Object-oriented design (OOD) is the process of planning a program based on "object", which contains encapsulated data and procedures grouped together to represent a type of entity. **Object-oriented programming (OOP)** is a programming paradigm based on "objects". So developers should firstly group both data/attributes (data members) and code/methods (member functions) together into a class, in order to represent a type of objects. Then they can create objects of the class and achieve required functionalities by handling those objects. (The methods can access and modify data fields directly within the same class. There are five basic concepts in OOD, which are: Classes/Objects, Encapsulation/Data Protection, Inheritance, Interface/Abstraction, Polymorphism.)

Neither Java nor C++ is pure object-oriented language. For Java, ① It has primitive data types (boolean, char, byte, short, int, float, double, long) which are different from objects. (A pure object-oriented language should contain only objects and treat all primitive values, such as integers and characters, as objects.) ② The "*static*" keyword in Java allows us to use a class method without creating an object of that class. ③ Some objects in Java, like String, can be used without calling its methods. For example, we can concatenate two strings by only using the arithmetic operator +.

For C++, ① The main function in C++ is always outside of any class, which means class and object are optional when programming. ② In C++, you can declare a variable globally, which is accessible from anywhere. (In Java, any declaration must be in a class) ③ A friend function in C++ will always violate the object-oriented feature.

Friend functions in C++ are functions that don't belong to the class (defined either globally or in another class) but still can access private and protected members of that class. They should be used when a function need to access private data of two or more different classes. (A friend can also be a class, in which case all members in the class are friends.)

Encapsulation is used to bind together both data and methods which manipulate those data, to make sure that certain data can only be used in a proper way and keep them safe from outside. **Advantage:** ① Data hiding: Have no idea about the inner implementation; ② Flexibility: Can make the variables of a class to be read-only or write-only, by only providing the get methods or the set methods; ③ Testing: Easy for a unit test; ④ Reusability: Can always reuse part of code or modify with new requirements. **Implementation:** In Java, encapsulation is achieved by class and access modifiers/access specifiers.

Access Modifier/Access Specifier is access restrictions to classes, methods and variables to limit certain access from other parts of program. There are 4 types of access modifiers: default (no keyword specified), private, protected and public.

	Default	Private	Protected	Public
Same Class	Yes	Yes	Yes	Yes
Subclass in Same Package	Yes	No	Yes	Yes
Non-subclass in Same Package	Yes	No	Yes	Yes
Subclass in Different Package	No	No	Yes	Yes
Non-subclass in Different Package	No	No	No	Yes

Inheritance allows us to arrange classes in a hierarchy, so that we can create a class and inherit its features from another existing class, reuse parts of code and run the program faster. (A class that is derived from another class is called a subclass/derived class/child class/extended class, and the class being inherited by another class is called a superclass/base class/parent class.) Usually the subclass will automatically inherit the data members and member functions of its superclass, and not inherit constructors, destructors and private functions of the superclass (Java doesn't have destructors because of the garbage collection). If an instance of the subclass is created, the constructor of the superclass will be called firstly before the constructor of the subclass (constructor chaining). There are 5 types of inheritance:

- **Single Inheritance:** A subclass has exactly one superclass.
- **Multiple Inheritance:** A subclass may have multiple superclasses and inherits features from all superclasses. It can be dangerous which may cause a diamond problem.
- **Multilevel Inheritance:** A subclass inherits from another subclass. That is, there are at least 3 classes. Class A serves as a superclass for the subclass B, and the subclass B in turn serves as a superclass of its subclass C.
- **Hierarchical Inheritance:** A superclass has multiple subclasses.
- **Hybrid Inheritance:** A mix of two or more types of inheritance.

Implementation: In Java, the keyword "*extends*" is used for inheritance. When an object of the subclass is created, a copy

```
class Parent{
    void show() { System.out.println("Parent's show()"); }
}
class Child extends Parent{
    @Override  // This method overrides show() of Parent
    void show() { System.out.println("Child's show()"); }
}
class Main{
    public static void main(String[] args){
        // A Parent type variable refers to a Parent object
        Parent obj1 = new Parent();
        obj1.show();          // Output "Parent's show()"

        // A Parent type variable refers to a Child object
        //This is called RUN TIME POLYMORPHISM.
        Parent obj2 = new Child();
        obj2.show();          // Output "Child's show()"

        Child obj3 = new Parent();  // Illegal, type mismatch
    }
}
```

of all methods and variables of its superclass will get memory in this object. Java achieves multiple inheritance via interface. Object class is the only one class in Java that has no superclass, and is the default superclass of any other classes that don't have a superclass. In Java, a subclass can inherit variables, methods and nested classes from its superclass, and can't inherit constructors, destructors and private members of its superclass (constructors and destructors aren't class members), but a subclass can invoke constructors of its superclass, and access private members with help of public or protected methods (like set and get) in its superclass. If a subclass overrides a variable from its superclass, while not override the method which will access that variable, then even called by the subclass, the method will always refer to the variable in its superclass. **Object Class (java.lang.Object)** is the top most class in Java, that any other classes are directly or indirectly derived from the Object class. The Object class is the only one class in Java that has no superclass.

Prototype-based Programming/Prototypal Inheritance/Prototype-oriented/Classless/Instance-based Programming is an OOP concept that behavior reuse is performed by reusing existing objects via delegation. **Delegation** is the language feature that supports prototype-based programming. So for comparison, inheritance allows one class to inherit from another class, while delegation allows one object to delegate to another. Javascript supports prototype-based programming.

Methods Overriding in OOP is a feature that allows the subclass to provide a different implementation of method that already provided by its superclass. So that the implementation in the subclass will rewrite/replace/override the implementation in the superclass with the same method name, parameters and return type.
In Java, the version of a method that will be executed is determined by the object that is used to invoke it. If an object of a superclass is used to invoke the method, then the method in the superclass will be executed; if an object of the subclass is used to invoke it, then the method in the subclass will be executed. Notice it's the type of the object being instantiated (not the type of the reference variable) that determines which version of the method should be executed.

Comparison of Inheritance in C++ and Java: ① In Java, all classes directly or indirectly inherit from the Object class; while in C++, there is no such top-most class and you can always define one as the root class to start the inheritance. ② Assume you have a grandparent class, a parent class and a child class, where each class has implemented a same function (method overriding). In C++, an instance of the child class can directly access the method in its grandparent class, by using the scope resolution operator (double colon "::"). However, in Java, an instance of child class can only visit the method in grandparent class via the parent class, by using "*super*" keyword (indirectly). ③ As for access specifier, the keyword "*protected*" in C++ and Java are different. In C++, protected members can be visited by methods in the same class or in its subclass (and friend functions considering the definition of friend); while in Java, protected members can also be visited by classes in the same package, even there is no inheritance relation between those two classes. ④ Unlike C++, Java doesn't provide inheritance specifier, so public and protected members of superclass in Java will remain public or protected in its subclass. ⑤ In C++, methods are non-virtual by default. If you want to achieve dynamic polymorphism, you must explicitly mark the method in the superclass as virtual, using keyword "*virtual*". But in Java, methods are virtual by default. So if you want to avoid dynamic polymorphism and method overriding, you must explicitly mark the method in the superclass as final, using "final" keyword. ⑥ In Java, interface and abstract class are two conceptions which can both achieve inheritance separately; While in C++, interface and abstract class are almost the same (or we can say interface is implemented using abstract class), that a class is made to be abstract when at least one function in the class is declared as pure virtual function (Statement: "virtual int name()=0;"). ⑦ C++ support multiple inheritance while Java class doesn't (Java interface support multiple inheritance).

Polymorphism is the ability of an object to take on multiple forms. So with polymorphism, developers can have functions with the same name, but different implementations. There are basically 2 types of polymorphism: static polymorphism and dynamic polymorphism. **Static Polymorphism/Static Binding/Static Dispatch/Early Binding/Compile Time Polymorphism/Static Linkage:** In Java, static polymorphism allows a class to have multiple methods that use the same name, but different signatures (method signature may indicate the number, types or sequences of parameters) with different implementations. This is achieved by **overloading** (Java doesn't support custom operator overloading while C++ does). The compiler will determine which method to be invoked at the compiling time. **Dynamic Polymorphism/Dynamic Dispatch/Late Binding/Runtime Polymorphism/Dynamic Linkage:** In Java, dynamic polymorphism can't be achieved by the compiler. Instead, the JVM will do that at runtime. It occurs mostly in inheritance with method overriding. Usually there are 3 cases: ① You declare a superclass variable and instantiate it with a superclass object, so the method in the superclass should be invoked; ② You declare a subclass variable and assign it with a subclass object, and so the method in the subclass should be invoked; ③ You declare a superclass variable and assign it with a subclass object, in this way, the method in the subclass should be invoked, and this is called **upcasting**. **Comparison:** Static binding is done during compile-

time while dynamic binding is done during runtime; private, final and static members use static binding and bound by compiler, while overridden methods are bound by JVM based on type of runtime object.
C++ achieves dynamic polymorphism with keyword "*virtual*". So by defining a virtual function in the superclass with the overriding version in a subclass, C++ will call the corresponding function based on the instance rather than its reference.

Method Overloading means you can have multiple methods with the same name, but different method signatures (that is, different in number, types or sequences of parameters). Overloading is related to static polymorphism. In Java, we can also overload static methods or the main function. But we can't overload methods if they only differ in the "*static*" keyword.

Overload vs Override: ① Methods overloading is about same method name with different method signatures in the same class; while method overriding is about same method name, same method signature but in different classes connected through inheritance. ② Overloading is related to static polymorphism while overriding is related to dynamic polymorphism.

Object Composition means combination of simple objects or data types into more complex ones. In this way, objects of one class may contain objects of other classes, which maintains a "has-a" relationship.
Composition over Inheritance/Composite Reuse Principle in OOP requires a class to achieve polymorphism and code reuse by object composition, that is, by containing instances of other classes to implement desired functionalities, rather than by inheritance from a superclass. So in order to achieve composite reuse principle, we should firstly create various interfaces representing different functionalities that an object should have. (We can also create various classes and include instances of those classes in the target class, but using interface allows the program to support a better polymorphic behavior.) Then build the class to implement multiple desired interfaces and finally we can realize all required behaviors of the class without inheritance. **Advantages:** CRP provides us more flexibility, simplicity in implementation and is closer to our real-world experience. Because CRP enables us to build a class based on its multiple components and maintain a "has-a" relationship; while inheritance requires us to find the commonness and differences between two classes, maintain an "is-a" relationship and finally build a large hierarchy tree, which is kind of rare. For example, a car may have several functionalities which are achieved by its several components, like the wheels are used to facilitate a car's movement. So we can say the car "has a" wheel, but it will be so weird to say a car "is a" wheel or a car is derived from a wheel. **Drawback:** CRP may require some additional programming effort since you have to implement all methods provided by individual component interfaces. But you don't need to re-implement method in subclass which has the same function as that in superclass.

Diamond Problem is an ambiguity occurs when two classes, B and C, both inherit from a common superclass A, and class D has multiple inheritance from both B and C. Although object from class D can invoke the right version of method between B and C using scope resolution (::), it's ambiguous for D object to invoke methods in the common superclass A. (Because C++ always follows each inheritance path separately, so there will be two copies of A objects within a single D object.) **Migration:** (Java class doesn't support multiple inheritance.) C++ resolves the diamond problem by virtual inheritance. That is, let both B and C inherit virtually from A (statement: "class B: virtual public A"). Therefore, there is only one object of class A in D's memory. (Java 8 introduces default methods on interfaces. If A, B, C and D are interfaces, where B and C each provide a different default implementation to an abstract method of A, which causing the diamond problem. Then, either interface D must re-implement that default method, or the ambiguity will be rejected as a compile error.)

Interface is like a blueprint of class, which enforces some behaviors a class has to do without telling how to do that. In Java, an interface can have variables and methods, but all methods should be by default public and abstract, and all variables should be public, static and final. A class that implements an interface must implement all methods in that interface, using the keyword "*implements*". The keyword "*interface*" is used to declare an interface. An interface can't be instantiate (we can't create instance of interface); a class can implement more than one interface; an interface can extend multiple interfaces.
Starting from Java 8, we can add default implementations for methods in interfaces, using "*default*" keyword. Such default implementations should be used when you have a class to implement that interface, and new methods are added to this class. So the method with default implementation in the interface can be invoked directly in the newly added method of the implementation class. The Java 8 also enables static methods so that static methods in the interface can be called independently in the implementation class without an object.

```java
// Interfaces can have methods from JDK 1.8
interface in1{
    final int a = 10;
    static void display(){
        System.out.println("hello");
    }
}
// A class that implements interface.
class Main implements in1{
    public static void main (String[] args){
        in1.display();
    }
}
```

Abstract method is a method that is declared, but contains no implementation. **Abstract Class** is a class that contains one or more abstract methods. Abstract class cannot be instantiate and requires subclasses to provide implementations for its abstract methods. So abstract class provides a way where classes can only be inherited but not instantiated. In Java, the keyword "*abstract*" is used to make a class or method abstract.

Abstract Class vs Interface are both used to hide the internal implementations of methods and only show the functionality. They are both used for abstraction. In Java, their differences are: ① Interfaces contain only abstract methods (before Java 8) and therefore methods are by default abstract without being declared with keyword. Abstract class can have both abstract and non-abstract methods, and so abstract methods must be declared with keyword "*abstract*". ② Variables in interfaces are by default static and final while in abstract classes they can be static or non-static, final or non-final. ③ Abstract class can implement one or multiple interface using keyword "implements", abstract class can extend only one class (either abstract or non-abstract, considering the Object class). An interface can extend multiple interfaces. ④ Members in an interface are public by default, while members in an abstract class can also be protected or private.
Usage: If there are some related classes that want to share some lines of code, then put the common code into the abstract class and let each class extend it. Or if you want to define non-static or non-final variables and let the method access and modify them, or require access modifiers other than public (such as private and protected), then use abstract class. Otherwise, if classes only share common properties but are totally different in implementations, (total abstraction of interface) or if you need multiple inheritance, then use interface.

Garbage Collection is a memory management technique in some programming languages so that developers don't need to worry about memory deallocation after creating new objects, because garbage collector will destroy those objects automatically. The main purpose of Garbage Collector is to track live object, free heap memory by destroying unreachable objects, while unreachable objects are objects that are no longer being referenced (haven't been assigned to any variable).
Advantages: Garbage collection is useful to prevent certain memory bugs, such as ① Double Free Bug where a memory is freed twice, especially considering that a memory can be re-assigned before its second deallocation; ② Memory Exhaustion/Leakage where the program fails to free unreachable memory and finally all available memory have been used up; ③ Dangling Pointer Bug which occurs when deallocating a memory without dereference its pointer, causing the reference variable point to a deallocated memory, or even worse if this previously freed memory is allocated again.
Disadvantages: ① Garbage collection may cause additional performance problems, such as consuming additional resource to determine which memory to free; (When garbage collection is running, all other threads have to pause) ② It's also unpredictable about when the garbage collection will run at the program background, which may cause recesses that especially unacceptable for real-time environments; ③ The garbage collector may not deallocate memory in the same way as you want, which may cause incompatible in memory arrangement.
There are several ways to make an object ready to be garbage collected: ① Make the reference variable point to null or re-assign it to another object, so its previously pointed object will be unreachable; ② Create an object inside a method, and once the method returns, all objects created inside that method will be unreachable; ③ Create an object without assigning it to a reference variable, as there is no way to refer to that object, it becomes unreachable; ④ All objects within the island of isolation are unreachable (**Island of Isolation** is a group of objects that reference each other but they are not referenced by any active object in the program. In fact, even a single unreferenced object is an island of isolation.)
Implementation in Java: Garbage collection in Java works in the way that tracking only live objects and then treating everything else as garbage. Since all objects are allocated on the heap, Java maintains several object trees, where each tree may have one or more root objects. So as long as the program can reach a root, the whole tree starting from that root are reachable. **GC Root (garbage collection root)** is a kind of special objects in Java that's always reachable, and therefore each object that has a GC root is reachable by the program. There are 4 kinds of GC roots in Java: ① Local variables (that are kept alive by the stack of a thread) in the main method; ② Active Java threads; ③ Static variables; ④ JNI References (Java objects that the native code has created as part of a JNI call, and objects thus created are treated specially since the JVM doesn't know if it's being referenced by the native code or not.)
To determine which object is no longer in use, the JVM intermittently runs a "**Mark-and-Sweep Algorithm**", which contains two steps: ① Mark → Traverse all object references, starting with the GC roots, and marks every object found as alive; ② Sweep → Clear all heap memory that is marked as unreachable (not occupied by a live object) and set it to be free.
Garbage collection is intended to remove the cause for classic memory leaks: unreachable-but-not-deleted objects in memory. However, this works only for memory leaks in the original sense. It's possible to have unused objects that are still reachable in the program if the developer forget to dereference them. Such objects can't be garbage-collected. Such a logical memory leak can't be detected by any software. Even the best analysis software can only highlight suspicious objects.

This keyword is used to refer to the current object. It can be used in the following ways: ① Use "this()" function to invoke the current class constructor (constructor call must be the first statement in the constructor); ② To invoke the current class method; ③ To indicate the current class variables (if there is a local variable in a method with the same name as an instance variable, then the local variable will hide the instance variable; or, you can use "*this*" to indicate the instance variable explicitly); ④ To return the current class instance; ⑤ To serve as a method parameter.

```java
// 1. Use this() to invoke the current class constructor
// Output: Parameterized Constructor Default Constructor
class Main{
    Main(){                    // Default constructor
        this(10, 20);
        System.out.print(" Default Constructor");
    }
    Main(int a, int b){        // Parameterized constructor
        System.out.print("Parameterized Constructor");
    }
    public static void main(String[] args){
        Main object = new Main();
    }
}
```

```java
// 2. Invoke current class methods
// Output: Inside Display
class Main{
    Main(){
        this.display();
    }
    void display(){
        System.out.println("Inside Display");
    }
    public static void main(String[] args){
        Main object = new Main();
    }
}
```

```java
// 3. Indicate current class variables
// Output: a=10
class Main{
    int a;
    Main(int a){this.a=a;}
    void display(){
        System.out.print("a="+this.a);
    }
    public static void main(String[] args){
        Main object = new Main(10);
        object.display();
    }
}
```

```java
// 4. Return the current class instance
// Output: a=10 b=20
class Main{
    int a, b;
    Main(){
        a=10;
        b=20;
    }
    Main get(){
        return this;
    }
    void display(){
        System.out.print("a="+a+" b="+b);
    }
    public static void main(String[] args)
        Main object = new Main();
        object.get().display();
    }
}
```

```java
// 5. To serve as a method parameter
// Output: a=10 b=20
class Main{
    int a, b;
    Main(){
        a=10;
        b=20;
    }
    void get(){
        display(this);
    }
    void display(Main obj){
        System.out.print("a="+obj.a+" b="+obj.b);
    }
    public static void main(String[] args){
        Main object = new Main();
        object.get();
    }
}
```

Super keyword is used to indicate the object of superclass, for the purpose of resolving ambiguity. It can be used in the following ways: ① If both superclass and subclass have same members (either data members or member functions), we can use "*super*" to indicate the superclass member; ② The "super()" function, either parametric or non-parametric, can be used to call the superclass constructor, and it must be the first statement in the subclass' constructor. (In fact, the program will always call the superclass constructor firstly before calling the constructor of subclass.)
Explicit constructor call should always be the first statement in a constructor.

Constructor Chaining is the process of calling one constructor from another constructor. It can be done in two ways: ① Within the same class, use "this()" keyword (either parametric or non-parametric); ② The subclass constructor will always invoke the constructor of its superclass, either explicitly (using keyword "*super()*") or implicitly, which forms a whole chain of constructor calls from the Object class down to the current subclass (superclass's constructor is always invoked before the subclass's constructor).

Static keyword is used for efficient memory management purpose. In Java, you can declare static blocks, variables, methods and nested classes with keyword "*static*". Once a member is declared static, it can be accessed before any instance of its class created, or without reference to any instance.
Static Block/Static Clause will be executed exactly once at the first time the class is loaded (either when you create the first

instance of the class or when you first access a static member of that class). The static block will also be executed before constructors. **Static Variable** will get memory only once. They are shared by all instances of that class. Static variables are essentially global variables. Similarly, **static method** within a class always belongs to the class rather than objects of that class. They can be called without creating an instance of that class. They can directly access and modify static variables, and call other static methods of the same class. They can never invoke a non-static method or use a non-static variable. During implementation, we should mark properties that are common to all objects as "*static*" and use static methods to access them. **Nested Class** is the class that being enclosed within another (outer) class. A class can be static only when it is a nested class, which is called **static nested class**.

```
// Static block, output is "Executed only once.0".
class Test{
    static int i;
    static{ System.out.print("Executed only once."); }
}
class Main{
    public static void main(String[] args){
        System.out.print(Test.i);  // Access variable i.
    }
}
```

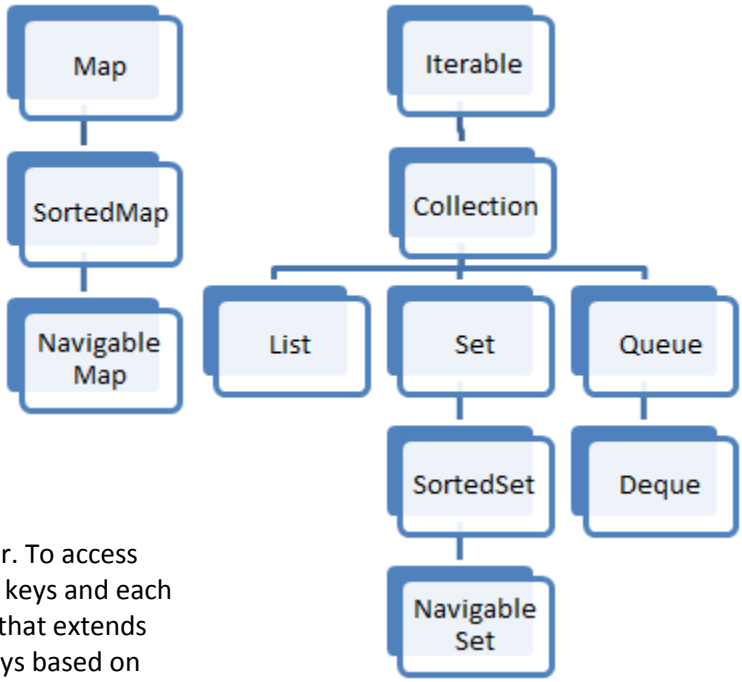
Static Main: The main function is always called by the JVM before any objects are made. So only by making it static can the main function be directly invoked via the class.
Static method is also called as **class method**, while non-static method can be called as **instance method**. In inheritance, class method corresponds to method hiding and instance method corresponds to method overriding. **Method Hiding** means subclass has defined a class method with the same signature as that in its superclass, and therefore we say that the superclass method is hidden by the subclass. So in this case, instead of determined by the object used to instantiate the reference variable, the version of method to be executed should be determined by the type of that reference variable. Data members/variables in a class are always determined by the type of reference variable, rather than the object.

Final keyword in Java can be used for variables, methods and classes, to prevent specific modifications. So a **final variable** means a constant, and you must initialize that final variable. If such final variable is a reference, this means that it can't be re-bound to another object (but the inner state of that object can still be changed). A final variable declared in a class can be initialized during declaration, or inside constructors or static block (If there are more than one constructor in a class, you must initialize that final variable in each constructor). A final variable declared in a method/constructor/block is called **local final variable**, and must be initialized once when it is created. A **final method** can't be overridden, which means we are required to follow the same implementation throughout all subclasses to prevent any unexpected behavior. In addition, considering that dynamic polymorphism relies on overriding, if a method is declared as final, a call to that method should be resolved at compiling time, which is static polymorphism. A **final class** should never be inherited or extended. Declare a class as final means all of its methods are final. A class can never be both abstract and final.
In C++, a variable declared with keyword "*const*" has the similar function as final variable in Java. But const variable in C++ must be assigned a value at the time it is declared, which is not necessary for final variables in Java. A final variable can be assigned a value later, but must be assigned only once.

Abstract keyword in Java can be used for classes and methods (but not variables) to achieve abstraction. So **abstract methods** will only have method declarations without method body. They are always put into the superclass and so it's the responsibility of its subclass to override a specific abstract method and provide the implementation**. Abstract classes** are classes that have at least one abstract method. We can never instantiate an abstract class, instead, we should use a subclass to extends an abstract class, and each subclass should either implement all abstract methods of that abstract class or be declared as abstract, too. Although abstract classes can't be instantiated, they can be used to create object reference, and let that reference point to one of its concrete subclass instance, as an approach to achieve dynamic polymorphism in Java. Keywords "*final*" and "*abstract*" can never be used together. For classes, final is used to prevent inheritance while abstract classes should depend on the subclasses to complete implementation. For methods, final is used to prevent overriding while abstract methods need to be overridden in subclasses.

Memory Allocation: (The heap is used for dynamic memory allocation. In C++, objects can be allocated on either stack or heap, where dynamic and global objects are allocated on heap and others on stack.) In Java, stack memory is responsible for holding references to heap objects and for storing primitive data types, and all objects are dynamically allocated on heap (when we declare a variable of a class type, only a reference is created and memory is not allocated. To allocate memory to an object, we must use "*new*". So the object memory is always allocated on heap.) Usually OS allocates heap memory in advance to be managed by the JVM while the program is running. So object creation is faster since global synchronization with the OS is not needed for every single object. An allocation simply claims some parts of a memory array and moves the offset pointer forward. The next allocation starts from this offset and claims the next parts of the array.

Collection is a group of individual objects treated as a single unit in Java. There are mainly two root interfaces designed for collection, which are Map interface (java.util.Map) and Collection interface (java.util.Collection).
Implementation Classes (only partial):
List {ArrayList, LinkedList, Stack, Vector};
Set {HashSet, LinkedHashSet, TreeSet};
SortedSet {TreeSet};
NavigableSet {TreeSet};
Queue {LinkedList, PriorityQueue};
Deque {LinkedList};
Map {HashMap, Hashtable, LinkedHashMap, TreeMap};
SortedMap {TreeMap};
NavigableMap {TreeMap}.



Map represents a mapping relationship within a key-value pair. To access the value you must know its key. And there can't be duplicate keys and each key can map to at most one value. **SortedMap** is an interface that extends the Map interface, which maintains some orders among its keys based on either the natural ordering or specific comparator.
HashMap is the class that provides basic implementations for Map interface. It is implemented based on hashing and maintains an array of buckets (the capacity of HashMap refers to the number of buckets) while each bucket is a linked list key-value pairs. HashMap can't maintain a constant order among elements over time. The get, put and containsKey operations in HashMap are basically O(1) and the worst case is O(n) (but it's not guaranteed since it may depends on how much time it takes to compute the hash value), and the time complexity of iterations over HashMap is usually proportional to the sum of its capacity and its number of key-value pairs. HashMap is unsynchronized which means it's not thread-safe.
LinkedHashMap is similar to HashMap but can maintain the initial insertion order among elements.
TreeMap is a class implements the SortedMap interface. It has functions of a Map while can also maintain a sorted order among its keys based on some comparator. It is implemented with a Red-Black Tree in the background. The operations like insert, remove and search (containsKey) take O(logn) complexity, and sorted traversal may take O(n) time because the ordering of keys has already been implemented during insertion, and what need to do is just traverse each key one by one.

List is an ordered collection in Java that can store duplicate values. It provides add, remove, get and set operations based on the numerical position of each element, together with search and other operations.
ArrayList is like a dynamic array in Java whose size can be increased or shrunk based on the total number of elements in the list. It also allows random access to elements. **LinkedList** is a linear data structure that is achieved by assigning some specific pointers for each node, and let pointers of each node point to both of its previous and next nodes in the list. **Vector** is very similar to ArrayList which can grow or shrunk as required and have random access to elements via various methods. However, Vector is synchronized which can be thread-safe. **Stack** is a Java collection with the restrictions that elements can be pushed/added only onto the top and popped/removed only from the top of the Stack, which known as LIFO property.

Array vs ArrayList: ① An array in Java has fixed size and we can create it in the way like simply declaring and initializing a variable; ArrayList is a class that implements the List interface, which will create an array with dynamic size. ② We can access elements in an array using a pair of square brackets, while we need various functions to access elements in an ArrayList. ③ Elements in Array can be either primitive data type or objects of a class, while ArrayList doesn't support primitive data type. ④ Array is faster in performance considering that ArrayList may need to resize during execution.
Vector vs ArrayList: ① Vector is synchronized which means only one thread can access to it at a time, and therefore indicates the thread-safe property; while ArrayList isn't. ② ArrayList is faster while Vector has a lower performance. This is because vector should achieve synchronization, which means if one thread works on Vector has acquired the lock, other threads have to wait till the lock is released. ③ Vector and ArrayList are different in increasing their capacity. The ArrayList will increase 50% of its current size once exceeds its capacity while the Vector will increase 100% (double its current size).
ArrayList vs LinkedList: ① ArrayList is a dynamic array while LinkedList is a doubly linked list. ② Insertions and removals are faster on LinkedList because for ArrayList, you may need to resize the array, copy its content to a new array and update indexes, which requires an O(n) time complexity. But for LinkedList, insertion and removal are O(1). ③ LinkedList has more

memory overhead because in ArrayList, you only need to store the data for each index, while in LinkedList, for each node you need to store both data and addresses that indicating its previous and next node.

Set is an unordered collection in Java that cannot store duplicate values. It provides basically add, remove, contains operations, etc. All the classes of Set are internally backed up by Map. **SortedSet** is an ordered collection that extends the Set interface, which maintains some orders among its elements. All elements in a SortedSet must be mutually comparable. **HashSet** is one class that implements the Set interface so duplicate values are not allowed either. The underlying data structure of HashSet is hashtable. HashSet stores elements based on their hashcode, rather than the order in which you insert them. The **load factor** of HashSet is a measure of how full it is allowed to get before its capacity is automatically increased. When the number of entries in the hash table exceeds the product of the load factor and the current capacity, the internal data structure is rebuilt (hash table is rehashed). **Implementation:** HashSet is internally implemented by HashMap. So the value we insert into a HashSet acts as a key of HashMap, and Java uses a constant variable as the value. That is, all keys have the same value. **Average time complexity** for add, remove and contains methods in HashSet is O(1). **LinkedHashSet** is an ordered version of HashSet that maintains a doubly-linked list among all elements, so the order of traversing a LinkedHashSet is predictable.
TreeSet is one class that implements the SortedSet interface and duplicate values are not allowed either. It maintains orders among elements based on keys or some specific comparators (rather than preserving the insertion order). It is implemented with a self-balanced binary search tree (like Red-Black Tree). So operations like add, remove and contains take O(log n) time. And traversal of TreeSet with n elements in sorted order takes O(n) time since ordering has already been implemented during insertion.

Queue is an ordered list of objects with the restrictions that objects must be inserted at the end of the queue and removed from the start of the queue, which is FIFO (First-In-First-Out) principle. A Queue interface can be implemented with both LinkedList and PriorityQueue class, where PriorityQueue will sort elements in the Queue based on priority.
PriorityQueue can process objects in the queue based on priority. Such priority may be based on their natural ordering, or some user-defined comparator. **Deque** is like a double-ended queue so that insertion and removal of elements are available at either end. It can be used as both a Queue and a Stack.

LinkedHashMap vs LinkedHashSet: ① LinkedHashMap has mappings of keys to values while LinkedHashSet simply stores a collection of elements with no duplicates. ② LinkedHashMap extends HashMap and LinkedHashSet extends HashSet.
HashMap vs HashTable: HashMap and HashTable are both used to store key/value pairs in a hash table. So we can further fetch a value by referring to its corresponding key. The hash code is computed for the key and stored in the table, and used as the index of the value that stored in the table. Their differences are: ① HashMap is non-synchronized. It's not thread-safe and can't be shared among many threads without proper synchronization code. But Hashtable is synchronized, it is thread-safe and can be shared with many threads; ② HashMap allows one null key and multiple null values but Hashtable doesn't allow any null key or value; ③ HashMap is more generally preferred if thread synchronization is not needed.
HashMap vs TreeMap: HashMap is an implementation of Map interface, which stores the key/value pairs in a Hashtable, while doesn't maintain any order based on the key or the value. The get/put/containsKey) operation are all O(1) time complexity in average case, but in fact, it all depends on how much time it takes to compute the hash value of the key. TreeMap is an implementation of SortedMap interface, which is the child of Map interface. In TreeMap, elements always maintain a sorted order. The add/remove/containsKey operations take O(logn) time complexity.
HashSet vs TreeSet: Neither HashSet nor TreeSet hold any duplicate elements, and they are both non-synchronized, which means they are not thread-safe and you must make explicit synchronization on them when a thread-safe operation is required. Their differences are: ① HashSet doesn't maintain any order among elements while elements in TreeSet are sorted in increasing order by default; ② HashSet gives better performance than TreeSet for the operations like add, remove, contains, size etc, where HashSet takes O(1) time and TreeSet takes O(logn) time complexity for such operations.

Red–Black Tree is a kind of self-balanced binary search tree that each tree node has an extra bit, which is often interpreted as color (red/black) of that node. The color bits are used to ensure the tree remains approximately balanced at insertions and deletions. So in addition to maintain properties of a binary search tree, a red-black tree should also have 5 properties: ① Each node is either red or black; ② All leaves (NIL) are black; ③ No two red nodes are adjacent (parent and children node of a red node must be black); ④ The root node must be black; ⑤ Every path from a given node down to a NIL/leaf node should contain the same number of black nodes. The time complexity of insert, delete and search operations are all O(logn) in average.

Error vs Exception: An error is a fatal problem or abnormal condition that the program should never try to catch it (since they should never be predicated to occur in a normal application). An exception is an unexpected event occurring at either compiling time or running time of the program. In Java, there is a class called Throwable (java.lang.Throwable), which is derived directly from the Object class. And the Throwable class has two subclasses, which are Error class (java.lang.Error) and Exception class (java.lang.Exception). All exceptions and errors in Java belong to these two classes. Errors in the Error class are always caused by the running environment, such as stack overflow or memory used up. And there is no way to recover it, so the execution will be terminated immediately. Exceptions in the Exception class are always caused by the program itself, and mostly they can be predicted, handled and recovered, and so the program can keep running as normal.
Checked vs Unchecked: In Java, all exceptions in both Exception class can be categorized into two types: checked and unchecked. **Checked exceptions** are those exceptions that are checked at the compiling time. In other words, they are noticed by the compiler, and therefore the compiler will force the programmer to either handle them with the try/catch block, or simply throw them and then deal with them in the caller. All classes derived from the Exception class, except the RuntimeException class, belong to the checked exceptions. **Unchecked exceptions** are those exceptions that are not checked at the compiling time, and will only occur during the running time. So if such exceptions are not handled properly, the program will always terminate immediately without raising a compiling error. All exceptions in the RuntimeException class are unchecked exceptions. All errors in the Error class are also considered as unchecked exceptions.

Control Flow in Try-Catch or Try-Catch-Finally: So if an exception is raised in the try block, then the rest code of that try block will not be executed and the control will be passed to a corresponding catch block. Here we assume this exception is handled in one catch block. So the corresponding catch block will be executed, and if there is also a finally block present, the code in the finally block will be executed. Then the program will go on its execution of the remaining code. Otherwise, if the exception occurred in the try block is not handled by any catch block, if there is finally block, the code in the finally block will be executed. Then this exception will be handled by default handling mechanism, usually it is stop execution and generate an error. If there is no exception occurred in the try block, the finally block, if present, will always be executed, and followed by the remaining part of the program.
Control Flow in Try-Finally: No matter whether there is an exception raised in the try block, the finally block will always be executed. The only difference is that, if an exception is raised in the try block, the exception will be handled by the default handling mechanism after the finally block; and if no exception occurred, the finally block will be followed by the remaining part of the program.

Throw vs Throws: The "throw" keyword is used to explicitly throw an exception from a method or a block. It can throw either checked or unchecked exceptions. In fact, it's usually used for custom exceptions. Once an exception is thrown, the nearest try-catch block will check to see whether there is a matching type of exception. And if no matching found, then the second nearest try-catch block will check and so on. The "throws" keyword is always write after the method signature, followed by a list of exceptions, to indicate that this method may throw one of listed exceptions and require the its caller to handle this exception in a try-catch block.

Compiled Language can be translated into native machine instructions of the target machine, which in turn be executed directly by the hardware. In other words, compiled language is implemented by compilers, which serve as translators to generate machine code from source code. C++ is compiled language. (For example, an addition "+" operation in C++ source code could be translated directly to the "ADD" instruction in machine code.) **Advantages:** ① It is faster, since the program is compiled only once and after that, you can run the native code directly without referring to the source code; ② There can be some error check (syntax and type) at the compilation stage. **Disadvantages:** ① It is not platform independent and always requires a compiler; ② May need some extra effort in programming.
Interpreted Language is that the original program should be executed directly without a previous compilation stage. An interpreter is needed for a step-by-step/line-by-line conversion and execution of source code. In this way, the operating system will never directly execute the program, it just runs the interpreter, and interpreter should execute the interpreted program. PHP is interpreted language. (For example, the "+" operation would be recognized by the interpreter at run time, and the interpreter will call its own "add" function and run it directly.) **Advantages:** ① It's platform free, the code can be easily executed elsewhere as long as there is an interpreter for that language; ② There can be more flexible in achieving dynamic typing and dynamic scoping. **Disadvantages:** ① It's running slower than compiled language; ② The source code is more readily to be viewed by others and even suffer from some security risks, such as code injection.
Intermediate: There is an intermediate stage between compiled and interpreted language, where the original program is firstly compiled into byte code (or some other representations), and then be executed by "interpreter" (for Java it is Java

virtual machine). The interpreter's execution on byte code is pretty similar to the hardware's execution on machine code, but the interpreter is still "software processed" language that on the top of hardware. Examples are Python and Java.

Difference between C++ and Java: There are too many differences between C++ and Java. ① C++ has pointers but Java doesn't. ② C++ is a compiled language while Java utilizes both bytecode compilation and interpreter (JVM). ③ Java has a special feature called garbage collection, which is a memory management technique that can automatically destroy unreachable objects in Java, to prevent memory leakage or data corruption. ④ C++ store its objects on both stack and heap, where static objects will be stored on the stack and global and dynamic objects may be stored on the heap; But Java will store all its static and dynamic objects on the heap. ⑤ Following are what Java doesn't support while C++ support (and can be even more): pointer, scope resolution operator (::), user-defined operator overloading, inline functions, friend functions, unsigned int data type, global variables, templates, etc.

Strengths of Java: ① Java is object-oriented which is easy to model and understand. ②Java is simpler and easier to use than C++. One main reason may be that Java is able to achieve memory allocation and deallocation automatically. ③Java is platform-independent and it's easy to move the program from one computer system to another, which is especially useful for the Internet. ④ Java provides many security considerations in designing. For example, Java adds runtime limitations for JVM, has a security manager so that untrusted code can be put into a sandbox, provides multiple APIs that related to security (like the standard cryptographic algorithms, authentication, and secure communication protocols). ⑤ Java offers cross-functionality and cross-platform so that Java programs can run on desktops, mobiles, embedded systems, etc. ⑥ Java has multiple IDEs that provide effective debugging, testing and other properties. ⑦ It's easy for Java to write network programs or program for cloud service.
Drawbacks of Java: ① The running speed of Java is not optimal, since it must be translated into byte code firstly at each runtime, and then be executed by its interpreter JVM. ② The memory management method in Java is a little expensive, considering that garbage collection technique may require additional time and resource consumption (When garbage collection program is running, all other threads have to be paused, and it may need additional memory space for garbage collection algorithm to determine which object to deallocate). ③ Java doesn't support templates, and therefore we can't pass multiple types to one single function at the same time (not good for code reuse).

JDK (Java Development Kit) is a software development environment used for developing Java applications and applets. It includes the Java Runtime Environment (JRE), an interpreter/loader (Java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc) and other tools needed in Java development.
JRE (Java Runtime Environment) may also be written as “Java RTE.” It provides the minimum requirements for executing a Java application, which consists of the Java Virtual Machine (JVM), core classes, and supporting files.
JVM (Java Virtual Machine) is a virtual machine that enables a computer to run a Java program. JVM has 3 notions: ① A specification, which is a document that formally describes what is required of a JVM implementation. ② An implementation, which is a computer program that meets the requirements of the JVM specification. (But implementation provider is independent to choose the algorithm. Its implementation has been provided by Sun and other companies.) ③ A runtime instance, which is an implementation running in a process that executes a program compiled into Java bytecode. Whenever you write java command on the command prompt to run the java class, an instance of JVM is created.
JDK=JRE+Development Tools, JRE=JVM+Library Classes

Why method overloading is not possible by changing the return type in Java? In C++ and Java, functions can't be overloaded if they differ only in the return type. The return type of functions is not a part of what is generated by the compiler for uniquely identifying each function. The number of arguments, type of arguments and sequence of arguments are the parameters which are used to generate the unique signature for each function. It is on the basis of these unique signatures that compiler can understand which function to invoke even if the names are same (overloading).

What is Blank Final Variable? A final variable in Java can be assigned a value only once, either in declaration or later, while a blank final variable in Java is a final variable that is not initialized during declaration.

Can we overload main() method? In fact, the main method is like any other method and can be overloaded in a similar way, and JVM always looks for the method signature to launch the program. The normal main method acts as an entry point for the JVM to start the program. So we can overload the main method in Java. But the program can execute the overloaded main method only after we call the overloaded main method from the actual main method only.

Wrapper Class in Java is the kind of classes that encapsulates primitive data types, and therefore primitive data types are able to be treated as objects. (**Usage**: ① The Collection frameworks in Java only accept object rather than primitive data type; ② An object of a wrapper class can achieve synchronization in multithreading.) Corresponding to the eight primitive data types in Java, the eight wrapper classes are: Boolean, Character, Byte, Short, Integer, Float, Double and Long.

HashMap Implementation: A HashMap in Java contains an array of buckets to store the key/value pairs, and uses the key's hashcode to determine where to place/find that key/value pair. That is, when you pass a key to a HashMap, it computes the key's hashcode. Then the HashMap calculates the bucket index based on that hashcode (using xor) to find the right bucket corresponding to the key's hashcode. In this way, the HashMap can quickly determine which bucket it should put or retrieve the pair. **Collision:** However, sometimes multiple hashcodes of keys may map to the same bucket, which causes a collision. The HashMap responds to a collision by constructing a Linkedlist among all conflict key-value pairs under that bucket. Then, it will query each entry of the Linkedlist, by comparing the values of keys, to find the right key/value pair. Three steps after calling of the get() method: ① Compute the key's hashcode and query the bucket index corresponding to that hashcode; ② Retrieve the whole list of key/value pairs under the bucket with the certain index; ③ Perform a sequential/tree-based search through each entry until a key that equals to the key passed into the get() method is found. (In fact, start from Java 8, when the number of entries in one bucket exceeds a certain threshold, the Linkedlist structure under that bucket will be reconstructed to be a balanced tree, which will improve the worst case performance from O(n) to O(log n).)

What happens when you type a URL into the web browser? ① Once the URL is parsed by the browser, the browser will firstly check the local cache for a DNS record of the corresponding IP address. (There may be 4 caches to check, which are the browser cache, the OS cache, the router cache and the ISP cache) ② If no record found in the cache, the computer will query your ISP's DNS servers, which will perform DNS queries on the behalf of yourself, and help you resolve the IP address of that URL. ③ Once the browser receives the correct IP address, it will start to build connections with that server matching to this IP address to transfer information. Usually the browser firstly initiates a TCP connection with three-way handshaking (SYN-SYN/ACK-ACK). After the TCP connection is established, the browser will send a HTTP request to the server (usually it's a GET request to fetch the webpage, but may also be the POST request to submit a form), together with some additional information about the browser. ④ The server can then receive the request from the browser, pass the request to a request handler to read and generate a response. The response is assembled in a particular format, which contains response status, code, compression type, any cookies and also the webpage content if you requested. ⑤ The browser will wait for the server's response. Once receiving, the browser will firstly parse the response status/code. If the response code starts with 2, which indicates a success, then the browser will render and display the HTML content extracted from the message body; If the response code starts with 3, then the browser may need to be redirected to the provided address; If the response code starts with 4 or 5, then the browser needs to parse the error response and display it to the users.

String in Java is a class representing objects that contains an immutable sequence of unicode characters. (In C/C++, string is simply an array of chars.) Unlike an ordinary Java class, **Java String is special in:** ① String is associated with string literal in the form of double-quoted texts (such as "Hello World!"). And you can either call the constructor to create a String instance (explicit construction), or simply assign a string literal directly to a String variable-just like a primitive data type (implicit construction); ② The '+' operator is overloaded to concatenate two String operands (String is the only class that has the '+' operator overloaded and '+' is the only operator that is internally overloaded to support string concatenate); ③ String is immutable, that its content can't be modified. For functions called by a String object, such as toUpperCase(), a new String object should be constructed and returned instead of modifying the original one. **Reason:** Strings receive special treatment in Java since they are used frequently in program, and therefore, efficiency (in terms of computation and storage) is crucial. (Instead of making everything an object, Java designers decided to preserve primitive types in the language to improve the programming performance. Primitives are stored in the call stack, which require less storage spaces and are cheaper to manipulate, while objects are stored in the heap, which require complex memory management and more storage spaces.) **String Literal vs String Object:** String literals are stored in a common pool. Multiple String variables can share common storage of strings in the pool as long as they hold the same contents (and this is why String is immutable); String objects allocated via "new" keyword are stored in the heap, and there is no sharing of storage for the same contents. The equal() method in the String class is used to compare only the content of two Strings, while the "==" (relational equality operator) is used to compare the reference/pointer of two strings. **StringBuffer and StringBuilder** are both classes that help to build mutable strings, and are more efficient than String objects if you need to modify them frequently (String is more efficient if there is no need to modify). Their only difference is: StringBuffer is synchronized while StringBuilder is not synchronized in multithread programming. In sing-thread program, StringBuilder, without synchronization overhead, is more efficient.

Divide and Conquer is an algorithm design paradigm that you recursively break down a large problem into several, smaller sub-problems, until each sub-problem becomes simple enough to be solved directly. Then you combine the solution of each sub-problem together so as to give the solution of the original problem.
Quick Sort is a kind of Divide and Conquer sorting algorithm which has time complexity of O(nlogn). The steps are: ① Pick an element, called a pivot, from the array; ② Partition and reorder the array, so that elements that are less than the pivot come before the pivot, while elements that are greater than the pivot come after the pivot, and in this way, the pivot will be in its final position, with two subarrays before and after it; ③ Recursively apply the above steps to the two subarrays until all elements are sorted. The pivot can be picked in mainly four ways: ① Pick the first element as the pivot; ② Pick the last element as the pivot; ③ Pick a random element as the pivot; ④ Pick the median as the pivot.
Merge Sort is a kind of Divide and Conquer sorting algorithm which has time complexity of O(nlogn). The steps are: ① Recursively divide an unsorted array into several subarrays, until each subarray contains only 1 element. (And this single element is considered to be sorted); ② Then you repeatedly merge two sorted subarrays to product one new sorted subarray, until there is only one subarray remaining, which is the sorted array we want.

Insertion Sort is done by iteration, which has the time complexity of $O(n^2)$. So in each iteration, you fetch only one element from the array and find its position among those already sorted elements. In this way, after you find the right position of the last element, the whole array is sorted.
Bubble Sort takes $O(n^2)$ time complexity, where you repeatedly traverse through the array, compare each pair of adjacent elements. If the two adjacent elements are in a wrong order, then simply swap them and go to the next pair.
Selection Sort takes $O(n^2)$ time complexity. The algorithm will divide the unsorted array into two parts, where the first part is sorted and the second part is unsorted. Initially the sorted part is empty while the unsorted part should hold the entire array. Then each time it will choose the smallest element among the all unsorted elements and swap it with the left-most elements in the unsorted part, which can also be considered as the last element in the sorted part.

Heap Sort is like the selection sort that divides the input array into two parts, a sorted part (the heap part) and an unsorted part, and it iteratively shrinks the unsorted part by extracting the smallest element and moving that to the sorted part. However, the heap sort utilizes a heap data structure rather than a linear structure to find the min/max element, and it place elements with a layout of a complete binary tree. In the binary tree, each node corresponds to one element in the array. For a 1-based array (index start from 1), if the parent node corresponds to the index i, then its left child node corresponds to the index (2*i) and its right child node corresponds to the index (2*i+1).
A sorted array is created by repeatedly removing the root node of the heap and inserting it into the array. The heap is updated after each removal to maintain the heap property. Once all objects have been removed from the heap, the resultant array is sorted. A min heap will always put the min node on the root, so the array is sorted in ascending order; while a max heap will always put the max node on the root and therefore the array is sorted in descending order.
The heap sort algorithm (**HEAPSORT**) involves preparing the array by first turning it into a max/min heap (**BUILD-HEAP**), and then repeatedly swaps the first value of the list with the last value, decreasing the number of values considered in the heap operation by one, and shifting the new first value into its position in the heap to maintain the heap property (**HEAPIFY**). This repeats until the range of considered values is one value in length.
The **BUILD-MAX/MIN-HEAP** procedure, which runs in linear time, produces a max/min heap from an unordered input array. It is implemented in the way that calling the HEAPIFY in a bottom-up manner from the middle (array.length/2) node down to the first node.
The **MAX/MIN-HEAPIFY** procedure, which runs in O(logn) time, is the key to maintaining the max/min heap's property. So assuming bot the left subtree and right subtree are already heaps, but the current node violates the heap's property. Then the HEAPIFY procedure should be implemented to make the current node move down to either its left subtree or right subtree until find its right position.
The **HEAPSORT** procedure, which runs in O(nlogn) time, sorts an array in place.
The **MAX/MIN-HEAP-INSERT**, **HEAP-EXTRACT-MAX/MIN**, **HEAP-INCREASE-KEY**, and **HEAP-MAXIMUM/MINIMUM** procedures, which run in O(logn) time, allow the heap data structure to implement a priority queue.

Bucket Sort/Bin Sort works by distributing the elements of an array into a number of buckets. Each non-empty bucket is sorted individually, either using a different sorting algorithm or applying the bucket sort algorithm recursively. Finally, visit all buckets in order and gather all elements back into the original array. In average, the time complexity of bucket sort is O(n+k) where n is the length of array and k is the number of buckets. However, the worst performance can also be $O(n^2)$ especially when elements in the array cannot be uniformly distributed among all buckets.

Nested Class is used to logically indicate that the nested class is a member of its outer/enclosing class and is useful only to its outer class, which may be helpful in developing more readable code and increasing the encapsulation level (considering that nested class can always directly access to the private members of its outer class). There are two types of nested class, which are non-static nested class/inner class and static nested class. **Comparison:** ① Static and non-static nested classes can both directly access to public, protected and private members (variables and functions) of their outer class, but static nested class can only directly access to static members while non-static nested class can directly access to both static and non-static members. ② A static nested class is almost behaviorally the same as any other top-level classes that it invokes (non-static) members of its outer class by firstly creating an instance of its outer class and then using that instance to invoke (public, protected and private) members of its outer class. ③ Non-static nested class can never define static members within it, while static nested class can declare both static and non-static members within it. (In fact, static members can only be declared within a top-level class or within a static nested class. Even a static method cannot define static variables within itself.) The following shows how to create objects of nested classes and how to call their methods.

```
public class OuterClass {
    static class Staticnested{
        public void inStatic(){    // Declare a non-static method
            System.out.println("Non-static method in the static nested class");
        }
        public static void staticInStatic(){    // Declare a static method
            System.out.println("Static method in static nested class");
        }
    }
    class Nonstaticnested{
        public void inNonStatic(){    // Can only declare non-static method
            System.out.println("Non-static method in the inner class");
        }
    }
    public static void main(String[] args){
        Staticnested staticObject=new Staticnested();
        // Invoke non-static method with object of static nested class
        staticObject.inStatic();
        // Invoke static method with or without object of static nested class
        staticObject.staticInStatic();
        Staticnested.staticInStatic();
        // Create an object of non-static nested class and invoke its method
        OuterClass outerObject=new OuterClass();
        Nonstaticnested nonStaticObject=outerObject.new Nonstaticnested();
        nonStaticObject.inNonStatic();
    }
}
```

```
class OuterClass {
    static class Staticnested{
        public void inStatic(){    // Declare a non-static method
            System.out.println("Non-static method in the static nested class");
        }
        public static void staticInStatic(){    // Declare a static method
            System.out.println("Static method in static nested class");
        }
    }
    class Nonstaticnested{
        public void inNonStatic(){    // Can only declare non-static method
            System.out.println("Non-static method in the inner class");
        }
    }
}
public class OtherClass {
    public static void main(String[] args){
        OuterClass.Staticnested staticObject=new OuterClass.Staticnested();
        // Invoke non-static method with object of static nested class
        staticObject.inStatic();
        // Invoke static method with or without object of static nested class
        staticObject.staticInStatic();
        OuterClass.Staticnested.staticInStatic();
        // Create an object of non-static nested class and invoke its method
        OuterClass outerObject=new OuterClass();
        OuterClass.Nonstaticnested nonStaticObject=outerObject.new Nonstaticnested();
        nonStaticObject.inNonStatic();
    }
}
```

equals() function is used to check whether two instances are the same. By default (that is, in the Object class), an instance is "equal" to another one if and only if they both refer to the same object in memory. Usually, different classes may have different implementations. For example, in the String class, the equals() function will return true if two strings hold the same value, even though they point to totally different memory location; and instead, the relational equality operator "==" is used to compare whether two strings have exactly the same reference in memory. In addition, if you assign the same string literal to two String objects, it will also return true even though the relational equality operator "==" is used to compare the two objects, because same string literal will always share a common memory in the common pool. Results are the same of the equals() function and the relational equality operator "==" in the Integer class.

```
String sa="abc", sb="abc";
String sc=new String("abc"), sd=new String("abc");
System.out.println(sa.equals(sb));  // True
System.out.println(sa==sb);         // True
System.out.println(sc.equals(sd));  // True
System.out.println(sc==sd);         // False
Integer ia=1, ib=1;
Integer ic=new Integer(1), id=new Integer(1);
System.out.println(ia.equals(ib));  // True
System.out.println(ia==ib);         // True
System.out.println(ic.equals(id));  // True
System.out.println(ic==id);         // False
```

ConcurrentMap<K, V> in Java is an interface that derived from the Map interface, and **ConcurrentHashMap<K, V>** is a class that implements the ConcurrentMap interface; **SynchronizedMap<K, V>** is a private static nested class in the Collections class and **synchronizedMap(Map<K, V> m)** is a static method provided by the Collections class, which will return a Map object with thread-safe property. **Comparison:** ① Both ConcurrentMap/ConcurrentHashMap and SynchronizedMap/Collections.synchronizedMap(), together with Hashtable can achieve thread-safety on map object. ② Hashtable is an old implementation which is introduced from JDK1.1. Each method in the Hashtable class will lock the entire map using the "*synchronized*" keyword, which makes it very slow especially when there is a large number of threads. It is highly discouraged because of such a scalability issue. ③ SynchronizedMap is similar to Hashtable that each of its method will acquire locks on the entire map. The inner implementation of SynchronizedMap is simply enclosing all codes of each method in the Map interface with the "synchronized" keyword; Collections.synchronizedMap() will convert a thread-unsafe Map object to be a SynchronizedMap object. They are both introduced in JDK 1.5. ④ ConcurrentMap/ConcurrentHashMap is optimized to be more scalable to hold more threads and has better performance. This is because it will only lock partial of

```
Map<Integer, Integer> map=new HashMap<>();
Map<Integer, Integer> synmap=Collections.synchronizedMap(map);
```

the map. In addition, only its write methods need to acquire locks while multiple threads can always read from the same segment concurrently. (The inner implementation is achieved by dividing the entire map into multiple segments, and then only locking the corresponding segment where the new key-value pair should be placed. By default there are 16 segments, so the default concurrency level is 16, but you can modify it in initialization.)

Comparable and Comparator are both interfaces in Java used to compare and sort objects. The Comparable interface is in java.lang package while the Comparator is in java.util package. They both need classes to implement them and need the implementation classes to override their compare methods. **Differences:** ① The compare method in Comparable is "int compareTo(Object o)", which takes only one parameter and compare that with the current object; the compare method in Comparator is "int compare(Object o1, Object o2)", which takes two parameters and compare them with each other. ② A class which implements the Comparable interface can only sort in a single way, while a class implements the Comparator interface can sort in multiple ways. For example, a Movie class is needed with attributes like rating and release year. To implement the Comparable interface, provide your implementation of the compareTo method, which will compare Movie objects based on their ratings. Then in the driver class, create a Collections object with the Movie type, after calling the "Collections.sort()" method with single parameter, which is the Collections object, all Movie objects in the collections will be sorted based on their ratings. However, if you want to sort in two ways based on ratings and release years, then Comparator should be used here. So firstly create two classes to implement the Comparator interface, where the first class implements the compare method based on ratings and the second one compares based on release years. In the driver class, call the "Collections.sort()" method with two parameters, one is the Collections object and the other one is a Comparator object based on your custom Comparator class. In this way, object collections can be sorted in multiple ways as you want.

ExecutorService is an interface in Java which allows you to pass tasks to be executed by threads asynchronously. It creates and maintains a reusable pool of threads to execute submitted tasks. It also manages a queue, which is used when there are more tasks than the number of threads in the pool, so tasks can be arranged in the queue temporarily until there are free threads available to execute the task. **Callable** is an interface in Java which encapsulates a task which can run on a thread (similar to the Runnable interface). **Differences between Callable and Runnable:** ① The Runnable interface is a functional interface and has a single run() method which doesn't accept any parameters and doesn't return any value; while the Callable interface is a generic interface containing a single call() method which needs to return an object based on corresponding object type. ② The run() method in the Runnable interface don't need to throw any exception, while the call() method in the Callable interface must throw an exception. ③ A task/class implementing the Runnable interface can be used to directly create a Thread object; while the Callable interface can never be used with the Thread class directly to create a Thread object. **Future** is an interface in Java which is designed to hold the result of an asynchronous computation. Methods are provided in the Future interface to check if a computation is done, to wait for its completion, to retrieve the result of the computation or to cancel a task. The exception returned by the Callable interface will also be collected by a Future object. The **FutureTask** is a concrete class that implements both Runnable and Future interface.

CPU Processor Core Thread

Process is an abstraction or an instance of a running program, which includes some current variable values of this program. By switching the CPU from process to process, it creates an illusion of (pseudo) parallelism.

Thread is like a mini-process, that multiple threads can exist in only one process and they will share some resource of this process while running independently. We need threads since: ① Many programs consist of multiple activities that can run simultaneously. However, as processes can't share memory, threads can be used here; ② Threads are faster and easier to create and destroy than a process. **Per-Process Items:** address space, global variables, open files, child processes, pending alarms, signals and signal handlers, accounting information, etc.; **Per-Thread Items:** program counter, registers, stack, state.
Multithreading is a feature that allows concurrent execution of two or more parts of a program, in order to maximize the utilization of CPU. Each part of such program is called a thread. So, threads are light-weight processes within a process.
In Java, threads can be created by: ① extending the Thread class; ② implementing the Runnable Interface.

Create a new class to extend the Thread (java.lang.Thread) class and override the run() method available in the Thread class. A thread begins its lifecycle inside the run() method. Then an object of this subclass can be created and call start() method to start the execution of a thread. Start() invokes the run() method on the Thread object.	Implement the Runnable (java.lang.Runnable) interface and the run() method in it. Then instantiate a *Thread* object and call start() method on this object. (Thread class can provide some built-in methods like start(), interrupt(), yield(), etc. which help us achieve some basic functionality of a thread, but they are not available in Runnable interface.)

```java
// Java code for thread creation by extending the Thread class
class MultithreadingDemo extends Thread{
    public void run(){
        try{
            // Displaying the thread that is running
            System.out.println("Thread "+Thread.currentThread().getId()+" is running");
        }catch(Exception e){
            System.out.println ("Exception is caught");    // Throwing an exception
        }
    }
}
// Main Class
public class Main{
    public static void main(String[] args){
        for (int i=0; i<8; i++){
            MultithreadingDemo object = new MultithreadingDemo();
            object.start();
        }
    }
}
```

```java
// Java code for thread creation by implementing the Runnable Interface
class MultithreadingDemo implements Runnable{
    public void run(){
        try{
            // Displaying the thread that is running
            System.out.println("Thread "+Thread.currentThread().getId()+" is running");
        }catch (Exception e){
            System.out.println("Exception is caught");  // Throwing an exception
        }
    }
}
// Main Class
class Main{
    public static void main(String[] args){
        for (int i=0; i<8; i++){
            Thread object = new Thread(new MultithreadingDemo());
            object.start();
        }
    }
}
```

* The Runnable interface has only one method which is run().

Notice that a thread can only be started by calling the start() function (rather than calling the run() function), and the start() function will invoke the inner run() method. This is because the start() function is to create a separate call stack for the thread (Stack memory in Java is allocated per thread while heap memory is allocated for each running JVM process and shared by all threads in that process). After a separate call stack is created by it, JVM will call run() based on that stack. (What happens when a function is called: ① Arguments are evaluated; ② A new stack frame is pushed into the call stack; ③ Parameters are initialized; ④ Method body is executed; ⑤ Value is returned and current stack frame is popped out from the call stack.) There are six thread states in Java: New, Runnable, Blocked, Waiting, Timed Waiting, Terminated.
Main Thread is the one that begins automatically and immediately when a Java program starts up. It is also the thread from which other children threads generated and must be the last thread to finish execution by performing various shutdown actions. For each program, a Main thread is created by JVM. The Main thread will firstly verify the existence of the main() method, and then initialize the class. To control the Main thread we must obtain a reference to it, which can be done by calling the method currentThread( ) provided in Thread class. This method returns a reference to the thread on which it is called. "*Thread t=Thread.currentThread();*"

Singleton class is a class that has only one instance of the class at any time. That is, no matter how many times you instantiate the singleton class, there's only one instance and each reference variable will point to that instance. Singletons can control access to resources, such as database connections or sockets. To design a singleton class, there are three requirements: ① Make constructor private, since we need the private constructor to prevent any additional instantiations; ② Need a static method that has a return type of the class type, so that the singleton class can use that method, which is public and static, to create instance of the class and return it. ③ Need a private static variable to hold that instance.

```java
class Singleton{
    // Static variable with Singleton type to hold instance
    private static Singleton instance = null;
    // Private constructor to prevent any other instantiation
    private Singleton(){ }
    // Static method to create and maintain only one instance
    public static Singleton getInstance() {
        if (instance == null) instance = new Singleton();
        return instance;
    }
}
```

Synchronization: Java uses synchronized methods or blocks to make sure only one thread can access the resource at a given point of time. Synchronized methods/blocks are marked with the "*synchronized*" keyword, to be synchronized on some object. All synchronized methods and blocks synchronized on the same object can only have one thread executing inside them at a time. All other threads attempting to enter the synchronized method/block are blocked until the thread inside the synchronized block exits the block. The synchronization is implemented in Java with a concept of "monitors". Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor. Synchronization is only needed when object is mutable. If shared object is immutable or all the threads which share the same object are only reading the object's state while not modifying it, then there is no need to synchronize it.

```java
// A Java program to demonstrate working of synchronized.
class Sender{    // A Class used to send a message
    public void send(String msg){
        System.out.println("Sending "+msg);
        try{
            Thread.sleep(1000);
        }catch (Exception e){
            System.out.println("Thread Interrupted.");
        }
        System.out.println(msg+" Sent");
    }
}
class ThreadSend extends Thread{ // Send messages using Threads
    private String msg; Sender  sender;
    // A Sender object and a string message to be sent
    ThreadSend(String m, Sender obj){msg = m; sender = obj;}
    public void run(){
        // Only one thread can send a message at a time.
        synchronized(sender){sender.send(msg);}
    }
}
class Main{          // Driver class
    public static void main(String args[]){
        Sender snd = new Sender();
        ThreadSend S1=new ThreadSend( "Hi",snd );
        ThreadSend S2=new ThreadSend( "Bye",snd );
        S1.start(); S2.start(); // Start two threads of ThreadSend
        try{     // Wait for threads to send
            S1.join(); S2.join();
        }catch(Exception e){
            System.out.println("Interrupted");
        }
    }
}
```

```java
class Sender{
    public synchronized void send(String msg){
        System.out.println("Sending "+msg);
        try{
            Thread.sleep(1000);
        }catch (Exception e){
            System.out.println("Interrupted.");
        }
        System.out.println(msg+" Sent");
    }
}
```

```java
class Sender{
    public void send(String msg){
        synchronized(this){
            System.out.println("Sending "+msg);
            try{
                Thread.sleep(1000);
            }catch (Exception e){
                System.out.println("Interrupted.");
            }
            System.out.println(msg+" Sent");
        }
    }
}
```

yield() method in Java is a way to prevent execution of a thread, indicating that the thread is *not doing anything particularly important* and if any other threads or processes need to run, they can run. Otherwise, the current thread will continue to run. The yield method gives hint to the thread scheduler that it is ready to pause its execution. If any thread executes yield() method , thread scheduler checks if there is any thread with same or higher priority than this thread. If processor finds any thread with higher or same priority then it will move the current thread to Ready/Runnable state and give processor to other thread. Otherwise, current thread will keep executing. Notice that: ① If many threads with same priority are waiting for the processor, it can't specify which thread will get execution chance first; ② The thread which executes the yield() method will enter in the Runnable state from Running state; ③ Once a thread pauses its execution, we can't specify when it will get chance again, which totally depends on thread scheduler; ④ Underlying platform must provide support for preemptive scheduling if using yield method.
sleep() method is another way to prevent execution of a thread, causing the current thread sleep for a specified number of milliseconds. So we can make a thread to be in sleeping state for a specified period of time. In addition, it causes the thread to definitely stop executing for a given amount of time; if no other thread or process needs to run, the CPU will be idle.
join() method makes the current thread to wait until another thread finishes its execution. If join() is called on a thread instance, the currently running thread will block until that thread instance has finished executing. For example, if an executing thread t1 calls join() on t2 (t2.join();), then t1 will immediately enter into waiting state until t2 completes its execution. And after t2 ends, t1 can continue to run.

Lifecycle of Thread: ① **New:** When a thread is created, it's in the new state. The thread in the new state is ready to start running. It can turn into the runnable state by calling the "start()" function on it; ② **Runnable:** A thread in the runnable state can be either actually running or ready to run. It's the responsibility of the thread scheduler to give the thread a timeslot to run. Usually, a multithread program allocates a fixed amount of time to each individual thread. Each thread will run for a short while, then pause and give up the CPU to another thread which is in the Runnable state, so other threads which are ready to run can get a chance to run; ③ **Blocked/Waiting**: If a thread is temporarily inactive, it may be either blocked or waiting. A thread is in the blocked state if it's trying to access protected/synchronized sections of code which is currently locked by another thread. When such part of code is unlocked, the scheduler will pick one of the threads which are blocked by that section and move it to the runnable state. In programming, a thread will falls into the blocked state after calling the "Object.wait()" function. A thread is in the waiting state if it's waiting for another thread on a condition. A waiting state occurs when calling "Object.wait()", "Thread.join()" or "LockSupport.park()" functions. Only when the condition is satisfied, the scheduler is notified and the waiting thread is moved to a runnable state. If a currently running thread is moved to a blocked/waiting state, another thread in the runnable state will be scheduled by the thread scheduler to run. It's the responsibility of the scheduler to determine which thread to run; ④ **Timed Waiting:** A thread in timed waiting state when it calls a method with a timeout parameter, and it will be in this state for a certain time period unless it is notified. The methods that will cause a timed waiting state include "Object.wait()", "Thread.sleep()", "Thread.join()", "LockSupport.parkNanos()" and "LockSupport.parkUntil()"; ⑤ **Terminated:** A thread can terminate for two reasons. Firstly the execution is fully completed, that code of the thread has been entirely executed. Second it's because of some unusual erroneous, like segmentation fault or unhandled exceptions. A terminated thread will never consume any cycles of CPU.

Critical Region is a part of program where shared resource is accessed. Access to shared resource simultaneously can lead to unexpected or erroneous behavior, so parts of the program where the shared resource is accessed must be protected.
Race Condition in programming is a situation when multiple processes/threads enter the same shared region and the output is totally dependent on the execution sequence of these processes or threads, which is uncontrollable.
Mutual Exclusion is a property which can help to prevent race conditions. It requires that one running thread/process can never enter the critical region when another concurrent thread/process is running inside it, and therefore ensure that simultaneous updates to the critical region would never occur. For multi-core system, it must ensure that only one processor can access the critical region memory while the instruction is executed, which might be done with help of some hardware mechanism, to make sure that each CPU can use the bus exclusively to access the memory. In Java multithreading, mutual exclusion can be achieved by locks/synchronization/monitors, inter-thread communication, semaphores, etc.

Polling is the process that testing a condition repeatedly until it becomes true. It's usually implemented with help of loops to check whether a particular condition is true or not, which will waste many CPU cycles and makes the implementation inefficient. For example, it may occur in a queue problem where one thread is producing data and another is consuming it.
Inter-Thread Communication can be used to avoid polling in multithread problem. In Java, there are three methods: ① **"wait()"** tells the calling thread to give up the lock and go to sleep until some other thread on the same object enters the monitor and notifies it ; ② **"notify()"** will wake up one single thread that called wait() on the same object. Note that calling notify() doesn't actually release a lock on resource, so in order to get the desired result, it is advised to use notify only at the end of a method; ③ **"notifyAll()"** will wake up all threads that called wait() on the same object. All three methods are final and belong to the Object class, so all classes in Java have them. They must be used only within a synchronized block. Notice that both wait() and sleep() will make the current thread paused, but wait() is from the Object class and is invoked with an object, while sleep() is a static method in Thread class and should be invoked with a thread. Also, wait() will release the lock and is used for inter-thread communication, while sleep() will not release a lock and is used to pause an execution.

Thread Pool: Server programs such as database and web servers sometimes need to repeatedly handle requests from multiple clients, to process a large number of short tasks. One way to build a server application is to create a new thread each time a request arrives and process this request in the newly created thread. However, this method has significant disadvantages, since the server that creates a new thread for every request would spend more time and consume more system resources in creating and destroying threads than processing actual requests. In Java, a JVM creating too many threads at the same time can cause the system run out of memory. So it's better if we can limit the number of threads being created, and this can be resolved with help of Thread Pool. A thread pool can reuse previously created threads to execute current tasks, and therefore, offers a solution to the problem of thread cycle overhead and resource thrashing. As the thread is already existed when the request arrives, the delay introduced by thread creation is eliminated, making the application more responsive. **Implementation:** A thread pool in Java can be implemented with help of the Executor

interface, its subinterface, ExecutorService, and the implementing class, ThreadPoolExecutor. Using such an executor framework, developers only need to implement the Runnable objects and send them to the executor to execute. As for a fixed-size thread pool, if all threads are currently running in the pool, then the pending tasks are placed in a queue and can be executed when a thread becomes idle. **Risks:** ① Deadlock: While deadlock can occur in any multi-thread program, thread pools introduce another case of deadlock, where all the running threads are waiting for the results from the blocked threads which are waiting in the queue due to a lack of threads for execution; ② Thread Leakage: Thread leakage occurs if a thread is removed from the pool to execute a task but not returned to it when the task completed. For example, if the thread throws an exception and the pool class doesn't catch this exception, then the thread will simply exit, reducing the size of the thread pool by one. If this repeats many times, then the pool would eventually become empty and no threads would be available to execute other requests; ③ Resource Thrashing: If the size of the thread pool is very large then time is wasted in context switching between threads. Having more threads than the optimal may cause a starvation problem that leading to resource thrashing. **Attention:** ① Don't put tasks that concurrently wait for results from other tasks, as it can lead to a deadlock; ② Be careful while using threads for a long lived operation, as it may cause the thread waiting forever and would eventually lead to resource leakage; ③ The thread pool has to be ended explicitly at the end, otherwise the program will go on executing and never end. Call shutdown() on the pool to end the executor. If another task is sent to the executor after shutdown, it will throw a RejectedExecutionException.

Deadlock is a state where each process/thread in a group is blocked and waiting for resource which is held by some other waiting processes/threads. The deadlock condition may occur in a Java multithread program when there is synchronizations on resource. It's usually caused when one thread locks on some resource while also tries to acquire some other resource which is outside of the lock region. **Coffman Conditions** describe all requirements that must be satisfied simultaneously to cause a deadlock: ① Mutual Exclusion: The resources involved must be unsharable, only one process/thread can use them at any time; ② Hold and Wait: A process/thread must be currently holding at least one resource and requesting additional resources which are being held by other processes/threads; ③ No Preemption: A resource can be released only voluntarily by the process/thread holding it; ④ Circular Wait: Each process/thread must be waiting for a resource which is being held by another process/thread, which in turn is waiting for another process/thread to release the resource. **Prevention** of a deadlock can be achieved by preventing one of four Coffman conditions: ① Remove the mutual exclusion condition so that no process/thread can have exclusive access to resource, which is always impossible in reality (may cause race condition); ② Remove hold and wait condition by requiring processes/threads to acquire all resources they need before start a certain set of operations. However, it's difficult to satisfy, and may cause an inefficient usage of resource. Another way is to request resources only when a process/thread has none. But it's also impractical because resources may be allocated and remain unused for a long time. And a process/thread requiring a popular resource may have to wait indefinitely as such resource may always be allocated to other process/thread, causing a resource starvation; ③ Non preemption condition is difficult to avoid as a process/thread has to be able to have a resource for a certain time, otherwise the result may be inconsistent or thrashing may occur. However, inability to enforce preemption may interfere with a priority algorithm; ④ Approaches to avoid circular waits include avoid nested locks (which is the main reason for deadlock), avoid unnecessary locks, use the join() function (only in Java) or create a certain order on locks so that they are acquired only in a specific sequence.

Atomic operation is an operation that can't be subdivided into smaller parts, in order to prevent any interruptions before the operation is done. Implementation methods include TSL (test and set lock, it should be implemented in hardware), XCHG (similar with TSL but available on Intel processors), semaphores, etc. **Semaphore** is like a counter which indicates the number of available resource for a specific thread, in order to help control the thread access to the resource. Using semaphore also guarantees that actions of both checking resource availability and processing resource can be done in a single, atomic operation, and therefore, can safely handle resource to prevent race conditions. Generally, when using semaphore, a thread which wants to enter the shared resource should be granted permission from it. **Steps:** ① If the semaphore indicates a count which is not zero, then the thread can acquire a permission, cause the count be decremented by 1; ② Otherwise, if the count is zero, then the thread will be blocked until a permission can be acquired; ③ If the thread no longer needs to access to the resource, it will release the permission, causing the count of semaphore being increased by 1. Then other threads can be granted with the permission.

Daemon thread is an utmost low priority thread that runs in background to perform tasks such as garbage collection. **Property:** ① They can't prevent the JVM from exiting when all the user threads finish their execution; ② JVM terminates itself when all user threads finish their execution. If JVM finds running daemon threads, it will terminate the daemon thread and then shutdown itself; ③ JVM doesn't care whether Daemon thread is running or not.