**Java Interview Questions**

► **What is Java?**

The Java® programming language is a general-purpose, high-level, strongly and statically typed, concurrent, class-based, object-oriented and functional [Java 8] language. It is normally compiled to the bytecode instruction set defined in the JVM Specification. New **Java SE 8** features - lambda expressions, method references, and functional interfaces ("first-class functions") - offer a programming model that combines **object-oriented** and **functional** styles.

► **What were the new features added in Java 5?**

**JDK 5.0** (2004) (Java language 3rd edition) added or updated features:

**generics**, **annotations**, **autoboxing** / unboxing, **enum** types, **foreach** loops, variable arity methods, static imports, complete revision of Java memory model, **StringBuilder** class.

► **What were the new features added in Java 6?**

Java SE 6 (2006) mostly enhanced some features:

Collections framework, internationalization, Java I/O, reflection, JVM monitoring and management API, support for Internationalized Domain Names.

► **What were the new features added in Java 7?**

Java SE 7 (2011) added or updated features:

binary literals, underscores in numeric literals, allowed **strings in switches**, the **try-with-resources** (AutoCloseable, Closeable Interface), **catching multiple exception** types and rethrowing exceptions with improved type checking; concurrency Utilities - the **Fork/Join** framework; enhancements in Java I/O; type inference for generic instance creation; JVM Garbage-First collector; the SunJSSE provider supports **TLS 1.2.**

►► **What were the new features added in Java 8 (that you used)?**

**Java SE 8** (**2014**, end of public updates – Jan **2019**) is a major release since Java 5 with updates in language, compiler, libraries, tools, runtime (JVM):

* **lambda** expressions,
* **functional interfaces**,
* **method references**,
* **Streaming API** (java.util.stream.Stream);
* **arrays with parallel operations**;
* default and static methods in **interfaces**;
* repeating annotations;
* better type inference;
* extended annotations support;
* variable arity methods,
* **receiver parameters**;
* **java.util.Optional<T>** class;
* **Date/Time API** (JSR 310);
* Nashorn JavaScript engine;
* **Base64** encoding;
* class dependency analyzer: jdeps;
* the PermGen replaced with **Metaspace** (JEP 122);
* the **JDBC-ODBC bridge removed**.

► **What were new features added in Java 9 (JSR 379)?**

**Java SE 9** (**2017**, end of public updates –Mar **2018**) is a major release which is already obsolete (**end of support**):

* A new kind of Java component introduced - the **module** (set of packages containing types). Java SE itself is built as 26 modules with dependencies between them.
* More concise **try-with-resources** statements.
* **Private interface methods** are allowed so that non-abstract methods of an interface could share code.
* can use diamond syntax in conjunction with anonymous inner classes.

► **What were new features added in Java 10 (JSR 383)?**

**Java SE 10** (Mar **2018**, end of public updates – **Sep 2018**) is a minor release with few changes:

* **Local-Variable type** inference to declarations of local variables with initializers:

**var** url = new URL("http://www.oracle.com/");

* Several new APIs added to facilitate creation of **unmodifiable collections**.
* Optional.orElseThrow() added (synonymous with existing get() method).
* Added support for the TLS session.
* Improve docker container detection and resource configuration usage.
* Some deprecated APIs removed.
* The policytool security tool has been removed from the JDK.

# Java Language Basics

►► **Explain integer division and remainder, e.g. what are results of 13/2, -13/2, 13%2, -13%2?**

Integer division (both operands are integers) rounds toward 0: 13/2 = 6; -13/2 = -6

The remainder operation for integer operands produces a result such that (a/b)\*b + (a%b) = a.

13%2 = 1; -13%2 = -1

►►► **How many objects will be created? Why Java uses the concept of string literal? What is output?**

String s1 = "Welcome";

String s2 = "Welcome";

String s3 = s2;

String s4 = new String("Welcome");

System.out.println( s1==s2 );

System.out.println( s3==s1 );

System.out.println( s4==s2 );

One string object in **constant pool,** two **heap** references to that constant, and one **heap** reference to the new String. Java uses string constant pool to make memory usage more efficient, strings are immutable and reused if already exist in string constant pool. The output is: **true, true, false**.

►►► **What are wrapper classes?**

Each of Java's eight primitive data types has a **class** dedicated to it:

**Boolean, Byte, Character, Double, Float, Integer, Long, Short.**

These are known as **wrapper** classes because they "wrap" the primitive data type into an object of that class. The wrapper classes are part of the **java.lang** package, which is imported by default into all Java programs. All of the wrapper classes are **immutable.** The wrapper classes have few purposes:

* To ‘wrap’ primitive values in an object so that we can use them as objects, e.g. added to ArrayList, HashSet, HashMap etc. collection.
* To provide an assortment of utility functions for primitives like converting primitives to and from string, converting to various bases like binary, octal or hexadecimal, or comparing.
* Allow **null** value which could mean “undefined”, or using with [Java 8] Optional: **Optional<Byte> byte**.

**►►► Do you know class Byte? How many bits in byte? What will be printed out by example?**

The **Byte** is immutable wrapper class for primitive type **byte**. An object of Byte contains a single byte field. It also provides several methods for converting a byte to a String, short, int, long, as well as few constants. Byte has **8 bits** with values [-128 .. +127].

►► **What is immutable object? Examples of Java immutable and mutable classes? Difference between immutable and final?**

An object is considered **immutable** if its **state** (composite **value**) cannot change after it is constructed. Immutable objects are particularly useful in concurrent applications. Since they cannot change state, they cannot be corrupted by thread interference or observed in an inconsistent state. When you have reference to an immutable object, the contents of that instance can’t be altered without changing the reference.

All of the java.lang package wrapper classes are **immutable**: Boolean, Byte, Character, Double, Float, Integer, Long, Short, String. Examples of **mutable** classes are java.util.Date, java.math.BigInteger, BigDecimal. E.g. we can change only a year of a Date.

The **final** keyword means that you can't change the object's **reference** to point to another reference. For primitives, it means that value can’t be changed.

►►► **How can we make Date [BigInteger, Person] class member immutable?**

Make an immutable wrapper class and put a private member of type Date in your class. A strategy / pattern for defining classes of immutable objects is:

* Make all fields **final** and **private**.
* Don't provide **setter** methods — methods that modify fields or objects referred to by fields.
* Don't allow subclasses to **override** methods. The simplest way to do this is to **declare the class as final**. A more sophisticated approach is to make the constructor **private** and construct instances in factory methods.
* If the instance fields include references to mutable objects, don't allow those objects to be changed:
  + Don't provide methods that modify the mutable objects.
  + Don't share references to the mutable objects, e.g. in **getters**. Never store references to external, mutable objects passed to the constructor; if necessary, create (deep) **copies**, and store references to the copies. Similarly, create copies of your internal mutable objects when necessary to avoid returning the originals.

**[Java 8]** provides new Time API with all its core classes immutable and thread-safe, with methods called with and return new objects, rather than using setters, e.g. **java.time.LocalDate**, LocalTime, LocalDateTime.

►►► **Why should we override equals()?**

The equals method is used to check whether two objects are same or not. It needs to be overridden if we want to check the objects based on their properties, e.g. patient id or SSN, or patient name, DOB, etc.

►►► **Can you implement equals without hashCode? Could two different objects have the same hash code?**

The hashCode is supported for the benefit of **hash tables** such as those provided by HashMap. Normally we would implement either just equals (e.g. if object is not used as a hash key) or **both** methods.

The hashCode is allowed to return the same value for different objects (while returning different hash codes may improve the performance of hash tables). So, if we have hash collision with two different objects in the bucket having the same hash, equals will be used to find matching object. Thus, if we implemented hashCode we should also properly implement equals. Implementing equals puts restraint back on hashCode - if two objects are equal or the same object, then calling hashCode method on each of the two objects must produce the same integer.

► **What is the initial value of an object reference which is defined as an instance variable?**

The object references are all initialized to null in Java.

► **What is constructor?**

A constructor is used in the creation of an instance of a class. It looks just like a method declaration that has no result. Constructors are not members, never inherited and therefore not subject to hiding or overriding.

► **What is the purpose of default constructor?**

The default constructor provides the default values to the objects. The java compiler creates a default constructor only if there is no constructor in the class.

► **Does constructor return any value?**

Yes, that is current instance, while you can’t use return type yet it returns a value.

►►► **Can you make constructor final, or static, or abstract?**

No. constructor **can't be final** because constructor is not inherited and can't be overridden. Constructor is called implicitly when the **new** keyword is used so it can't lack a body and thus **can’t be abstract**. Static method belongs to the class, not to an object but the constructor is implicitly called to initialize an object, so there is no purpose in having a static constructor.

**►►► What happens in JVM memory when you create new instance?**

When a new instance is created, e.g. A a = **new** A(…); space is allocated for the new class instance, normally in **heap** memory (but registers could also be used). If there is insufficient space to allocate the object, an OutOfMemoryError is thrown. The new object contains new instances of all the fields declared in the specified class type and all its superclasses. As each new field instance is created, it is initialized to its default value. The result is a reference to the newly created object of the specified class. Every time the expression is evaluated, a fresh object is created.

► **What is static variable?**

Static variable is used to refer the common property of all objects (that is not unique for each object) e.g. company name of employees, college name of students, etc. Static variable gets memory only once in class area at the time of class loading.

► **What is static method?**

A static method belongs to the class rather than object of a class. A static method can be invoked without the need for creating an instance of a class. Static method can access static data member and can change the value of it.

► **What is static block?**

Is used to initialize the static data member. It is executed before main method at the time of class loading.

► **Can we execute a program without main() method?**

Yes, one of the way is static block.

► **Are there pointers in Java?**

Pointer is a variable that refers to the memory address. They are not used in java because they are unsafe and complex to understand.

► **Can you use both this() and super() in a constructor?**

No. Because super() or this() must be the first statement.

► **What is object cloning?**

The object cloning is used to create the exact copy of an object.

► **What is shallow copy?**

Object clone with only top level members copied.

► **Can I import the same package/class twice?**

One can import the same package or same class multiple times. Neither compiler nor JVM complains about it. But the JVM will internally load the class only once no matter how many times you import the same class.

► **What is static import?**

By using static import, we can access the static members of a class directly, without need to qualify it with class name.

► **What are parameters pass by value and pass by reference?**

Passing parameter by value means passing a copy of the original value. Any changes to that value will not affect original value, i.e. no side-effects. Passing parameter by reference means passing reference, pointer to, or address of the original value - primitive, composite, or object, which would allow "side-effects" to change original object. One of the early issues was passing **large arrays** of data. To avoid creating a copy of the array (i.e. pass by value) each time procedure is called, a reference to that array was passed instead.

An example from **ALGOL-68** explicitly specifies which parameters are passed by reference, including primitives:

proc p = (int a, **ref** int b, [] int c, **ref** [] real d, proc (real) real f) real: ...

Language **Pascal** had three kinds of parameters: value parameters (passed "by value"), variable parameters (passed "by reference"), and procedure or function parameters. E.g.

procedure s( x: integer, **var** y: integer, function f: real);

►►► **What type of parameter passing does Java support? Are objects passed by value or by reference?**

Most answers on Internet argue that "in Java the arguments are **always passed by value**", meaning that Java passes both primitive values as well as references [to objects] by value.

None of Java Language specifications mentions passing parameters "by value" or "by reference". Parameters passing is explained as "when the method is invoked, the values of the actual argument expressions initialize newly created parameter variables, each of the declared type, before execution of the body of the method." For primitives, it means that parameters are passed "by value". For reference types, it means that copy of the reference is created, still pointing to original object. It allows to change referenced object. In classical meaning, it means that objects are passed by reference. To have objects passed by value, we would need to create a "**deep clone**" of the object, and pass reference to that clone.

► **[Java 8] What is variable arity method? How ellipsis could be used?**

If the last formal parameter is a variable arity parameter, the method is a variable arity method. Otherwise, it is a fixed arity method. Variable arity parameter is special **last** formal parameter of a method or constructor, indicated by an ellipsis following the type. Note that the ellipsis (...) is a token unto itself. It is possible to put whitespace between it and the type, but this is discouraged as a matter of style.

public static void main(String... args)

► **[Java 8] What is “receiver” parameter?**

The receiver parameter is an optional syntax for an **instance** method or an inner class's constructor which represents

the object for which the method is invoked. For an inner class's constructor, the receiver parameter represents the immediately enclosing instance of the newly constructed object. Either way, the receiver parameter exists solely to allow the **type** of the represented object to be denoted in source code, so that the type may be annotated.

The receiver parameter is not a formal parameter; more precisely, it is not a declaration of any kind of variable, it is never bound to any value passed as an argument in a method invocation expression or qualified class instance creation expression, and it has no effect whatsoever at run time. Examples:

void m(Test this) {} // receiver parameter in an instance method

class A {

A(Test Test.this) {} // receiver represents Test immedialtely enclosing instance of A

void m(Test.A.B this) {} // instance of B for which B.m() is invoked.

► **What is Optional? How/When to use it?**

[Java 8] introduced a new class called java.util.**Optional**<T>, inspired by Haskell and Scala. It is a single-value container class that encapsulates an optional value. It makes it clear whether a given value is allowed to be missing and could be used to rewrite typical null-check patterns. Its purpose is to help design more-comprehensible APIs so that by just reading the signature of a method, you can tell whether you can expect an optional value.

public class Soundcard {

private **Optional<USB>** usb;

public **Optional<USB>** getUSB() { ... }

}

Optional<Soundcard> sc = Optional.of(soundcard); // throws NullPointerException if value is null

sc.ifPresent(System.out::println);

► **What are the different scopes for [declarations of] variables?**

The **scope of a declaration** is the region of the program within which the entity declared by the declaration (e.g. variable) can be referred to using a simple name, provided it is visible. Declarations of variables can have one of the scopes:

* instance variables - these are object level variables, initialized to default values at the time of object creation.
* local variables - defined within a method or block, remain accessbile only during method excecution. The scope of a local variable declaration in a block is the **rest** of the block, starting with its own initializer.
* formal parameters of a method, constructor, or lambda expression have as a scope the entire body of the method, constructor, or lambda expression.
* static variables - class level variables, initialized when the class is loaded in JVM and remain there as long as the class remains loaded. The scope of a class variable includes the **entire body** of the class.

► **What is shadowing?**

Some declarations may be **shadowed** in part of their scope by another declaration of the same name, in which case a simple name cannot be used to refer to the declared entity. Declaration of a formal parameter, local variable, or local class may be **shadowed** in a class declaration nested within a method, constructor, or lambda expression; and the declaration of an exception parameter may be shadowed inside a class declaration nested within the Block of the catch clause. E.g. shadowing of a field declaration by a local variable declaration:

class Test {

static int x = 1;

public static void main(String[] args) {

int x = 0; ...

}

► **What is obscuring?**

A simple name may occur in contexts where it may potentially be interpreted as the name of a variable, a type, or a package. In these situations, the rules specify that a variable will be chosen in preference to a type, and that a type will

be chosen in preference to a package. Thus, it is may sometimes be impossible to refer to a visible type or package declaration via its simple name. Such a declaration is **obscured**. Naming conventions help reduce obscuring.

Obscuring involving class and interface type names is rare. Names of fields, parameters, and local variables normally do not obscure type names because they conventionally begin with a lowercase letter whereas type names conventionally begin with an uppercase letter. **Method** names cannot obscure or be obscured by other names. Obscuring involving **field** names is rare.

► **What is hiding?**

If the class declares a **field** with a certain name, then the declaration of that field is said to **hide** any and all accessible declarations of fields with the same name in super-classes, and super-interfaces of the class. In this respect, hiding of fields differs from hiding of **methods**, for there is no distinction drawn between static and non-static fields in field hiding whereas a distinction is drawn between static and non-static methods in method hiding. It is permissible for an instance variable to hide a static variable. A hidden field can be accessed by using a **qualified name** if it is static, or by using a field access expression that contains the keyword **super** or a cast to a superclass type.

An overridden (i.e. hidden) **instance method** can be accessed by using a method invocation expression that contains the keyword **super**. A qualified name or a cast to a superclass type is not effective in attempting to access an overridden method. In this respect, overriding of methods differs from hiding of fields.

► **What is the difference between shadowing, hiding, and obscuring?**

Name conflicts may arise unintentionally as types developed by different programmers or different organizations evolve. For example, types, methods, and fields may have the same name. Shadowing is distinct from hiding, which applies only to members which would otherwise be inherited but are not because of a declaration in a subclass. Shadowing is also distinct from obscuring. Method names cannot obscure or be obscured by other names.

►► **What is Reference class and subclasses?**

Java.lang.ref.**Reference** is abstract base class for reference objects. Its subclasses are:

* PhantomReference,
* SoftReference,
* WeakReference.

► **What is PhantomReference?**

Phantom reference objects, which are enqueued after the garbage collector determines that their referents may otherwise be reclaimed. Phantom references are most often used for scheduling **pre-mortem cleanup** actions in a more flexible way than is possible with the Java finalization mechanism. Unlike soft and weak references, phantom references are not automatically cleared by the garbage collector as they are enqueued.

►► **What is SoftReference?**

Soft reference objects are cleared at the discretion of the garbage collector in response to memory demand. As long as memory is still available, soft references will not be garbage collected. Soft references are most often used to implement **memory-sensitive caches**. All soft references to softly-reachable objects are guaranteed to have been cleared before the Java VM throws an OutOfMemoryError. **UV** uses SoftReferences to cache resources.

public class ResourceCache<K, R extends Serializable> {

protected ConcurrentHashMap<K, SoftReference<CachedEntry<R>>> resourceCache;

►►► **What is WeakReference? WeakHashMap?**

WeakReference is a Reference which does not prevent the referenced object from being finalized and reclaimed by a garbage collector, unlike a normal (strong) reference. If garbage collector determines that an object is **weakly reachable**, it will atomically clear all weak references to that object and all weak references to any other weakly reachable objects from which that object is reachable.

Weak references are cleared aggressively and are not used for caching. It could be used to associate some extra information with some objects that you can’t extend, e.g. to implement **canonicalizing mappings** by the java.util.**WeakHashMap**. You can add key-value pairs to a WeakHashMap via the put() method. But the **key** objects are held via weak references that are associated with a reference queue. If the garbage collector determines that a key object is weakly reachable, it will clear and enqueue any weak reference objects that refer to the key.

# OOAD / OOP

►► **What is Inheritance?**

Inheritance is a mechanism in which one class could acquire properties and behavior of another class. It represents "IS-A" relationship, e.g. “Patient is [subclass of] Person”. It allows code **reusability** and method **overriding**.

► **Is there multiple inheritance in Java?**

To reduce the complexity and simplify the language, multiple inheritance is not supported in java in case of class but is supported for interfaces.

► **Which class is the superclass for every class?**

Object class.

►►► **What are methods of Object?**

Class Object is the root of the class hierarchy. Every class has Object as a superclass.

All objects, including arrays, implement the methods of this class:

clone(), equals(object), finalize(), getClass(), hashCode(), toString(),

notify(), notifyAll(), wait(), wait(timeout), wait(timeout,nanos)

► **Is constructor inherited?**

No, constructor is not inherited.

►►► **What is composition? Do we have composition in Java?**

It depends on the definition of composition. **Composition in UML** (proper UML name is "composite aggregation") is a **"strong" form of aggregation**, a **whole/part** relationship - a part could be included in at most one composite (whole) at a time, and if a composite is deleted, all of its parts are "normally" also deleted. Primitive fields are usually not considered as “parts”.

Because Java uses object references, composition in Java is “**weak**” and is in fact “[**shared] aggregation**”, i.e. part is independent of the composite, and when composite is deleted, part could survive garbage collection if there is another reference to.

►► **What is [runtime] Polymorphism?**

Polymorphism is the ability for different objects to respond differently to the same message. We can define several methods with the same signature in inheritance hierarchy of classes. [Runtime] Polymorphism (aka dynamic method dispatch) is a process in which a call to an overridden method is resolved at runtime rather than at compile time. In this process, an overridden method is called through the reference variable of a super class.

► **Can we achieve runtime polymorphism by data members?**

No. Only methods are polymorphic.

► **What is the difference between static binding and dynamic binding?**

In case of static binding (static polymorphism) type of object or specific method is determined at compile time whereas in dynamic binding (dynamic polymorphism) type of object or specific method is determined at runtime.

► **What is method overloading?**

If a class have multiple methods by same name but different parameters / signature.

► **Is it possible to overload method by changing the return type?**

No, return type is not part of signature, and because of ambiguity.

► **Can we overload main() method?**

Yes, we can have several main() methods with different signature in a class.

► **What is method overriding?**

When a subclass provides a specific implementation of a method that is already provided by its parent class. It is used for runtime polymorphism and to provide more specific implementation of the method. In Java we can explicitly annotate method as overridden (explicitly "virtual"). This annotation requires Java compiler to verify that signature is correct and the method is in fact overridden.

**@Override** public void prepare(Vector<Object> row, String[] columnNames)

► **Can we override static method?**

No, you can't override the static method because they are the part of class not object.

► **Can we override overloaded method?**

Yes.

► **How to prevent method from overriding?**

Final methods can't be overridden.

► **What is the difference between overridden and overloaded methods?**

Method overloading occurs within a class, while overriding – between class and subclass. Overloaded methods have different signature. Method overriding provides more specific implementation of the method.

► **Are there virtual functions in Java?**

All methods in Java are “virtual” by default.

► **What is covariant return type?**

Since Java 5, it is possible to override any method by changing the return type if the return type of the subclass overriding method is subclass type.

► **What is final class?**

Final class can't be inherited from.

**►►► What is encapsulation?**

Encapsulation is one of the loosely defined OOAD concepts / development techniques with different interpretations. It was mentioned in Barbara Liskov's 1977 article describing abstraction mechanisms in programming language CLU in the context of **hiding details of implementation**. Encapsulation includes:

* creating new data types (classes) by combining both information (**structure**) and **behaviors**, and
* hiding details / restricting access to implementation details.

**►►► What is abstraction?**

There is no single commonly accepted definition of abstraction in OOAD. For example, abstract data types (ADT) were introduced in 197x in CLU with construct called a "cluster". Classic example of ADT in CLU is **stack** with push() and pop() operations. CLU also introduced separation of **abstraction** from its **implementation**(s):

*“An abstraction isolates* ***use*** *from* ***implementation****: an abstraction can be used without knowledge of its implementation and implemented without knowledge of its use.”*

Abstraction lets you focus on what the object does instead of how it does it. Separate Java interface from concrete implementation class.

► **What is an abstract method?**

An abstract method is a method that is declared without an implementation (without braces, and followed by a semicolon), like this:

abstract void moveTo(double deltaX, double deltaY);

► **Can there be an abstract method without abstract class?**

If a **class** includes abstract method(s), then the class itself must be declared abstract. Method in an **interface** that is not declared as default or static is implicitly abstract.

► **Can a method be both abstract and final?**

No, abstract method needs to be overridden whereas you can't override final method.

► **Is it possible to instantiate the abstract class?**

No, abstract class can never be instantiated.

► **What is a JavaBean?**

Reusable software component written in Java, designed to be manipulated visually by a software development environment, like JBuilder or VisualAge for Java.

# Classes and Interfaces

► **What are public, private, protected, default modifiers?**

The Java provides mechanisms for **access control**, to prevent the users of a package or class from depending on unnecessary details of the implementation of that package or class. If access is permitted, then the accessed

entity is said to be accessible.

* **Public** class is visible in other packages, field is visible everywhere (class must be public too).
* **Protected** is available to all classes in the **same package** and to **all subclasses** (even in a different package) of the class that owns the protected feature. Use for subclasses.
* **Package** access member is available to all classes in the **same package.** More restrictive for subclasses in different packages. No access modifier implicitly (by default) means packageaccess.
* A **private** class member or constructor is accessible only within the body of the class that encloses the declaration. It is “**not inherited”** by subclasses.

►►► **What is static / nested class?**

A nested class is any class whose declaration occurs within the body of another class or interface. Nested classes are divided into two categories: **static** and **non-static**. Nested classes that are declared **static** are simply called (static) nested classes. Non-static nested classes are called **inner** classes. There are several types of nested classes:

* (static) nested class,
* inner member class,
* inner [non member] local class,
* inner [non member] anonymous class.

► **What is member type (class, interface)?**

A member class is a nested class whose declaration is directly enclosed in the body of another class or interface declaration. Member class may be **static**, in which case it has no access to the instance variables of the surrounding class; or it may be **inner** class. A member interface is a nested interface whose declaration is directly enclosed in the body of another class or interface. Member type (class or interface) could be inherited from superclass or superinterface, or member type from superclass or interface could be hidden if member type is declared with the same name.

► **What is inner class?**

An inner class is a nested class that is **not** (explicitly or implicitly) **static**. An inner class may be:

* a non-static member class,
* a local class, or
* an anonymous class.

Objects that are instances of an inner class exist within an instance of the outer class.

class OuterClass {

...

class InnerClass {

...

}

}

An instance of InnerClass can exist only within an instance of OuterClass and has direct access to the methods and fields of its enclosing instance.

► **Can interface have inner class?**

No, a member class of an interface is implicitly static so is never considered to be an inner class.

► **Difference between nested classes and inner classes?**

Inner classes are non-static nested classes.

► **What is local class?**

A local class is an inner class that is:

* not a member of any class (local class declaration is immediately contained by a block), and
* has a name.

Local class declaration statements may be intermixed freely with other kinds of statements in the block.

Example of local class declared in method:

void print() {

int i;

**class Line {**

**String s;**

**}**

Line l;

}

► **What is anonymous class?**

An anonymous class is an inner class that is:

* not a member of any class (local class declaration is immediately contained by a block), and
* has no name.

An anonymous class is never abstract, always implicitly final, always an inner class, i.e. it is never static.

An anonymous class is implicitly declared - declaration is automatically derived from a class instance creation expression. An anonymous class cannot have an explicitly declared constructor.

interface Colorable {

void setColor(byte r, byte g, byte b);

}

Colorable c = new Colorable() { public void setColor(...){...}; };

Declares an anonymous class that implements the Colorable interface. Note that an expression such as new Colorable() is not valid because it is not possible to create an instance of an interface, only of a class.

► **What is the difference between local and anonymous class?**

Both local and anonymous classes are inner non member classes.

Local class has a name, while anonymous has no name.

► **Can we access non-final local variable inside the local inner class?**

No, local variable must be constant if you want to access it in local inner class.

► **What is interface?**

An interface declaration introduces a new reference type (a blueprint of a class) whose members are:

* classes,
* interfaces,
* constants (fields),
* abstract, default [Java 8], or static [Java 8] methods.

Every interface is implicitly abstract. It can be used to achieve full abstraction and multiple inheritance. There are two kinds of interfaces:

* normal interfaces,
* annotation types.

► **Can we have static method in interface?**

Static methods were not allowed in interfaces until [Java 8]. In [Java 8] an interface can declare static methods, which are invoked without reference to a particular object. An interface does not inherit static methods from its superinterfaces.

► **What is nested interface?**

Any interface declared inside another interface or class. It is static by default.

► **What is marker interface?**

An interface that has no data members and no methods. For example, Serializable, Cloneable etc.

►►► **What is difference between abstract class and interface?**

An abstract class can have non abstract (concrete) public, protected, and private methods. Interface have abstract and default [Java 8] methods. An abstract class can have static methods. Starting with [Java 8] an interface can also declare static methods. With interfaces all methods that you declare or define (as default methods) are public. An abstract class can have constructor. An abstract class can have non static instance variables, final or not. With interfaces, all fields are automatically public, static, and final. You can extend only one class (single inheritance), whether or not it is abstract, whereas you can implement any number of interfaces (multiple inheritance)

► **Which should you use, abstract classes or interfaces?**

Consider using **abstract classes** if any of these statements apply to your situation:

* You want to share code among several closely related classes.
* You expect that classes that extend your abstract class have many common methods or fields, or require access modifiers other than public (such as protected and private).
* You want to declare non-static or non-final fields. This enables you to define methods that can access and modify the state of the object to which they belong.

Consider using **interfaces** if any of these statements apply to your situation:

* You expect that unrelated classes would implement your interface. For example, the interfaces Comparable and Cloneable are implemented by many unrelated classes.
* You want to specify the behavior of a particular data type, but not concerned about who implements its behavior.
* You want to take advantage of multiple inheritance of type.

►► **Can we define private or protected modifiers for variables or methods in interfaces?**

Until [Java 9] they are only (implicitly) public. Private and protected access in interfaces is allowed in [ava 9].

►►► **[Java 8] What is default (interface) method?**

A default method is a method that is declared in an interface with the "**default**"modifier; its body is always represented by a block. It provides a default implementation for any class that implements the interface without overriding the method. Default methods are implicitly **public** as other interface methods. Default methods are distinct from concrete methods, which are declared in classes and are **not part of interface contract**, so a "functional interface" could have several default methods.

interface Top {

**default** String name() { return "unnamed"; }

}

Default methods enable you to add new functionality to existing interfaces and ensure binary compatibility with code written for older versions of those interfaces. In particular, default methods enable you to add methods that accept lambda expressions as parameters to existing interfaces.

► **[Java 8] What happens with default methods when extending interface?**

When you extend an interface that contains a default method, you can do the following:

* Not mention the default method at all, which lets your extended interface **inherit** the default method.
* Redeclare the default method, which makes it abstract.
* Redefine the default method, which overrides it.

► **Can we create an instance of interface?**

Interfaces may not be directly instantiated. A variable declared as an interface may have as its value a reference to any instance of a class which implements the specified interface. It is not sufficient that the class happen to implement all the abstract methods of the interface; the class or one of its superclasses must actually be declared to implement the interface, or else the class is not considered to implement the interface.

List list = new ArrayList();

► **What is reflection? Where to use it?**

Reflection enables Java code to discover information about the fields, methods and constructors of loaded classes, and to use reflected fields, methods, and constructors to operate on their underlying counterparts, within security restrictions. **Reflection API** is located in **java.lang.reflect** package.

Reflection is commonly used by programs which require the ability to examine or modify the runtime behavior of applications running in the JVM: IDE, debugger, test tools, etc. For normal application code, if it is possible to perform an operation without using reflection, then it is preferable to avoid using it because of performance overhead, security restrictions, and breaking encapsulation.

► **What is introspection?**

Introspection was introduced for **Java Beans** around 1996 to allow to figure out which properties, events, and methods a Java Bean supports at runtime and in the builder environment. Introspection is a composite mechanism. First, low level reflection mechanism is used to study the methods supported by a bean and second, simple design patterns are applied to deduce from those methods what properties, events, and public methods are supported. For example, design patterns to locate simple properties are by looking for methods of the form:

public <PropertyType> get<PropertyName>();

public void set<PropertyName>(<PropertyType> a);

►► **What is the difference between introspection and reflection?**

Introspection is used to determine JavaBean properties by using (1) reflection and (2) design patterns.

Reflection API allows not just to discover information about class but also to manipulate those, e.g. to change value of private field or to execute private method, not accessible otherwise.

► **Can you access the private method from outside the class?**

Yes, by changing the runtime behavior of a class using reflection, and if the class is not secured.

► **What is a native method?**

A native method is a method that is implemented in a language other than Java, usually C++.

► **What is the purpose of the System class?**

The purpose of the System class is to provide access to system resources.

# Annotations

► **[Java 5] What is annotation?**

An annotation is a marker (modifier) which associates information with a program construct, but has no effect at run time. An annotation denotes a specific invocation of an annotation type and usually provides values for the elements of that type. An annotation type declaration is a special kind of **interface** declaration.

► **Where could we use annotations?**

Annotations may be used as modifiers in any declaration, e.g package, class, interface, field, method, parameter, constructor, or local variable. Annotations may also be used on enum constants. Annotations are conventionally placed before all other modifiers, but this is not a requirement; they may be freely intermixed with other modifiers.

@Preliminary // Marker class annotation

public class TimeTravel { ... }

// Single-element complex annotation

@Author(@Name(first = "Joe", last = "Hacker"))

public class BitTwiddle { ... }

@Override // method override

public boolean equals(Object \_other) { ... }

► **Are there different kinds of annotation?**

There are three kinds of annotations. The first kind is the most general, while the others are merely shorthands for the first kind.

* normal annotation
* marker annotation
* single element annotation

► **How to declare annotation?**

An annotation type declaration is a special kind of **interface** declaration. To distinguish an annotation type declaration from an ordinary interface declaration, the keyword interface is preceded by an at sign (@).

public **@interface** Copyright {

String value();

}

public **@interface** Name {

String first();

String last();

}

► **Are there predefined annotations?**

Several annotation types are predefined in the libraries of the Java platform, e.g.:

[Java 5] @Target, @Retension, @Inherited, @Override, @SuppressWarnings, @Deprecated

[Java 7] @SafeVarargs

[Java 8] @Repeatable, @FunctionalInterface

# Expressions and Statements

► **What is poly expression?**

When some expressions appear in certain contexts, they are considered poly expressions. The following forms of expressions may be poly expressions:

* Parenthesized expressions
* Class instance creation expressions
* Method invocation expressions
* Method reference expressions
* Conditional expressions
* Lambda expressions

► **When / how should we use assertions?**

Typically, assertion checking is enabled during program development and testing, and disabled for deployment, to improve performance. Because assertions may be disabled, programs must not assume that the expressions contained in assertions will be evaluated.

// avoid side effects, action should not be part of assertion

boolean nullsRemoved = names.remove(null); // Runs whether or not asserts are enabled

assert nullsRemoved;

► **Can we use assertions for argument checking?**

Assertions should **not** be used for argument checking in public methods. Argument checking is typically part of the contract of a method, and this contract must be upheld whether assertions are enabled or disabled. Also, if assertion fails, **AssertionError** is thrown. Erroneous arguments should result in an appropriate run-time exception (such as IllegalArgumentException, ArrayIndexOutOfBoundsException, or NullPointerException).

# [Java 8] Lambda Expressions and Functional Interfaces

►► **What is lambda (expression)?**

A lambda expression is "poly expression", which looks a lot like a method declaration with a list of formal parameters and a body expressed in terms of those parameters and local variables. We can consider lambda expression as **anonymous method**. Evaluation of a lambda expression produces an instance of a **functional interface.**

A lambda expression consists of the following:

* A comma-separated list of formal parameters enclosed in parentheses. The parentheses could be omitted if there is only one parameter. The formal parameters of a lambda expression may have either declared types (explicitly typed) or inferred types (implicitly typed). Mixing inferred and declared types is not allowed.
* The arrow token, **->**
* A body, which consists of a single expression or a statement block. A return statement is not an expression; you must enclose statements in braces { }.

► **Examples of lambda expressions?**

* () -> { System.gc(); } // No parameters, void block body
* (String s) -> s.length() // Single declared-type parameter
* (Thread t) -> { t.start(); } // Single declared-type parameter
* (int x, int y) -> x+y // Multiple declared-type parameters
* (x, y) -> x+y // Multiple inferred-type parameters
* p -> p.getGender() == Person.Sex.***MALE*** && p.getAge() >= 18 && p.getAge() <= 65
* (x, int y) -> x+y // Illegal: can't mix inferred and declared types
* (x, final y) -> x+y // Illegal: no modifiers with inferred types

► **Can lambda expressions declare type parameters?**

Lambda expressions cannot declare type parameters.

foo( (x) <y,z> (w) -> v ) // Illegal

► **What is the result of lambda expression?**

Evaluation of a lambda expression produces an instance of a **functional interface**, similar to a class instance creation.

Either a new instance of a class with the properties is allocated and initialized, or an existing instance of a class with the properties is referenced. Lambda expression evaluation does not cause the execution of the expression's body.

This may occur later when an appropriate method of the functional interface is invoked.

► **How the [return] type of a lambda expression is determined?**

To determine the type of a lambda expression, the Java compiler uses the **target type of the context** or situation in which the lambda expression was found. It follows that you can only use lambda expressions in situations in which the Java compiler can determine a target type:

* Variable declarations
* Assignments
* Return statements
* Array initializers
* Method or constructor arguments
* Lambda expression bodies
* Conditional expressions, ?:
* Cast expressions

For method arguments, the Java compiler determines the target type with two other language features: **overload resolution** and type argument **inference**.

► **Can we use non final local variables in lambda expressions?**

Any local variable, formal parameter, or exception parameter used but not declared in a lambda expression must either be declared final or be "effectively final" (no assignments), or a compile-time error occurs where the use is attempted.

void m(int x) {

foo( () -> x+1 );

x++; // Illegal: x is not effectively final.

}

► **Can we serialize lambda expressions?**

You can serialize a lambda expression if its target type and its captured arguments are serializable. However, like inner classes, the serialization of lambda expressions is **strongly discouraged**.

► **What is functional interface?**

A functional interface is an interface that has just **one abstract method** (aside from the methods of Object and **default** methods), probably inherited and overridden with equivalent signature, and thus represents a **single function contract**. E.g:

interface Runnable {

void run();

}

Because functional interface contains only one abstract method, you can omit the **name** of that method when you implement it using an anonymous class expression or lambda expression,

► **How to create an instance of a functional interface?**

Instances of functional interfaces can be created

* by usual process of creating an interface instance by declaring and instantiating a class,
* by method reference expression,
* by lambda expression.

► **What are Java standard functional interfaces?**

Java 8 provides several standard functional interfaces (annotated **@FunctionalInterface**) in **java.util.function** package, which could be used as the **assignment target** for a lambda expression or method reference, including interfaces below as well as several primitive specializations (e.g. BooleanSupplier, DoubleToLongFunction, ObjIntConsumer<T>):

* Predicate<T>
* Consumer<T>
* Supplier<T>
* UnaryOperator<T>
* BinaryOperator<T>
* Function<T,R>

► **What is Predicate?**

Predicate<T> is a Java standard functional interface which represents a **predicate** (boolean-valued function) of a single argument with functional method: boolean test(T t).

Predicate<Person> predicate = // p is Person and lambda return will be Predicate

p -> p.getGender() == Person.Sex.***MALE*** && p.getAge() >= 18 && p.getAge() <= 65;

Person person = **new** Person(

"Fred", IsoChronology.***INSTANCE***.date(1980, 6, 20), Person.Sex.***MALE***, "fred@example.com");

**if** (predicate.**test**(person)) {

person.printPerson();

}

► **What is Consumer?**

Consumer<T> is a Java standard functional interface which represents an operation that accepts a **single input argument** and **returns no result**. Functional method is: void accept(T t). Unlike most other functional interfaces, Consumer is expected to operate via side-effects. E.g.

Consumer<Integer> consumer = (y) -> { System.out.println("y = " + y); };

consumer.accept( 73);

► **What is Function?**

Function<T,R> is a Java standard functional interface which represents an operation that accepts **single input argument** and produces **single result**. Functional method is: R apply(T t). E.g.:

Function<Person, String> mapper = p -> p.getEmailAddress();

Person person = new Person(

"Fred", IsoChronology.INSTANCE.date(1980, 6, 20), Person.Sex.MALE, "fred@example.com");

String email = mapper.apply(person);

# Collections and Concurrent Collections

► **What is the difference between StringBuffer and StringBuilder?**

StringBuffer is synchronized whereas StringBuilder is not synchronized.

► **What is the difference between ArrayList and Vector?**

ArrayList is not synchronized, it increases size 50% compared to array. Vector is a legacy class, it is synchronized. Vector doubles array size.

►► **What is Stack?**

Stack is **legacy** class representing a Last-In-First-Out (**LIFO**) stack of objects. It extends **Vector** with five operations that allow a vector to be treated as a stack - **push, pop, peek, search, empty**. A more complete and consistent set of LIFO stack operations is provided by the **Deque** interface and its implementations (e.g. ArrayDeque), which should be used instead of Stack.

► **What is Deque?**

Deque<E> extends Queue<E> and represents a linear collection that supports element **insertion and removal at both ends**. The name deque is short for "double ended queue" and is usually pronounced "deck". Deques can also be used as **stacks** in preference to the legacy **Stack** class. When a deque is used as a stack, elements are pushed and popped from the beginning of the deque. Most Deque implementations place no fixed limits on the number of elements they may contain, but this interface supports capacity-restricted deques as well as those with no fixed size limit.

Deque<Integer> stack = new ArrayDeque<Integer>();

**►► What is the difference between Queue / Deque and Stack?**

Queues typically (but do not necessarily) order elements in a **FIFO** (first-in-first-out) manner. Among the exceptions are **priority queues** (PriorityQueue<E>) which order elements according to a supplied comparator or the elements' natural ordering, and **LIFO queues** (or **stacks**). **Deque** supports element insertion and removal at both ends.

**►►► What is doubly linked list? Minimal interface?**

A doubly linked list is a linked data structure that consists of a set of sequentially linked records called **nodes**. Each node contains two fields, called **links**, that are references to the **previous** and to the **next** node in the list. The minimal interface for doubly linked list is:

* next
* previous
* insert
* remove

**►►**► **What is LinkedList? Is LinkedList a doubly linked list?**

**LinkedList** is java.util class implementing **List**<E> and **Deque**<E> (and **Queue**<E>). It is **implemented** as a doubly linked list but it does not provide / expose doubly linked interface methods **next** and **previous**. We can use get(int index) instead to get the element at the specified position in the list.

**►►**► **What is the difference between ArrayList and LinkedList? If you had to add 100k elements to a data structure, would you use an ArrayList or LinkedList?**

ArrayList and LinkedList both implement **List** interface and maintain insertion order. Both are **not synchronized**.

* **ArrayList** internally uses dynamic array to store elements and implements **RandomAccess** marker interface, i.e. provides fast (generally constant time) random access. Adding or removing elements is slower than LinkedList. If size is unknown, adding elements could require multiple reallocations of the internal array and memory copy. The add() operation runs in amortized constant time, that is, adding n elements requires **O(n)** time. ArrayList is better for storing and accessing data (sequentially or randomly).
* **LinkedList** is doubly-linked list implementation of List and Deque. It is generally faster to add or remove element at either end as it has constant time to add / delete (if we are already at needed index). It has get(index) method to find specific index in the list but without true random access - by traversing the list from the beginning or the end, whichever is closer to the needed index.
* It is doubtful which one is better if you want to be able to insert items in the middle of the large list such as a priority queue. It might be better to store each priority list separate.

►►► **What is the difference between List and Set?**

List is **ordered** and can contain **duplicate** elements whereas Set is unordered and contains only unique elements.

► **What is the difference between HashSet and TreeSet?**

HashSet maintains no order whereas TreeSet maintains ascending order.

►► **What is the difference between Set and Map?**

Set contains only unique values, whereas Map contains [key x value] entries with unique keys and possibly duplicate values.

►► **What is hash collision [in Hashtable] and how is it handled in Java?**

Two different keys with the same hash value is known as hash collision. Two different entries will be kept in a single hash **bucket** to avoid the collision.

► **What is the default load factor in hashing based collection?**

The default load factor is **0.75**.

default capacity = initial capacity x load factor.

For example, 16 \* 0.75 = 12. So, 12 is the default capacity of Map.

►►► **What is HashMap, its structure, what happens when you add to HashMap?**

HashMap is not synchronized implementation of the Map interface which permits **null** values and the **null** key and with all optional map operations. HashMap instance has two parameters affecting its performance: **initial capacity** and **load factor**. The capacity is the number of **buckets** in the hash table. The load factor is a measure of how full the hash table could get before its capacity is automatically increased. When the number of entries in the hash table exceeds [the load factor \* current capacity], the hash table is **rehashed** with **twice** the number of buckets.

**►►► How fast are lookups and iterations a HashMap?**

HashMap provides **constant-time performance, O(1) time** for the basic get and put operations, assuming the hash function disperses the elements evenly among the **buckets**. Extra linear time will be needed if hashing was not efficient and a bucket contain many entries with the same hash.

Iteration over collection requires time proportional to the capacity of the HashMap instance (the number of buckets) plus its size (the number of key-value mappings), linear time O(n). Thus, it is important not to set the initial capacity too high or the load factor too low if iteration performance is important.

►►► **What is the difference between HashMap and Hashtable?**

**Hashtable** is a legacy collection extending obsolete **Dictionary.** It is synchronized and cannot contain null key or null value. **HashMap** is not synchronized and can contain one null key and multiple null values. If a thread-safe implementation is not needed, it is recommended to use **HashMap**. If a thread-safe highly-concurrent implementation is desired, then it is recommended to use **ConcurrentHashMap**.

►►► **What is the difference between HashSet and HashMap?**

Both collections utilize hashing. **HashSet** contains only unique values and can be iterated. **HashMap** contains entries [key x value] and needs to be converted into Set of Entries, keys or values to be iterated.

►►► **How to create ordered list with unique elements?**

One way is to subclass ArrayList and override add() methods to test if inserted object is unique by using contains() method. Another way is to use java.util.**LinkedHashSet** which preserves the **order of insertion** and has the **set** semantics. It maintains a doubly-linked list running through all entries which defines the iteration ordering.

Several ways:

* subclass **ArrayList** and override **add()** method to check for uniqueness.
* **SortedSet** interface - a set of **Comparable** elements, implementations **TreeSet** and **ConcurrentSkipListSet**.
* **LinkedHashSet** class preserves the order of insertion and returns iterator with that order. It allows to re-insert existing values (but without affecting insertion order). It maintains a doubly-linked list of elements.

► **What is the difference between HashMap and TreeMap?**

HashMap maintains no order but TreeMap maintains ascending order.

► **What is the Dictionary class?**

Dictionary<K,V> class is obsolete abstract class which maps keys to values. Every key is associated with at most one value. Both key and value should not be null. The only subclass now is **Hashtable**. New implementations should implement the Map interface.

► **What is the difference between Collection and Collections?**

Collection is an interface whereas Collections is a class. Collection interface provides normal functionality of data structure to List, Set and Queue. But, Collections class is to sort and synchronize collection elements.

**►►**► **What is the difference between Comparable and Comparator?**

**Comparable** is in java.lang package, it provides only one sort of sequence with method compareTo(). We implement Comparable interface with actual class modified.

**Comparator** is in java.util package, it provides multiple sort of sequences with method named compare(). Actual class is not modified using Comparator.

► **What is the advantage of generic collection?**

If we use generic class, we don't need typecasting. It is typesafe and checked at compile time.

► **What are concurrent collections?**

The java.util.concurrent package includes a number of additions to the Java Collections Framework. All of these collections help avoid Memory Consistency Errors by defining a happens-before relationship. These are most easily categorized by the collection interfaces provided:

* *BlockingQueue* <- ArrayBlockingQueue, LinkedBlockingQueue, LinkedBlockingDeque, ConcurrentLinkedQueue
* *ConcurrentMap* <- ConcurrentHashMap
* *ConcurrentNavigableMap* <- ConcurrentSkipListMap
* *List* <- CopyOnWriteArrayList
* *Set* <- ConcurrentSkipListSet

► **What is BlockingQueue?**

BlockingQueue is concurrent collection which defines a first-in-first-out (**FIFO**) data structure (queue) that blocks or times out when you attempt to add to a full queue, or retrieve from an empty queue.

► **What is ConcurrentMap? ConcurrentHashMap? Difference from HashMap and HashTable?**

ConcurrentMap is a subinterface of java.util.Map that defines useful atomic operations. These operations remove or replace a key-value pair only if the key is present, or add a key-value pair only if the key is absent. Making these operations atomic helps avoid synchronization.

**ConcurrentHashMap** is standard implementation of ConcurrentMap, and is a concurrent analog of **HashMap and** an alternative to **Hashtable**. HashMap is not synchronized and should be used either by single thread or with synchronization. ConcurrentHashMap uses multiple "buckets" to store data. This avoids read locks and greatly improves performance over a Hashtable. Both are thread safe, but there are obvious performance wins with ConcurrentHashMap.

► **What is ConcurrentNavigableMap? ConcurrentSkipListMap? Difference from TreeMap?**

ConcurrentNavigableMap is a subinterface of ConcurrentMap that supports approximate matches. The standard implementation of ConcurrentNavigableMap is **ConcurrentSkipListMap**, which is a concurrent analog of **TreeMap**.

► **Is there ConcurrentHashSet? How to create one?**

The java.util.concurrent package does not include **concurrent hash set.** Some developers use ConcurrentHashMap with dummy values. In [Java 8] static method **newKeySet()** was added to ConcurrentHashMap. It creates a new Set backed by a ConcurrentHashMap from the given type to Boolean.TRUE. The Set returned by newKeySet() is like any normal Set implementations and allows any Set operation:

Set<Style> newStyle = ConcurrentHashMap.newKeySet();

The old keySet() method returns a **Set view** of the keys in this map.

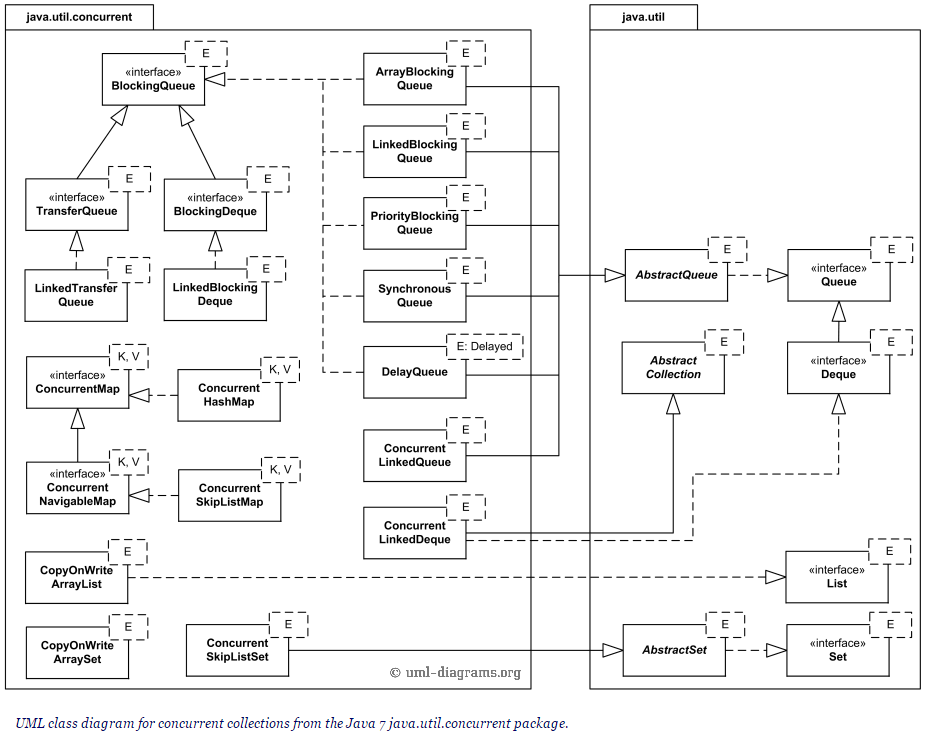
► **[Java 8] Explain Arrays with parallel operations / methods.**

In [Java 8] parallel operations were added to java.util.**Arrays**. These methods very efficiently utilize **multi-core** systems.

* Arrays.paralellSort()
* Arrays.parallelPrefix()
* Arrays.parallelSetAll()
* Spliterator

E.g. the code below modifies all elements in the array in parallel by multiplying each by 100:

Arrays.parallelSetAll( array, x -> x \* 100 );



**Synchronized HashMap：**

1. Each method is synchronized using an object level lock. So the get and put methods on synchMap acquire a lock.
2. Locking the entire collection is a performance overhead. While one thread holds on to the lock, no other thread can use the collection.

**ConcurrentHashMap was introduced in JDK 5.**

1. There is no locking at the object level,The locking is at a much finer granularity. For a ConcurrentHashMap, the locks may be at a hashmap bucket level.
2. The effect of lower level locking is that you can have concurrent readers and writers which is not possible for synchronized collections. This leads to much more scalability.
3. ConcurrentHashMap does not throw a ConcurrentModificationException if one thread tries to modify it while another is iterating over it.

### ConcurrentHashMap

* You should use ConcurrentHashMap when you need very high concurrency in your project.
* It is thread safe without synchronizing the whole map.
* Reads can happen very fast while write is done with a lock.
* There is no locking at the object level.
* The [locking](https://crunchify.com/how-to-create-your-own-non-blocking-queue-in-java-same-as-evictingqueue/) is at a much finer granularity at a hashmap bucket level.
* ConcurrentHashMap doesn’t throw a ConcurrentModificationException if one thread tries to modify it while another is iterating over it.
* ConcurrentHashMap uses multitude of locks.

### SynchronizedHashMap

* Synchronization at [Object level](https://crunchify.com/java-hashmap-containskeyobject-key-and-containsvalueobject-value-check-if-key-exists-in-map/).
* Every read/write operation needs to acquire lock.
* Locking the entire collection is a performance overhead.
* This essentially gives access to only one thread to the entire map & blocks all the other threads.
* It may cause contention.
* SynchronizedHashMap returns Iterator, which fails-fast on concurrent modification.

#### Now let’s take a look at code

# [Java 8] Multthreading

► **What are the different types of threadpools?**

There are various thread pools in java:

* Single Thread Executor : A thread pool with only one thread. So all the submitted tasks will be executed sequentially. Method : Executors.newSingleThreadExecutor()
* Cached Thread Pool : A thread pool that creates as many threads it needs to execute the task in parrallel. The old available threads will be reused for the new tasks. If a thread is not used during 60 seconds, it will be terminated and removed from the pool. Method : Executors.newCachedThreadPool()
* Fixed Thread Pool : A thread pool with a fixed number of threads. If a thread is not available for the task, the task is put in queue waiting for an other task to ends. Method : Executors.newFixedThreadPool()
* Scheduled Thread Pool : A thread pool made to schedule future task. Method : Executors.newScheduledThreadPool()
* Single Thread Scheduled Pool : A thread pool with only one thread to schedule future task. Method : Executors.newSingleThreadScheduledExecutor()

# [Java 8] Streams

► **What is [purpose of] Stream[ing] API?**

Before Java 8, the imperative style of getting an iterator and looping over a collection means all the collection processing is done external to the collection. After Java 8, the **functional style** of passing a lambda expression to a stream means the collection processing is done internally. Lambdas - or, more specifically, functional interfaces - play a big part in the Streaming API (**java.util.stream.Stream**), while lambdas are also useful outside of the Streaming API.

Streams also made intermediate collections and maps no longer necessary. Before Java 8, collection processing was complex and involved multiple steps. After Java 8, streams can be filtered and mapped to different streams or collected and reprocessed all before producing a final result.

Let's suppose you have a List<Person> object and you want to sort this list first by the cities (alphabetically) that each person lives in, and then, within each city, you want the people living in that city listed by last name in reverse alphabetical order. This complex, multiple step process can easily be done with the Streaming API and lambdas:

List<Person> result =

persons.stream().sorted(

Comparator.comparing( Person::getCity)

.thenComparing( Person::getLastName, Comparator.reverseOrder())

)

.collect( Collectors.toList());

►► **What are differences between collections and streams?**

Collections and streams, while bearing some superficial similarities, have different goals. Collections are primarily concerned with the efficient management of, and access to, their elements. By contrast, streams do not provide a means to directly access or manipulate their elements, and are instead concerned with declaratively describing their source and the computational operations which will be performed in aggregate on that source. However, if the provided stream operations do not offer the desired functionality, the BaseStream.iterator() and BaseStream.spliterator() operations can be used to perform a controlled traversal.

► **What are aggregate (stream) operations?**

Aggregate (also called stream) operations process elements from a stream, not directly from a collection, e.g. filter, mapToInt, forEach, average. Sum, min, max, average, and string concatenation are all special cases of **reduction**.

► **What are differences between aggregate operations and iterators?**

They have several fundamental differences. Aggregate operations:

* use internal iteration. They do not contain a method like next to process the next element of the collection.
* can take advantage of parallel computing. External iteration can only iterate over collection sequentially.
* process elements from a stream, not directly from a collection.
* support behavior as parameters: You can specify lambda expressions as parameters for most aggregate operations. This enables you to customize the behavior of a particular aggregate operation.

► **What is Stream?**

Stream<T> in java.util.stream is interface representing a sequence of elements supporting sequential and parallel aggregate operations.

Example of aggregate operation using Stream, IntStream and lambdas. In this example, widgets is a Collection<Widget>. We create a sequential stream of Widget objects via Collection.stream(), filter it to produce a stream containing only the red widgets, and then transform it into a stream of integer values representing the weight of each red widget. Then this stream is summed to produce a total weight:

int redWeight = widgets.stream()

.filter(w -> w.getColor() == RED)

.mapToInt(w -> w.getWeight())

.sum();

► **How to create a Stream?**

* Stream<E> Collection<E>.stream() returns a **sequential** Stream with collection as its source
* Stream<E> Collection<E>.parallelStream() returns a (possibly) **parallel** Stream
* static <T> Stream<T> Stream.empty() returns an empty sequential Stream.
* static <T> Stream<T> Stream.of(T t) returns a sequential Stream containing a single element.
* static <T> Stream<T> Stream.generate(Supplier<T> s) returns an infinite sequential unordered stream where each element is generated by the provided Supplier.

E.g.:

Collection<Widget> widgets = ...

Stream<Widget> stream = widgets.parallelStream();

► **What is [stream] pipeline?**

To perform a computation, stream operations are composed into a stream pipeline. A stream pipeline consists of

* a source (which might be an array, a collection, a generator function, an I/O channel, etc),
* zero or more intermediate operations (which transform a stream into another stream, such as filter(Predicate)),
* a terminal operation (which produces a result or side-effect, such as count() or forEach(Consumer)).

Example below prints the male members contained in the collection roster with a pipeline that consists of the aggregate operations filter and forEach, and lambdas:

roster.stream()

.filter(e -> e.getGender() == Person.Sex.MALE)

.forEach(e -> System.out.println(e.getName()));

► **Are streams lazy?**

Yes, streams are lazy; computation on the source data is only performed when the terminal operation is initiated, and source elements are consumed only as needed.

► **What is parallel stream?**

Parallel stream allows parallel execution: the Java runtime partitions the stream into multiple substreams. Aggregate operations iterate over and process these substreams in parallel and then combine the results.

For example, the following statement calculates the average age of all male members in parallel:

double average = roster

.parallelStream()

.filter(p -> p.getGender() == Person.Sex.MALE)

.mapToInt(Person::getAge)

.average()

.getAsDouble();

► **What is interference?**

Interference occurs when the source of a stream is modified while a pipeline processes the stream.

For example, code attempts to concatenate the strings contained in the List listOfStrings, which will cause ConcurrentModificationException. Lambda expressions in stream operations should not interfere.

► **What is the order in which elements of stream are processed?**

The order in which a pipeline processes the elements of a stream depends on whether the stream is executed in serial or in parallel, the source of the stream, and intermediate operations.

# Generics

► **What is type variable / type parameter?**

A type variable is an unqualified identifier declared as **type parameter** by

* generic class declaration
* generic interface declaration
* generic method declaration
* generic constructor declaration

Type parameters may have an optional **bound** – "extends" some class or interface, and other additional interfaces. Examples of type parameters:

<T>

<T extends C>

<T extends C & I>

<T extends C & I1, I2>

The members of a type variable <T extends C & I1, ... In> are the members of the intersection type T & I1 ... In appearing at the point where the type variable is declared. Private members are not inherited as well as package, if C had been declared in a different package than T.

►► **Explain generic class / interface?**

A class or interface is generic if it declares one or more **type variables** known as the type parameters of the class. The type parameter section follows the class name and is delimited by angle brackets. It defines one or more type variables that act as parameters. E.g. class Pair below declares two type parameters/variables – T and S:

class Pair**<T, S>** {

private T fst;

private S snd;

Pair (T f, S s) { //-- parameter types used in constructor

fst = f; snd = s;

}

}

► **What is parameterized type?**

A generic class or interface declaration defines a set of parameterized types, one for each possible invocation of the type parameter section. A parameterized type consists of a generic class or generic interface name and an actual type argument list C<T1 , ... , Tn>. Type argument may be either reference type or wildcard ("?"), and wildcard could be unbounded or may have optional bounds. Examples:

* Vector<String>
* Seq<Seq<A>>
* Pair<String,String>
* Collection<? extends S>
* Collection<? super T>
* // Vector<int> -- illegal, primitive types cannot be arguments
* // Pair<String> -- illegal, not enough arguments

Parameterized types share **the same class** at runtime. For instance, executing the code below will result in the variable b holding true.

Vector<String> x = new Vector<String>();

Vector<Integer> y = new Vector<Integer>();

boolean b = x.getClass() == y.getClass();

► **When to use wildcards in parameterized types, why not use Object instead?**

Wildcards ("?") are useful in situations where only partial knowledge about the type parameter is required. Wildcard could be unbounded or may have optional bound:

• upper bound: extends ReferenceType

• lower bound: super ReferenceType

boolean addAll(Collection<? extends E> c) { … }

void printCollection(Collection<?> c) { // wildcard collection

for (Object o : c) {

System.out.println(o);

}

}

Using Collection<Object> instead as the type of the incoming parameter would not as useful; the method could only be used with an actual parameter that had type Collection<Object>, which would be quite rare. In contrast, the use of an unbounded wildcard allows any kind of collection to be used as a parameter.

►► **Explain generic methods?**

A method is generic if it declares one or more **type variables** (formal type parameters) the same way as a type parameter list of a class or interface. The scope of a method’s type parameter is the entire declaration of the method, including the type parameter section itself. Therefore, type parameters can appear as parts of their own bounds, or as bounds of other type parameters declared in the same section. Type parameters of generic methods need not be provided explicitly when a generic method is invoked, they are almost always **inferred**. E.g. method addDebugInfo declares type variable Value used as class of parameter \_value. Return type of the method is void.

public **<Value>** void addDebugInfo(String \_name, **Value** \_value) {

debugInfo.add(\_name + "=" + \_value.toString());

}

Long groupId = group.getId();

bug.addDebugInfo("groupId", groupId);

► **What is type erasure?**

Type erasure is a mapping from types possibly including parameterized types and type variables to types that are never parameterized types or type variables. We write |T| for the erasure of type T. The erasure mapping is defined as follows:

* The erasure of a parameterized type G<T1, ... ,Tn> is |G|.
* The erasure of a nested type T.C is |T|.C.
* The erasure of an array type T[] is |T|[].
* The erasure of a type variable is the erasure of its leftmost bound.
* The erasure of every other type is the type itself.

► **What are reifiable types?**

Types that are completely available at run time are known as reifiable types. Usually some type information is erased during compilation, and not all types are available at run time. A type is reifiable if and only if one of the following holds:

* It refers to a non-generic type declaration.
* It is a parameterized type in which all type arguments are unbounded wildcards.
* It is a raw type.
* It is a primitive type
* It is an array type whose component type is reifiable.

The decision **not to make all generic types reifiable** in **Java 5** is one of the most crucial, and controversial design decisions involving the language’s type system. Ultimately, the most important motivation for this decision was compatibility of generics with existing legacy code.

► **What is raw type?**

Raw type is defined to be either:

* erasure of a parameterized type – i.e. the name of a generic type declaration used without any accompanying actual type parameters.
* any non-static type member of a raw type R that is not inherited from a superclass or superinterface of R, esp. inner classes.

For example, **Vector<String> is parameterized type**, **Vector is raw type**. Raw types were introduced to facilitate interfacing with legacy, non-generic code. Variables of a raw type can be assigned from values of any of the type’s parametric instances. For instance, it is possible to assign a Vector<String> to a Vector. The reverse assignment from Vector to Vector<String> is unsafe since the raw vector might have had a different element type, but is still permitted using unchecked conversion in order to enable interfacing with legacy code. In this case, a compiler will issue an unchecked warning.

Raw types are closely related to **wildcards**. Raw types can be thought of as wildcards whose type rules are deliberately unsound, to accommodate interaction with legacy code. The use of raw types in code written after the introduction of genericity into the Java is **strongly discouraged**.

► **What is rare type?**

Partially raw type is called a "rare" type. E.g. Outer.Inner<Double> is rare type, as Outer is raw.

# Exceptions

► **What is the difference between checked and unchecked exception?**

Checked exception extends Throwable except RuntimeException and Error, e.g. IOException, SQLException etc.

Checked exceptions are checked at compile-time.

Unchecked exception extends RuntimeException, e.g. ArithmeticException, NullPointerException etc.

Unchecked exceptions are not checked at compile-time.

► **What is the base class for Error and Exception?**

Throwable.

► **Is it necessary that each try block must be followed by a catch block?**

It is not necessary that each try block must be followed by a catch block. It should be followed by either a catch block OR a finally block.

►► **Can finally block be used without catch?**

Yes

► **Is there any case when finally will not be executed?**

Finally block will not be executed if program exits (either by calling System.exit() or by causing a fatal error that causes the process to abort).

► **Can subclass overriding method declare an exception if parent class method doesn't throw an exception?**

Yes, but only unchecked exception.

# I/O

► **What is the difference between the Reader / Writer and InputStream / OutputStream?**

The Reader/Writer class hierarchy is **character-oriented**, and the InputStream/OutputStream class hierarchy is **byte-oriented**.

**►►► What is serialization / serializable object?**

To serialize an object means to convert its **state** to a **byte stream** so that the byte stream can be reverted back into a copy of the object. A Java object is **serializable** if its class or any of its superclasses implements either the java.io.**Serializable** interface or its subinterface, java.io.**Externalizable**. The Java specifies a **default** way by which serializable objects are serialized.

**► How to customize serialization?**

A class can override default serialization and define its own way of serializing objects of that class. The class can optionally define the following methods:

* **writeObject** to control what information is saved or to append additional information to the stream.
* **readObject** either to read the information written by the corresponding writeObject method or to update the state of the object after it has been restored.
* **writeReplace** to allow a class to nominate a replacement object to be written to the stream.
* **readResolve** to allow a class to designate a replacement object for the object just read from the stream

► **What is transient [keyword]?**

If you define any data member as transient, it will not be serialized.

► **What is Externalizable?**

For Externalizable objects, only the identity of the class of the object is saved by the container; the **class itself must save and restore the contents**. Use Externalizable if you want complete control over serialization process, e.g. by using own **encoding** or **compression**, and concerned about bandwidth and time consumed. In earlier versions of Java, reflection was very slow, and so Serializing large object graphs was a performance problem. A downside of Externalizable is maintenance - if you add, remove or change a field in your class, you have to change methods to account for it.

The Externalizable class must implement the **java.io.Externalizable** interface:

* Implement a **writeExternal** method to save the state of the object (It must explicitly coordinate with its supertype to save its state.)
* Implement a **readExternal** method to read the data written by the writeExternal method from the stream and restore the state of the object (It must explicitly coordinate with the supertype to save its state.)
* Have the writeExternal and readExternal methods be solely responsible for the **format**, if an externally defined format is written.

Note: The writeExternal and readExternal methods are **public** and raise the **risk** that a client may be able to write or read information in the object other than by using its methods and fields. The Externalizable interface cannot be used for **inner classes** (they should implement the Serializable interface) because inner classes associated with enclosing instances cannot have no-arg constructors.

Externalizable interface is used to write the state of an object into a byte stream in compressed format. It is **not** a marker interface as Serializable.

# (Low Level) Multithreading - Threads

► **What is thread?**

A thread is a lightweight subprocess. It is a separate path of execution.

► **What is multithreading?**

Multithreading is a process of executing multiple threads simultaneously. Its main advantage is:

Threads share the same address space. Thread is lightweight.

Cost of communication between processes is low.

► **What is the difference between preemptive scheduling and time slicing?**

Threads may be supported by having many hardware processors, by time-slicing a single hardware processor, or by time-slicing many hardware processors. Under **time-slicing**, a task executes for a predefined slice of time and then reenters the pool of ready tasks. The scheduler then determines which task should execute next, based on priority and other factors. Under **preemptive scheduling**, the highest priority task executes until it enters the waiting or dead states or a higher priority task comes into existence.

**►**► **What is [the purpose of] join()?**

The join() method waits for a thread to die. In other words, it causes the currently running threads to stop executing until the thread it joins with completes its task.

► **Can we call the run() method instead of start()?**

Yes, but it will not work as a thread rather it will work as a normal object so there will not be context-switching between the threads.

**►►► How to stop a Thread? Why is Thread.stop deprecated?**

**Thread.stop()** is deprecated because it is inherently **unsafe**. Stopping a thread causes it to unlock all the monitors that it has locked. If any of the objects previously protected by these monitors were in an inconsistent state, other threads may now view these objects in an inconsistent state. Such objects are said to be **damaged**.

Most uses of stop should be replaced by code that simply modifies some **variable** to indicate that the target thread should stop running. The target thread should check this variable regularly, and simply **return** from its run method in an orderly fashion. To ensure prompt communication of the stop-request, the variable must be **volatile** (or access to the variable must be synchronized).

**►► How do I stop a thread that waits for long periods (e.g., for input)?**

That's what the **Thread.interrupt** method is for, to interrupt the wait:

public void shutdown() {

Thread moribund = waiter;

waiter = null;

**moribund.interrupt();**

}

For this technique to work, it's critical that any method that catches an interrupt exception and is not prepared to deal with it immediately **reasserts** the exception, rather than rethrows, because it is not always possible to rethrow the exception. If the method that catches the InterruptedException is not declared to throw this (checked) exception, then it should "reinterrupt itself" with the following incantation: Thread.currentThread().interrupt(). This ensures that the Thread will reraise the InterruptedException as soon as it is able.

► **What are daemon threads?**

Thread could be either **user thread** or a **daemon thread**. Daemon threads are basically threads that provide services to the user threads. An example for a daemon thread is the garbage collection.

When a JVM starts up, there is usually a single user (non-daemon) thread. The method setDaemon(bool) may be used to change whether or not a thread is a daemon. The JVM exits when the only threads running are all daemon threads, i.e. daemon thread does not prevent the JVM from exiting.

► **Can we convert user thread to daemon thread (if thread is started)?**

No, if you do so it will throw IllegalThreadStateException. Method setDaemon(bool) must be invoked before the thread is started.

► **When should we interrupt a thread?**

We should interrupt a thread if we want to break out the sleep or wait state of a thread.

►► **What is monitor? Mutually exclusive synchronization?**

The most basic way of communication between threads is **synchronization**, which is implemented using **monitors**. Each object in Java is associated with a monitor, which a thread can lock or unlock. Only one thread at a time may hold a lock on a monitor. A thread t may lock a particular monitor **multiple times**; each unlock reverses the effect of **one lock** operation.

Any other threads attempting to lock that monitor are blocked, waiting to acquire a lock in Thread.State.**BLOCKED** until they can obtain a lock on that monitor. Thus it is called "**mutualy exclusive**" synchronization. A thread may lock a particular monitor (acquire a lock) multiple times (otherwise we would often had deadlocks); each unlock reverses the effect of one lock operation.

Monitor-based concurrency was originally introduced with the **Mesa** programming language.

►►► **How to acquire lock / lock monitor? Can a thread enter a synchronized block without acquiring a lock?**

To acquire lock, use some synchronization approach, which ensures mutually exclusive access to the shared resource and prevents **data race.** If execution of the method's body is ever completed, either normally or abruptly, an unlock action is automatically performed on that same monitor.

* synchronized instance method (the monitor associated with 'this' is used.)
* synchronized statement / block,
* synchronized class method (static method locks on the Class object of the class).

A synchronized method **acquires a monitor before it executes**. Only one thread at a time may hold a lock on a monitor. Any other threads attempting to lock that monitor are blocked. A thread t may lock a particular monitor **multiple times**; each unlock reverses the effect of **one lock** operation.

► **What is locked by synchronized statement? Can we lock null?**

The synchronized statement computes provided expression to get a non-null reference to an object. If the value of the expression is null, a NullPointerException is thrown. It then attempts to acquire a mutual-exclusion lock (locks the monitor) on behalf of the executing thread on that object's monitor, and does not proceed further until the lock action has successfully completed. After block is executed, it releases the lock.

► **Can we have synchronized constructor?**

No, a constructor cannot be synchronized because there is no practical need. It would lock the object under construction, which is normally not available to other threads until all constructors for the object completed their work.

► **What happens with monitor if exception happens?**

The exception mechanism of Java is integrated with its synchronization model, so that monitors are unlocked as synchronized statements and invocations of synchronized methods complete abruptly.

►► **Is it possible to access object without lock? Can object be locked for exclusive use by a thread?**

Acquiring the lock associated with an object does not in itself prevent other threads from accessing fields of the object or invoking un-synchronized methods on the object. To achieve mutual exclusion other threads should also use synchronized method or the synchronized statement.

► **What is deadlock? How Java prevents deadlocks?**

Deadlock is a situation when two threads are waiting on each other to release a resource. Each thread waiting for a resource which is held by the other waiting thread. The Java neither prevents nor requires detection of deadlocks. Programs where threads hold (directly or indirectly) locks on multiple objects should use conventional techniques for deadlock avoidance, creating higher-level locking primitives that do not deadlock, if necessary.

► **What is wait set / wait-notify pattern?**

Every object, in addition to having an associated **monitor**, has an associated **wait set** of threads. Wait sets are manipulated solely through the methods Object.**wait**, Object.**notify**, and Object.**notifyAll**.

The **wait-notify pattern** is used in a set of cases where one thread needs to tell other threads that some event has occurred. It is commonly used to implement a thread pool or producer-consumer scenario, where a particular thread or threads need to "pick up jobs" created by other threads (in this case, the "event" that has occurred is that a job has arrived for one of the threads to pick up).

There is less need now to use wait()/notify(), since other classes are available in the Java concurrency package (java.util.concurrent) to handle these common situations. E.g. in a producer-consumer pattern, such as a logging thread, use BlockingQueue, to coordinate threads use CountDownLatch.

► **How is synchronization implemented in Java?**

Each object in Java is associated with a **monitor** and a **wait set** of threads. Only one thread at a time may hold a lock on a monitor. Using monitors is sometimes called **mutually exclusive** type of thread synchronization.

The wait sets support **thread cooperation** (aka **inter-thread communication**) in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed. It is implemented by the methods of Object: wait(), notify(), notifyAll().

Other mechanisms, such as reads and writes of **volatile** variables and the use of classes in the **java.util.concurrent** package (concurrency framework) and **java.util.concurrent.atomic** package with classes that support lock-free thread-safe programming on single variables, provide alternative ways of synchronization.

**►► What are challenges of Producer-Consumer problem / design pattern?**

The **producer-consumer problem** (also known as the **bounded-buffer** problem) is a classic Java example of a multi-process **synchronization** problem. The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer used as a queue. The producer’s job is to generate a piece of data, put it into the buffer and start again. At the same time, the consumer is consuming the data (i.e., removing it from the buffer) one piece at a time. The challenges are to make sure that the producer won’t try to add data into the buffer if it’s full and that the consumer won’t try to remove data from an empty buffer, produced items will not be left unconsumed and no starting items skipped.

► **What is volatile?**

**Volatile** field is an alternative to locking mechanism, to allow threads access to shared variable and ensure that shared variable is consistently and reliably updated. A **non-final** instance or static field may be declared **volatile**, in which case the Java Memory Model ensures that all threads see a consistent value for the variable. Threads will read the value of the volatile variable from main memory and don’t cache it locally, accesses to the shared values occur exactly as many times, and in exactly the same order, as they appear to occur during execution of the program text.

What is the **Java volatile** keyword? ... Declaring a **volatile Java** variable means: The value of this variable will never be cached thread-locally: all reads and writes will go straight to "main memory"; Access to the variable acts as though it is enclosed in a synchronized block, synchronized on itself.

► **Are the primitive data types like int, short, double thread-safe in Java? Is a volatile int in Java thread-safe?**

A single write to a non-volatile **long** or **double** value is treated as **two separate writes**: one to each 32-bit half. This can result in a situation where a thread sees the first 32 bits of a 64-bit value from one write, and the second 32 bits from another write. Writes and reads of **volatile** long and double values are always **atomic**.

**► What is atomic action? Explain atomic read/writes?**

An atomic action is one that effectively happens all at once. An atomic action cannot stop in the middle: it either happens completely, or it doesn't happen at all. No side effects of an atomic action are visible until the action is complete. Atomic actions cannot be interleaved, so they can be used without fear of thread interference. However, this does not eliminate all need to synchronize atomic actions, because **memory consistency** errors are still possible.

Writes and reads of **volatile** long and double values are always **atomic**. Writes to and reads of **references** are always atomic, regardless of whether they are implemented as 32-bit or 64-bit. An increment expression, such as **i++** is **not atomic**.

[Java 5] introduced **java.util.concurrent.atomic** package with classes that support lock-free thread-safe programming on single variables - AtomicBoolean, AtomicInteger, AtomicLong, AtomicReference, AtomicIntegerArray, etc

►►► **Why do we need wait(), just use synchronized? Does wait() method have to be called from synchronized method or block? What happens if you do call wait from outside a synchronized block?**

The two concepts are combined, not mutually-exclusive. When you use wait() **you need to own the monitor** on that object. So you need to have synchronized(..) on it before that. Using .wait() makes the current thread stop until another thread calls .notify() on the object it waits on. If thread executes the wait method on object and thread does not already possess the lock for target, then an IllegalMonitorStateException is thrown.

When wait() and notify() or notifyAll() come in is where your trying to write more efficient code. E.g., if you have a list of items that multiple threads share then if you put it in synchronized block of a monitor then threads will constantly jump in and run the code back and forth even with an empty list. The wait() is hence used on the monitor as a mechanism to tell all threads to chill out and stop using CPU cycles until further notice or notifyAll(). Thread waiting on the monitor's condition is in Thread.State.**WAITING**.

►►► **What is the difference between wait() and sleep() method? How to delay thread for some time?**

The Object.wait(t) method releases monitor lock and waits until either another thread invokes notify() or notifyAll() for this object, or a specified amount of time has elapsed.

The Thread.sleep(millis) method **keeps holding** the lock while sleeping. An alternative to delay is **TimeUnit**.MILLISECONDS.sleep(t) (which internally performs Thread.sleep) from java.util.concurrent.

► **Explain notify methods. What is the difference between notify() and notifyAll()?**

Notify methods **wake up** one or all threads waiting on this object's monitor. A thread waits on an object's monitor by calling one of the wait methods. The awakened thread(s) will not be able to proceed until the current thread releases (relinquishes) the lock on this object. The awakened thread has no reliable privilege or disadvantage in being the next thread to lock this object, it will compete in the usual manner with any other threads that might be actively competing to synchronize on this object. These methods should only be called by a thread that is the owner of this object's monitor. After calling notify(), we should exit the synchronized block as quickly as possible.

The notify() is used to awaken one waiting thread, chosen arbitrary, implementation dependent, if any threads are waiting on this object. The notifyAll() is used to awaken all the threads in waiting state.

► **What is shutdown hook?**

The shutdown hook is a thread invoked implicitly before JVM shuts down. So we can use it to perform some clean up. The JVM shuts down in response to two kinds of events:

* The program exits normally, when the last non-daemon thread exits or when the exit (equivalently, System.exit) method is invoked, or
* The JVM is terminated in response to a user interrupt, such as typing ^C, or a system-wide event, such as user logoff or system shutdown.

In rare circumstances the JVM may abort, that is, stop running without shutting down cleanly. If the JVM aborts then no guarantee can be made about whether or not any shutdown hooks will be run.

To add a hook, use:

Runtime.addShutdownHook(Thread hook)

# Concurrency and Parallelism

►►► **What is Concurrency framework?**

The concurrency utilities packages provide a powerful, extensible framework of high-performance threading utilities such as thread pools and blocking queues. This package frees the programmer from the need to craft these utilities by hand, in much the same manner the collections framework did for data structures. Additionally, these packages provide low-level primitives for advanced concurrent programming.

java.util.concurrent

java.util.concurrent.atomic

java.util.concurrent.locks

The concurrency utilities include:

• Task scheduling framework (Executor interface, etc)

• Fork/join framework

• Concurrent collections

• Atomic variables

• General purpose synchronization classes, including semaphores, barriers, latches, phasers, and exchangers

• Locks

• Nanosecond-granularity timing

►► **What are Executors?**

Threads work well for small applications, but in large-scale applications, it makes sense to separate thread management and creation from the rest of the application. Objects that encapsulate these functions are known as executors:

• Executor Interfaces define the three executor object types - Executor <- ExecutorService <- ScheduledExecutorService.

• Thread Pools are the most common kind of executor implementation.

• Fork/Join is a framework (new in JDK 7) for taking advantage of multiple processors.

► **What is Fork / Join framework?**

The fork / join framework is an implementation of the ExecutorService interface that helps you take advantage of **multiple processors (cores)**. It is designed for work that can be broken into smaller pieces recursively. The goal is to use all the available processing power to enhance the performance of your application.

The fork/join framework distributes tasks to worker threads in a thread pool. The fork/join framework is distinct because it uses a work-stealing algorithm. Worker threads that run out of things to do can steal tasks from other threads that are still busy. The center of the fork/join framework is the ForkJoinPool class which implements the core work-stealing algorithm and can execute ForkJoinTask processes.

# Networking

► **How to get server IP address or host name?**

InetAddress addr = InetAddress.getLocalHost();

localHost = addr.getHostName(); //-- local host name

localHostIp = addr.getHostAddress(); //-- local host IP

Currently there is no way to request IPv4 or IPv6.

► **How to convert a numeric IP address like 192.18.97.39 into a hostname like java.sun.com?**

By InetAddress.getByName("192.18.97.39").getHostName().

# Localization / Internationalization

► **What is Locale?**

A Locale object represents a specific geographical, political, or cultural region.

An operation that requires a Locale to perform its task is called locale-sensitive and uses the Locale to tailor information for the user.

For example, displaying a number is a locale-sensitive operation — the number should be formatted according to the customs and conventions of the user's native country, region, or culture.

A Locale object logically consists of the:

• language, e.g. "en" (English), "ja" (Japanese), "kok" (Konkani)

• script, e.g. "Latn" (Latin), "Cyrl" (Cyrillic)

• country (region), e.g. "US" (United States), "FR" (France), "029" (Caribbean)

• variant

• extension

Java provides a number of classes that perform locale-sensitive operations, e.g.

NumberFormat.getCurrencyInstance(myLocale)

► **How will you create a specific locale?**

The Locale class provides three constructors, factory method and a number of convenient constants that you can use to create Locale objects for commonly used locales.

For example, the following creates a Locale object for the United States:

Locale.US

► **What is ResourceBundle?**

Resource bundles contain locale-specific objects. When your program needs a locale-specific resource, a String for example, your program can load it from the resource bundle that is appropriate for the current user's locale. In this way, you can write program code that is largely independent of the user's locale isolating most, if not all, of the locale-specific information in resource bundles.

This allows you to write programs that can:

• be easily localized, or translated, into different languages

• handle multiple locales at once

• be easily modified later to support even more locales.

► **How will you load a specific locale?**

When your program needs a locale-specific object, it loads the ResourceBundle class using the getBundle method:

ResourceBundle myResources = ResourceBundle.getBundle("MyResources", currentLocale);

Resource bundles contain key/value pairs. The keys uniquely identify a locale-specific object in the bundle. If the object is not found, the getter method throws a MissingResourceException.

buttonOk = new Button( myResources.getString("OkKey"));

Java provides two subclasses of ResourceBundle, ListResourceBundle and PropertyResourceBundle, or you can write your own ResourceBundle subclass.

# JVM, JDK

► **What is JVM?**

A virtual machine (VM) is a high level abstraction on top of the native operating system, that emulates a physical machine.

► **What are types of memory allocated by JVM?**

* Class (Method) Area
* Heap
* Stack
* CPU registers
* Program Counter register
* Native Method Stack

**►►► Is the JVM stack based or register based?**

There are two main ways to implement a VM: stack based or register based. A **stack based VM** stores operands in a stack. Operations are carried out by popping operands from the stack, processing them and pushing in back the results. A stack based design makes very few assumptions about the target hardware (registers, CPU features), so it's easy to implement a VM on a wide variety of hardware. Since the operands for instructions are largely implicit, the object code will tend to be smaller.

**Register based** implementation of a VM the data structure stores operands in the registers of the CPU. There is no push / pop operations here, but operations need to contain the addresses of the registers. Code generators for such VM should know number and types of registers of the CPU.

Examples of **stack based** VM’s are the **Java VM**, the .Net CLR, and is the widely used method for implementing virtual machines. Examples of **register based** VMs are the Lua VM, and the Google’s Dalvik VM.

► **What gives Java its 'write once and run anywhere' nature?**

Java is compiled to be a **bytecode** which is the intermediate language between source code and machine code. This byte code is not platform specific and hence can run on any platform.

► **What is JIT compiler?**

Just-In-Time(JIT) compiler is used to improve the performance. JIT compiles parts of the byte code that have similar functionality at the same time, and hence reduces the amount of time needed for compilation. Here the term “compiler” refers to a translator from the instruction set of a Java virtual machine (JVM) to the instruction set of a specific CPU.

► **What is classloader?**

The classloader is a subsystem of JVM that is used to load classes and interfaces. There are many types of classloaders e.g. Bootstrap classloader, Extension classloader, System classloader, Plugin classloader etc.

► **What is Garbage Collection?**

Garbage collection is a process of reclaiming the runtime unused objects. It is performed for memory management.

► **What is gc()?**

gc() is a daemon thread. gc() method is defined in System class that is used to send request to JVM to perform garbage collection.

► **What is the purpose of finalize() method?**

finalize() method is invoked just before the object is garbage collected.

It is used to perform cleanup processing.

► **Can an unrefrenced objects be referenced again?**

Yes

► **What is the purpose of the Runtime class?**

The purpose of the Runtime class is to provide access to the Java runtime system.

► **How will you invoke any external process in Java?**

By Runtime.getRuntime().exec(?) method.

► **What is the differences between PermGen and Metaspace?**

Until [Java 7] there was an area in JVM memory called PermGen (permanent generation), where JVM used to keep its classes. PermGen always has a fixed maximum size, including default maximum.

You cannot make PermGen auto increase, if more memory is needed.

If there are too many classes loaded exceeding that maximum, JVM dies with "java.lang.OutOfMemoryError: PermGen space".

In [Java 8] PermGen has been completely removed, and classes metadata is now stored in the native heap called Metaspace. Metaspace by default auto increases its size, only limited by the amount of available native memory. Still, if needed you can set a fixed maximum for Metaspace with JVM parameter -XX:MaxMetaspaceSize=<NNN>.

A garbage collection is induced to collect the dead classloaders and classes when the class metadata usage reaches MetaspaceSize (initial amount of space). Set MetaspaceSize to a higher value to delay the induced garbage collections.

# Servlets

**►► Difference between [HTTP] GET and POST?**

The HTTP protocol defines **GET** requests as being idempotent, i.e. it is used for viewing something, without changing it. The GET method appends name/value pairs to the URL. Unfortunately, the length of a URL is limited, so this method only works if there are only a few parameters. The URL could be truncated if the form uses large number of parameters, or if the parameters contain large amounts of data. Also, parameters passed on the URL are visible in the address field of the browser, stored in history, etc. GET for AJAX requests is usually **cached**.

POSTs may have side effects as it could be used for changing something. This may result in the creation of a new resource or the updates of existing resources or both. This method packages the name/value pairs inside the body of the HTTP request, which makes for a cleaner URL and imposes no size limitations on the forms output. It is also more secure. POST for AJAX requests is usually **not** **cached**.

**► What Is a Servlet?**

A servlet is a Java class used to extend the capabilities of servers that host applications accessed by means of a request-response programming model. All servlets must implement the **Servlet** interface, which defines **lifecycle** methods. The javax.servlet and javax.servlet.http packages provide interfaces and classes for writing servlets.

**►► What is Filter?**

A filter is a Java web component that can transform the content of HTTP requests, responses, and header information. Filters do not generally create a response or respond to a request as servlets do, rather they modify or adapt the requests for a resource, and modify or adapt responses from a resource. Filters can act on dynamic or static web resources. Developer creates a filter by implementing the **javax.servlet.Filter** interface and providing a public **constructor** taking no arguments. Examples of Filters:

* Authentication filters
* Logging, auditing filters, resource access event triggers,
* Image conversion filters
* Data compression filters
* Encryption filters
* XSL/T filters that transform XML content.

**►► What is Servlet Lifecycle?**

The lifecycle of a servlet is controlled by the **container** in which the servlet has been deployed. When a request is mapped to a servlet, the container performs the following steps.

* If an instance of the servlet does not exist, the web container
  + Loads the servlet class.
  + Creates an instance of the servlet class.
  + Initializes the servlet instance by calling the **init** method.
* Invokes the **service** method, passing request and response objects.

If it needs to remove the servlet, the container finalizes the servlet by calling the servlet’s **destroy** method.

**►► What is the difference between GenericServlet and HttpServlet?**

**GenericServlet** is an abstract class which provides framework to create a Servlet for **any protocol**, e.g. you can write Servlet to receive content from FTP, SMTP etc, **HttpServlet** is abstract subclass of GenericServlet which is implemented to support **HTTP** requests.

**►► In web.xml file <load-on-startup>4</load-on-startup> is defined in <servlet> tag. What does it mean? What does 0 value mean?**

Servlets are normally **lazy loaded** and initialized, when there is incoming request to that servlet. The element <load-on-startup> indicates that this servlet should be loaded (instantiated and have its init() called) on the startup of the Web application. The integer indicates the order in which the servlet should be loaded. If the value is a **negative** integer, or the element is not present, the container is free to load the servlet whenever it chooses. If the value is a **positive**

integer or **0**, the container must load and initialize the servlet as the application is deployed. The container must guarantee that **servlets marked with lower integers are loaded before servlets marked with higher integers**.

**►► For initializing a servlet can we use a constructor in place of init()?**

No, we can’t use constructor for initializing a servlet because for initialization it needs an object of **servletConfig** to get all the parameters which are defined in deployment descriptor. Also, Java doesn't allow interfaces to declare constructors.

**►► What is the difference between ServletConfig and ServletContext?**

**ServletConfig** provides the information about the **configuration of a servlet** which is defined inside the web.xml file or deployment descriptor, it is a specific object for each servlet.

**ServletContext** is a web **application** specific object which is shared by all the servlet belonging to the application, it is a single object which represents our application could be used by servlets to communicate with the container.

**►► What methods you use in Servlets? What is difference between doGet() and doPost() method?**

The service provided by a servlet is implemented in the **service** method of a GenericServlet, in the **doGet**, **doPost**, **doPut**, **doDelete**, **doHead**, **doOptions**, **doTrace** methods of an HttpServlet object, or in any other protocol-specific methods defined by a class that implements the Servlet interface.

Typically when developing HTTP-based servlets, a servlet developer will only concern himself with the **doGet** and **doPost** methods to handle HTTP GET and POST requests.

**►► Who calls doGet() and doPost() methods?**

The servlet container calls the HTTP servlet's **service()** method in response to servlet requests, which automatically calls the appropriate servlet's doGet() or doPost() method.

**►► What are major enhancement or features added in Servlet 3.0 API**

Main areas of improvements and additions to Java™ Servlet 3.0 API (JSR 315) are

* **Pluggability** 
  + modularizing web.xml to allow frameworks / libraries to have their own entities defined
  + Adding APIs to ServletContext to allow addition of Servlets, Filters and Listeners at startup.
  + Use of **annotations** to declare all the components within a web application
* Ease of development – **annotations**, **generics**, **optional web.xml**
* **Async servlet** support
* Security enhancements

**►► What are servlet annotations?**

**Servlet API 3.0** defines **@WebServlet**, **@WebFilter**, **@WebListener** annotations.

Use the @WebServlet annotation to define a servlet component in a web application. This annotation is specified on a class and contains metadata about the servlet being declared. The annotated servlet must specify at least one URL pattern. Classes annotated with @WebServlet must extend the HttpServlet class. E.g.:

import javax.servlet.annotation.WebServlet;

**@WebServlet("/report")**

public class MoodServlet extends HttpServlet { . . . }

**►►► Explain async servlet processing.**

Async processing relies on a persistent HTTP connection between server and client. Two strategies:

* Streaming - browser opens a single persistent connection to the server for all events each time the server sends a new event, the browser interprets it.
* Long polling - a new request for each event (or set of events)

It could be used when a filter and/or servlet is unable to complete the processing of a request without **waiting for a resource or event** before generating a response. For example, a servlet may need to wait for an available JDBC connection, for a **response from a remote web service**, for a JMS message, or for an application event, before proceeding to generate a response. Waiting within the servlet is an inefficient operation as it is a blocking operation that consumes a thread and other limited resources.

**Servlet 3.0** introduces the ability for **asynchronous processing of requests** so that the web server thread may return to the container and perform other tasks. When asynchronous processing begins on the request, another thread or callback may either generate the response and call complete or dispatch the request so that it may run in the context

of the container using the AsyncContext.dispatch(). Some methods are: AsyncContext ServletRequest.startAsync(), setTimeout(long msec), void addListener(), void dispatch(path).

A typical sequence of events for asynchronous processing is:

1. The request is received and passed via normal filters for authentication etc. to the servlet.
2. The servlet processes the request parameters and/or content to determine the nature of the request.
3. The servlet issues requests for resources or data, for example, sends a remote web service request or joins a queue waiting for a JDBC connection.
4. The servlet returns without generating a response.
5. After some time, the requested resource becomes available, the thread handling that event continues processing either in the same thread or by dispatching to a resource in the container using the **AsyncContext**.

**►►► How do you upload a file [using Servlet 3.0]?**

If a request is of type **multipart/form-data** and if the servlet handling the request is annotated using the **@MultipartConfig**, the HttpServletRequest can make available the various parts of the multipart request via the following methods

* public Collection<Part> **getParts**()
* public Part **getPart**(String name).

Each part provides access to the headers, content type related with it and also the content via the **getInputStream**() method.

**►►► How to get / set cookies?**

The **HttpServletRequest** interface provides the **getCookies()** method to obtain an array of cookies that are present in the request. Typically, the only information that the client web browser sends as part of a cookie is the cookie name and value. Other cookie attributes are not typically returned.

Class **Cookie** creates a cookie. A cookie has a name, a single value, and optional attributes such as a comment,

path and domain qualifiers, a maximum age, and a version number. The servlet sends cookies to the browser by using the **HttpServletResponse.addCookie(Cookie)** method, which adds fields to HTTP response headers. The browser is expected to support 20 cookies for each Web server and may limit cookie size to 4 KB each.

HTTP 1.0 does not cache pages that use cookies created with this class. This class does not support the cache control defined with HTTP 1.1. This class supports both the Version 0 (by Netscape) and 1 (by RFC 2109) cookie specs. By default, cookies are created using **Version 0** to ensure the best interoperability.

**►►► How to set persistent cookie? How to delete cookie?**

Cookie.setMaxAge(int expiry) sets the maximum age of the cookie in seconds. A **positive** value indicates that the cookie will be **persisted** and will expire after the specified seconds have passed. Browser should delete expired cookies automatically. A **negative** value means that the cookie is not stored persistently (“session” cookie) and will be deleted when the Web browser exits. A **zero** value causes the cookie to be deleted.

**►►► How to secure cookie? How is it secured?**

**Cookie.setSecure(true)** indicates to the browser that the cookie should only be sent using HTTPS / SSL. The default value is **false**.

**►►► Explain session id**

Default Java EE session tracking could be done using session cookie (most used) or with URL rewriting mechanism (not secure, violates OWASP recommendations). HTTP cookie session tracking is required to be supported by all servlet containers. The container generates and sends a cookie to the client. The client will then return the

cookie on each subsequent request to the server, unambiguously associating the request with a session. The name of the session tracking cookie must be **JSESSIONID**. To service HTTP requests from clients that do not support or disable the use of cookies web containers commonly support the **URL rewriting** mechanism.

# Java Server Pages (JSP)

<https://www.journaldev.com/2110/jsp-interview-questions-and-answers>

**► What is JSP and why do we need it?**

JSP stands for JavaServer Pages. JSP is java server side technology to create dynamic web pages. JSP is extension of Servlet technology to help developers create dynamic pages with HTML like syntax. Containers support **hot deployment** of JSP pages.

**► What are the JSP lifecycle phases / methods?**

* Translation
* Compilation
* Class loading
* Instantiation
* Initialization – jspInit()
* Request processing – jspService()
* Destroy – jspDestroy()

**► Which JSP lifecycle methods can be overridden?**

We can override jspInit() and jspDestroy() methods

**► How can we avoid direct access of JSP pages from client browser?**

Anything inside WEB-INF directory can’t be accessed directly in web application, so we can place our JSP pages in WEB-INF directory to avoid direct access to JSP page from client browser. In this case, we will have to configure it in deployment descriptor just like Servlets.

**► What are different types of comments in JSP?**

JSP pages provide two types of comments that we can use:

* **HTML Comments**: Since JSP pages are like HTML, we can use HTML comments like <!-- HTML Comment -->. These comments are sent to client also and we can see it in HTML source.
* **JSP Comments**: JSP Comments are written using scriptlets like <%-- JSP Comment --%>. These comments are present in the generated servlet source code and doesn't sent to client.

**► What is Scriptlet, Expression and Declaration in JSP?**

Scriptlets, Expression and Declaration are scripting elements in JSP page using which we can add java code in the JSP pages. A scriptlet tag starts with <% and ends with %>. Any code written inside the scriptlet tags go into the \_jspService() method. For example;

JSP Expression starts with <%= and ends with %>.

JSP Declarations are used to declare member methods and variables of servlet class. JSP Declarations starts with <%! and ends with %>.

**►► What are JSP implicit objects?**

JSP implicit objects are created by container while translating JSP page to Servlet source to help developers. We can use these objects directly in scriptlets that goes in service method, however we can't use them in JSP Declaration because that code will go at class level. We have 9 implicit objects that we can directly use in JSP page. Seven of them are declared as local variable at the start of \_jspService() method whereas two of them are part of \_jspService() method argument that we can use:

out - object that writes to output stream, page scope

request - incoming request, request scope

response - response, page scope

config - servlet config, page scope

application - servlet context, application scope

session - HTTP session object

pageContext - page context is container for page objects with convenience methods, page scope

page - instance of implementation class for this page, page scope

exception - uncaught throwable that resulted in this error page invoked, page scope.

**► Can we use JSP implicit objects in a method defined in JSP Declaration?**

No, we can't because JSP implicit objects are local to **service()** method and added by JSP Container while translating JSP page to servlet source code. JSP Declarations code goes outside the service method and used to create class level variables and methods and hence can't use JSP implicit objects.

**► Which implicit object is not available in normal JSP pages?**

JSP **exception** implicit object is not available in normal JSP pages and it's used in JSP error pages only

**► What are the benefits of PageContext implicit object?**

JSP pageContext implicit object is instance of javax.servlet.jsp.PageContext abstract class implementation. We can use pageContext to get and set attributes with different scopes and to forward request to other resources. pageContext object also hold reference to other implicit object. This is the only object that is common in both JSP implicit objects and in JSP EL implicit objects.

**► How do we configure init params for JSP?**

We can configure init params for JSP similar to servlet in web.xml file, we need to configure JSP init params with servlet and servlet-mapping element.

**► Why use of scripting elements in JSP is discouraged? Can we define a class in a JSP Page?**

JSP pages are mostly used for view purposes and all the business logic should be in the servlet or model classes. We should pass parameters to JSP page through attributes and then use them to create the HTML response in JSP page. we can define a class inside a JSP Page.

**► How can we disable java code or scripting in JSP page?**

We can disable scripting elements in JSP pages through deployment descriptor configuration

<scripting-invalid>true</scripting-invalid>

**► Explain JSP Action Elements or Action Tags?**

JSP action elements or action tags are HTML like tags that provide useful functionalities such as working with Java Bean, including a resource, forwarding the request and to generate dynamic XML elements. JSP action elements always starts with jsp: and we can use them in JSP page directly without the need to import any tag libraries or any other configuration changes. Some of the important action elements are jsp:useBean, jsp:getProperty, jsp:setProperty, jsp:include and jsp:forward.

**►►► What is the difference between include directive and include action?**

The primary difference between JSP include **directive** and include **action** tag is that in include directive the content of another resource is added at the **translation** time whereas with include action it happens at **runtime**.

As a result, if included file changes at runtime, **include directive** will not notice it (unless including JSP is also changed and include directive can’t accept runtime parameters. E.g. copyright text is usually stable but changing year could become an issue:

<%@ include file="copyright.jsp" %>

Include action is more flexible by including content dynamically for each request and accepts runtime parameters. If included content is large, it could slow down performance. E.g. we can pass year as parameter and for most of the year inserted copyright text will be the same, while taking some runtime from each request:

<jsp:include page=" copyright.jsp " />

<jsp:param name="year" value="2018" />

</jsp:include>

**► How can we handle exceptions thrown by JSP service method?**

To handle exceptions thrown by the JSP page, all we need is an error page and define the error page in JSP using page directive.

**► How can we prevent implicit session creation in JSP?**

By default JSP page creates a session. To disable the session creation, we can use it like below.

<%@ page session="false" %>

**► What is difference between JspWriter and Servlet PrintWriter?**

PrintWriter is the actual object responsible for writing the content in response. JspWriter uses the PrintWriter object behind the scene and provide buffer support. When the buffer is full or flushed, JspWriter uses the PrintWriter object to write the content into response.

# Expression Language (EL)

**► What is JSP Expression Language and what are its benefits?**

For web designers, java code is hard to understand and that’s why JSP Specs 2.0 introduced Expression Language (EL) through which we can get attributes and parameters easily using HTML like tags. EL syntax is ${name} and we can use EL implicit objects and EL operators to retrieve the attributes from different scopes and use them in JSP page.

**► What are JSP EL implicit objects and how it's different from JSP implicit Objects?**

JSP EL provides many implicit objects that we can use to get attributes from different scopes and parameter values. Note that these are different from JSP implicit objects and contains only the attributes in given scope. The only common implicit object in JSP EL and JSP page is pageContext object.

**► How to use JSP EL to get HTTP method name?**

We can use pageContext JSP EL implicit object to get the request object reference and use dot operator to get the HTTP method name in JSP page. The JSP EL code for this will be ${pageContext.request.method}.

# Standard Tag Library (JSTL)

**► What is JSP Standard Tag Library (JSTL), provide some example usage?**

JSP Standard Tag Library or JSTL is more versatile than JSP EL or Action elements because we can loop through a collection or escape HTML tags to show them like text in response.

* Core Tags provide support for iteration, conditional logic, catch exception, url, forward or redirect response etc.
* Formatting and Localization Tags - for formatting of Numbers, Dates and i18n support through locales and resource bundles.
* SQL Tags - support for interaction with relational databases such as Oracle, MySql etc.
* XML Tags - to work with XML documents such as parsing XML, transforming XML data and XPath expressions evaluation.
* JSTL Functions Tags

**► Give an example where you need JSP Custom Tag?**

Let’s say we want to show a number with formatting with commas and spaces. This can be very useful for user when the number is really long. So we want some custom tags like below:

<mytags:formatNumber number="123456.789" format="#,###.00"/>

**► How do we catch exception and process it using JSTL?**

We can use JSTL Core tags c:catch and c:if to catch exception inside the JSP service method and process it. c:catch tag catches the exception and wraps it into the exception variable and we can use c:if condition tag to process it. Below code snippet provide sample usage.

<c:catch var ="exception">

<% int x = 5/0;%>

</c:catch>

<c:if test = "${exception ne null}">

<p>Exception is : ${exception} <br />

Exception Message: ${exception.message}</p>

</c:if>

# Java Server Faces (JSF)

**► What Is a JSF?**

JavaServer Faces (JSF) is a user interface **(UI)**, **event driven framework** for Java web app. It is designed to significantly ease writing and maintaining applications that run on a Java app server and render their UIs back to a target client. It helps manage **UI state** across server requests, provides a simple model for wiring **client-generated events** to server-side application code, allows custom UI components to be easily built and re-used.

**► What are the available implementations of JSF?**

The main implementations of JavaServer Faces are:

* Apache MyFaces is an open source implementation.
* ADF Faces is Oracle’s implementation.

**► What is view object / backing bean?**

A view object is a **model object** used specifically in the presentation tier. It contains the data that must display in the view layer and the logic **to validate user input, handle events, and interact with the business logic** tier. The **backing bean** is the view object in a JSF-based application. Backing bean and view object are interchangeable terms. Backing bean management separates the definition of **UI component** objects from objects that perform application-specific processing and hold data.

**► What is Managed Bean?**

JavaBean objects managed by a JSF implementation are called **managed beans**. A managed bean describes how a bean is created and managed.

**► Explain the meaning of components in JSF and what are the different types?**

Components in JSF are objects that manage interaction with a user. Developers create UIs by assembling a number of components, associating them with object properties and event handlers. The different components in JSF are elements like text box, button, table etc. that are used to create user interfaces of JSF Applications. Components in JSF are of two types:

* Simple Components like text box, button
* Compound Components like table, data grid etc.

**► Explain the way by which the components of JSF are rendered?**

JSF libraries need to be added in an application. On the .jsp page, a tag libraries needs to be added:

<%@ taglib uri=”http://java.sun.com/**jsf/core**” prefix=”**f**”%>

<%@ taglib uri=”http://java.sun.com/**jsf/html**” prefix=”**h**”%>

**► When is automatic conversion supplied by JSF Implementation?**

JSF implementation automatically converts component data between **presentation view** and **model** when the bean property associated with the component is of one of the types supported by the component’s data. For example, if a UISelectBoolean component is associated with a bean property of type Boolean, then JSF implementation will automatically convert the data **from String to Boolean**.

**► What do you mean by a ‘Render Kit’ in JSF?**

Component classes generally transfer the task of generating output to the renderer. All JSF components follow it. Render kit is a set of related renderers. javax.faces.render.RenderKit is the class which represents the render kit. The default render kit contains renderers for **html** but it’s up to you to make it for other markup languages.

**► What do you mean by rendering of page in JSF?**

Every JSF page as described has various components made with the help of JSF library. JSF may contain h:form, h:inputText, h:commandButton, etc. Each of these is rendered (translated) to **HTML output**. This process is called **encoding**. The encoding procedure also assigns each component with a **unique ID** assigned by framework. The ID generated is random.

**► What does a typical JSF application consists of?**

A typical JSF application consists of the following parts:

* JSF configuration file
* JavaBeans components for managing application state and behavior.
* Event-driven development (via listeners as in traditional GUI development).
* Pages that represent MVC-style views; pages reference view roots via the JSF component tree.

**► Explain the JSF life-cycle phases?**

The phases that are part of the JSF application lifecycle are:

* **Restore View:** A request comes through the FacesServlet controller. The controller examines the request and extracts the view ID, which is determined by the name of the JSP page.
* **Apply request values**: For each component retrieve its current state. The components must first be retrieved or created from the FacesContext object, followed by their values.
* **Process validations**: Each component will have its values validated against the application’s validation rules.
* **Update model values**: Updates the actual values of the server-side model by updating the properties of backing beans.
* **Invoke application**: JSF controller invokes the application to handle Form submissions.
* **Render response**: JSF displays the view with all of its components in their current state.

**► Explain how you can declare managed beans?**

The bean instance is configured in the **faces-config.xml** file:

<managed-bean>

<managed-bean-name>login</managed-bean-name>

<managed-bean-class>com.j2eebrainJsf.loginBean</managed-bean-class>

<managed-bean-scope>request</managed-bean-scope>

</managed-bean>

Another way is to use annotations:

@ManagedBean

@SessionScoped

public class HelloBean implements Serializable

**► Explain the bean scopes in JSF?**

JSF supports three bean scopes:

* **Request** Scope - dies of when the response is sent back to the requestor.
* **Session** Scope - is available till the session is ended.
* **Application** Scope - for the entire duration of the web application.

**► Explain the main tags that are used in JSF?**

In JSF applications the JSP pages are used to represent the views. JSF provides 43 tags in two standard JSF tag libraries:

* JSF Core Tags Library.
* JSF Html Tags Library.

**► Explain the different types of JSF events?**

JSF is an **event driven** framework.

* **Action Events**: bound to UI Command objects like a Command Button or a Hyper-link. Whenever a user presses a Command Button or clicks a hyperlink these Events get generated.
* **Value Change Events**: bound to UI Components like Text Field, Check-Box, List and Radio Buttons. The Value Change Event is fired as soon as the value that is displayed in the view is modified.
* **Phase Events**: As you saw earlier in the JSF overview blog, the request processing life-cycle in JSF includes six phases and any JSF implementation will fire Phase events during the start and end of each phase. If we want to capture the Phase Events, then can define a Phase Listener. These are handy for debugging as well.

**► Explain the way by which the events handled in JSF? What is the difference between these event handling mechanisms?**

Action handlers and event listeners provide an event driven mechanism. Every time a user does something like clicking a button, selecting an item from a drop down, or submitting a form, an event occurs. Event notification is then **sent via HTTP to the server and handled by the FacesServlet**. Events can invoke custom business logic or initiate page navigation.

JSF provides two types of methods for handling events; **listeners** and **action handlers**, may be defined within a managed bean. A listener takes an FacesEvent as a parameter and a void return type, while an action handler takes no parameters and returns a String.

**► Explain viewstate in JSF?**

Viewstate is associated with each page, which is **passed back and forth with each submits**. The reason for the viewtate is that the **HTTP is a stateless protocol**. The state of the components across requests need to be maintained.

**► How to add context path to URL for outputLink?**

Current JSF implementation does not add the context path for outputLink if the defined path starts with ‘/’. To correct this problem use:

#{facesContext.externalContext.requestContextPath}

prefix at the beginning of the outputLink value attribute. For example:

<h:outputLink value=”#{facesContext.**externalContext**.requestContextPath}/myPage.faces”>

**► How to get current page URL from backing bean?**

You can get a reference to the HTTP request object via FacesContext like this:

FacesContext fc = FacesContext.getCurrentInstance();

HttpServletRequest request = (HttpServletRequest) fc.**getExternalContext**().getRequest();

and then use the normal request methods to obtain path information. Alternatively,

context.getViewRoot().getViewId();

will return you the name of the current JSP (JSF view IDs are basically just JSP path names)

**► How to access web.xml init parameters from jsp page?**

You can get it using **initParam** pre-defined **JSF EL** valiable. For example, if you have:

<context-param>

<param-name>productId</param-name> <param-value>2004Q4</param-value>

</context-param>

You can access this parameter with #{initParam[‘productId’]}. For example:

Product Id: <h:outputText value=”#{initParam[‘productId’]}”/>

**► How to access web.xml init parameters from java code?**

You can get it using externalContext getInitParameter method. For example

FacesContext fc = FacesContext.getCurrentInstance();

String connection = fc.**getExternalContext**().getInitParameter("connectionString");

**► How to terminate the session?**

In order to terminate the session you can use session **invalidate**() method. E.g. action method of a **backing bean**:

public String logout() {

FacesContext fc = FacesContext.getCurrentInstance();

HttpSession session = (HttpSession) fc.getExternalContext().getSession(false);

session.invalidate();

return "login\_page";

}

**► How to implement “Please, Wait…” page?**

The client-side solution might be very simple. Wrap the jsp page (or part of it you want to hide) into the DIV, then you can add one more DIV that appears when user clicks the submit button. This DIV can contain the animated gif you speak about or any other content.

**► Is it possible to have more than one Faces configuration file?**

Yes. You can define the **list of the configuration files** in the web.xml.

<context-param>

<param-name>**javax.faces.CONFIG\_FILES**</param-name>

<param-value>/WEB-INF/faces-config-navigation.xml,/WEB-INF/faces-beans.xml</param-value>

</context-param>

**► How to mask actual URL to the JSF page?**

Implement your own version of javax.faces.**ViewHandler** and register that view handler in **faces-config.xml**.

**► How to reload the page after ValueChangeListener is invoked?**

At the end of the ValueChangeListener, call FacesContext.getCurrentInstance().**renderResponse**()

**► How to download PDF file (with JSF)?**

It can be done with action listener of the backing bean. Load pdf file on server, open faces response as ServletOutputStream, set content type as "application/pdf", write pdf to the output:

public void viewPdf(ActionEvent event) {

// use your own method that reads file to the byte array

…

FacesContext faces = FacesContext.getCurrentInstance();

HttpServletResponse response = (HttpServletResponse) faces.getExternalContext().getResponse();

response.setContentType("application/pdf");

response.setContentLength(pdf.length);

response.setHeader( "Content-disposition", "inline; filename=\""+fileName+"\"");

// write pdf

faces.responseComplete();

}

This is a jsp file snippet:

<h:commandButton immediate=”true” actionListener=”**#{backingBean.viewPdf}”** value=”Read PDF” />

**► What is the different between getRequestParameterMap() and getRequestParameterValuesMap() ?**

getRequestParameterValuesMap() similar to getRequestParameterMap(), but contains **multiple values** for the parameters with the same name. It is important if you use one of the components such as <h:selectMany>.

# Enterprise Java Beans (EJB)

**► EJB 1.1 features?**

The EJB 1.1 Specification provided EJB clients with **Java RMI** interfaces - a remote interface (javax.ejb.EJBObject) and a remote home interface (javax.ejb.EJBHome) to interact with EJB instances, achieving location transparency.Remote clients communicate with EJBs using interfaces and arguments that are compatible with Java RMI-IIOP. Arguments and method results are passed by value within the same container or across the network to remote containers.

**► EJB 2.0 features?**

EJB 2.0 provided EJB clients with a local interface and local home interface to interact with EJB instances that share the same JVM. The EJB local interface (javax.ejb.EJBLocalObject) and local home interface (javax.ejb.EJBLocalHome) are **not Java RMI** interfaces and are new to EJB 2.0. Local clients—such as other EJBs—communicate with EJBs directly using pass-by-reference arguments in the same JVM. This technique eliminates network latency issues, argument copying and the need to comply with Java RMI-IIOP. It also means that EJB clients can access services using a more lightweight programming model. This approach is well suited for fine-grained method calls.

**► EJB 3.0 features?**

EJB 3.0 is part of **Java EE 5** (**2006**). One of the major changes was replacement of **Entity EJBs** with lightweight persistent domain objects of **Java Persistence API (JPA)** which was heavily influenced by **Hibernate.**

EJB 3.0 beans now include:

* Stateless Session beans
* Stateful Session beans
* Message Driven beans (MDB)