**(I) CPU**

***(1) What is CPU? What does CPU do?* CPU -** Central Processing Unit. CPU processes instructions written in **machine language.**

***(2) What is a machine language? How many bits in a byte?* Machine language** consists of binary digits [0 - off, 1 – on]. 1 byte = 8 bits

***(3) List the three steps of CPU instruction processing? What are the two CPU units?***

**(a) Input**: fetches instructions. **(b)** Follow the instructions [**Unit: a)** control and arithmetic unit**b)** logic unit]. **(c)** **Output:** produces some outcome.

**(II) Memory**

***(1) What memory is divided into? What does each byte contain?* Memory** is divided into **bytes**. Each byte is assigned a unique number, an **address**.

***(2) What is RAM? What RAM is used for? What happens to the contents of RAM when power is off?***

**RAM** (Random Access Memory): a volatile memory, used only for temporary storage. When power is off, the contents of RAM are erased.

***(3) What is Secondary Storage? What Secondary Storage is used for? What happens to the contents of secondary storage when power is off?***

**Secondary Storage** [disc drive] is a type of memory that can hold data for long periods of time. When power is off, the contents are erased.

**(III) Java vs. C++**

***(1) What is the difference in Garbage Collection?* Java -** memory is **garbage collected.** No manual memory allocation or deallocation.

***(2) What is the difference in pointers?* Java- no pointers**, “Everything is a pointer.”

***(3) What is the difference in operator overloading? What is the difference between an assignment and test for meaningful equality in Java?***

**Java** - no operator overloading. Clear difference between **assignment** [*==*] and **test for meaningful equality** [*st1.equals(st2)*].

***(4) What is the difference in multiple inheritance? What replaces multiple inheritance in Java?* Java** – **No Multiple inheritance**, replaced with **interface**.

***(5) What is the difference in passing values to function?* Java** – pass by value. **C++** – pass by value, pointer, reference.

***(6) What is WORA/WORE/WOCA? What language is more portable? What is created for many environments in Java and C++?***

**Java** – write once, run anywhere, everywhere [**WORA/WORE**] on any device with JVM. **JVM** is created for many environments. **[Highly portable]**

**C++** – write once, compile everywhere [**WOCA**]. **Compiler** is created for many environments. Compiler is more difficult to develop.

**(IV) Compile and Run**

***(1) What is source file? What does source file contain? Where source file is written?* Source file** [**text editor**] contains programmer-readable code.

***(2)What does Java compiler do? Where does compiler store information? What command starts compiler? What errors are caught by compiler?***

**[*javac test.java*]** Compilertranslates the **source file** into a **byte-code file** [platform independent code],and stores it into **test.class** file [**syntax errors**].

***(3)What does JVM do? What command starts JVM? What errors are caught by JVM?***

**[*java test*]** The **JVM** converts **class file** into a **machine code** [platform-dependent code]**,** computer run a program [**runtime errors-** logical errors].

**(V) Source File Declaration Rules**

***(1) What is Java program?*** A **Java program** is a collection of objectstalking to other objects by invoking each other's *methods*.

***(2) How does the name of Java file depend on the name of public class? How many public classes source file have? How to name Java classes?***

Name of Java file = name of the public class [1 public class per source file]. **Naming:** uppercase nouns. **Ex:** *public class Dog{ }* must be in *Dog.java* file.

***(3) How many non-public classes can source file have? Does the name of Java file depend on the name of non-public class? What Java app must have?***

A file can have many non-public classes. Files with non-public classes can have any filename. Every Java application must have a **main method**.

***(4) If the class is part of the package, what is the order of the following statements – class declaration, import statement, package statement?***

If the class is part of a package: **1.** *package statement.***2.** *import statement.* **3.** *class declaration.*

***(5) What can/cannot legal variable identifier start with? Is variable identifier case-sensitive?***

**Legal variable Identifiers:** must start with a letter, ($), (\_). **No number** or **space**! Identifiers are case-sensitive.

**(VI) Access Modifiers**

***List access modifiers [Class, Package, Subclass, World].* Public:** ok,ok,ok,ok. **Protected:** ok,ok,ok,x. **Default:** ok,ok,x,x. **Private:** ok,x,x,x.

**(VII) Variable Declarations**

***(1) What does primitive variable represent? If primitive variable is declared, can type be changed? value? Can objects be created out of primitive variables?***

**Primitive variable** represents **RAM** location. If declared, type cannot change, value can change. Objects cannot be created.

***(2) List all primitive variables [default values, bytes #, signed/unsigned, designation].***

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **byte** [def: 0]: 1 byte | **short** [def: 0]: 2 byte | **int** [def: 0]: 4 byte | **long** [def: 0]: 8 byte **[L]** | **float** [def: 0.0]: 4 byte **[F]** | **double** [def:0.0]:8 byte |
| A **boolean** [def: false] true/false, the range is JVM dependent. | | | **char** [def: '\u0000'] 2-byte, [‘’], an unsigned int. | | |

***(3) Characteristics of primitive # types: What is signed? How many bits in a byte? Range formula. Is it the same on all systems? Declare Sc. not.***

Signed (-/+), 8-bit bytes, the leftmost bit - sign [1(-), 0(+)]. **Range**: **min** [-2(bits-1)], **max.** [2(bits-1)–1]the same on all computers. **Sc. not.:** 1.49E11 (1.49E-11).

***(4) What is the main difference between character and short?* char vs. short:** char has a larger range [short uses 1 bit for sign].

***(5) What does reference variable refer to? Once declared, can type be changed? object? Can it be reassigned to other objects? What does type determine?***

**Reference variable** refers to object, subtype, interface. Type cannot change, object can. It can be reassigned to other objects. Type determines the methods.

***(6) What is the other name of instance variable? Does it need to be initialized? What is the scope?***

**Instance variable** [field, property, attribute] belongs to object. **Scope:** initialized [default values] when instance is created, live until instance is removed.

***(7) Where is the local variable declared? Does it need to be initialized? What is the scope?***

**Local variable** is declared within a method. **Must be initialized** with default values. Local variables live until method remains on the stack.

***(8) What is a final variable? Can the final primitive/reference variable be reassigned? Can the state of final reference be changed?***

**Final** is a constant, cannot be changed once declared. Final **reference variable** allows changing the state of the object, can’t be reassigned to other objects.

***(9) What is the transient variable? What kind of variables does it apply to?***

**Transient variable** [Java 5] are ignored by JVM when object is serialized.Only applies to instance variables.

***(10) What is volatile variable? What kind of variables does it apply to?***

**Volatile variable:** thread accessing the variable reconciles its own private copy of variable with master copy in memory. Only applies to instance variables.

***(11) What is a static member? How many copies of the static member in the memory? Does it need to be initialized? Can static method access non-static variable? How to access instance variable? Can static method be overridden? Redefined? What is the scope of static member?***

**Statics** declared within class. **No initialization** [default]. One copy regardless of # instances. A static method can't access a non-static variable. **Access**: (.) with class name. Static methods **can't be overridden** but redefined. **Scope:** variables created when class is loaded, live until class stays loaded in JVM.

**(VIII) Methods**

***(1) What is the method naming standards? argument? parameter? How to use final parameter? What values can be returned?[4 types]***

**Methods [behavior**]: storage of the class' logic, data manipulation. **Naming standards:** camelCase, the verb-noun pairs. **Ex**: *get/setBalance, isEmpty*

**Arguments:** values in parentheses when method is called: *doStuff("a", 2);*

**Parameters:** values in **method's signature** [method name, types]: *void doStuff(String s, int a) { }* **Final parameter**: *void doStuff(final int a) { }*

**Return: a.** null: *return null*; **b.** array: *return new int[] {1, 2}* or *int[] nums = {1,2};**return nums;* **c.** *return (int) f;* **d.** object type/subtype: *return new Horse();*

***(2) Methods: System.out.println(), Math.sqrt(), toString().***

**System.out.println()**: **System class** holds **out object.** The **println method** is a method of **out object** - takes a **string parameter**.

**Math.sqrt()** does not use any object, it invokes a **static method sqrt** - takes a number **parameter**.

**toString** returns a String that represents the state of the object – typically overridden. [(object’s class name)@(int value, object memory address)].

***(3) What is abstract method? How to declare an abstract method? Where should abstract method be implemented? What modifiers cannot be used? [3]***

**Abstract method** - *declared* as abstract, not *implemented*. No curly braces, no method body but a semicolon. If class has one abstract method, class must be abstract. *public abstract void showSample();* The first **concrete** [non-abstract] subclass must implement all abstract methods of the superclass. A method can never be marked as abstract/final, abstract/private, or static.

***(4) What is a native method? What is strictfp method? What is a synchronized method? How to declare a synchronized method? What is a final method?***

**Native method** indicates that a method is implemented in platform-dependent code, often in C.

**Strictfp method [class]** forces floating points to adhere to IEEE 754 standard. You can predict how fp behaves regardless of platform JVM is running on.

**Synchronized methods** can be accessed by only one thread at a time. *public synchronized Record retrieveUserInfo(int id){}*

**Final method**: cannot be overridden in a subclass for safety and security. *public final void showSample(){}*

***(5) What is a method constructor? When is it invoked? What happens when there is no constructor? How to declare method constructor? What modifiers cannot be used? [3] What does the first line in constructor call? Do abstract classes have a constructor? interface? Is constructor inherited? Overridden?***

**Constructor** initializes instance variables at runtime. Class has at least one constructor. If no constructor, compiler generates **default no-arg.** Constructors must have the same name as the class in which they are declared. Constructors can't be marked static, final or abstract.The first line in a constructor: a call to super() or this(). Abstract classes have constructors, interfaces don’t. Constructors are never inherited, they are not methods, and cannot be overridden.

**(IX) Assignment**

***(1) Where do instance variables live? objects? local variables?*** Instance variables and objects - **heap**. Local variables - **stack**.

***(2) What do primitive variables hold? What do reference variables hold?***

**Primitive Variables** hold the actual data item with a designated type. *byte val = 6* [*val* holds an actual bit pattern is 00000110, representing the 8 bits].

**Reference variable** holds bits representing the memory address, a way to get to the object on heap. [or null if the reference is not assigned].

***(3) If primitive/reference variable a is assigned to primitive/reference variable b, then a/b is changed. How does the change effect the other variable ?***

Assigning primitive variable to another primitive variable:*int a = 6; int b = a;* if the contents of a or b changed, the other variable **is not affected**.

Assigning reference variable to other reference variable: *Obj a = new Obj (5,10); Obj b = a;* if the contents of a or b changed, the other variable **is affected.**

***(4) How values are passed in Java? How primitive variables are passed to the function? reference variables?***

Java is pass-by-value for all variables running within a single JVM.**Pass-by-value** - pass-by-copy-of-the-variable.

**Primitive variable** - copy of the bits of actual value is passed. Method gets its own copy of the value. Method does not modify the original varue.

**Reference variable -** copy of the bits of memory address is passed. Method gets a copy of memory address. The caller and the called method have identical copies of the reference; they refer to the same object on the heap (*not* a copy). Method modifies the original object.

**(X) Package**

***(1) What is the package? How does the package name changes the name of the class? What is an import statement? Which package no need to import?***

**Package** - group of related classes. **Naming:** append class name to reverse domain name. I**mport** tells the compiler where package class located. **Ex**: if domain name is geeks.com, class Utilities. Name: *package com.geeks.client* becomes *com.geeks.client.Utilities*. **java.lang package** –imported automatically.

**(XI) Reference *this***

***(1) What is shadowing? What does reference this refer to? Why reference this is used?***

**Shadowing** [name collision]: name local variable = name instance variable. ***this*** refers to the currently executing object. *this* resolves the name collision.

***(2) Give an example of Rectangle constructor [name, length] using this? Give an example of an overloaded Rectangle constructor [“Unknown”, length]***

***a)*** *public Rectangle(String name, int length){ this.name = name; this.length = length;}* ***b)*** *public Rectangle(int length){this(“Unknown”, length);}*

**(XII) Class**

***(1) What are the benefits of Objected Oriented programming? [3 benefits] What is UML?***

**Benefits of Object Orientation** (OO): flexibility, maintainability, extensibility. **UML** [Unified Modeling Language] is graphic depiction of OO systems.

***(2) What is the class? Where related classes need to go? What is the superclass of each class? What is the object? Object state? How is the object created?***

**Class**: template [blueprint] that describes the state, behavior of the object. Put classes into packages. Every class is a subclass of class Object.

**Object:** an instance of the class created at runtime when **new**is used. Each object have its own unique set of instance variables, **state**, as defined in the class.

***(3) What is Enum? Where an enum should be declared? What is coupling? cohesion?***

**Enum** [Java 5] restricts a variable to a few pre-defined values. Enums must not be declared within a method. [semicolumn, caps are optional]

**Coupling** [classes interact with each other]: what one class knows about other class. **Cohesion** [design of a class]: class has focused set of responsibilities.

***(4) What are the class access modifiers? How to use a class if it is in the different package?***

**Default**: A class can be seen only by classes within the same package. **Public**:*all* classes can see class. **Import** is needed when classes in different packages.

***(5) What is strictfp? final? abstract class? Can abstract class be instantiated? extended? What modifiers aren’t allowed? Can it have non-/abstract method?***

**strictfp class** conforms a class to the IEEE 754 for fp. Without modifier, floating points used in the methods might behave in a platform-dependent way.

**Final Classes:** no class can extend a final class, cannot be overridden. **Ex:** the String final class cannot be subclassed.

**Abstract Classes** cannot be instantiated, only extended. It cannot be abstract and final. Non-abstract methods can be in abstract class.

**(XIII) Reading keyboard input with Scanner Class**

***(1) How to declare a Scanner class? What is the keyboard buffer? What is a Scanner method waiting for?***

Import **java.util.Scanner**.*Scanner keyboard = new Scanner(System.in);* The keystrokes stored in **keyboard buffer.** The method waiting for the enter key.

***(2) What is the difference nextDouble() and nextLine()? How to resolve a problem of reading a String followed by reading a double variable?***

***nextDouble()*** skips a leading new line character, stops reading. ***nextLine()*** doesn’t skip a new line character. **Solution:** [*nextDouble(),nextLine(),nextLine()*].

**(XIV) Encapsulation**

***(1) What is encapsulation? What is the access modifier of instance variable? methods? What is setter/getter method [method signature]?***

**Encapsulation:** data hiding. **Instance variable- private, methods- public** [set/get<property>]. **Setter method** [mutator, changes a value]**:** public, void return type, argument of the property type. **Getter method** [accessor, retrieves a value]**:** public, no arguments, return type match the setter method type.

**(XV) Inheritance**

***(1) What is the inheritance? Why? Does superclass know about subclass? Where methods can be overloaded? overridden? Is there multiple inheritance?***

**Inheritance**: subclass extends superclass. The superclass knows nothing of subclass. Subclass methods override superclass methods. Methods can be overloaded in superclass/subclass. Inheritance allows a r**euse** of the classes. **No multiple inheritance**: A class must have one superclass.

***(2) Does subclass inherit public member? private member? Does it depend on a package? Can a private method be overridden by a subclass?\***

*S*ubclass inherits a public member [regardless of the package]. A private member is not inherited. A private method cannot be overridden by a subclass.

**(XVI) Is-A vs. Has-A Relationship**

***(1) What IS-A relationship? HAS-A relationship? What is an aggregation? How to create instance fields? What values are returned by getter method?***

**IS-A** is based on inheritance or interface implementation. ["A is a type of B."] If the expression “*A instanceof B*” true, then class A IS-A class B.

**HAS-A** is based on usage. Class A HAS-A Class B if code in class A has a reference to an instance of class B. **Ex:** The course has an instructor.

**Aggregation**:objects are made of other objects. **Good design:** separate related items into their own classes.

**a.** Use a **Deep Copy** [copy objects] when creating field objects **not** **shallow copy** [reference copy]. **Constructor:** *instructor = new Instructor(instr);*

**b. Return copy of field objects**, **not the originals**. *public Instructor getInstructor( return new Instructor(instructor) );*

**(XVII) Polymorphism**

***(1) What is polymorphism? What methods depend on? Why all objects are polymorphic? Does it apply to which variables? How methods get selected?***

**Polymorphism** ("many forms"): treat subclassas a superclass. Methods depend on variable type, applies to instance methods. *All* objects are polymorphic [pass the IS-A for class Object]. Refer to an object with a general reference type, but at runtime, dynamic selection based on object type(not referencetype).

**(XVIII) Interface**

***(1) What is interface? How to name interfaces? What are the modifiers for methods? variables? What kind of modifiers are not allowed?***

**Interface** [inheritance companion]: contract for *what* a class can do, but not *how* to do it, 100% abstract superclass. **Naming:** adjectives. **Ex:** *Serializable*.

Interface methods are **public abstract** [nobody, semicolon], no static/final/strictfp/native. **Constants**-public, static, final— no instance variables.

***(2) Can interface be extended? implemented by the other interface? How to implement methods? Can an implementation class be abstract?***

An interface can extendinterfaces. **Ex:** *interface Bounceable extends Moveable, Spherical* but cannot implement other interface. *public class Ball implements Bounceable, Runnable{}.* Provide concrete implementation for all methods from the declared interface. An implementation class can be abstract.

**(XIX) Overloading**

***(1) What are the requirements for method overloading [arguments, return type, access modifier, exceptions]? Where a method can be overloaded?***

**Overloaded**methods must have different arguments (optionally, different return type, different access modifier, new/broader checked exceptions).

A method can be overloaded in the same class or in a subclass. If argument match isn't found, the JVM uses arguments that are wider than parameter.

***(2) What determines the overloaded version of method? where? Where does a virtual method invocation happen? What is binding?***

**Reference type** determines the overloaded method at compiletime [based on name, parameter -**binding**]. The virtual method invocation happens at runtime.

**(XX) Overriding**

***(1) What are the requirements for method overridden [arguments, return type, access modifier, exceptions]?***

Arguments must not change; return must be the same [subtype], cannot have more restrictive access; no new/broader exceptions [eliminate].

***(2) Where a method can be overridden? What access modifier is not overridden?*** Subclass’s method can override superclass method (no private, final, static).

***(3) What determines the overridden version of method? where?*** ***How to call a superclass method from a subclass?***

**Object type** (type of actual instance on the heap) determines which method is selected at runtime.To **call a Superclass method**:*super.method(args){}*

**(XXI) Arrays**

***(1) What is an array? What does it store? What is the largest subscript? Can the size of array change? Do array elements get default values?***

**Array**- object on the heap, stores vars of the same type/subtype. The largest subscript : *array.length – 1*. Size cannot be changed. Elems get default values.

***(2) How arguments are passed to the main method from command line?***

**main method** takes an array of Strings. Items are typed on the command-line, separated by spaces, after class name: *java className One Two*

***(3) What is vararg? What is the basic syntax?*  Var-args [ellipsis]:** method with **variable-length argument list** accepts a variable number of arguments. **Basic syntax:** *public static int sum(int… nums){ }* A var-arg must be the last parameter.*void doStuff2(char c, int... x) { }*

**(XXII) Casting**

***(1) What is variable casting? What are the differences between types of the variable casting?***

**Variable casting** converts values from one type to another. **Implicit cast**: no cast, automatic conversion [**widening conversion:** *int y = 10; double x = y;].***Explicit cast**: cast, possible loss of precision [**narrowing conversion:** *double**x = 6; int y = (int) x;*].

**(XXIII) Wrapper classes**

***(1) What is wrapper class? Why wrapper classes are used? Are classes immutable? Can it be null? Can you widen it? Methods: valueOf, xValue, parseX***

Wrapper class exists for every primitive - the same name as type of primitive variable [capitalized] except for Character. It provides functions for primitives, used for adding to Collections. Classes are **immutable**: once declared, cannot be changed, can be null. Wrapper classes don’t widen from one to another!

*Integer i2 = Integer.****valueOf****("101011", 2);* *Integer i2 = new Integer(42); byte b = i2.****byteValue****();**double d4=Double.****parseDouble****("3.14");*

**(XXIV) Garbage Collection**

***(1) What is Garbage Collection? What gets cleaned up by GC? What runs GC? When does object get deleted? What exception is thrown for low memory?***

**GC** is automatic memory management in Java. The heap is the only part of memory cleaned by GC. It’s up to the JVM to run GC, not the user. An object is deleted when no live thread can access it. GC might/might not delete the object. If memory is low, GC runs before it throws an OutOfMemoryException.

***(2) How to eligibilize an object for GC?*** *Account account = new Account(500.0); Account tempAccount = account;**account = null; tempAccount = null;*

***(3) Can Garbage Collection be forced? How to suggest Garbage Collection to JVM? [2 ways] What is finalize() method?***

**GC** isn’t forced, suggested to JVM. ***a)*** Ask for GC: **System.gc(); *b)*** The **Runtime class** has a single object (a Singleton) for each main program. Runtime is direct communication with JVM. [*Runtime rt = Runtime.getRuntime(); rt.gc()*] ***finalize()*** [no arguments, void type - inherited from Object] It is called only once before an instance of the class is destroyed by GC. It can save an object from deletion by passing a reference to the object to another object.

**(XXV) Thread**

***(1) What is Thread? [2 definitions] Where does the Thread live? How many call stack per one thread? List Object/Thread/Runnable methods?***

***a.*** An instance of **java.lang.Thread** lives/dies on the heap. ***b.*** Thread of execution - a process that has its own call stack, one call stack per thread.

**Class Object Methods [**wait(), notify(), notifyAll()**]; Class Thread[**start(), yield, sleep(), join()**]; Interface Runnable[**run()**]**

***(2) What is thread can guarantee? What is the min # of threads in a program? What is the 1st method on stack? What is the scheduler? runnable pool?***

Thread starts, runs to completion.At least 1 thread runs in a program. Thread has its own execution stack and lifecycle. The main() **-** **main thread – 1st** method on the stack. **Scheduler** decides which runnablethread from **Runnable pool [**not runnable queue] will run, threads move in/out of the running state.

***(3) What are the two ways to instantiate a thread? [2 ways] Methods: run(), start(), join(), sleep(), yield(). User vs. Daemon thread? List states of thread?***

**Extend java.lang.Thread class**: *class MyThread extends Thread {public void run() {…}} MyThread t = new MyThread();* [poor OOP; if Thread extended, cannot extend anything else]. **b. Implement** **Runnable**: *class Dog implements Runnable {public void run() {...}} MyThread t = new MyThread(new Dog());*

*Thread t = new Thread(new Runnable(){public void run(){…}});*

*t.start()-* thread is **alive**; starts a *Thread,* not *Runnable; new state -> runnable state*. A thread is **dead** once *run()* completes – cannot use *start()* again.

*t.join()* - lets thread "join onto the end" of another thread. The thread B won’t be runnable until A has finished (and entered the dead state).

*t.sleep()* - static method, slows down a thread, helps threads to get a chance to run. *try{ Thread.sleep()}catch(InterruptedException e){…}*

*t.yield()* - static method makes the currently running thread head back to runnable to allow other threads of the same priority to get their turn.

**User vs. Daemon thread:** JVM exits an application only when all user threads are complete, regardless of the state of any daemon threads.

* **New** [not alive]**:** after the Thread instance has been created, but the start() method has not been invoked on the thread.
* **Runnable** [alive]**:** eligible to run, but scheduler has not selected it to be the running thread. *start()* is invoked.
* **Running:** the thread scheduler selects thread (from the runnable pool) to be the currently executing process.
* **Waiting/blocked/sleeping** [alive, currently not eligible to run]**: i**t is not runnable, but it might return to a runnable state later.
* **Dead** [not alive]**:** *run()* method completes. Once a thread is dead, it can never be brought back to life.

***(4) How does scheduler work? What is the priority? How to set thread priority? What is time slicing? How does scheduler organize threads? [2 ways]***

The **scheduler** has a **priority-based scheduling**. Threads have priority (1-10). **Time slicing** [not *all* JVMs]*:* thread is given time, then sent back to runnable to give other thread a chance. Usually, the running thread will be of equal or greater priority than highest priority threads in the pool. **a.** A scheduler can pick a thread to run until it blocks or completes. **b.** A scheduler time slice the threads in the pool to give everyone an equal opportunity to run.

**Setting a Thread's Priority:** A thread gets a default priority*.* Set a thread's priority using *setPriority()* on a Thread instance. The**default priority**is 5*,* the Thread class has constants that define the range of priorities: *Thread.MIN\_PRIORITY (1); Thread.NORM\_PRIORITY (5); Thread.MAX\_PRIORITY (10);*

***(5) What is the condition of threads? What modifiers avoid data corruption? What is lock? How does lock operate? What is the deadlock?***

**Race condition -** threads access the same resource (instance variables), and produce corrupted data. **Solution:** **private** variables, **synchronized** methods that change variables. Objects have a built-in **lock** [**1 lock per object** associated with *this*] - synchronized non-static method. The lock is free until some other thread enters a synchronized method. If one thread has picked up the lock, no thread can pick up the lock until the first thread releases the lock. If the lock is taken, the thread goes into a pool and sits there until the lock is released, then thread can become runnable/running. If a thread goes to sleep, it holds locks.

**Deadlock**: two threads are blocked, each waiting for the other's lock. Neither can run until the other gives up its lock, so they'll sit there forever.

***(6) How to synchronize code? static method? What static/instance methods lock on? Can it run at the same time? Object methods: wait, notify, notifyAll***

**Synchronize a block of code** if it changes data: *class SyncTest { synchronized(this) { System.out.println("synchronized"); }}}* Synchronized **static methods**:*public static synchronized int getCount() { return count; }* The static method locks on a Class instance while the non-static method locks on *this* instance. The static synchronized method and non-static synchronized method will not block each other, run at the same time.

***wait()* -** thread executing the code gives up its lock on the object immediately**. *notify()* -** thread does not give up its lock at the moment.

***notifyAll()* -** notifies all waiting threads that are waiting on a particular object, all of the threads will be notified and start competing to get the lock.

**(XXVI) Operators**

***(1) What is overloaded (+)?overloaded (</=/>)? How to use division operator for fractions? What is ternary operator? Meaningful equality vs. equality.***

(+) concatenation operator if either operand is a String. Unicode char is used as # for comparison. **Ex:** *if(‘A’ < ‘B’) System.out.println(‘A’<‘B’);*

**Division operator:** *double num = 17/3* [The fractional part of the result is discarded] instead *double parts = 17.0 / 3*

**Conditional Operator** is a ternary operator. The operator: *x = (boolean expression) ? value to assign if true : value to assign if false.*

**(==)** compares the variable bit pattern [**primitive**-actual values, **object**-address]. The **equals** tests if objects are "meaningfully equality.”

***(2) Short-circuit [&&], non-short-circuit [&]. Short-circuit [||], non-short-circuit [&]. Non-short-circuit exclusive-OR******[^]. Pre-fix vs. Post-fix.***

**The short-circuit [&&]** evaluates the left side of the operation first, and if it resolves to false, it doesn't look at the right side of the expression.

**Non-short-circuit**logical operators [(&), (|)] evaluate both sides of the expression, very inefficient.

**The short-circuit [||]**: evaluates the left side of the operation first, and if it resolves to true, it doesn't look at the right side of the expression.

**Non-short-circuit exclusive-OR** **[^]** evaluates to true if EXACTLY one operand is true.

**Pre-fix:** *int m = 7; int a = 2 \* ++m;*// a = 16; m = 8. **Post-fix: (1)** *int m = 7; int a = 2 \* m++;*// a = 14; m = 8

**(XXVII) String**

***(1) Are strings immutable? Strings and memory: how/where strings are stored? How to declare a string? How to declare a String? [2 ways]***

**Strings-** immutable objects, if it is created, it can’t change. If there is a change, a new String object is created. **Ex:** *String s=”abc"; s=s.concat(”abc");* 3 Strings in memory, original String didn’t change but it is not referenced. StringBuffer/Builder objects can change.

**Strings and Memory:** JVM sets aside a "**String constant pool**." [memory efficient] When compiler sees a String, it checks the pool to see if an identical String already exists. If it does, the reference to the new literal is directed to the existing String, no new String literal object is created. Several references might refer to the same String without knowing, it is bad if any of them could change then String's value.

**a)** *String s = "abc";* [1 String object, 1 ref. var.] - “abc" goes in the pool and s will refer to it.

**b)** *String s = new String("abc");* [2 objects, 1 ref. var.] - new String object created in non-pool memory, s refers to it. The "abc" will be placed in the pool.

***(2) StringBuffer vs. StringBuilder. What is the capacity?***

If String changes a lot, abandoned Strings get in the String pool. **Java.lang.StringBuffer/Builder** [Java 5] **Classes** modify string, no abandoned Strings **[capacity:** string length + 16 chars**]**. StringBuilder/Buffer class have the same API, but StringBuilder is not thread safe [not synchronized]. Use StringBuilder instead of StringBuffer whenever possible because StringBuilder will run faster.

**(XXVIII) Flow Control**

***(1) How to stop the entire loop? stop current execution? stop program execution? List loops. [4 loops] What is the labeled variety? sentinel value?***

**break** – stops the entire loop. **continue** – stops the current execution. **System.exit(0)** - program execution stops, JVM shuts down.

**while loop**: pre-test loop stops iteration if the condition is false from the beginning, use it with **sentinel value** [**Ex:** –1 to end iteration].

**do-while loop**: post-test loop iterates at least once. **for loop**: pre-test loop [ built-in initializing, testing, updating] iterates the exact # of times.

**Enhanced for loop**: compiler converts it to a traditional loop that uses an iterator.  **Labeled varieties** indicate which of the nested loops to break from.

**(XXIX) File Navigation and I/O**

***(1) File Objects, FileReader/BufferedReader, FileWriter/BufferedWriter, PrintWriter, Console. Methods: exists(), createNewFile(), flush(), close().***

**File Objects:** files [no data] or directories. **FileReader** reads the stream of chars [fixed # of chars] from file. **BufferedReader** reads large chunks of data from a file at once, keeps data in a buffer [min # of IO operations]. **FileWriter** writes the stream of characters [fixed # of chars] to file. **BufferedWriter** writes large chunks of data to a file at once, keeps data in a buffer [min # of IO operations]. **PrintWriter:** have methods format(), printf(), and append().

**Console** [ Java 6] reads input from console and writes formatted output to the console.

**boolean exists()**: method returns true if it can find the actual file. **boolean createNewFile()**: creates a new file if it doesn't already exist.

**flush()**: writing the last data to the file [reading - no flushing]. **close()**: use when done using the file, frees up operating system resources.

**(XXX) Parsing, Tokenizing, and Formatting**

***(1) Search:***  ***word char, non-word char, digit, non-digit, whitespace char, non-whitespace char; word of 2-20 chars, min, no max, hexadecimal, wildcard.***

A **regex search** [**java.util.regex.\***]: **[A-Za-z]**, **[0-9]**; **[^A - G]** **-** not A – G; **w** or **[A-Za-z0-9\_]** - a word character; **\\W** - non-word character; **\\d** - digit; **\\D** - non-number; **\\s** **-** a whitespace character; **\\S** **-** non-white space characters; **[A-Za-z]{2, 20}** or **\\w{2, 20}** - search for a word 2-20 chars; **\\{5, }** – min, no max; **{5}** – find a specific number ; **0[xX][0-9a-fA-F]** – hexadecimal; **&&** - intersection of sets. "**.**" – wildcard.

***(2) Tokenize, split(), formatting with printf() [flags, width, conversion].***

**Tokenizing**: breaking string to pieces. **Tokens:** actual data, kept. **Delimiters** separatetokens, discarted.

The String **split()** takes a string, returns a String array of tokens.

**Formatting with printf():** *printf("formatString", argument(s));* The format string: *%[arg\_index$][flags][width][.precision][conversion]* [ ] - optional.

**Flags**: "-" Left justify; "+" Include a sign (+/-); "0" Pad with zeroes; "," group separators (123,456); "(" Enclose negative numbers in parentheses

**Width**: min # of characters to print in a given field. Ex: %7d - 7 spaces wide. Precision: # digits to print after decimal. **Ex:** %.2f - 2 digits after decimal.

**Conversion:** The type of argument to be formatted. [**%b** boolean, **%c** char, **%d** integer, **%f** floating point, **%s** string]

**(XXXI) Javadoc**

***(1) What is a documentation comment? javadoc? What is the command to run javadoc? Write a method/class comment with tags [description + 3 tags]***

**javadoc** processes documentation comments: reads source code files, generates HTML files - comments are part of HTML.

***javadoc sourceFile.java***Documentation files are generated in the same directory as a source file [one of the files is index.html].

**Documentation comment**: /\*\*… \*/ before a class/ method header, description of class/method. */\*\* This class calculates company payroll.\*/*

**Method comment**: */\*\*  This methodName Sets...****@param*** *start The…* ***@return*** *The…* ***@exception*** *IllArgExc When… \*/*

**(XXXII) Recursion**

***(1) What are the cases of recursion? [2 cases] What does substitute recursion? Which solution is more efficient? why?***

**Base case**: no recursion, solves and returns. **Recursive case:**  the problem is reduced it to a smaller but similar problem and calls itself to solve it. Recursive problem can be solved **iteratively**. **Overhead** takes place with each method call**:** recursive algorithms are less efficient. [JVM allocates memory for parameters/local variables, and stores the address of the program location where control returns after the method terminates].

**(XXXIII) “Big O” notation**

***(1) Provide the name and an example for each running time: O(1), O(log n), O(n), O(nlogn),* O(n2), O(nx)**

* **O(1) [constant time]:** an increase in input size does not change the running time of the algorithm. [**Ex:** best case of binary search]
* **O(log n) [logarithmic time]:** an increase in input size results in a small increase in the running time of the algorithm. [**Ex:** binary search algorithm]
* **O(n) [linear time]**: an increase in input size results in a proportionate increase in the running time of the algorithm. [**Ex:** sequential search]
* **O(nlog n):** input size increase results in a slightly greater increase in the running time. [**Ex:** average-case Quicksort, worst-case MergeSort/Heapsort]
* **O(n2) [quadratic time]:** an increase in input size results in increase of the running time. [**Ex:** worst-case bubble/selection/insertion/quicksort]
* **Polynomial time** is the union of all classes O(n1), O(n2), O(n3), …

**(XXXIV) Handling Exceptions**

***(1) What are subclasses of a Throwable class? What is superclass of a Throwable class? All exceptions are instances of what class? What is an exception?***

Exception and Error are subclasses of **Throwable class** thatderives from the Object class. All exceptions are instances of Exception class. Error classes represent unusual situations that are not caused by program errors [JVM running out of memory]. Program won’t recover from Error, don’t handle it.

**Exception** is object generated in memory due to error/unexpected event. When exception occurs, exception is "**thrown**." "**Exception handler**" catches the thrown exception.Exception handlers detect exception, prevent it from halting program. **Default exception handler**: print error, crash program.

***(2) try/catch/finally.***

**try** defines a block of code in which exceptions may occur. Try must be used with catch and/or finally, never alone.

**catch** clauses match a specific exception to a block of code that handles it. Catch blocks immediately follow the try block. Catch blocks must all follow each other, without any other statements in between. Once control jumps to the catch block, it never returns to complete the balance of the try block.

**finally** encloses code that is executed after the try block, whether an exception was thrown or not. This is the place to close your files, release your network sockets, and perform cleanup code. If the try block executes with no exceptions, the finally block is executed immediately after the try block completes.

***(3) What is a call stack? Top of the stack? StackOverFlow? printTraceStack? unwinding the stack? catchall exception handler?***

**Call stack**- the chain of methods program executes to get to the current method. **Ex:** main() calls a() which calls b() which calls c(). Call stack [bottom-up]: main, a, b, c. Last method - **top of the stack**. An exception that's never caught will cause an application to stop running. It's possible to grow the stack so large that the OS runs out of space to store the call Stack, **StackOverflowError**. Throwable has **printStackTrace()** - prints stack trace [recent method first and works its way down the call stack - **unwinding the stack**]. *try {}catch (IndexOutOfBoundsException e) {e.printStackTrace();}*

Avoid **catchall exception**:*catch (Exception e){}.* Specific exceptions go first: *try {} catch (FileNotFoundException ex) {}catch (IOException e) {…}*

***(4) What exceptions does compiler check? How to deal with exceptions? Throw vs. throws. Checked vs. unchecked exceptions.***

If method throws a checked exception: **(1)** It must **handle** the exception, or **(2)** it must have a **throws clause** listed in the method header.

**throw** creates and throws exception. **throws** clause informs the compiler a method throws exceptions, it allows a method to re-throw an exception that has not been dealt with: *void myFunction() throws MyException1, MyException2 {}* **NOTE:** if *main* method does not perform IO operations, but calls a method *buildFIle* that performs IO, both methods should have a *throws IOException* clause in their headers. Otherwise, a compiler error occurs.

**Non-Runtime Exceptions** - "**checked**" exceptions, compiler checks handling/declaring,must be caught. If method that throws checked exception invoked and checked exception is not caught, code will not compile. **Runtime exceptions** [java.lang.RuntimeException] are **unchecked** exceptions.

***(5) What are the assertions? How to enable/disable/select assertions? What happens when the code is deployed? How to declare an assertion? [2 ways]***

**Assertions:** assumptions at development. [Java 6- assert keyword by default.] Assertions disabled by default, enable or disable your assertions at **runtime**.

**Enabling Assertions at Runtime**: *java –ea [or –enableassertions] com.geeks.TestClass*

**Disabling Assertions at Runtime:** *java -da [or-disableassertions] com.geeks.TestClass*

**Selective:** *java -ea -da:com.geeks.Foo* [ disable assertions in a single class, but keep them enabled for all others]

The assertion code evaporates when the program is deployed, leaving no overhead or debugging code to track down and remove.

**Declaration: a.** assert (y > x); **b.** assert (y > x): "y is " + y + " x is " + x; // generates a message containing a debugging information.

**(XXXV) Object Serialization**

***(1) What is serialization? transient variable? Do class’s reference field need to implement Serializable? sublclasses? Can statics be serialized?***

**Serialization:** converts object to series of bytes, saving it to file [**transient** instance vars are ignored]. Superclass must implement **Serializable interface** – no methods/fields, classes -fields of class must implement Serializable, subclasses don’t. When you serialize an object, it saves object's entire "object graph." Serialization is not for statics: Static variables are not as part of the object's state.

***(2) What is Deserialization? What happens to transient variables at deserialization?***

**Deseriazation** - reconstructing a serialized object. **Deserializing object:** The constructor does not run. Instance variables are NOT given their initially assigned values. **Transient variables** will not be restored to their original state. Object references marked transient will always be reset to null.

**(XXXVI) Generics**

***1. What is a generic class/method? What are generic symbols? What variables can be generic? What is raw type? How to use raw ArrayList? How to pass generic type as an argument to a method? What is wildcard? Extends vs. super. How to define any type that is a number and pass it to the function? Can generic type have many parameters? Can interfaces be generic? How to implement is? How to constraint a type that implements the Comparable interface?***

**- Generic class/method [Java 5]** allows specifying types. If wrong type, **error** occurs at **compile time**, compiler performs type-checking. The **ArrayList** can be generic class.*ArrayList<String> myList = new ArrayList<String>();* **Non-generic ArrayList class** –use the cast operator with the get method.

**- T** - general type; **S** - general type when T was already used; **E** – type of element of Collection; **K** – key in a map; **V** – value in a map.

**- Only Reference Types Can Be Generic Parameters. Ex:** Double, Integer types are passed instead of the primitive double, int types.

**- Raw type** [type: Object]**:** creating an instance of generic class without providing type argument [not recommended]. *ArrayList myList = new ArrayList();*

**- Pass instances as arguments to method: *public static void printPoint(Point<Integer> point){ … }* [Only an instance of Point<Integer> can be passed to it.]**

**-** Use wildcard [**type** **?**] to accept argument of any type: ***public static void printPoint(Point<?> point){ … }***

**-** Defining any type that is Number [**java.lang package**] or a subclass of Number: ***public static void printPoint(Point<? extends Number> point){ … }***

**Alternative:***public static <? extends Number> void printPoint(Point<T> point){ … }*

**-** The **extends** Constrains a Type to an Upper Bound. **Ex**: *<? extends Number>*: any type below Number (or Number itself) but not above Number.

**-** The **super** Constrains a Type to a Lower Bound: **Ex**: *<? super Integer>*: Integer, or any superclass of Integer [Number, Object].

**- A generic class/method can have multiple type parameters:** *public class MyClass<T extends Number, S extends Date>{…}*

**- Interfaces can be generic:** *public interface Comparable<T>{ int compareTo(T o);}* **Implement**: *public class Tree implements Comparable<T>{ … }*

**-** E is constrained to types that implement the Comparable interface: *public static < E extends Comparable<E> > int sequentialSearch(E[] arr, E val){}*

***2. What is an erasure? What happens to generic type at source code level? byte code level? Give an example of class, method, and method with extends.***

**Erasure:** compiler processes a generic class/method, it erases the generic notation and substitutes an actual type for each type parameter. **Generic notation** exists in the **source code**. Compiler converts it into a **raw type**, a non-generic type, at the **bytecode level**. Compiler also inserts cast operators.

**Class:** *public class Point<T> { private T num;…}*🡪 *public class Point { private Object num;…}*

**Class with an Extends Constraint:***){…} public class Point<T extends Number>{private T num;…}* 🡪 *public class Point{private Number num;…}*

**Method 1:** *public static <E> void displayArray(E[] arr){…}* 🡪 *public static void displayArray(Object[] arr){}*

**Method 2:** *public static < E extends Comparable<E> > int search(E[] arr, E val)* 🡪 *public static int search(Comparable[] arr, Comparable val) {}*

***3. What are the restictions of the generic type? [4 restrictions]***

**- Cannot create an instance:** *public class MyClass<T>{ public MyClass(){ T myObject = new T(); //* ***Error!*** *} }* **but** *E[] list = (E[]) (new Object[5]);*

**-** Cannot create an array of genetic class: *ArrayList<String>[] a = new ArrayList<String>[100];*

**-** A generic class’s type parameter cannot be the type of a static field, and cannot be referred to in a static method.

**-** Cannot make an exception class generic.

**(XXXVII) Java Collections Framework**

***What is a collection? What is java.util.Collections? What is java.util.Collection? Does Set, List, Map, Queue extend/implement Collection? What is List? Set? Map? [order, duplicates] What is queue? What is an iterator method? iterator? Overriding equals and compareTo methods.***

**collection** – data structure, objects stored and iterated. **Element** is object in a collection. **java.util.Collections class** holds static utility methods for use with collections. **java.util.Collection:** top interface; **Set, List, Queue** [not implements] extend Collection interface. **Maps** do not extend Collection.

**java.util.List [array] —**an **ordered sequence** of elements. Element are accessed by **index**. **Duplicates** are **allowed**.

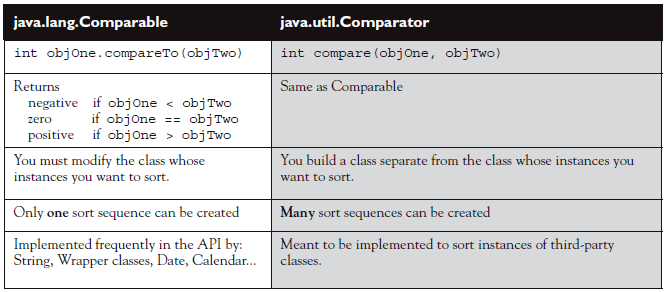
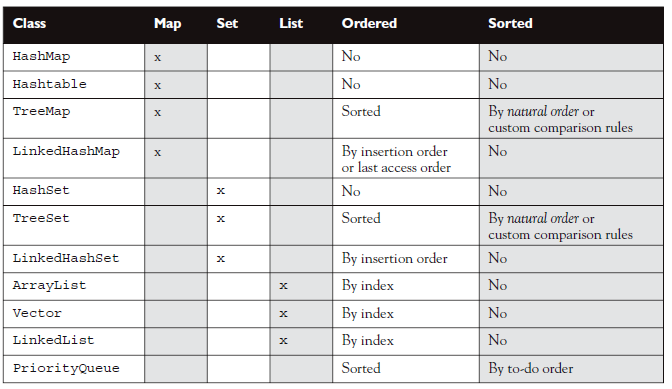
**java.util.Set—** an **unordered sequence**, optimized for searching of large amount of data. **No duplicates**, each element is unique.

**java.util.Map—**element [key is associated with value]. Key is used to locate value. **No duplicate keys** but **duplicate values** are allowed.

**Queues** – an **unordered sequence**, elements are arranged by the order in which they are to be processed.

**Iterator method [**Collection interface] returns a reference to an Iterator object. **Iterator** is an object used to retrieve the elements in a collection.

**override equals()** - take an argument of type Object; **override compareTo()**[implemented once in a class] – take an argument of the type you’re sorting.



**(XXVIII) Array-Based Lists**

***(1) What’s structure ArrayList use? What is it best for? What is size and capacity of an arraylist? Autoboxing/unboxing. Change capacity? Memory type?***

**ArrayList** uses an array. **Best for:** fast random access, a minimum number of insertions and deletions, and thread safety is not a concern.

**ArrayList** [**java.util.ArrayList** - an array of unfixed size]uses an array to hold elements. **Resizable**: expands/shrinks. **Capacity** - 10 elements.

- Java **autoboxes** [add method] and **unboxes** values that are retrieved [get method].

- Instantiation: to **change capacacity**: *ArrayList<String> myList = new ArrayList<String>(100);* **or** *List<String> myList = new ArrayList<String>(100);*

- An ArrayList **holds objects**: *ArrayList<BankAccount> account = new ArrayList<BankAccount>();*

**Contiguous allocation**: allocates storage for successive elements in consecutive memory locations. A **drawback**: moving large numbers of elements.

***(2) What is vector? What is it best for? Vector vs. ArrayList.***

**Vector** uses an array. **Best for:** multithreaded lists. **Vector** = ArrayList, but Vector methods are **synchronized** **-** safe for **multithreading** - concurrent threads access data in the program’s memory at the same time. If a few threads attempt to modify the same data at the same time - data will be corrupted. Vector methods never simultaneously access an element in the Vector - no accidental data corruption. Vector is slower than an ArrayList.

***(3) What is LinkedList? What is it best for? How to instantiate? What is the running time? ArrayList vs. LL, array vs. LL. Memory type?***

**Linked list** uses connected nodes to hold elements. **Best for** large data where numerous insertions and deletions take place. **Node** has a link to its successor [a reference to the next node in the list], last node has a **null** reference. **Instantiation**: *List<String> myList = new LinkedList<String>();*

**- O(1) time:** insertion and deletion at the beginning of a list are very fast, it involves changing of a few references.

**- O(N) time:** finding, deleting, or inserting next to a specific item requires searching through, on the average, half the items in the list.

**- Array vs. LL:** An array is also **O(N)** but the linked list is faster because nothing needs to be moved when an item is inserted or deleted.

**- ArrayList vs. LL:** LL is better than ArrayList, LL uses exactly as much memory as it needs and can expand to fill all of available memory.

**- Linked allocation** uses a pointer to the memory location of its successor.

**(XXIX) Sets**

***(1) What is HashSet class? How does it store objects? What is HashCode method? why overridde? collision? How does HashSet add objects?***

**HashSet Class** uses hashCode for fast search. Every object has a **hashCode method**, inherited from the Object class - returns a unique value [based on object’s memory address]. Object class’s hashCode is overridden so that it returns the same hash code for equivalent objects. **Adding objects:** the HashSet calls the object’s hashCode method to get its hash code. It checks if an object with the same hash code is already stored in set. If an object with same hash code is found in set, it calls the equals method. If equals method returns true, HashSet does not store the object in set. **Collision:** the same hash code is calculated for two objects [different data], object is added to the bucket. **TIP**: if you override the equals method you should override the hashCode method.

***(2) What is initial capacity? Load factor? LinkedHashSet? Treeset?***

The **initial capacity** is the number of “buckets” that the HashSet initially has. The **load factor** is the percentage of the buckets that have to be occupied before the capacity is increased. **Ex:** with a load factor of 0.75, the capacity will be increased when 75% of the buckets have a value stored in them.

Having too many unequal objects with the same hash code will degrade the performance of an algorithm that uses the hash codes for searching.

**LinkedHashSet Class** accesses elements in order that they were inserted.

**TreeSet Class [SortedSet Interface]: natural order** [ascending-red-black tree]. **Comparator** compares objects [greater, less than, or equal]. Objects must implement Comparable interface that has compareTo method used to compare the calling object with the object that is passed as argument to the method.

**(XL) Map**

***(1) What is mapping? How data are stored/retrieved from a HashMap? HashTable vs. HashMap.***

Elements [**mapping**] consist of a **key** and a **value**. Key is stored by its hash code and it is associated with value Key is used to locate value. **HashMap** uses hash code to store and retrieve data. *Map<String, Car> carMap = new HashMap<String, Car>();*

**Hashtable** is the HashMap with synchronized methods. HashMap allows null values and 1 null key, a Hashtable doesn't allow null.

***(2) How to iterate through HashMap? [2 ways] What is LinkedHashMap? TreeMap?***

**No iterators.** **Iteration**: use **keySet method** to get **Set** with all keys, use the **values method** to get a **Collection** with all values, iterate over the resulting Set or Collection. **or** retrieve a single set that contains both the keys and the values [Key/value pair is stored together in a single element using **entrySet method**]. The **entrySet method** returns a Set of **Map.Entry** objects. Map.Entry has **getKey/getValue methods**. The elements in the set and collection are references to mappings, changes in elements in the set or collection are reflected in map, and vice versa.

**LinkedHashMap Class** extends HashMap that displays elements in: **Insertion order** order in which mappings were inserted into the map; **Access order** - order in which the mappings were last accessed (put or get). *Map<String, Car> carMap = new LinkedHashMap<String, Car>();*

**TreeMap Class [SortedMap Interface]:**A **SortedMap** is a map in which the keys are sorted in **ascending order**. Keys that are stored in a SortedMap must either implement the Comparable interface, or have a Comparator.*SorterdMap<String, Car> carMap = new TreeMap<String, Car>();*

**(XLI) Queue**

***(1) What’s queue? What structure handles it?***

**Queue Interface** are FIFO (first-in, first-out). **PriorityQueue** [Java 5]: queues are handled with a LinkedList/array. PriorityQueue are "priority-in, priority out" [not FIFO]. **Natural ordering** (first sorted elements are accessed first) or using a Comparator. The elements' ordering represents their relative priority.