of bounds for length 12
        at Main.main(Main.java:15)
ricegf@antares:~/dev/202108/03/code_from_slides/date3@ java Main
In what year were you born? 1921
In what month were you born (1-12)? 3
On what day were you born? 35
Exception in thread "main" java.lang.IllegalArgumentException: Day must be between 1 and 31
        at Date.<init>(Date.java:10)
        at Main.main(Main.java:15)
ricegf@antares:~/dev/202108/03/code_from_slides/date3@
```

- We'll discuss how to catch and properly handle those exceptions in Lecture 05
  - Aborting isn't really what users want to see!

Java Enum is Also a Class!

- A Java enum is actually a full-fledged Java class
- While C / C++ allows you to assign an int to each enum element, Java allows *any* fields and methods
  - End the enum list inside the } with a ;
  - Then define the field(s) for each element
  - Then define how to initialize each field with a constructor
  - Then define other methods to manipulate them

Adding the Month Number

- The resulting Java enum looks like this

```
public enum Month {January(1),  February(2),  March(3),
                   April(4),    May(5),       June(6),
                   July(7),     August(8),    September(9),
                   October(10), November(11),  December(12);

    // Field for the integer representing this month
    private final int monthID;

    // Constructor for setting the field
    private Month(int monthID) {
        this.monthID = monthID;
    }
    // Method that returns the associated month ID for a month
    public int asInt() {
        return monthID;
    }
}
```

Elements

The enum list must terminate with a ; but only if other members follow

The value for the enum is stored in this *field*

The *constructor* sets up the field(s)

This *method* (*getter*) returns the field

Month.java

```
public class Date {
    public static void main(String[] args) {
        for(Month month : Month.values()) {
            System.out.println(month + " (" + month.asInt() + ")");
        }
    }
}
```

Date.java

Unlike C/C++, Java can iterate over all enum values with a for-each!

Adding the Month Number

- The resulting Java enum looks like this

```
public enum Month {January(1),   February(2),   March(3),
                   April(4),     May(5),         June(6),
                   July(7),      August(8),      September(9),
                   Octob
```

Elements

```
    // Field for the int
    private final int mo

    // Constructor for s
    private Month(int mo
        this.monthID = m
    }
    // Method that retur
    public int asInt() {
        return monthID;
    }
}
```

The enum list must terminate with a ; but only if other

```
ricegf@antares:~/dev/202108/03/code_from_slides/date2$ javac Date.java
ricegf@antares:~/dev/202108/03/code_from_slides/date2$ java Date
January (1)
February (2)
March (3)
April (4)
May (5)
June (6)
July (7)
August (8)
September (9)
October (10)
November (11)
December (12)
ricegf@antares:~/dev/202108/03/code_from_slides/date2$
```

Date.java

```
public class Date {
    public static void main(String[] args) {
        for(Month month : Month.values()) {
            System.out.println(month + " (" + month.asInt() + ")");
        }
    }
}
```

Unlike C/C++, Java can iterate over all enum values with a for-each!

Adding the Gemstone

- An enum (like a class) may hold any number of fields

```
// Each enumerated value calls the constructor below with its parameter
public enum Month {January(1,"garnet"),      February(2,"amethyst"),
                   March(3,"aquamarine"),    April(4,"diamond"),
                   May(5,"emerald"),         June(6,"pearl"),
   Elements        July(7,"ruby"),           August(8,"peridot"),
                   September(9,"sapphire"),  October(10,"tourmaline"),
                   November(11,"citrine"),   December(12,"tanzanite");

    // Field for the integer representing this month
    private final int monthID;
    private final String gem;

    // Constructor for setting the field
    private Month(int monthID, String gem) {
        this.monthID = monthID;
        this.gem = gem;
    }
    // Method that returns the associated month ID for a month
    public int asInt() {
        return monthID;
    }
    // Method that returns the associated month's gemstone
    public String asGemstone() {
        return gem;
    }
}
```

Month.java

Adding the Gemstone

- An enum (like a class) may hold any number of fields

```
// Each enumerated value calls the constructor below with its parameter
public enum Month {January(1,"garnet"),      February(2,"amethyst"),
                   March(3,"aquamarine"),    April(4,"diamond"),
                   May(5,"emerald"),         June(6,"pearl"),
                   July(7,"ruby"),           August(8,"peridot"),
                   September(9,"sapphire"),  October(10,"tourmaline"),
                   November(11,"citrine"),   December(12,"tanzanite");

    // Field for the integer representing this month
    priv
    priv

    // C
    priv

    }
    // M
    publ

    }
    // M
    publ
        return gem;
    }
}
```

Elements

```
ricegf@antares:~/dev/202108/03/code_from_slides/date2g$ javac Date.java
ricegf@antares:~/dev/202108/03/code_from_slides/date2g$ java Date
January's gem is garnet
February's gem is amethyst
March's gem is aquamarine
April's gem is diamond
May's gem is emerald
June's gem is pearl
July's gem is ruby
August's gem is peridot
September's gem is sapphire
October's gem is tourmaline
November's gem is citrine
December's gem is tanzanite
ricegf@antares:~/dev/202108/03/code_from_slides/date2g$
```

Default String Value for Enum

```
// Each enumerated value calls the constructor below with its parameter
public enum Month {January(1,"garnet"),     February(2,"amethyst"),
                   March(3,"aquamarine"),   April(4,"diamond"),
                   May(5,"emerald"),        June(6,"pearl"),
   Elements        July(7,"ruby"),          August(8,"peridot"),
                   September(9,"sapphire"), October(10,"tourmaline"),
                   November(11,"citrine"),  December(12,"tanzanite");
    // Field for the integer representing this month
    private final int monthID;
    private final String gem;

    // Constructor for setting the field
    private Month(int monthID, String gem) {
        this.monthID = monthID;
        this.gem = gem;
    }
    @Override
    public String toString() {
        return this.name() + " (" + monthID + "," + gem + ")";
    }
}
```

Month.java

As with classes, method toString is invoked when an enum variable is referenced in a String context such as println.
It converts the enum value to a String.

`this.name()` returns the name of this enum, for example, in `Month month = Month.May`, `month.name()` returns “May”.

```
public class Date {
    public static void main(String[] args) {
        for(Month month : Month.values()) {
            System.out.println(month);
        }
    }
```

Date.java

Default String Value for Enum

```
// Each enumerated value calls the constructor below with its parameter
public enum Month {January(1,"garnet"),     February(2,"amethyst"),
```

```
ricegf@antares:~/dev/202108/03/code_from_slides/date2toString$ javac Date.java
ricegf@antares:~/dev/202108/03/code_from_slides/date2toString$ java Date
January (1,garnet)
February (2,amethyst)
March (3,aquamarine)
April (4,diamond)
May (5,emerald)
June (6,pearl)
July (7,ruby)
August (8,peridot)
September (9,sapphire)
October (10,tourmaline)
November (11,citrine)
December (12,tanzanite)
ricegf@antares:~/dev/202108/03/code_from_slides/date2toString$
```

```
    }
}
```

Date.java

```
public class Date {
    public static void main(String[] args) {
        for(Month month : Month.values()) {
            System.out.println(month);
        }
    }
}
```

Roll from Lecture 02

Rewriting Roll with Class!

Here's the original version.
Remember this?

```
import java.util.Arrays;

public class Roll {
    public static void main(String[] args) {
        String nl = System.lineSeparator(); // System independent
        if(args.length != 2) {
            System.err.println("usage: java Roll [#dice] [#sides]");
            System.exit(-1);
        }
        int numSides = Integer.parseInt(args[1]);
        int numDice = Integer.parseInt(args[0]);

        int dice[] = new int[numDice];
        for(int i=0; i<numDice; ++i)
            dice[i] = 1 + (int) (numSides * Math.random());

        Arrays.sort(dice);
        int sum = 0;
        for(int d : dice) {
            System.out.print(" " + d);
            sum += d;
        }
        System.out.println(nl + " Sum=" + sum);
        System.out.println(" Average=" + ((double) sum / (double) numDice));
    }
}
```

Roll from Lecture 02
Rewriting Roll with Class!

- First, we *encapsulate* the die
  - The number of faces becomes a field (encapsulated *data*)
    - "**`private`**" means it can only be accessed by methods in the same class
    - "**`final`**" means it cannot be changed once initialized (like **`const`** in C)
  - The constructor **`Die`** initializes the field
  - The getter method **`getFaces`** provides read-only access to the private field
  - The method **`roll`** performs the math using the field

```
public class Die {
    public Die(int faces) {     // constructor
      this.faces = faces;
    }
    public int getFaces() {     // (getter) returns # faces set by constructor
        return faces;
    }
    public int roll() {         // roll the die and return the value
        return (int) (1 + faces * Math.random());
    }

    private final int faces;  // number of faces on this die
}
```

Roll from Lecture 02

Rewriting Roll with Class!

- Now we use a Die to calculate our results!

```
import java.util.Arrays;

public class Roll {
    public static void main(String[] args) {
        String nl = System.lineSeparator(); // System independent

        if(args.length != 2) {
            System.err.println("usage: java Roll #dice #sides");
            System.exit(-1);
        }
        int numDice = Integer.parseInt(args[0]);
        Die die = new Die(Integer.parseInt(args[1]));

        int dice[] = new int[numDice];
        for(int i=0; i<numDice; ++i)
            dice[i] = die.roll();

        Arrays.sort(dice);
        int sum = 0;
        for(int d : dice) {
            System.out.print(" " + d);
            sum += d;
        }

        System.out.println(nl + " Sum=" + sum);
        System.out.println(" Average=" + ((double) sum / (double) numDice));
    }
}
```

Roll from Lecture 02

Reusing Die for Other Games

```
import java.util.Scanner;

public class HighLow {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);// Instance a Scanner to read guesses

        Die d4  = new Die(4);          Die d6  = new Die(6);
        Die d8  = new Die(8);          Die d12 = new Die(12);
        Die d20 = new Die(20);         // Instance 5 Platonic solids

        // Calculate the sum of one roll each
        int sum = d4.roll() + d6.roll() + d8.roll() + d12.roll() + d20.roll();

        System.out.print(  // Show the instructions
            "I've rolled one of each Platonic solid - 4, 6, 8, 12, and 20 sides.\n"
          + "Try to guess the sum!\nGuess:  ");

        int guesses = 0;       // Number of guesses thus far
        int guess = 0;         // The current guess
        while(guess != sum) { // Accept guesses until correct
            guess = in.nextInt(); ++guesses;
            if(guess > sum)       System.out.print  ("Lower:  ");
            else if(guess < sum)  System.out.print  ("Higher: ");
            else                  System.out.println("Exactly!");
        }
        System.out.println("You guessed " + guesses + " times.");
    }
}
```

Classes are the Key to OOP

- The **class** is fundamental to object-oriented programming
  - A class usually directly represents a concept in a program
    - A **physical item** – candy bars in a vending machine,
      products on Amazon.com,
      players in a football simulation
    - A **logical item** – debits and credits on an accounting ledger,
      positions in 3D space in the solar system,
      mathematical concepts like complex numbers
  - **A class is a user-defined type**
    - You create *variables* to refer to its instances, as with ints and doubles
    - You may use *generics* such as ArrayList<> to store them, just like generics store Integer and Double and String values
    - Its functionality may be **extended via inheritance** without modifying the class itself (coming in Lecture 07!)

Definition of Encapsulation

- Object-Oriented Programming =

Encapsulation – Bundling data and code into a restricted container

We'll cover these later!

+ **Inheritance** – Reuse and extension of fields and method implementations from another class

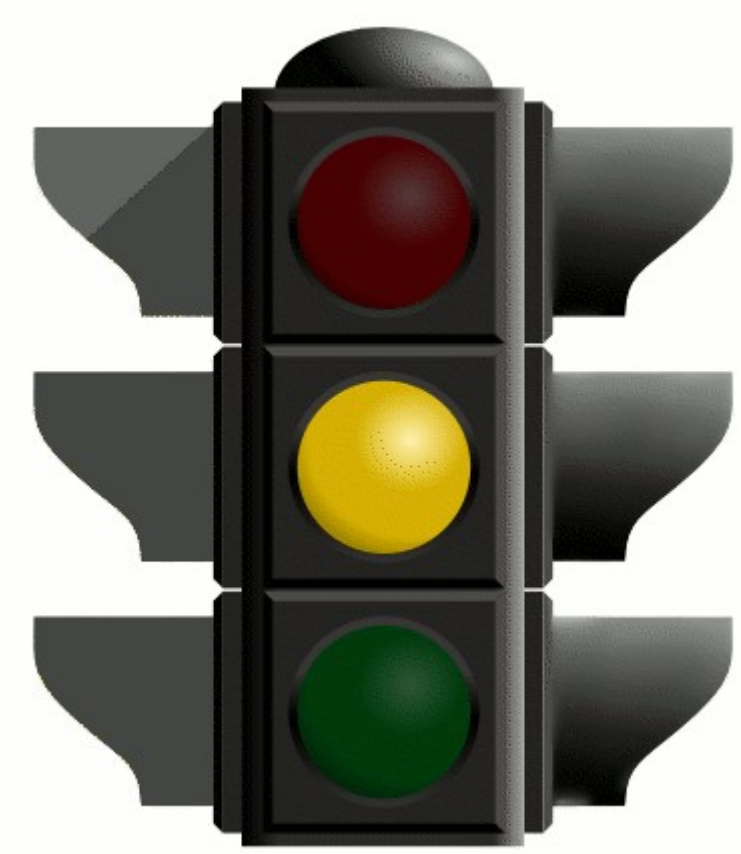
+ **Polymorphism** – The provision of a single interface to multiple derived classes, enabling the same method call to invoke different derived methods for different results

- With encapsulation we limit access to our *fields* (data structures) via associated *methods* (functions)

Pie photo by Evan-Amos [Public domain]
https://commons.wikimedia.org/wiki/File:Cherry-Pie-Slice.jpg

Consider the Traffic Light

- What data describes one face of a traffic light?
- What methods would we need to control the traffic light?
- How do we describe this so ~~everyone~~ understands?
  ok, us *professionals!*



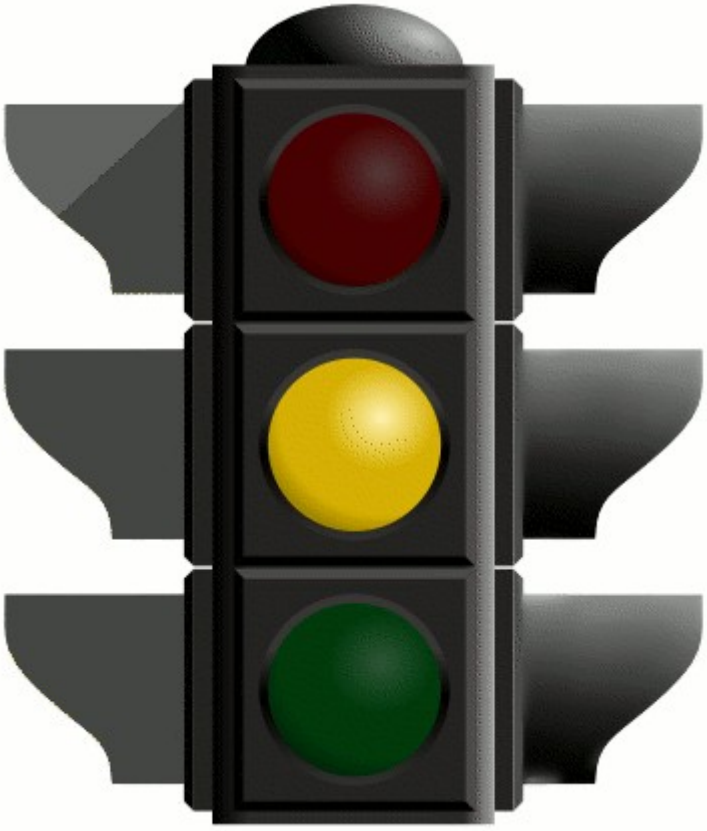
Traffic light image by TheGoTeam, CC0 / Public Domain
https://en.wikipedia.org/wiki/File:Traffic_light_yellow-766495.png

Specifying Traffic Light in English

- What data describes one face of a traffic light?
- What classes and methods would we need to control the traffic light?

English is perhaps not our best choice...

- We need a Color enum
  - green, yellow, and red values
- We need a TrafficLight class
  - 2 fields: “operating” as Boolean and “color” of type Color
  - We’ll need “getters” and toString using both fields, and a “setter” for “operating” to turn the light on and off
  - We’ll need a “nextColor” method to cycle the light



Traffic light image by TheGoTeam, CC0 / Public Domain
https://en.wikipedia.org/wiki/File:Traffic_light_yellow-766495.png

Specifying Traffic Light in Java

- In code, your interface can be difficult to quickly digest
  - In a mixed-language program, it's worse

```
public enum Color{green, yellow, red}
```

```
public class TrafficLight {
    private boolean operating;
    private Color light;
    TrafficLight() {operating = true; light = Color.red;}
    public boolean isOperating() {return operating;}
    public void power(boolean operating)
        {this.operating = operating;}
    public Color thisColor() {return light;}
    public Color nextColor() {
        light = switch(light) {
                    case green  -> Color.yellow;
                    case yellow -> Color.red;
                    case red    -> Color.green;
                };
        return light;
    }
    @Override
    public String toString() {
        return operating ? light.name() : "off";
    }
}
```

Look, a switch *expression*!

...and a ternary!

What we need is a notation in which to describe each class' interface and its relationship to other classes in our programs

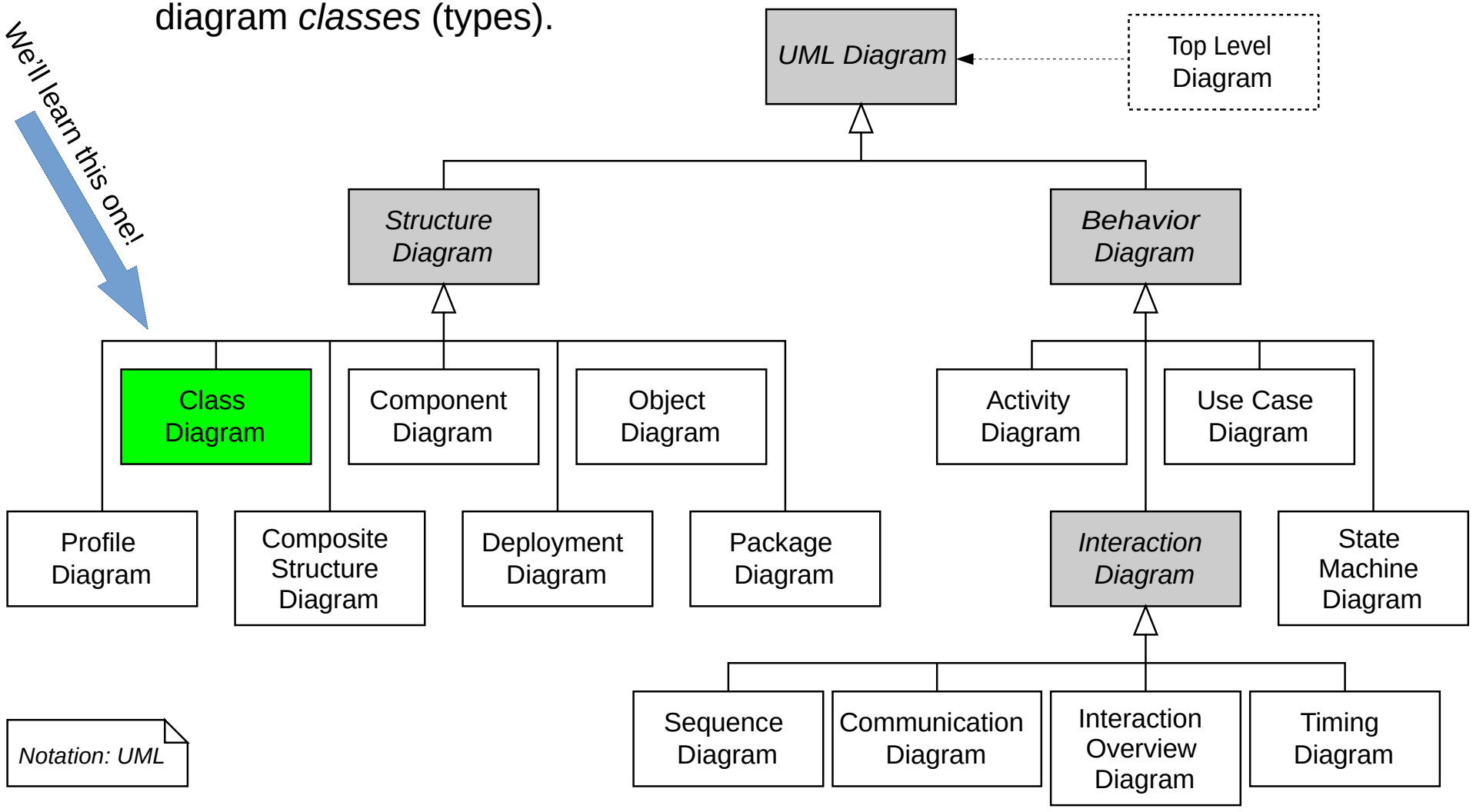
- **Non-ambiguous (clear)**
- **Standard**
- **Language-independent**
- **Graphical (people♥pics!)**

Unified Modeling Language (UML)

- The UML is the standard visual modeling language used to describe, specify, design, and document the structure and behavior of software systems, particularly OO
  - UML can **formally** specify a system so that code can be generated
  - UML can **informally** specify a system to enhance team communication
  - UML can **casually** represent a system to enhance your understanding during implementation and maintenance
- We'll cover just enough UML this semester for you to read and answer the assignment and exam questions!

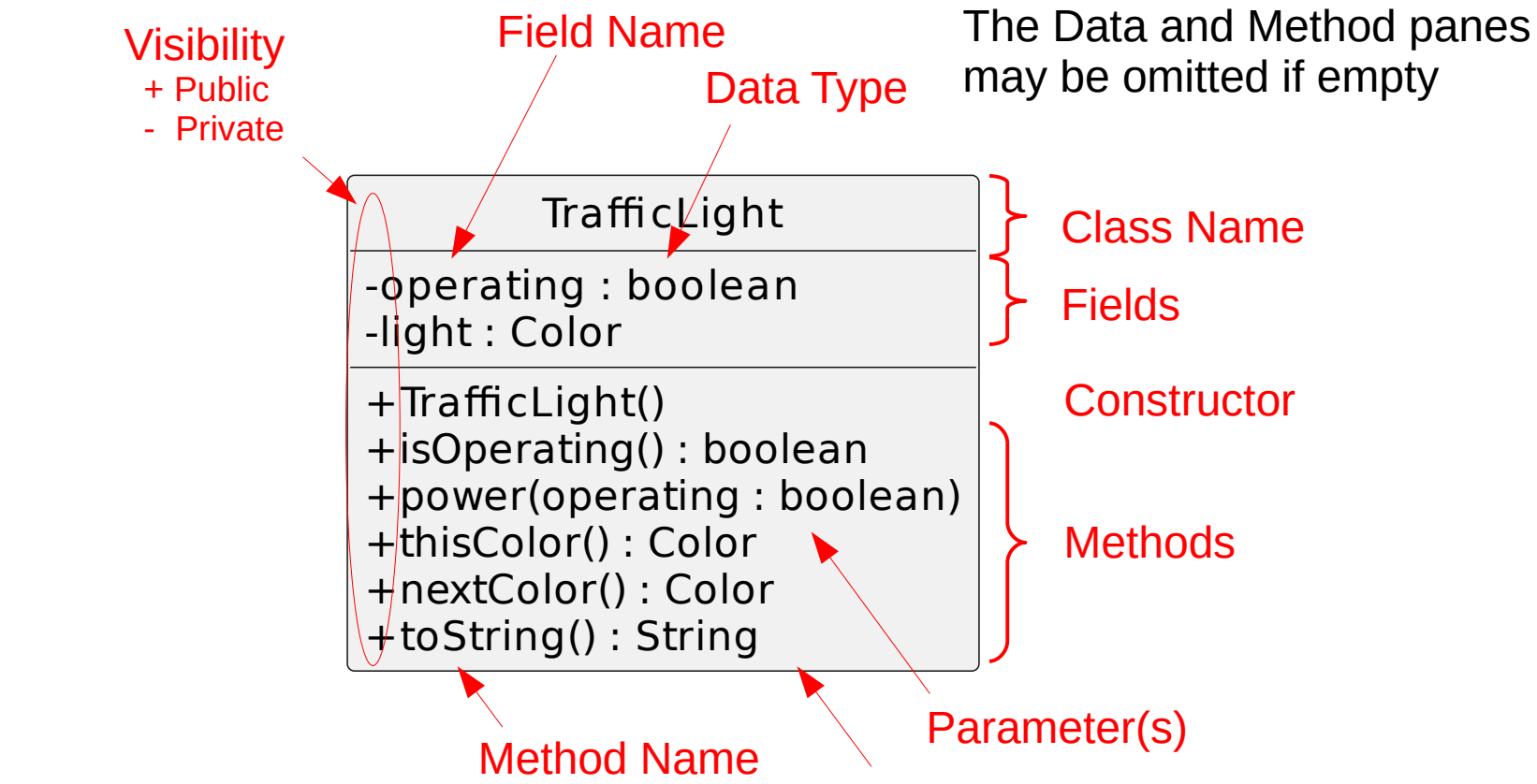
UML Includes a LOT of Diagrams

Grey boxes are
diagram *classes* (types).



Original source: Wikipedia, Public Domain SVG

UML Class Interface



Parameters are formatted like fields, e.g.,

```
+power(operating : boolean)
```

Parameter Name Parameter Type (required)

Class Interface

- Which interface description is more readily grasped?

```
public enum Color{green, yellow, red}
```

```
public class TrafficLight {
    private boolean operating;
    private Color light;

    TrafficLight() {operating = true; light = Color.red;}
    public boolean isOperating() {return operating;}
    public void power(boolean operating)
        {this.operating = operating;}
    public Color thisColor() {return light;}
    public Color nextColor() {
        light = switch(light) {
                    case green  -> Color.yellow;
                    case yellow -> Color.red;
                    case red    -> Color.green;
                };
        return light;
    }
    @Override
    public String toString() {
        return operating ? light.name() : "off";
    }
}
```

1

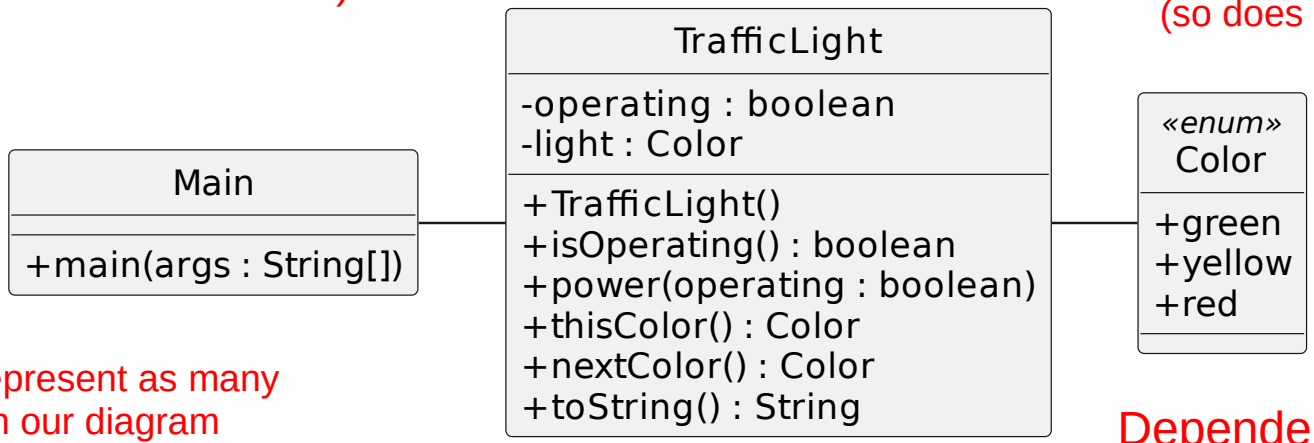
- We need a Color enum
  - green, yellow, and red values
- We need a Traffic_light class
  - 2 attributes: “operating” as Boolean and “color” of type Color
  - We’ll need “getters” and to_string for both attributes, and a “setter” for “operating” to turn the light on and off
  - We’ll need a “next_color” method to cycle the light

3

UML Also Represents Relationships
(More on Those in Lectures 06 and 07!)

Class Diagram

For Java, our main method is shown exactly where implemented.
(For C++ and Python, we often create a faux Main class to hold our main "method".)



Enum
"Fields" with no type are your clue that Color is an enum rather than a true class (so does «enum», called a "stereotype").

Class
We can represent as many classes on our diagram as we like, one per 3-paned rectangle.

Dependency association
TrafficLight depends on Color.

Note that the class diagram shows *static* but not *dynamic* (constructor / method body) info.
We'll provide that as text in the Requirements PDF until you learn more UML in later courses.

The class diagram is your implementation "battle map".
Simply (ahem) write the classes and their respective members in Java.

Creating UML Diagrams

- You won't need to draw class diagrams this semester
  - We draw'em, you code'em
- Numerous tools support UML directly
  - PlantUML **sudo apt install plantuml** http://plantuml.com/class-diagram
    - Creates diagram from simple text specification
    - Works *great* with git and scripts
  - Umbrello **sudo apt install umbrello** https://umbrello.kde.org/
    - Can auto-generate diagram *using* Java files
    - Can auto-generate Java files *from* diagram
    - Stores data in xml – works well with git and (with effort) scripts
  - Draw.io https://about.draw.io/uml-class-diagrams-in-draw-io/
    - General purpose web drawing tool similar to Visio
    - Or you can use Microsoft Visio or LibreOffice Draw

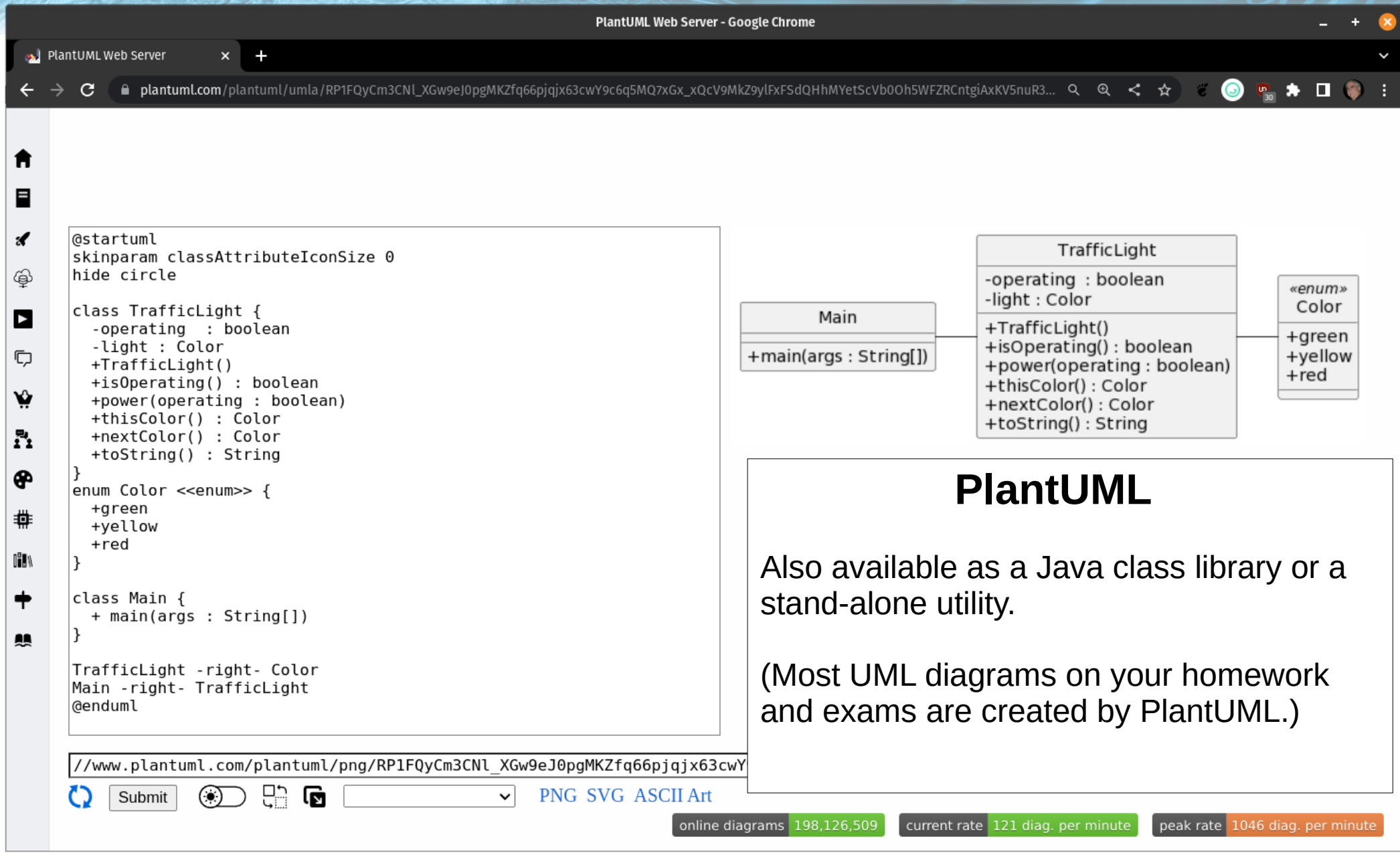
My Preference!

NOT permitted for CSE1325!

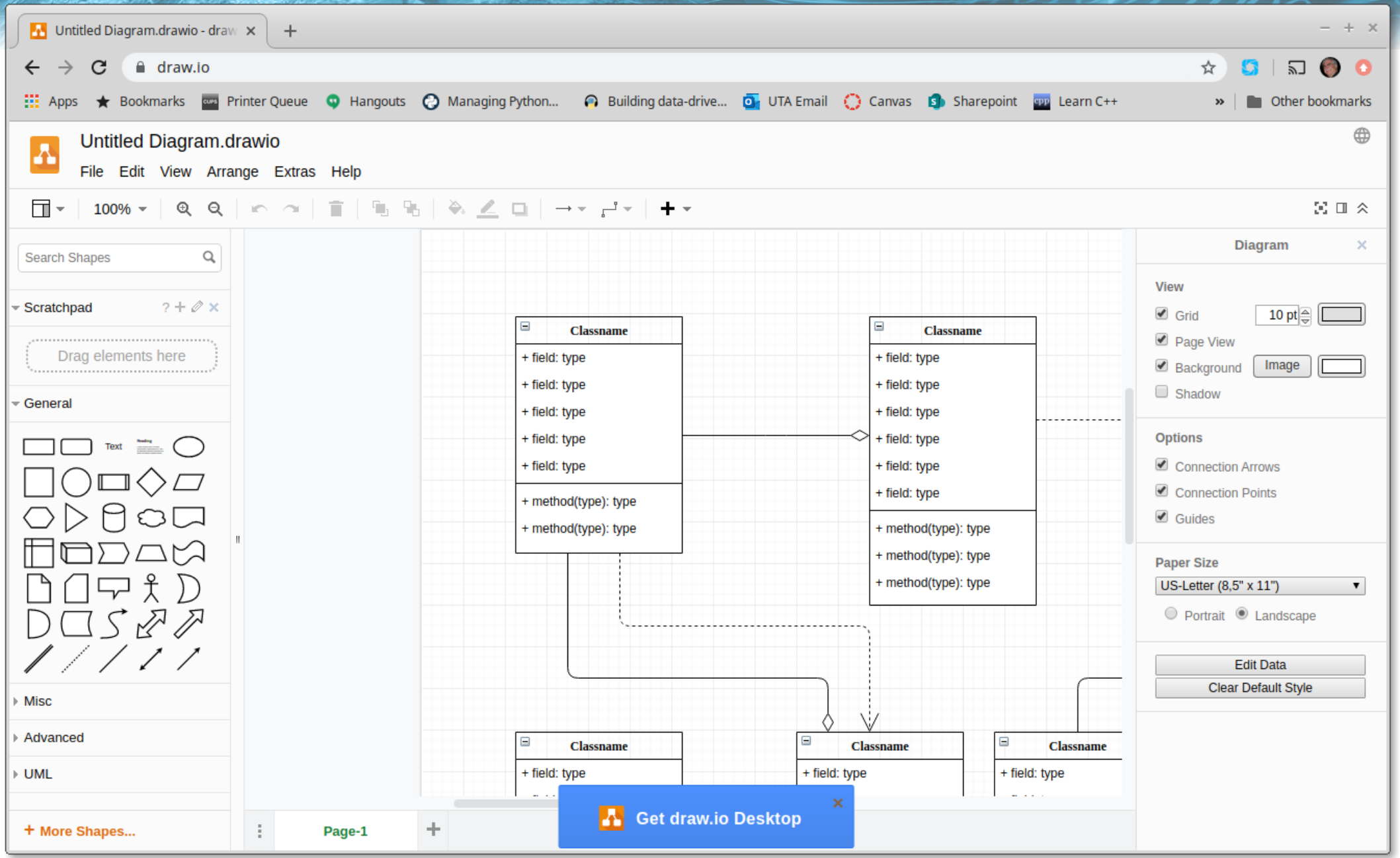
These are but a few of MANY options!

PlantUML

(The screenshot below is a link)

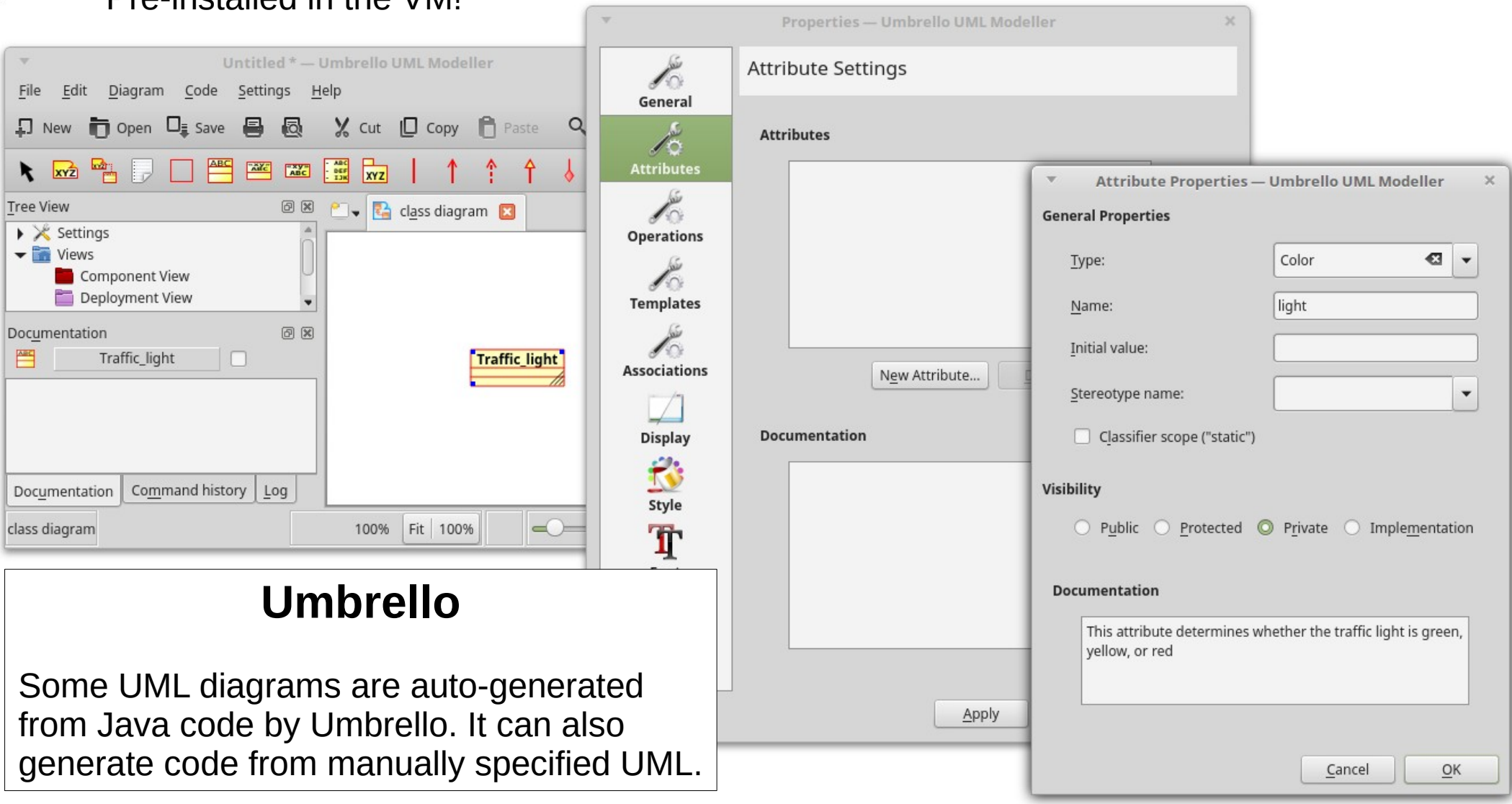


Draw.io


Untitled Diagram.drawio - draw
draw.io
Apps
Bookmarks
Printer Queue
Hangouts
Managing Python...
Building data-drive...
UTA Email
Canvas
Sharepoint
Learn C++
Other bookmarks
Untitled Diagram.drawio
File Edit View Arrange Extras Help
100%
Search Shapes
Scratchpad
Drag elements here
General
Misc
Advanced
UML
+ More Shapes...
Classname
+ field: type
+ method(type): type
Page-1
Get draw.io Desktop
Diagram
View
Grid
10 pt
Page View
Background
Image
Shadow
Options
Connection Arrows
Connection Points
Guides
Paper Size
US-Letter (8,5" x 11")
Portrait
Landscape
Edit Data
Clear Default Style

Umbrello

Pre-installed in the VM!

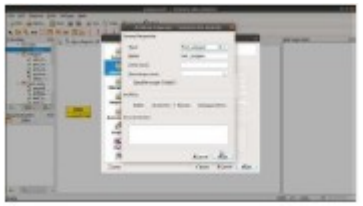


Umbrello

Some UML diagrams are auto-generated from Java code by Umbrello. It can also generate code from manually specified UML.

Learn to Create UML Class Diagrams in Umbrello

- A 13 minute screencast walks you through Umbrello class diagrams
  - Creating classes and relationships
  - Editing classes
  - Saving as XML Metadata Interchange (.xmi) files (the standard UML file format)
  - Exporting diagrams as images
- https://youtu.be/p0WtDSwwBfI



This is NOT required for CSE1325, as you are no longer required to create UML diagrams. It is still available for advanced and proactive students who came for the knowledge. :-)

What We Learned Today

- The 3 foundations of Object-Oriented Programming
  - **Encapsulation** – Controlling access to private data
  - **Inheritance** – Reusing and extending encapsulated data
  - **Polymorphism** – Dynamically selecting encapsulated behavior
- Two custom type definitions in Java
  - **Class** – Encapsulated fields, constructors, and methods
  - **Enum** – Class plus “named ints”
- Members of enum and class types in Java
  - **Elements** (enum only, with associated field values)
  - **Fields** (attributes, class variables) to encapsulate and protect data
  - **Constructors** to initialize fields and perform other setup
  - **Methods** (class functions) to manipulate the fields

Don’t forget to finish the Lecture 02 videos by Thursday!