Java OCA se7 II

1 Java Class Design

- [x] Use access modifiers: private, protected, and public
- [x] Override Methods
- [x] Overload constructors and Methods
- [ ] Use the -instanceof- operator and casting
- [ ] Use virtual method invocation
- [ ] Override the hashcode, equals, and toString methods from the Object class to improve the functionality of your Class
- [ ] Use package and import statements

The OCP Java SE Programmer 03.07.2019
II certification is designed to tell would-be employers that you really know your basic & advanced Java. It certifies that you understand & can work with design patterns & advanced Java concepts like concurrency, multithreading, Localization, string processing & JDBC. [1]

You need to know Java inside & out, & you need to understand the certification process so that you're ready for the exam - i.e. 1Z0-804 [2]

- Introduction to OCP Java SE 7 Programmer II certification 1Z0-804 covers intermediate & advanced topics/concepts of Java programming, threads, concurrency, localization JDBC, String processing, and design patterns. [3]

Real, on-the-job projects need you to understand & work with multiple basic & advanced concepts. Finer details covering the finer details of basic Java-like class design, covers advanced Java topics - Multithreading, concurrency, localiz-ation, File-I/O [4]

- string processing, exception handling, assertions, collections API & design patterns. This certification establishes your expertise with these topics, increasing your prospects for better projects, Jobs, remuneration, responsibilities, & designations. 03.07.2019

Oracle Certified Professional Java SE 7 Programmer, requires successfully completing the following two exams

OCA Java SE 7 1 1Z0-803
OCP Java SE 7 II 1Z-804 [5]
Objectives :

Java class design: Classes & interface are building blocks of an application. Efficient & effective class design makes a significant impact on the overall application design. [6]

The creation of overloaded methods is another domain, which is an important class design decision. It eases instance creation & use of methods. [7]

Class design decisions require 03. 07. 2019
an insight into understanding correct &
appropriate implementation practices.
When armed with adequate information you'll
be able to select the best practices &
approach to designing your classes.

- Access Modifiers
- Method Overloading
- Method overriding
- Use of the instanceof operator & casting
- Override methods from class Object to improve the functionality of your class
- How to create packages & use classes from other packages

[1.1] Use access modifiers: 03. 07. 2019
private, protected & public

When you design applications & create classes,
you need to answer multiple questions
- How do I restrict other classes from accessing certain members of a class?
- How do I prevent classes from modifying the state of objects of a class, both within the same & seperate packages?

Java access modifiers can be applied to classes, interfaces, & their members (methods & variables) by other classes & interfaces within the same or seperate packages. By using the appropriate access modifiers, you can limit access to your classes or interfaces, & their members (instance & class variables & methods

Default access modifier - when you don't use an access modifier. ¢3. ¢7 2¢19
" [1.2] A1

1.1.1
Public access modifier
The least restrictive access modifier. Classes & interfaces defined using the public access modifier are accessible across all packages - from derived to unrelated classes.
[9] [A2]

1.1.2 Protected access modifier
The members of a class defined using the protected access modifier are accessible to

- Classes & interfaces defined in the same package
- All derived classes, even if they're defined in seperate packages.

" graph A.3

graph A.4 ¢3.11.2¢19

A derived class inherits the protected members of its base class, irrespective of the packages in which they are defined.

```
package building;
import library.Book;
class StoryBook extends Book {
    StoryBook() {
        Book book = new Book();
        Sting b = book.author;
        book. modifyTemplate();
    }
}
```

Book & StoryBook defined in seperate packages

Protected members of Book aren't accessible in StoryBook if accessed using an instance of Book

★ A derived class can inherit & access protected membets of its base class, regardless of the package in which it is defined

A derived class in a seperate package can't access protected members of its base class using reference variables ''' img. A.1 — 03.21.2019

''' Default access (package access) the members of a class defined without using any explicit access modifier are defined with package accessibility (also called default accessibility) the members with a package access are only accessible to classes & interfaces defined in the same package.
* The default access is also referced to as package-private

Think of a package as your home, classes as rooms, & things in rooms as variables with default access. These things are not limited to one room. They can be accessed across all the rooms in your home. But they are still private to your home. When you define a package, you might want to make accessible members of classes to all the other classes across the same package. — 03.21.2019
* - The package-private access is valid as the other access levels - in real packages, it often appears as the result of inexperience-d developers forgetting to specify the access modifier of Java components.

```
package library;
public class Tome {
    int issueCount;        ← issueCount with default access
    void issueHistory()    ← issueHistory() with default access
    { }
```

Notice how classes from the same package & separate packages, derived classes, & unrelated classes access class Tome & its members, i.e. instance variable issueCount & method issueHistory() — 03.25.2019
''' graph A.2

''' Because classes CourseBook and Librarian are defined in the same package as class Tome, they can access the members issueCount & issueHistory(). Because classes House & StoryBook are <u>not</u> defined in the same package as class Tome, they can't access the members issueCount & issueHistory(), no duh!

class House is unaware of the existence of issueHistory() it fails compilation

So What happens if you define a class with default access? What will happen to the accessability of its members? If the class itself is has default accessibility? i.e within the package.

A package is like an island $03.25.2019
in Java, whatever services on the island are accessable to the denizens of the island... e.g. Tom Hanks & Wilson. A class defined on the island, e.g. Wilson's bug eyed burgers is available to both Tom Hanks, and Wilson. Wilson's bug eyed burgers is not open to outsiders.

A class defined, with default (package) access is visible & accessible only within the package in which it is defined. It cannot be accessed outside of its package.
Define: class Tome with default package access as follows →

```
package library;
class Tome {          ← Tome now has default access.
    //... class members
}
```

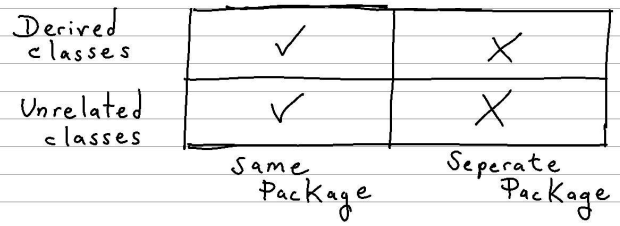
The behavior of class Tome remains the same for classes CourseBook & Librarian, which are defined in the same package.

But class Tome can't be accessed by classes House & StoryBook, which reside in a seperate' package. 03.26.2019

e.g.

```
package building;
import library.Tome;
public class House {}
```

← Tome is not accessable in class House



Class House fails compilation with the following :

House.java : library.Tome is not public in library; cannot be accessed from outside package

import library.Tome;

A lot of programmers are confused about which members are made accessible by using the 'protected' access modifier (default), this exam tip should help → 03.26.2019

★ Default access can be compared to package-private (accessible only within a package) & protected access can be compared to 'package-private + derived classes. Derived classes can access protected members only by inheritance & not by reference ~ i.e using the dot operator to access members on an object.

1.1.4. The Private Access Modifier

The private Access Modifier is the most restrictive : The members of a class defined using the 'private' AM. are accessible only to them. e.g. internal organs, heart, lungs etc.)

It doesn't matter whether §3.26.2§19
the class or interface in question is from another package or has extended the class ↳ Private members are not accessible outside the class in which they are defined.

* Members of an interface are implicitly public. You cannot define interface members as private - they will not compile.

''' diagram 1.11

''' Private members in action → by adding a private method 'countPages()' to class Tome ↳

```
package library
class Tome {
    private void countPages() {
    }
    protected void modifyTemplate(){
        countPages();
    }
}
```

Private method → only Tome can access its own countPages()

None of the classes defined §3.26.2§19
in any of the packages (whether derived - or not) can access the private method countPages()

```
package library;
class TextBook extends Tome {
    TextBook() {
        countPages();
    }
}
```

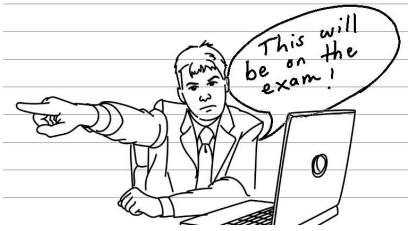
TextBook can't access private method countPages()
Because class TextBook tries to access private members of class Tome, it shall not compile!
Likewise, if any of the other classes (StoryBook, Librarian, or House) try to access private method countPages() of class Tome, it shall not compile

X	X	Derived classes
X	X	Unrelated classes
Same Package	Seperate Package	

1.1.5 Access Modifiers & Java entities 03.26.2019

Can every access modifier be applied to all Java entities? No!

Entity name	Public	Protected	Private
Top-level class, interface, enum	✓	X	X
Nested class, interface, enum	✓	✓	✓
class Variables & methods	✓	✓	✓
Instance variables & methods	✓	✓	✓
Method params & local variables	X	X	X



Try to combine, none of the combinations will compile. 03.27.2019

```
protected class TopLevelClass {}
private class TopLevelClass {}
protected interface TopLevelInterface {}
protected enum TopLevelEnum {}
```

Won't compile

top-level class, interface & enums can't be defined with protected & private access

```
void stansMethod(private int param) {}
void stansMethod(int param) {
        public int localVariable = 42;
}
```

Won't compile! & local variables can't be defined using any explicit access modifiers

A change in the access modifier of a member of a class can break the code of other classes. §3.27.2§19

When you modify the access modifier of the member author to default access from protected it can no longer be accessed by class Scifibook because they reside in seperate packages.

- Accessibility is decreased
  - a public member is made private

Accessibility is increased a private member is made public (duh!)

When accessibility of an entity is decreased [ie. be more restrictive]
Decreasing the accessibility of entities can affect the overall application in very critical ways. * consider when designing APIs & maintaning monolithic software. Do not make the mistake of decreasing the accessibility of methods or fields. This can result in access issues with other components in the system.

When accessibility of an entity is increased (less restrictive) §3.27.2§19

There should be no issues when entity is made less restrictive, e.g. when access of an entity is changed from default to protected or public.

With increased access, an entity may become visible to other classes, interfaces & enums to which it wasn't visible earlier.
This topic will be on the exam. Learn access modifier behaviors.
Carefully examine all the code on the exam, look for trick (tweaked code) focus on the access modifiers ~ the one's that will cause compile issues. You may see the same 'right' answer, just like those physics problems in school. Pick the most right answer, or the question might differ only so slightly. This could change the behavior of the code.

```
package library
public class Tome {
    protected String author;
}
```

class written by Priya — 03.27.2019

```
package building;
import library.Tome;
class Story extends Tome {
    { author = "King"; }
}
```

class written by Benji'mon Marley

Priya made some updates, and pushed/merged her changes in Git. Benji'mon Marley wakes up the next Monday and observes the following →


```
package library;
class Tome {
    protected String author;
}
package building;
import library.Tome;
class Story extends Tome {
    { author = "King"; }
}
```

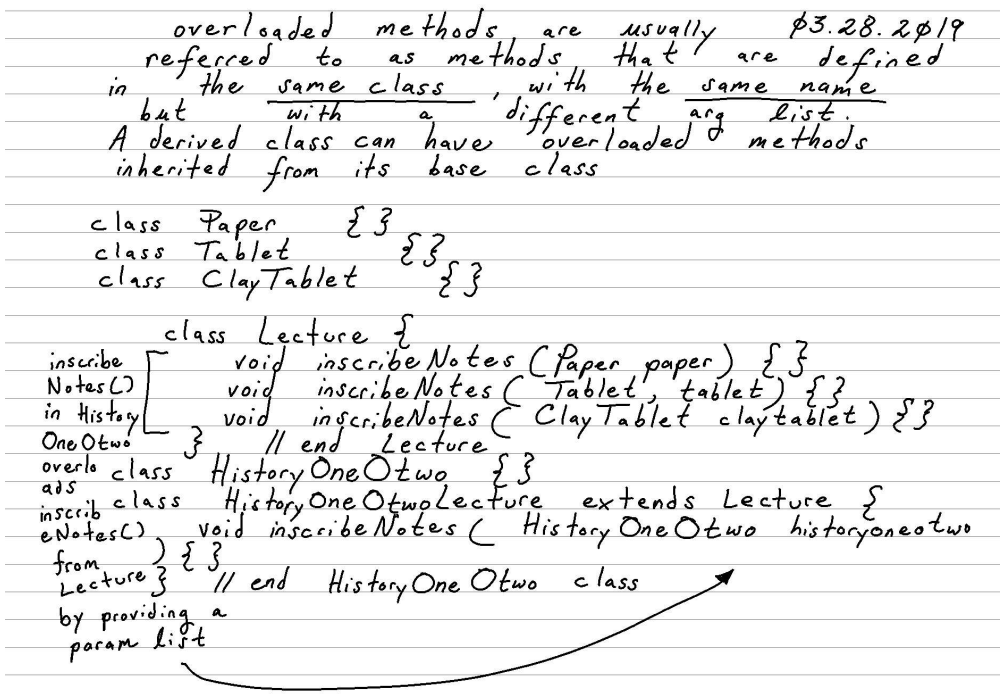
Overloaded Methods & constructors — 03.27.2019

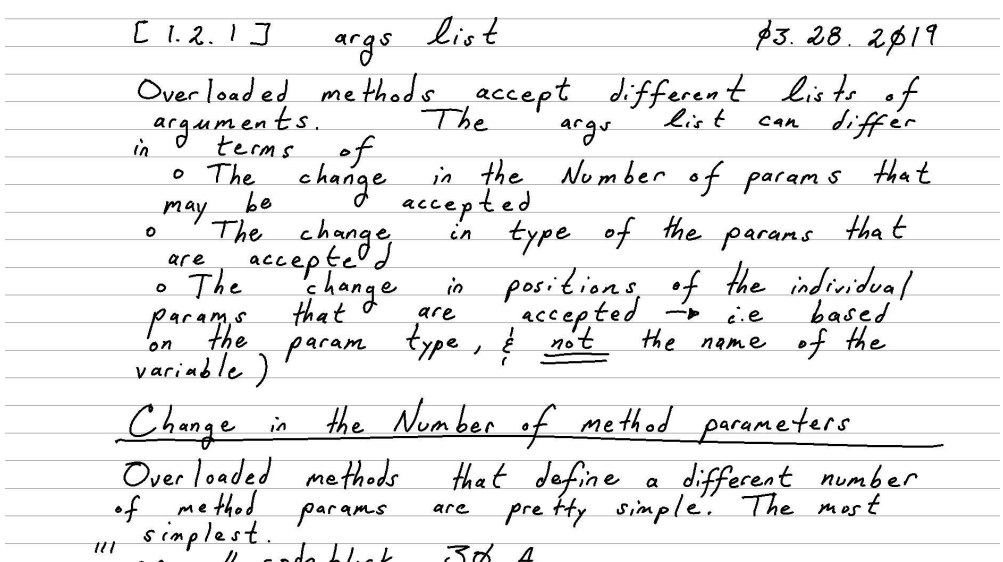
1.3. Overload constructors & methods

Overloaded methods are methods with the same name but different method parameter lists.
e.g. You can take notes on a paper notepad or on a digital tablet or on actual clay tablets like the clay tablets used by the Summerians.

```
class Paper {}
class Tablet {}
class ClayTablet {}
class Lecture {
    void takeNotes(Paper paper) {}
    void takeNotes(Tablet tablet) {}
    void takeNotes(ClayTablet claytablet)
{}
} // end class Lecture
```

overloaded method takeNotes()






```
//code stub 30-A
class Result {
    double calcAvg(int grade01, int grade02) {
        return(grade01 + grade02)/2;
    }
    double calcAvg(int grade01, int grade02, int grade03) {
    return (grade01 + grade02 + grade03)/3;
    }
}
```

''' Change in type of method params
In the following code block, the difference

in the args list due to the change in type of the params // code block 31A 03.28.2019
'''

```
//code stub 31-A
class Result {
    double calcAvg(int grade01, double grade02) {
        return(grade01 + grade02)/2;
    }
    double calcAvg(double grade01, double grade02) {
    return (grade01 + grade02)/2;
    }
}
```

''' You see you can pass Dev object, no problem-o to the bookTicketCS method
But what happens if you define overloaded methods that can accept object references of classes of which do share an inheritance relationship?
'''//codeblock 31C

```
//code 31B
class Employee{}
class SoftwareEngineerInTest extends Employee {}
class CEO extends Employee {}
class Travel {
    static String bookTicket(SoftwarEngineerInTest val) {
        return "coach";
  }
    static String bookTicket(CEO val) {
        return "business class";
  }
}
```

```
//code stub 31-C
class Employee {}
class CEO extends Employee {}
class Travel {
    static String bookTicket(Employee val) {
        return "coach";
```

```
    }
    static String bookTicket(CEO val) {
        return "business class";
    }
}
```

''' // codeblock 32 A ϕ3. 28. 2ϕ19

```
//code stub 32-A
class TravelAgent {
    public static void main(String[] args) {
        System.err.println(Travel.bookTicket ( new CEO ())); //prints business cl
     }
}
```

''' The preceding code calls overloaded method bookTicket() of which accepts a Dev object - because without any explicit reference variable, new Dev() is referring to using a Dev variable.

Now determine this →

```
class PromotionAgent {
    public static void main (String [] ... args)
{      Employee emp = new Dev();
    System.err.println( Promotion.createJobTitle
    (emp));
```

Notice the console log prints "Associate Developer" & not Software Engineer, Senior

because the type of the reference variable emp is → Employee. The overloaded methods are bound by at compile time & not at runtime.

To resolve the call to the overloaded methods, the compiler considers the type of variable that's used to refer to an object.

Calls to overloaded methods are resolved at Compilation

e.g. if you are a test engineer you are an employee, if you are a ceo, you are also an employee. If the test engineer books a flight -> because they are an employee, they will be flied on coach, same with CEO, because the CEO is still referred as a type of employee, i.e. the referrence type Not the object in itself.

For the overloaded method bookTicket() that defines the method parameter of either Engineer or CEO, watch for tricks & traps on the exam that try to call it using a reference variable of Employee

33. A §3.28.2§19

```
class Emp { }
class Engineer extends Emp { }
class CEO extends Emp { }
class Travel {
    static String bookTicket (Engineer val) { // accept Engineer
        return " coach ";
    }
    static String bookTicket ( CEO val ) { // Accepts
        return " business class ";          // Ceo
    }
    public static void main ( String args[] ) {
        Emp emp = new CEO();
        System.out.println ( bookTicket(emp));
    }
}
```


```
// src/33B.java
double  calcAvg( double grade1, int grade2 )
{
  return (grade1 + grade2 ) / 2;
}
double calcAvg ( int grade1, double grade2 ) {
  return ( grade1 + grade2 ) / 2;
}
```

Change in the positions of Method params 03.29.2019

The methods are correctly overloaded if they only change the positions of the parameters that are passed to them →

33.B
```
double calcAvg(double grade1, int grade2)
{
  return (grade1 + grade2) / 2; // args double & int
}
double calcAvg(int grade1, double grade2) {
  return (grade1 + grade2) / 2;
}
```
// args int & double

The arguments being accepted are the same. The Java compiler treats them as different args lists. So this is a valid example of overloaded methods An issue will arise - i.e when you try to execute this method using values that can be passed to both versions of the over-loaded method. The method main() will not compile.

```
// src/GradeResultsAvg.java
class GradeResultsAvg {
  public double calcAvg(int grade1, int grade2) {  //access is public
    return (grade1 + grade2) / 2;
  }
  protected double calcAvg(int grade1, int grade2) { // access is protected
    return (grade1 + grade2) / 2;
  }
}
```

① defines the calcAverage() method, which accepts two params : a double & an int.
② defines an overloaded method calcAverage(), which accepts two method params : 1st → an int, & then a double.
Because an int literal value can be passed to a variable of type double, literal values 2, & 3 can be passed to both overloaded methods, declared in ① & ② because this method call is dubious
@ ③ fails to compile

34.A

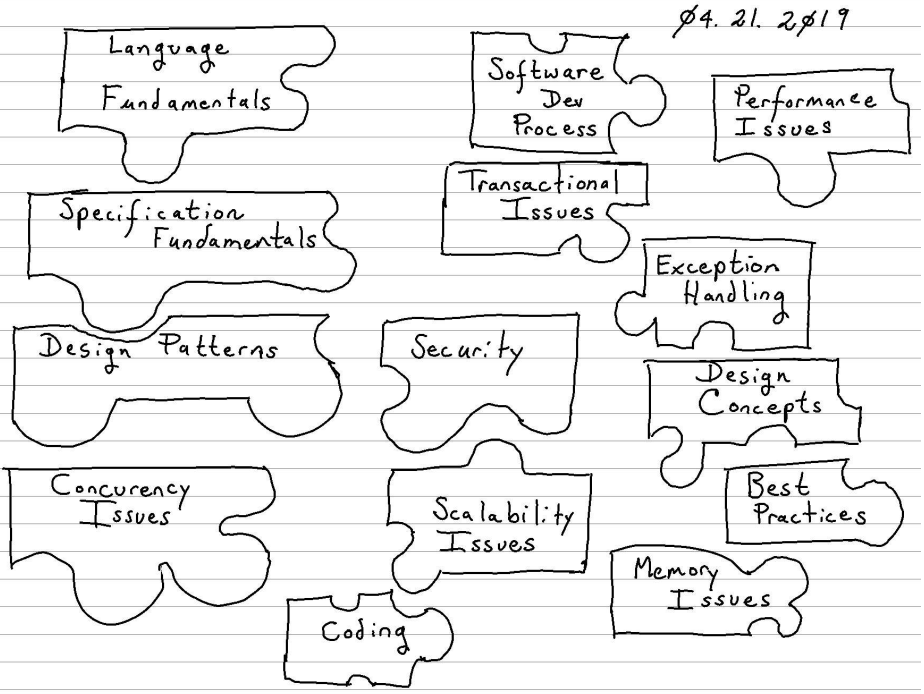
''' e.g. static double calcAvg (int grade1, grade2)

Should work since type is of int.

When methods can't be defined as overloaded methods

* Overloaded methods give you the flexability of defining methods

04.21.2019



04.21.2019

By Access Modifier 04.21.2019

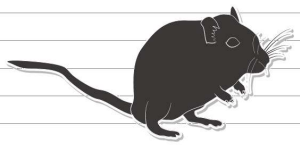
Methods can't be defined as overloaded if they only differ in their access modifiers as follows

```
class Result {
    public double calcAvg(int g1, int g2)
    {
    } return (int1 + int2) /2
}
```

Access is public

```
class Result {
    protected double
    }
}
```

access is protected



By Non-access Modifier 04.21.2019

Methods can't be defined as overloaded if they only differ in their nonaccess modifiers → like so

```
class synchronized double calcAvg(int g1, int g2)
{
    return (g1 + g2) /2
}

public final double calcAvg(          ) {
    }
    return (g1 + g2) /2
}
```

* Notice Nonaccess modifier synchronized
* Notice Nonaccess modifier final

Here are the rules for defining overloaded methods

-- contin... Rules for defining overload methods 04.21.2019

Remember this →

- A class can overload its own methods & methods inherited from its base class
- Overloaded methods must be defined with the same name
- Overloaded methods might define a different return type or access or nonaccess modifier, but they can't be defined with only a change in their return types or access or nonaccess modifier.
- Overloaded methods must be defined with different param lists

Let's create special overloaded methods called constructors - used for creating objects

Overloaded Constructors 04.22.2019

Creating instances of a class - you may need to assign default values to some of its variables & assign explicit values to the rest. This can be done by overloading the constructors.

Overloaded constructors

- Follow the same rules as overloaded methods
- Overloaded constructors must be defined using a different arg list.
- Overloaded constructors can't be defined by a mere change in their access modifiers

Becareful on the exam - non access modifiers can't be used with constructors -! The code won't compile

04.22.2019

```
class Emp {
    String name;
    int age;
    Emp() {                          ① No-arg constructor
        name = "Dave";
        age = 19;
    }
    Emp(String newName) {            ② Constructor with one String arg
        name = newName;
        age = 19;
    }
    Emp(int newAge, String newName) {   ③ Constructor with 2 args (int, String)
        name = newName;
        age = newAge;
    }
    Emp(String newName, int newAge) {   ④ Constructor with 2 args String & int
        name = newName;
        age = newAge;
    }
}
```

04.22.2019

Notice → ① defines a constructor that doesn't accept any args & the code at ② defines another constructor that accepts a single arg.

The constructor defined at ③ & ④ - Both accept two args String & int. But the placement of these two args is not the same (order) This is acceptable.

Invoking an overloaded constructor from another constructor

It is common to define multiple constructors within a class.

Overloaded constructors are invoked using the this keyword.

Which is an implicit reference accessible to an object → to refer to itself.

```
class Emp {                                          04.23.2019

    String name;
    int age;
    Emp() {          <-- A: No arg constructor
    }
        this(null, 0);   <-- Invokes B constructor that accepts 2 args.

    Emp(String newName, int newAge) {   <-- C - Constructor that accepts 2 args.
        name = newName;
        age = newAge;
    }
}
```

A. The code defines a no-arg constructor

B. This constructor calls the overloaded constructor by passing to its values null & 0

C. defines an overloaded constructor that accepts 2 args.

It's a common mistake to try to invoke a constructor from another constructor using the class's name. e.g. 04.23.2019

```
class Emp {

    String name;
    int age;
    Emp() {
        Emp(null, 0);
    }
```

```
Emp ( String newName, int newAge) { ϕ4.22.2ϕ19
    name = newName;
    age = newAge;
  }
}
```

When invoking an overloaded constructor using the keyword 'this' → it should be the first the first statement in your constructor

```
class Emp {
    String name;
    int age;
    Emp () {
        System.err.println ("no-arg constructor");
        this (null, ϕ);
    }
    Emp ( String newName, int newAge ) {
        name = newName;
        age = newAge;
    }
}
```

You will not be able to call a constructor from any other method in your class. None of the other methods of class Emp can invoke its constructor. ϕ4.22.2ϕ19

Some rules → for defining overloaded constructors

Here is a quick list for the exam i.e defining using overloaded constructors

- Overloaded constructors must be defined using different args list.
- Overloaded constructors can't be defined by a change in the access modifiers
- Overloaded constructors can be defined using different access modifiers
- A constructor can call another overloaded constructor by using the keyword 'this'
- A constructor →

→ can't invoke a constructor by using its class's name.

04.23.2019

- If present, the call to another constructor must be the first statement in a constructor.

Instance Initializers → you can also define an 'Instance initializer' to Initialize to initialize the instance variables of your class. An instance initializer is a code block defined within a class, using {}

You can define multiple instance initializers in your class.

Each instance initializer is invoked when an instance is created, in the order they're defined in a class. They're invoked before a class constructor is invoked

Why do this?

04.22.2019

Why do you think you need an instance initializer if you can initialize your instances using constructors?

The reasons →

- For a big class it make sense to place the variable initialization after its declaration
- All the initializers are invoked irrespective of the constructor that's used to instantiate an object.
- Initializers can be used to initialize variables of anonymous classes that can't define constructors

e.g → "" code block 38A

```
//codeblock 38A
class Pencil {
    public Pencil() {
        System.err.println("Pencil:constructor");
    }
    public Pencil(String a) {
        System.err.println("Pencil:constructor2");
    }
    {
        System.err.println("Pencil:init1");
    }
    {
        System.err.println("Pencil:init2");
```

```
    }

    public static void main(String[] args) {
        new Pencil();
        new Pencil("a Value");
    }
}
```

''' The output $

```
Pencil : init 1
Pencil : init 2
Pencil : constructor
Pencil : init 1
Pencil : init 2
```

Pencil : constructor 2

```
class Emp {                                          04.23.2019
    String name;
    int age;
    Emp() {
        this("Wilson", 52);
    }
    Emp(String newName, int newAge) {
        this();
        name = newName;
        age = newAge;
    }
    void print() {
        print(age);
    }
    void print(int age) {
        print();
    }
}
```

What's the output ?

Order of execution of parent class constructor, instance initializers, & a class constructor ϕ4.24.2ϕ19

1. Parent initializer block
2. Parent constructor
3. Child initialization block
4. Child constructor

* If a parent or child class defines static initializer block(s), they execute before all parent & child class constructors & instance initializers. In order of parent, then child.

Method overiding & virtual method Invocation

method Overriding → classes can inherit behavior from other classes. But they can redefine the behavior that they inherit → aKa method overriding. ϕ4.24.2ϕ19

Method overriding is an OOP feature that enables a derived class to define a specific implementation of an existing base class method to extend & its own behavior.

"A derived class can override an instance method defined in a base class by defining an instance method with the same method signature / method name & number & types of method params.

Overridden methods are also synonymous with polymorphic methods

The static methods of a base can't be overriden however they can be hidden by defining methods with the same signature →

→ in the derived class. 04.24.2019

Virtual Method - A method can be overriden by a derived class, i.e. a virtual method.

Virtual Method invocation - is the invokation of the correct overriden method, which is based on the type of the object referred to by an object reference & not by the object reference itself.
* It is determined at runtime not at compilation.

You will be asked on the need for overriden methods; the correct syntax of overriden methods; the difference in between over-loaded, overriden & hidden methods

Take a look at some common mistakes for overriden methods & virtual method →

→ invocation 04.24.2019

* A base class method is referred to as the overriden method & the derived class method is referred to as the overriding method.

Why do you need 'Overridden methods'

Just as humans inherit some of their parents' behavior, a derived class can inherit the behavior & properties of its base class but still be different in its own unique manner.

A derived class can "choose" to define a different course of action for its base class method by overriding it. e.g.

''' 41.A

```
//code-stub 41.A
class Tome {
    void issueTome(int days) {
        if (days > 0)
            System.out.println("Tome is is issued");
        else
            System.err.println("Cannot issue tome for zero or negative days.");
    }
}
```

''' This is another class, SpellBook, which inherits class Depot. This class needs to override issueTome - Because SpellBook

can't be issued if its only for reference. SpellBook can't be issued for 13 or more days. ''' 42. A

04. 24. 2019

```
//code-block 42.A
class SpellBook extends Tome {
    boolean onlyForReference;
    SpellBook(boolean value) {
        onlyForReference = value;
    }
    @Override
    void issueTome(int days) {
    if (onlyForReference)
        System.out.println("Reference Tome");
    else
        if (days < 666)
            super.issueTome(days);
        else
            System.err.println("days => 666");
    }
}
```

''' Notice at 1, the annotation @Override This notifies the compiler that the method overrides a base class

The @Override annotation is optional, but comes in handy — when you try to override a method the wrong way.

The code at 2. defines issueTome() with the same name & method params as defined in class Depot.

The code at 3. calls method issueTome(), defined in class Depot ~ But it isn't mandatory to do so. It depends on whether the derived class wants to execute the same code as defined in the Base class.

Important → 04.26.2019

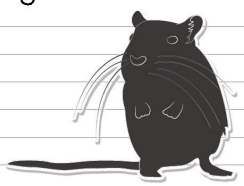
Whenever you intend to override methods in a derived class, use the @Override annotation. The compiler/linker will warn you if a method can't be overriden - or if you are actually overloading a method rather than overriding it.

e.g.
```
class TomeExmp {
    public static void main(String[] args) {
        Depot d = new Spellbook(true);
        d.issueTome(true);
        d.issueTome(1000);
        d = new Spellbook(false);
        d.issueTome(200);
        d = newTome();
        d.issueTome(200);
    }
}
```

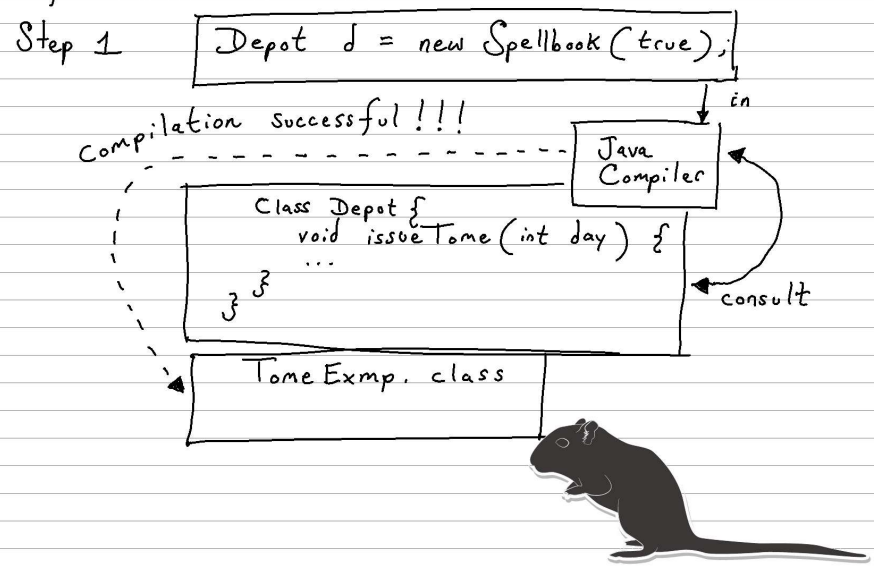
Prints "Reference Tome"

Prints "days >= 14"

Prints Tome issued

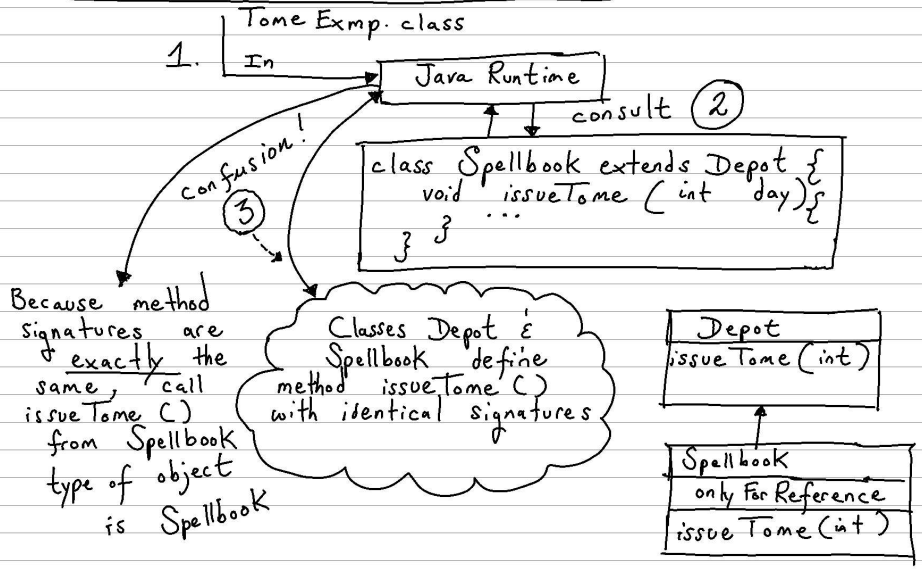


- Step 1 : The compile time uses the reference type for the method check
- Step 2 : The runtime uses the instance type for method invocation



Step 2 04.26.2019

```
Depot d = new Spellbook (true);
d.issueTome (100);
```



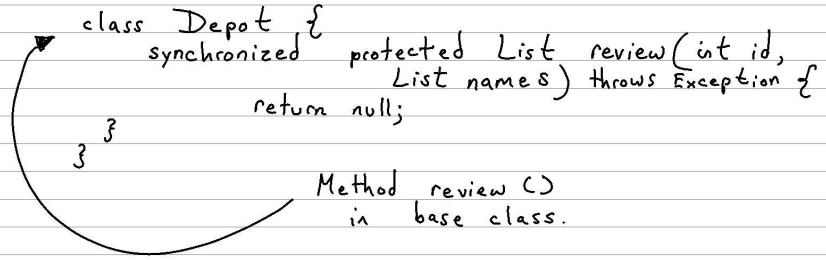
↑↑↑ 04.26.2019

To compile d.issueTome(), the Java Compiler refers only to the definition of class Depot. To execute d.issueTome(), The JRE uses the actual method implementation of issueTome() from Spellbook class.

More Java Class design → graph 44.a

''' Syntax of overriding methods

Look at this overriden method review()



```
class Spellbook extends Depot {   ϕ4.26.2ϕ19
  @Override
  final public ArrayList review(int id,
                                List names) throws IOException
                                                    {
                                return null;
                              }
}
```

overriden method review()
in derived class Spellbook

Method Components & their acceptable values for an overri'dding method

graph 44.b

*The rule on Exceptions in overriding methods only applies to checked exceptions, An overriding method can throw any unchecked exception i.e (RuntimeException or Error) ϕ4.26.2ϕ19 even if the overriden method doesn't. The unchecked exceptions ain't part of the method signature & are not checked by the compiler

Important Exam points on overridden method combination that are not valid.

You may not see @Overriden on the exam!

- Access Modifiers - A derived class can assign the same or more access but not a weaker access to the overriding method in the derived class:

```
class Tome {
  protected void review(int id, List names) {}
}
class Spellbook extends Tome {
    void review(int id, List names) {}
}
```

Won't compile! overriding methods in derived can't use a weaker access

Non access Modifiers 04.26.2019

A derived class can't override a base class method marked 'final.'

```
class Tome {
    final void review(int id, List names) {}
}
class Spellbook extends Tome {
    void review(int id, List names) {}
}
```

* Won't compile! final methods can't be overriden.

Argument list & covariant Return Types

a Covariant return type: When the overriding method returns a subclass of the return type of the overriden method, it is known as a covarient return type

This important to remember → To override a method, the param list of the methods in the base & derived classes must be the same. Otherwise you would then be overloading the method.

i.e use covariant Return types in the arg list you'll end up overloading the methods & not overriding them. 04.26.2019

e.g.

```
class Tome {
    void review(int id, List name)
    throws Exception {
        System.err.println("Base: review");
    }
}
```

[argument list → int, List]

```
class Spellbook extends Tome {
    void review(int id, ArrayList names)
    throws IOException {
        System.err.println("Derived: Review");
    }
}
```

@ 1, method review() in base class Tome accepts an object of type List. Method review() in derived class Spellbook accepts a subtype ArrayList (ArrayList implements List). These methods are not overridden, rather they are overloaded. 04.26.2019

```
class Verify {
    public static void main(String[] args) throws
    Exception
```

Arguments & Covariant Return type 05.01.2019
cont

```
class Verify {
    public static void main (String[] args)
throws Exception {
        Tome tome = new Spellbook(); ①
        tome.review(1, null);
    }
}
```

calls review in Tome; prints "Base: Review"

reference variable of type Tome used to refer to object Spellbook

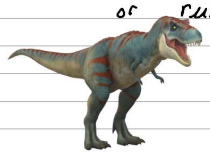
The code at ① uses a reference variable of type Tome to refer to an object of type Spellbook. The compilation process assigns execution of method review() from base class Tome to the reference value tome. Because method review() in class Spellbook does not override the →

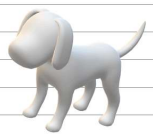
→ the review method in class Tome, the JRE doesn't have any "confusion" regarding whether to call method review() from class Tome or from class → SpellBook → The JRE moves forward with calling review() from Tome. 05. 01. 2019

★ Remember: It is the reference type that dictates which overloaded method will be chosen. This choice is made at compilation time.

{Exceptions Thrown} → An overriding method must either declare to throw no exception, the same exception, or subtype of the exception declared to be thrown by the base class method → it will fail to compile.}

This, however, does not apply to error classes or runtime exceptions.





06. 19. 2019

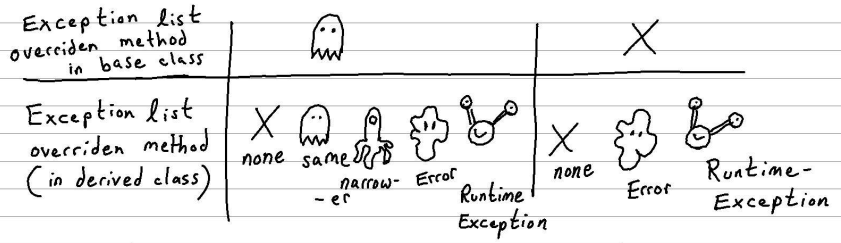
```
class Tome {
  void review() throws Exception { }
  void read() throws Exception { }
  void close() throws Exception { }
  void write() throws NullPointerException { }
  void skip() throws IOException { }
  void modify() { }
}
  class SpellBook extends Tome {
    void review() { }
    void read() throws IOException { }
    void close() throws Error { }
    void write() throws RuntimeException { }
    void skip() throws Exception { }
    void modify() throws IOException { }
```

Does not compile, declares to throw Exception - which is a superclass of IOException An Overriden method can't declare to throw broader exceptions - than declared to be thrown by

Does not compile declares to throw IOException → overriding method can't declare to throw a checked exception if, overridden method doesn't

→ overriden method. 06.19.2019

> An overriding method can declare to throw any RuntimeException or Error, even if the overriden method does not.



☆ When comparing exceptions to monsters → i.e when an overriden method declares to throw a checked exception (a monster), the overriding method can declare to throw a checked exception (a monster), the overriding method can declare to throw none, the same, or a "narrower" checked exception. An overriding method can declare to throw any Error or RuntimeException

☆ As mentioned before "an overriding method must either declare to throw no exception, the same exception or subtype of the exception declared to be thrown by the base-class method, or else it will fail compilation. 06.19.2019
And again → this rule does not apply to error classes or runtime exceptions. e.g. overriding method, when the overridden method doesn't declare to throw a checked exception & when it declares to throw a checked exception.
(1.3.3)

☆ Can you override all methods from the base class or invoke them virtually?

No, you may not! → You can override only the following methods from the base class

- Methods accessible to a derived class
- Nonstatic base class methods

Methods Accessible To a Base ϕ6. 19. 2ϕ19
→ base - class

The accessibility of a method in a derived class depends on its access modifier. e.g. a private method defined in a base class isn't available to a derived class in another package. A class can't override the methods that it can't access.

Only Nonstatic Methods can be overriden

If a derived class defines a static method with the same name & signature as the one defined in its base class, it hides its base class method and doesn't override it. You can't override static methods.

static Method in base class →

```
class Tome {
    static void printTitle () {
        System.out.println ("Necronomicon ex-Mortis");
    }
}
```

ϕ6. 19. 2ϕ19

```
class SpellBook extends Tome {
    static void printTitle () {
        System.out.println (" Spell Book ");
    }
}
```

static method in a derived class

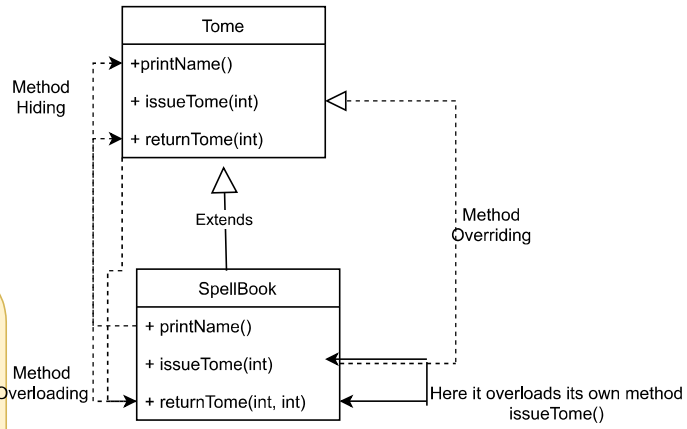
Method printTitle () in class SpellBook hides printTitle in class Tome. It doesn't override it. Because the static methods are bound at compile time, the method printTitle () that's called depends on the type of the reference variable :

```
class TomeExampleStaticMethod {
    public static void main (String [] args) {
        Tome base = new Tome();
        base.printTitle ();          // Prints "Necronomicon ex-mortis"
        Tome derived = new SpellBook();
        derived.printTitle ();
    }
}
```

graph ''' 49. A ϕ6. 19. 2ϕ19

```
class Tome {
    public static void printName() {

 }
   public int issueTome () {

 }
   public class int returnTome() {
   }
}
```

```
class SpellBook {
   public static void printName() {

 }
 public int issueTome(int days) {

 }
public int issueTome() {

 }
  public int returnTome( int a, int b) {

 }

}
```



1.3.4 → Identifying method overriding, overloading & hiding.

It's easy to get confused with Method Overloading, overriding, & hiding.

The above diagram helps identifying these methods in classes Tome & SpellBook. On the right side are the class definitions, and on the left their UML representations.

☆ When a class extends another class, it can overload, override, or hide its base class methods. A class can't override or hide its own methods — it can only overload its own methods

a)
```
class Tome {
    static void print() { }
}
class SpellBook extends Tome {
    static void print() { }
}
```

b). class Tome {
static void print () { }
}
class SpellBook extends Tome {
void print () { }
}

c). class Tome {
void print () { }
}
class SpellBook extends Tome {
static void print() { }
}

d) class Tome {
void print () { }
}
class SpellBook extends Tome {
void print () { }
}

Identify which are Overriden Print () method ✱
Which is hidden print () method ✱
Compilation error ⊗

a is hidden, b ⊗, c, ⊗, d. overridden!

1.3.5 Can you override base class constructors or invoke them virtually? 06.20.2019

No! Constructors aren't inherited by a derived class. Only inherited methods can be overridden, constructors cannot be overriden by a derived class. Lookout for questions that try to trick you e.g. a question that queries you on overriding a base class constructor.

- Constructors can't be overriden because a base class constructor isn't inherited by a derived class.
- It is important to override the methods of class java.lang.object

(1.4) Overriding methods of class object

↳ 1.1.61 Override methods from the Object class to improve the functionality of your class.

All the classes in Java - classes from the Java API, user defined classes, or classes from any other

API → extend class java.lang.Object, §6.2§.2§19 either implicity or explicitly. Section talks about overriding the methods from class Object → look into & - final & nonfinal & final methods

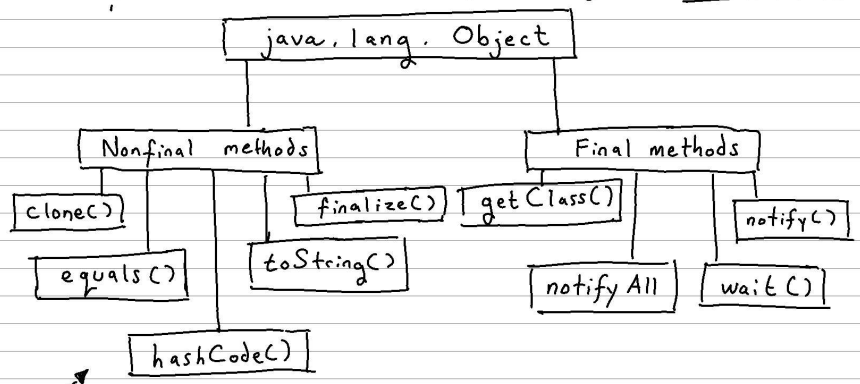
o As you have less control over who uses your class & how it's, the importance of correctly overriding methods from class Object class methods & so that these classes can be used efficiently by other users. Incorrect overriden methods can result in increased debug time.

* Final classes can't be overriden → discuss the nonfinal methods of class Object.

They are the following methods → clone(), equals(), hashCode(), toString(), & finalize() → define a contract, a set of rules on how to override these methods, → specified by the Java API docs.

(1.4.1)

Overriding method toString() → Method toString() is called when you try to print out the value of a referenc variable or use a reference variable in a concat operator. The default implementation of method toString() returns the name class by @ & the hash code of the object it represents. §6.2§. 2§19



Categorization of final & non-final methods of class java.lang.Object

The code of method toString() 06.20.2019
defined in class Object in the Java API

```
public String toString() {
    return getClass.getName() + "@" + Integer.
  toHexString(hashCode());
```

Following is an example of class Tome, which does not override method toString() ~ In this case, a request to print the reference variable of this class will call method toString() defined in class Object:

```
class Tome {
    String title;
}
class PrintTome {
    public static void main (String[] args) {
        Tome tome = new Tome();
        System.out.println(tome);
    } }
```

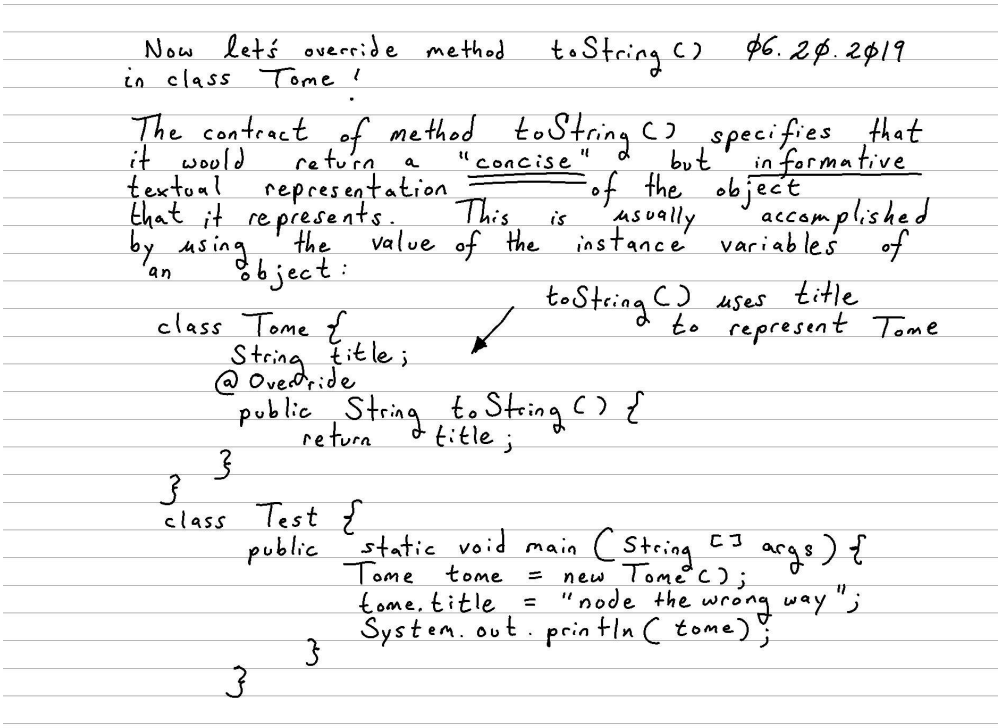
Prints something like Tome@6875309

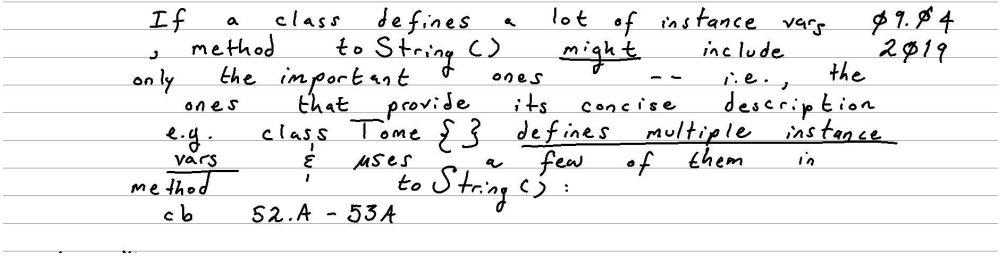
Now let's override method toString() 06.20.2019
in class Tome!

The contract of method toString() specifies that it would return a "concise" but informative textual representation of the object that it represents. This is usually accomplished by using the value of the instance variables of an object:

toString() uses title to represent Tome

```
class Tome {
    String title;
    @Override
    public String toString() {
        return title;
    }
}
class Test {
    public static void main (String[] args) {
        Tome tome = new Tome();
        tome.title = "node the wrong way";
        System.out.println(tome);
    }
}
```






```
//codeblock 52.A - 53
class Tome {
    String title;
    String isbn;
    String[] author;
    java.util.Date.publishDate;
    double price;
    int version;
    String publisher;
    boolean eBookVersion;
    @Override
    public String toString() {
        return title + ", ISBN:"+isbn + ", Lead Author:"+author[0];
    }
}

class Testing {
    public static void main(String[] args) {
        Tome tome = new Tome();
        tome.title = "Necronomicon Ex-mortis";
```

```
        tome.author = new String[] {"Bruce", "Campbell"};
        tome.isbn = "666";
        System.err.println(tome); // Prints Necronomicon Ex-mortis, ISNB:666, Lea
    }
}
```

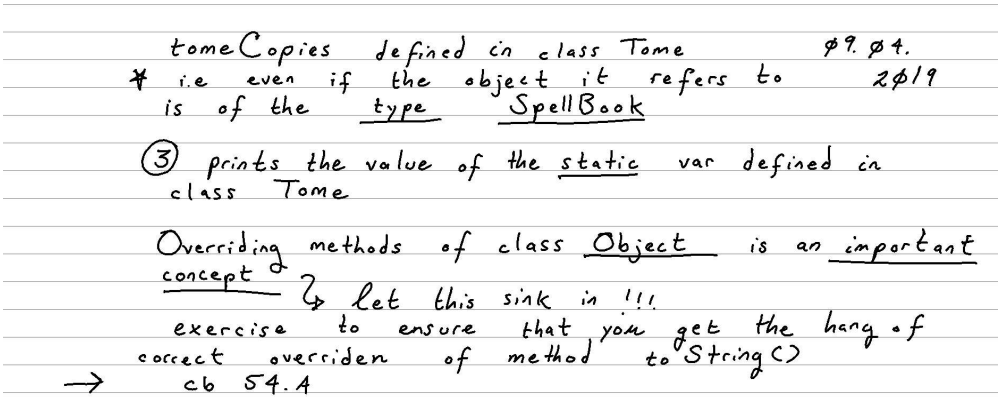
↳ '''
here the method toString() is "inappropriately" overriden
-- i.e if it returns any text that's specific
to a particular class, e.g. the name of a
class or a value of a static variable:
→ cb 53B
'''

```
//codeblock 53B
class Tome {
    String title;
    static int tomeCopies = 666;
    @Override
    public String toString() {
        return title + ", Copies: " + tomeCopies; //Overrides toString() uses sta
    }
}

class SpellBook extends Tome {
    static int tomeCopies = 42; // static variable tomeCopies also defined in Spe
}

class TomeOverrideToString {
    public static void main(String[] args) {
    SpellBook spellbook = new SpellBook();
    spellbook.title = "Necronomicon Ex-mortis";
    System.err.println(spellbook); // Prints "Necronomicon Ex-mortis:666"
```

'''
① → shows inappropriate overriding of method toString()
because it uses a static var

② defines a static var tomeCopies in class SpellBook

★ Because static members are bound at compile
time the method toString() will refer to
the var '→




```
//codeblock 54A
class Tome1 {
    String title;
    int copies = 1000;
    public String toString() {
        return "Class Tome, Title: " + title;
    }
}

class Tome2 {
    String title;
    int copies - 1000;
    public String toString() {
        return ""+copies * 42;
    }
}

class Tome3 {
    String title;
    int copies = 1000;
    public String toString() {
        return title;
    }
}

class Tome4 {
    String title;
    int copies = 1000;
    public String toString() {
        return getClass().getName() + ":" + title;
    }
}
```

(1.4.2) Overriden method equals()

Method equals() is used to determine whether 2 objects of a class should be considered equal or not

* The method equals() returns a boolean value that determines whether two objects should be considered equal or not!

The default implementation of method equals() in class Object compares the 2 object references & returns true if both reference vars refer to the same obj, false otherwise. 09.04.2019

Meaning → it only returns true if an object is compared to itself.

Following is the default implementation of method equals() in class java.lang.Object

e.g.
```
public boolean equals(Object obj) {
    return (this == obj)
}
```

↳ The OCA-1Z0-804 → will question us on the following

- The need to override method equals()
- Overriding method equals() correctly
- Overiding method equals() incorrectly

The Need to Override Method equals()

09.07.2019

* You need to override method equals() for objects that we wish to equate logically which normally depends on the state of the object i.e. the value of its instance variables.

The goal of overriding method equals() is to check for equality of the objects

* Not to check for the same variable references.

eg. We have two objects obj1, obj2,

the equal() checks whether obj1 is logically equal to obj2

important: In this case obj1 is not pointing to the exact same object as obj2.

e.g. class String { } overrides equals() to check whether 2 String objects define the exact same sequence of chars 09.07.2019

```
String name1 = "Rick";
String name2 = new String("Rick");
System.err.println(name1.equals(name2));

// true
```

e.g. You need to find out whether the same undergrad course is or not offered by multiple Universities.

Represent the Univ. as a class, e.g. Univ { }, each course being offered using a class, e.g. Course { }

Now Assume that every Univ, offers a list of courses → we can override equals() in class Course { } to determine if two Course objects can be considered equal,
cb 55A

Rules For Overriding Method equals() 09.07.2019

- Method equals() defines an elaborate contract (set of rules), e.g. From the JAVA API doc

1. It is reflexive - For any non-null reference value x, x.equals(x) -- should return True. This rule states that an object should be equal to itself.

2. It is symmetric - For any non-null reference values x & y, x.equals(y) should return true if & only if y.equals(x) return true. This rule states that two objects should be comparable to each other in the same way.

3. It's transitive - For any non-null reference values, x, y, & z, if x.equals(y) returns true & y.equals(z) returns true & y.equals(z) returns True then x.equals(z) = true

The rule states that while comparing objects, you shouldn't selectively compare the values based on the type of an object. 09.07 2019

4. It's consistent - For any non-null reference values x & y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.

This rule states that method equals() should rely on the value of instance variables that can be accessed from the memory & should not try to rely on values like dynamic vals - e.g. IPs (new IPs can be assigned, duh!)

5. For any non-null reference value x, x.equals(null) should return false.

Meaning: that a non-null object can never be equal to null.

Make sure you remember this → 09.07.2019

i.e. ↳ The Rules For Overriding Equals()

1. It's reflexive, 2. Symmetric

3. Transitive, 4. consistent

(5) ~ (null == !null) = false;

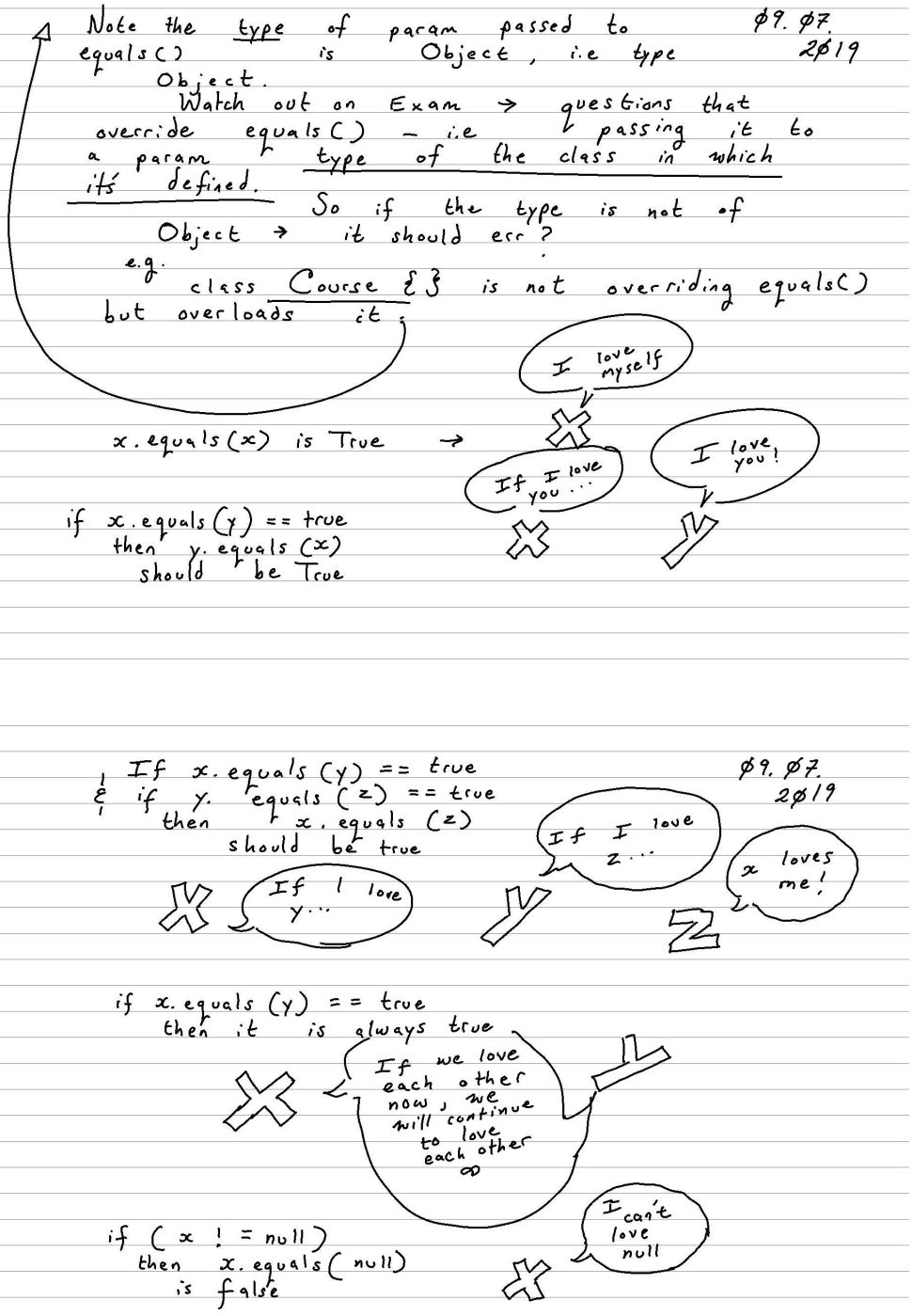
Use this way to remember → compare equals to love. (emotional connection)

When you see x.equal(x) → read this as

x.loves(x) if x.equals(y) = True, then y.equals(x) must return True →
so " if x loves y, then y loves x)

↘ Correct & incorrect Overriding of Method equals()

cb 56A • To override toString() correctly, follow the method overiding rules.



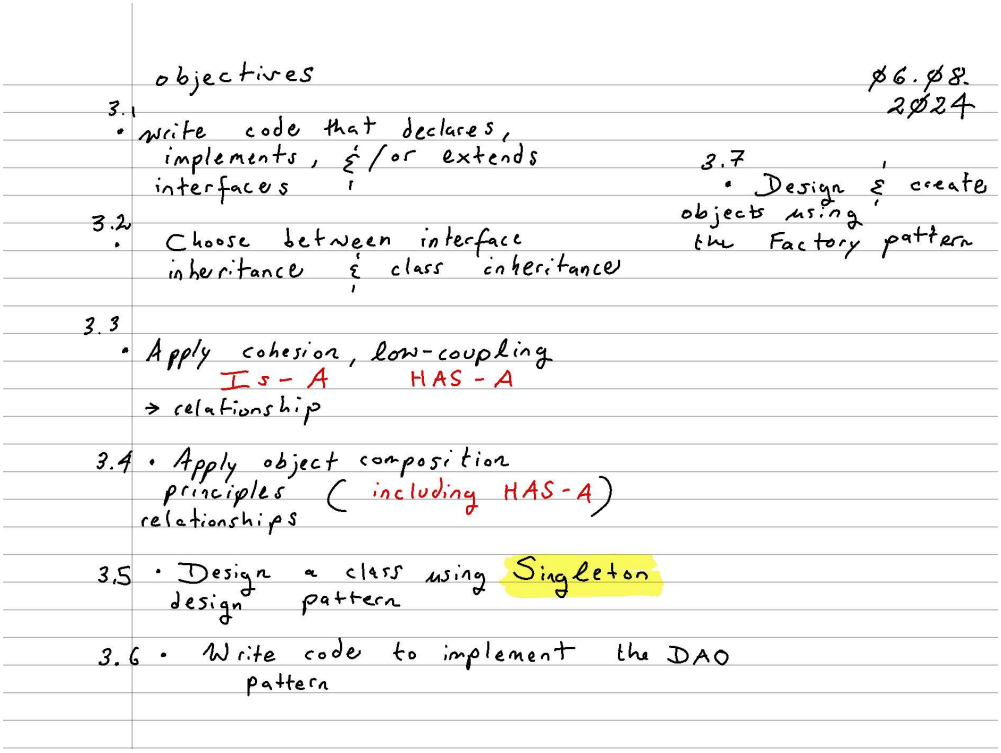
2 Advanced Class Design

- [ ] Identify when and how to apply abstract classes
- [ ] Construct abstract Java classes and subclasses
- [ ] Use the static and final keywords

- [ ] Create top-level and nested subclasses
- [ ] Use enumerated types

3 Object-oriented Design Principles

- [ ] Write code that declares, implements, and/or extends interfaces
- [ ] Choose between interface inheritance and class inheritance
- [ ] Apply cohesion, low-coupling, IS-A, and HAS-A Principles
- [ ] Apply object composition Principles (including HAS-A relationships)
- [ ] Design a class using the **the Singleton design pattern**
- [ ] Design and create objects using the **Factory Pattern**



Need to Know → understand 06.08 2024

3.1 . The need for interfaces. How to declare, implement, & extend interfaces. Implications of implicit modifiers that are added to an interface & its members

3.2 . The difference & similarities between implementing inheritance by using interfaces & by using abstract or concrete classes.
Factors that favor using interface inheritance over class inheritance & vice versa

3.3 . Given a set of IS-A & HAS-A relationships, how to implement them in code.
Given code snippets → how to correctly identify the relationships IS-A or HAS-A → implemented by them

. How to identify & promote low coupling & high cohesion

3.4 . Given that an object can be composed of multiple other objects, how to determine the types of compositions -- & implement them in code. 06.08. 2024

3.5. How to implement the Singleton design pattern. The need for existence of exactly one copy of a class.

3.6 . The usability of the DOA pattern. How this pattern enables seperation of data access code in an app

.. What can be the secret(s) behind the most successful people? their habits →

. Never hit the snooze button when the alarm goes off in the morning →

. First thing first → prioritize your works

- Follow your passion & do what you love 06.08.2024
→ because you'll be working almost all your life.

These principles that "successful" people follow (though perhaps in different manner) to achieve the greatest hieght of success.

Similarly → in O-O design principles → enable you to create better app(s) design → which are manageable & extensible
eg → as a dev / designer → we know that app requirements typically change →

★ This happens to Sel-webDriver frameworks (★ think of Page-Object-Model)

If your App / frameworks design uses the the design principles of "low coupling & high cohesion" → chances are low that changes → in a class will affect another class.

Design Patterns → the Singleton pattern 06.08.2024 also help you design better apps. Unlike inheritance → a design pattern is an example of experience reuse NOT code reuse.

Building sloping roofs in areas that receive a lot of snowfall can be compared to using a design pattern.

★ A sloping roof was identified as a solution to avoid accumulation of snow on rooftops after multiple folks faced issues with flat roofs

Just as a sloping roof is applicable specifically to areas receiving snowfall, a design pattern resolves a specific design issue.

The OCA exam will test you on what OO design principles are & how to apply them in your apps

★ You will find questions on the creation of & prefered use of classes

& interfaces to design your app & how 06.08.2024
to relate & use Java objects together.

- Interfaces → their declaration, implementation & application
- Choosing between class inheritance & interface inheritance
- Relationships between Java objects
- Application of object composition principles
- Implementation of IS-A & HAS-A relationships between objects
- Singleton design Pattern
- Data Access Object (DOA) design pattern
- Factory Design Pattern

Interfaces are one of the most powerful concepts in Java → 06.08.2024

Not many designers completely understand how to use them effectively.

To be a good app designer → you must know how to do so.

how many ways can one refer to their father ?
Apart from being their father, they could be refered to as friend, guide, husband, swimmer, orator, teacher, manager etc

How can you acheive the same thing in Java → to refer to the same object by using multiple types ?

For this you need to learn about the need for using interfaces
↳ followed by declaring, implementing, & extending

3,1 Interfaces → 06.08 2024

Let's write some code that declares → implements & / or extends interfaces

Deep Dive → the term interface has multiple meanings. 1st, an interface is a TYPE created by using the keyword ⟹ interface e.g. →

```
interface Moveable {
    void move();
}
```

The general meaning → 2nd → It's how two systems can interact with each other (like a T.V. & its remote)

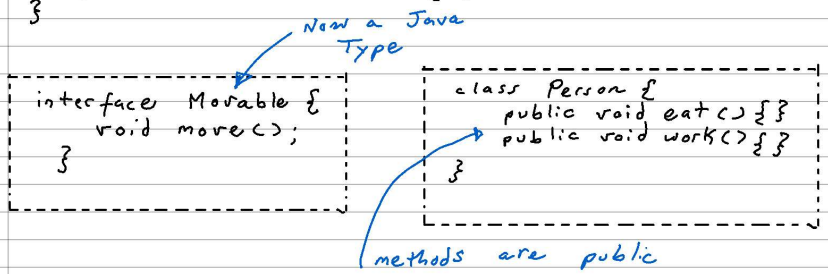
* It's how classes can interact with each other, using their public methods. e.g →

class Person → its an interface notice → the public methods

.. refers to its methods eat() & work()

06.12. 2024

```
class Person {
    public void eat() {}
    public void work() {}
}
```



observe → Moveable is a type, & methods are public ...

Why are methods public? It would not make sense if the methods are private → other classes won't be able to use them.

3.1.1 Lets' understand Interfaces ϕ6.12. 2ϕ24

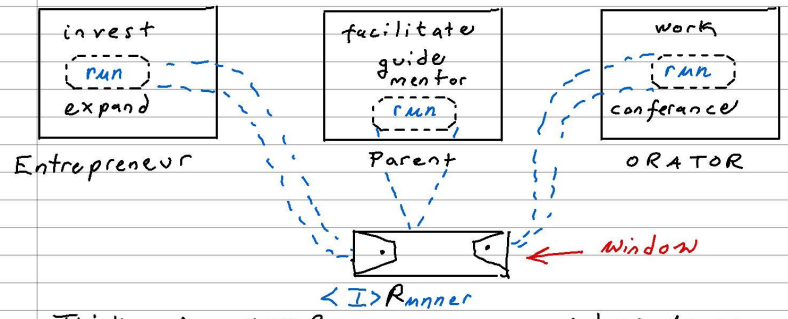
when you refer to your "manager" as a "runner" would you care if your manager was a parent, an orator or an entrepreneur?
But as a person → he/she is also able to run.
"The term runner enables you to refer to unrelated individuals, by opening a small window to each person & accessing behavior that's applicable to only that person's capacity as a runner.
★ Someone can be referred to as a runner only if that person supports characteristics relevant to running, though the specific behavior can depend on the person.

you can compare the term runner to a Java interface → which defines the required behavior → run()

★ !Remember → An interface can define a set of behaviors ... methods() & constants properties that don't change

An Interface delegates the implementation of the behavior to the classes that implement it. ϕ6.12. 2ϕ24

★ Interfaces are used to refer to multiple related or unrelated objects that share the same set of behavior.



Think of <I>Runner as a window to an object which is concerned only about the running capabilities of that object

.. when you design your app by using interfaces, you can use similar "windows" also referred to as specifications or contracts to specify the behavior that we need from an object, without caring about specific type of objects. 06.12.2024

What are the "many" benefits interfaces → Seperating the required behavior from its implementation has many benifits.

★ As an app designer, we can use interfaces to establish the behavior that's required from objects, promoting flexibility in the design (new classes that implement an interface can be created & used later.

Interfaces make an app manageable, extensible & less prone to propagation of errors due to changes to existing types

06.12.2024

```
interface Runner {
 int speed();
 double distance = 42;
}
```

Becomes

```
interface Runner {
    public abstract int speed();
    public static final double distance = 42;
}
```

All the methods of an interface are implicitly public & abstract. Its vars are implicitly → public, static & final.

3.1.2 Declaring interfaces §6.12 2024

We can define methods & constants in an Interface. constants'? → only? Simple to declare Interfaces. But for the OCA OCP-SE7 1Z0-804 it is important to understand the implicit modifiers that are added to the members of an interface.

Remember → all methods of an interface are implicitly public & abstract, all members, i.e. the vars, are implicitly public, static & final.

Look @ the interface Runner that defines a method speed() & var distance()

```
interface Runner {
  int speed();
  double distance = 42;
}
```

Becomes

```
interface Runner {
  public abstract int speed();
  public static final double distance = 42;
}
```

see how implicit modifiers are added to the members of <I> Runner → during the compilation process

```
}
```

ok? Why does this happen? §6.12. 2024

Because an interface is used to define a contract... it doesn't make sense to limit access to its members -- right, the methods & vars should be public.

Remember → an interface can't be instantiated you cannot create objects of Interfaces.. Remember static classes are not instantiated. Because <I> can't be instantiated, so, the value of its vars should be defined & accessible in a static context → which makes them implicitly static.

Because an interface is a contract, its implementation should NOT be able to change it, so the interface vars are implicitly final

Interface methods are implicitly abstract so that it is mandatory for classes to implement them. Remember all methods are abstract & NOT implemented.

Recall that abstract classes can contain both abstract & concrete methods & must contain at least one method, can either be abstract or concrete

§6.12, 2024

- An abstract class might not necessarily define an abstract method & can exist without any abstract method
- An abstract class can define multiple constructors

★ A final class can't be extended. But Abstract classes are "meant" to be extended by other classes.

The OCA exam will test us on the various components of an interface declaration ... which will include access & none-access modifiers

- Access modifiers
- Non access modifiers
- Interface name
- All extended interfaces, if the interface is extending any interfaces

- All extended interfaces, if the interface is extending any interfaces

§6.12, 2024

- Interface body (vars & methods), included within a pair of {}

Recall that abstract class can be extended only once! While Interfaces can be extended multiple times.

ok, lets take a look → modify the declaration of the interface Runner :

```
public strictfp interface Runner extends Athlete, Walker {}
```

public	strictfp	interface	Runner	extends	Athlete, Walker	{}
Access modifier	Non-Access modifier	Keyword	Interface name	Keyword	Name of interfaces extended by interface Runner	Curly Braces
optional	optional	↑ compulsory		optional	optional	compulsory

★ To understand this → §6.12 2024

The components of the interface Runner

To declare any interface, you must include the keyword interface must contain name, must contain body {}

Compulsory	Optional
Keyword interface	Access modifier
Name of interface	Non access modifier
Body {}	Keyword extends together with the name of the 'base interface (s) interfaces can extend multiple interfaces!

★ The declaration of an interface can't include a class name. An interface can never extend any class

You can NOT define a top-level protected interface

--> you must know the answer to such questions on the OCA exam!

deep dive into the nuances of Interface declaration §6.12 2024

Valid Access Modifiers for an Interface

you can declare a top-level interface

The one that isn't declared with any other class or interface → with the following access levels

- public
- No modifier - default access

You cannot declare top-level interfaces by using other access modifiers such as protected or private

private interface StanInterface {}
protected interface StanInterface {}
} NO

Tip → All the top-level Java types §6.12, 2024

↳ classes, enums, interfaces

↳ can be declared using only two access levels:

↳ public & default

Inner or nested types can be declared using any access level. That's right inner classes can be of any access ->

```
interface StanInterface {
    private int numb = 42; ← will not compile
    protected void stanMethod(); ← will not compile
    interface interface2{} ← ok, it will be implicitly public
    public interface interface3{}
}
```

This is ok!

Special Characteristics of Method & Variables of an Interface §6.16. 2024

Methods in interfaces are public & abstract by default.

observe →

int getMembers(); = public abstract int getMembers();

★ Vars defined in interfaces are public, static & final by default

↳ int maxMembers = 42; is the same as public static final int

you see that static members cannot be made to objects

This also means that all your vars are constant by default - as well

go-ahead → see what happens when you do NOT init these constants. ∅6.17, 2∅24

```
interface StanInterface {
    int number;
}
```

will not compile

you will need to initialize vars in an Interface

Valid Non Access Modifiers for an Instance

you can declare a top-level interface with only the following nonaccess modifiers

abstract

strictfp ← the strictfp keyword guarantees that results of all floating-point calculations are identical on all platforms.

If you declare your top-level interfaces by using the other nonaccess modifiers such as → final, static, transient, synchronized or volatile ∅6.17 2∅24

The interface will not compile.

A nested interface can be defined using the nonaccess static only static for nested interface like so →

```
class Outer {
    static interface StunInterface { }
}
```

nested interface is inside the class.

3.1.3 Implementing interfaces

You can compare implementing an interface to signing a contract!

When a concrete class declares its implementation of an interface → it agrees to implement all of its abstract methods.

A class can implement multiple interfaces §6.17. 2024

↳ e.g. →

```
interface Livable {
    void live();
}
interface GuestHouse {
    void welcome();
}
class Home implements Livable, GuestHouse{
    public void live() {}
    public void welcome() {}
}
```

abstract method live()

abstract method welcome()

Here the Home class uses the implements keyword

If you don't implement all the methods defined in the implemented interfaces, a class can't compile as a concrete class.

like so →

```
class Home implements Livable, GuestHouse {
    public void welcome() {}
}
```

won't compile

so → you have two methods defined §6.17
in the interface Liveable {} & GuestHouse {} 2024

↳ & here you are only implementing one of those methods

```
class Home implements Livable, GuestHouse {
    public void welcome() {}
}
// public void live(){}
```

missing this method, which was defined in the Livable {} interface.

* So a class "can" choose not to implement all the methods from the implemented interface(s) & still compile successfully

→ ** only if it's defined as an abstract class

06.17.2024

```
abstract class Home implements Liveable,
    GuestHouse {
        public void welcome() { }
    }
```

notice → abstract class doesn't have to implement all methods from implemented interfaces

TIP → A concrete class must implement all the methods from the interface(s) that it implements

An abstract class can "choose" not to implement all the methods from the interfaces that it implements.

Defining & Accessing Variables with the same Name

06.17.2024

A class can define an instance of a static var with the same name as the var defined in the interface that it implements.

e.g. → the following class →

the interface Liveable defines vars status & ratings.

Class Home implements Livable & defines instance vars status & static vars ratings, with a default access level:

```
interface Livable {
    boolean status = true;
    int ratings = 11;
}
```

```
class Home implements Liveable {                    §6, 17. 2024
    boolean status;
    static int ratings = 42;                  ] vars with default access
    Home() {
        System.out.println(status);
        System.out.println(Livable.status);
        System.out.println(ratings);
        System.out.println(Livable.ratings);
    }
```

prints false

// true

// prints "42"

// prints 11

TIP > A class can define an instance or a static variable with the same name as the var defined in the interface that it implements. These vars can be defined using any access level.

Following Method Overriding Rules For Implementing Interface methods

The methods in an interface are public, by default. (implicitly) §6. 17. 2024

So, trying to assign weaker access to the implemented method in a class won't compile:

```
interface Livable {
    void live();          <-- public method
}
class Home implements Livable {
    void live() {
    }
```

won't compile ⇒ because you are implementing live() using weaker access

TIP > Because interface methods are implicitly public, the implementing class must implement them as public methods, or else the class will NOT compile!

Implementing Multiple Interfaces that define methods with the same Name

Methods in an interfaces don't define any implementation; they come without any "baggage." But what happens if a class implements multiple interfaces that define methods with the same name? §6.17. 2024

ok → add a method live() to interface GuestHouse

```
interface Livable {
    void live();
}
interface GuestHouse {
    void welcome();
    void live();
}
```

Interface Livable defines method live()

// Interface GuestHouse also defines a method live()

Class Home implements two interfaces Livable & GuestHouse → Both of which define a method live():

```
class Home implements Livable, GuestHouse {
    public void live() {
        System.out.println("live");
    }
    public void welcome() {
        System.out.println("welcome");
    }
}
```

§6.17 2024

Both the Java compiler & Java Runtime Environment are A-OK! with the preceding code.

Why? → Because the method signature of method live() is the same in both interfaces, Livable & GuestHouse → class Home needs to define only one implementation for method live() to fulfill both contracts

(interface implementations)

Overlapping Method Implementations with their overload Versions

A class can try to implement multiple interfaces that define methods with the same name. But doing so, we can have a not-so-pleasant (cocktail) of overlapping method implementations & their overloaded versions. §6.17, 2024

We have two scenarios →

- Correctly overloaded methods
- Invalid overloaded methods

Overloaded methods are defined by using the same name but a different parameter list. e.g → when implemented in class Home, method live() defined in the interface Livable overloads method live() defined in the interface GuestHouse.

class Home must implement both these methods:

§6.17 2024

```
interface Livable {
    void live();
}

interface GuestHouse {
    void live(int days);
}

class Home implements Livable, GuestHouse {
    public void live() {
        System.out.println("live");
    }
    public void live(int days) {
        System.out.println("live for " + days);
    }
}
```

No params to be accepted

live() accepts a param

Correctly overloaded method

live() method

Correctly overloaded method live() from GuestHouse

you can't define overloaded methods by changing only the return type of methods. 06.17 2024

So, what will happen if live() in the interface Livable {} & GuestHouse {} returns different types ?

In this case → class Home needs to implement both versions of method live(), which can't be qualified as overloaded methods. So in this case → Home won't compile !

returns String
returns nothing
won't compile i.e. class Home will NOT compile

```
interface Livable {
    String live();
}
interface GuestHouse {
    void live();
}
class Home implements Livable, GuestHouse {
    public String live() {
        return null;
    }
    public void live() {
        System.out.println("live");
    }
}
```

06.17, 2024

When implemented in class Home, both versions of live() qualify as incorrectly overloaded methods.

TIP > A class can implement methods with the same name from multiple interfaces But these must qualify as correctly overloaded methods

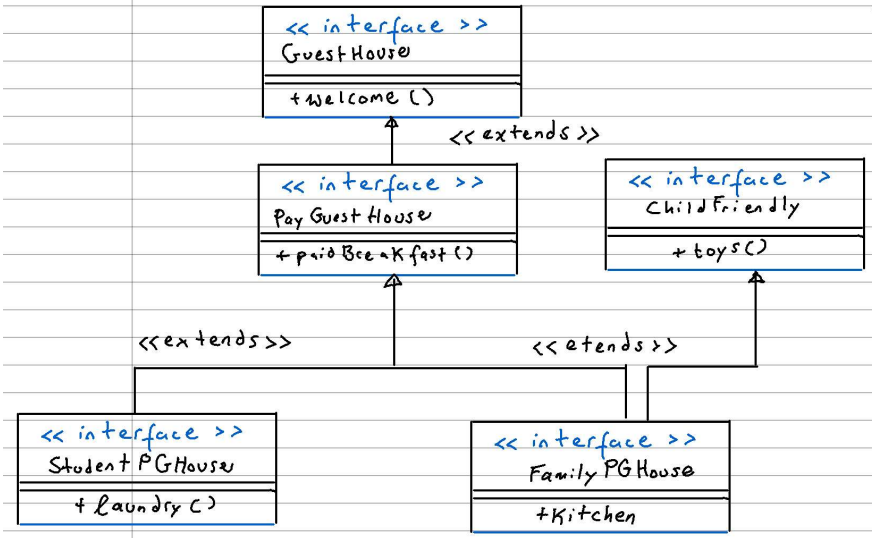
3.1.4 Extending Interfaces

An interface can inherit multiple interfaces. Because all the members of an

Interface are implicitly public 06.17.2024
a derived interface inherits all the methods of its super-interface(s)
An interface uses the keyword extends to inherit an interface

```
interface GuestHouse {
    void welcome();
}
interface PayingGuestHouse extends GuestHouse {
    void paidBreakfast();
}
interface StudentPGHouse extends PayingGuestHouse {
    void laundry();
}
interface ChildFriendly {
    void toys();
}
interface FamilyPGHouse extends ChildFriendly,
        PayingGuestHouse {
    void kitchen();
}
```

06.19.2024



By extending interfaces, you can combine methods of multiple interfaces. 06.19.24

observe → the interface FamilyPGHouse combines the methods of the interfaces → ChildFriendly, PayingGuestHouse & GuestHouse!

When a class implements an interface, it should implement all methods of the interfaces (defined) in the interface & its base interfaces, unless it's declared as abstract. If, e.g. → {} a class implements the interface PayingGuestHouse, that class must implement method paidBreakfast() → which is defined in the interface → PayingGuestHouse{} & method welcome() defined in the interface GuestHouse{} → what is a concrete example of a class implementing interfaces? * let's look into that later.

Rules To IRemember about Interfaces

- An interface is abstract by definition! 06.21.2024
- An interface can define only public, abstract methods & public, static-final vars.
- An interface uses the keyword extends to inherit other interfaces
- A class can implement multiple interfaces → so can inner classes!
- If a class doesn't implement all the methods of the interface that it implements, the class must be defined as an abstract class.
- If a class implements multiple interfaces that define methods with the same name, the interface methods must either qualify as correctly overloaded or overriden methods -- or the class will NOT compile.

Class inheritance vs. interface Inheritance

3.2. Choose between interface inheritance & class inheritance. §6.21. 2§24

A class can implement interface(s) & a class can extend a class & override its methods.

So → while designing classes & interfaces in our cool app, how do you implement inheritance to reuse existing code? Or → would we prefer that our class inherit another (abstract or concrete) class or implement an interface?

No right or wrong answer → not straight forward --> this will depend on the requirements → in which to extend a class or implement an interface → because each offers a different set of benefits.

How do you make this informed decision? here are some similarities & differences in both approaches.

3.2.1 Comparing class inheritance & interface inheritance §6.21. 2§24

An interface does NOT include implementation details

A class Does include implementation details

What about abstract class - ? No initialization?

	Class inheritance	Interface inheritance
Instantiation of derived class	Instantiation of a derived class instantiates its base class	Interfaces can't be instantiated.
how many	A class can extend only ONE base class	A class can implement multiple interfaces.
Reusing implementation details	A class can reuse the implementation details of its base-class.	An interface doesn't include implementation details
Modification to base class		

§6.

	Class Inheritance	Interface 21. inheritance 24
Modification to base-class implementation details	With the modified base class, a derived class might cease to offer the functionality it was originally created for; it may fail to compile	Once again, interfaces don't include implementation details.

for the similitarities between class & interface inheritance. In both cases, you can refer to a derived class or implementing class by using a variable of the base class or implementing interface.

Preferring class inheritance over interface inheritance

Class inheritance scores better when we want to reuse the implementation already defined in a base class.

§6.21. 2024.

Class inheritance also does well when we want to add new behavior to an existing base class →

Reusing the Implementation from the base class

When you create any class → you extend & Reuse class java.lang.Object. Is this the "cosmic object (cosmic class) The class Object defines code to take care of all the threading & object-locking issues, together → with providing default implementation for methods toString(), hashCode(), & equals()
We know what method toString() does → returns a textual representation (string) of an instance. While methods hashCode() → enables objects to be stored & retrieved efficiently in hash-based collection.

What if java.lang.Object was an interface ★ Think about this

This would then mean you would have to implement all these methods contained in that interface, IN java 8 you can use the default method not to break this rule. §6.21. §24.

Stick to creating class inheritance i.e → extending classes when you have certain boiler plate code which you wish to use across many implementation classes.

Adding New Behavior in ALL derived classes

say you have a set of entities, Lions, tigers & bears, you can identify their common behavior → & moved the common behavior to their common base class (Animal)

Because we control the definition of all these classes, we might add new behavior to your base class & make it available to all the derived classes. → Lets take a look ↓

→ the abstract class Animal & none-abstract class Lion → which extends the class Animal §6.21. §24

```
public abstract class Animal{
    public abstract void move();
    public abstract void live();

public class Hamster extends Animal {
    public void move() { /*... */}
    public void live() { /*... */}
```

OK, add another method to Animal

```
public abstract class Animal {
    public abstract void move();
    public abstract void live();
    public void eat() { /*... */ }

public class Gerbil extends Animal {
    public void move() { /*... */}
    public void live() { /*... */}
```

The addition of public method eat() in class Animal {} makes it available to all subclasses of Gerbil → implicitly. ¶6.21. 2024

But adding or modifying behavior in a base class is NOT always a straight forward process.

3.2.3

Preferring interface inheritance over class inheritance

Why would you prefer interface inheritance over class inheritance?

★ you would prefer interface inheritance over class inheritance → when you need to define multiple contracts for classes.

↳ This is important

Implementing Multiple Interfaces

Say you need to use a class that can be executed in a seperate thread & can be attached as an ActionListener() to a GUI component. We can achieve this by making our class implement multiple Interfaces that support these functionalities -- interfaces IRunnable & ActionListener: ¶6.21. 2024

```
class StanClass implements Runnable, ActionListener {
    //... code to implement methods from interface
    //... Runnable & ActionListener
}
```

Interface implementation has ONE major advantage: a class can implement multiple interfaces → this to support multiple functionality.

Here you can pass instances of class StanClass to all methods that define params of type Runnable or ActionListener

The dude abides

Defining A New Contract For Existing Classes to Abide By

maths Olympiad Parsoon Kumar 06.21.2024

In Java 7 → a new language feature has been added to the exception handling. try with resources? yes! try-with-resources statement.

Try-with-resources: Intent is to define a try statement that can use streams → that can be auto-closed - releasing any sys resources associated with them. This prevents Java objects (plain Ol' Java Objects?) from using resources that are no longer required. does this happen @ runtime? * yes, I think so.
Streams can be auto closed → releasing any sys resources associated with them.
Preventing Java Objects from using resources that are no longer required.
unused & unclosed resources can lead to resource 'leakage'

Though a try statement provides finally {} which will exec in the try/catch block always → {} is used to close streams.

Try-with-resources allows you to do this automagically. 06.21.2024

The objects that can be used with this statement need to define a close method → so this method can be called to automagically close & release used resources.
To 'apply this constraint → Java designers at Oracle started defining interface
java.lang.AutoClosable

```
package java.io;
import java.io.IOException;
    public interface Closable extends AutoClosable{
    public void close() throws IOException;
}
```

* Any user-defined class that implements the interface java.lang.AutoClosable or any of its subinterfaces can be used with a try-with-resources

e.g.
```
void openF(String fname) throws Exception{
try (FileInputStream fis = new FileInputStream(new File(
        filename))) {
```

06.21.2024

object of FileInputStream can be declared in try-with-resources because FileInputStream implements Closable (which extends AutoClosable)
you will visit try-block in try-with-resources in §6. can exist without any companion catch or finally block.

Interface inheritance added new behavior to classes like Reader & Writer without breaking their existing code. (default methods)
Inheritance of the interface AutoClosable by Closable defines multiple contracts for instances of these classes.
* They can now be assigned to a reference var of Type AutoClosable → enabling them to be used with a try-with-resources statement.
JDBC interface(s) implement/extend AutoClosable → (Connection, Statement, ResultSet) & several Java Sound API interfaces.

Fragile Derived Classes

06.21.2024

Adding to or modifying a base class can affect its decived class -- we know this
Adding new methods to a base class can result in breaking code of a derived class.
* We also know this.

→ e.g.
```
public abstract class Animal {
    void move() {}

    class Gerbil extends Animal {
        void live() {}
    }
```

ok, now a change → add a new method live() to base class Animal.

Because live() clashes (because of an incorrectly overriden method → with the existing (* name conflict) method live() in its derived class Gerbil. Gerbil will no longer compile

```
public abstract class Animal {                    ϕ6.21.
    void move() { }                               2ϕ24
    String live() {   <- new method added
        return "live";     to Animal
    }
    class Gerbil extends Animal {
        void live() { } <- this method does not
                          override live()
                          in Animal { }
```

TIP → Class inheritance not always a good choice → why? because derived classes become fragile. When you make changes in the base-class, if you do not accommodate the sub-classes they could break. While extending classes from another package - and/or bad documentation aren't that great either.

★ If a base class "chooses" to modify the implementation details of its methods, the derived classes might not be able to offer the functionality they were supposed to, or they might respond differently →

look at this → just look @ it → ϕ6.22 2ϕ24

```
public abstract class Animal {
    String currentPosition;
    public void move (String newPosition) {
        currentPosition = newPosition;
    }
}
class Gerbil extends Animal {
    void changePosition (String newPosition) {
    super.move (newPosition);
    System.out.println ("New Position: " + newPosition);
    }
}
class RunGerbil {
    public static void main (String args[]) {
        new Gerbil(). changePosition ("Cage");
```

prints "New Position: Cage", once

Now → let's imagine that Animal adds another line of code ↓ to move()

observe this → 06.22.2024

```
public abstract class Animal {
    String currentPosition;
    public void move(String newPosition) {
        currentPosition = newPosition;
        System.out.println("New pos-" + newPosition);
```

Implementation details augmented

```
class Gerbil extends Animal {
    void changePosition(String newPosition) {
        super.move(newPosition);
        System.out.println("New pos-" + newPosition);

public static void main(String args[]) {
    new Gerbil().changePosition("Cage");
```

→ will now print "Cage" x 2

TIP → There is NO clear winner when it comes to selecting the best option for design using inheritance, either via extending classes or implementing interfaces. Analyze the given conditions or situations carefully to answer questions on these topics. (great advice!) 06.22.2024

Imagine the thought process required to modify the core Java classes when a new version is released. As you observed → changes to the base class can break the code / change the behavior of its derived classes

3.1 > try-with-resources statement can declare objects, i.e. resources that implement the interface java.lang.AutoCloseable or any of its subinterfaces. A dev has defined a class StanLaptop → & use it with try-with-resources statement. ' Which option will enable the dev to achieve this goal ?

```
class StanLaptop {                                  Ø6.22
    public int open() {                             2Ø24
        /* some useless code */
        return Ø; // exit code
}
public void chargebattery() {
    /* more useless code */
}

public int close() {
    /* even more useless code */
    return 1; // bad exit code
}
}
```

A. Make StanLaptop implement interface java.lang.AutoClosable

B. Make StanLaptop implement interface java.io.Closeable → which extends interface java.lang.AutoClosable

C. Create a new Interface MyClosable that extends java.lang.AutoClosable, & make StanLaptop implement it.

d. StanLaptop can't implement interface Ø6.22, java.lang.AutoClosable or any of its 2Ø24 sub interfaces because of the definition of its method close()

→ identify & implement IS-A & HAS-A principle in code

Apply cohesion → low-coupling, IS-A & HAS-A principles

How to identify & implement Is-A → HAS-A relationships within classes & objs

ONE Simple IRule → follow the literal meaning of those terms:

- IS-A : This relationship is implemented when → A class Extends another class (derived class IS-A base class)

→ is-a → ϕ6.22 2ϕ24

An interface extends another interface (derived-interface IS-A base interface)

- A class implements an interface (class IS-A implemented interface)

★ HAS-A → This relationship is implemented by defining an instance variable. If a class → say e.g. → StanClass → defines an instance var of another class → say BobsClass -- StanClass HAS-A BobsClass
If StanClass defines an interface variable of an interface → e.g. → BobsInterface → BobsClass HAS-A relationship with BobsInterface.

TIP → IRepresent IS-A, HAS-A relationship by using UML. You may do this on the exam → The exam OCA OCPe 7 will test whether we can identify & implement these relationships in our code → e.g. IS-A →

3.3.1 Identifying & Implementing an IS-A IRelationship

ϕ6.22 2ϕ24

an IS-A relationship is implemented by extending classes or interfaces & implementing interfaces.

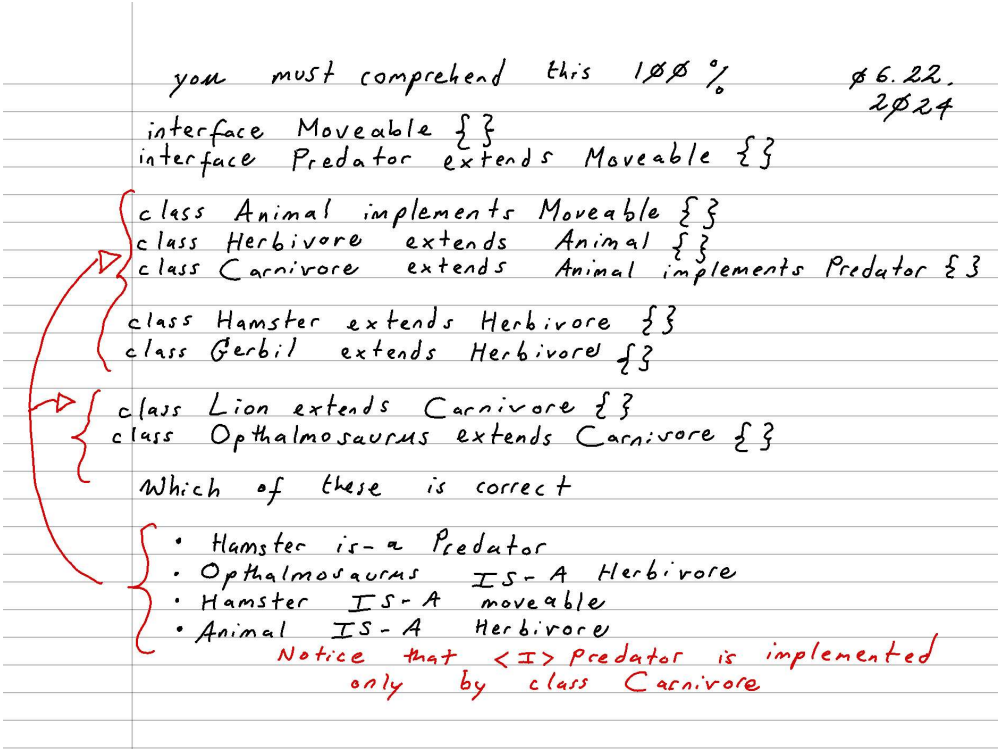
Traverse the inheritance tree up the hierarchy to identify this relationship.

A derived class IS-A, TYPE of its base-class-> makes sense, & its implemented interfaces.

A derived Interface IS-A TYPE of its base-interface.

★ The reverse is NOT TRUE
the relationship IS-A goes down ↓ the inheritance.

Identify An IS-A relationship



4 **Generics and Collections**

- [ ] Create Generic classes
- [ ] Use the diamond for type inference
- [ ] Analyze the interoperability of collections that use raw types and generic types
- [ ] Use wrapper classes, auto-boxing, and unboxing
- [ ] Create and use **List**, **Set** and **Deque** implementations
- [ ] Create and use **Map** implementations
- [ ] Use **java.util.Comparator** and **java.lang.Comparable**
- [ ] Sort and search arrays and lists

5 String processing

- [ ] Search, parse and build strings (including **Scanner, String-Tokenizer, StringBuilder, String**, and **String Formatter**)
- [ ] Search, parse, and replace strings by using regular expressions, using expression patterns for matching limited to:. .(dot), *(star)*, + (plus), ?, \d, \D, \s, \S, \w, \W, \b, \B, [], ()

- [ ] Format strings using the formatting parameters: %b, %c, %d, %f, and %s in format strings

6 Exceptions and Assertions

- [ ] Use **throw** and **throws** statements