**Advanced Java: Chapters 9, 10, 11**

What are the two defining characteristics of a primitive?

What are the three defining characteristics of an object?

What is an object?

What is a class?

What is the relationship between an object and a class?

What are some of the characteristics of OOP?

polymorphism

encapsulation and data hiding

dynamic binding

inheritance

abstraction

What is UML?

What is a Class Diagram?

What are the *two* relationships between classes?

**Access to a class:**

public: public data and members are accessible from anywhere outside the class

private: private data and members cannot be accessed from outside the class

protected: protected data members can be accessed by “child” classes

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Modifier | Class | Package | Subclass | World |
| public | Y | Y | Y | Y |
| protected | Y | Y | Y | N |
| no modifier | Y | Y | N | N |
| private | Y | N | N | N |

“no modifier” is sometimes called “package private.” This is the default access modifier in Java.

Some view this as a mistake. An additional twist is that protected members can be accessed from outside the package in which it is declared *only* by code that is responsible for the implementation of that object. In other words, subclasses in other packages can access protected members of their superclasses only on instances of their own type. Because type is checked at compile-time, not run-time, protected members of arbitrary instances of superclasses are off limits. We avoid these problems by avoiding packages entirely in this course.

**Methods of a class:**

**Constructors**

The *two* jobs of the constructor is to create an instance of the object and fully specify the state of that new object.

Empty constructor – uses default values

Copy constructor – parameter is of the same type as the class; makes a “deep copy”

**Accessors**

Accessors allow access to some or all of the state, usually through a copy

**Mutators**

Mutators allow alteration of the state of the object

**Members**

Member methods perform some operation that is specific to the class

*static* members can be invoked by the class name itself.

*non-static­* members can only be invoked by an instance of the class; they act on their

arguments and nothing more.

Some classes in Java have *only* static members; *e.g.* Math.

You can’t even create an object of type Math any more (the constructor is private!)

**Interface**

In Java an “interface” is a class that contains only method signatures and constant declarations. They cannot have state, nor can they include method definitions; they are used (as the name implies) to specify an interface.

**Inheritance in Java**

In Java we have a limited form of multiple inheritance; an object can inherit from only one “stateful” parent but many interfaces. We use the keyword “extends” for our one and only stateful parent, and “implements” for as many interfaces as we desire.

public class MyClass extends Parent implements Interface1, Interface2, Interface3

**CLASS CONSTRUCTION AND INHERITANCE**

We’ll start with a parent class…

|  |
| --- |
| **Person** |
| - firstName: String  - lastName: String  - dob: OCCCDate |
| + Person (String firstName, String lastName)  + Person (String firstName, String lastName, OCCCDate dob)  + Person (Person p) // copy constructor  + String getFirstName()  + String getLastName()  + void setFirstName(String fn)  + void setLastName(String ln)  + OCCCDate getDOB();  + String toString() // format lastName, firstName (birthdate)  // Gordon, Freeman (05/19/1955)  + boolean equals(Person) // ignore case on the first and last names  + void eat() // simply prints “Person is eating…” on the console  + void sleep() // …as above  + void play() // …as above  + void run() // …as above |

On the eat / sleep / play / run methods make sure you obtain the identity of the class using a method; that way when the children invoke it *their* class types will be displayed.

…and a calendar class that is really a wrapper around the built-in GregorianCalendar class.

|  |
| --- |
| **OCCCDate** |
| - dayOfMonth: int // 1..31  - monthOfYear: int // 1..12  - year: int // e.g. 2013  - gc: GregorianCalendar // a private helper object  - dateFormat: boolean // true for US date format, false for Euro format; default is true  - dateStyle: boolean // true for months as numbers, false as months as names  + static final boolean OCCCDATE\_US (true) // for the convenience of our clients  + static final boolean OCCCDATE\_EURO (false) // another one  + static final boolean OCCCDATE\_STYLE\_NUMBERS (true)  + static final boolean OCCCDATE\_STYLE\_NAMES (false) |
| + OCCCDate() // default constructor, uses current date and time  + OCCCDate(int day, int month, int year)  + OCCCDate(GregorianCalendar gc)  +OCCCDate(OCCCDate d) // copy constructor  + int getDayofMonth()  + String getDayOfWeek() // Sunday, Monday, Tuesday..  + int getMonth() // 1 = January, 2 = February…  + String getNameOfMonth() // January, February, March…  + int getYear() // Gregorian year, *e.g.* 2016  + void setDateFormat(boolean df)  + void setStyleFormat(boolean sf)  + boolean equals(OCCCDate dob) // compare only day, month, and year  + String toString() // US format mm/dd/yyyy or monthName dd, yyyy  // Euro format dd/mm/yyyy or dd monthName yyyy |

We’ll add to this a derived class:

|  |
| --- |
| RegisteredPerson |
| - govID: String |
| + RegisteredPerson (String firstName, String lastName, OCCCDate dob, String govID)  + RegisteredPerson (Person p, String govID)  + RegisteredPerson (RegisteredPerson rp)  + String getGovernmentID()  + boolean equals(RegisteredPerson rp) // the usual equals method  + boolean equals(Person p) // compare only Person fields, ignore government ID |

**HOMEWORK**

Write a program in which you implement all three classes above and demonstrate every behavior of each. Make sure that the user has ample opportunity to input names and dates. Submit your source codes on Moodle as usual.

If you follow the UML, *my* “test” program should work with *your* class files.