In Java, **Wrapper Classes**, **Auto-boxing**, **Unboxing**, **Utility Methods**, and **Immutability** are important concepts that are widely used when working with primitive data types, collections, and ensuring the integrity of objects.

**1. Wrapper Classes**

Wrapper classes are used to convert primitive types into objects. In Java, each primitive data type has a corresponding wrapper class in the java.lang package. For example:

* int → Integer
* char → Character
* boolean → Boolean
* double → Double

// Primitive type

int num = 10;

// Wrapper class for Integer

Integer numWrapper = Integer.valueOf(num);

System.out.println("Wrapped Integer: " + numWrapper);

In the above example, the primitive int is wrapped into an Integer object using Integer.valueOf().

**2. Auto-boxing and Unboxing**

**Auto-boxing:**

Auto-boxing is the automatic conversion of a primitive type to its corresponding wrapper class when an object is required (such as in collections).

**Example of Auto-boxing:**

java

Copy code

public class AutoBoxingExample {

public static void main(String[] args) {

int primitive = 5;

// Auto-boxing: primitive int is automatically converted to Integer

Integer wrapper = primitive;

System.out.println("Wrapper Object: " + wrapper);

}

}

In the example above, the primitive int is automatically converted to an Integer object when assigned to the wrapper.

**Unboxing:**

Unboxing is the reverse process where a wrapper class object is automatically converted back into its corresponding primitive type.

**Example of Unboxing:**

java

Copy code

public class UnBoxingExample {

public static void main(String[] args) {

Integer wrapper = 10; // Auto-boxed integer

// Unboxing: the Integer object is automatically converted to primitive int

int primitive = wrapper;

System.out.println("Unboxed primitive: " + primitive);

}

}

In the example, the Integer object wrapper is automatically unboxed to a primitive int when assigned to primitive.

**3. Utility Methods in Wrapper Classes**

Wrapper classes in Java provide several **utility methods** to manipulate or convert values. Some of the most commonly used methods are:

* parseInt(), parseDouble(), parseBoolean() — used to convert a String to a primitive type.
* valueOf() — used to convert a primitive to a wrapper object.
* toString() — used to convert a wrapper object to a String.

**1. parseInt(), parseDouble(), parseBoolean()**

These methods are part of the Integer, Double, and Boolean wrapper classes respectively, and they convert a String into a corresponding primitive type.

**Example:**

java

Copy code

public class ParseMethodsExample {

public static void main(String[] args) {

// Using parseInt()

String intString = "123";

int number = Integer.parseInt(intString);

System.out.println("Converted Integer: " + number); // Output: 123

// Using parseDouble()

String doubleString = "45.67";

double decimal = Double.parseDouble(doubleString);

System.out.println("Converted Double: " + decimal); // Output: 45.67

// Using parseBoolean()

String booleanString = "true";

boolean flag = Boolean.parseBoolean(booleanString);

System.out.println("Converted Boolean: " + flag); // Output: true

}

}

**Explanation:**

* parseInt() converts the numeric String into an int. If the String contains non-numeric characters (e.g., "123a"), it will throw a NumberFormatException.
* parseDouble() converts the numeric String with a decimal into a double. Similar to parseInt(), it will throw a NumberFormatException for invalid input.
* parseBoolean() converts "true" (case insensitive) to true and any other input to false.

**2. valueOf()**

This method converts a primitive type into its corresponding wrapper object. It is a static method of wrapper classes like Integer, Double, and Boolean.

**Example:**

java

Copy code

public class ValueOfExample {

public static void main(String[] args) {

// Using valueOf() for Integer

int num = 456;

Integer integerObject = Integer.valueOf(num);

System.out.println("Integer Object: " + integerObject); // Output: 456

// Using valueOf() for Double

double dec = 78.91;

Double doubleObject = Double.valueOf(dec);

System.out.println("Double Object: " + doubleObject); // Output: 78.91

// Using valueOf() for Boolean

boolean bool = true;

Boolean booleanObject = Boolean.valueOf(bool);

System.out.println("Boolean Object: " + booleanObject); // Output: true

}

}

**Explanation:**

* valueOf() wraps a primitive type in its corresponding wrapper class.
* The wrapper object can then be used in places where an object is required (e.g., collections).

**3. toString()**

This method is used to convert a wrapper object to a String. It can also be applied to primitives by first boxing them into their wrapper objects.

**Example:**

java

Copy code

public class ToStringExample {

public static void main(String[] args) {

// Convert Integer to String

Integer integerObject = 789;

String intString = integerObject.toString();

System.out.println("String representation of Integer: " + intString); // Output: "789"

// Convert Double to String

Double doubleObject = 123.45;

String doubleString = doubleObject.toString();

System.out.println("String representation of Double: " + doubleString); // Output: "123.45"

// Convert Boolean to String

Boolean booleanObject = false;

String booleanString = booleanObject.toString();

System.out.println("String representation of Boolean: " + booleanString); // Output: "false"

}

}

**Explanation:**

* toString() converts the value of the wrapper object into its String form.
* This method is particularly useful when you need the String representation of a wrapper object to concatenate with other strings or display in the user interface.

**Summary:**

* **parseInt(), parseDouble(), parseBoolean()**: Convert String to primitive types.
* **valueOf()**: Converts a primitive type to a wrapper object.
* **toString()**: Converts a wrapper object to its String representation.

**Example of Utility Methods:**

java

Copy code

public class UtilityMethodsExample {

public static void main(String[] args) {

// Parsing String to primitive type

String str = "123";

int parsedValue = Integer.parseInt(str);

System.out.println("Parsed int value: " + parsedValue);

// Converting primitive to wrapper object

Integer wrapper = Integer.valueOf(parsedValue);

System.out.println("Wrapper Integer value: " + wrapper);

// Converting wrapper object to String

String stringFromWrapper = wrapper.toString();

System.out.println("String from wrapper: " + stringFromWrapper);

}

}

**4. Immutability in Java**

An object is considered **immutable** if its state cannot be changed after it is created. Immutability is useful in multi-threaded environments because immutable objects are thread-safe.

**How to Make a Class Immutable:**

1. Make the class final so it cannot be subclassed.
2. Make all fields private and final.
3. Provide a constructor to initialize the fields.
4. Don't provide setter methods.
5. If the class contains mutable objects, ensure that they cannot be modified by the outside world (e.g., by returning copies of them).

**Example of an Immutable Class:**

java

Copy code

public final class Person {

private final String name;

private final int age;

// Constructor to initialize values

public Person(String name, int age) {

this.name = name;

this.age = age;

}

// Getter methods to access the fields

public String getName() {

return name;

}

public int getAge() {

return age;

}

// No setters to modify the fields

@Override

public String toString() {

return "Person{name='" + name + "', age=" + age + "}";

}

}

public class ImmutableExample {

public static void main(String[] args) {

// Creating an immutable object

Person person = new Person("John", 25);

System.out.println(person);

// Attempting to modify the state of the object (not possible)

// person.name = "Alice"; // Compilation error

}

}

In the Person class:

* The class is final to prevent subclassing.
* The fields are private and final to prevent modification.
* There are no setter methods, ensuring the object's state cannot be changed.

**Benefits of Immutability:**

* **Thread-safety**: Immutable objects can be shared between multiple threads without synchronization.
* **Security**: The state of immutable objects cannot be changed by any external code.
* **Simplicity**: Immutability reduces the complexity of programs as you don’t need to track changes in state.

**Summary:**

* **Wrapper Classes** convert primitive types to objects and provide utility methods for manipulation.
* **Auto-boxing** and **Unboxing** provide automatic conversion between primitive types and wrapper objects.
* **Utility Methods** in wrapper classes allow parsing and conversion operations, such as parseInt(), toString(), etc.
* **Immutability** is a design concept where the state of an object cannot be changed after it is created, providing safety in multi-threaded environments and improving program stability.