**Exception and Hierarchy**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a video

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer program

Description automatically generated**

**A white background with black text

Description automatically generated**

**In case of Date mutable object**

**A screenshot of a computer

Description automatically generated**

**A close up of a computer code

Description automatically generated**

**A computer screen shot of a code

Description automatically generated**

**In case of List<String>**

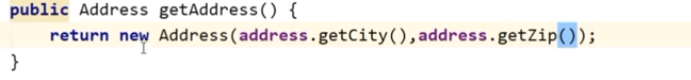
**A close up of a text

Description automatically generated**

**For User defined object such as Address here**

**A screenshot of a computer

Description automatically generated**

****

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**A black text on a white background

Description automatically generated**

**Here second object creation is preferable,it’s example of runtime polymorphism**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer error message

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer screen

Description automatically generated**

**JAVA8 Stream**

**A white text with black text

Description automatically generated**

**A screenshot of a phone

Description automatically generated**

**A yellow arrow with blue text

Description automatically generated**

[CompletableFuture](https://www.java67.com/2022/02/completablefuture-in-java-with-example.html) is a class introduced in Java 8 that allows us to write asynchronous, non-blocking code. It is a powerful tool that can help us write code that is more efficient and responsive.

Here’s an example of how to use [CompletableFuture](https://www.java67.com/2022/02/completablefuture-in-java-with-example.html" \t "_blank) in Java:

CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {  
 try {  
 Thread.sleep(5000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 return "Hello, world!";  
});  
  
future.thenAccept(result -> System.out.println(result));

In the example above, we’re creating a [CompletableFuture](https://www.java67.com/2022/02/completablefuture-in-java-with-example.html" \t "_blank) that will supply a result in the future. We're passing a lambda expression that simulates a long-running operation by sleeping for 5 seconds. After the operation is complete, it will return the string "Hello, world!".

We’re then calling the thenAccept() method on the [CompletableFuture](https://www.java67.com/2022/02/completablefuture-in-java-with-example.html" \t "_blank)object to specify what to do when the operation is complete. In this case, we're passing a lambda expression that simply prints the result to the console.

When we run this code, it will print “Hello, world!” to the console after a delay of 5 seconds.

Here’s another example that shows how to use [CompletableFuture](https://www.java67.com/2022/02/completablefuture-in-java-with-example.html" \t "_blank)to chain multiple asynchronous operations together:

CompletableFuture<Integer> future = CompletableFuture.supplyAsync(() -> 10)  
 .thenApplyAsync(result -> result \* 2)  
 .thenApplyAsync(result -> result + 5);  
  
future.thenAccept(result -> System.out.println(result));

In this example, we’re creating a [CompletableFuture](https://www.java67.com/2022/02/completablefuture-in-java-with-example.html" \t "_blank)that starts by supplying the value 10. We're then using the thenApplyAsync() method to chain two additional operations together: multiplying the result by 2, and then adding 5 to the result.

Finally, we’re calling thenAccept() to specify what to do when the entire operation is complete. In this case, we're simply printing the final result to the console.

When we run this code, it will print “25” to the console.

## Future vs CompletableFuture

[Future](https://javarevisited.blogspot.com/2015/01/how-to-use-future-and-futuretask-in-Java.html) and CompletableFuture are both abstractions for representing a result that will be available in the future, but there are some important differences between them.

1. Blocking vs non-blocking**:** One of the key differences between Future and CompletableFuture is that Future is a blocking API, whereas CompletableFuture is non-blocking. With a Future object, you must call the get() method to retrieve the result, but this method blocks until the result is available. In contrast, with a CompletableFuture object, you can use various non-blocking methods to retrieve the result, such as thenApply(), thenAccept(), or join().
2. Composition: CompletableFuture provides a more powerful composition API than Future. With Future, it is difficult to chain multiple asynchronous operations together or to combine the results of multiple operations. CompletableFuture, on the other hand, provides methods such as thenCompose(), thenCombine(), and allOf() that make it easy to compose multiple asynchronous operations and to handle their results in a non-blocking way.
3. Exception Handling: CompletableFuture provides better exception handling than Future. With Future, you can only check if the computation completed successfully or not. If an exception occurs during the computation, you have to catch it explicitly. In contrast, with CompletableFuture, you can handle exceptions in a more declarative way using methods like exceptionally() and handle().
4. Completion: With a Future object, there is no way to explicitly complete the future. Once you submit a task to an executor service and get a Future object in return, you can only wait for the task to complete. With CompletableFuture, you have more control over the completion of the future. You can complete it explicitly by calling complete(), completeExceptionally(), or cancel() methods.

In summary, CompletableFuture provides a more flexible and powerful API for working with asynchronous computations than [Future](https://javarevisited.blogspot.com/2015/01/how-to-use-future-and-futuretask-in-Java.html). It offers non-blocking methods, composition methods, better exception handling, and explicit completion methods, which makes it easier to write robust and scalable concurrent code.

**Run multiple futures in parallel using CompletableFuture**

To run multiple futures in parallel using CompletableFuture, you can use the CompletableFuture.allOf() method. Here's an example:

CompletableFuture<String> future1 = CompletableFuture.supplyAsync(() -> {  
 // Some long-running operation  
 return "Result 1";  
});  
  
CompletableFuture<String> future2 = CompletableFuture.supplyAsync(() -> {  
 // Some long-running operation  
 return "Result 2";  
});  
  
CompletableFuture<String> future3 = CompletableFuture.supplyAsync(() -> {  
 // Some long-running operation  
 return "Result 3";  
});  
  
CompletableFuture<Void> allFutures = CompletableFuture.allOf(future1, future2, future3);  
  
allFutures.thenRun(() -> {  
 // All futures completed  
 String result1 = future1.join();  
 String result2 = future2.join();  
 String result3 = future3.join();  
 System.out.println(result1 + ", " + result2 + ", " + result3);  
});

In this example, we’re creating three CompletableFuture objects, each representing a long-running operation that returns a String result. We're then using the CompletableFuture.allOf() method to combine all three futures into a single future that completes when all three are complete.

We’re then calling the thenRun() method on the allFutures object to specify what to do when all three futures have completed. In this case, we're using the join() method to get the results of each future, and then printing them to the console.

When we run this code, it will start all three futures in parallel and then wait for them to complete before printing the results. The order of the results may vary depending on which future completes first, but all three results will be printed to the console once they are all available.

**Handle errors — CompletableFuture**

CompletableFuture provides a number of methods for handling errors that can occur during asynchronous operations. Here are a few examples:

1. Handling an exception in a single CompletableFuture:

CompletableFuture<Integer> future = CompletableFuture.supplyAsync(() -> {  
 int result = 10 / 0; // Causes an ArithmeticException  
 return result;  
});  
  
future.exceptionally(ex -> {  
 System.out.println("Exception occurred: " + ex.getMessage());  
 return 0; // Default value to return if there's an exception  
}).thenAccept(result -> {  
 System.out.println("Result: " + result);  
});

In this example, we’re creating a CompletableFuture that will throw an ArithmeticException because we're dividing by zero. We're then calling the exceptionally() method to specify what to do if there's an exception. In this case, we're printing an error message to the console and returning a default value of 0.

We’re then calling the thenAccept() method to specify what to do when the operation is complete, whether there was an exception or not. In this case, we're simply printing the result to the console.

When we run this code, it will print “Exception occurred: / by zero” to the console followed by “Result: 0”.

2. Handling errors in multiple CompletableFuture objects:

CompletableFuture<Integer> future1 = CompletableFuture.supplyAsync(() -> {  
 // Some long-running operation  
 return 10;  
});  
  
CompletableFuture<Integer> future2 = CompletableFuture.supplyAsync(() -> {  
 int result = 10 / 0; // Causes an ArithmeticException  
 return result;  
});  
  
CompletableFuture<Integer> future3 = CompletableFuture.supplyAsync(() -> {  
 // Some long-running operation  
 return 20;  
});  
  
CompletableFuture<Void> allFutures = CompletableFuture.allOf(future1, future2, future3);  
  
allFutures.exceptionally(ex -> {  
 System.out.println("Exception occurred: " + ex.getMessage());  
 return null; // Default value to return if there's an exception  
}).thenRun(() -> {  
 // All futures completed  
 int result1 = future1.join();  
 int result2 = future2.join();  
 int result3 = future3.join();  
 System.out.println(result1 + ", " + result2 + ", " + result3);  
});

In this example, we’re creating three CompletableFuture objects, two of which will throw an exception. We're then using the CompletableFuture.allOf() method to combine all three futures into a single future that completes when all three are complete.

We’re then calling the exceptionally() method on the allFutures object to specify what to do if there's an exception. In this case, we're printing an error message to the console and returning a default value of null.

We’re then calling the thenRun() method to specify what to do when all three futures have completed. In this case, we're using the join() method to get the results of each future, and then printing them to the console.

When we run this code, it will print “Exception occurred: / by zero” to the console followed by “10, 0, 20”. Note that the second future returned a default value of 0 because of the exception, which is why we see “0” in the output.

**Async Methods of CompletableFuture**

CompletableFuture provides a set of asynchronous methods to execute multiple tasks concurrently and process the results as soon as they become available. These methods allow you to create a chain of dependent tasks and execute them in parallel, improving the performance of your application.

Here are some examples of the async methods available in CompletableFuture:

1. thenApplyAsync(): This method is used to process the result of a task asynchronously and return a new CompletableFuture with the transformed result. The processing is done by a separate thread in the ForkJoinPool.commonPool(). Here's an example:

CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> "Hello");  
  
CompletableFuture<Integer> transformedFuture = future.thenApplyAsync(s -> {  
 System.out.println("Thread: " + Thread.currentThread().getName());  
 return s.length();  
});  
  
transformedFuture.thenAccept(length -> {  
 System.out.println("Thread: " + Thread.currentThread().getName());  
 System.out.println("Length of Hello: " + length);  
});

In this example, we’re creating a CompletableFuture that returns the string "Hello". We're then using the thenApplyAsync() method to transform the result to its length using a separate thread. The thenAccept() method is then used to print the length of the string.

The output of this code will vary depending on the number of available threads in the ForkJoinPool, but it will look something like this:

Thread: ForkJoinPool.commonPool-worker-1  
Thread: ForkJoinPool.commonPool-worker-1  
Length of Hello: 5

2. thenAcceptAsync(): This method is used to consume the result of a task asynchronously, without returning a value. The processing is done by a separate thread in the ForkJoinPool.commonPool(). Here's an example:

thenAcceptAsync(): This method is used to consume the result of a task asynchronously, without returning a value. The processing is done by a separate thread in the ForkJoinPool.commonPool(). Here's an example:

n this example, we’re creating a CompletableFuture that returns the string "Hello". We're then using the thenAcceptAsync() method to print the string in uppercase using a separate thread.

The output of this code will vary depending on the number of available threads in the ForkJoinPool, but it will look something like this:

Thread: ForkJoinPool.commonPool-worker-1  
HELLO

3. runAsync(): This method is used to execute a task asynchronously, without returning a value. The processing is done by a separate thread in the ForkJoinPool.commonPool(). Here's an example:

CompletableFuture<Void> future = CompletableFuture.runAsync(() -> {  
 System.out.println("Thread: " + Thread.currentThread().getName());  
 System.out.println("Hello from async task");  
});

In this example, we’re using the runAsync() method to execute a task asynchronously that prints a message to the console using a separate thread.

The output of this code will vary depending on the number of available threads in the ForkJoinPool, but it will look something like this:

Thread: ForkJoinPool.commonPool-worker-1  
Hello from async task

4. thenComposeAsync() is a method in CompletableFuture that allows you to chain multiple asynchronous tasks together in a non-blocking way. This method is used when you have one CompletableFuture object that returns another CompletableFuture object as its result, and you want to execute the second task after the first one has completed.

The thenComposeAsync() method takes a Function object as its argument, which takes the result of the first CompletableFuture object as its input and returns another CompletableFuture object as its result. The second task is executed when the first one completes, and its result is passed to the next stage of the pipeline.

Here is an example of using thenComposeAsync():

CompletableFuture<String> future1 = CompletableFuture.supplyAsync(() -> "Hello");  
  
CompletableFuture<String> future2 = future1.thenComposeAsync(s -> CompletableFuture.supplyAsync(() -> s + " World"));  
  
future2.thenAccept(result -> System.out.println(result));

In this example, we create a CompletableFuture that returns the string "Hello". We then use the thenComposeAsync() method to chain another CompletableFuture to the first one. The second CompletableFuture takes the result of the first one and adds the string " World" to it. Finally, we use the thenAccept() method to print the result of the second CompletableFuture.

The output of this code will be:

Hello World

In this example, thenComposeAsync() creates a dependent CompletableFuture that takes the result of the first one and applies a transformation to it. The thenAccept() method is then used to consume the result of the dependent CompletableFuture. Note that both tasks are executed asynchronously, so the main thread doesn't block while they are running.

**Example**

We have 3 services asynchronously using RestTemplate and aggregates all the responses before sending the response to the caller. Here’s how you can do it:

A diagram of a software application

Description automatically generated

sequence diagram showing parallel calls

1. Create a service that uses RestTemplate to call the three endpoints asynchronously. You can use Spring’s AsyncRestTemplate to achieve this. Here's an example of how you can create a service that calls three endpoints and aggregates their responses:

@Service  
public class AggregatorService {  
 @Autowired  
 private AsyncRestTemplate restTemplate;  
  
 public CompletableFuture<AggregatedResponse> getAggregatedResponse() {  
 CompletableFuture<User[]> usersFuture = CompletableFuture.supplyAsync(() -> {  
 return restTemplate.getForObject("http://localhost:8080/users", User[].class);  
 });  
   
 CompletableFuture<Product[]> productsFuture = CompletableFuture.supplyAsync(() -> {  
 return restTemplate.getForObject("http://localhost:8080/products", Product[].class);  
 });  
   
 CompletableFuture<Order[]> ordersFuture = CompletableFuture.supplyAsync(() -> {  
 return restTemplate.getForObject("http://localhost:8080/orders", Order[].class);  
 });  
   
 return CompletableFuture.allOf(usersFuture, productsFuture, ordersFuture)  
 .thenApply(v -> new AggregatedResponse(usersFuture.join(), productsFuture.join(), ordersFuture.join()));  
 }  
}

In the example above, we’re calling three endpoints /users, /products, and /orders asynchronously using AsyncRestTemplate. We're using CompletableFuture to handle the asynchronous calls and aggregate their responses. Once all the responses are received, we're returning an AggregatedResponse object containing the data from all the endpoints.

2. Create a REST endpoint that calls the AggregatorService and returns the aggregated response. Here's an example of how you can create a REST endpoint that calls the AggregatorService:

@RestController  
public class AggregatorController {  
 @Autowired  
 private AggregatorService aggregatorService;  
  
 @GetMapping("/aggregate")  
 public CompletableFuture<AggregatedResponse> getAggregatedResponse() {  
 return aggregatorService.getAggregatedResponse();  
 }  
}

AggregatedResponse class

public class AggregatedResponse {  
 private User[] users;  
 private Product[] products;  
 private Order[] orders;  
  
 //getters and setters  
}

**Difference between java8 and java11**

|  | **Java 8** | **Java 11** |
| --- | --- | --- |
| **Appletviewer** | The appletviewer tool is available in Java 8. | The appletviewer tool is not available in Java 11. |
| **String Techniques** | It has fewer string methods. | Several new methods of String such as isBlank(), lines(), repeat(n), and many more. |
| **Lambda Expressions** | In lambda expressions, no special variables are used. | In lambda expressions, var variables are used. |
| **Java Deployment Technologies** | This technology is available in this version. | This technology is removed from this version. |
| **Patterns** | Pattern recognition is not possible. | With the help of the asMatchPredicate() method, pattern recognition is possible |
| **Garbage Collection** | It has less garbage collector memory | It has a better garbage collection system |
| **TLS Version** | It uses TLS 1.2 version. | It uses TLS 1.3 version. |
| **Security** | It is less secure when compared with JAVA 11. | In this version, the applications are quite secure than the applications in JAVA 8. |
| **Modularity** | The Modularity feature is not available in Java 8. | The Modularity feature is available in Java 11. |
| **Var Keyword** | Var keyword is not available. | Var keyword is available in Java 11, which is termed as a developer-friendly keyword. |
| **AWTUtilities** | Though the AWTUtilities class is available, it is not recommended. | The AWTUtilities class is not available. |
| **JMC and JavaFX** | These both are available in the Oracle JDK. | These both are removed from the Oracle JDK. |
| **Performance** | In JAVA 8, there is no suitable method to work with files. | In JAVA 11, there are various methods to work with the file such as writeString(), readString(), and isSameFile(). |

**Difference between java11 and java17**

Java 17 is the latest [LTS(Long Term Support)](https://blogs.oracle.com/javamagazine/post/java-long-term-support-lts) version of the Java programming language, released on September 14, 2021. If you’re currently using Java 11, it may be time to consider migrating to Java 17 to take advantage of its new features and improvements. In this article, we’ll discuss what’s new in Java 17, although some of the features discussed here have been introduced in version following up to Java 17 from Java 11.

# Why should we move from Java 11?

Although Java 11 is also an LTS version and is used by many application, there are some major reasons why we might want to shift to Java 17.

1. [Ending Support for Java 11](https://www.oracle.com/java/technologies/java-se-support-roadmap.html): Java 11 will be supported till September 2023 and extended support will be provided till September 2026. This means that after the support ends, we would have no patches(not even the security ones).
2. [Spring 6](https://spring.io/blog/2022/11/16/spring-framework-6-0-goes-ga): The latest version of Spring, Spring 6 will require Java 17 to work, and as there are many libraries which work along with them, they would also be moving to Java 17. If your applications rely on the Spring Framework, you should definitely consider moving to Java 17.
3. [Free Oracle JDK present for Java 17](https://www.oracle.com/cis/java/technologies/javase/jdk-faqs.html#:~:text=Oracle%20JDK%208%20and%20Oracle,Conditions%20License%20for%20all%20users.): Java 17 is issued under the new NFTC (Oracle No-Fee Terms and Conditions) license. It is therefore again allowed to use the Oracle JDK version for free for production and commercial use.(was not allowed for Java 11).

# Whats new in Java 17?

With Java 17 , there have been several improvements and new features introduced which will have long term support.

# Text Blocks

Java has introduced text blocks to make the code more readable and to avoid unnecessary string formatting. Now, we can place our text in between the triple quotes and have multiple double quoted strings inside it without having to use escape characters. An example is shown below:

private static void jsonBlock() {  
 String text = """  
 {  
 "name": "John Doe",  
 "age": 45,  
 "address": "Doe Street, 23, Java Town"  
 }  
 """;  
 System.out.println(text);  
}

As we can see, this makes it very easy to write Json and similar string which would require heavy usage of escape characters.

Also, the ending three double quotes indicate the beginning of the Text Block or its indentation in the output. In the example above, there will be two spaces for each line in the output as the position of the double quotes are two spaces behind the last character.

Two new escape characters have been introduced for use inside text blocks, ‘**\s’** for adding a space and ‘**\’** for removing newline. Especially useful when writing long SQL statements.

private static void sqlStatement() {  
 String sql = """  
 SELECT id, firstName, lastName\s\  
 FROM Employee  
 WHERE departmentId = "IT" \  
 ORDER BY lastName, firstName""";  
 System.out.println(text);  
}

# Improved Switch Statements

Switch Expressions will allow you to return values from the switch case and use these return values in assignments, etc. Java allows use of operator **->(arrow)** instead of **: (colon)**to denote the return expression. **break** keyword will not be needed when returning using switch in this expression, but **default** case is required.

private static void improvedSwitch(Fruit fruit) {  
 String text = switch (fruit) {  
 case APPLE, PEAR -> {  
 System.out.println("the given fruit was: " + fruit);  
 yield "Common fruit";  
 }  
 case ORANGE, AVOCADO -> "Exotic fruit";  
 default -> "Undefined fruit";  
 };  
 System.out.println(text);  
}

In case of multiple operations done inside a switch case, we can have a case block and denote the return value using the **yield** keyword. **yield** here is a context dependent keyword i.e. you can have a variable name **yield** somewhere else inside the function.

# ‘record’ Type

record classes are a special kind of immutable class which is meant to replace data transfer objects(DTOs). Normally if we want to use some POJO inside our class or methods, we would have to declare the class along with defining all the getters, setters, equals and hashcode functions. For example to use a sample Fruit class in other places, we would have to define our class someway like below:

public class Fruit {  
 private String name;  
 private int price;  
  
 //getters, setters, equals and hashcode methods  
}

Although we can reduce most of our boilerplate code by using libraries like [lombok](https://projectlombok.org/" \t "_blank), we can still reduce it even further with the help of records. With records the same code becomes:

public static void doSomething() {  
 record Fruit(String name, int price) {}  
 Fruit fruit = new Fruit("Apple", 100);  
 System.out.println(fruit.getPrice());  
}

As we can see, we can even define method local record objects. The records object automatically provides us with getter, setter, equals and hashcode methods for all its fields.

The fields inside the record cannot be changed, and it can only be defined by the arguments given when declaring the record as shown above(but we can define static variables). We can also define a custom constructor which can validate the fields. It is recommended that we do not override the getters and setters of records which could affects its immutability. An example of a record with multiple constructors and static variables and methods is shown below:

public record Employee(int id, String firstName,  
 String lastName)  
{  
  
 static int empToken;  
  
 // Compact Constructor  
 public Employee  
 {  
 if (id < 100) {  
 throw new IllegalArgumentException(  
 "Employee Id cannot be below 100.");  
 }  
 if (firstName.length() < 2) {  
 throw new IllegalArgumentException(  
 "First name must be 2 characters or more.");  
 }  
 }  
  
   
 // Alternative Constructor  
 public Employee(int id, String firstName)  
 {  
 this(id, firstName, null);  
 }  
  
 // Instance methods  
 public void getFullName()  
 {  
 if (lastName == null)  
 System.out.println(firstName());  
  
 else  
 System.out.println(firstName() + " "  
 + lastName());  
 }  
  
 // Static methods  
 public static int generateEmployeeToken()  
 {  
 return ++empToken;  
 }  
}

Some more qualities of record classes are:  
1. You can use nested classes and interfaces inside a record.

2. You can have nested records too, which will implicitly be static.

3. A record can implement interfaces.

4. You can create a generic record class.

5. Records are serializable.

More on records can be found [**here**](https://www.baeldung.com/java-record-keyword):

# ‘sealed’ Classes

sealed class will give us more control over which classes are allowed to extend our classes. In Java 11, a class can be final or extended. If you want to control which classes can extend your super class, you can put all classes in the same package and you give the super class package visibility. However, it is not possible anymore to access the super class from outside the package. As an example, see the code below:

public abstract class Fruit {  
}  
public final class Apple extends Fruit {  
}  
public final class Pear extends Fruit {  
}

private static void problemSpace() {  
 Apple apple = new Apple();  
 Pear pear = new Pear();  
 Fruit fruit = apple;  
 class Avocado extends Fruit {};  
}

Here, we cannot stop Avocado to extend the Fruit class. If we make the Fruit class [default](https://www.bogotobogo.com/Java/tutorials/defaultandprotected.php#:~:text=A%20class%20with%20default%20access,classes%20within%20the%20same%20package.), then the assignment of **apple** to **fruit** object would not compile. Hence, now we can use sealed classes to allow only specific classes to extend our superclass. An example is given below:

public abstract sealed class FruitSealed permits AppleSealed, PearSealed {  
}  
public non-sealed class AppleSealed extends FruitSealed {  
}  
public final class PearSealed extends FruitSealed {  
}

As we see, we use a new keyword **sealed** to denote that this is a sealed class. We define the classes that can be extended using the **permits** keyword. Any class which extends the sealed class can be either **final** like **PearSealed** or can be extended by other classes by using the **non-sealed** keyword when declaring the class as with **AppleSealed**.

This implementation would allow **AppleSealed** to be assigned to **FruitSealed**classbut wont allow any other classes not defined by permits keyword to extend FruitSealed class. More on sealed classes [**here**](https://www.baeldung.com/java-sealed-classes-interfaces).

# Pattern Matching with ‘instance of’

In Java 11, we usually use the **instance of**operator to check whether an object belongs to a certain class. We need to explicitly cast the object to that particular class if we want to perform some operation on it once the instance of check returns true. An example is shown below:

private static void oldStyle() {  
 Object o = new Grape(Color.BLUE, 2);  
 if (o instanceof GrapeClass) {  
 Grape grape = (Grape) o;  
 System.out.println("This grape has " + grape.getPits() + " pits.");  
 }  
}

Here, we needed to explicitly cast the object to type **Grape** and then find out the number of pits. With Java 17, we can change this to:

private static void patternMatchingInJava17() {  
 Object o = new Grape(Color.BLUE, 2);  
 if (o instanceof Grape grape) {  
 System.out.println("This grape has " + grape.getPits() + " pits.");  
 }  
}

We can pair the **instance of** check with an **&&(**and) condition but not **||**(or) as in case of an “**or**” condition the statement can reach the other condition even if instance of check return false

The scope of the variable grape can even extend beyond the if block if the **instance of** check return true. In the example below, Runtime Exception will be thrown if the object is not of **Grape** type, hence the compiler will know for sure that the grape object should exist when it reaches the print statement. More on pattern matching with **instance of** can be found [**here**](https://www.baeldung.com/java-pattern-matching-instanceof)**.**

private static void patternMatchingScopeException() {  
 Object o = new Grape(Color.BLUE, 2);  
 if (!(o instanceof Grape grape)) {  
 throw new RuntimeException();  
 }  
 System.out.println("This grape has " + grape.getPits() + " pits.");  
}

# Helpful NullPointerException

In Java 11 , when we get a NullPointerException, we only get the line number on which the exception occurred but we don’t get the method or variable which resolved to null. With Java 17, the messaging has been improved as the NullPointerException message also tells us the exact method invocation which caused the NullPointerException.

public static void main(String[] args) {  
 HashMap<String, Grape> grapes = new HashMap<>();  
 grapes.put("grape1", new GrapeClass(Color.BLUE, 2));  
 grapes.put("grape2", new GrapeClass(Color.white, 4));  
 grapes.put("grape3", null);  
 var color = ((Grape) grapes.get("grape3")).getColor();  
}

As we can see here, we are trying to get the colour of a “grape3” object which is null. When we compare the error message we get in Java 11 and Java 17, we see the difference in the error messaging as now we get to know exactly that calling the get method on the null object present in the map caused the exception.

// Java 11  
Exception in thread "main" java.lang.NullPointerException  
 at com.rg.java17.HelpfulNullPointerExceptions.main(HelpfulNullPointerExceptions.java:13)

// Java 17  
Exception in thread "main" java.lang.NullPointerException: Cannot invoke "com.rg.java17.Grape.getColor()" because the return value of "java.util.HashMap.get(Object)" is null  
 at com.rg.java17.HelpfulNullPointerExceptions.main(HelpfulNullPointerExceptions.java:13)

# Some more improvements

## Compact Number Formatting Support

A factory method is added to **NumberFormat** class in order to format numbers in compact, human-readable form according to the [Unicode](https://unicode.org/reports/tr35/tr35-numbers.html#Compact_Number_Formats) standard. There is a **SHORT** and **LONG** format available, an example is shown below:

NumberFormat shortFormat = NumberFormat.getCompactNumberInstance(Locale.ENGLISH, NumberFormat.Style.SHORT);  
System.out.println(shortFormat.format(1000))  
  
NumberFormat longFormat = NumberFormat.getCompactNumberInstance(Locale.ENGLISH, NumberFormat.Style.LONG);  
System.out.println(shortFormat.format(1000))

// Output  
1K  
1 thousand

## Day Period Support Added

A new pattern “**B”** is added to DateTime pattern allowing it to specify the time of the day.

DateTimeFormatter timeOfDayFomatter = DateTimeFormatter.ofPattern("B");  
System.out.println(timeOfDayFomatter.format(LocalTime.of(8, 0)));  
System.out.println(timeOfDayFomatter.format(LocalTime.of(13, 0)));  
System.out.println(timeOfDayFomatter.format(LocalTime.of(20, 0)));  
System.out.println(timeOfDayFomatter.format(LocalTime.of(23, 0)));  
System.out.println(timeOfDayFomatter.format(LocalTime.of(0, 0)));

// Output  
in the morning  
in the afternoon  
in the evening  
at night  
midnight

# Performance Benchmarks

Java 17 has also shown improvement over Java 11 in terms of its memory usage and time complexity. One such benchmark has been done where they tally code written in both version on their performance by making them do a series of tasks. The complete results and task descriptions can be found [here](https://www.optaplanner.org/blog/2021/09/15/HowMuchFasterIsJava17.html#:~:text=Java%2017%20is%208.66%25%20faster,than%20the%20G1%20Garbage%20Collector.).

Some of the general results which have been noted are:

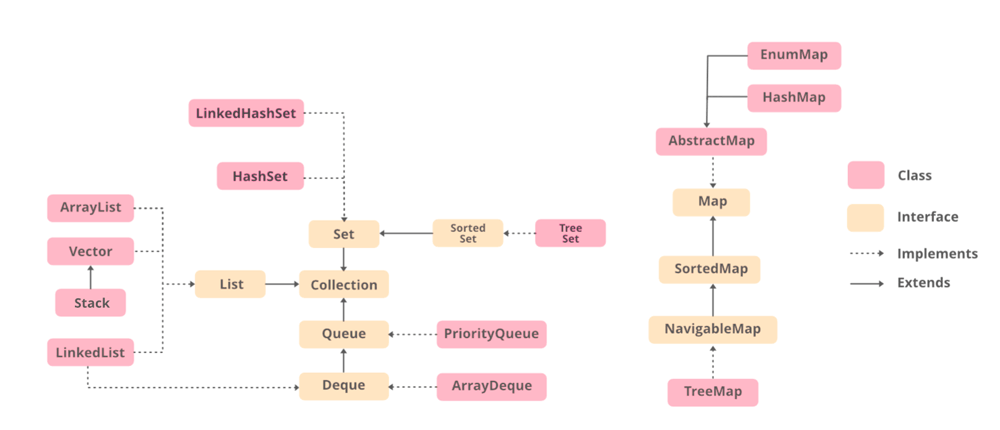
1. Java 17 is 8.66% faster than Java 11 and 2.41% faster than Java 16 for G1GC (default garbage collector).

2. Java 17 is 6.54% faster than Java 11 and 0.37% faster than Java 16 for ParallelGC(Parallel Garbage Collector).

3. The Parallel Garbage Collector(Available in Java 17) is 16.39% faster than the G1 Garbage Collector(Used in Java 11).

Migrating from Java 11 to Java 17 can provide many benefits, including new features and improved performance. However, it’s essential to be aware of potential bottlenecks that may arise during the migration process. Many libraries would also be upgrading to newer versions to support Java 17. Hence we should be extremely careful if we are using external libraries in our projects. By understanding these potential issues and taking the necessary steps to address them, you can ensure a smooth and successful migration to Java 17.

**ArrayList Vs LinkedList.**



|  | **ArrayList** | **LinkedList** |
| --- | --- | --- |
| **1.** | This class uses a dynamic array to store the elements in it. With the introduction of [generics](https://www.geeksforgeeks.org/generics-in-java/), this class supports the storage of all types of objects. | This class uses a [doubly linked list](https://www.geeksforgeeks.org/doubly-linked-list/) to store the elements in it. Similar to the ArrayList, this class also supports the storage of all types of objects. |
| **2.** | Manipulating ArrayList takes more time due to the internal implementation. Whenever we remove an element, internally, the array is traversed and the memory bits are shifted. | Manipulating LinkedList takes less time compared to ArrayList because, in a doubly-linked list, there is no concept of shifting the memory bits. The list is traversed and the reference link is changed. |
| **3.** | Inefficient memory utilization. | Good memory utilization. |
| **4.** | It can be one, two or multi-dimensional. | It can either be single, double or circular LinkedList. |
| **5.** | Insertion operation is slow. | Insertion operation is fast. |
| **6.** | This class implements a [List interface](https://www.geeksforgeeks.org/list-interface-java-examples/). Therefore, this acts as a list. | This class implements both the List interface and the [Deque interface](https://www.geeksforgeeks.org/deque-interface-java-example/). Therefore, it can act as a list and a deque. |
| **7.** | This class works better when the application demands storing the data and accessing it. | This class works better when the application demands manipulation of the stored data. |
| **8.** | Data access and storage is very efficient as it stores the elements according to the indexes. | Data access and storage is slow in LinkedList. |
| **9.** | Deletion operation is not very efficient. | Deletion operation is very efficient. |
| **10.** | It is used to store only similar types of data. | It is used to store any types of data. |
| **11.** | Less memory is used. | More memory is used. |
| **12.** | This is known as static memory allocation. | This is known as dynamic memory allocation. |

ArrayList is an**implementation of the List**interface that uses an array to store its elements. It has a fast indexed access time, which means that retrieving elements from an **ArrayList by an index is very quick.**

For example, the following code demonstrates how to retrieve an element from an ArrayList:

* Java