1.CHAPTER

PACKAGES & ITS CLASSES

1) packages present in java library ?

Java provides a vast standard library that includes a set of packages containing classes and interfaces that can be used in developing Java applications. Some of the commonly used packages in the Java library include:

1. **java.lang:** This package contains classes that are fundamental to the Java programming language, such as String, Integer, Double, etc.
2. **java.util:** This package contains various utility classes and interfaces, including the collections framework, date and time classes, and various other utility classes.
3. **java.io:** This package contains classes for performing input and output operations, such as reading from and writing to files.
4. **java.net:** This package provides classes for networking functionality, including sockets, URLs, and HTTP requests.
5. **java.awt:** This package contains classes for creating graphical user interfaces (GUIs), including buttons, labels, and text fields.
6. **javax.swing:** This package provides a more advanced set of GUI components, including dialog boxes, menus, and tabbed panes.
7. **java.sql**: This package contains classes for working with databases using the Java Database Connectivity (JDBC) API.
8. **java.security:** This package provides classes for implementing security measures, such as encryption and digital signatures.

These are just a few examples of the many packages available in the Java library. Developers can also create their own packages containing custom classes and interfaces.

A) classes present in java.lang package ?

The **java.lang** package is a built-in package in Java that contains classes that are fundamental to the language. Some of the classes present in the **java.lang** package are:

1. **Object** - The Object class is the root class of all Java classes. It defines a set of methods that every Java class inherits, such as toString(), equals(), and hashCode().
2. **String** - The String class represents a sequence of characters. It is used to store and manipulate text in Java.
3. **Integer** - The Integer class is a wrapper class that allows primitive int values to be treated as objects. It provides methods to convert int values to and from other data types, such as String and long.
4. **Double** - The Double class is also a wrapper class that allows primitive double values to be treated as objects. It provides methods to convert double values to and from other data types, such as String and float.
5. **Boolean** - The Boolean class is a wrapper class that represents a Boolean value, which can be either true or false.
6. **System** - The System class provides access to the system resources, such as standard input, standard output, and error streams. It also contains methods to manage the JVM, such as exit() and gc().

7. **Math** - The Math class provides methods for performing mathematical operations, such as trigonometric functions, exponential and logarithmic functions, and rounding.

These are just a few examples of the classes present in the **java.lang** package. They are automatically imported into every Java program, so developers do not need to explicitly import them.

B) classes present in java.util package ?

The **java.util** package is a built-in package in Java that provides various utility classes and interfaces. Some of the classes present in the **java.util** package are:

1. **ArrayList** - The ArrayList class is a dynamic array implementation that provides methods to add, remove, and access elements of the array.
2. **LinkedList** - The LinkedList class is a linked list implementation that provides methods to add, remove, and access elements of the list.
3. **HashSet** - The HashSet class is a collection that contains no duplicate elements. It provides methods to add, remove, and access elements of the set.
4. **HashMap** - The HashMap class is a collection that maps keys to values. It provides methods to add, remove, and access elements of the map.
5. **Date** - The Date class represents a specific instant in time, with millisecond precision.
6. **Calendar** - The Calendar class is an abstract class that provides methods to manipulate dates and times, such as adding or subtracting time units and formatting dates.
7. **StringTokenizer** - The StringTokenizer class allows a string to be broken into tokens, which are delimited by specified characters.
8. **Scanner** - The Scanner class allows a program to read input from various sources, such as files and user input.

These are just a few examples of the classes present in the **java.util** package. Developers can import these classes and use their methods to implement various functionalities in their Java programs.

C) classes present in java.io package ?

The **java.io** package is a built-in package in Java that provides classes for performing input and output operations. Some of the classes present in the **java.io** package are:

1. **File** - The File class represents a file or directory path in the file system. It provides methods to manipulate files and directories, such as creating, deleting, and renaming files.
2. **FileInputStream** - The FileInputStream class is used to read data from a file in byte-oriented format.
3. **FileOutputStream** - The FileOutputStream class is used to write data to a file in byte-oriented format.
4. **FileReader** - The FileReader class is used to read data from a file in character-oriented format.

5. **FileWriter** - The FileWriter class is used to write data to a file in character-oriented format.
6. **BufferedReader** - The BufferedReader class is used to read text from a character stream with efficiency and performance.
7. **BufferedWriter** - The BufferedWriter class is used to write text to a character stream with efficiency and performance.
8. **ByteArrayInputStream** - The ByteArrayInputStream class is used to read data from an input stream in byte-oriented format.
9. **ByteArrayOutputStream** - The ByteArrayOutputStream class is used to write data to an output stream in byte-oriented format.

These are just a few examples of the classes present in the **java.io** package. Developers can import these classes and use their methods to implement various file input and output operations in their Java programs.

D) classes present in java.net package ?

The **java.net** package is a built-in package in Java that provides classes for networking operations. Some of the classes present in the **java.net** package are:

1. **URL** - The URL class represents a Uniform Resource Locator, which is a reference to a web resource, such as a webpage or a file.
2. **HttpURLConnection** - The HttpURLConnection class is a subclass of URLConnection and is used to make HTTP requests and handle HTTP responses.
3. **InetAddress** - The InetAddress class represents an Internet Protocol (IP) address. It provides methods to get the host name and IP address of a machine.
4. **Socket** - The Socket class is used to create a client-side socket, which can be used to establish a connection to a server over a network.
5. **ServerSocket** - The ServerSocket class is used to create a server-side socket, which can listen for incoming connections from clients.
6. **DatagramPacket** - The DatagramPacket class is used to send or receive packets of data over a network using User Datagram Protocol (UDP).
7. **DatagramSocket** - The DatagramSocket class is used to create a socket for sending or receiving packets of data over a network using User Datagram Protocol (UDP).

These are just a few examples of the classes present in the **java.net** package. Developers can import these classes and use their methods to implement various networking operations in their Java programs.

E) classes present in java.awt package ?

The **java.awt** package is a built-in package in Java that provides classes for creating and managing graphical user interface (GUI) components. Some of the classes present in the **java.awt** package are:

1. **Frame** - The Frame class represents a top-level window with a title and border. It provides methods to add and manage components, such as buttons and labels.

2. **Panel** - The Panel class represents a container for other components, such as buttons and labels. It provides a space for arranging components in a specific layout.
3. **Button** - The Button class represents a push button component. It provides methods to set the label and action of the button.
4. **Label** - The Label class represents a text label component. It provides methods to set the text and font of the label.
5. **TextField** - The TextField class represents a text input field component. It provides methods to set the default text, size, and maximum length of the field.
6. **Checkbox** - The Checkbox class represents a check box component. It provides methods to set the label and state of the check box.
7. **CheckboxGroup** - The CheckboxGroup class represents a group of check boxes. It provides methods to add and manage check boxes in the group.

These are just a few examples of the classes present in the **java.awt** package. Developers can import these classes and use their methods to create and manage GUI components in their Java programs.

F) classes present in javax.swing package ?

The **javax.swing** package is a built-in package in Java that provides classes for creating and managing graphical user interface (GUI) components using a modern approach known as Swing. Some of the classes present in the **javax.swing** package are:

1. **JFrame** - The JFrame class represents a top-level window with a title, border, and menu bar. It provides methods to add and manage components, such as buttons and labels.
2. **JPanel** - The JPanel class represents a container for other components, such as buttons and labels. It provides a space for arranging components in a specific layout.
3. **JButton** - The JButton class represents a push button component. It provides methods to set the label and action of the button.
4. **JLabel** - The JLabel class represents a text label component. It provides methods to set the text and font of the label.
5. **JTextField** - The JTextField class represents a text input field component. It provides methods to set the default text, size, and maximum length of the field.
6. **JCheckBox** - The JCheckBox class represents a check box component. It provides methods to set the label and state of the check box.
7. **JRadioButton -** The JRadioButton class represents a radio button component. It provides methods to set the label and state of the radio button.
8. **JList** - The JList class represents a list component. It provides methods to add and remove items from the list.
9. **JComboBox** - The JComboBox class represents a combo box component. It provides methods to add and select items from a drop-down list.

These are just a few examples of the classes present in the **javax.swing** package. Developers can import these classes and use their methods to create and manage modern GUI components in their Java programs using Swing.

G) classes present in java.sql package ?

The **java.sql** package is a built-in package in Java that provides classes for interacting with relational databases. Some of the classes present in the **java.sql** package are:

1. **Connection** - The Connection interface represents a connection to a database. It provides methods to connect to a database, create statements, and manage transactions.
2. **Statement** - The Statement interface represents a SQL statement that is executed against a database. It provides methods to execute queries and updates, and retrieve results.
3. **PreparedStatement** - The PreparedStatement interface extends the Statement interface and represents a precompiled SQL statement that can be executed multiple times. It provides methods to set parameters and execute queries and updates.
4. **ResultSet** - The ResultSet interface represents a set of results returned from a database query. It provides methods to retrieve data from the result set.
5. **ResultSetMetaData** - The ResultSetMetaData interface provides metadata about a result set, such as the number and types of columns.
6. **DatabaseMetaData** - The DatabaseMetaData interface provides metadata about a database, such as the database name, version, and supported features.
7. **DriverManager** - The DriverManager class provides methods to manage the JDBC drivers that are used to connect to a database.

These are just a few examples of the classes present in the **java.sql** package. Developers can import these classes and use their methods to interact with relational databases in their Java programs using the Java Database Connectivity (JDBC) API.

H) classes present in java.security package ?

The **java.security** package is a built-in package in Java that provides classes for implementing security features, such as encryption and digital signatures. Some of the classes present in the **java.security** package are:

1. **MessageDigest** - The MessageDigest class provides methods to compute a cryptographic hash function, such as SHA-256 or MD5, on input data.
2. **KeyPairGenerator** - The KeyPairGenerator class provides methods to generate pairs of public and private keys for use in digital signatures and encryption.
3. **KeyFactory** - The KeyFactory class provides methods to generate public and private keys from their encoded representations.
4. **KeyStore** - The KeyStore class represents a database of cryptographic keys and certificates. It provides methods to load, store, and manage keys and certificates.
5. **SecureRandom** - The SecureRandom class provides a cryptographically secure random number generator that is used for generating random keys and nonces.
6. **Signature** - The Signature class provides methods to generate and verify digital signatures.

7. **Cipher** - The Cipher class provides methods to encrypt and decrypt data using various cryptographic algorithms.
8. **Provider** - The Provider class represents a security provider, such as the SunJCE provider, that provides cryptographic algorithms and services.

These are just a few examples of the classes present in the **java.security** package. Developers can import these classes and use their methods to implement security features in their Java programs.

2.CHAPTER

CLASSES & ITS METHODS

1) Inbuild methods in object class ?

The **Object** class is the root class of all Java classes and provides some basic methods that are inherited by all classes. The following are some of the commonly used methods in the **Object** class:

1. **public boolean equals(Object obj)**: This method compares the current object with the specified object for equality and returns a boolean value indicating whether they are equal or not.
2. **public int hashCode()**: This method returns the hash code value of the object. The hash code value is used for hashing-based data structures like **HashMap**, **HashSet**, etc.
3. **public String toString()**:This method returns a string representation of the object. By default, it returns the fully qualified name of the class followed by the hash code value of the object in hexadecimal form.
4. **public final Class<?> getClass()**: This method returns the **Class** object that represents the runtime class of the object.
5. **protected Object clone() throws CloneNotSupportedException**: This method creates and returns a copy of the current object. It throws a **CloneNotSupportedException** if the class does not support the **clone()** method.
6. **public final void notify()**: This method wakes up a single thread that is waiting on the object's monitor.
7. **public final void notifyAll()**: This method wakes up all threads that are waiting on the object's monitor.
8. **public final void wait() throws InterruptedException**: This method causes the current thread to wait until another thread calls the **notify()** or **notifyAll()** method for the object, or until the specified amount of time has elapsed.
9. **protected void finalize() throws Throwable**: This method is called by the garbage collector when the object is no longer reachable. It can be overridden to perform finalization tasks such as releasing resources, closing streams, etc.

2) inbuild methods in string class ?

The **String** class in Java provides a number of built-in methods for manipulating and working with strings. Some commonly used methods are:

1. **public int length()**: Returns the length of the string.
2. **public char charAt(int index)**: Returns the character at the specified index in the string.
3. **public boolean equals(Object obj)**: Compares this string to the specified object and returns true if they are equal, false otherwise.
4. **public boolean isEmpty()**: Returns true if the string is empty, false otherwise.
5. **public String substring(int beginIndex)**: Returns a new string that is a substring of the original string, starting from the specified index.

6. **public String substring(int beginIndex, int endIndex)**: Returns a new string that is a substring of the original string, starting from the specified beginIndex and ending at the specified endIndex.
7. **public String replace(char oldChar, char newChar)**: Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.
8. **public boolean contains(CharSequence s)**: Returns true if this string contains the specified sequence of characters, false otherwise.
9. **public String[] split(String regex)**: Splits this string around matches of the given regular expression and returns an array of strings.
10. **public String toUpperCase()**: Returns a new string that is all uppercase.
11. **public String toLowerCase()**: Returns a new string that is all lowercase.
12. **public String trim()**: Returns a new string that is a copy of this string with leading and trailing whitespace removed.
13. **public int compareTo(String anotherString)**: Compares two strings lexicographically and returns an integer value that indicates whether the first string is less than, equal to, or greater than the second string.
14. **public boolean startsWith(String prefix)**: Returns true if this string starts with the specified prefix, false otherwise.
15. **public boolean endsWith(String suffix)**: Returns true if this string ends with the specified suffix, false otherwise.

3) inbuild methods in integer class ?

The **Integer** class in Java provides a number of built-in methods for working with integer values. Some commonly used methods are:

1. **public static int parseInt(String s)**: Parses the string argument as a signed decimal integer and returns the resulting integer value.
2. **public static String toString(int i)**: Returns a string representation of the integer argument as an unsigned decimal value.
3. **public static Integer valueOf(String s)**: Returns an **Integer** object holding the value of the specified **String**.
4. **public static int compare(int x, int y)**: Compares two **int** values numerically and returns the value 0 if **x == y**; a value less than 0 if **x < y**; and a value greater than 0 if **x > y**.
5. **public static int max(int a, int b)**: Returns the greater of two **int** values.
6. **public static int min(int a, int b)**: Returns the smaller of two **int** values.
7. **public static int sum(int a, int b)**: Adds two **int** values together.
8. **public static int parseInt(String s, int radix)**: Parses the string argument as a signed integer in the radix specified by the second argument and returns the resulting integer value.
9. **public static int reverse(int i)**: Reverses the order of the bits in the specified **int** value.
10. **public static int numberOfLeadingZeros(int i)**: Returns the number of leading zeros in the binary representation of the specified **int** value.
11. **public static int numberOfTrailingZeros(int i)**: Returns the number of trailing zeros in the binary representation of the specified **int** value.

12. **public static int rotateLeft(int i, int distance)**: Rotates the bits of the specified **int** value to the left by the specified distance.
13. **public static int rotateRight(int i, int distance)**: Rotates the bits of the specified **int** value to the right by the specified distance.
14. **public static int bitCount(int i)**: Returns the number of one-bits in the two's complement binary representation of the specified **int** value.
15. **public static int toUnsignedLong(int x)**: Converts the specified **int** value to a **long** value with the same bit pattern. The resulting **long** value is therefore positive.

4) inbuild methods in double class ?

The **Double** class in Java provides a number of built-in methods for working with double-precision floating point numbers. Some commonly used methods are:

1. **public static double parseDouble(String s)**: Parses the string argument as a double and returns the resulting double value.
2. **public static String toString(double d)**: Returns a string representation of the double argument.
3. **public static Double valueOf(String s)**: Returns a **Double** object holding the value of the specified **String**.
4. **public static int compare(double d1, double d2)**: Compares two **double** values numerically and returns the value 0 if **d1 == d2**; a value less than 0 if **d1 < d2**; and a value greater than 0 if **d1 > d2**.
5. **public static double max(double a, double b)**: Returns the greater of two **double** values.
6. **public static double min(double a, double b)**: Returns the smaller of two **double** values.
7. **public static double abs(double d)**: Returns the absolute value of a **double** value.
8. **public static double ceil(double d)**: Returns the smallest (closest to negative infinity) **double** value that is greater than or equal to the argument and is equal to a mathematical integer.
9. **public static double floor(double d)**: Returns the largest (closest to positive infinity) **double** value that is less than or equal to the argument and is equal to a mathematical integer.
10. **public static double round(double d)**: Returns the closest **long** to the argument, with ties rounding to positive infinity.
11. **public static double sqrt(double d)**: Returns the correctly rounded positive square root of a **double** value.
12. **public static double pow(double a, double b)**: Returns the value of the first argument raised to the power of the second argument.
13. **public static double log(double d)**: Returns the natural logarithm (base e) of a **double** value.
14. **public static double exp(double d)**: Returns Euler's number **e** raised to the power of a **double** value.
15. **public static double random()**: Returns a **double** value greater than or equal to 0.0 and less than 1.0.

5) inbuild methods in boolean class ?

In Java, the **boolean** data type is a primitive data type, not a class, so it doesn't have any methods of its own. However, the **Boolean** class is a wrapper class that encapsulates a **boolean** value and provides several built-in methods for working with **boolean** values.

Here are some of the inbuilt methods available in the **Boolean** class:

1. **booleanValue():** This method returns the **boolean** value that is encapsulated by the **Boolean** object.
2. **compareTo(Boolean anotherBoolean):** This method compares the **Boolean** object with another **Boolean** object. It returns **0** if the two objects are equal, a value less than **0** if the object is less than the other object, and a value greater than **0** if the object is greater than the other object.
3. **equals(Object obj):** This method compares the **Boolean** object with another object. It returns **true** if the other object is also a **Boolean** object and has the same value as the **Boolean** object.
4. **hashCode():** This method returns the hash code value for the **Boolean** object.
5. **toString():** This method returns a **String** representation of the **Boolean** object. The returned **String** is either **"true"** or **"false"**.
6. **logicalAnd(boolean a, boolean b):** This method performs a logical AND operation between two **boolean** values. It returns **true** if both values are **true** and **false** otherwise.
7. **logicalOr(boolean a, boolean b):** This method performs a logical OR operation between two **boolean** values. It returns **true** if at least one of the values is **true** and **false** otherwise.
8. **logicalXor(boolean a, boolean b)**: This method performs a logical XOR operation between two **boolean** values. It returns **true** if only one of the values is **true** and **false** otherwise.

These are some of the inbuilt methods available in the **Boolean** class in Java.

6) inbuild methods in system class in java?

The **System** class in Java provides several built-in methods for working with system resources, such as input and output streams, environment variables, and the system clock. Here are some of the inbuilt methods available in the **System** class:

1. **currentTimeMillis()**: This method returns the current time in milliseconds since the epoch (January 1, 1970, 00:00:00 GMT).
2. **nanoTime()**: This method returns the current value of the system's high-resolution time source, in nanoseconds.
3. **exit(int status)**: This method terminates the currently running Java Virtual Machine (JVM) with the given status code.
4. **gc()**: This method requests that the JVM run the garbage collector to free up unused memory.

5. **getProperty(String key)**: This method gets the system property associated with the given key. The key is case-insensitive.
6. **getenv(String name)**: This method gets the value of the environment variable with the given name.
7. **setIn(InputStream in)**: This method sets the standard input stream to the specified input stream.
8. **setOut(PrintStream out)**: This method sets the standard output stream to the specified print stream.
9. **setErr(PrintStream err)**: This method sets the standard error stream to the specified print stream.
10. **arraycopy(Object src, int srcPos, Object dest, int destPos, int length)**: This method copies an array from the specified source array, beginning at the specified position, to the specified position of the destination array.

These are some of the inbuilt methods available in the **System** class in Java.

7) inbuild methods in math class in java?

The **Math** class in Java provides several built-in methods for performing mathematical operations. Here are some of the inbuilt methods available in the **Math** class:

1. **abs(int a)**: This method returns the absolute value of the given integer.
2. **abs(double a)**: This method returns the absolute value of the given double.
3. **ceil(double a)**: This method returns the smallest integer that is greater than or equal to the given double.
4. **floor(double a)**: This method returns the largest integer that is less than or equal to the given double.
5. **max(int a, int b)**: This method returns the larger of the two given integers.
6. **max(double a, double b)**: This method returns the larger of the two given doubles.
7. **min(int a, int b)**: This method returns the smaller of the two given integers.
8. **min(double a, double b)**: This method returns the smaller of the two given doubles.
9. **pow(double a, double b)**: This method returns the value of the first argument raised to the power of the second argument.
10. **sqrt(double a)**: This method returns the square root of the given double.
11. **sin(double a)**: This method returns the sine of the given angle (in radians).
12. **cos(double a)**: This method returns the cosine of the given angle (in radians).
13. **tan(double a)**: This method returns the tangent of the given angle (in radians).
14. **log(double a)**: This method returns the natural logarithm (base e) of the given double.
15. **exp(double a)**: This method returns e raised to the power of the given double.

These are some of the inbuilt methods available in the **Math** class in Java.

8) inbuild methods in arraylist class in java?

The **ArrayList** class in Java provides several built-in methods for working with dynamic arrays. Here are some of the inbuilt methods available in the **ArrayList** class:

1. **add(E e)**: This method adds the specified element to the end of the list.

2. **add(int index, E element)**: This method inserts the specified element at the specified position in the list.
3. **addAll(Collection<? extends E> c)**: This method adds all of the elements in the specified collection to the end of the list, in the order in which they are returned by the collection's iterator.
4. **clear()**: This method removes all of the elements from the list.
5. **contains(Object o)**: This method returns true if the list contains the specified element.
6. **get(int index)**: This method returns the element at the specified position in the list.
7. **indexOf(Object o)**: This method returns the index of the first occurrence of the specified element in the list, or -1 if the list does not contain the element.
8. **isEmpty()**: This method returns true if the list contains no elements.
9. **remove(int index)**: This method removes the element at the specified position in the list.
10. **remove(Object o)**: This method removes the first occurrence of the specified element from the list, if it is present.
11. **size()**: This method returns the number of elements in the list.
12. **toArray()**: This method returns an array containing all of the elements in the list, in the same order as they are returned by the list's iterator.

These are some of the inbuilt methods available in the **ArrayList** class in Java.

9) inbuild methods in linkedlist class in java?

The **LinkedList** class in Java provides several built-in methods for working with linked lists. Here are some of the inbuilt methods available in the **LinkedList** class:

1. **add(E e)**: This method adds the specified element to the end of the list.
2. **add(int index, E element)**: This method inserts the specified element at the specified position in the list.
3. **addAll(Collection<? extends E> c)**: This method adds all of the elements in the specified collection to the end of the list, in the order in which they are returned by the collection's iterator.
4. **clear()**: This method removes all of the elements from the list.
5. **contains(Object o)**: This method returns true if the list contains the specified element.
6. **get(int index)**: This method returns the element at the specified position in the list.
7. **indexOf(Object o)**: This method returns the index of the first occurrence of the specified element in the list, or -1 if the list does not contain the element.
8. **isEmpty()**: This method returns true if the list contains no elements.
9. **remove(int index)**: This method removes the element at the specified position in the list.
10. **remove(Object o)**: This method removes the first occurrence of the specified element from the list, if it is present.
11. **size()**: This method returns the number of elements in the list.
12. **toArray()**: This method returns an array containing all of the elements in the list, in the same order as they are returned by the list's iterator.
13. **addFirst(E e)**: This method inserts the specified element at the beginning of the list.

14. **addLast(E e)**: This method appends the specified element to the end of the list.
15. **getFirst()**: This method returns the first element in the list.
16. **getLast()**: This method returns the last element in the list.
17. **removeFirst()**: This method removes and returns the first element in the list.
18. **removeLast()**: This method removes and returns the last element in the list.

These are some of the inbuilt methods available in the **LinkedList** class in Java.

10) inbuild methods in hashset class in java?

The **HashSet** class in Java provides several built-in methods for working with sets. Here are some of the inbuilt methods available in the **HashSet** class:

1. **add(E e)**: This method adds the specified element to the set if it is not already present.
2. **addAll(Collection<? extends E> c)**: This method adds all of the elements in the specified collection to the set if they are not already present.
3. **clear()**: This method removes all of the elements from the set.
4. **contains(Object o)**: This method returns true if the set contains the specified element.
5. **isEmpty()**: This method returns true if the set contains no elements.
6. **iterator()**: This method returns an iterator over the elements in the set.
7. **remove(Object o)**: This method removes the specified element from the set if it is present.
8. **size()**: This method returns the number of elements in the set.
9. **toArray()**: This method returns an array containing all of the elements in the set.

These are some of the inbuilt methods available in the **HashSet** class in Java. It's worth noting that **HashSet** is an implementation of the **Set** interface in Java, and as such, it inherits all of the methods of that interface as well.

11) inbuild methods in hashmap class in java?

The **HashMap** class in Java provides several built-in methods for working with hash maps. Here are some of the inbuilt methods available in the **HashMap** class:

1. **put(K key, V value)**: This method associates the specified value with the specified key in the map.
2. **get(Object key)**: This method returns the value to which the specified key is mapped, or null if the map contains no mapping for the key.
3. **containsKey(Object key)**: This method returns true if the map contains a mapping for the specified key.
4. **containsValue(Object value)**: This method returns true if the map contains one or more mappings to the specified value.
5. **entrySet()**: This method returns a set view of the mappings contained in the map.
6. **keySet()**: This method returns a set view of the keys contained in the map.
7. **values()**: This method returns a collection view of the values contained in the map.
8. **remove(Object key)**: This method removes the mapping for the specified key from the map if it is present.

9. **size()**: This method returns the number of mappings in the map.
10. **clear()**: This method removes all mappings from the map.
11. **isEmpty()**: This method returns true if the map contains no mappings.
12. **putAll(Map<? extends K, ? extends V> m)**: This method copies all of the mappings from the specified map to this map.

These are some of the inbuilt methods available in the **HashMap** class in Java. It's worth noting that **HashMap** is an implementation of the **Map** interface in Java, and as such, it inherits all of the methods of that interface as well.

12) inbuild methods in date class in java?

The **Date** class in Java provides several built-in methods for working with dates. However, it's worth noting that the **Date** class has been mostly deprecated in favor of the newer **java.time** package introduced in Java 8. Here are some of the inbuilt methods available in the **Date** class:

1. **after(Date when)**: This method returns true if this date is after the specified date.
2. **before(Date when)**: This method returns true if this date is before the specified date.
3. **compareTo(Date anotherDate)**: This method compares this date to another date and returns an integer indicating whether this date is before, after, or equal to the other date.
4. **equals(Object obj)**: This method returns true if this date is equal to the specified object.
5. **getTime()**: This method returns the time value of this date as the number of milliseconds since January 1, 1970, 00:00:00 GMT.
6. **toString()**: This method returns a string representation of this date.
7. **hashCode()**: This method returns a hash code value for this date.

These are some of the inbuilt methods available in the **Date** class in Java. However, as mentioned earlier, it is recommended to use the **java.time** package for working with dates and times in Java as it provides a more modern and robust API.

13) inbuild methods in calender class in java?

The **Calendar** class in Java provides several built-in methods for working with dates and times. Here are some of the inbuilt methods available in the **Calendar** class:

1. **getInstance()**: This method returns a **Calendar** object initialized with the current date and time in the default time zone.
2. **get(int field)**: This method returns the value of the given calendar field.
3. **set(int field, int value)**: This method sets the value of the given calendar field.
4. **add(int field, int amount)**: This method adds the specified amount to the given calendar field.
5. **roll(int field, int amount)**: This method adds the specified amount to the given calendar field without changing the larger fields.
6. **getTime()**: This method returns a **Date** object representing this **Calendar**'s time value.

7. **getActualMaximum(int field)**: This method returns the maximum value for the given calendar field, taking into account the current date and time.
8. **getActualMinimum(int field)**: This method returns the minimum value for the given calendar field, taking into account the current date and time.
9. **getDisplayName(int field, int style, Locale locale)**: This method returns the display name of the given calendar field in the specified style and locale.

These are some of the inbuilt methods available in the **Calendar** class in Java. However, it's worth noting that the **Calendar** class has been mostly deprecated in favor of the newer **java.time** package introduced in Java 8, which provides a more modern and robust API for working with dates and times.

14) inbuild methods in stringtokenizer class in java?

The **StringTokenizer** class in Java provides several built-in methods for working with strings. Here are some of the inbuilt methods available in the **StringTokenizer** class:

1. **hasMoreTokens()**: This method returns **true** if there are more tokens available in the string.
2. **nextToken()**: This method returns the next token in the string.
3. **countTokens()**: This method returns the number of tokens remaining in the string.
4. **nextElement()**: This method returns the next token in the string as an object.
5. **hasMoreElements()**: This method returns **true** if there are more tokens available in the string.

These are some of the inbuilt methods available in the **StringTokenizer** class in Java. The **StringTokenizer** class is often used to break a string into tokens based on a specified delimiter or set of delimiters. It is useful for parsing strings that contain multiple values separated by a delimiter.

15) inbuild methods in scanner class in java?

The **Scanner** class in Java provides several built-in methods for reading input from various sources. Here are some of the inbuilt methods available in the **Scanner** class:

1. **hasNext()**: This method returns **true** if there is another token in the input.
2. **hasNextLine()**: This method returns **true** if there is another line of input available.
3. **next()**: This method returns the next token in the input as a string.
4. **nextLine()**: This method returns the next line of input as a string.
5. **nextInt()**: This method returns the next token in the input as an integer.
6. **nextDouble()**: This method returns the next token in the input as a double.
7. **useDelimiter(String pattern)**: This method sets the delimiter pattern to use for breaking the input into tokens.

These are some of the inbuilt methods available in the **Scanner** class in Java. The **Scanner** class is often used for reading input from the console or from a file. It is a very useful class for interactive command-line programs, where the user is prompted for input and the program responds accordingly.

16) inbuild methods in file class in java?

The **File** class in Java provides several built-in methods for working with files and directories. Here are some of the inbuilt methods available in the **File** class:

1. **exists()**: This method returns **true** if the file or directory exists.
2. **isFile()**: This method returns **true** if the **File** object represents a file.
3. **isDirectory()**: This method returns **true** if the **File** object represents a directory.
4. **getName()**: This method returns the name of the file or directory.
5. **getPath()**: This method returns the path of the file or directory.
6. **canRead()**: This method returns **true** if the file or directory can be read.
7. **canWrite()**: This method returns **true** if the file or directory can be written to.
8. **length()**: This method returns the length of the file in bytes.
9. **delete()**: This method deletes the file or directory.
10. **mkdir()**: This method creates a new directory.
11. **mkdirs()**: This method creates a new directory and any necessary parent directories.
12. **list()**: This method returns an array of strings naming the files and directories in the directory.
13. **listFiles()**: This method returns an array of **File** objects representing the files and directories in the directory.

These are some of the inbuilt methods available in the **File** class in Java. The **File** class is often used for working with files and directories on the file system. It is a very useful class for performing file I/O operations in Java.

17) inbuild methods in fileinputstream class in java?

The **FileInputStream** class in Java provides several built-in methods for reading data from a file. Here are some of the inbuilt methods available in the **FileInputStream** class:

1. **available()**: This method returns the number of bytes that can be read from the input stream.
2. **read()**: This method reads the next byte of data from the input stream and returns it as an integer value.
3. **read(byte[] b)**: This method reads up to **b.length** bytes of data from the input stream into an array of bytes.
4. **read(byte[] b, int off, int len)**: This method reads up to **len** bytes of data from the input stream into an array of bytes, starting at the specified offset.
5. **skip(long n)**: This method skips over and discards **n** bytes of data from the input stream.
6. **close()**: This method closes the input stream.

These are some of the inbuilt methods available in the **FileInputStream** class in Java. The **FileInputStream** class is often used for reading binary data from a file. It is a very useful class for performing file I/O operations in Java.

18) inbuild methods in fileoutputstream class in java?

The **FileOutputStream** class in Java provides several built-in methods for writing data to a file. Here are some of the inbuilt methods available in the **FileOutputStream** class:

1. **write(int b)**: This method writes the specified byte to the output stream.
2. **write(byte[] b)**: This method writes the specified array of bytes to the output stream.
3. **write(byte[] b, int off, int len)**: This method writes **len** bytes from the specified array of bytes, starting at the specified offset, to the output stream.
4. **close()**: This method closes the output stream.
5. **flush()**: This method flushes the output stream.

These are some of the inbuilt methods available in the **FileOutputStream** class in Java. The **FileOutputStream** class is often used for writing binary data to a file. It is a very useful class for performing file I/O operations in Java. Note that if you want to write text data to a file, it is recommended to use the **BufferedWriter** class, which provides additional methods for writing text data to a file.

19) inbuild methods in filereader class in java?

The **FileReader** class in Java provides several built-in methods for reading text data from a file. Here are some of the inbuilt methods available in the **FileReader** class:

1. **read()**: This method reads a single character from the input stream and returns it as an integer value.
2. **read(char[] cbuf)**: This method reads characters from the input stream into an array of characters.
3. **read(char[] cbuf, int off, int len)**: This method reads up to **len** characters from the input stream into an array of characters, starting at the specified offset.
4. **skip(long n)**: This method skips over and discards **n** characters of data from the input stream.
5. **close()**: This method closes the input stream.

These are some of the inbuilt methods available in the **FileReader** class in Java. The **FileReader** class is often used for reading text data from a file. It is a very useful class for performing file I/O operations in Java. Note that if you want to read text data from a file with better performance, it is recommended to use the **BufferedReader** class, which provides additional methods for reading text data from a file.

20) inbuild methods in filerwriter class in java?

The **FileWriter** class in Java provides several built-in methods for writing text data to a file. Here are some of the inbuilt methods available in the **FileWriter** class:

1. **write(int c)**: This method writes a single character to the output stream.
2. **write(String str)**: This method writes a string of characters to the output stream.
3. **write(char[] cbuf)**: This method writes an array of characters to the output stream.

4. **write(char[] cbuf, int off, int len)**: This method writes a portion of an array of characters to the output stream, starting at the specified offset.
5. **close()**: This method closes the output stream.
6. **flush()**: This method flushes the output stream.

These are some of the inbuilt methods available in the **FileWriter** class in Java. The **FileWriter** class is often used for writing text data to a file. It is a very useful class for performing file I/O operations in Java. Note that if you want to write text data to a file with better performance, it is recommended to use the **BufferedWriter** class, which provides additional methods for writing text data to a file.

21) inbuild methods in bufferedreader class in java?

The **BufferedReader** class in Java provides several built-in methods to read data from a character stream with buffering for efficient reading. Some of the common methods are:

1. **read() -** Reads a single character from the input stream and returns the character as an integer value.
2. **read(char[] cbuf, int off, int len) -** Reads characters into a portion of an array. The method reads up to **len** characters from the input stream and stores them into the character array **cbuf**, starting at the array offset **off**.
3. **readLine() -** Reads a line of text from the input stream and returns it as a string. The method returns **null** if the end of the stream is reached.
4. **skip(long n)** - Skips **n** characters from the input stream. The method returns the actual number of characters skipped.
5. **reset() -** Resets the input stream to its last marked position. This method throws an **IOException** if the stream has not been marked.
6. **mark(int readAheadLimit) -** Marks the current position in the input stream. The method sets a limit on the number of characters that may be read while still preserving the mark.
7. **close() -** Closes the input stream and releases any system resources associated with the stream.

There are also several other methods available in the **BufferedReader** class for handling specific types of input, such as **readDouble()**, **readBoolean()**, **readLineToCharArray()**, and more.

22) inbuild methods in bufferedwriter class in java?

The **BufferedWriter** class in Java provides several built-in methods to write data to a character stream with buffering for efficient writing. Some of the common methods are:

1. **write(int c) -** Writes a single character to the output stream.
2. **write(char[] cbuf, int off, int len) -** Writes a portion of an array of characters to the output stream. The method writes **len** characters from the character array **cbuf**, starting at the array offset **off**.
3. **write(String str, int off, int len)** - Writes a portion of a string to the output stream. The method writes **len** characters from the string **str**, starting at the string offset **off**.
4. **newLine() -** Writes a line separator to the output stream.
5. **flush()** - Flushes the output buffer to the underlying stream.
6. **close() -** Closes the output stream and releases any system resources associated with the stream.

There are also several other methods available in the **BufferedWriter** class for handling specific types of output, such as **write(double d)**, **write(boolean b)**, **write(char c)**, and more.

23) inbuild methods in bytearrayinputsystem class in java?

The **ByteArrayInputStream** class in Java provides several built-in methods for reading bytes from an in-memory byte array. Some of the common methods are:

1. **read() -** Reads the next byte of data from the input stream and returns it as an integer in the range 0 to 255.
2. **read(byte[] b, int off, int len) -** Reads up to **len** bytes of data from the input stream into an array of bytes. The method reads bytes into the buffer array **b**, starting at the offset **off**.
3. **skip(long n) -** Skips **n** bytes of data from the input stream. The method returns the actual number of bytes skipped.
4. **available()** - Returns the number of bytes that can be read from the input stream without blocking.
5. **mark(int readlimit) -** Sets a mark in the input stream. The method allows up to **readlimit** bytes to be read before the mark position is invalidated.
6. **reset() -** Resets the input stream to the last marked position.
7. **close() -** Closes the input stream and releases any system resources associated with the stream.

These methods allow you to read bytes from a byte array in a flexible and efficient manner.

24) inbuild methods in bytearrayoutputsystem class in java?

The **ByteArrayOutputStream** class in Java provides several built-in methods for writing bytes to an in-memory byte array. Some of the common methods are:

1. **write(int b) -** Writes a byte to the output stream.
2. **write(byte[] b, int off, int len) -** Writes **len** bytes from the specified byte array starting at offset **off**.
3. **toByteArray() -** Returns the current contents of the output stream as a byte array.
4. **reset() -** Resets the output stream to zero.
5. **size() -** Returns the current size of the output stream.
6. **close() -** Closes the output stream and releases any system resources associated with the stream.

These methods allow you to write bytes to a byte array in a flexible and efficient manner. You can then retrieve the resulting byte array using the **toByteArray()** method. The **reset()** method is particularly useful for resetting the output stream to zero so that it can be reused to write new data.

25) inbuild methods in url class in java?

The **URL** class in Java provides several built-in methods for working with URLs. Some of the common methods are:

1. **openStream() -** Opens a connection to the URL and returns an input stream for reading from the connection.

2. **openConnection() -** Opens a connection to the URL and returns a URLConnection object that can be used to read or write to the connection.
3. **getFile() -** Returns the file name of the URL.
4. **getHost() -** Returns the host name of the URL.
5. **getPath() -** Returns the path of the URL.
6. **getProtocol() -** Returns the protocol of the URL.
7. **getQuery() -** Returns the query string of the URL.
8. **toString() -** Returns a string representation of the URL.

These methods allow you to work with URLs in a flexible and efficient manner. You can open connections to URLs for reading or writing data, retrieve various parts of the URL, and convert URLs to string representations for display or storage. Additionally, the **URL** class provides methods for working with different types of URLs, such as **file:** and **http:** URLs.

26) inbuild methods in httpurlconnection class in java?

The **HttpURLConnection** class in Java extends the **URLConnection** class and provides additional methods for working specifically with HTTP connections. Some of the common methods are:

1. **setRequestMethod(String method)** - Sets the request method to be used for the HTTP request, such as GET, POST, PUT, DELETE, etc.
2. **setRequestProperty(String key, String value)** - Sets the value of the specified request header field.
3. **getResponseCode()** - Returns the HTTP response status code.
4. **getResponseMessage()** - Returns the HTTP response status message.
5. **getHeaderField(String name)** - Returns the value of the specified header field.
6. **getHeaderFields()** - Returns a Map of the header fields for the HTTP response.
7. **connect()** - Opens a connection to the URL and prepares the HTTP request.
8. **getInputStream()** - Returns an input stream for reading the HTTP response.
9. **getOutputStream()** - Returns an output stream for writing the HTTP request body.
10. **disconnect()** - Disconnects the HTTP connection and releases any system resources associated with the connection.

These methods allow you to work with HTTP connections in a flexible and efficient manner. You can set the request method and request headers, retrieve the HTTP response status code and message, read the response body, write the request body, and disconnect from the server when you are done. Additionally, the **HttpURLConnection** class provides methods for handling redirects and managing authentication.

27) inbuild methods in inetaddress class in java?

The **InetAddress** class in Java provides several built-in methods for working with IP addresses and host names. Some of the common methods are:

1. **getByName(String host)** - Returns an InetAddress object representing the specified host name.
2. **getHostAddress()** - Returns the IP address string in textual presentation format.

3. **getHostName()** - Returns the host name for this IP address.
4. **getCanonicalHostName()** - Returns the fully qualified domain name for this IP address.
5. **isReachable(int timeout)** - Tries to reach this IP address for the specified timeout (in milliseconds).
6. **toString()** - Returns a string representation of the IP address.

These methods allow you to work with IP addresses and host names in a flexible and efficient manner. You can retrieve the InetAddress object for a given host name, retrieve the IP address and host name strings, determine if an IP address is reachable, and convert an InetAddress object to a string representation. Additionally, the **InetAddress** class provides methods for working with IP addresses in a byte array format.

28) inbuild methods in socket class in java?

The **Socket** class in Java provides several built-in methods for working with sockets. Some of the common methods are:

1. **connect(SocketAddress endpoint)** - Connects the socket to the specified endpoint.
2. **getInputStream()** - Returns an input stream for reading data from the socket.
3. **getOutputStream()** - Returns an output stream for writing data to the socket.
4. **getLocalAddress()** - Returns the local address to which the socket is bound.
5. **getLocalPort()** - Returns the local port number to which the socket is bound.
6. **getRemoteSocketAddress()** - Returns the remote socket address to which the socket is connected.
7. **getReceiveBufferSize()** - Returns the size of the receive buffer for the socket.
8. **setReceiveBufferSize(int size)** - Sets the size of the receive buffer for the socket.
9. **getSendBufferSize()** - Returns the size of the send buffer for the socket.
10. **setSendBufferSize(int size)** - Sets the size of the send buffer for the socket.
11. **isConnected()** - Returns true if the socket is connected to a remote endpoint.

These methods allow you to work with sockets in a flexible and efficient manner. You can connect to remote endpoints, read and write data to the socket, retrieve information about the local and remote socket addresses and port numbers, and set the send and receive buffer sizes for the socket. Additionally, the **Socket** class provides methods for setting various socket options and for shutting down the input and/or output streams for the socket.

29) inbuild methods in datagrampacket class in java?

The **DatagramPacket** class in Java provides several built-in methods for working with datagrams, which are packets of data that are sent over a network using the UDP protocol. Some of the common methods are:

1. **DatagramPacket(byte[] buf, int length)** - Constructs a datagram packet for sending a message of the given length to the specified port number on the specified host.
2. **DatagramPacket(byte[] buf, int offset, int length, SocketAddress address)** - Constructs a datagram packet for sending a message of the given length to the specified socket address.

3. **getData()** - Returns the buffer containing the data to be sent or received.
4. **getLength()** - Returns the length of the data to be sent or received.
5. **getSocketAddress()** - Returns the socket address of the datagram packet.
6. **setSocketAddress(SocketAddress address)** - Sets the socket address of the datagram packet.
7. **setLength(int length)** - Sets the length of the data to be sent or received.
8. **setData(byte[] buf)** - Sets the data to be sent or received.
9. **setData(byte[] buf, int offset, int length)** - Sets a subset of the data to be sent or received.
10. **getAddress()** - Returns the IP address of the datagram packet.
11. **getPort()** - Returns the port number of the datagram packet.

These methods allow you to work with datagram packets in a flexible and efficient manner. You can construct datagram packets for sending and receiving data, retrieve and set the data buffer and length, retrieve and set the socket address, and retrieve the IP address and port number of the datagram packet. Additionally, the **DatagramPacket** class provides methods for working with datagrams in a byte array format.

30) inbuild methods in datagramsocket class in java?

The **DatagramSocket** class in Java provides several built-in methods for working with datagram sockets, which are used to send and receive datagram packets over a network using the UDP protocol. Some of the common methods are:

1. **DatagramSocket()** - Constructs a datagram socket and binds it to any available port on the local host.
2. **DatagramSocket(int port)** - Constructs a datagram socket and binds it to the specified port on the local host.
3. **bind(SocketAddress address)** - Binds the datagram socket to the specified local socket address.
4. **getLocalAddress()** - Returns the local address to which the datagram socket is bound.
5. **getLocalPort()** - Returns the local port number to which the datagram socket is bound.
6. **getBroadcast()** - Returns the setting for the SO_BROADCAST option.
7. **setBroadcast(boolean on)** - Sets the SO_BROADCAST option to the specified value.
8. **getReceiveBufferSize()** - Returns the size of the receive buffer for the datagram socket.
9. **setReceiveBufferSize(int size)** - Sets the size of the receive buffer for the datagram socket.
10. **getSendBufferSize()** - Returns the size of the send buffer for the datagram socket.
11. **setSendBufferSize(int size)** - Sets the size of the send buffer for the datagram socket.
12. **send(DatagramPacket p)** - Sends a datagram packet from this socket.
13. **receive(DatagramPacket p)** - Receives a datagram packet from this socket.

These methods allow you to work with datagram sockets in a flexible and efficient manner. You can bind the socket to a local socket address, retrieve information about the local socket

address and port number, set and retrieve socket options such as the broadcast option and buffer sizes, and send and receive datagram packets. Additionally, the **DatagramSocket** class provides methods for joining and leaving multicast groups, setting the time-to-live (TTL) value for multicast packets, and disabling or enabling the SO_REUSEADDR socket option.

31) inbuild methods in frame class in java?

The **Frame** class in Java is an abstract class that provides the common functionality required by all frames or windows in a graphical user interface (GUI) application. Some of the common methods in the **Frame** class are:

1. **setTitle(String title)** - Sets the title of the frame to the specified string.
2. **getTitle()** - Returns the title of the frame as a string.
3. **setSize(int width, int height)** - Sets the size of the frame to the specified width and height.
4. **getSize()** - Returns the size of the frame as a **Dimension** object.
5. **setResizable(boolean resizable)** - Enables or disables user resizing of the frame.
6. **setLocation(int x, int y)** - Sets the location of the frame to the specified coordinates.
7. **setVisible(boolean visible)** - Sets the visibility of the frame to the specified value.
8. **setIconImage(Image image)** - Sets the image to be displayed as the icon for the frame.
9. **toFront()** - Brings the frame to the front of the display.
10. **toBack()** - Sends the frame to the back of the display.

These methods allow you to work with frames in a flexible and efficient manner. You can set the title, size, and location of the frame, enable or disable user resizing, set the visibility of the frame, set the icon image for the frame, and bring the frame to the front or send it to the back of the display. Additionally, the **Frame** class provides methods for working with menus, menu bars, and other components that can be added to the frame. However, since the **Frame** class is abstract, you would typically use one of its subclasses, such as **JFrame** or **Dialog**, which provide additional functionality for creating and managing frames in a GUI application.

32) inbuild methods in panel class in java?

The **Panel** class in Java is a lightweight container that provides a basic layout functionality for its child components. Some of the common methods in the **Panel** class are:

1. **setLayout(LayoutManager layout)** - Sets the layout manager for the panel.
2. **add(Component comp)** - Adds the specified component to the panel.
3. **remove(Component comp)** - Removes the specified component from the panel.
4. **removeAll()** - Removes all components from the panel.
5. **getComponents()** - Returns an array of all the components in the panel.
6. **setPreferredSize(Dimension preferredSize)** - Sets the preferred size of the panel.
7. **getPreferredSize()** - Returns the preferred size of the panel.
8. **setSize(Dimension size)** - Sets the size of the panel.
9. **getSize()** - Returns the size of the panel.

10. **setVisible(boolean visible)** - Sets the visibility of the panel to the specified value.

These methods allow you to work with panels in a flexible and efficient manner. You can set the layout manager for the panel, add and remove child components, get the list of components in the panel, set and get the preferred size of the panel, set and get the size of the panel, and set the visibility of the panel. Additionally, the **Panel** class provides methods for working with the graphics context of the panel, such as **paint(Graphics g)** and **repaint()**, which can be overridden in a subclass to provide custom drawing functionality.

33) inbuild methods in button class in java?

The **Button** class in Java represents a button component that can be added to a user interface (UI). Some of the common methods in the **Button** class are:

1. **setLabel(String label)** - Sets the label of the button to the specified string.
2. **getLabel()** - Returns the label of the button as a string.
3. **setEnabled(boolean enabled)** - Enables or disables the button.
4. **addActionListener(ActionListener listener)** - Adds an action listener to the button.
5. **removeActionListener(ActionListener listener)** - Removes an action listener from the button.
6. **setActionCommand(String command)** - Sets the action command for the button.
7. **getActionCommand()** - Returns the action command for the button.
8. **setPreferredSize(Dimension preferredSize)** - Sets the preferred size of the button.
9. **getPreferredSize()** - Returns the preferred size of the button.
10. **setVisible(boolean visible)** - Sets the visibility of the button to the specified value.

These methods allow you to work with buttons in a flexible and efficient manner. You can set the label of the button, enable or disable the button, add or remove an action listener to handle events when the button is clicked, set and get the action command for the button, set and get the preferred size of the button, and set the visibility of the button. Additionally, the **Button** class provides methods for working with the graphics context of the button, such as **paint(Graphics g)** and **repaint()**, which can be overridden in a subclass to provide custom drawing functionality.

34) inbuild methods in label class in java?

The **Label** class in Java represents a component that displays a single line of read-only text or an image. Some of the common methods in the **Label** class are:

1. **setText(String text)** - Sets the text for the label.
2. **getText()** - Returns the text of the label as a string.
3. **setImage(Image image)** - Sets the image for the label.
4. **getImage()** - Returns the image of the label as an object of the **Image** class.
5. **setAlignment(int alignment)** - Sets the alignment of the label text or image.
6. **getAlignment()** - Returns the alignment of the label text or image.
7. **setPreferredSize(Dimension preferredSize)** - Sets the preferred size of the label.
8. **getPreferredSize()** - Returns the preferred size of the label.
9. **setVisible(boolean visible)** - Sets the visibility of the label to the specified value.

These methods allow you to work with labels in a flexible and efficient manner. You can set the text or image for the label, set the alignment of the label text or image, set and get the preferred size of the label, and set the visibility of the label. Additionally, the **Label** class provides methods for working with the graphics context of the label, such as **paint(Graphics g)** and **repaint()**, which can be overridden in a subclass to provide custom drawing functionality.

35) inbuild methods in textfield class in java?

The **TextField** class in Java represents a component that allows the user to enter a single line of text. Some of the common methods in the **TextField** class are:

1. **setText(String text)** - Sets the text of the text field to the specified string.
2. **getText()** - Returns the text in the text field as a string.
3. **setEditable(boolean editable)** - Sets whether the text field is editable or not.
4. **isEditable()** - Returns **true** if the text field is editable, **false** otherwise.
5. **setCaretPosition(int position)** - Sets the position of the caret (text insertion point) in the text field.
6. **getCaretPosition()** - Returns the position of the caret in the text field.
7. **setSelectionStart(int position)** - Sets the start position of the selected text in the text field.
8. **getSelectionStart()** - Returns the start position of the selected text in the text field.
9. **setSelectionEnd(int position)** - Sets the end position of the selected text in the text field.
10. **getSelectionEnd()** - Returns the end position of the selected text in the text field.
11. **setColumns(int columns)** - Sets the number of columns in the text field.
12. **getColumns()** - Returns the number of columns in the text field.
13. **setPreferredSize(Dimension preferredSize)** - Sets the preferred size of the text field.
14. **getPreferredSize()** - Returns the preferred size of the text field.
15. **setVisible(boolean visible)** - Sets the visibility of the text field to the specified value.

These methods allow you to work with text fields in a flexible and efficient manner. You can set the text of the text field, control whether the text field is editable or not, set and get the position of the caret and the start and end positions of the selected text, set the number of columns in the text field, and set and get the preferred size of the text field. Additionally, the **TextField** class provides methods for working with the graphics context of the text field, such as **paint(Graphics g)** and **repaint()**, which can be overridden in a subclass to provide custom drawing functionality.

36) inbuild methods in checkbox class in java?

The **Checkbox** class in Java represents a component that allows the user to select or deselect a boolean value. Some of the common methods in the **Checkbox** class are:

1. **setLabel(String label)** - Sets the label for the checkbox.
2. **getLabel()** - Returns the label of the checkbox as a string.

3. **setState(boolean state)** - Sets the state of the checkbox to the specified boolean value.
4. **getState()** - Returns the state of the checkbox as a boolean value.
5. **setSelected(boolean selected)** - Sets whether the checkbox is selected or not.
6. **isSelected()** - Returns **true** if the checkbox is selected, **false** otherwise.
7. **addItemListener(ItemListener listener)** - Adds the specified item listener to the checkbox.
8. **removeItemListener(ItemListener listener)** - Removes the specified item listener from the checkbox.
9. **setPreferredSize(Dimension preferredSize)** - Sets the preferred size of the checkbox.
10. **getPreferredSize()** - Returns the preferred size of the checkbox.
11. **setVisible(boolean visible)** - Sets the visibility of the checkbox to the specified value.

These methods allow you to work with checkboxes in a flexible and efficient manner. You can set the label for the checkbox, set and get the state and selection of the checkbox, add and remove item listeners to the checkbox, set and get the preferred size of the checkbox, and set the visibility of the checkbox. Additionally, the **Checkbox** class provides methods for working with the graphics context of the checkbox, such as **paint(Graphics g)** and **repaint()**, which can be overridden in a subclass to provide custom drawing functionality.

37) inbuild methods in checkboxgroup class in java?

The **CheckboxGroup** class in Java represents a group of checkboxes in which only one checkbox can be selected at a time. Some of the common methods in the **CheckboxGroup** class are:

1. **getSelectedCheckbox()** - Returns the currently selected checkbox in the group.
2. **setSelectedCheckbox(Checkbox checkbox)** - Sets the currently selected checkbox in the group to the specified checkbox.
3. **getSelectedIndex()** - Returns the index of the currently selected checkbox in the group.
4. **setSelectedIndex(int index)** - Sets the currently selected checkbox in the group to the checkbox at the specified index.
5. **getSelectedObjects()** - Returns an array of the currently selected checkboxes in the group as objects.
6. **getSelectedCheckbox(ItemEvent e)** - Returns the selected checkbox in the group that generated the specified item event.
7. **addItemListener(ItemListener listener)** - Adds the specified item listener to the checkbox group.
8. **removeItemListener(ItemListener listener)** - Removes the specified item listener from the checkbox group.

These methods allow you to work with checkbox groups in a flexible and efficient manner. You can get and set the currently selected checkbox in the group, get the index of the currently selected checkbox, get an array of the currently selected checkboxes as objects, and get the selected checkbox that generated a specified item event. Additionally, the

CheckboxGroup class provides methods for adding and removing item listeners from the group, which can be used to monitor changes to the selected checkbox.

38) inbuild methods in jframe class in java?

The **JFrame** class in Java is a top-level container that represents the main window of a GUI application. Some of the common methods in the **JFrame** class are:

1. **add(Component comp)** - Adds the specified component to the content pane of the frame.
2. **remove(Component comp)** - Removes the specified component from the content pane of the frame.
3. **setLayout(LayoutManager mgr)** - Sets the layout manager for the content pane of the frame.
4. **setDefaultCloseOperation(int operation)** - Sets the default operation that occurs when the frame is closed.
5. **setTitle(String title)** - Sets the title of the frame.
6. **setSize(int width, int height)** - Sets the size of the frame.
7. **setLocation(int x, int y)** - Sets the location of the frame.
8. **setVisible(boolean b)** - Sets the visibility of the frame.
9. **setResizable(boolean resizable)** - Sets whether the frame is resizable or not.
10. **setIconImage(Image image)** - Sets the icon for the frame.
11. **getContentPane()** - Returns the content pane of the frame.
12. **setJMenuBar(JMenuBar menubar)** - Sets the menu bar for the frame.
13. **validate()** - Validates the frame and its contents.
14. **repaint()** - Repaints the frame.

These methods allow you to work with frames in a flexible and efficient manner. You can add and remove components from the content pane of the frame, set the layout manager for the content pane, set the default operation that occurs when the frame is closed, set the title, size, location, visibility, and resizability of the frame, set the icon for the frame, get the content pane of the frame, set the menu bar for the frame, validate the frame and its contents, and repaint the frame. Additionally, the **JFrame** class provides methods for working with the graphics context of the frame, such as **paint(Graphics g)** and **repaint()**, which can be overridden in a subclass to provide custom drawing functionality.

39) inbuild methods in jpanel class in java?

The **JPanel** class in Java is a container that can be used to group and organize other components. Some of the common methods in the **JPanel** class are:

1. **add(Component comp)** - Adds the specified component to the panel.
2. **remove(Component comp)** - Removes the specified component from the panel.
3. **setLayout(LayoutManager mgr)** - Sets the layout manager for the panel.
4. **validate()** - Validates the panel and its contents.
5. **repaint()** - Repaints the panel.
6. **setBorder(Border border)** - Sets the border for the panel.
7. **setOpaque(boolean isOpaque)** - Sets whether the panel is opaque or not.

8. **getPreferredSize()** - Returns the preferred size of the panel.
9. **getMinimumSize()** - Returns the minimum size of the panel.
10. **getMaximumSize()** - Returns the maximum size of the panel.
11. **getLayout()** - Returns the layout manager for the panel.
12. **getComponent(int n)** - Returns the nth component in the panel.
13. **getComponentCount()** - Returns the number of components in the panel.
14. **getComponents()** - Returns an array of all the components in the panel.

These methods allow you to work with panels in a flexible and efficient manner. You can add and remove components from the panel, set the layout manager for the panel, validate the panel and its contents, and repaint the panel. Additionally, the **JPanel** class provides methods for working with the size of the panel, such as **getPreferredSize()**, **getMinimumSize()**, and **getMaximumSize()**, which can be overridden in a subclass to provide custom sizing functionality. The **JPanel** class also provides methods for working with the components in the panel, such as **getComponent(int n)**, **getComponentCount()**, and **getComponents()**, which can be used to access and manipulate the components in the panel.

40) inbuild methods in jbutton class in java?

The **JButton** class in Java is used to create a push button control. Some of the common methods in the **JButton** class are:

1. **addActionListener(ActionListener listener)** - Adds an **ActionListener** to the button to handle button events.
2. **setText(String text)** - Sets the text on the button.
3. **setToolTipText(String text)** - Sets the tooltip text for the button.
4. **setIcon(Icon icon)** - Sets the icon for the button.
5. **setEnabled(boolean enabled)** - Enables or disables the button.
6. **isSelected()** - Returns **true** if the button is selected, **false** otherwise.
7. **setSelected(boolean selected)** - Sets the selection state of the button.
8. **getModel()** - Returns the **ButtonModel** that represents the state of the button.
9. **setMnemonic(int mnemonic)** - Sets the mnemonic for the button.
10. **setMargin(Insets insets)** - Sets the margin for the button.
11. **setFont(Font font)** - Sets the font for the button.
12. **getPreferredSize()** - Returns the preferred size of the button.
13. **doClick()** - Simulates a button click.

These methods allow you to work with **JButton** controls in a flexible and efficient manner. You can add an **ActionListener** to handle button events, set the text on the button, set the tooltip text and icon, enable or disable the button, and get or set the selection state of the button. Additionally, the **JButton** class provides methods for working with the button's **ButtonModel**, such as **getModel()**, which represents the state of the button, and **setMnemonic(int mnemonic)**, which sets the mnemonic for the button. Other methods, such as **setMargin(Insets insets)**, **setFont(Font font)**, and **getPreferredSize()**, allow you to customize the appearance of the button. Finally, the **doClick()** method can be used to simulate a button click programmatically.

41) inbuild methods in jlabel class in java?

The **JLabel** class in Java is used to create a component that displays a text or an image or both. Some of the common methods in the **JLabel** class are:

1. **setText(String text)** - Sets the text on the label.
2. **setToolTipText(String text)** - Sets the tooltip text for the label.
3. **setIcon(Icon icon)** - Sets the icon for the label.
4. **setHorizontalAlignment(int alignment)** - Sets the horizontal alignment of the label's content.
5. **setVerticalAlignment(int alignment)** - Sets the vertical alignment of the label's content.
6. **setAlignmentX(float alignmentX)** - Sets the horizontal alignment of the label within its container.
7. **setAlignmentY(float alignmentY)** - Sets the vertical alignment of the label within its container.
8. **setFont(Font font)** - Sets the font for the label.
9. **getPreferredSize()** - Returns the preferred size of the label.
10. **setOpaque(boolean isOpaque)** - Sets the opacity of the label.
11. **setEnabled(boolean enabled)** - Enables or disables the label.
12. **setBorder(Border border)** - Sets the border for the label.
13. **setBackground(Color color)** - Sets the background color of the label.
14. **setForeground(Color color)** - Sets the foreground color of the label.

These methods allow you to work with **JLabel** controls in a flexible and efficient manner. You can set the text on the label, set the tooltip text and icon, set the horizontal and vertical alignment of the label's content, set the alignment of the label within its container, and customize the appearance of the label with methods like **setFont(Font font)**, **setOpaque(boolean isOpaque)**, and **setBorder(Border border)**. Additionally, you can enable or disable the label, set its background and foreground colors, and get its preferred size using the **getPreferredSize()** method. All of these methods allow you to create and customize **JLabel** controls according to your needs.

42) inbuild methods in jtextfield class in java?

The **JTextField** class in Java is used to create a component that allows the user to input text. Some of the common methods in the **JTextField** class are:

1. **setText(String text)** - Sets the text in the text field.
2. **getText()** - Returns the text in the text field.
3. **setEditable(boolean editable)** - Sets whether the text field is editable or not.
4. **setColumns(int columns)** - Sets the number of columns in the text field.
5. **setHorizontalAlignment(int alignment)** - Sets the horizontal alignment of the text in the text field.
6. **setCaretPosition(int position)** - Sets the position of the caret (the insertion point) in the text field.
7. **selectAll()** - Selects all the text in the text field.
8. **setSelectionStart(int start)** - Sets the starting point of the selected text in the text field.

9. **setSelectionEnd(int end)** - Sets the ending point of the selected text in the text field.
10. **getSelectedText()** - Returns the selected text in the text field.
11. **setToolTipText(String text)** - Sets the tooltip text for the text field.
12. **getToolTipText()** - Returns the tooltip text for the text field.
13. **setEnabled(boolean enabled)** - Enables or disables the text field.
14. **setBorder(Border border)** - Sets the border for the text field.
15. **setBackground(Color color)** - Sets the background color of the text field.
16. **setForeground(Color color)** - Sets the foreground color of the text field.

These methods allow you to work with **JTextField** controls in a flexible and efficient manner. You can set the text in the text field, set whether the text field is editable or not, set the number of columns in the text field, set the horizontal alignment of the text in the text field, and customize the selection and caret position using methods like **setSelectionStart(int start)** and **setCaretPosition(int position)**. Additionally, you can select all or part of the text, set and retrieve the tooltip text, enable or disable the text field, set its border, and customize its background and foreground colors using the relevant methods. All of these methods allow you to create and customize **JTextField** controls according to your needs.

43) inbuild methods in jcheckbox class in java?

The **JCheckBox** class in Java is used to create a component that represents a check box that can be selected or deselected by the user. Some of the common methods in the **JCheckBox** class are:

1. **isSelected()** - Returns a boolean value that indicates whether the check box is selected or not.
2. **setSelected(boolean selected)** - Sets whether the check box is selected or not.
3. **setText(String text)** - Sets the text displayed next to the check box.
4. **getText()** - Returns the text displayed next to the check box.
5. **setEnabled(boolean enabled)** - Enables or disables the check box.
6. **setToolTipText(String text)** - Sets the tooltip text for the check box.
7. **getToolTipText()** - Returns the tooltip text for the check box.
8. **setHorizontalAlignment(int alignment)** - Sets the horizontal alignment of the check box and its text.
9. **setBorder(Border border)** - Sets the border for the check box.
10. **setBackground(Color color)** - Sets the background color of the check box.
11. **setForeground(Color color)** - Sets the foreground color of the check box.
12. **addItemListener(ItemListener listener)** - Adds an **ItemListener** to the check box.
13. **removeItemListener(ItemListener listener)** - Removes an **ItemListener** from the check box.

These methods allow you to work with **JCheckBox** controls in a flexible and efficient manner. You can check whether the check box is selected or not, set the selected state of the check box, set and retrieve the text displayed next to the check box, enable or disable the check box, set and retrieve the tooltip text, and customize the horizontal alignment, border, background, and foreground colors of the check box using the relevant methods. Additionally, you can add and remove **ItemListener**s to the check box to listen for selection

events. All of these methods allow you to create and customize **JCheckBox** controls according to your needs.

44) inbuild methods in jradiobutton class in java?

The **JRadioButton** class in Java is used to create a component that represents a radio button that can be selected or deselected by the user. Some of the common methods in the **JRadioButton** class are:

1. **isSelected()** - Returns a boolean value that indicates whether the radio button is selected or not.
2. **setSelected(boolean selected)** - Sets whether the radio button is selected or not.
3. **setText(String text)** - Sets the text displayed next to the radio button.
4. **getText()** - Returns the text displayed next to the radio button.
5. **setEnabled(boolean enabled)** - Enables or disables the radio button.
6. **setToolTipText(String text)** - Sets the tooltip text for the radio button.
7. **getToolTipText()** - Returns the tooltip text for the radio button.
8. **setHorizontalAlignment(int alignment)** - Sets the horizontal alignment of the radio button and its text.
9. **setBorder(Border border)** - Sets the border for the radio button.
10. **setBackground(Color color)** - Sets the background color of the radio button.
11. **setForeground(Color color)** - Sets the foreground color of the radio button.
12. **addItemListener(ItemListener listener)** - Adds an **ItemListener** to the radio button.
13. **removeItemListener(ItemListener listener)** - Removes an **ItemListener** from the radio button.

These methods allow you to work with **JRadioButton** controls in a flexible and efficient manner. You can check whether the radio button is selected or not, set the selected state of the radio button, set and retrieve the text displayed next to the radio button, enable or disable the radio button, set and retrieve the tooltip text, and customize the horizontal alignment, border, background, and foreground colors of the radio button using the relevant methods. Additionally, you can add and remove **ItemListener**s to the radio button to listen for selection events. All of these methods allow you to create and customize **JRadioButton** controls according to your needs.

45) inbuild methods in jlist class in java?

The **JList** class in Java is used to create a component that displays a list of items. Some of the common methods in the **JList** class are:

1. **setListData(Object[] listData)** - Sets the data that will be displayed in the list.
2. **setSelectedIndex(int index)** - Selects the item at the specified index in the list.
3. **getSelectedIndex()** - Returns the index of the currently selected item in the list.
4. **getSelectedValue()** - Returns the value of the currently selected item in the list.
5. **getSelectedValues()** - Returns an array of the values of all the currently selected items in the list.

6. **addListSelectionListener(ListSelectionListener listener)** - Adds a **ListSelectionListener** to the list to listen for selection events.
7. **removeListSelectionListener(ListSelectionListener listener)** - Removes a **ListSelectionListener** from the list.
8. **setEnabled(boolean enabled)** - Enables or disables the list.
9. **setToolTipText(String text)** - Sets the tooltip text for the list.
10. **getToolTipText()** - Returns the tooltip text for the list.
11. **setSelectionMode(int mode)** - Sets the selection mode for the list.
12. **setLayoutOrientation(int orientation)** - Sets the layout orientation of the list.
13. **setCellRenderer(ListCellRenderer<? super E> renderer)** - Sets the renderer that will be used to display the items in the list.
14. **setPrototypeCellValue(E prototypeCellValue)** - Sets the prototype value for the cells in the list.
15. **setFixedCellWidth(int width)** - Sets the fixed width of the cells in the list.
16. **setFixedCellHeight(int height)** - Sets the fixed height of the cells in the list.

These methods allow you to work with **JList** controls in a flexible and efficient manner. You can set the data that will be displayed in the list, select and retrieve the selected item(s) from the list, add and remove **ListSelectionListener**s to the list to listen for selection events, enable or disable the list, set and retrieve the tooltip text, set the selection mode and layout orientation of the list, customize the renderer that will be used to display the items in the list, and set the prototype value, fixed width, and fixed height of the cells in the list using the relevant methods. All of these methods allow you to create and customize **JList** controls according to your needs.

46) inbuild methods in jcombobox class in java?

The **JComboBox** class in Java is used to create a component that combines a button or editable field with a drop-down list of items. Some of the common methods in the **JComboBox** class are:

1. **setModel(ComboBoxModel<E> aModel)** - Sets the data model for the combo box.
2. **getModel()** - Returns the data model for the combo box.
3. **setSelectedItem(Object anObject)** - Selects the item in the combo box that matches the specified object.
4. **getSelectedItem()** - Returns the selected item in the combo box.
5. **getSelectedIndex()** - Returns the index of the selected item in the combo box.
6. **setSelectedIndex(int anIndex)** - Selects the item in the combo box at the specified index.
7. **addItem(Object anObject)** - Adds an item to the combo box.
8. **insertItemAt(Object anObject, int anIndex)** - Inserts an item at the specified index in the combo box.
9. **removeItemAt(int anIndex)** - Removes the item at the specified index from the combo box.
10. **removeItem(Object anObject)** - Removes the specified item from the combo box.
11. **removeAllItems()** - Removes all items from the combo box.
12. **setEditable(boolean aFlag)** - Sets whether the combo box is editable or not.
13. **isEditable()** - Returns whether the combo box is editable or not.

14. **getEditor()** - Returns the editor component for the combo box.
15. **setRenderer(ListCellRenderer<? super E> aRenderer)** - Sets the renderer that will be used to display the items in the combo box.
16. **getRenderer()** - Returns the renderer that is currently being used to display the items in the combo box.
17. **addActionListener(ActionListener l)** - Adds an **ActionListener** to the combo box to listen for selection events.
18. **removeActionListener(ActionListener l)** - Removes an **ActionListener** from the combo box.
19. **setEnabled(boolean enabled)** - Enables or disables the combo box.
20. **setToolTipText(String text)** - Sets the tooltip text for the combo box.
21. **getToolTipText()** - Returns the tooltip text for the combo box.

These methods allow you to work with **JComboBox** controls in a flexible and efficient manner. You can set the data model for the combo box, select and retrieve the selected item(s) from the combo box, add and remove **ActionListener**s to the combo box to listen for selection events, enable or disable the combo box, set and retrieve the tooltip text, set whether the combo box is editable or not, get the editor component for the combo box, customize the renderer that will be used to display the items in the combo box, and add, insert, remove, and clear items in the combo box using the relevant methods. All of these methods allow you to create and customize **JComboBox** controls according to your needs.

47) inbuild methods in connection interface in java?

The **Connection** interface in Java represents a connection to a database. It provides methods to create statements, commit or rollback transactions, and manage the connection properties. Some of the inbuilt methods in the **Connection** interface are:

1. **createStatement()**: Creates a **Statement** object for sending SQL statements to the database.
2. **prepareStatement(String sql)**: Creates a **PreparedStatement** object for sending parameterized SQL statements to the database.
3. **commit()**: Commits the current transaction and makes all changes made during the transaction permanent.
4. **rollback()**: Rolls back the current transaction and discards all changes made during the transaction.
5. **setAutoCommit(boolean autoCommit)**: Sets the auto-commit mode for the connection. When auto-commit mode is on, each SQL statement is treated as a transaction and is automatically committed.
6. **close()**: Closes the connection to the database. All statements, result sets, and other resources associated with the connection are also closed.
7. **getMetaData()**: Returns a **DatabaseMetaData** object that contains metadata about the database to which this connection is connected.
8. **setTransactionIsolation(int level)**: Sets the transaction isolation level for this connection.
9. **getTransactionIsolation()**: Returns the current transaction isolation level for this connection.

10. **setReadOnly(boolean readOnly)**: Sets the read-only mode for the connection. When read-only mode is on, any attempt to modify the database will result in an exception.

Note that these are just some of the methods provided by the **Connection** interface, and there are many more that can be used to manage database connections and transactions.

48) inbuild methods in statement interface in java?

The **Statement** interface in Java represents a SQL statement that can be executed against a database. It provides methods to execute statements, retrieve results, and manage the statement properties. Some of the inbuilt methods in the **Statement** interface are:

1. **executeQuery(String sql)**: Executes a SQL query and returns a **ResultSet** object containing the results.
2. **executeUpdate(String sql)**: Executes a SQL statement that modifies the database and returns the number of rows affected.
3. **addBatch(String sql)**: Adds a SQL statement to the current batch of statements to be executed.
4. **clearBatch()**: Clears the current batch of statements to be executed.
5. **executeBatch()**: Executes the current batch of statements and returns an array of update counts.
6. **getResultSet()**: Returns the **ResultSet** object generated by the most recent execution of this statement.
7. **getUpdateCount()**: Returns the number of rows affected by the most recent execution of this statement.
8. **setMaxRows(int maxRows)**: Sets the maximum number of rows that can be retrieved by the **ResultSet** generated by this statement.
9. **setFetchSize(int fetchSize)**: Sets the number of rows to be fetched from the database at a time.
10. **close()**: Closes the **Statement** object and releases any resources associated with it.

Note that these are just some of the methods provided by the **Statement** interface, and there are many more that can be used to execute SQL statements and retrieve results.

49) inbuild methods in preparedstatement interface in java?

The **PreparedStatement** interface in Java extends the **Statement** interface and represents a precompiled SQL statement that can be executed against a database. It is used to execute parameterized SQL queries, which are SQL statements that contain placeholders for input parameters. Some of the inbuilt methods in the **PreparedStatement** interface are:

1. **setInt(int parameterIndex, int value)**: Sets the value of the specified parameter to the given integer value.
2. **setString(int parameterIndex, String value)**: Sets the value of the specified parameter to the given string value.
3. **setDouble(int parameterIndex, double value)**: Sets the value of the specified parameter to the given double value.

4. **setDate(int parameterIndex, Date value)**: Sets the value of the specified parameter to the given **java.sql.Date** value.
5. **setTime(int parameterIndex, Time value)**: Sets the value of the specified parameter to the given **java.sql.Time** value.
6. **setTimestamp(int parameterIndex, Timestamp value)**: Sets the value of the specified parameter to the given **java.sql.Timestamp** value.
7. **setNull(int parameterIndex, int sqlType)**: Sets the value of the specified parameter to **NULL**.
8. **executeQuery()**: Executes the SQL query and returns a **ResultSet** object containing the results.
9. **executeUpdate()**: Executes the SQL statement that modifies the database and returns the number of rows affected.
10. **addBatch()**: Adds the set of parameters to the current batch of statements to be executed.
11. **clearParameters()**: Clears the values of all the parameters.
12. **getResultSet()**: Returns the **ResultSet** object generated by the most recent execution of this statement.
13. **getUpdateCount()**: Returns the number of rows affected by the most recent execution of this statement.
14. **setMaxRows(int maxRows)**: Sets the maximum number of rows that can be retrieved by the **ResultSet** generated by this statement.
15. **setFetchSize(int fetchSize)**: Sets the number of rows to be fetched from the database at a time.
16. **close()**: Closes the **PreparedStatement** object and releases any resources associated with it.

Note that these are just some of the methods provided by the **PreparedStatement** interface, and there are many more that can be used to execute parameterized SQL queries and retrieve results.

50) inbuild methods in resultset interface in java?

The **ResultSet** interface in Java represents the result set of a database query. It provides methods for iterating over the rows in the result set and retrieving the values of the columns for each row. Some of the inbuilt methods in the **ResultSet** interface are:

1. **next()**: Moves the cursor to the next row in the result set and returns **true** if there is another row, or **false** if the end of the result set has been reached.
2. **getInt(int columnIndex)**: Retrieves the value of the specified column as an integer.
3. **getString(int columnIndex)**: Retrieves the value of the specified column as a string.
4. **getDouble(int columnIndex)**: Retrieves the value of the specified column as a double.
5. **getDate(int columnIndex)**: Retrieves the value of the specified column as a **java.sql.Date** object.
6. **getTime(int columnIndex)**: Retrieves the value of the specified column as a **java.sql.Time** object.
7. **getTimestamp(int columnIndex)**: Retrieves the value of the specified column as a **java.sql.Timestamp** object.

8. **getObject(int columnIndex)**: Retrieves the value of the specified column as an **Object**.
9. **wasNull()**: Returns **true** if the value of the last retrieved column was **NULL**, or **false** otherwise.
10. **getMetaData()**: Returns a **ResultSetMetaData** object that contains metadata about the columns in the result set.
11. **absolute(int row)**: Moves the cursor to the specified row number in the result set.
12. **beforeFirst()**: Moves the cursor before the first row in the result set.
13. **afterLast()**: Moves the cursor after the last row in the result set.
14. **first()**: Moves the cursor to the first row in the result set.
15. **last()**: Moves the cursor to the last row in the result set.
16. **isBeforeFirst()**: Returns **true** if the cursor is before the first row in the result set, or **false** otherwise.
17. **isAfterLast()**: Returns **true** if the cursor is after the last row in the result set, or **false** otherwise.
18. **getRow()**: Returns the current row number in the result set.
19. **close()**: Closes the **ResultSet** object and releases any resources associated with it.

Note that these are just some of the methods provided by the **ResultSet** interface, and there are many more that can be used to retrieve values from the result set and navigate through the rows.

51) inbuild methods in resultsetmetadata interface in java?

The **ResultSetMetaData** interface in Java provides methods to get information about the types and properties of the columns in a **ResultSet**. Some of the commonly used methods of this interface are:

- **getColumnCount()**: Returns the number of columns in the **ResultSet**.
- **getColumnName(int column)**: Returns the name of the specified column in the **ResultSet**.
- **getColumnLabel(int column)**: Returns the label for the specified column in the **ResultSet**.
- **getColumnType(int column)**: Returns the SQL type of the specified column in the **ResultSet**.
- **getColumnTypeName(int column)**: Returns the database-specific type name of the specified column in the **ResultSet**.
- **getColumnDisplaySize(int column)**: Returns the maximum width of the specified column in characters.
- **isNullable(int column)**: Returns whether the specified column in the **ResultSet** allows **NULL** values.
- **getPrecision(int column)**: Returns the precision of the specified column in the **ResultSet**.
- **getScale(int column)**: Returns the scale of the specified column in the **ResultSet**.

These methods can be used to retrieve information about the columns in a **ResultSet** and to perform operations on the data in those columns.

52) inbuild methods in databasemetadata interface in java?

The **DatabaseMetaData** interface in Java provides methods to get information about the database, such as the supported SQL syntax, tables, columns, indexes, and other schema objects. Some of the commonly used methods of this interface are:

- **getDatabaseProductName()**: Returns the name of the database product.
- **getDatabaseProductVersion()**: Returns the version of the database product.
- **getDriverName()**: Returns the name of the JDBC driver.
- **getDriverVersion()**: Returns the version of the JDBC driver.
- **getTables(String catalog, String schemaPattern, String tableNamePattern, String[] types)**: Returns information about the tables in the database.
- **getColumns(String catalog, String schemaPattern, String tableNamePattern, String columnNamePattern)**: Returns information about the columns in the specified table.
- **getIndexInfo(String catalog, String schema, String table, boolean unique, boolean approximate)**: Returns information about the indexes in the specified table.
- **getPrimaryKeys(String catalog, String schema, String table)**: Returns information about the primary key columns in the specified table.
- **getImportedKeys(String catalog, String schema, String table)**: Returns information about the foreign key columns that reference the specified table.

These methods can be used to retrieve metadata about the database and its objects, which can be useful for writing generic code that can work with different databases, as well as for generating database schema documentation or performing other schema-related tasks.

53) inbuild methods in drivermanager interface in java?

The **DriverManager** class in Java provides methods to manage JDBC drivers, including loading drivers, registering drivers, and obtaining database connections. Some of the commonly used methods of this class are:

- **registerDriver(Driver driver)**: Registers the given driver with the DriverManager.
- **deregisterDriver(Driver driver)**: Deregisters the given driver from the DriverManager.
- **getConnection(String url)**: Attempts to establish a connection to the given database URL.
- **getConnection(String url, Properties info)**: Attempts to establish a connection to the given database URL with the specified properties.
- **getDriver(String url)**: Attempts to locate a driver that understands the given database URL.
- **getDrivers()**: Returns an Enumeration of all the currently loaded JDBC drivers.

These methods can be used to manage JDBC drivers and establish database connections, which is a fundamental task in JDBC programming. For example, the **getConnection()** method can be used to obtain a connection to a database, which can then be used to execute SQL queries and update statements.

54) inbuild methods in messagedigest class in java?

The **MessageDigest** class in Java provides cryptographic hash functions to compute a fixed-length message digest (hash value) for a given input data. Some of the commonly used methods of this class are:

- **getInstance(String algorithm)**: Returns a MessageDigest object that implements the specified digest algorithm.
- **reset()**: Resets the digest for further use.
- **update(byte[] input)**: Updates the digest using the specified byte array.
- **digest()**: Completes the hash computation by performing final operations such as padding, and returns the resulting digest as a byte array.
- **digest(byte[] input)**: Computes the digest of the specified byte array, and returns the resulting digest as a byte array.
- **digest(byte[] input, int offset, int len)**: Computes the digest of a subarray of the specified byte array, and returns the resulting digest as a byte array.

These methods can be used to compute message digests of input data, which is useful for verifying data integrity and ensuring message authenticity. For example, the **digest()** method can be used to compute a hash value for a password or a file, which can then be stored or transmitted securely. The **getInstance()** method can be used to obtain a **MessageDigest** object that implements a specific hash algorithm, such as SHA-256 or MD5.

55) inbuild methods in keypairganerator class in java?

The **KeyPairGenerator** class in Java is used for generating pairs of public and private keys for various encryption algorithms. Some of the commonly used methods of this class are:

- **getInstance(String algorithm)**: Returns a KeyPairGenerator object that generates public/private key pairs for the specified algorithm.
- **initialize(int keysize)**: Initializes the key pair generator with the specified key size (in bits).
- **initialize(AlgorithmParameterSpec params)**: Initializes the key pair generator with the specified algorithm parameters.
- **generateKeyPair()**: Generates a new key pair consisting of a public and a private key.
- **getAlgorithm()**: Returns the name of the algorithm used by this key pair generator.
- **getProvider()**: Returns the provider of this key pair generator.

These methods can be used to generate public/private key pairs for various encryption algorithms such as RSA, DSA, and EC. The **getInstance()** method can be used to obtain a **KeyPairGenerator** object that generates key pairs for a specific algorithm, such as "RSA" or "DSA". The **initialize()** method can be used to set the key size or algorithm parameters, depending on the algorithm being used. The **generateKeyPair()** method generates a new key pair, which can then be used for encryption, decryption, or digital signatures. The **getAlgorithm()** method returns the name of the algorithm being used by the **KeyPairGenerator**, while the **getProvider()** method returns the provider of the algorithm.

56) inbuild methods in keyfactory class in java?

The **KeyFactory** class in Java is used for converting cryptographic keys between their abstract representation and their corresponding key specification. Some of the commonly used methods of this class are:

- **getInstance(String algorithm)**: Returns a **KeyFactory** object that converts keys for the specified algorithm.
- **generatePrivate(KeySpec keySpec)**: Generates a private key object that corresponds to the given key specification.
- **generatePublic(KeySpec keySpec)**: Generates a public key object that corresponds to the given key specification.
- **getKeySpec(Key key, Class<T> keySpec)**: Returns a specification of the given key in the requested format.
- **translateKey(Key key)**: Translates a key into a form that can be used by the **KeyFactory**.

These methods can be used to convert keys between different formats, such as from a **PublicKey** object to a **X509EncodedKeySpec** object, or from a **PrivateKey** object to a **PKCS8EncodedKeySpec** object. The **getInstance()** method is used to obtain a **KeyFactory** object that supports a specific algorithm, such as "RSA" or "DSA". The **generatePrivate()** and **generatePublic()** methods are used to generate private and public key objects, respectively, based on a key specification. The **getKeySpec()** method is used to obtain a specification of the given key in the requested format, such as a **PKCS8EncodedKeySpec** object that contains the key data in PKCS#8 format. The **translateKey()** method is used to translate a key into a form that can be used by the **KeyFactory**, such as a **PrivateKey** object that can be used for decryption.

57) inbuild methods in keystore class in java?

The **KeyStore** class in Java is used for managing a collection of cryptographic keys and certificates. It can be used to store, load, and manage keys and certificates in a secure manner. Some of the commonly used methods of this class are:

- **getInstance(String type)**: Returns a **KeyStore** object that implements the specified type of keystore.
- **load(InputStream stream, char[] password)**: Loads the keystore data from the given input stream and decrypts it with the specified password.
- **store(OutputStream stream, char[] password)**: Writes the keystore data to the given output stream and encrypts it with the specified password.
- **getKey(String alias, char[] password)**: Returns the key associated with the given alias, using the specified password to access it.
- **getCertificate(String alias)**: Returns the certificate associated with the given alias.
- **setKeyEntry(String alias, Key key, char[] password, Certificate[] chain)**: Associates the given key and certificate chain with the specified alias, protected by the specified password.

These methods can be used to create, load, and manipulate a keystore. The **getInstance()** method is used to obtain a **KeyStore** object that implements a specific type of keystore, such as "JKS" or "PKCS12". The **load()** method is used to load the keystore data from an input stream, such as a file or network socket, and decrypt it with the specified password. The **store()** method is used to write the keystore data to an output stream and encrypt it with the specified password. The **getKey()** method is used to retrieve the key associated with a specific alias, using the specified password to access it. The **getCertificate()** method is used to retrieve the certificate associated with a specific alias. The **setKeyEntry()** method is used to associate a key and certificate chain with a specific alias, protected by the specified password.

58) inbuild methods in securerandom class in java?

The **SecureRandom** class in Java provides methods to generate cryptographically strong random numbers. Some of the commonly used methods in the **SecureRandom** class are:

- **void nextBytes(byte[] bytes)**: Generates a user-specified number of random bytes and stores them in the given byte array.
- **int nextInt()**: Generates a 32-bit signed random integer.
- **long nextLong()**: Generates a 64-bit signed random long.
- **double nextDouble()**: Generates a random double between 0.0 and 1.0.
- **float nextFloat()**: Generates a random float between 0.0 and 1.0.
- **boolean nextBoolean()**: Generates a random boolean value.

These methods can be used to generate random numbers for applications that require secure random data, such as cryptographic keys or random passwords.

59) inbuild methods in signature class in java?

The **Signature** class in Java provides methods for generating and verifying digital signatures using different cryptographic algorithms. Some of the commonly used methods in the **Signature** class are:

- **void initSign(PrivateKey privateKey)**: Initializes the **Signature** object for signing with the specified private key.
- **void initVerify(PublicKey publicKey)**: Initializes the **Signature** object for verification with the specified public key.
- **void update(byte[] data)**: Updates the **Signature** object with the specified data for signing or verification.
- **byte[] sign()**: Generates the digital signature for the previously updated data.
- **boolean verify(byte[] signature)**: Verifies the digital signature with the previously updated data.

These methods can be used to generate and verify digital signatures for various purposes, such as authenticating messages or ensuring the integrity of data. The **Signature** class supports a wide range of cryptographic algorithms, including RSA, DSA, and ECDSA.

60) inbuild methods in cipher class in java?

The **Cipher** class in Java provides methods for encrypting and decrypting data using different cryptographic algorithms. Some of the commonly used methods in the **Cipher** class are:

- **void init(int opmode, Key key)**: Initializes the **Cipher** object for encryption or decryption with the specified key.
- **void init(int opmode, Key key, AlgorithmParameters params)**: Initializes the **Cipher** object for encryption or decryption with the specified key and algorithm parameters.
- **void init(int opmode, Key key, SecureRandom random)**: Initializes the **Cipher** object for encryption or decryption with the specified key and random number generator.
- **byte[] doFinal(byte[] input)**: Encrypts or decrypts the specified input data and returns the result.
- **byte[] getIV()**: Returns the initialization vector used for the most recent encryption or decryption operation.

These methods can be used to encrypt and decrypt data for various purposes, such as securing communication channels or protecting sensitive information. The **Cipher** class supports a wide range of cryptographic algorithms, including AES, DES, and RSA.

61) inbuild methods in provider class in java?

The **Provider** class in Java represents a security provider, which is a package of cryptographic services and algorithms that can be used by applications. Some of the commonly used methods in the **Provider** class are:

- **String getName()**: Returns the name of the provider.
- **double getVersion()**: Returns the version number of the provider.
- **Set<Provider.Service> getServices()**: Returns a **Set** of **Provider.Service** objects that represent the cryptographic services provided by the provider.
- **Provider.Service getService(String type, String algorithm)**: Returns the **Provider.Service** object that provides the specified cryptographic service and algorithm.
- **Object get(String key)**: Returns the value of the specified attribute of the provider, if it exists.

These methods can be used to retrieve information about the cryptographic services provided by a security provider and to access those services. The **Provider** class also provides methods for registering a new security provider and for removing an existing one. Applications can use multiple security providers to access a wider range of cryptographic services and to enhance the security of their operations.