|  |
| --- |
| // Java program to illustrate Simple Predicate    **import** java.util.function.Predicate;  **public** **class** PredicateInterfaceExample1 {  **public** **static** **void** main(String[] args)      {          // Creating predicate          Predicate<Integer> lesserthan = i -> (i < 18);            // Calling Predicate method          System.out.println(lesserthan.test(10));      }  } |

Output:

True

**Example 2: Predicate Chaining**

* Java

|  |
| --- |
| // Java program to illustrate Predicate Chaining    **import** java.util.function.Predicate;  **public** **class** PredicateInterfaceExample2 {  **public** **static** **void** main(String[] args)      {          Predicate<Integer> greaterThanTen = (i) -> i > 10;            // Creating predicate          Predicate<Integer> lowerThanTwenty = (i) -> i < 20;  **boolean** result = greaterThanTen.and(lowerThanTwenty).test(15);          System.out.println(result);            // Calling Predicate method  **boolean** result2 = greaterThanTen.and(lowerThanTwenty).negate().test(15);          System.out.println(result2);      }  } |

Output:

True

False

**Example 3: Predicate in to Function**

* Java

|  |
| --- |
| // Java program to illustrate  // passing Predicate into function    **import** java.util.function.Predicate;  **class** PredicateInterfaceExample3 {  **static** **void** pred(**int** number, Predicate<Integer> predicate)      {  **if** (predicate.test(number)) {              System.out.println("Number " + number);          }      }  **public** **static** **void** main(String[] args)      {          pred(10, (i) -> i > 7);      }  } |

Output:

Number 10

**Example 4: Predicate OR**

* Java

|  |
| --- |
| // Java program to illustrate OR Predicate    **import** java.util.function.Predicate;  **class** PredicateInterfaceExample4 {  **public** **static** Predicate<String> hasLengthOf10 = **new** Predicate<String>() {          @Override  **public** **boolean** test(String t)          {  **return** t.length() > 10;          }      };    **public** **static** **void** predicate\_or()      {            Predicate<String> containsLetterA = p -> p.contains("A");          String containsA = "And";  **boolean** outcome = hasLengthOf10.or(containsLetterA).test(containsA);          System.out.println(outcome);      }  **public** **static** **void** main(String[] args)      {          predicate\_or();      }  } |

Output:

True

**Example 5: Predicate AND**

* Java

|  |
| --- |
| // Java program to illustrate AND Predicate    **import** java.util.function.Predicate;  **import** java.util.Objects;    **class** PredicateInterfaceExample5 {  **public** **static** Predicate<String> hasLengthOf10 = **new** Predicate<String>() {          @Override  **public** **boolean** test(String t)          {  **return** t.length() > 10;          }      };    **public** **static** **void** predicate\_and()      {          Predicate<String> nonNullPredicate = Objects::nonNull;            String nullString = **null**;    **boolean** outcome = nonNullPredicate.and(hasLengthOf10).test(nullString);          System.out.println(outcome);            String lengthGTThan10 = "Welcome to the machine";  **boolean** outcome2 = nonNullPredicate.and(hasLengthOf10).          test(lengthGTThan10);          System.out.println(outcome2);      }  **public** **static** **void** main(String[] args)      {          predicate\_and();      }  } |

Output:

False

True

**Example 6: Predicate negate()**

* Java

|  |
| --- |
| // Java program to illustrate  // negate Predicate    **import** java.util.function.Predicate;  **class** PredicateInterfaceExample6 {  **public** **static** Predicate<String> hasLengthOf10 = **new** Predicate<String>() {          @Override  **public** **boolean** test(String t)          {  **return** t.length() > 10;          }      };    **public** **static** **void** predicate\_negate()      {            String lengthGTThan10 = "Thunderstruck is a 2012 children's "                                  + "film starring Kevin Durant";    **boolean** outcome = hasLengthOf10.negate().test(lengthGTThan10);          System.out.println(outcome);      }  **public** **static** **void** main(String[] args)      {          predicate\_negate();      }  } |

Output:

False

**Example 7: Predicate in Collection**

* Java

|  |
| --- |
| // Java program to demonstrate working of predicates  // on collection. The program finds all admins in an  // arrayList of users.  **import** java.util.function.Predicate;  **import** java.util.\*;  **class** User  {      String name, role;      User(String a, String b) {          name = a;          role = b;      }      String getRole() { **return** role; }      String getName() { **return** name; }  **public** String toString() {  **return** "User Name : " + name + ", Role :" + role;      }    **public** **static** **void** main(String args[])      {          List<User> users = **new** ArrayList<User>();          users.add(**new** User("John", "admin"));          users.add(**new** User("Peter", "member"));          List admins = process(users, (User u) -> u.getRole().equals("admin"));          System.out.println(admins);      }    **public** **static** List<User> process(List<User> users,                              Predicate<User> predicate)      {          List<User> result = **new** ArrayList<User>();  **for** (User user: users)  **if** (predicate.test(user))                  result.add(user);  **return** result;      }  } |

Output:

[User Name : John, Role :admin]

The same functionality can also be achieved by using [Stream API](https://www.geeksforgeeks.org/stream-in-java/)  
and [lambda functions](https://www.geeksforgeeks.org/lambda-expressions-java-8/#targetText=lambda%20expressions%20are%20added%20in,object%20and%20executed%20on%20demand.) offered since JDK 1.8 on top of the Collections API.

The **Stream API** allows "streaming" of collections for dynamic processing. Streams allow concurrent and parallel computation on data (using internal iterations), to support database-like operations such as grouping and filtering the data (similar to GROUP BY and WHERE clause in SQL). This allows the developers to focus on "what data is needed" instead of "how data is needed" since streaming hides the details of the implementation and provides the result. This is done by providing predicates as inputs to functions operating at runtime upon the streams of collections. In the following example, we illustrate how Stream API can be used along with predicates to filter the collections of data as achieved in Example 7.

* Java

|  |
| --- |
| // Java program to demonstrate working of predicates  // on collection. The program finds all admins in an  // arrayList of users.  **import** java.util.function.Predicate;  **import** java.util.\*;  **import** java.util.stream.Collectors;  **import** java.util.stream.Stream;    **class** User  {      String name, role;      User(String a, String b) {          name = a;          role = b;      }      String getRole() { **return** role; }      String getName() { **return** name; }  **public** String toString() {  **return** "User Name : " + name + ",      Role :" + role;      }    **public** **static** **void** main(String args[])      {          List<User> users =  **new** ArrayList<User>();          users.add(**new** User("John", "admin"));          users.add(**new** User("Peter", "member"));        // This line uses Predicates to filter      // out the list of users with the role "admin".      // List admins = process(users, (User u) ->      // u.getRole().equals("admin"));        // Replacing it with the following line      // using Stream API and lambda functions      // produces the same output        // the input to the filter() is a lambda      // expression that returns a predicate: a      // boolean value for each user encountered      // (true if admin, false otherwise)      List admins = users.stream()      .filter((user) -> user.getRole().equals("admin"))      .collect(Collectors.toList());        System.out.println(admins);      }  } |

**Output:**

[User Name : John, Role :admin]