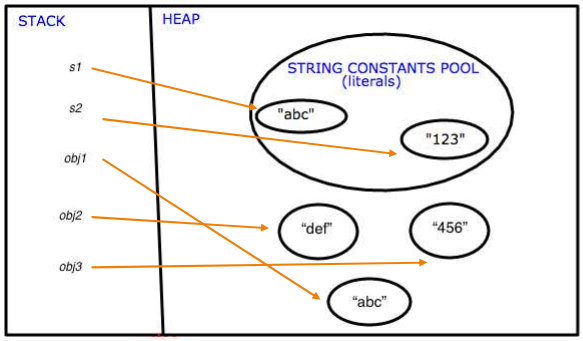
Core Java

Andy

1

Topic: String - Memory & Constant Pool




```
String s1 = "abc";
String s2 = "123";
String obj1 = new String("abc");
String obj2 = new String("def");
String obj3 = new String("456");
```

2

Topic : String

- **String is immutable**
  - String *str1* = “abc”; String *str2* = new String(“abc”); String *str3* = “abc”; String *str4* = str1 + str2;
    - Reference vs Value: 1) str1 == str2 ? 2) Str1.equals(str2) ? 3) str1.compareTo(str2)
    - Constant Pool: How many objects ? (Pool)
    - Built-in Functions: charAt(), length(), substring(,), contains(), equals(), toUpperCase()
- **StringBuilder & StringBuffer**
  - StringBuilder sb = new StringBuilder(“abc”); sb.append(“abc”);
    - sb.append(“abc”); sb.reverse(); sb.delete(startIndex, endIndex)
    - Mutable
    - toString()
    - Synchronized vs non-synchronized
- StringTokenizer

```
List<String> tokens = new ArrayList<String>();
StringTokenizer stringTokenizer = new StringTokenizer(input, delim: ", /.");
while(stringTokenizer.hasMoreTokens()){
    tokens.add(stringTokenizer.nextToken());
}
```

3

Topic: final

- Variable :
  - public static final String APP_NAME=“testApp”
  - Purpose: define constants
- Method:
  - public final int add(int a, int b){ return a + b; }
  - Purpose: prevent override
- Class:
  - final class MyClass(){}
  - Purpose: 1) prevent inheritance, like Integer, String etc; 2) Make class immutable
- Difference: final, finally, finalize

4

Topic: static –variable/block

- Only One Instance!

```
public class UserRepository {

    private static Set<User> users = new HashSet<>();

    static {
        users.add(new User( id: 1,  email: "a@gmail.com",  password: "a@gmail.com",  role: "USER"));
        users.add(new User( id: 2,  email: "b@gmail.com",  password: "b@gmail.com",  role: "ADMIN"));
    }

    public Set<User> getUsers() {
        return users;
    }

}
```

Question

1. Java Variable Scope ?
2. scope of static block ?

5

Topic: static - method

- 2) Static Method

```
public class StaticMethodTest {

    public static int add(int a, int b){
        return a + b;
    }

    public static void main(String[] args) {
        int result = StaticMethodTest.add( a: 3, b: 5);
        System.out.println(result == 8);
    }
}
```

Question:

1) Can static method access non-static variables ?
2) How to call static method in/outside of enclosing class?
3) Common Static Methods?
4) When to use static Methods?

6

Topic: static - class

- 3) Static Class

```
public class CarParts {

    //static wheel class
    public static class StaticWheel {
        public StaticWheel(){ System.out.println("Static Wheel created");}
        public void drive(){System.out.println("drive static wheel");}
    }

    //Non static wheel class
    public class NonStaticWheel {
        public NonStaticWheel(){System.out.println("Non Static Wheel created");}
    }

    //default class
    public CarParts(){System.out.println("Car Parts Object Created!");}

    //static method
    public static void combine(){System.out.println("combine in car parts!");}
}
```

```
public class App {

    public static void main(String[] args){
        CarParts.StaticWheel staticWheel = new CarParts.StaticWheel();
        staticWheel.drive();
        CarParts.combine();
        CarParts.NonStaticWheel nonStaticWheel = new CarParts().new NonStaticWheel();
        nonStaticWheel.toString();
    }
}
```

7

Serializable

```
public class Employee implements Serializable {

//  private static final long serialVersionUID = -6470090944414208496L;
    private static final long serialVersionUID = 1L;

    private String name;
    transient private int salary;

    //getter and setter methods
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getSalary() {
        return salary;
    }

    public void setSalary(int salary) {
        this.salary = salary;
    }
}
```

- 1) serialVersionUID
- 2) transient: to avoid serialization
- Static field or class is not serialized

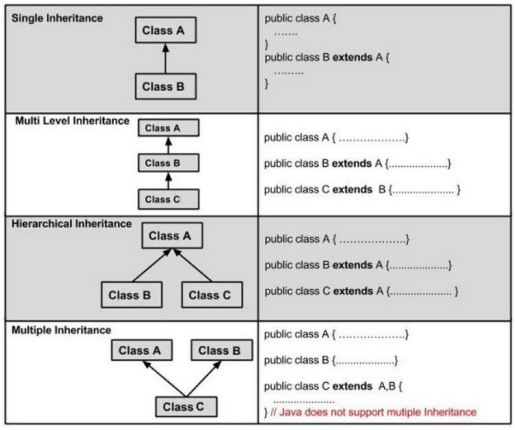
8

Topic: OOP(PIE)

- Inheritance

  Extends: Class

  Implements: Interface



Question:

1. Abstract Class
2. Abstract class vs Interface
3. Marker Interface (Serializable, Cloneable)

Tips:

> Multiple Inheritance (extends) is not allowed

> Multiple Implementation (implements) is allowed

> Use "super()" to access parent class

Cloneable: shallow clone

shallow clone vs deep clone

9

Topic: OOP(PIE)

- Polymorphism
  - Static Polymorphism / Dynamic Polymorphism
  - Overload (same class) / Override (child class)

```
public class JusticeLeague{
    public JusticeLeague(){
        System.out.println("Justice League founded!");
    }

    public void fight(){
        System.out.println("All fight together");
    }

    //overload
    public void fight(String member){
        System.out.println("Only" + member + "flight");
    }
}

public class Batman extends JusticeLeague{
    public Batman(){
        super();
        System.out.println("*******inside batman*********");
    }

    @Override
    public void fight(String member){
        System.out.println("Batman and " + member + "will fight together");
    }
}
```

```
public class TestJusticeLeague{

    public void test(){
        JusticeLeague justiceLeague = new JusticeLeague();
        justiceLeague.fight();
        justiceLeague.fight( member: "Flash");
        Batman batman = new Batman();
        batman.fight( member: "superman");
    }

}
```

Override can NOT be applied to static method

10

Topic: OOP(PIE)

- Encapsulation

```java
/* File name : EncapTest.java */
public class EncapTest {
   private String name;
   private String idNum;
   private int age;

   public int getAge() {
      return age;
   }

   public String getName() {
      return name;
   }

   public String getIdNum() {
      return idNum;
   }

   public void setAge( int newAge) {
      age = newAge;
   }

   public void setName(String newName) {
      name = newName;
   }

   public void setIdNum( String newId) {
      idNum = newId;
   }
}
```

```java
/* File name : RunEncap.java */
public class RunEncap {

   public static void main(String args[]) {
      EncapTest encap = new EncapTest();
      encap.setName("James");
      encap.setAge(20);
      encap.setIdNum("12343ms");

      System.out.print("Name : " + encap.getName() + " Age : " + encap.getAge());
   }
}
```

Tips: use getter/setter instead of direct access

Industry standard code structure

11

Topic: Design Pattern (Singleton)

```java
public class Singleton implements Serializable, Cloneable{
    private static Singleton instance;
1   private Singleton(){}
    public static synchronized Singleton getInstance(){
2       if(instance == null){
            instance = new Singleton();
        }
        return instance;
    }

    //prevent Clone
    @Override
3   protected Object clone() throws CloneNotSupportedException{
        throw new CloneNotSupportedException();
    }

    //prevent Serializable/DeSerializable
4   protected Object readResolve(){
        return instance;
    }
}

//prevent Java Reflection    *
public enum SingletonEnum{
    INSTANCE;
    private SingletonEnum(){}
}
```

- Steps to make singleton
  - 1. private Constructor
  - 2. static synchronized getInstance method
  - 3. override clone() method
  - 4. override readResolve() method
- Disadvantage of Singleton
  - 1. it carries state of object
  - 2. it prevents dependency injection and cannot be unit tested

*To Clone(): 1) class should implements Cloneable; 2) It is only a shallow clone; 3) avoid to use

12

Topic: Design Pattern (Factory)

```
public abstract class Foxconn {
    public abstract makePhones(){};
}

public class Iphone extends Foxconn {
    //make iphone
}

public class Samsung extends Foxconn {
    //make samsung phone
}

//factory class
public class PhoneFactory{
    public static Foxconn makePhone(String brand){
        if("iphone".equals(brand)){
            return new Iphone();
        }
        if("samsung".equals(brand)){
            return new Samsung();
        }
    }
}
```

Example 1

```
public interface DBConnection{
    public Connection getConnection();
}

public class OracleConnection implements DBConnection{
    public Connection getConnection(){
        DataSource oracleDS = new OracleDataSource();
        Connection conn = driver.getConnection(oracleDS);
        return conn;
    }
}

public class SqlServerConnection implements DBConnection{
    public Connection getConnection(){
        DataSource sqlserverDS = new SQLServerDataSource();
        Connection conn = driver.getConnection(sqlserverDS);
        return conn;
    }
}

public class DBConnectionFactory{
    public DBConnection getDBConnection(String db){
        if("oracle".equals(db)){
            return new OracleConnection();
        }
        if("sqlserver".equals(db)){
            return new SqlServerConnection();
        }
    }
}
```

Example 2

13

Topic: Design Pattern (Observer)

- It is similar to "subscribe" in Angular/Redux and Reactive Programming

```
//Observable
public class News {
   private List<MediaCompany> observers = new ArrayList<MediaCompany>();
   private String subject;

   public String getSubject() {
      return subject;
   }

   public void setSubject(String subject) {
      this.subject = subject;
      notifyAllObservers();
   }

   public void addObserver(MediaCompany observer){
      observers.add(observer);
   }

   public void notifyAllObservers(){
      for (MediaCompany observer : observers) {
         observer.update();
      }
   }
}
```

```
//Observer interface
public abstract class MediaCompany{
    protected News news;
    public abstract void update();
}

public class CNN extends MediaCompany{
    public CNN(News news){
      this.news = news;
      this.news.addObserver(this);
    }

   @Override
   public void update() { System.out.println("CNN");}
}

public class FOX extends MediaCompany{
    public FOX(News news){
      this.news = news;
      this.news.addObserver(this);
    }

   @Override
   public void update() { System.out.println("FOX");}
}

public class NBC extends MediaCompany{
    public NBC(News news){
      this.news = news;
      this.news.addObserver(this);
    }

   @Override
   public void update() { System.out.println("NBC");}
}
```

```
public class ObserverPatternDemo {
   public static void main(String[] args) {
      News news = new News();

      new CNN(news);
      new FOX(news);
      new NBC(news);

      news.setSubject("News: Happy New Year!");
      news.setSubject("News: Happy July 4th!");
   }
}
```

14

Topic: Design Pattern (Others)

- Popular Design Pattern to choose:
  - Decorator, Façade, Strategy, Builder, Composite etc
- DO NOT SAY “MVC”

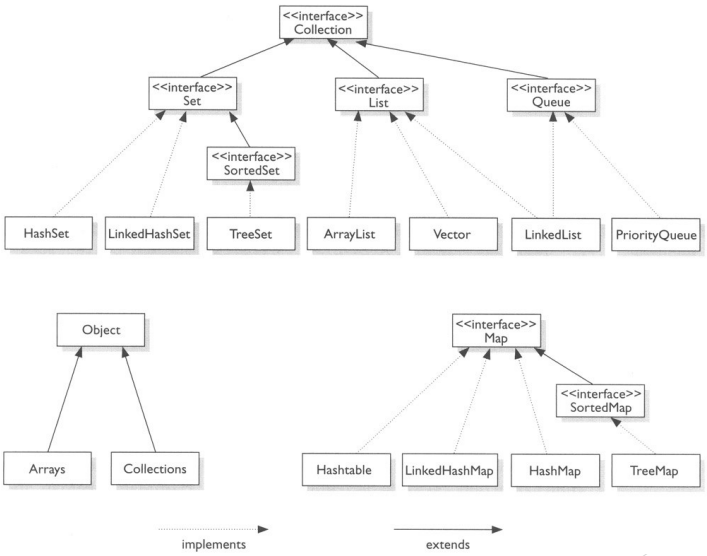
15

Topic: Real Scenario

- 1. Design a Vending Machine
- 2. Design an Elevator System
- 3. Design a Parking Lot

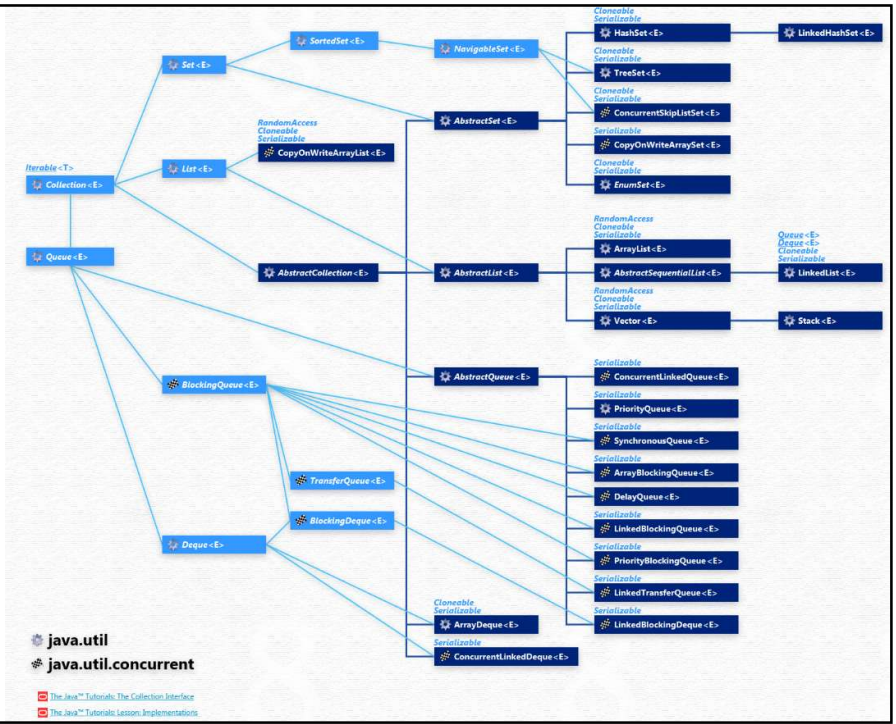
16

Topic: Collection



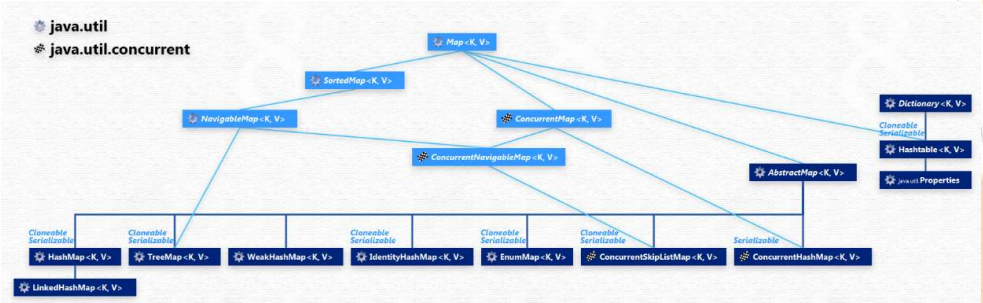
17

Collection + Concurrent



18

Map + Concurrent



19

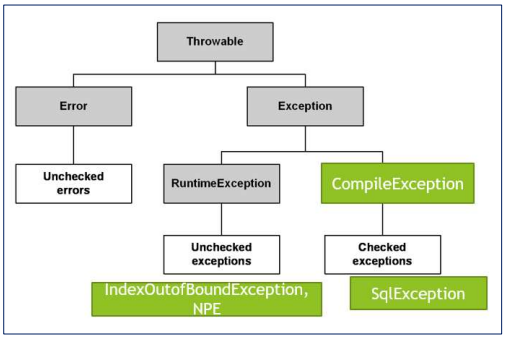
Topic: Iterator vs Enumeration

```
public User getUserById(int id) {
    Enumeration<User> enumUser = Collections.enumeration(userDB);
    while(enumUser.hasMoreElements()){
        User temp = enumUser.nextElement();
        if(temp.getId() == id){
            return temp;
        }
    }
    return null;
}


public void deleteUserById(int id) {
    Iterator<User> itUser = userDB.iterator();
    while(itUser.hasNext()){
        User tempUser = itUser.next();
        if(tempUser.getId() == id){
            itUser.remove();
        }
    }
}
```

20

Topic: Exception Handling



Exception Structure

```
class MyOwnException extends Exception {

    MyOwnException(String msg)
    {
        super(msg);
    }
}

class Javaapp {

    public static void main(String[] args) {

        try{
            int age = 130;
            if(age>=100)
            {
                throw new MyOwnException("Age = "+age+" It's not possible");
            }
        }catch(MyOwnException ex)
        {
            System.out.println(ex);
        }
        System.out.println("Execution Complete");
    }
}
```

Define your own Exception

(try-catch, throw, throws)

21

Topic: Multithreading(thread – java18)

- 1. Create a thread

```
public class MyThread extends Thread {

  public void run(){
     System.out.println("MyThread running");
  }
}
```

```
public class MyRunnable implements Runnable {

  public void run(){
     System.out.println("MyRunnable running");
  }
}
```

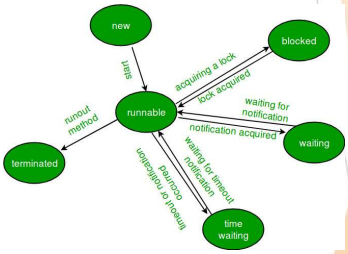
- 2. Race Condition

  Multiple Threads try to change the same object status.

- 3. Start() vs Run()

  A: t.start() create and run a new thread. t.run() run the thread in current thread

- 4. Thread LifeCycle



- 5.ThreadLocal vs Volatile

  A: ThreadLocal: create thread level copy of object

  Volatile: All threads share and update same object

22

Topic: Multithreading

- 6. join(*s*) tells other threads to wait until this thread is completed or just wait for the given *s* seconds

Question: Given three threads, how to let them run in order?

```
public class ThreadSequence {

 public static void main(String[] args) {
  SeqRun sr = new SeqRun();
  // Three threads
  Thread t1 = new Thread(sr);
  Thread t2 = new Thread(sr);
  Thread t3 = new Thread(sr);

  try {
   // First thread
   t1.start();
   t1.join();
   // Second thread
   t2.start();
   t2.join();
   // Third thread
   t3.start();
   t3.join();
  } catch (InterruptedException e) {
   // TODO Auto-generated catch block
   e.printStackTrace();
  }
 }
}
```

23

Topic: Synchronized

- Synchronized on Method;
- Synchronized on Block

- Static synchronized vs Synchronized

  (Analyze 4 scenarios)

```
public class SynchronizedScope{

    public Employee employee;
    public EmployeeService service;

    public synchronized void modifyEmployee(){
        service.modify();
    }

    public void modifyEmployee(boolean isValid){
        if(!isValid){
            return;
        }

        synchronized(employee){
            service.modify();
        }
    }
}
```

```
Public SynchronizedTest{
    pubic static synchronized method1(){}
    public synchronized method2(){}
    public synchronized method3(){}
}

SynchronizedTest s1 = new SynchronizedTest();
SynchronizedTest s2 = new SynchronizedTest();

//scenario 1:
thread1.run(s1.method1);
thread2.run(s1.method2);

//scenario 2:
thread1.run(s1.method2);
thread2.run(s1.method3);

//scenario 3:
thread1.run(s1.method2);
thread2.run(s2.method2);

//scenario 4:
thread1.run(s1.method1);
thread2.run(s2.method1);
```

24

Topic: Concurrency
-- 1) ExecutorService

- **ExecutorService** maintains thread pool
- Executors.***newCachedThreadPool()***

  newFixedThreadPool()
- Execute(), submit(), invokeAny(), invokeAll()
- Runnable vs Callable

```
ExecutorService executorService = Executors.newSingleThreadExecutor();

executorService.execute(new Runnable() {
    public void run() {
        System.out.println("Asynchronous task");
    }
});

executorService.shutdown();
```

```
Future future = executorService.submit(new Callable(){
    public Object call() throws Exception {
        System.out.println("Asynchronous Callable");
        return "Callable Result";
    }
});

System.out.println("future.get() = " + future.get());
```

- **Future**.get(*time), Future.isDone(), Future.cancel()
- Executor.shutdown()

```
public static void runSameTime(){
    //TODO: finish and print "B.getMethod A.getMethod"
    ExecutorService service = Executors.newFixedThreadPool( nThreads: 2);

    try{
        Future<String> aFuture = service.submit(()-> new tmobile_2019_06_03_i2.A().getMethod());
        Future<String> bFuture = service.submit(()-> new tmobile_2019_06_03_i2.B().getMethod());

        boolean allDone = false;
        while(!allDone){
            if(aFuture.isDone() && bFuture.isDone()){
                System.out.println(bFuture.get() + " " + aFuture.get());
                allDone = true;
            }
        }
    }catch (InterruptedException e) {
        e.printStackTrace();
    }catch (Exception e){
        e.printStackTrace();
    }finally {
        service.shutdown();
    }
}
```

25

Topic: Concurrency
-- 2) Semaphore and Mutex

- Semaphore – limits the number of threads to access one resource concurrently

  Semaphore semaphore = new Semaphore(4)
- Mutex – limits only 1 thread to access one resource concurrently

  Mutex is not native in java, Semaphore(1) is Mutex

```
package com.bht.aop.main;
import java.util.concurrent.Semaphore;

public class SemaphoreMain {
   static Semaphore semaphore = new Semaphore(3);
  //static Semaphore semaphore = new Semaphore(1);
   static class MyATMThread extends Thread {
      String name;
      MyATMThread(String name){ this.name = name; }
      public void run(){
         try{
            semaphore.acquire();
            System.out.println(name + " : got the permit!");
            try{
               System.out.println(name + " is doing work now! ");
               Thread.sleep(5000);
            }finally {
               System.out.println(name + " : releasing lock...");
               semaphore.release();
            }
         }catch (InterruptedException e){e.printStackTrace();}
      }
   }

   public static void main(String[] args) {
      System.out.println("Total available Semaphore permits :" +
semaphore.availablePermits());
      MyATMThread t1 = new MyATMThread("A");
      t1.start();
      MyATMThread t2 = new MyATMThread("B");
      t2.start();
      MyATMThread t3 = new MyATMThread("C");
      t3.start();
      MyATMThread t4 = new MyATMThread("D");
      t4.start();
      MyATMThread t5 = new MyATMThread("E");
      t5.start();
   }
}
```

26

Topic: Java 8(default & static methods)

- Problem with java interfaces: children must implements all methods in interface, if adding new definitions to interface, all children have to update.
- Solution: ***default*** methods in interface

```
interface MyInterface{

    default void newMethod(){
        System.out.println("Newly added default method");
    }
    void existingMethod(String str);
}
interface MyInterface2{

    default void newMethod(){
        System.out.println("Newly added default method");
    }
    void disp(String str);
}
```

```
public class Example implements MyInterface, MyInterface2{
        // implementing abstract methods
    public void existingMethod(String str){
        System.out.println("String is: "+str);
    }
    public void disp(String str){
        System.out.println("String is: "+str);
    }
    //Implementation of duplicate default method
    public void newMethod(){
        System.out.println("Implementation of default method");
    }
    public static void main(String[] args) {
        Example obj = new Example();

        //calling the default method of interface
        obj.newMethod();


    }
}
```

- Problem: multiple implementations -> override
- ***Static*** methods: the same as defaults method, but does not allow override

27

Topic: Java 8(lambda)

Why introducing lambda?

- To replace anonymous inner class
- Work with functional interface (next slide)

Tooltips:

- Avoid large block of code and “return”
- Avoid () around single input
- Avoid input type

```
public class ComparatorTest {

  public static void main(String[] args) {

    List<Person> personList = Person.createShortList();

    // Sort with Inner Class
    Collections.sort(personList, new Comparator<Person>(){
      public int compare(Person p1, Person p2){
        return p1.getSurName().compareTo(p2.getSurName());
      }
    });

    System.out.println("=== Sorted Asc SurName ===");
    for(Person p:personList){
      p.printName();
    }

    // Use Lambda instead

    // Print Asc
    System.out.println("=== Sorted Asc SurName ===");
    Collections.sort(personList, (Person p1, Person p2) -> p1.getSurName().compareTo(p2.getSurName()));

    for(Person p:personList){
      p.printName();
    }

    // Print Desc
    System.out.println("=== Sorted Desc SurName ===");
    Collections.sort(personList, (p1,  p2) -> p2.getSurName().compareTo(p1.getSurName()));

    for(Person p:personList){
      p.printName();
    }

  }
}
```

28

Topic: Java 8(Functional Interface)

- has one Single Abstract Method
- @FunctionalInterface – for sanity check
- Can have default methods
- Lambda is the implementation of the abstract method

```
@FunctionalInterface
public interface ITrade {
  public boolean check(Trade t);
}
```

```
// Lambda for big trade
ITrade bigTradeLambda = (Trade t) -> t.getQuantity() > 10000000;

// Lambda that checks if the trade is a new large google trade
ITrade issuerBigNewTradeLambda = (t) -> {
    return t.getIssuer().equals("GOOG") &&
           t.getQuantity() > 10000000 &&
           t.getStatus().equals("NEW");
  };
```

29

Topic: Java 8(stream)

- filter

```
public class NowJava8 {

    public static void main(String[] args) {

        List<Person> persons = Arrays.asList(
                new Person("mkyong", 30),
                new Person("jack", 20),
                new Person("lawrence", 40)
        );

        Person result1 = persons.stream()
                .filter((p) -> "jack".equals(p.getName()) && 20 == p.getAge())
                .findAny()
                .orElse(null);

        System.out.println("result 1 :" + result1);

        //or like this
        Person result2 = persons.stream()
                .filter(p -> {
                    if ("jack".equals(p.getName()) && 20 == p.getAge()) {
                        return true;
                    }
                    return false;
                }).findAny()
                .orElse(null);

        System.out.println("result 2 :" + result2);

    }

}
```

Predicate

map

```
public class NowJava8 {

    public static void main(String[] args) {

        List<Staff> staff = Arrays.asList(
                new Staff("mkyong", 30, new BigDecimal(10000)),
                new Staff("jack", 27, new BigDecimal(20000)),
                new Staff("lawrence", 33, new BigDecimal(30000))
        );

        // convert inside the map() method directly.
        List<StaffPublic> result = staff.stream().map(temp -> {
            StaffPublic obj = new StaffPublic();
            obj.setName(temp.getName());
            obj.setAge(temp.getAge());
            if ("mkyong".equals(temp.getName())) {
                obj.setExtra("this field is for mkyong only!");
            }
            return obj;
        }).collect(Collectors.toList());

        System.out.println(result);

    }

}
```

Function

Output

```
[
    StaffPublic{name='mkyong', age=30, extra='this field is for mkyong only!'},
    StaffPublic{name='jack', age=27, extra='null'},
    StaffPublic{name='lawrence', age=33, extra='null'}
]
```

30

Topic: Java 8(stream)

- Joining

```
//joining -- concatenate list into string
//replace "wal" with "Sams"

String s = "walabcwALexyWALxzsfwalmx";
Arrays.asList(s.split("(?i)Wal")).stream().collect(Collectors.joining("Sams"));
```

- GroupingBy

```
//GroupingBy - split list into multiple categories, return map
//GroupingBy takes two parameters: a)grouping key; b)how to group the list according to key

List<String> strList = Arrays.asList("Jan", "Feb", "Jan", "July");

Map<String, List<String>> map1 = strList.stream()
                                .collect(Collectors.groupingBy(o -> o));

Map<String, Integer> map2 = strList.stream()
                                .collect(Collectors.groupingBy(o->o, Collectors.summingInt(o->1)));
```

31

Topic: Java 8(optional)

- To avoid Null checks and run time NullPointerExceptions

```
public class Mobile{
    private long id;
    private String brand;
    private String name;
    private Optional<DisplayFeatures> displayFeatures;
}

public class DisplayFeatures{
    private String size;
    private Optional<ScreenResolution> resolution;
}

public class ScreenResolution{
    private int width;
    private int height;
}
```

```
//Non - Java 8
public class MobileService {
    public int getMobileScreenWidth(Mobile mobile){
        if(mobile != null){
            DisplayFeatures df = mobile.getDisplayFeatures();
            if(df != null){
                ScreenResolution sr = df.getResolution();
                if(sr != null){
                    return sr.getWidth();
                }
            }
        }
        return 0;
    }
}

//Java 8
public class MobileService {
    public int getMobileScreenWidth(Mobile mobile){
        return mobile.flatMap(Mobile::getDisplayFeatures)
                     .flatMap(DisplayFeatures::getResolution)
                     .map(ScreenResolution::getWidth)
                     .orElse(0);
    }
}
```

32

Topic: Java 8(forEach)

1.1 Normal way to loop a Map.

```
Map<String, Integer> items = new HashMap<>();
items.put("A", 10);
items.put("B", 20);
items.put("C", 30);
items.put("D", 40);
items.put("E", 50);
items.put("F", 60);

for (Map.Entry<String, Integer> entry : items.entrySet()) {
    System.out.println("Item : " + entry.getKey() + " Count : " + entry.getValue());
}
```

1.2 In Java 8, you can loop a `Map` with `forEach` + lambda expression.

```
Map<String, Integer> items = new HashMap<>();
items.put("A", 10);
items.put("B", 20);
items.put("C", 30);
items.put("D", 40);
items.put("E", 50);
items.put("F", 60);

items.forEach((k,v)->System.out.println("Item : " + k + " Count : " + v));

items.forEach((k,v)->{
    System.out.println("Item : " + k + " Count : " + v);
    if("E".equals(k)){
        System.out.println("Hello E");
    }
});
```

2.1 Normal for-loop to loop a List.

```
List<String> items = new ArrayList<>();
items.add("A");
items.add("B");
items.add("C");
items.add("D");
items.add("E");

for(String item : items){
    System.out.println(item);
}
```

2.2 In Java 8, you can loop a `List` with `forEach` + lambda expression or method reference.

```
List<String> items = new ArrayList<>();
items.add("A");
items.add("B");
items.add("C");
items.add("D");
items.add("E");

//lambda
//Output : A,B,C,D,E
items.forEach(item->System.out.println(item));

//Output : C
items.forEach(item->{
    if("C".equals(item)){
        System.out.println(item);
    }
});

//method reference
//Output : A,B,C,D,E
items.forEach(System.out::println);

//Stream and filter
//Output : B
items.stream()
    .filter(s->s.contains("B"))
    .forEach(System.out::println);
```

33