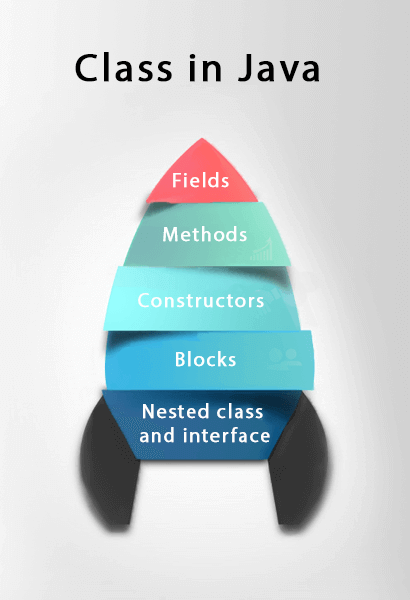
**Java OOP**

**Releases, Versions and Features**

<https://www.marcobehler.com/guides/a-guide-to-java-versions-and-features>

<https://howtodoinjava.com/java-version-wise-features-history/>

<https://www.oracle.com/technetwork/java/javase/jdk-relnotes-index-2162236.html>

Object – instance of a class that has identity, state and behavior, Object is the root class of all classes in Java

1. **Declaration**: The code set in **bold** are all variable declarations that associate a variable name with an object type.
2. **Instantiation**: The new keyword is a Java operator that creates the object.
3. **Initialization**: The new operator is followed by a call to a constructor, which initializes the new object.

**Point originOne** = new Point(23, 94);

**Memory** <https://www.baeldung.com/java-stack-heap>

1. **Stack Memory**

* Used for static memory allocation and the execution of a thread
* Contains primitive values specific to method and references to objects in heap
* When a new method is called, a new block on top of the stack is created which contains values specific to that method
* Access faster than heap and threadsafe
* StackOverFlowError – memory is full

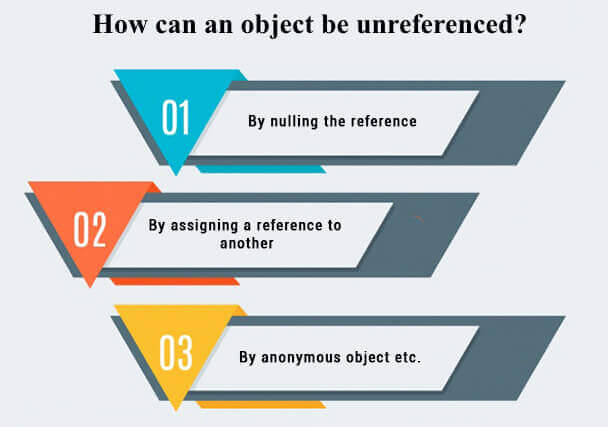
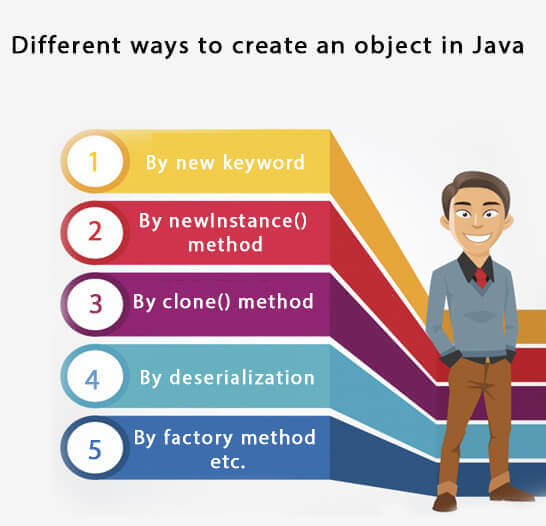
1. **Heap Space**

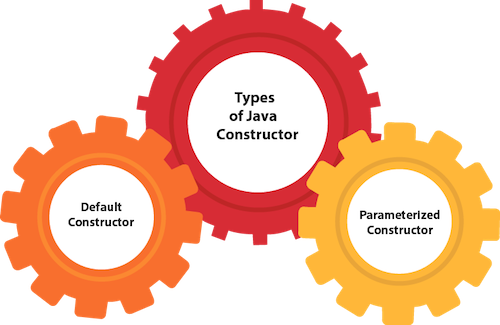
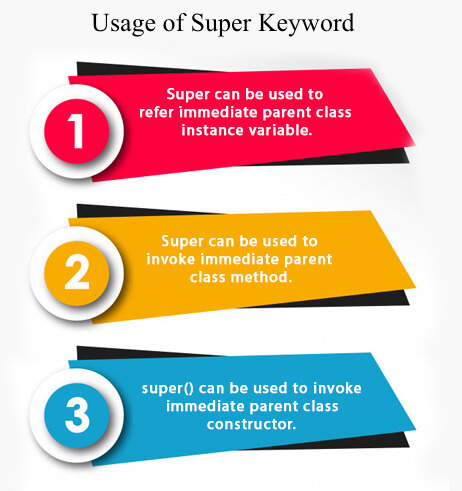
* Used for dynamic memory allocation for Java objects and runtime classes
* New objects are always created in heap space and references stored in stack
* OutOfMemoryError – heap space is full
* Slower, memory isn’t automatically deallocated, not threadsafe

****

**Garbage Collection**

Some object-oriented languages require that you keep track of all the objects you create and that you explicitly destroy them when they are no longer needed. Managing memory explicitly is tedious and error-prone. The Java platform allows you to create as many objects as you want (limited, of course, by what your system can handle), and you don't have to worry about destroying them. The Java runtime environment **deletes objects when it determines that they are no longer being used**. This process is called *garbage collection*. An object is eligible for garbage collection when there are no more references to that object. References that are held in a variable are usually dropped when the variable goes out of scope. Or, you can explicitly drop an object reference by setting the variable to the special value null.

**Variable types – primitive(8) vs reference**

**4 Variables:**

**Instance variables** are created inside the class but outside the method. Instance variable doesn't get memory at compile time. It gets memory at runtime when an object or instance is created.

**Class** (static)

**Local** (method)

**Parameter** ie: public static void main(String[] args)

***literal*** is the source code representation of a fixed value

**Data Type      Possible Literal**

* int                   33 (0)
* byte                 6 (0)
* short               94 (0)
* long                53092437L (0L)
* char                'c'
* boolean          true or (false)
* float                1.053f (0.0f)
* double            3.03532453

**Operator**

Assignment (=)

Arithmetic (+, -, \*, /, %)

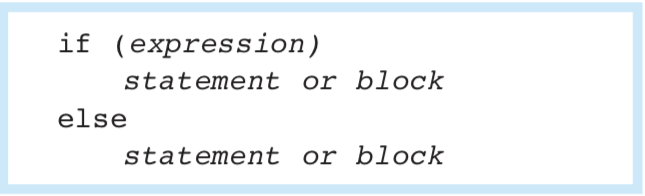
Unary (+, -, ++, --, !)

Equality/Relational (!=, ==, >, <. >=. <=)

Conditional (&&, ||)

Type Comparison (instanceof)

Bitwise, Bit Shift



Expression (ie: 1\*2)

Statement (ie: int sum = 1\*2)

Block – zero or more statements surrounded by {}

Branching Statements

**Break** – exit loop

**Continue** - continue

**Return** – exit method

You can use a construct called ***varargs*** to pass an arbitrary number of values to a method. You use varargs when you don't know how many of a particular type of argument will be passed to the method. It's a shortcut to creating an array manually (the previous method could have used varargs rather than an array).

public Polygon polygonFrom(Point... corners) {}

**Instance Initializer block** is used to initialize the instance data member. It run each time when object of the class is created. Invoked at the time of object creation. The runtime system guarantees that *static initialization blocks are called in the order that they appear* in the source code.

**class** Bike7{

**int** speed;

    Bike7(){System.out.println("speed is "+speed);}

    {speed=100;}

**public** **static** **void** main(String args[]){

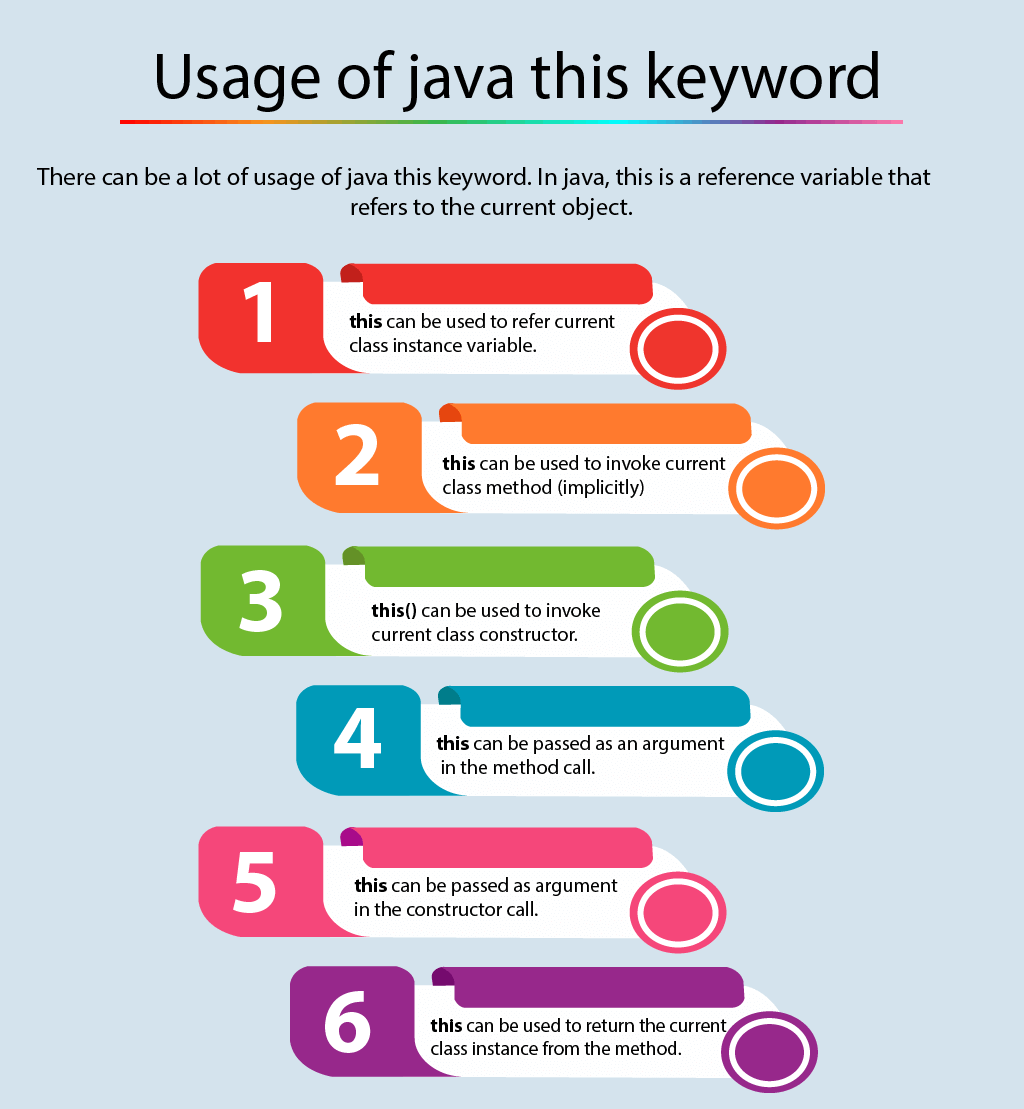
    Bike7 b1=**new** Bike7();  // prints speed is 100

    Bike7 b2=**new** Bike7();  // prints speed is 100

    }

}

**this** is a **reference variable** that refers to the current object



**Java does NOT support multiple inheritance** (ie: class C extends A, B where A and B have the same method – compile time error. Use multiple interfaces instead)

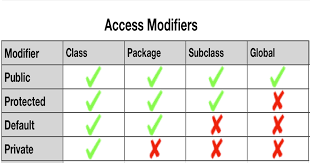


**Encapsulation** – binding code/data into single unit accessible through getters/setters, modifiers

**Packages** – built in and user defined; help avoid conflicts with same class names defined in different packages

Modifiers- access and non

There are many non-access modifiers, such as static, abstract, synchronized, native, volatile, transient, etc. Here, we are going to learn the access modifiers only.



Other modifiers:

* **Static** - Static fields or methods are class members, whereas non-static ones are object members. Class members don't need any instance to be invoked; You can not have a TOP level static class; you can have static nested classes
* **Final** – (variable can NOT be changed, class can NOT be extended and method can NOT be overridden), blank final variable can only be initialized in constructor
* **Abstract** – can't be instantiated. Instead, they are meant to be subclassed
* **Synchronized** – can use it with the instance as well as with static methods and code blocks. When we use this keyword, we make Java use a monitor lock to provide synchronization on a given code fragment.
* **Volatile** - can only use it together with instance fields. It declares that the field value must be read from and written to main memory – bypassing the CPU cache. All reads and writes for a volatile variable are atomic.

**class** Bike10{

**final** **int** speedlimit;//blank final variable

  Bike10(){

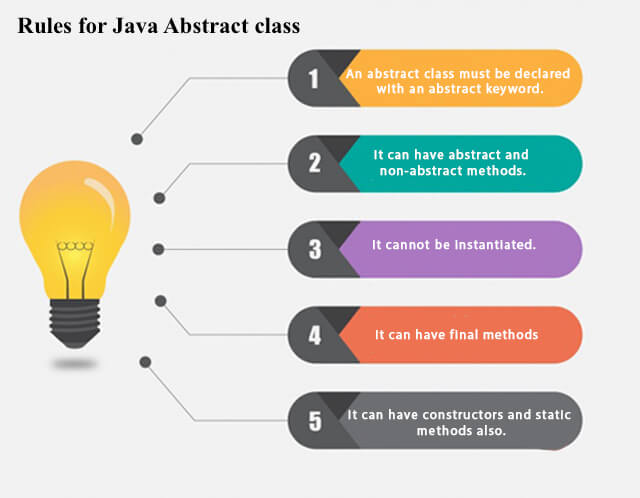
  speedlimit=70;

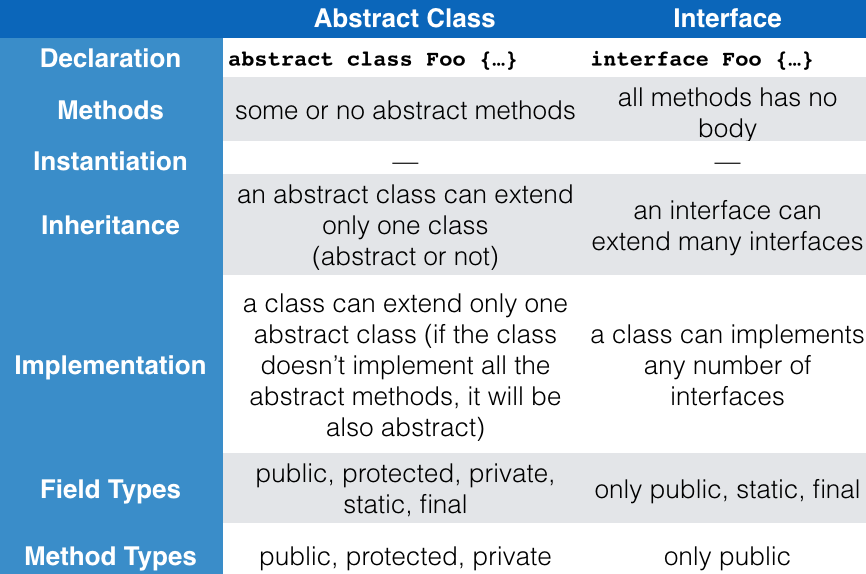
  System.out.println(speedlimit);

  }

}

**Abstraction** – hiding implementation and showing functionality. Abstraction enables you to focus on what the object does instead of how it does it. Two ways – interface and abstract class.





**Abstract class vs Interface**

1. **Type of methods:** Interface can have only abstract methods. Abstract class can have abstract and non-abstract methods. From Java 8, it can have default and static methods also.
2. **Final Variables:** Variables declared in a Java interface are by default final. An abstract class may contain non-final variables.
3. **Type of variables:**Abstract class can have final, non-final, static and non-static variables. Interface has only static and final variables.
4. **Implementation:** Abstract class can provide the implementation of interface. Interface can’t provide the implementation of abstract class.
5. **Inheritance vs Abstraction:** A Java interface can be implemented using keyword “implements” and abstract class can be extended using keyword “extends”.
6. **Multiple implementation:** An interface can extend another Java interface only, an abstract class can extend another Java class and implement multiple Java interfaces.
7. **Accessibility of Data Members:** Members of a Java interface are public by default. A Java abstract class can have class members like private, protected, etc.

Reasons to choose Interface



**abstract** **class** Bank{

**abstract** **int** getRateOfInterest();

}

**class** SBI **extends** Bank{

**int** getRateOfInterest(){**return** 7;}

}

**class** PNB **extends** Bank{

**int** getRateOfInterest(){**return** 8;}

}

**abstract** **class** Bike{

   Bike(){System.out.println("bike is created");}

**abstract** **void** run();

**void** changeGear(){System.out.println("gear changed");}

 }

//Creating a Child class which inherits Abstract class

**class** Honda **extends** Bike{

**void** run(){System.out.println("running safely..");}

 }

//Creating a Test class which calls abstract and non-abstract methods

**class** TestAbstraction2{

**public** **static** **void** main(String args[]){

  Bike obj = **new** Honda();

  obj.run();

  obj.changeGear();

 }

}

**interface** Printable{

**void** print();

}

**interface** Showable{

**void** show();

}

**class** A7 **implements** Printable,Showable{

**public** **void** print(){System.out.println("Hello");}

**public** **void** show(){System.out.println("Welcome");}

**public** **static** **void** main(String args[]){

A7 obj = **new** A7();

obj.print();

obj.show();

 }

}

**Polymorphism** – one task performed in different ways (method overriding and overloading), **runtime** and **compiletime**; Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.

**Method overloading** – change number of arguments OR change data types

**Method overriding** – subclass with same method as parent class (can’t override static method)

**Covariant return type** – method overriding where return type is different. Below example A’s get returns A while B’s get returns B

**class** A{

A get(){**return** **this**;}

}

**class** B1 **extends** A{

B1 get(){**return** **this**;}

**void** message(){System.out.println("welcome to covariant return type");}

**public** **static** **void** main(String args[]){

**new** B1().get().message();

}

}

**Runtime polymorphism** or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

A method is overridden, **not the data members**, so runtime polymorphism can't be achieved by data members.

**Upcasting and Downcasting**

<https://stackoverflow.com/questions/23414090/what-is-the-difference-between-up-casting-and-down-casting-with-respect-to-class>

Upcasting is casting to a supertype, while downcasting is casting to a subtype. Upcasting is always allowed, but downcasting involves a type check and can throw a ClassCastException. **Upcasting is done automatically but downcasting needs to be done manually by programmer.**

A cast from a Dog to an Animal is an upcast, because a Dog is-a Animal. In general, you can upcast whenever there is an is-a relationship between two classes.

Downcasting would be something like this:

Animal animal = new Dog(); //upcast

Dog castedDog = (Dog) animal; // downcast

Basically what you're doing is telling the compiler that you know what the runtime type of the object *really* is. The compiler will allow the conversion, but will still insert a runtime sanity check to make sure that the conversion makes sense. In this case, the cast is possible because at runtime animal is actually a Dog even though the static type of animal is Animal.

However, if you were to do this:

Animal animal = new Animal();

Dog notADog = (Dog) animal;

You'd get a ClassCastException. The reason why is because animal's runtime type is Animal, and so when you tell the runtime to perform the cast it sees that animal isn't really a Dog and so throws a ClassCastException.

To call a superclass's method you can do super.method() or by performing the upcast.

To call a subclass's method you have to do a downcast. As shown above, you normally risk a ClassCastException by doing this; however, you can use the instanceof operator to check the runtime type of the object before performing the cast, which allows you to prevent ClassCastExceptions:

Animal animal = getAnimal(); // Maybe a Dog? Maybe a Cat? Maybe an Animal?

if (animal instanceof Dog) {

// Guaranteed to succeed, barring classloader shenanigans

Dog castedDog = (Dog) animal;

}

**class** Animal{

**void** eat(){System.out.println("animal is eating...");}

}

**class** Dog **extends** Animal{

**void** eat(){System.out.println("dog is eating...");}

}

**class** BabyDog1 **extends** Dog{

**public** **static** **void** main(String args[]){

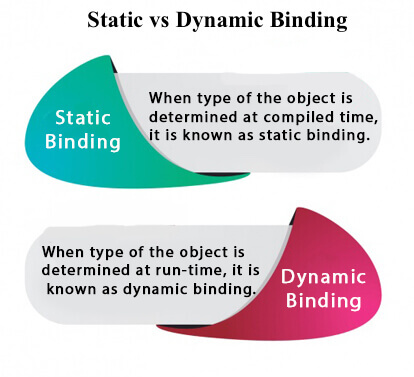
Animal a=**new** BabyDog1();

a.eat();

}}

// prints dog is eating bc BabyDog is not overriding the eat() method, so eat() method of Dog class is invoked

|  |  |  |
| --- | --- | --- |
|  | **compile-time polymorphism** | **Runtime polymorphism** |
| 1 | In compile-time polymorphism, call to a method is resolved at compile-time. | In runtime polymorphism, call to an overridden method is resolved at runtime. |
| 2 | It is also known as static binding, early binding, or overloading. | It is also known as dynamic binding, late binding, overriding, or dynamic method dispatch. |
| 3 | Overloading is a way to achieve compile-time polymorphism in which, we can define multiple methods or constructors with different signatures. | Overriding is a way to achieve runtime polymorphism in which, we can redefine some particular method or variable in the derived class. By using overriding, we can give some specific implementation to the base class properties in the derived class. |
| 4 | It provides fast execution because the type of an object is determined at compile-time. | It provides slower execution as compare to compile-time because the type of an object is determined at run-time. |
| 5 | Compile-time polymorphism provides less flexibility because all the things are resolved at compile-time. | Run-time polymorphism provides more flexibility because all the things are resolved at runtime. |



If there is any private, final or static method in a class, there is static binding.

**class** Dog{

**private** **void** eat(){System.out.println("dog is eating...");}

**public** **static** **void** main(String args[]){

  Dog d1=**new** Dog();

  d1.eat();

 }

}

In the below example object type cannot be determined by the compiler, because the instance of Dog is also an instance of Animal.So compiler doesn't know its type, only its base type.

**class** Animal{

**void** eat(){System.out.println("animal is eating...");}

}

**class** Dog **extends** Animal{

**void** eat(){System.out.println("dog is eating...");}

**public** **static** **void** main(String args[]){

  Animal a=**new** Dog();

  a.eat();

 }

}

The **java instanceof operator** is used to test whether the object is an instance of the specified type (class or subclass or interface).

**class** Simple1{

**public** **static** **void** main(String args[]){

 Simple1 s=**new** Simple1();

 System.out.println(s **instanceof** Simple1);//true

 }

}

**interface** Printable{}

**class** A **implements** Printable{

**public** **void** a(){System.out.println("a method");}

}

**class** B **implements** Printable{

**public** **void** b(){System.out.println("b method");}

}

**class** Call{

**void** invoke(Printable p){//upcasting

**if**(p **instanceof** A){

A a=(A)p;//Downcasting

a.a();

}

**if**(p **instanceof** B){

B b=(B)p;//Downcasting

b.b();

}

}

}//end of Call class

**class** Test4{

**public** **static** **void** main(String args[]){

Printable p=**new** B();

Call c=**new** Call();

c.invoke(p);  // outputs ‘b method’

}

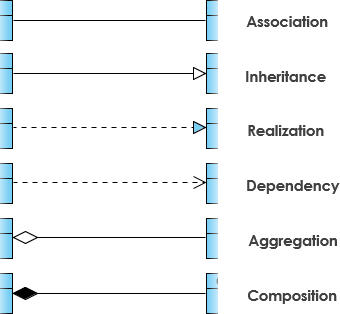
}

**Inheritance** – when one object acquires all of the properties and behaviors of its parent object

**Coupling** - Coupling refers to the knowledge or information or dependency of another class. It arises when classes are aware of each other. If a class has the details information of another class, there is strong coupling. In Java, we use private, protected, and public modifiers to display the visibility level of a class, method, and field. You can use interfaces for the weaker coupling because there is no concrete implementation.



**Cohesion** - Cohesion refers to the level of a component which performs a single well-defined task. A single well-defined task is done by a highly cohesive method. The weakly cohesive method will split the task into separate parts. The java.io package is a highly cohesive package because it has I/O related classes and interface. However, the java.util package is a weakly cohesive package because it has unrelated classes and interfaces.



public class A {  
 private B; // Association  
 private C; // Association  
   
 public void setB(B b) {  
 B = b; // Aggregation  
 }  
   
 public void setC() {  
 C = new C(); // Composition  
 }  
   
 public void useD(D d) {  
 d.callMethod(); // Dependency  
 }  
}



**Association** - Association represents the relationship between the objects. Here, one object can be associated with one object or many objects. There can be four types of association between the objects:

* **One to One**
* **One to Many**
* **Many to One**
* **Many to Many**

Let's understand the relationship with real-time examples. For example, One country can have one prime minister (one to one), and a prime minister can have many ministers (one to many). Also, many MP's can have one prime minister (many to one), and many ministers can have many departments (many to many).

Association can be undirectional or bidirectional.

**Aggregation** - Aggregation is a way to achieve Association. Aggregation represents the relationship where one object contains other objects as a part of its state. It represents the weak relationship between objects. It is also termed as a ***has-a*** relationship in Java. Like, inheritance represents the *is-a* relationship. It is another way to reuse objects.

**Composition** - The composition is also a way to achieve Association. The composition represents the relationship where one object contains other objects as a part of its state. There is a strong relationship between the containing object and the dependent object. It is the state where containing objects do not have an independent existence. If you delete the parent object, all the child objects will be deleted automatically.

Dependency (local scope) vs Association (class scope)

Association --> **A *has-a* C** object (as a member variable)

Dependency --> **A *references* B** (as a method parameter or return type)

public class A {

private C c;

public void myMethod(B b) {

b.callMethod();

}

}

An association is a strong (static) dependency. Aggregation and Composition are even stronger. Both types of association.

Composition – you create an object of a class inside another class

public class A {

B b;

public void setB(){

this.b= new B();

}

}

Aggregation – weaker type of association between 2 objects

public class A {

B b;

public void setB(B b\_ref){

this.b= b\_ref;

/\* object B is passed as an argument of a method \*/

}

}

*Autoboxing* is the automatic conversion that the Java compiler makes between the primitive types and their corresponding object wrapper classes. For example, converting an int to an Integer, a double to a Double, and so on. If the conversion goes the other way, this is called *unboxing*.

**(Auto)boxing** (value to obj) vs **Unboxing**

Integer number = 100; // autoboxing

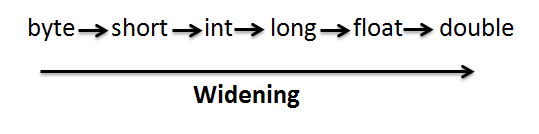
int inum = number; // unboxing

|  |  |
| --- | --- |
| **Primitive type** | **Wrapper class** |
| boolean | Boolean |
| byte | Byte |
| char | Character |
| float | Float |
| int | Integer |
| long | Long |
| short | Short |
| double | Double |

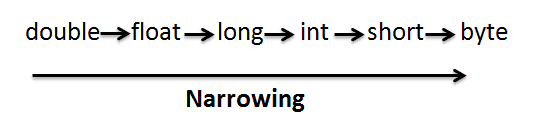
Casting: **Implicit (widen)** vs **Explicit (narrow)** Conversion

**Type Casting** in Java is nothing but converting a primitive, interface or class into another type.

Implicit – safe, no risk of losing data



Explicit – it is likely conversion could lose data



<https://javainterviewpoint.com/type-casting-java-implicit-explicit-casting/>

An ***enum*** *type* is a special data type that enables for a variable to be a set of predefined constants. The variable must be equal to one of the values that have been predefined for it. Common examples include compass directions (values of NORTH, SOUTH, EAST, and WEST) and the days of the week.

\*Enums are thread safe and popularly used in Singletons.

**Nested Class** – **Static** and **Non Static/Inner**

**Why use Nested Classes?**

They’re a way to logically group classes only used in one place, increases encapsulation, more readable/useable code. Use them if your requirements are similar to those of a local class, you want to make the type more widely available and you don’t require access to local variables or method parameters

Non-static nested classes (inner classes) have access to other members of the enclosing class, even if they are declared private. Static nested classes do not have access to other members of the enclosing class.

An instance of an Inner Class can exist only within an instance of the outer class.

There are 2 types of Inner Classes – **Local** and **Anonymous**

**Shadowing**

If a declaration of a type (such as a member variable or a parameter name) in a particular scope (such as an inner class or a method definition) has the same name as another declaration in the enclosing scope, then the declaration *shadows* the declaration of the enclosing scope. **You cannot refer to a shadowed declaration by its name alone**.

public class ShadowTest {

public int x = 0;

class FirstLevel {

public int x = 1;

void methodInFirstLevel(int x) {

System.out.println("x = " + x);

System.out.println("this.x = " + this.x);

System.out.println("ShadowTest.this.x = " + ShadowTest.this.x);

}

}

public static void main(String... args) {

ShadowTest st = new ShadowTest();

ShadowTest.FirstLevel fl = st.new FirstLevel();

fl.methodInFirstLevel(23);

}

}

The following is the output of this example:

x = 23

this.x = 1

ShadowTest.this.x = 0

This example defines three variables named x: the member variable of the class ShadowTest, the member variable of the inner class FirstLevel, and the parameter in the method methodInFirstLevel. The variable x defined as a parameter of the method methodInFirstLevel shadows the variable of the inner class FirstLevel. Consequently, when you use the variable x in the method methodInFirstLevel, it refers to the method parameter. To refer to the member variable of the inner class FirstLevel, use the keyword this to represent the enclosing scope:

System.out.println("this.x = " + this.x);

Refer to member variables that enclose larger scopes by the class name to which they belong. For example, the following statement accesses the member variable of the class ShadowTest from the method methodInFirstLevel:

System.out.println("ShadowTest.this.x = " + ShadowTest.this.x);

**Local class** – defined in a block (group of zero or more statements between balanced brances) typically in the body of a method; has access to members of its enclosting class and local variables ONLY IF they’re declared final

**Anonymous class** – similar to local classes but do NOT have a name (expresssions); use if you need to use a local class only once and if you need to declare fields/additional methods

**Lambda Expressions** – use if…

1. You are encapsulating a single unit of behavior that you want to pass to other code (for example, you want a certain action performed on each element of a collection, when a process is completed, or when a process encounters an error)
2. You need a simple instance of a functional interface and none of the preceding criteria apply (for example, you do not need a constructor, a named type, fields, or additional methods)

**Syntax:**

lambda operator -> body

<https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html>

**Compile-time:** the time period in which you, the developer, are compiling your code.

**Run-time:** the time period which a user is running your piece of software.

An **exception** is an event that occurs during the execution of a program that disrupts the normal flow of instructions. When an error occurs within a method, the method creates an object and hands it off to the runtime system. The object, called an *exception object*, contains information about the error, including its type and the state of the program when the error occurred. After a method throws an exception, the runtime system attempts to find something to handle it. The set of possible "somethings" to handle the exception is the ordered list of methods that had been called to get to the method where the error occurred. The list of methods is known as the *call stack*. The runtime system searches the call stack for a method that contains a block of code that can handle the exception. This block of code is called an *exception handler*. The search begins with the method in which the error occurred and proceeds through the call stack in the reverse order in which the methods were called. When an appropriate handler is found, the runtime system passes the exception to the handler. An exception handler is considered appropriate if the type of the exception object thrown matches the type that can be handled by the handler.

The exception handler chosen is said to *catch the exception*. If the runtime system exhaustively searches all the methods on the call stack without finding an appropriate exception handler, as shown in the next figure, the runtime system (and, consequently, the program) terminates.

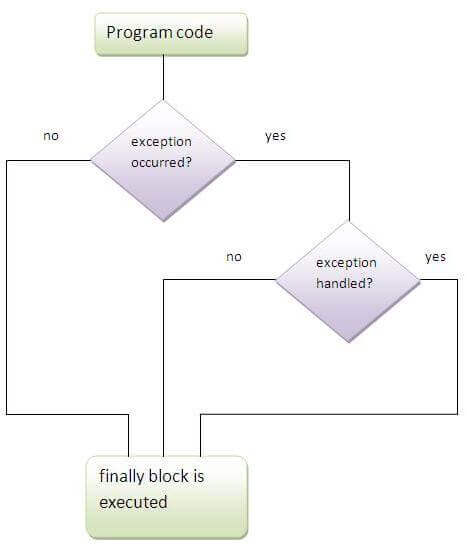
Exceptions must be thrown using

1. try/catch or
2. specify requirement (method that specifies it can throw the exception)

Three Types: (Runtime and Error are both considered Unchecked; Runtime exceptions *are not subject* to the Catch or Specify Requirement.)

* **Checked** are the exceptions that are checked at compile time. If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using *throws*keyword. Extends Throwable class except RunTimeException and Error **(user error)**
* **Runtime** These are exceptional conditions that are internal to the application, and that the application usually cannot anticipate or recover from. Classes that extend RunTimeException **(code error)**
* **Error** are external to the application, and that the application usually cannot anticipate or recover from





|  |  |
| --- | --- |
| **throw keyword** | **throws keyword** |
| 1) The **throw** keyword is used to throw an exception explicitly. | The **throws** keyword is used to declare an exception. |
| 2) The checked exceptions cannot be propagated with throw only. | The checked exception can be propagated with throws |
| 3) The **throw** keyword is followed by an instance. | The **throws** keyword is followed by class. |
| 4) The **throw** keyword is used within the method. | The **throws** keyword is used with the method signature. |
| 5) You cannot throw multiple exceptions. | You can declare multiple exceptions, e.g., public void method()throws IOException, SQLException. |

Throw Statement:

public Object pop() {

Object obj;

if (size == 0) {

**throw new EmptyStackException();**

}

obj = objectAt(size - 1);

setObjectAt(size - 1, null);

size--;

return obj;

}

Throws Statement

public void writeList() **throws IOException, IndexOutOfBoundsException** {

finally – block at end of try/catch that always executes

try with resources – resources closed when program is finished with them

**Concurrency**

**Process** – program in execution

**Thread** - Threads exist within a process — every process has at least one. Each thread is associated with an instance of the class. An application that creates an instance of Thread must provide the code that will run in that thread. There are two ways to do this: **implement Runnable** or **extend Thread**

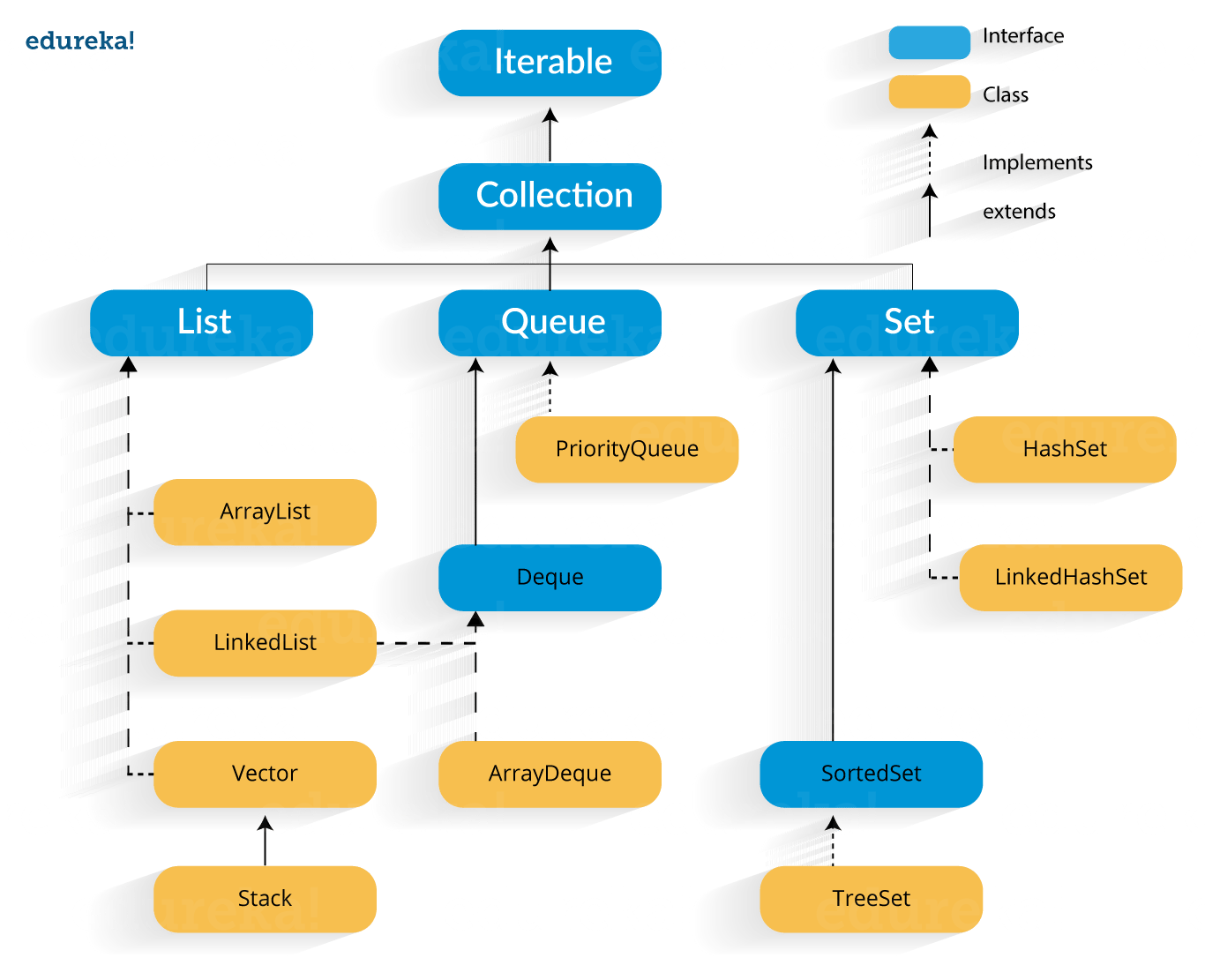
**Implementing Runnable** ismore flexible, but it is applicable to the high-level thread management APIs

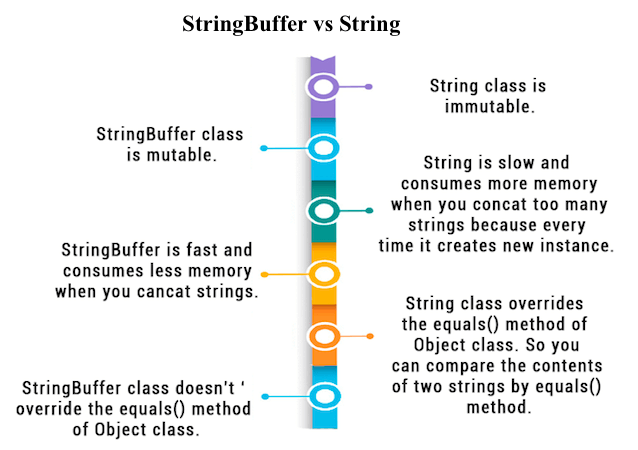
**Start** starts

**Sleep** pauses

**Interrupt** is an indication to a thread that it should stop what it is doing and do something else. It's up to the programmer to decide exactly how a thread responds to an interrupt, but it is very common for the thread to terminate.

**Join** allows one thread to wait for the completion of another





|  |  |  |
| --- | --- | --- |
| **No.** | **String** | **StringBuffer** |
| 1) | The String class is immutable. | The StringBuffer class is mutable. |
| 2) | The String is slow and consumes more memory when you concat too many strings because every time it creates a new instance. | The StringBuffer is fast and consumes less memory when you cancat strings. |
| 3) | The String class overrides the equals() method of Object class. So you can compare the contents of two strings by equals() method. | The StringBuffer class doesn't override the equals() method of Object class. |

|  |  |  |
| --- | --- | --- |
| **No.** | **StringBuffer** | **StringBuilder** |
| 1) | StringBuffer is *synchronized*, i.e., thread safe. It means two threads can't call the methods of StringBuffer simultaneously. | StringBuilder is *non-synchronized*,i.e., not thread safe. It means two threads can call the methods of StringBuilder simultaneously. |
| 2) | StringBuffer is *less efficient* than StringBuilder. | StringBuilder is *more efficient* than StringBuffer. |

|  |  |  |
| --- | --- | --- |
| **No.** | **Serializable** | **Externalizable** |
| 1) | The Serializable interface does not have any method, i.e., it is a marker interface. | The Externalizable interface contains is not a marker interface, It contains two methods, i.e., writeExternal() and readExternal(). |
| 2) | It is used to "mark" Java classes so that objects of these classes may get the certain capability. | The Externalizable interface provides control of the serialization logic to the programmer. |
| 3) | It is easy to implement but has the higher performance cost. | It is used to perform the serialization and often result in better performance. |
| 4) | No class constructor is called in serialization. | We must call a public default constructor while using this interface. |

References:

<https://docs.oracle.com/javase/tutorial/java/TOC.html>

<https://docs.oracle.com/javase/tutorial/essential/TOC.html>