Java Scanner Class

The Scanner class of the java.util package is used to read input data from different sources like input streams, users, files, etc. Let's take an example.

Example 1: Read a Line of Text Using Scanner

```
import java.util.Scanner;

class Main {
  public static void main(String[] args) {

    // creates an object of Scanner
    Scanner input = new Scanner(System.in);

    System.out.print("Enter your name: ");

    // takes input from the keyboard
    String name = input.nextLine();

    // prints the name
    System.out.println("My name is " + name);

    // closes the scanner
    input.close();
  }
}
```

Output

```
Enter your name: Kelvin
My name is Kelvin
```

In the above example, notice the line

```
Scanner input = new Scanner(System.in);
```

Here, we have created an object of `Scanner` named *input*.

The `System.in` parameter is used to take input from the standard input. It works just like taking inputs from the keyboard.

We have then used the `nextLine()` method of the `Scanner` class to read a line of text from the user.

Now that you have some idea about `Scanner`, let's explore more about it.

Import Scanner Class

As we can see from the above example, we need to import the `java.util.Scanner` package before we can use the `Scanner` class.

```
import java.util.Scanner;
```

Create a Scanner Object in Java

Once we import the package, here is how we can create Scanner objects.

```
// read input from the input stream
Scanner sc1 = new Scanner(InputStream input);

// read input from files
Scanner sc2 = new Scanner(File file);

// read input from a string
Scanner sc3 = new Scanner(String str);
```

Here, we have created objects of the Scanner class that will read input from InputStream, File, and String respectively.

Java Scanner Methods to Take Input

The `Scanner` class provides various methods that allow us to read inputs of different types.

Method	Description
`nextInt()`	reads an `int` value from the user
`nextFloat()`	reads a `float` value form the user
`nextBoolean()`	reads a `boolean` value from the user
`nextLine()`	reads a line of text from the user
`next()`	reads a word from the user

`nextByte()` reads a `byte` value from the user

`nextDouble()` reads a `double` value from the user

`nextShort()` reads a `short` value from the user

`nextLong()` reads a `long` value from the user

Example 2: Java Scanner nextInt()

```
import java.util.Scanner;

class Main {
  public static void main(String[] args) {

    // creates a Scanner object
    Scanner input = new Scanner(System.in);

    System.out.println("Enter an integer: ");

    // reads an int value
    int data1 = input.nextInt();

    System.out.println("Using nextInt(): " + data1);

    input.close();
  }
}
```

Output

```
Enter an integer:
22
Using nextInt(): 22
```

Example 3: Java Scanner nextDouble()

```
import java.util.Scanner;

class Main {
  public static void main(String[] args) {

    // creates an object of Scanner
    Scanner input = new Scanner(System.in);
    System.out.print("Enter Double value: ");

    // reads the double value
    double value = input.nextDouble();
```

```
    System.out.println("Using nextDouble(): " + value);

    input.close();
  }
}
```

Output

```
Enter Double value: 33.33
Using nextDouble(): 33.33
```

In the above example, we have used the `nextDouble()` method to read a floating-point value.

Example 4: Java Scanner next()

```
import java.util.Scanner;

class Main {
  public static void main(String[] args) {

    // creates an object of Scanner
    Scanner input = new Scanner(System.in);
    System.out.print("Enter your name: ");

    // reads the entire word
    String value = input.next();
    System.out.println("Using next(): " + value);

    input.close();
  }
}
```

Output

```
Enter your name: Jonny Walker
Using next(): Jonny
```

In the above example, we have used the `next()` method to read a string from the user.

Here, we have provided the full name. However, the `next()` method only reads the first name.

This is because the `next()` method reads input up to the **whitespace** character. Once the **whitespace** is encountered, it returns the string (excluding the whitespace).

Example 5: Java Scanner nextLine()

```
import java.util.Scanner;

class Main {
  public static void main(String[] args) {

    // creates an object of Scanner
    Scanner input = new Scanner(System.in);
    System.out.print("Enter your name: ");

    // reads the entire line
    String value = input.nextLine();
    System.out.println("Using nextLine(): " + value);

    input.close();
  }
}
```

Output

```
Enter your name: Jonny Walker
Using nextLine(): Jonny Walker
```

In the first example, we have used the nextLine() method to read a string from the user.

Unlike next(), the nextLine() method reads the entire line of input including spaces. The method is terminated when it encounters a next line character, \n.

Java Scanner with BigInteger and BigDecimal

Java scanner can also be used to read the big integer and big decimal numbers.

- **nextBigInteger()** - reads the big integer value from the user
- **nextBigDecimal()** - reads the big decimal value from the user

Example 4: Read BigInteger and BigDecimal

```
import java.math.BigDecimal;
import java.math.BigInteger;
import java.util.Scanner;
```

```
class Main {
  public static void main(String[] args) {

    // creates an object of Scanner
    Scanner input = new Scanner(System.in);
    System.out.print("Enter a big integer: ");

    // reads the big integer
    BigInteger value1 = input.nextBigInteger();
    System.out.println("Using nextBigInteger(): " + value1);

    System.out.print("Enter a big decimal: ");

    // reads the big decimal
    BigDecimal value2 = input.nextBigDecimal();
    System.out.println("Using nextBigDecimal(): " + value2);

    input.close();
  }
}
```

Output

```
Enter a big integer: 987654321
Using nextBigInteger(): 987654321
Enter a big decimal: 9.55555
Using nextBigDecimal(): 9.55555
```

In the above example, we have used the `java.math.BigInteger` and `java.math.BigDecimal` package to read `BigInteger` and `BigDecimal` respectively.

Working of Java Scanner

The Scanner class reads an entire line and divides the line into tokens. Tokens are small elements that have some meaning to the Java compiler. For example,

Suppose there is an input string:

```
He is 22
```

In this case, the scanner object will read the entire line and divides the string into tokens: "**He**", "**is**" and "**22**". The object then iterates over each token and reads each token using its different methods.

Note: By default, whitespace is used to divide tokens.

Java Type Casting

Type Casting

The process of converting the value of one data type (`int`, `float`, `double`, etc.) to another data type is known as typecasting.

In Java, there are 13 types of type conversion. However, in this tutorial, we will only focus on the major 2 types.

1. Widening Type Casting

2. Narrowing Type Casting

Widening Type Casting

In **Widening Type Casting**, Java automatically converts one data type to another data type.

Example: Converting int to double

```
class Main {
  public static void main(String[] args) {
    // create int type variable
    int num = 10;
    System.out.println("The integer value: " + num);

    // convert into double type
    double data = num;
    System.out.println("The double value: " + data);
  }
}
```

Output

```
The integer value: 10
The double value: 10.0
```

In the above example, we are assigning the int type variable named *num* to a double type variable named *data*.

Here, the Java first converts the int type data into the double type. And then assign it to the double variable.

In the case of **Widening Type Casting**, the lower data type (having smaller size) is converted into the higher data type (having larger size). Hence there is no loss in data. This is why this type of conversion happens automatically.

Note: This is also known as **Implicit Type Casting**.

Narrowing Type Casting

In **Narrowing Type Casting**, we manually convert one data type into another using the parenthesis.

Example: Converting double into an int

```
class Main {
  public static void main(String[] args) {
    // create double type variable
    double num = 10.99;
    System.out.println("The double value: " + num);

    // convert into int type
    int data = (int)num;
    System.out.println("The integer value: " + data);
  }
}
```

Output

```
The double value: 10.99
The integer value: 10
```

In the above example, we are assigning the double type variable named *num* to an int type variable named *data*.

Notice the line,

```
int data = (int)num;
```

Here, the int keyword inside the parenthesis indicates that that the *num* variable is converted into the int type.

In the case of **Narrowing Type Casting**, the higher data types (having larger size) are converted into lower data types (having smaller size). Hence there is the loss of data. This is why this type of conversion does not happen automatically.

Note: This is also known as **Explicit Type Casting**.

Let's see some of the examples of other type conversions in Java.

Example 1: Type conversion from int to String

```
class Main {
  public static void main(String[] args) {
    // create int type variable
    int num = 10;
    System.out.println("The integer value is: " + num);

    // converts int to string type
    String data = String.valueOf(num);
    System.out.println("The string value is: " + data);
  }
}
```

Output

```
The integer value is: 10
The string value is: 10
```

In the above program, notice the line

```
String data = String.valueOf(num);
```

Here, we have used the valueOf() method of the Java String class to convert the int type variable into a string.

Example 2: Type conversion from String to int

```
class Main {
  public static void main(String[] args) {
    // create string type variable
    String data = "10";
    System.out.println("The string value is: " + data);

    // convert string variable to int
    int num = Integer.parseInt(data);
    System.out.println("The integer value is: " + num);
  }
}
```

Output

```
The string value is: 10
The integer value is: 10
```

In the above example, notice the line

```
int num = Integer.parseInt(data);
```

Here, we have used the `parseInt()` method of the Java `Integer` class to convert a string type variable into an `int` variable.

Note: If the string variable cannot be converted into the integer variable then an exception named `NumberFormatException` occurs.

Java autoboxing and unboxing

Java Autoboxing - Primitive Type to Wrapper Object

In **autoboxing**, the Java compiler automatically converts primitive types into their corresponding wrapper class objects. For example,

```
int a = 56;

// autoboxing
Integer aObj = a;
```

Autoboxing has a great advantage while working with Java collections.

Example 1: Java Autoboxing

```
import java.util.ArrayList;

class Main {
   public static void main(String[] args) {

      ArrayList<Integer> list = new ArrayList<>();

      //autoboxing
      list.add(5);
      list.add(6);

      System.out.println("ArrayList: " + list);
   }
}
```

Output

```
ArrayList: [5, 6]
```

In the above example, we have created an array list of Integer type. Hence the array list can only hold objects of Integer type.

Notice the line,

```
list.add(5);
```

Here, we are passing primitive type value. However, due to **autoboxing**, the primitive value is automatically converted into an Integer object and stored in the array list.

Java Unboxing - Wrapper Objects to Primitive Types

In **unboxing**, the Java compiler automatically converts wrapper class objects into their corresponding primitive types. For example,

```
// autoboxing
Integer aObj = 56;

// unboxing
int a = aObj;
```

Like **autoboxing**, **unboxing** can also be used with Java collections.

Example 2: Java Unboxing

```
import java.util.ArrayList;

class Main {
   public static void main(String[] args) {

      ArrayList<Integer> list = new ArrayList<>();

      //autoboxing
      list.add(5);
      list.add(6);

      System.out.println("ArrayList: " + list);

      // unboxing
      int a = list.get(0);
      System.out.println("Value at index 0: " + a);
   }
}
```

Output

```
ArrayList: [5, 6]
Value at index 0: 5
```

In the above example, notice the line,

```
int a = list.get(0);
```

Here, the `get()` method returns the object at index *0*. However, due to **unboxing**, the object is automatically converted into the primitive type `int` and assigned to the variable *a*.

Java Lambda Expressions

Java lambda expression and the use of lambda expression with functional interfaces, generic functional interface, and stream API with the help of examples.

The lambda expression was introduced first time in Java 8. Its main objective to increase the expressive power of the language.

But, before getting into lambdas, we first need to understand functional interfaces.

What is Functional Interface?

If a Java interface contains one and only one abstract method then it is termed as functional interface. This only one method specifies the intended purpose of the interface.

For example, the Runnable interface from package java.lang; is a functional interface because it constitutes only one method i.e. run().

Example 1: Define a Functional Interface in java

```
import java.lang.FunctionalInterface;
@FunctionalInterface
public interface MyInterface{
    // the single abstract method
    double getValue();
}
```

In the above example, the interface MyInterface has only one abstract method getValue(). Hence, it is a functional interface.

Here, we have used the annotation `@FunctionalInterface`. The annotation forces the Java compiler to indicate that the interface is a functional interface. Hence, does not allow to have more than one abstract method. However, it is not compulsory though.

In Java 7, functional interfaces were considered as Single Abstract Methods or **SAM** type. SAMs were commonly implemented with Anonymous Classes in Java 7.

Example 2: Implement SAM with anonymous classes in java

```
public class FunctionInterfaceTest {
    public static void main(String[] args) {

        // anonymous class
        new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println("I just implemented the Runnable
Functional Interface.");
            }
        }).start();
    }
}
```

Output:

```
I just implemented the Runnable Functional Interface.
```

Here, we can pass an anonymous class to a method. This helps to write programs with fewer codes in Java 7. However, the syntax was still difficult and a lot of extra lines of code were required.

Java 8 extended the power of a SAMs by going a step further. Since we know that a functional interface has just one method, there should be no need to define the name of that method when passing it as an argument. Lambda expression allows us to do exactly that.

Introduction to lambda expressions

Lambda expression is, essentially, an anonymous or unnamed method. The lambda expression does not execute on its own. Instead, it is used to implement a method defined by a functional interface.

How to define lambda expression in Java?

Here is how we can define lambda expression in Java.

```
(parameter list) -> lambda body
```

The new operator (`->`) used is known as an arrow operator or a lambda operator. The syntax might not be clear at the moment. Let's explore some examples,

Suppose, we have a method like this:

```
double getPiValue() {
    return 3.1415;
}
```

We can write this method using lambda expression as:

```
() -> 3.1415
```

Here, the method does not have any parameters. Hence, the left side of the operator includes an empty parameter. The right side is the lambda body that specifies the action of the lambda expression. In this case, it returns the value 3.1415.

Types of Lambda Body

In Java, the lambda body is of two types.

1. A body with a single expression

```
() -> System.out.println("Lambdas are great");
```

This type of lambda body is known as the expression body.

2. A body that consists of a block of code.

```
() -> {
    double pi = 3.1415;
    return pi;
};
```

This type of the lambda body is known as a block body. The block body allows the lambda body to include multiple statements. These statements are enclosed inside the braces and you have to add a semi-colon after the braces.

Note: For the block body, you can have a return statement if the body returns a value. However, the expression body does not require a return statement.

Example 3: Lambda Expression

Let's write a Java program that returns the value of Pi using the lambda expression.

As mentioned earlier, a lambda expression is not executed on its own. Rather, it forms the implementation of the abstract method defined by the functional interface.

So, we need to define a functional interface first.

```
import java.lang.FunctionalInterface;

// this is functional interface
@FunctionalInterface
interface MyInterface{

    // abstract method
    double getPiValue();
}

public class Main {

    public static void main( String[] args ) {

    // declare a reference to MyInterface
    MyInterface ref;

    // lambda expression
    ref = () -> 3.1415;

    System.out.println("Value of Pi = " + ref.getPiValue());
    }
}
```

Output:

```
Value of Pi = 3.1415
```

In the above example,

- We have created a functional interface named *MyInterface*. It contains a single abstract method named getPiValue()
- Inside the *Main* class, we have declared a reference to *MyInterface*. Note that we can declare a reference of an interface but we cannot instantiate an interface. That is,

  // it will throw an error
  MyInterface ref = new myInterface();

  // it is valid
  MyInterface ref;

  We then assigned a lambda expression to the reference.

  ref = () -> 3.1415;

  Finally, we call the method getPiValue() using the reference interface. When

  System.out.println("Value of Pi = " + ref.getPiValue());

Lambda Expressions with parameters

Till now we have created lambda expressions without any parameters. However, similar to methods, lambda expressions can also have parameters. For example,

```
(n) -> (n%2)==0
```

Here, the variable n inside the parenthesis is a parameter passed to the lambda expression. The lambda body takes the parameter and checks if it is even or odd.

Example 4: Using lambda expression with parameters

```
@FunctionalInterface
interface MyInterface {
```

```
    // abstract method
    String reverse(String n);
}

public class Main {

    public static void main( String[] args ) {

        // declare a reference to MyInterface
        // assign a lambda expression to the reference
        MyInterface ref = (str) -> {

            String result = "";
            for (int i = str.length()-1; i >= 0 ; i--)
            result += str.charAt(i);
            return result;
        };

        // call the method of the interface
        System.out.println("Lambda reversed = " +
ref.reverse("Lambda"));
    }

}
```

Output:

```
Lambda reversed = adbmaL
```

Generic Functional Interface

Till now we have used the functional interface that accepts only one type of value. For example,

```
@FunctionalInterface
interface MyInterface {
    String reverseString(String n);
}
```

The above functional interface only accepts String and returns String. However, we can make the functional interface generic, so that any data type is accepted.

Example 5: Generic Functional Interface and Lambda Expressions

```
// GenericInterface.java
@FunctionalInterface
interface GenericInterface<T> {
```

```
    // generic method
    T func(T t);
}

// GenericLambda.java
public class Main {

    public static void main( String[] args ) {

        // declare a reference to GenericInterface
        // the GenericInterface operates on String data
        // assign a lambda expression to it
        GenericInterface<String> reverse = (str) -> {

            String result = "";
            for (int i = str.length()-1; i >= 0 ; i--)
            result += str.charAt(i);
            return result;
        };

        System.out.println("Lambda reversed = " +
reverse.func("Lambda"));

        // declare another reference to GenericInterface
        // the GenericInterface operates on Integer data
        // assign a lambda expression to it
        GenericInterface<Integer> factorial = (n) -> {

            int result = 1;
            for (int i = 1; i <= n; i++)
            result = i * result;
            return result;
        };

        System.out.println("factorial of 5 = " + factorial.func(5));
    }
}
```

Output:

```
Lambda reversed = adbmaL
factorial of 5 = 120
```

In the above example, we have created a generic functional interface named *GenericInterface*. It contains a generic method named func().

Here, inside the Main class,

- **`GenericInterface<String> reverse`** - creates a reference to the interface. The interface now operates on `String` type of data.
- **`GenericInterface<Integer> factorial`** - creates a reference to the interface. The interface, in this case, operates on the `Integer` type of data.

Lambda Expression and Stream API

The new java.util.stream package has been added to JDK8 which allows java developers to perform operations like search, filter, map, reduce, or manipulate collections like Lists.

For example, we have a stream of data (in our case a List of String) where each string is a combination of country name and place of the country. Now, we can process this stream of data and retrieve only the places from Nepal.

For this, we can perform bulk operations in the stream by the combination of Stream API and Lambda expression.

Example 6: Demonstration of using lambdas with the Stream API

```
import java.util.ArrayList;
import java.util.List;

public class StreamMain {

    // create an object of list using ArrayList
    static List<String> places = new ArrayList<>();

    // preparing our data
    public static List getPlaces(){

        // add places and country to the list
        places.add("Nepal, Kathmandu");
        places.add("Nepal, Pokhara");
        places.add("India, Delhi");
        places.add("USA, New York");
        places.add("Africa, Nigeria");

        return places;
    }

    public static void main( String[] args ) {

        List<String> myPlaces = getPlaces();
        System.out.println("Places from Nepal:");
```

```
        // Filter places from Nepal
        myPlaces.stream()
                .filter((p) -> p.startsWith("Nepal"))
                .map((p) -> p.toUpperCase())
                .sorted()
                .forEach((p) -> System.out.println(p));
    }

}
```

Output:

```
Places from Nepal:
NEPAL, KATHMANDU
NEPAL, POKHARA
```

In the above example, notice the statement,

```
myPlaces.stream()
        .filter((p) -> p.startsWith("Nepal"))
        .map((p) -> p.toUpperCase())
        .sorted()
        .forEach((p) -> System.out.println(p));
```

Here, we are using the methods like `filter()`, `map()` and `forEach()` of the Stream API. These methods can take a lambda expression as input.

We can also define our own expressions based on the syntax we learned above. This allows us to reduce the lines of code drastically as we saw in the above example.

Java Generics

In this tutorial, we will learn about Java Generics, how to create generics class and methods and its advantages with the help of examples.

The Java Generics allows us to create a single class, interface, and method that can be used with different types of data (objects).

This helps us to reuse our code.

Note: **Generics** does not work with primitive types (int, float, char, etc).

Java Generics Class

We can create a class that can be used with any type of data. Such a class is known as Generics Class.

Here's is how we can create a generics class in Java:

Example: Create a Generics Class

```
class Main {
  public static void main(String[] args) {

    // initialize generic class
    // with Integer data
    GenericsClass<Integer> intObj = new GenericsClass<>(5);
    System.out.println("Generic Class returns: " + intObj.getData());

    // initialize generic class
    // with String data
    GenericsClass<String> stringObj = new GenericsClass<>("Java
Programming");
    System.out.println("Generic Class returns: " +
stringObj.getData());
  }
}

// create a generics class
class GenericsClass<T> {

  // variable of T type
  private T data;

  public GenericsClass(T data) {
    this.data = data;
  }

  // method that return T type variable
  public T getData() {
    return this.data;
  }
}
```

Output

```
Generic Class returns: 5
Generic Class returns: Java Programming
```

In the above example, we have created a generic class named *GenericsClass*. This class can be used to work with any type of data.

```
class GenericsClass<T> {...}
```

Here, *T* used inside the angle bracket <> indicates the **type parameter**. Inside the Main class, we have created two objects of *GenericsClass*

- *intObj* - Here, the type parameter *T* is replaced by Integer. Now, the *GenericsClass* works with integer data.
- *stringObj* - Here, the type parameter *T* is replaced by String. Now, the *GenericsClass* works with string data.

Java Generics Method

Similar to the generics class, we can also create a method that can be used with any type of data. Such a class is known as Generics Method.

Here's is how we can create a generics class in Java:

Example: Create a Generics Method

```
class Main {
  public static void main(String[] args) {

    // initialize the class with Integer data
    DemoClass demo = new DemoClass();

    // generics method working with String
    demo.<String>genericsMethod("Java Programming");

    // generics method working with integer
    demo.<Integer>genericsMethod(25);
  }
}

class DemoClass {

  // creae a generics method
  public <T> void genericsMethod(T data) {
    System.out.println("Generics Method:");
    System.out.println("Data Passed: " + data);
  }
}
```

Output

```
Generics Method:
Data Passed: Java Programming
```

```
Generics Method:
Data Passed: 25
```

In the above example, we have created a generic method named *genericsMethod*.

```
public <T> void genericMethod(T data) {...}
```

Here, the type parameter <T> is inserted after the modifier public and before the return type void.

We can call the generics method by placing the actual type <String> and <Integer> inside the bracket before the method name.

```
demo.<String>genericMethod("Java Programming");

demo.<Integer>genericMethod(25);
```

Note: We can call the generics method without including the type parameter. For example,

```
demo.genericsMethod("Java Programming");
```

In this case, the compiler can match the type parameter based on the value passed to the method.

Bounded Types

In general, the **type parameter** can accept any data types (except primitive types).

However, if we want to use generics for some specific types (such as accept data of number types) only, then we can use bounded types.

In the case of bound types, we use the extends keyword. For example,

```
<T extends A>
```

This means *T* can only accept data that are subtypes of *A*.

Example: Bounded Types

```
class GenericsClass <T extends Number> {

  public void display() {
```

```
    System.out.println("This is a bounded type generics class.");
  }
}

class Main {
  public static void main(String[] args) {

    // create an object of GenericsClass
    GenericsClass<String> obj = new GenericsClass<>();
  }
}
```

In the above example, we have created a class named GenericsClass. Notice the expression, notice the expression

```
<T extends Number>
```

Here, *GenericsClass* is created with bounded type. This means *GenericsClass* can only work with data types that are children of Number (Integer, Double, and so on).

However, we have created an object of the generics class with String. In this case, we will get the following error.

```
GenericsClass<String> obj = new GenericsClass<>();
                                                ^
    reason: inference variable T has incompatible bounds
      equality constraints: String
      lower bounds: Number
  where T is a type-variable:
    T extends Number declared in class GenericsClass
```

Advantages of Java Generics

1. Code Reusability

With the help of generics in Java, we can write code that will work with different types of data. For example,

```
public <T> void genericsMethod(T data) {...}
```

Here, we have created a generics method. This same method can be used to perform operations on integer data, string data, and so on.

2. Compile-time Type Checking

The **type parameter** of generics provides information about the type of data used in the generics code. For example,

```
// using Generics
GenericsClass<Integer> list = new GenericsClass<>();
```

Here, we know that *GenericsClass* is working with Integer data only.

Now, if we try to pass data other than Integer to this class, the program will generate an error at compile time.

3. Used with Collections

The collections framework uses the concept of generics in Java. For example,

```
// creating a string type ArrayList
ArrayList<String> list1 = new ArrayList<>();

// creating a integer type ArrayList
ArrayList<Integer> list2 = new ArrayList<>();
```

In the above example, we have used the same ArrayList class to work with different types of data.

Similar to ArrayList, other collections (LinkedList, Queue, Maps, and so on) are also generic in Java.

Java File Class

The File class of the java.io package is used to perform various operations on files and directories.

There is another package named java.nio that can be used to work with files. However, in this tutorial, we will focus on the java.io package.

File and Directory

A file is a named location that can be used to store related information. For example,

main.java is a Java file that contains information about the Java program.

A directory is a collection of files and subdirectories. A directory inside a directory is known as subdirectory.

Create a Java File Object

To create an object of File, we need to import the java.io.File package first. Once we import the package, here is how we can create objects of file.

```
// creates an object of File using the path
File file = new File(String pathName);
```

Here, we have created a file object named *file*. The object can be used to work with files and directories.

Note: In Java, creating a file object does not mean creating a file. Instead, a file object is an abstract representation of the file or directory pathname (specified in the parenthesis).

Java File Operation Methods

Operation	Method	Package
To create file	`createNewFile()`	`java.io.File`
To read file	`read()`	`java.io.FileReader`
To write file	`write()`	`java.io.FileWriter`
To delete file	`delete()`	`java.io.File`

Java create files

To create a new file, we can use the `createNewFile()` method. It returns

- `true` if a new file is created.
- `false` if the file already exists in the specified location.

Example: Create a new File

```
// importing the File class
import java.io.File;

class Main {
  public static void main(String[] args) {

    // create a file object for the current location
    File file = new File("newFile.txt");

    try {

      // trying to create a file based on the object
      boolean value = file.createNewFile();
      if (value) {
        System.out.println("The new file is created.");
      }
      else {
        System.out.println("The file already exists.");
      }
    }
    catch(Exception e) {
      e.getStackTrace();
    }
  }
}
```

In the above example, we have created a file object named *file*. The file object is linked with the specified file path.

```
File file = new File("newFile.txt");
```

Here, we have used the file object to create the new file with the specified path.

If newFile.txt doesn't exist in the current location, the file is created and this message is shown.

```
The new file is created.
```

However, if newFile.txt already exists, we will see this message.

```
The file already exists.
```

Java read files

To read data from the file, we can use subclasses of either InputStream or Reader.

Example: Read a file using FileReader

Suppose we have a file named **input.txt** with the following content.

```
This is a line of text inside the file.
```

Now let's try to read the file using Java `FileReader`.

```
// importing the FileReader class
import java.io.FileReader;

class Main {
  public static void main(String[] args) {

    char[] array = new char[100];
    try {
      // Creates a reader using the FileReader
      FileReader input = new FileReader("input.txt");

      // Reads characters
      input.read(array);
      System.out.println("Data in the file:");
      System.out.println(array);

      // Closes the reader
```

```
      input.close();
    }
    catch(Exception e) {
      e.getStackTrace();
    }
  }
}
```

Output

```
Data in the file:
This is a line of text inside the file.
```

In the above example, we have used created an object of FileReader named input. It is now linked with the **input.txt** file.

```
FileReader input = new FileReader("input.txt");
```

To read the data from the **input.txt** file, we have used the read() method of `FileReader`.

Java write to files

To write data to the file, we can use subclasses of either OutputStream or Writer.

Example: Write to file using FileWriter

```
// importing the FileWriter class
import java.io.FileWriter;

 class Main {
   public static void main(String args[]) {

     String data = "This is the data in the output file";
     try {
       // Creates a Writer using FileWriter
       FileWriter output = new FileWriter("output.txt");

       // Writes string to the file
       output.write(data);
       System.out.println("Data is written to the file.");

       // Closes the writer
       output.close();
     }
     catch (Exception e) {
       e.getStackTrace();
```

```
    }
  }
}
```

Output

```
Data is written to the file.
```

In the above example, we have created a writer using the FileWriter class. The writer is linked with the **output.txt** file.

```
FileWriter output = new FileWriter("output.txt");
```

To write data to the file, we have used the write() method.

Here when we run the program, the **output.txt** file is filled with the following content.

```
This is the data in the output file.
```

Java delete files

We can use the `delete()` method of the *File* class to delete the specified file or directory. It returns

- `true` if the file is deleted.
- `false` if the file does not exist.

Note: We can only delete empty directories.

Example: Delete a file

```
import java.io.File;

class Main {
  public static void main(String[] args) {

    // creates a file object
    File file = new File("file.txt");

    // deletes the file
    boolean value = file.delete();
    if(value) {
      System.out.println("The File is deleted.");
    }
    else {
```

```
      System.out.println("The File is not deleted.");
    }
  }
}
```

Output

```
The File is deleted.
```

In the above example, we have created an object of File named file. The file now holds the information about the specified file.

```
File file = new File("file.txt");
```

Here we have used the delete() method to delete the file specified by the object.

Java Wrapper Class

In this tutorial, we will learn about the Java Wrapper class with the help of examples.

The wrapper classes in Java are used to convert primitive types (int, char, float, etc) into corresponding objects.

Each of the 8 primitive types has corresponding wrapper classes.

Primitive Type	Wrapper Class
byte	Byte
boolean	Boolean
char	Character
double	Double
float	Float
int	Integer
long	Long
short	Short

Convert Primitive Type to Wrapper Objects

We can also use the `valueOf()` method to convert primitive types into corresponding objects.

Example 1: Primitive Types to Wrapper Objects

```
class Main {
  public static void main(String[] args) {

    // create primitive types
    int a = 5;
    double b = 5.65;

    //converts into wrapper objects
    Integer aObj = Integer.valueOf(a);
    Double bObj = Double.valueOf(b);

    if(aObj instanceof Integer) {
      System.out.println("An object of Integer is created.");
    }

    if(bObj instanceof Double) {
      System.out.println("An object of Double is created.");
    }
  }
}
```

Output

```
An object of Integer is created.
An object of Double is created.
```

In the above example, we have used the valueOf() method to convert the primitive types into objects.

Here, we have used the instanceof operator to check whether the generated objects are of Integer or Double type or not.

However, the Java compiler can directly convert the primitive types into corresponding objects. For example,

```
int a = 5;
// converts into object
Integer aObj = a;

double b = 5.6;
// converts into object
```

```
Double bObj = b;
```

This process is known as **auto-boxing**.

Note: We can also convert primitive types into wrapper objects using Wrapper class constructors. But the use of constructors is discarded after Java 9.

Wrapper Objects into Primitive Types

To convert objects into the primitive types, we can use the corresponding value methods (intValue(), doubleValue(), etc) present in each wrapper class.

Example 2: Wrapper Objects into Primitive Types

```
class Main {
  public static void main(String[] args) {

    // creates objects of wrapper class
    Integer aObj = Integer.valueOf(23);
    Double bObj = Double.valueOf(5.55);

    // converts into primitive types
    int a = aObj.intValue();
    double b = bObj.doubleValue();

    System.out.println("The value of a: " + a);
    System.out.println("The value of b: " + b);
  }
}
```

Output

```
The value of a: 23
The value of b: 5.55
```

In the above example, we have used the intValue() and doubleValue() method to convert the Integer and Double objects into corresponding primitive types.

However, the Java compiler can automatically convert objects into corresponding primitive types. For example,

```
Integer aObj = Integer.valueOf(2);
// converts into int type
int a = aObj;

Double bObj = Double.valueOf(5.55);
// converts into double type
```

```
double b = bObj;
```

This process is known as **unboxing**.

Advantages of Wrapper Classes

- In Java, sometimes we might need to use objects instead of primitive data types. For example, while working with collections.

```
// error
ArrayList<int> list = new ArrayList<>();

// runs perfectly
ArrayList<Integer> list = new ArrayList<>();
```

In such cases, wrapper classes help us to use primitive data types as objects.

We can store the null value in wrapper objects. For example,

```
// generates an error
int a = null;

// runs perfectly
Integer a = null;
```

Note: Primitive types are more efficient than corresponding objects. Hence, when efficiency is the requirement, it is always recommended primitive types.

Java Command-Line Arguments

The **command-line arguments** in Java allow us to pass arguments during the execution of the program.

As the name suggests arguments are passed through the command line.

Example: Command-Line Arguments

```
class Main {
  public static void main(String[] args) {
    System.out.println("Command-Line arguments are");
```

```
    // loop through all arguments
    for(String str: args) {
      System.out.println(str);
    }
  }
}
```

Let's try to run this program using the command line.

1. To compile the code

```
javac Main.java
```

2. To run the code

```
java Main
```

Now suppose we want to pass some arguments while running the program, we can pass the arguments after the class name. For example,

```
java Main apple ball cat
```

Here *apple*, *ball*, and *cat* are arguments passed to the program through the command line. Now, we will get the following output.

```
Command-Line arguments are
Apple
Ball
Cat
```

In the above program, the main() method includes an array of strings named *args* as its parameter.

```
public static void main(String[] args) {...}
```

The String array stores all the arguments passed through the command line.

Note: Arguments are always stored as strings and always separated by **white-space**.

Passing Numeric Command-Line Arguments

The main() method of every Java program only accepts string arguments. Hence it is not possible to pass numeric arguments through the command line.

However, we can later convert string arguments into numeric values.

Example: Numeric Command-Line Arguments

```
class Main {
  public static void main(String[] args) {

    for(String str: args) {
      // convert into integer type
    int argument = Integer.parseInt(str);
    System.out.println("Argument in integer form: " + argument);
    }

  }
}
```

Let's try to run the program through the command line.

```
// compile the code
javac Main.java

// run the code
java Main 11 23
```

Here *11* and *23* are command-line arguments. Now, we will get the following output.

```
Arguments in integer form
11
23
```

In the above example, notice the line

```
int argument = Intege.parseInt(str);
```

Here, the `parseInt()` method of the `Integer` class converts the string argument into an integer.

Similarly, we can use the `parseDouble()` and `parseFloat()` method to convert the string into `double` and `float` respectively.

Note: If the arguments cannot be converted into the specified numeric value then an exception named `NumberFormatException` occurs.