ALGONQUIN COLLEGE

CST8224 OOP

Lecture 2 F18

Learning Outcomes this week….

1. Primitive versus reference variables.
2. Using Eclipse debugger.
3. toString().
4. Binary representation of numbers and overflow/Underflow and errors.
5. Keyword static as a modifier for fields and methods.
6. Memory maps
7. UML – Unified Modelling Language
8. printf
9. Examples

References:

UML Examples: Deitel section 1.5.11, 3.2.4, online appendix M, plus corresponding sections in

Memory Map Examples: Class notes and examples.

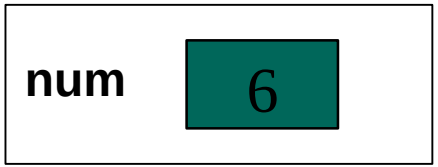
Recall Primitive Variables:

- In Java, variables of type int float double char boolean are called primitive.

- Memory is allocated *immediately* for such variables.
- Furthermore, the binary 1's and 0's in that memory location, mathematically represent the actual number.

Example: int num = 6;

- In memory, looks like:



Reference variables

- In Java, Objects of a class are also called reference variables (in other OOP such as C and C++, they are called *pointers*).
- Memory is NOT allocated immediately for such variables.

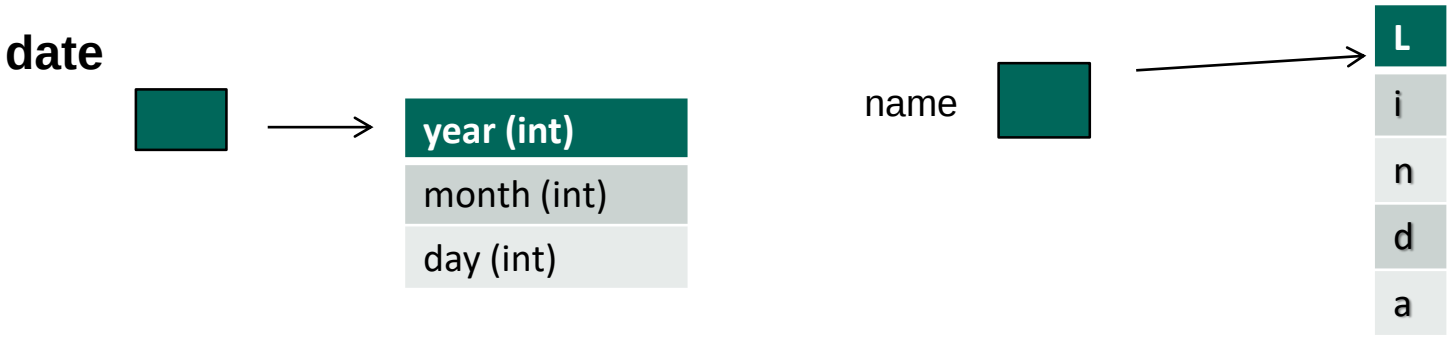
- Example: String name;
  Date date;
  - NOTE: date has no int memory (for year/month/day), and name has no chars.
- In memory, these reference variables looks like:

Reference variables

- Memory is actually allocated when keyword “**new**” is used

- Example: name = **new** String(“Linda”);
- date = **new** Date();

In memory, looks like:



What is a hashCode?

- A 32-bit signed integer.
- identityHashCode() will … "return distinct integers for distinct objects. (This is typically implemented by converting the internal address of the object into an integer." – java 8 api http://docs.oracle.com/javase/8/docs/api/index.html?overview-summary.html

```
public static void main(String[] args) {

   Patient patient1 = new Patient();

   System.out.println(patient1);
   System.out.println(System.identityHashCode(patient1));
}
```

Resulting output:

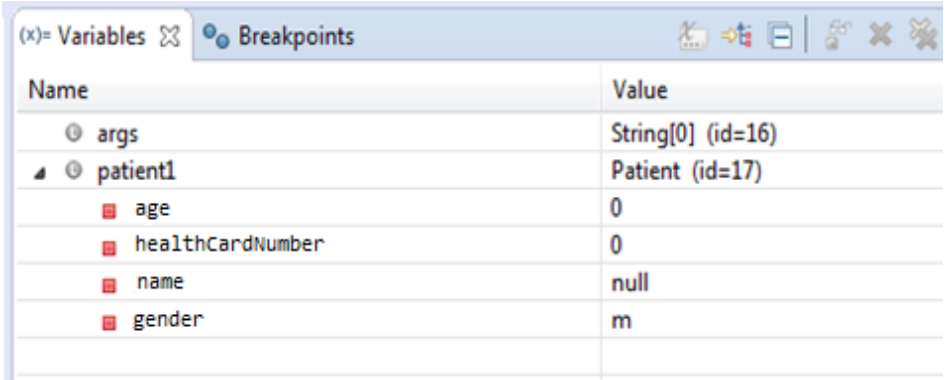
Patient@7852e922 ← **ClassName@hashcode in hexadecimal**

2018699554 ← **hashcode in decimal**

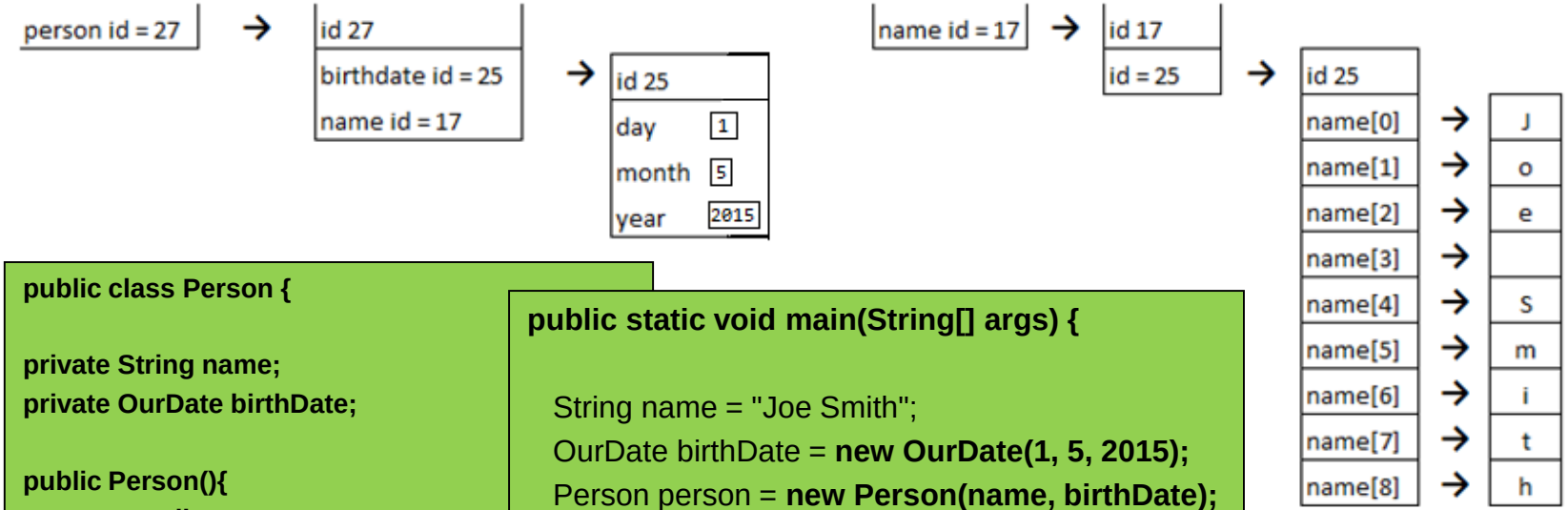
Do the math: $(7852e922)_{16} = (2018699554)_{10}$

What is the id = XX number in the Eclipse debugger?

- It is a number that is generated by the Eclipse debugger - it has *nothing* to do with the JVM.
- It is meant to be used as a tool so that reference variables may be kept track of.
- But….it can be “sort of” be thought as the memory location in the heap where the object’s fields are stored – it can be useful in the creation of memory maps.



Memory Map Example

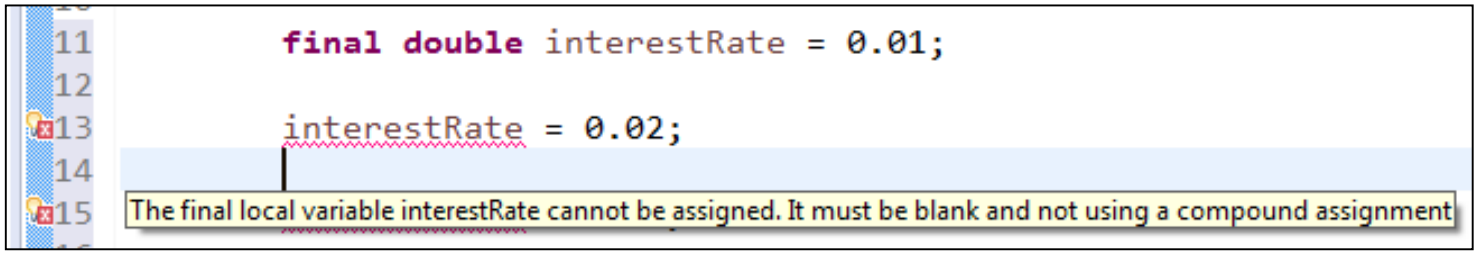



```
public class Person {

private String name;
private OurDate birthDate;

public Person(){
   name = null;
   birthDate = null;
}

public Person(String name, OurDate birthDate){
   this.name = name;
   this.birthDate = birthDate;
}
```

```
public static void main(String[] args) {

   String name = "Joe Smith";
   OurDate birthDate = new OurDate(1, 5, 2015);
   Person person = new Person(name, birthDate);
}
```

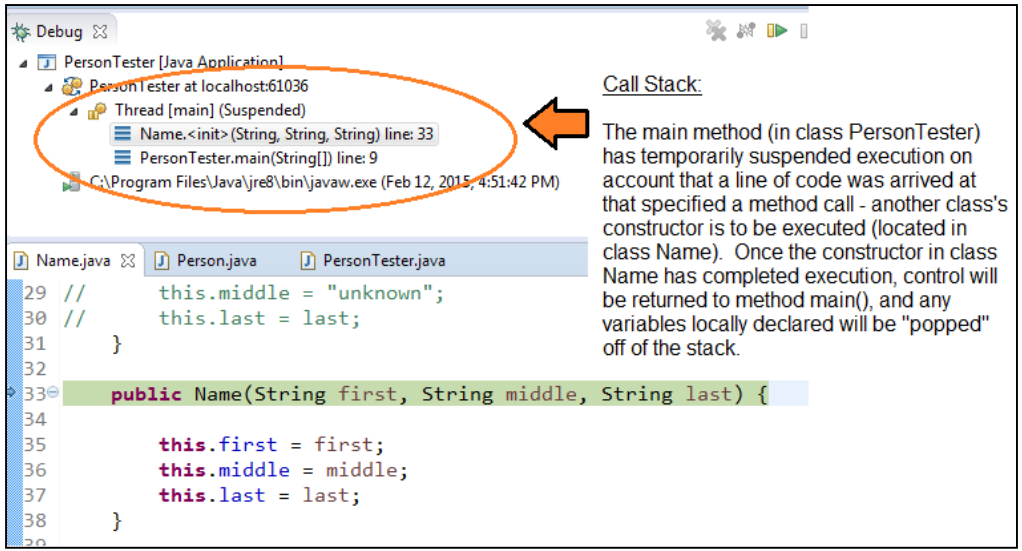
Keyword *final* as a modifier in a variable declaration

- Adding keyword *final* renders a variable into a constant.
- *final* variables must be initialized before they are used and cannot be modified thereafter.
- An attempt to modify a *final* variable after it's initialized causes a compilation error.
- Useful in preventing a variable's "accidental" modification, i.e. when specifying the size of an array (i.e. final int size = 10;).
- Examples see Deitel 6.3, 7.4, etc.



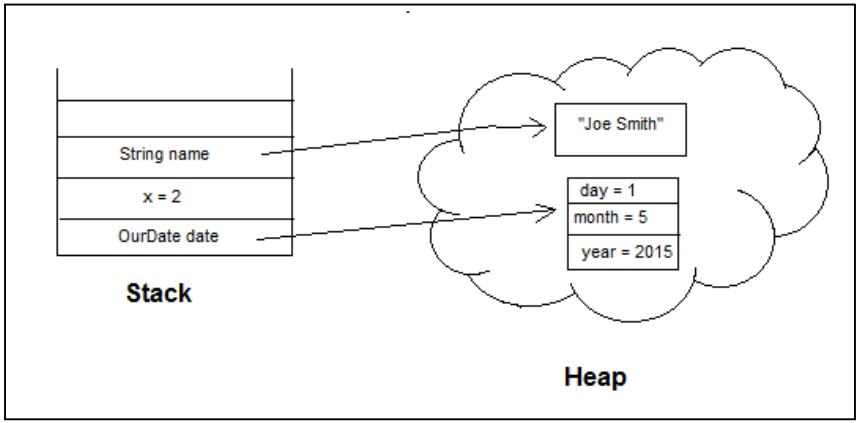
What is a stack?

- It is region of memory that stores temporary (local) variables.
- Stored in RAM therefore is faster than heap access.
- A data structure that grows and shrinks as needed - It is managed by the operating system, in this case the JVM – it does have a maximum size limit.
- In analogy, think of it as a stack of plates, where each plate is a variable - variables are “pushed” onto the stack and “popped” off of the stack.
- Stack variables only occupy memory when they are in “scope” – the method in which they are defined is currently being executed.
- Refer to Deitel chapter 6.6, 9th edition).



What is a heap?

- It is region of memory that stores variables associated with objects and JRE classes
- It is larger than the stack.
- A heap can grow as required but it does have a maximum size limit (OS dependent).
- When an object no longer has a reference to it, “garbage collection” occurs, and the memory occupied by the object’s fields is released.
- See example code.



A common way to envision stack and heap memory
Some online references include
http://stackoverflow.com/questions/79923/what-and-where-are-the-stack-and-heap

Stack Allocation of Variables versus Heap Allocation of Variables

Stack

- Local variables to methods.
- Input parameters to methods (even a reference variable).

Heap

- Anything that follows keyword “new”.
- Data fields of a class (i.e. private int x;)
- Anything following keyword this (i.e. in a method, this.x = x;)

Why?

- To manage memory: don’t want to end up with a exception: Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
- http://stackoverflow.com/questions/2381849/exception-in-thread-main-java-lang-outofmemoryerror-java-heap-space

Keyword static as a modifier for class fields

- Only ***one copy*** of the field exists per class i.e. every object of a class that is ever instantiated shares the same variable.
- Uses include keeping a count of how many instantiations of the class have occurred.

static in debug mode:

```
public class Fun  {
    private int x;
    private static int numberOfObjects = 0;
    private final int QUANTITY = 100;

    //default constructor
    public Fun(){
        x = 1;
        numberOfObjects++;
    }

    public int getNumberOfObjects(){ return numberOfObjects ; }

    public int getX(){  return x ; }
}
```

```
public static void main(String [] args) {

   Fun obj1 = new Fun();
   Fun obj2 = new Fun();
   Fun obj3 = new Fun();
}
```

obj1

this	After numberOfObjects++	Fun (id=19)
numberOfObjects	1	
QUANTITY	100	
x	1	

obj2

this	Before numberOfObjects++	Fun (id=20)
numberOfObjects	1	
QUANTITY	0	
x	0	

this	After numberOfObjects++	Fun (id=20)
numberOfObjects	2	
QUANTITY	100	
x	1	

obj3

this	Before numberOfObjects++	Fun (id=21)
numberOfObjects	2	
QUANTITY	0	
x	0	

this	After numberOfObjects++	Fun (id=21)
numberOfObjects	3	
QUANTITY	100	
x	1	

Problem:

- Write an Invoice class, with an OurDate object as a field.
- Declaring an object of one class as a field in another class is known as “composition”.
- Composition relationships are specified in a UML diagram with a line connecting the two classes.
- The class that has the object of another class will have a diamond shaped connector pointed to it (see upcoming slide).

Problem Specifications

OurDate class :

- *Three private integer fields to store year, month and day .*

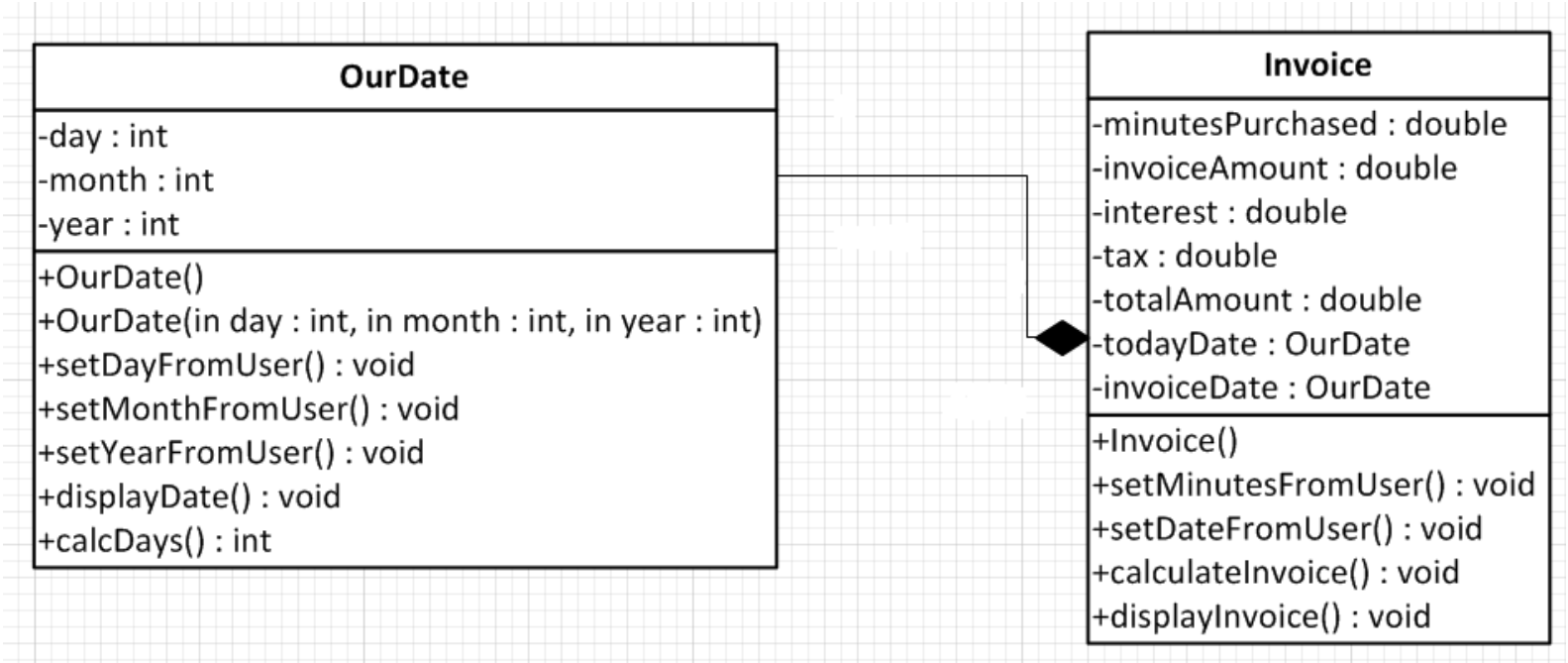
- *Methods:*
  - *Constructor – default date of Jan 1, 1900*
  - *Constructor – using three integers to set initial values for year, month and day*
  - *setYearFromUser – prompt user to enter a year*
  - *setMonthFromUser – prompt user to enter a month*
  - *setDayFromUser – prompt user to enter a day*
  - *displayDate – will display the date in yyyy/mm/dd format*
  - *calcDays()– returns an int*

Invoice class :

- *Five private float or double fields to store minutesPurchased, invoiceAmount, interest, tax, totalAmount*
- *Two private OurDate fields to store todayDate and invoiceDate.*

- *Methods:*
  - *Constructor – default amounts to 0*
  - *setMinutesFromUser – prompt user to enter cell service minutes*
  - *setDatesFromUser – prompt user to enter today's date and invoice date*
  - *calculateInvoice – calculate the invoice fields invoiceAmount, interest, tax and totalAmount*
  - *displayInvoice – display the invoice*

Problem Statement in a UML



Issues when using Composition:

Example:

```
public class  Employee {
        private float hoursWorked;
        private float rateOfPay;
        private OurDate startDate;
        private String name;
}
```

Issue: when you instantiate an object of type Employee (i.e. Employee employee = new Employee();) you need to use the constructors to allocate the memory for startDate and name. Simply declaring them as above will only initialize them to **null**).

Declaring objects of a Class in Another Class

```
public    class  Employee {
      private float hoursWorked;
      private float rateOfPay;
      private Date startDate;
      private String name;

       public Employee() {
            hoursWorked = 0.0f;
            rateOfPay =0.0f;
            startDate = new Date();
            name = new String();
      }
```

- Otherwise, you will get run time error if you try to use a reference that does not have memory allocated to it!
- Primitives don’t have this issue.

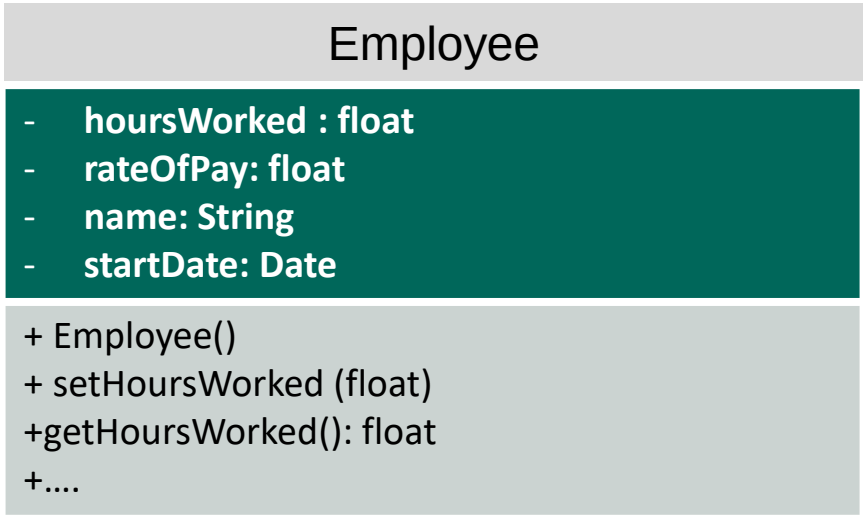
Introducing UML Diagrams

- UML – Unified Modelling Language.
- Deitel section 3.2.4: “UML class diagrams … summarize a class’s attributes and operations. In industry, UML diagrams help systems designers specify a system in a concise, graphical, programming-language-independent manner, before programmers implement the system in a specific programming language”.

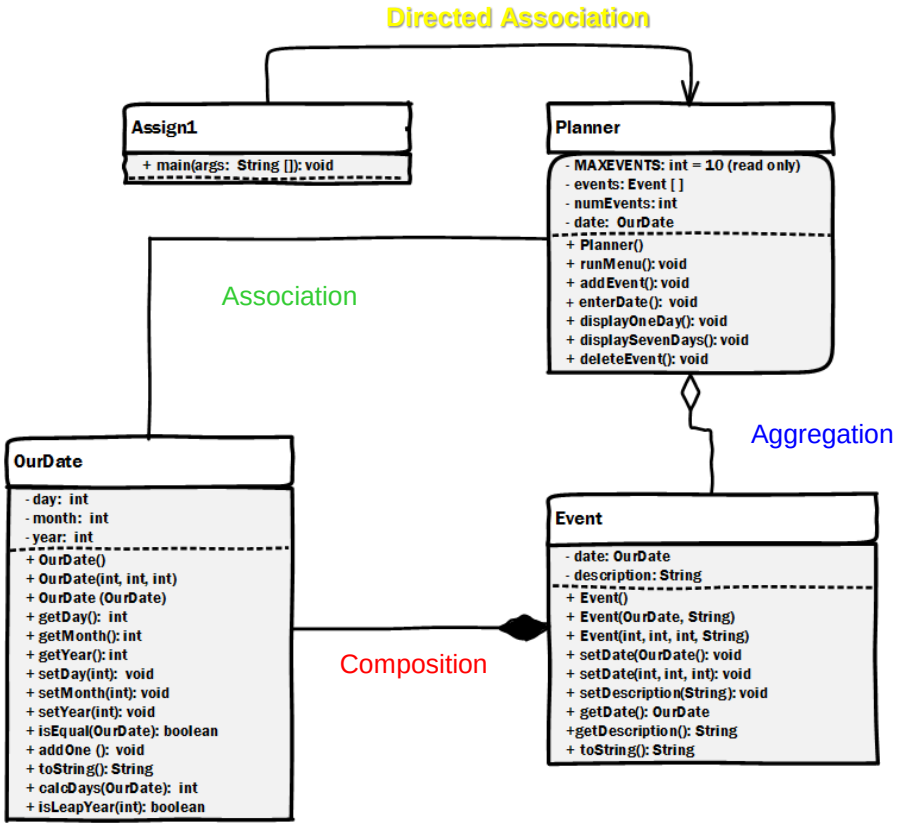
- In CST8224 we will only be ***interpreting*** basic UML diagrams.
- Next term we will be interpreting more complex UML diagrams.

UML Basics

- Used to document a class
- Two sections:
  - top section for data member
  - bottom section for methods.
  - Constructors will be listed before any other class methods.
- public is indicated with a +
- private is indicated with a –



UML Relationships



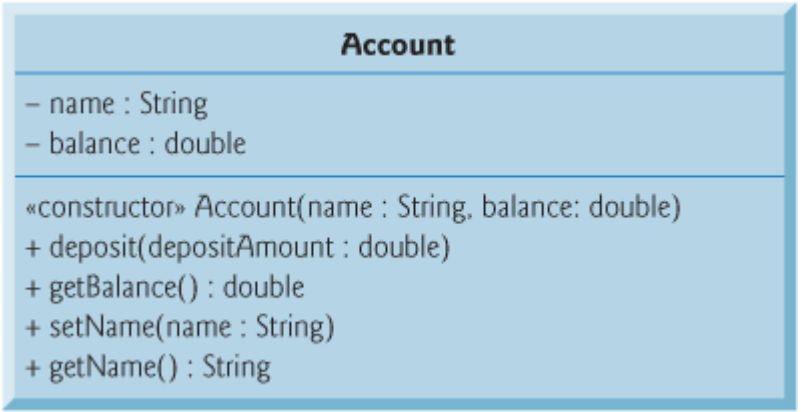
- Aggregation, composition and association all specify a type of “has a” relationship between classes.
- Initially, some type of ***association*** relationship between classes and objects is established.
- In the case of Assign1/Planner and OurDate/Planner there is only a “helper” relationship – i.e. OurDate objects in the Planner class are use only to retrieve values which ultimately will be part of an Event reference.
- Aggregation – the “has a” relationship between classes.
- Composition – a type of aggregation where the objects do not exist independently of the containing object – in this case, an Event must have a date.

The “is a” relationship, specifies inheritance

Java Concepts Early Objects by Cay Horstmann was referenced for parts of the UML relationship definitions

Example: Deitel 3.8

```
1  // Fig. 3.8: Account.java
2  // Account class with a double instance variable balance and a constructor
3  // and deposit method that perform validation.
4
5  public class Account
6  {
7     private String name; // instance variable
8     private double balance; // instance variable
9
10    // Account constructor that receives two parameters
11    public Account(String name, double balance)
12    {
13       this.name = name; // assign name to instance variable name
14
15       // validate that the balance is greater than 0.0; if it's not,
16       // instance variable balance keeps its default initial value of 0.0
17       if (balance > 0.0) // if the balance is valid
18          this.balance = balance; // assign it to instance variable balance
19    }
20
21    // method that deposits (adds) only a valid amount to the balance
22    public void deposit(double depositAmount)
23    {
24       if (depositAmount > 0.0) // if the depositAmount is valid
25          balance = balance + depositAmount; // add it to the balance
26    }
27
28    // method returns the account balance
29    public double getBalance()
30    {
31       return balance;
32    }
33
34    // method that sets the name
35    public void setName(String name)
36    {
37       this.name = name;
38    }
39
40    // method that returns the name
41    public String getName()
42    {
```



Deitel: To distinguish a constructor from the class's operations, the UML requires that the word "constructor" be enclosed in guillemets (« and ») and placed before the constructor's name. It's customary to list constructors before other operations in the third compartment.

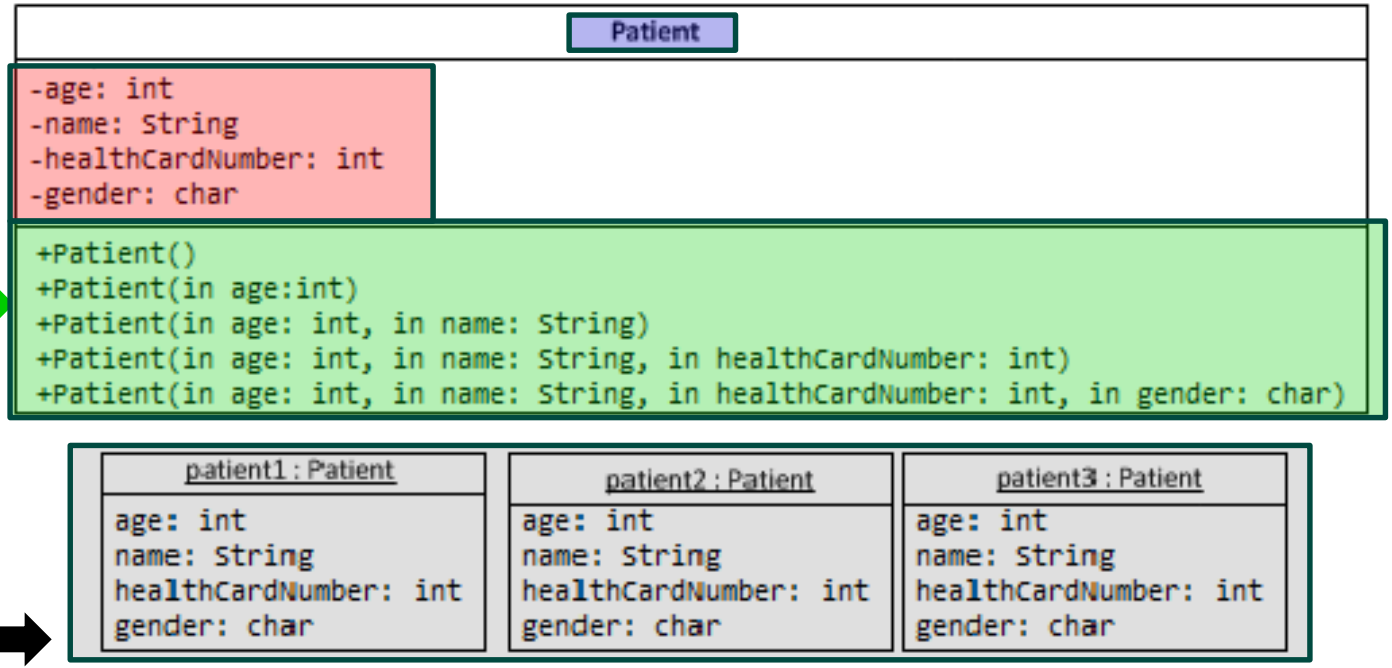
UML for Class Patient:

Class Name

Class fields
" – " specifies private access. Data type also specified.

Constructors and methods
"+" specifies public access
Return data types as well as data type for any formal parameters are specified

Three instantiated references to objects



NOTE!!! Different UML creation software can have minor differences in the layouts. The above UML was created with an earlier version of MS Visio.

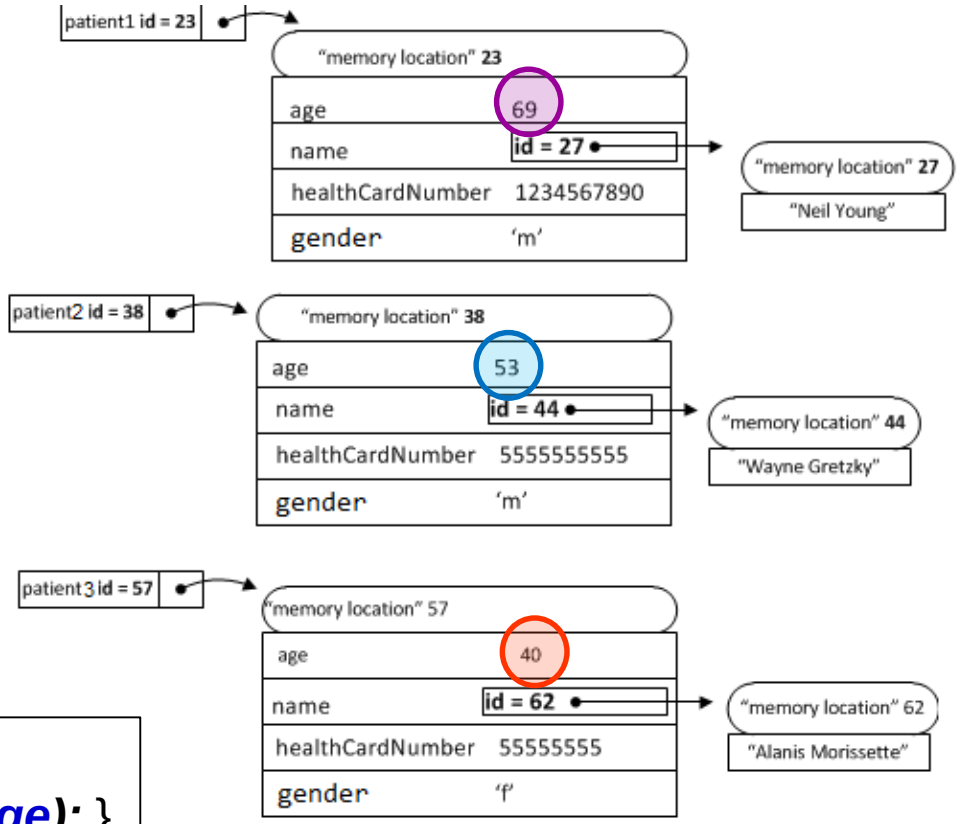
Memory Map representation of Patient Objects

Runtime fetching of heap variables:

patient1.printAge();

patient2.printAge();

patient3.printAge();




```
public void printAge(){
   System.out.println("age = " +this.age); }
```

***a printAge() function is not required for assign #1

Keyword *this*

- Keyword ***this*** refers to an object itself.
- It is a “placeholder”, for an object, such as in the case when a method is invoked with an object reference and the dot operator: **anyObject.anyMethod();**
- In the printAge() method to the right, the keyword ***this*** means “whatever object I happened to be talking about at this particular time, use *it’s* copy of the age field”.
- Notice how in main() that there are three Patient class objects instantiated: patient1, patient2 and patient3.
  - Each of these object references has it’s own copy of the field ***age*.**
  - Each time the printAge() method is invoked, the ***age*** field of the reference to the object that invoked the method it means to print to the console,.

```
public class Patient {

  private int age;
  private String name;
  private int healthCardNumber;
  private char gender;

//….other code

  public void printAge(){
    System.out.println("age = " +this.age);
  }

public static void main(String[] args) {

   Patient patient1 = new Patient();
   Patient patient2  = new Patient();
   Patient patient3 = new Patient();

   patient1.printAge();
   patient2.printAge();
   patient3.printAge();
}
```

More on keyword *this*

- Keyword ***this*** can be used in class definitions, as a means to differentiate between the data fields of a class, and the local variables or formal parameter list (the inputs) of methods defined in the class (refer to Patient class)

```
public class Patient {

  private int age;
  private String name;
  private int healthCardNumber;
  private char gender;


Patient(int age, String name, int healthCardNumber, char gender){

  age = age;  //has no effect – it is referring to the formal
              //parameter list variable age
  this.age = age;  //including keyword "this" in front of age
                   //confirms that it is the field variable age
  this.name = name;
  this.healthCardNumber = healthCardNumber;
  this.gender = gender;
}
```

Question:

Q. When an object is created in memory, describe specifically in order how the data fields in the object will be initialized. USE the statement Student student = new Student(); as an example in your answer.

Scope of answer required:

1. fields initialized to 0 or null.
2. any explicit initializations specified in the fields (i.e. private int x = 2;)
3. constructor initializations.

Note: for folks who want an even deeper answer, here are some links for you:

*https://docs.oracle.com/**java**se/**specs**/**jvms**/se8/**jvms8**.pdf*
http://docs.oracle.com/javase/tutorial/java/javaOO/initial.html
http://stackoverflow.com/questions/2420389/static-initialization-blocks
http://docs.oracle.com/javase/tutorial/java/javaOO/objectcreation.html

Q: What is the order of events when a program is compiled and run?

1. Class is loaded by class loader
2. Instance fields are given default values
3. Contructor(s) are called to set values into fields
4. Reference to object is returned

String.format() method

- The format() method has many similarities to the printf() method.
- If you don't want to print it, but just return the result formatted as a string, use String.format( ).
- ∴ it returns a formatted string but does not print it.
- In order to print it you would need to use it as an argument to a print() statement.

```
String name = "Joe";
double balance = 123.4567;
String message = String.format("The balance for %s is %.2f\n", name, balance);

System.out.println(message);
```

Output:

The balance for Joe is 123.46

toString() method

- Returns a *string representation* of an object.
- main() has a statement that invokes the toString() method in the argument of a println().
- In this case, a toString() method has been defined in the class definition.

Data type specified in method header must correspond with the data type specified after the “return” statement.

```
public String toString(){

return String.format("Name: %s  Age: %d Gender: %c\n", this.name,
this.age, this.gender);  //a string is returned
}

public static void main(String[] args) {

   Patient patient1 = new Patient(69, "Neil Young", 1234567890, 'm');
   System.out.println(patient1.toString());
}
```

Output:

Name: Neil Young Age: 69 Gender: m

toString()

- If a toString() method is not defined in a class, the superclass version of the toString() method will be used (refer to class Object but….the superclass Object will not be a serious topic until the *Data Structures* course.
- Passing an object reference to the class Object's version of toString() yields some "strange" output…seen it before????
- In this case, the ClassName@hashcode is the string representation of the object.
- In the toString() method definition on the previous slide, the annotation @override would be specified.

In order to obtain meaningful output (i.e. not something that resembles Patient@7852e922), and only when the subclass version of the toString() method is defined, it is not necessary to invoke this method explicitly:

i.e. System.***out*.println(patient1.toString());**

Alternatively, toString() may be invoked implicitly
i.e. System.***out*.println(patient1);**

```
public static void main(String[] args) {

Patient patient1 = new Patient(69, "Neil Young",
1234567890, 'm');

System.out.println(patient1.toString());
```

```
@Override
public String toString(){

return String.format("Name: %s  Age: %d Gender:
%c\n", this.name, this.age, this.gender);  //a string is
returned
}
```

Output: (if the above toString() method was not defined)

Patient@7852e922

Output: (toString() method is defined)

Name: Neil Young Age: 69 Gender: m

Override versus Overload

- “Overload”, as in overloaded method (or constructor), refers to methods having the same name *BUT*…the input parameter list is different.
- These methods are likely being used to implement the same basic functionality.

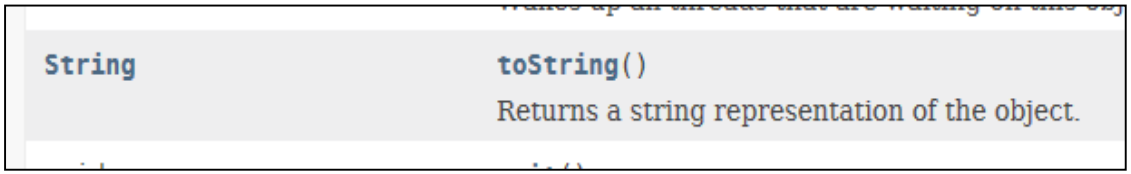
```
//overloaded method
public void searchForPatient(int healthCardNumber){

   //code for search algorithm
   //search by health card number field

}

//overloaded method
public void searchForPatient(String name){

   //code for search algorithm
   //search by name field
}
```

Override versus Overload

- "Override", as in @Override, is a term related to inheritance, that indicates that the method in question (the current subclass) should be used, instead of the same version of the method from the superclass .
- Specifying @overrride is neither code nor comment, but is a "gentle reminder" to the compiler to verify that the programmer has indeed specified a correct header for the definition: i.e. it corresponds with the signature of the superclass version of the method.
- *Sometimes,* the superclass version of the method is the one required to be invoked….to be discussed.

```
@Override
public String toString(){

return String.format("Name: %s  Age: %d Gender: %c\n",
this.name, this.age, this.gender);
}
```

Java 8 api documentation for class Object:
http://docs.oracle.com/javase/8/docs/api/index.html?overview-summary.html



Immutable versus Mutable

- Immutable objects are objects whose data cannot be changed once the object is created – such as String.
- Mutable objects are objects whose data ***can*** be changed – such as StringBuilder.

- String class….creates immutable objects
  - “The class String includes methods for examining individual characters of the sequence, for comparing strings, for searching strings, for extracting substrings, and for creating a copy of a string with all characters translated to uppercase or to lowercase”.

Recall the Binary Representation of Numbers:

- Numbers are represented as binary digits (1's and 0's) on a computer.
- In the chart on the right, three bits are required to represent the numbers 0-7.
- This makes eight decimal numbers in total that can be represented with three bits (0 is a number!).

Decimal	Binary
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Three bits are required if only the magnitude of the number is required.
If there were a sign bit, a fourth bit would be required in order to represent the numbers above.

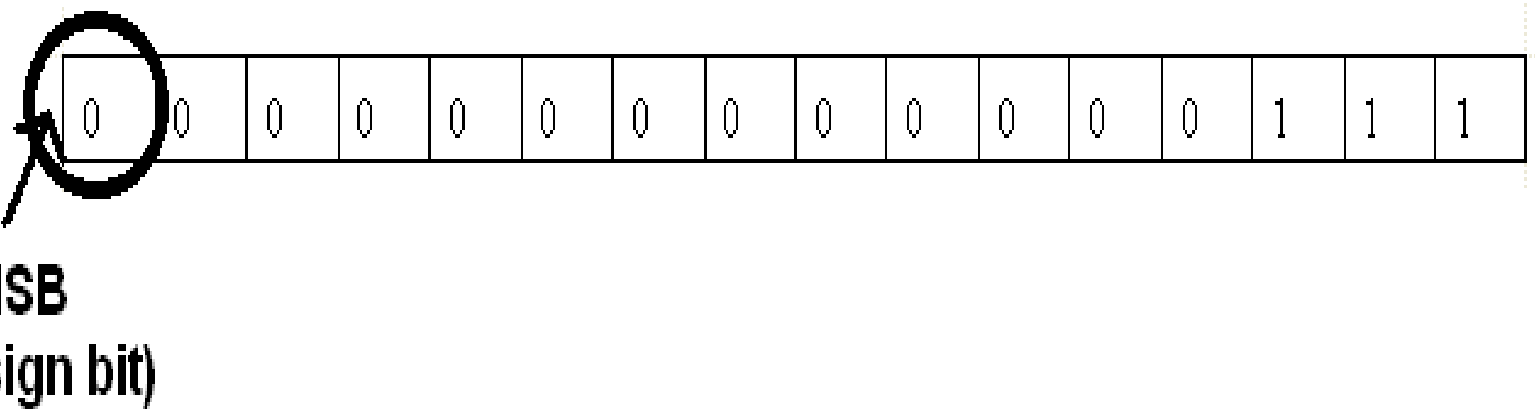
Binary Numbers

- If we expanded the table in the previous slide to show the binary representation of the decimal numbers 0-15, we would find that 4 binary digits are required.
- Even though, say the decimal integer 7, can be represented with 3 binary digits, computers allocate many more binary digits.
- How many bits are allocated for data is system dependent.
- With 32 bits, 31 are used to represent the magnitude of the decimal number and the MSB (most significant bit) is used to represent the sign (positive or negative number).

Bits

Example:

- a 16-bit machine representing a positive 7.
- Fifteen bits are used to represent the magnitude of the number and one bit is used for the sign: either +ve (binary 0) or -ve (binary 1)



Underflow Errors

- Ex. $1.23*10^{-20}$ = 1.23e-20

  = 0.00000000000000000000123
- Depending on the system, a result of 0 might be reported instead.

Round-off Errors

- Binary representation of integer numbers produces exact results.
- Binary representation of real numbers can produce errors.
- Ex. 1.0/3.0 = 0.33333333333
- At some point, truncation or rounding must occur.

Floating Point Numbers and Precision

- Floating point numbers can be *infinitely* repeating after the decimal point (i.e. 3.1415926….).
- But, there are a finite amount of bits to represent any number – therefore floating point numbers have ***imprecision*** associated with them.
- Unavoidable errors such as these round-off errors require a work-around solution:

Define a level of tolerance that can be accommodated without losing any correct results.

- Another common strategy can be implemented by taking the absolute value of the difference between two numbers, and then compare the result to the tolerance level:

$$|x - y| \leq \varepsilon$$

```
public static void main(String [] args){

   double result = Math.sqrt(2);

 //round off error
   double takePower = result * result - 2;
   System.out.println(takePower);
}
```

Console Output:
4.44089209850062E-16

Overflow Errors

- If binary arithmetic should happen to produce an overflow into the MSB position, the sign bit would change.
- In other words, a positive number would show itself as a negative number, which would be very bad.

A Pattern?

- It would seem that a formula exists in order to be able to determine mathematically, what is the largest number that can be represented, given a finite amount of binary digits:

$$Largest\_decimal\_number = 2^{n-1} - 1,$$
$$n = number\_of\_bits$$

Example

- You have a variable (short int) that requires 2 bytes of storage. What is the largest number that can be represented?

$$Largest\_decimal\_number = 2^{16-1} - 1$$

$$= 32768 - 1$$

$$= 32767$$