# JAVA

Java is OOP programming language. Java is good scalable and platform independent

##### **Naming of Java. What is J2?**

* Java 1.0 was developer and released, but had a lot of bugs
* Java 1.1 and Java 1.2 were released
* Java 1.2 was so successful and recognized by people that they decided that previously was “just Java” and now it is “Java2 = J2”. So it means Java2 version 8 = Java 1.8 – it is the same

It was developed by the company Sun Microsystems in 1995. It is a specification and includes topics like Servlet, JSP, Web Services, EJB, JPA

Java code is compiled into bytecode by compiler and can be run on multiple platforms

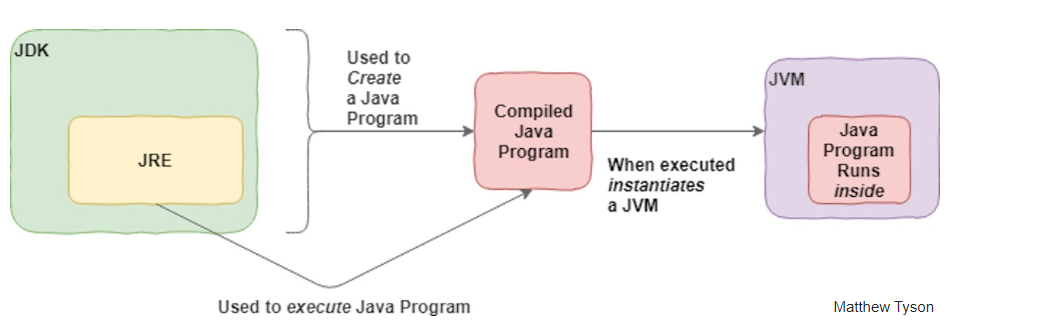
**JVM (Java Virtual Machine)** – is an abstract machine and it does not physically exist. It is a specification that provides runtime environment where bytecode will be executed. JVM are available for many platforms and OS. Therefore Java is platform independent

**JRE (Java Runtime Environment)** – it is used to provide runtime environment. It is the implementation of JVM

**JDK (Java Development Kit)** – is a software development environment which is used to develop Java applications. It contains JRE + development tools

**JDKs:**

* ME (Micro Edition) – for appliances, mob devices
* SE (Standard Edition)
* EE (Enterprise Edition)



# JAVA OOP

##### **History of programming:**

1. Algorithms
2. Procedural programming
3. OOP paradigm
4. Others: functional programming, aspect programming

**OOP:**

Purpose of OOP languages is to implement real word entities

OOP language over procedural

* It’s easy to develop and support applications over procedural language
* OOP provides data hiding. Whereas in procedural language global data can be accessed anywhere
* Make code reusable
* It is much easy to develop app by a few developers compared to procedural language

**CLASS** is blue print from which you can create [objects]

Class is logical entity

**OBJECT** is runtime entity that has state and behavior. Object is created from the class or instance of class. Everything that surrounds us is ab object (pen, table, chair, car)

Object is data and methods that process those data

Example1. Dog is an object.

* It has state - color, name, whiskers
* It has behavior – barking, eating, wagging tail

Example2. House is an object.

* It has state – address, color, area
* It has behavior – open door, close door

Object has address to the memory and therefore it takes up some space

Object is physical entity

Object has:

* **state** - is a value of all fields of object
* **behavior** – represents functionality (deposit, withdraw)
* **identity** – is ID. It is not visible to external users. It’s used internally by JVM to identify each object uniquely

[**new**] keyword is used to allocate memory at runtime. All objects get memory in [Heap memory] are

**Declaration -> Instantiation-> Initialization**

**1.Declaration** assign type to the variable

Person p;

**2.Instantiation** means to creating new object. It means allocation memory at runtime in [Heap] memory ] area

Person p = new Person();

Person p = new Person(“Alex”);

**3.Initialization** – passing parameters to constructor to define (=*storing data in Object*)

public class Person

{

private String name;

private String color;

public Person(String name)

{…}

public Person(String name, String color)

{…}

}

# INITIALIZE OBJECT

There are 3 ways to initialize object:

1. **by reference variable**



1. **by method**



1. **by constructor**



# INSTANTIATE OBJECT

It means allocation memory at runtime in [Heap] memory

There are 5 ways to do it:

* By [new ] keyword
* By [newInstance()] method
* By [clone] method
* By deserialization
* By factory method

**What does it mean create instance of class?**

*When we say Employee class, don't think of a specific employee. Rather, think generally about what an employee would be or have; an ID, name, job title, pay rate, and so on.*

*Then, if we hire an employee named Jane, she will also possess all of those attributes (pay rate, name, and so on). Jane is now considered to be an instance of the Employee class.*

# CONSTRUCTOR

**Constructor** is - a special method that does not have return method and must have the same name as class.

**Constructor can not be**:

* static
* final
* synchronized

Constructor *overloading* is allowed in Java. Overloading is just have constructor with different parameters

|  |  |
| --- | --- |
| CONSTRUCTOR | MEHTOD |
| *Constructor is used to initialize the state of Object* | Method is to make some behavior |
| It does not have return type | It must have return type |
| Constructor is invoked implicitly | method is invoked explicitly |
| Java provides default constructor | Method must be developed |
| Constructor name must match with class name | Any name |

public class Student

{

public Student()

{…}

public Student(String name)

{…}

public Student(int id, String name)

{…}

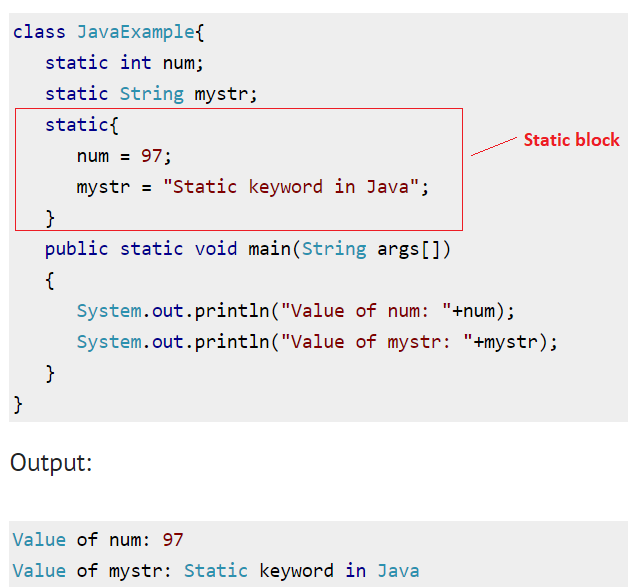
}

# STATIC

Static relates to class than instance.

Static can be

1. **variable** - it is used for cases when variables is used across all object. For example, SK, company name, university name. Static variable **gets memory only once**.
2. **method** -
3. **static block** – this block executed when class is loaded in the memory. It’s used to initialize static data



There are two main restrictions for the static method. They are:

* The static method can not use non static data member or call non-static method directly.
* this and super cannot be used in static context.

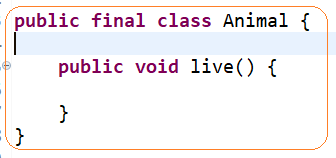
1. **nested class** -

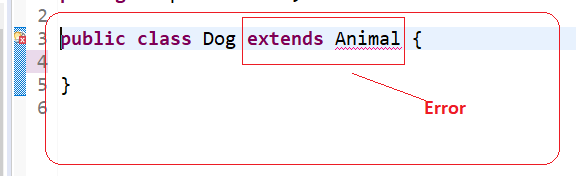
note: is it possible to execute java program without main method? It was possible in static bloch till JGK 1.6. After it’s not possible

##### STATIC NAME CONVENTION

# FINAL

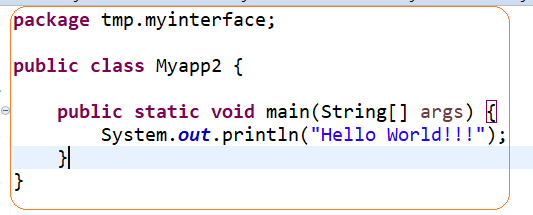
If class marked as final you cannot make inheritance. That is another class can not be extended by this final class. It is usually by done for security purposes





By name convention all constants must be in upper case: DAY\_OF\_THE\_DAY, SK

# RUN MAIN METHOD



**public** – access modifier which provides visibility. It means it is visible to all

**static** – is a keyword. There is no need to invoke object of class. So, it saves memory

**void** – return type

**main** - starting point of program

**String[] args** – is used for command line argument

# STATE OF OBJECT

**State of object** is a value of all fields of object

# ENCAPSULATION

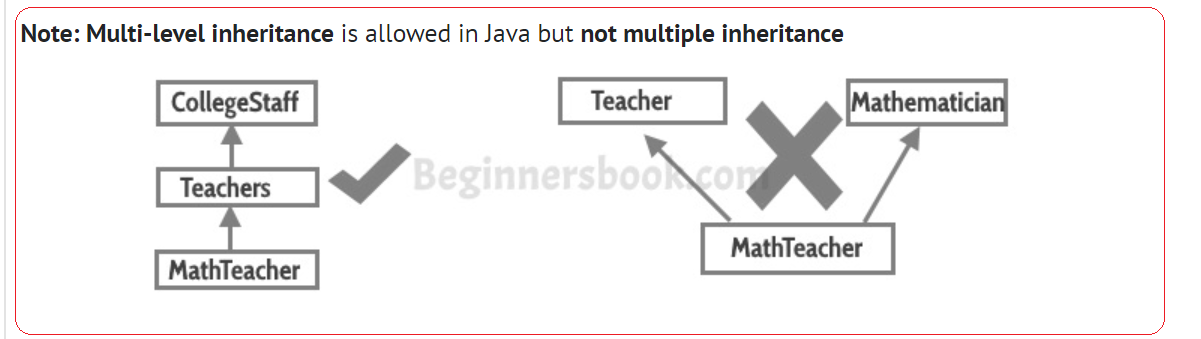
**ENCAPSULATION** - means when internal state of object can be changed only by this object.

The whole idea behind encapsulation is to hide the implementation details from users. If a data member is private it means it can only be accessed within the same class. No outside class can access private data member (variable) of other class. Getters and Setters can be used to access and change

# INHERITANCE

**INHERITANCE** – is OOP concept. The process by which one class acquires the **properties**(data members) and **behavior**(methods) of another class is called inheritance.

It’s used to achieve [runtime polymorphism]

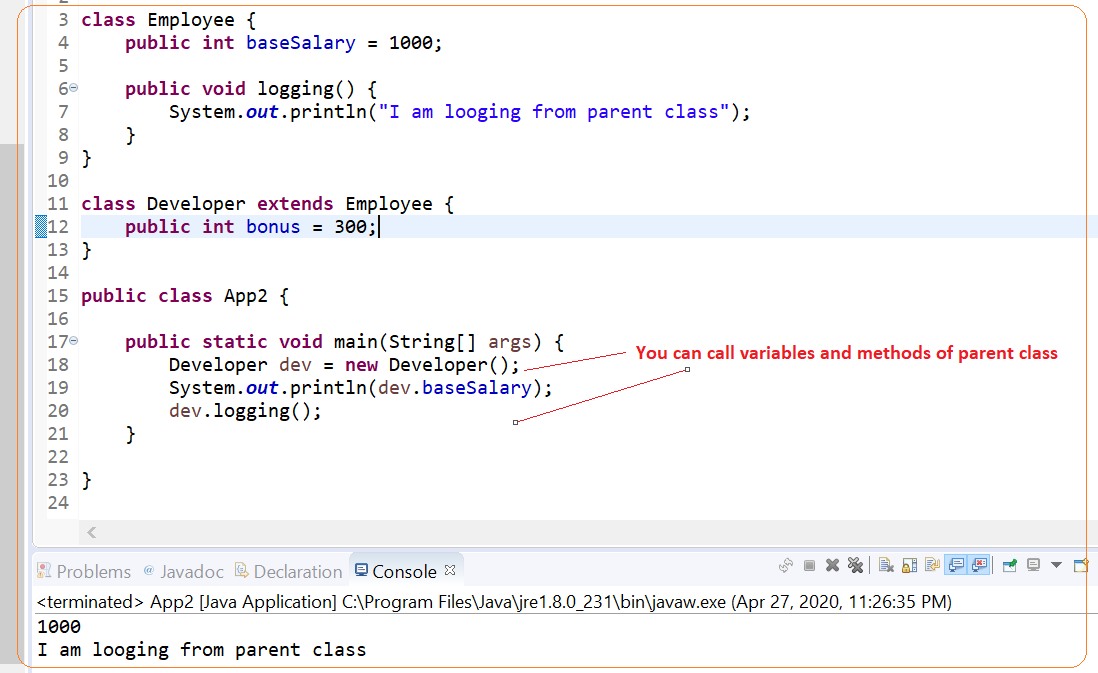


Why we need:

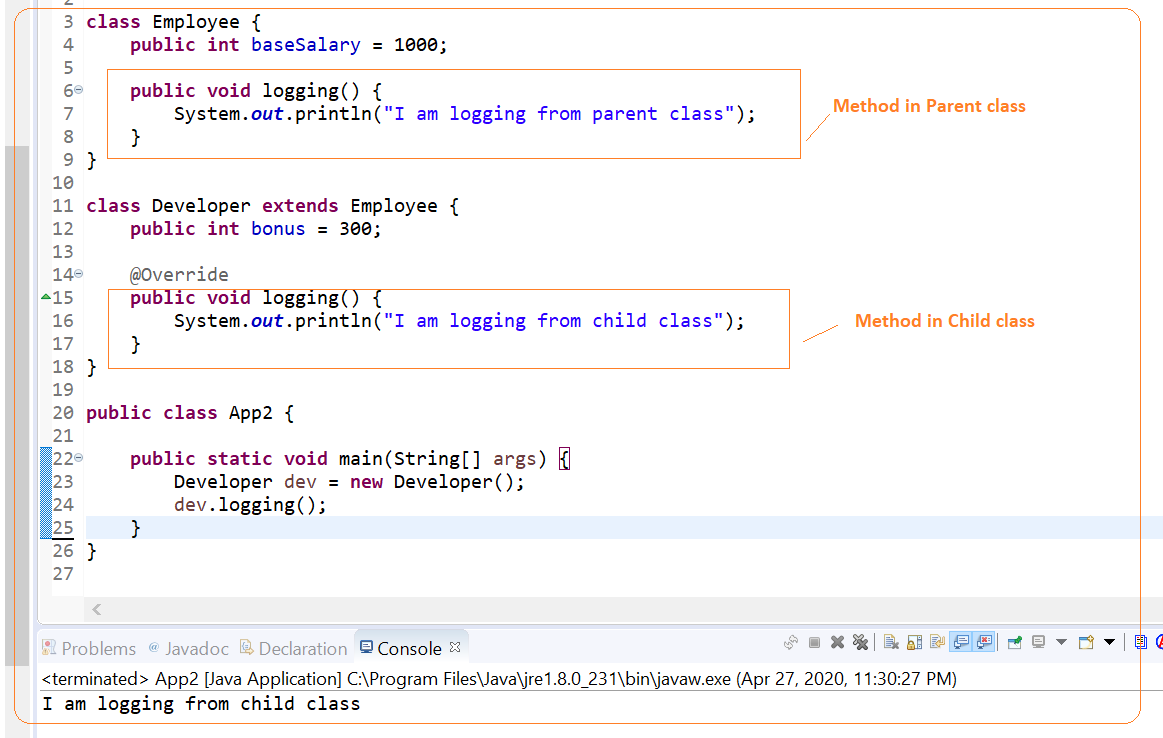
the idea of reusability of code and each sub class defines only those features that are unique to it, rest of the features can be inherited from the parent class.

Facts:

* Inheritance represents the IS-A relationship which is also known as a parent-child relationship.
* It’s used to achieve [runtime polymorphism]



**If [Parent] and [child] have methods with the same signature it’s called method overriding. You need redefine method of child**



# POLYMORFISM

**POLYMORFISM** is when task can be performed in different ways

**POLYMORFISM** is the ability of object takes different forms

Polymorphism:

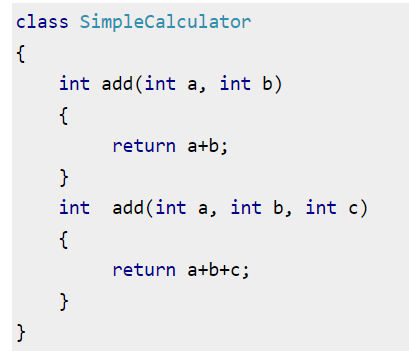
* **static binding** – method overloading is an example of static polymorphism = compile time
* dynamic binding – method overriding is an example of dynamic polymorphism = runtime time

##### STATIC BINDING

Method Overloading: This allows us to have more than one methods with same name in a class that differs in signature.

Polymorphism that is resolved during compiler time is known as **static polymorphism**. Method overloading is an example of compile time polymorphism.

Method Overloading: This allows us to have more than one method having the same name, if the parameters of methods are different in number, sequence and data types of parameters



##### RUNTIME BINDING

Dynamic polymorphism is a process in which a call to an overridden method is resolved at runtime, thats why it is called runtime polymorphism.

The main advantage of method overriding is that the class can give its own specific implementation to a inherited method without even modifying the parent class code.

Method Overriding is an example of runtime polymorphism. ***When a parent class reference points to the child class object then the call to the overridden method is determined at runtime, because during method call which method(parent class or child class) is to be executed is determined by the type of object***.

This is helpful when a class has several child classes, so if a child class needs to use the parent class method, it can use it and the other classes that want to have different implementation can use overriding feature to make changes without touching the parent class code.

**RULES OF OVERRIDING**

1. Argument list: The argument list of overriding method (method of child class) must match the Overridden method(the method of parent class). The data types of the arguments and their sequence should exactly match.
2. Access Modifier of the overriding method (method of subclass) cannot be more restrictive than the overridden method of parent class. For e.g. if the Access Modifier of parent class method is public then the overriding method (child class method ) cannot have private, protected and default Access modifier, because all of these three access modifiers are more restrictive than public.

For e.g. This is not allowed as child class disp method is more restrictive(protected) than base class(public)

1. private, static and final methods cannot be overridden as they are local to the class. private, static and final methods cannot be overridden as they are local to the class.

# ABSTRACTION

**ABSTRACTION** is a process of hiding details and showing only essential functionality. Abstraction may also be defined as the process of identifying only the required characteristics of an object ignoring the irrelevant details.

**Why we need an abstract class?Why we need an abstract class?**

Lets say we have a class Animal that has a method sound() and the subclasses(see inheritance) of it like Dog, Lion, Horse, Cat etc. Since the animal sound differs from one animal to another, we don’t know how implement this method in parent class, because each animal sounds in a different way

Example1: Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of car or applying brakes will stop the car but he does not know about how on pressing the accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of accelerator, brakes etc in the car. This is what abstraction is.

* *Abstraction can by achieved by interface and abstract class*
* **Abstract class can not be directly instantiated (with the new operator).**
* **Abstract method** is a method that is declared without an implementation.
* If class has at least one abstract method it has to be abstract

Pros of abstraction:

* Avoids code duplication and increases reusability.
* It reduces the complexity of viewing the things.
* Helps to increase security of an application or program as only important details are provided to the user.
* It I used for hierarchy
* It is used to achieve polymorphism (overriding methods)

**Abstract method** – method that has only signature (without body)

# ABSTRACTION vs ENCAPSULATION

Differences:

1. **Abstraction is implementation hiding, while encapsulation is data hiding**.
2. Abstraction is implemented using abstract class and interface while encapsulation is implemented using access modifiers(private, )

# INTERFACE

##### **Introduction**

Prior to java 8, interface in java can only have abstract methods. All the methods of interfaces are public & abstract by default. Java 8 allows the interfaces to have default and static methods. The reason we have default methods in interfaces is to allow the developers to add new methods to the interfaces without affecting the classes that implements these interfaces.

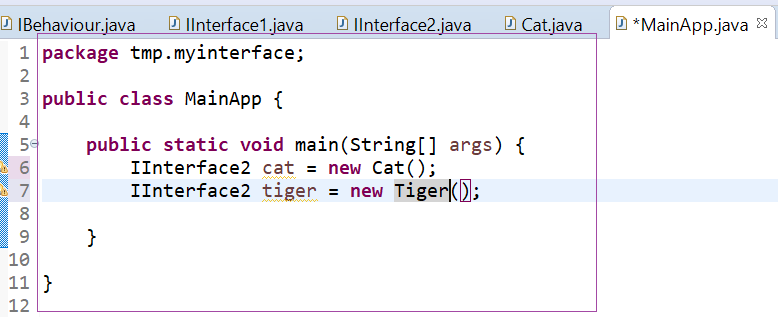
**Interface** – is a contract. It describes what behavior class will have. Interface provides a full abstraction while abstract class partial

**Why need interface**:

1. It describes what behavior class will have. For example, animal will have methods – sleep(), hunt(), eat(), but each class will have own implementation
2. Interfaces are used to provide **loose coupling = Achieve DI Dependency Injection**
3. Java does not support multiple inheritances but we can achieve the effect of multiple inheritances using interfaces. Java does not support multiple inheritances but we can achieve the effect of multiple inheritances using interfaces

**Tagged or Marker interface**

Tagged or Marker interface – is interface without methods. These methods are used to tag or marking a class. So that you can determine whether someclass is a child of those classes.



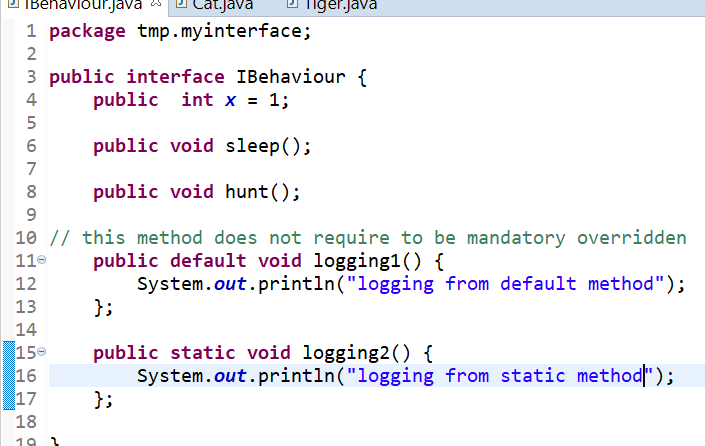
Example of marker interfaces are **Serializable, Cloneable.**

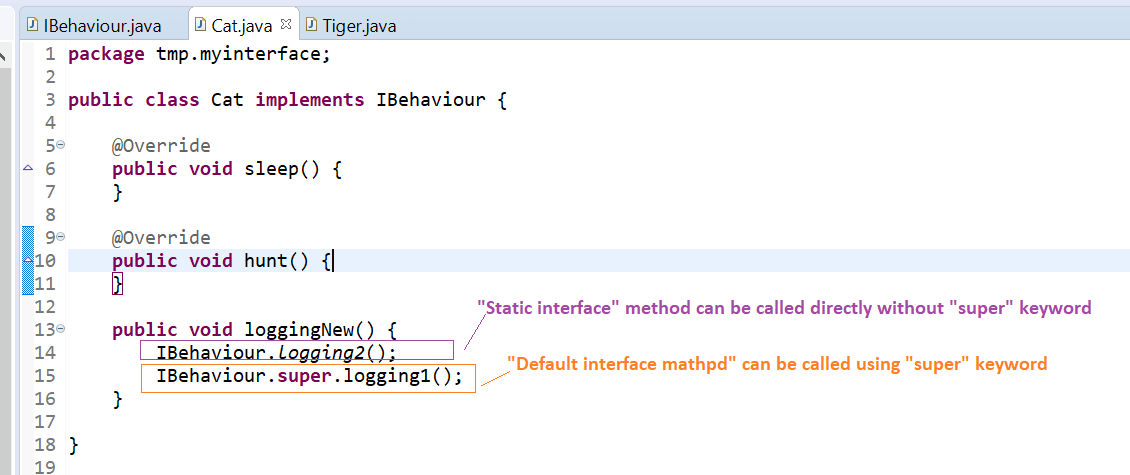
**Default interface methods**

For example, if several classes such as A, B, C and D implements an interface XYZ then if we add a new method to the XYZ interface, we have to change the code in all the classes(A, B, C and D) that implements this interface. In this example we have only four classes that implements the interface which we want to change but imagine if there are hundreds of classes implementing an interface then it would be almost impossible to change the code in all those classes. This is why in java 8, we have a new concept “default methods”. These methods can be added to any existing interface and we do not need to implement these methods in the implementation classes mandatorily, thus we can add these default methods to existing interfaces without breaking the code.

**Static interface methods**

Static methods in interfaces are similar to the default methods except that we cannot override these methods in the classes that implements these interfaces. Static methods in interfaces are similar to the default methods except that we cannot override these methods in the classes that implements these interfaces.





# ABSTRACTION vs INTERFACE

Difference:

* Abstract class can have constructor while Interface can not
* Interface provides a full abstraction while abstract class partial
* A Java interface can be implemented using keyword “implements” and abstract class can be extended using keyword “extends”. A Java interface can be implemented using keyword “implements” and abstract class can be extended using keyword “extends”.
* Java does not support multiple inheritances but we can achieve the effect of multiple inheritances using interfacesJava does not support multiple inheritances but we can achieve the effect of multiple inheritances using interfaces

# COUPLING

**COUPLING** refers to dependency to another class. You can use interface to achieve weakly coupling, because it does not have concrete implementation.

# COHESION

Weakly cohesive method will split the task into separate parts

# ASSOCIATION

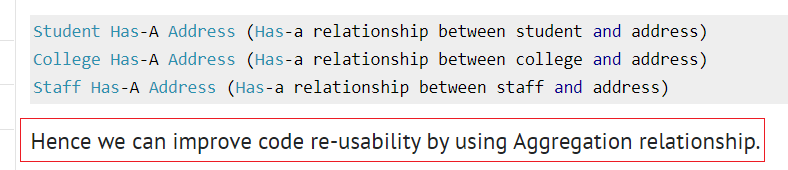
**ASSOCIATION** represents relationship between objects:

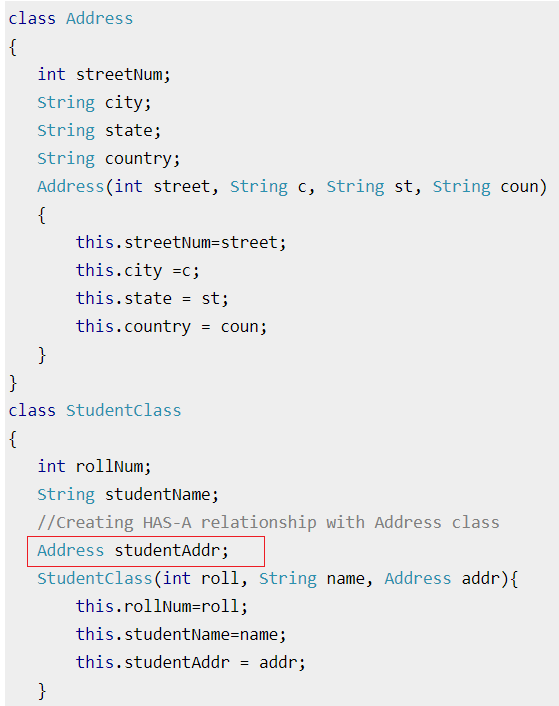
* 1:1
* 1:n
* n:1
* n:n

# AGGREGATION

**AGGREGATION** (**HAS-A**) is a way to achieve association.

**AGGREGATION** (**HAS-A**) represents the relationship where one object contains others as a part of its own state. It represents a **weak relationship** between objects





# COMPOSITION

**COMPOSITION**(**HAS-A**) is also way to represents the relationship where one object contains others as a part of its own state. It represents a **strong relationship** between objects. Dependent objects can exist without Parent

Composition is a restricted form of Aggregation in which two entities (or you can say classes) are highly dependent on each other. For e.g. Human and Heart. A human needs heart to live and a heart needs a Human body to survive. In other words when the classes (entities) are dependent on each other and their life span are same (if one dies then another one too) then its a composition. Heart class has no sense if Human class is not present.

# INHERITANCE

**INHERITANCE**(**IS-A**) relationship

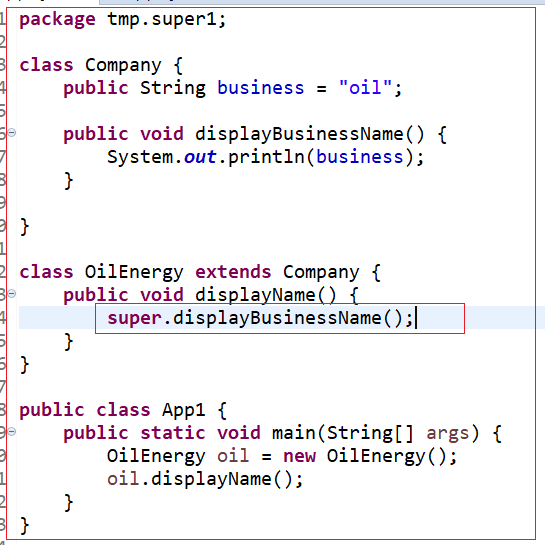
# [SUPER] keyword

1. [super] can be used to access data members of parent class
2. [super] can be used to access methods of parent class
3. [super] can be used to call constructor of parent class

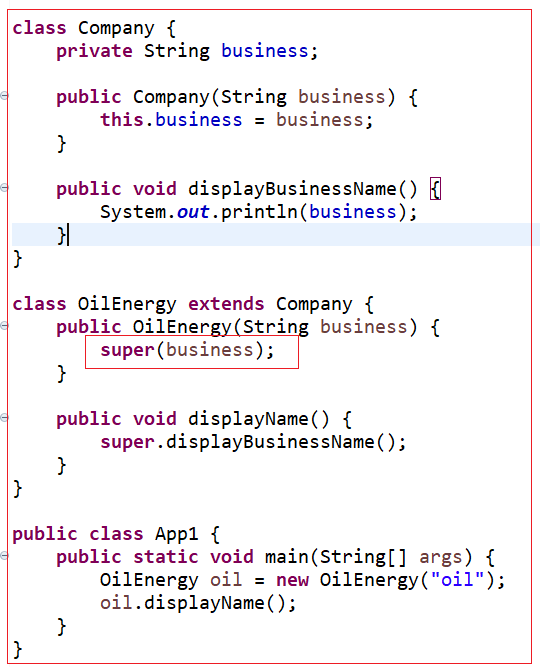
1.[super] can be used to access data members of parent class

# 

2. [super] can be used to access methods of parent class



3. [super] can be used to call constructor of parent class



# [THIS] keyword

There can be a lot of usage of java **[this]** keyword. In java, this is a reference variable that refers to the current object.

1. [this] can be used to refer to instance variable of current class
2. [this] can be used to invoke current class method (implicitly)
3. [this] can be used to invoke current constructor.
4. [this] can be passed as an argument in the method call.
5. [this] can be passed as argument in the constructor call.
6. [this] can be used to return the current class instance from the method.

Facts:

* You cannot use [this] for static methods

1.[this] can be used to refer to instance variable of current class



2.[this] can be used to invoke current class method



3.[this] can be used to invoke current constructor



4.[this] can be used to pass argument to the method



|  |  |
| --- | --- |
| **HOW ACHIEVE CONCEPT** | **DESC** |
| abstraction | Using abstract class and inheritance |
| runtime polymorphism | Inheritance (reference variables) |
| STRING |  |
|  |  |

# PACKAGE

**Package** is used to organize classes and interface.

There are 2 types:

* Built in packages
* User - defined

# METHOD SIGNATURE

**Method signature** - consists of method name and parameter lists

* Method signature does not include the return type of the method. A class cannot have two methods with same signature. If we try to declare two methods with same signature you will get a compile time error.

# DATA TYPE

There are 2 types of datatypes:

* Primitive
* Object

##### **objects**

My = new My(); new My() does not have object it has reference to My

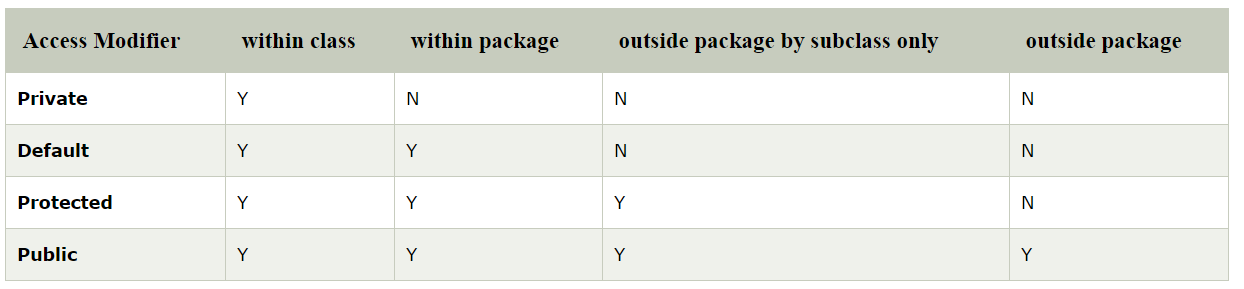
##### **primitives**

* For primitive types exist [obvertka]. For example, for [int] exists [Integer]
* You can not use primitives for collections

# BOXING and UNBOXING

# ACCESS MODIFIERS

* **private** -
* **default = package local** -
* **protected** – all childs of parent class will have access
* **public** -



PRIVATE

* If constructor private you cannot create instance of this class.
* Class and interface cannot be private
* If a class has private constructor then you cannot create the object of that class from outside of the class.

PROTECTED

* class cannot be protected

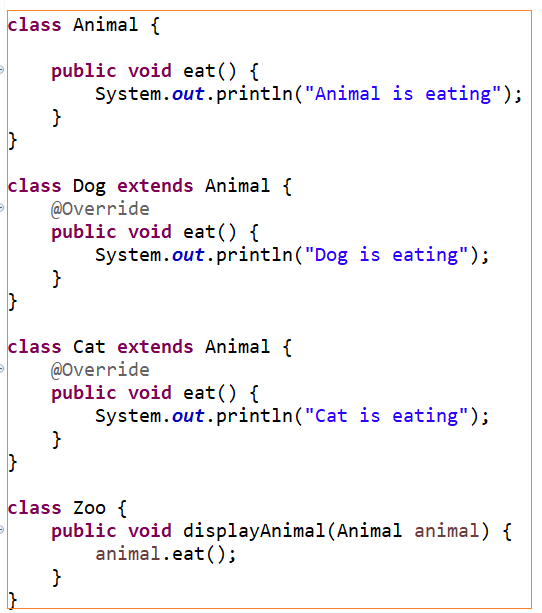
# UPCASTING

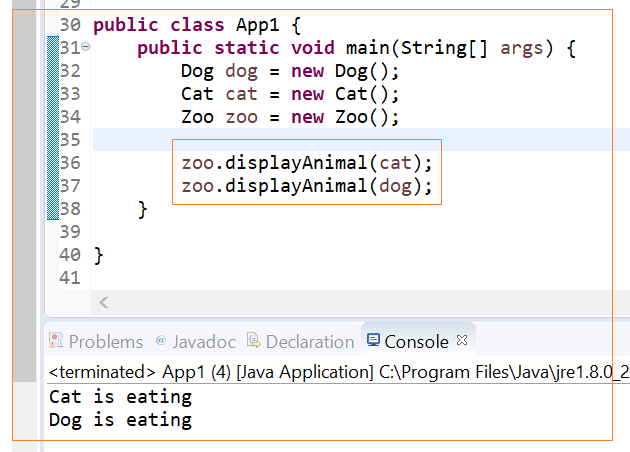
**Upcasting** is casting a subtype to a supertype, upward to the inheritance tree. Let’s see an example:

# 

**Why is Upcasting in Java?**

Generally, upcasting is not necessary. However, we need upcasting when we want to write general code that deals with only the supertype. Consider the following class:

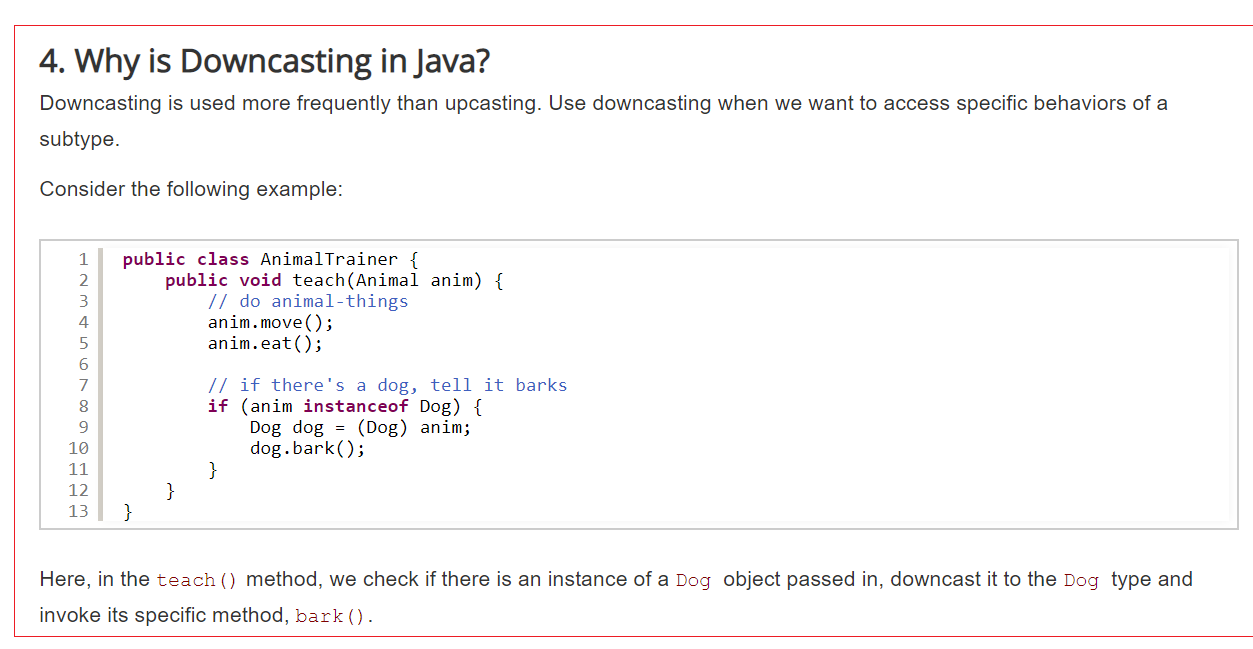




# DOWNCASTING

Downcasting - an object of parent class type is type cast into a child class type.

* Upcasting/Supercasting is always allowed, but downcasting/subcasting involves a type check and can throw a **ClassCastException**.
* So if you are not sure about the original object type, use the instanceof operator to check the type before casting. This eliminates the risk of a ClassCastException thrown.
* Down-casting is potentially unsafe, because you could attempt to use a method that the child class does not actually implement.



# LINKS

Links:

<https://www.javatpoint.com/this-keyword>

* equals:

<http://users.csc.calpoly.edu/~gfisher/classes/102/info/howToOverrideEquals.html>

* loose coupling

<https://www.interviewsansar.com/2018/03/24/loose-coupling-and-tight-coupling-in-java/>

* generics

<https://www.baeldung.com/java-generics-interview-questions>

<https://www.journaldev.com/1663/java-generics-example-method-class-interface>

<https://howtodoinjava.com/java/generics/complete-java-generics-tutorial/amp/>

* web

<https://www.javatpoint.com/get-vs-post>

(!!!!)

<https://www.restapitutorial.com/lessons/httpmethods.html>

<https://www.journaldev.com/1854/java-web-application-tutorial-for-beginners>

<https://www.journaldev.com/2513/tomcat-datasource-jndi-example-java>

<https://www.journaldev.com/1854/java-web-application-tutorial-for-beginners>

* jsp

<https://www.tutorialspoint.com/jsp/>

* Rest-api

<https://zapier.com/learn/apis/>

* Collection

(!!!) –super quide

<https://www.codejava.net/java-core/collections/java-list-collection-tutorial-and-examples>