# Additional Topics

# Java Type Casting

## Type Casting

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

1. Widening Type Casting

2. Narrowing Type Casting

## Widening Type Casting

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

### Example: Converting int to double

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

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

Output

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

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

Note: This is also known as Implicit Type Casting.

## Narrowing Type Casting

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

### Example: Converting double into an int

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

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

Output

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

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

Note: This is also known as Explicit Type Casting.

# Java Lambda Expressions

**What is Functional Interface?**

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

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

**Introduction to lambda expressions**

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

## How to define lambda expression in Java?

Here is how we can define lambda expression in Java.

(parameter list) -> lambda body
The new operator (->) used is known as an arrow operator or a lambda operator.

Suppose, we have a method like this:

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

We can write this method using lambda expression as:

```
() -> 3.1415
```

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

**Types of Lambda Body**

In Java, the lambda body is of two types.

**1. A body with a single expression**

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

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

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

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

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

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

```
import java.lang.FunctionalInterface;
// this is functional interface
@FunctionalInterface
interface MyInterface{
    // abstract method
    double getPiValue();
}
public class Main {
    public static void main( String[] args ) {
    // declare a reference to MyInterface
    MyInterface ref;
   // lambda expression
    ref = () -> 3.1415;
    System.out.println("Value of Pi = " + ref.getPiValue());
    }
}
```

**Example : Using lambda expression with parameters**

```
@FunctionalInterface
interface MyInterface {
   // abstract method
   String reverse(String n);
}
public class Main {
   public static void main( String[] args ) {
      // declare a reference to MyInterface
      // assign a lambda expression to the reference
      MyInterface ref = (str) -> {
         String result = "";
         for (int i = str.length()-1; i >= 0 ; i--)
         result += str.charAt(i);
         return result;
      };
      // call the method of the interface
      System.out.println("Lambda reversed = " + ref.reverse("Lambda"));
   }
}
```

# Java Generics

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

This helps us to reuse our code.

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

# Example: Create a Generics Class

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

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

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

// create a generics class
class GenericsClass<T> {
  // variable of T type
  private T data;
  public GenericsClass(T data) {
    this.data = data;
  }
  // method that return T type variable
  public T getData() {
    return this.data;
  }
}
```

# Example: Create a Generics Method

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

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

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

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

class DemoClass {

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

**Advantages of Java Generics**

1. Code Reusability

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

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

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

2. Compile-time Type Checking

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

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

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

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

3. Used with Collections

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

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

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

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

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

# Java Wrapper Class

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

Example 1: Primitive Types to Wrapper Objects

```
class Main {
  public static void main(String[] args) {
    // create primitive types
    int a = 5;
    double b = 5.65;
    //converts into wrapper objects
    Integer aObj = Integer.valueOf(a);
    Double bObj = Double.valueOf(b);
    if(aObj instanceof Integer) {
      System.out.println("An object of Integer is created.");
    }
    if(bObj instanceof Double) {
      System.out.println("An object of Double is created.");
    }
  }
}
```

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

# Java Command-Line Arguments

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

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

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

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

**1. To compile the code**

javac Main.java

**2. To run the code**

java Main

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

**java Main apple ball cat**

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

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

## Passing Numeric Command-Line Arguments

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

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

Example: Numeric Command-Line Arguments

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

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

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

# Thank You!!!