**Thinking in JAVA**

* JAVA forces people to think in OO way whereas C++ is **backward compatible**.
* All Objects in JAVA are derived from **Object** class and hence, identifying the type of the class is easier.
* Objects are always stored in **heap** and hence lifetime of object need not be stored(unlike C++, where compile time objects are stored in stack and hence must be released). However, garbage collection is important
* JAVA cannot tell compiler to store in **registers**
* Arrays store either reference of the objects or null . Hence , error handling is easier
* Inheritance can be **is-a or is-like-a** relationship.
* Don't method overload based on **return types**

**Objects**

* Finalize() method need not be used as it is not guaranteed to be called. Mostly called when using other language calls
* Garbage collector 1: Collect the reference count of all the objects and delete if the reference count is 0. **Disadvantages** Need to keep track of reference count. Circular list cannot be cleaned
* Garbage collector 2: Iterate through the references in stack and check if they have LIVE objects and copy to another heap. Later sweep the current heap **Disadvantages** More space needed. ‘Stop and collect’ model
* Garbage collector 3: Iterate through the references in stack and check if they have LIVE objects and flag them . ‘Mark and correct’ model
* **Just-In-Time compiler** converts class to native machine code . (If entire code is JIT, space is more. Hence, often used are alone converted)

**Static objects**

* Static objects are initialized when first instance is created or when the static object was accessed for first time.
* If inner class is static, it **cannot access** outer class non-static values .
* Declaration if inner Is static
  + i = **new** NestedStatic.Inner();
  + else
  + i = outerobj.**new** Inner();

**Final**

* Static final means of one location where value cannot be changed
* Final **object reference and primitive value** can’t be changed and always must be initialized
* Final in argument means that the arg cannot be changed
* Within constructors, don't call its own methods .
* **Private methods can be overridden** in JAVA but it is actually treated as new method. Hence, baseclass b = new childclass() will call base class

**Interface**

* Interface with **same name can be avoided** to remove conflict (overloaded method with different return type)
* You **need not implement** the interface if the base class has one
* Interface can be **public or abstract**. Methods are always public and hence the implementation class should also declare it as public
* Interface can be **private** or **protected** if nested

**Inner classes**

* Inner class objects can be created within the outer class or their instance can be returned by the outer class
* Inner class can be **private or protected**
* Inner class can be used **within a method** if the object that a method creates should not be exposed to other members of its own class.
* Extending just inner class- careful ( default constructor wont work and hence you need to have parameterized constructor with param as outer class
* Inner class should also be explicitly inherited to use them
* **CLASS FILE NAME : OUTER$INNER.class**
* **Anonymous class**

**public** ClassWithoutInterface getRef(**final** **int** read){

**return** **new** ClassWithoutInterface(read){

**private** **int** i = read;

**public** **int** getI(){

**return** i;

}

**public** **int** print(){

**return** 1;

}

};

}

* Anonymous class can implement Interface or Extend a class with no name as above
* If the args should be used inside the Anonymous class, declare it as **final** in args
* Call the base class constructor (final not needed)
* Remember to declare the methods in base or interface as the “Base type” is returned to client
* You can initialize the values directly in anonymous class and it’s a **“constructor”**

**THROWABLE**

**ERROR EXCEPTION**

**CHECKED RUN-TIME**

**instanceOf and isInstance -> Whether you are of same type or derived types**

**== or equals -> Whether you are of my type. (Derived class is not of Base type)**

**SERIALIZATION**

* Condition: class implements **Serializable interface** . ObjectOutput/InputStream performs the work. User **ByteInputStream for deep copy**
* **Externalizable : writeExternal, readExternal** helps to control values in Obj being serialized
* Add **readObject and writeObject** of ObjectOutput/InputStream in Serializable implementing class to control
* **Transient** -> says don't serialize
* Serialization must be “**ATOMIC**” . Don't save object from two streams (Same object becomes two different one if it happens)

**Cloning**

* Always use **Cloneable** interface though it Is just tagging interface
* Cloning generally produces shallow copy- Use deep copy – Use serialization
* Are copy constructors better than using **protected clone method ?** No, imagine we have a slice method. It needs to clone a Tomato and cut it. Now, it accepts anything as Fruit base class . When the clone of fruit is created with copy constructor, it loses the character of actual fruit - tomato