Learning Goals

Click to Continue

This session will help you to understand about,

- What is method references?
- How to use method references in functional programming?
- Variable arguments
- What is String Joiner?

Method References

Click to Continue

What is method references?

Reference data type is a variable we assign a reference object.

```
String C = new String("Hello");
```

Here C is a string reference data type which stores a string value.

Method reference is the same instead of a data value the variable stores the method implementation.

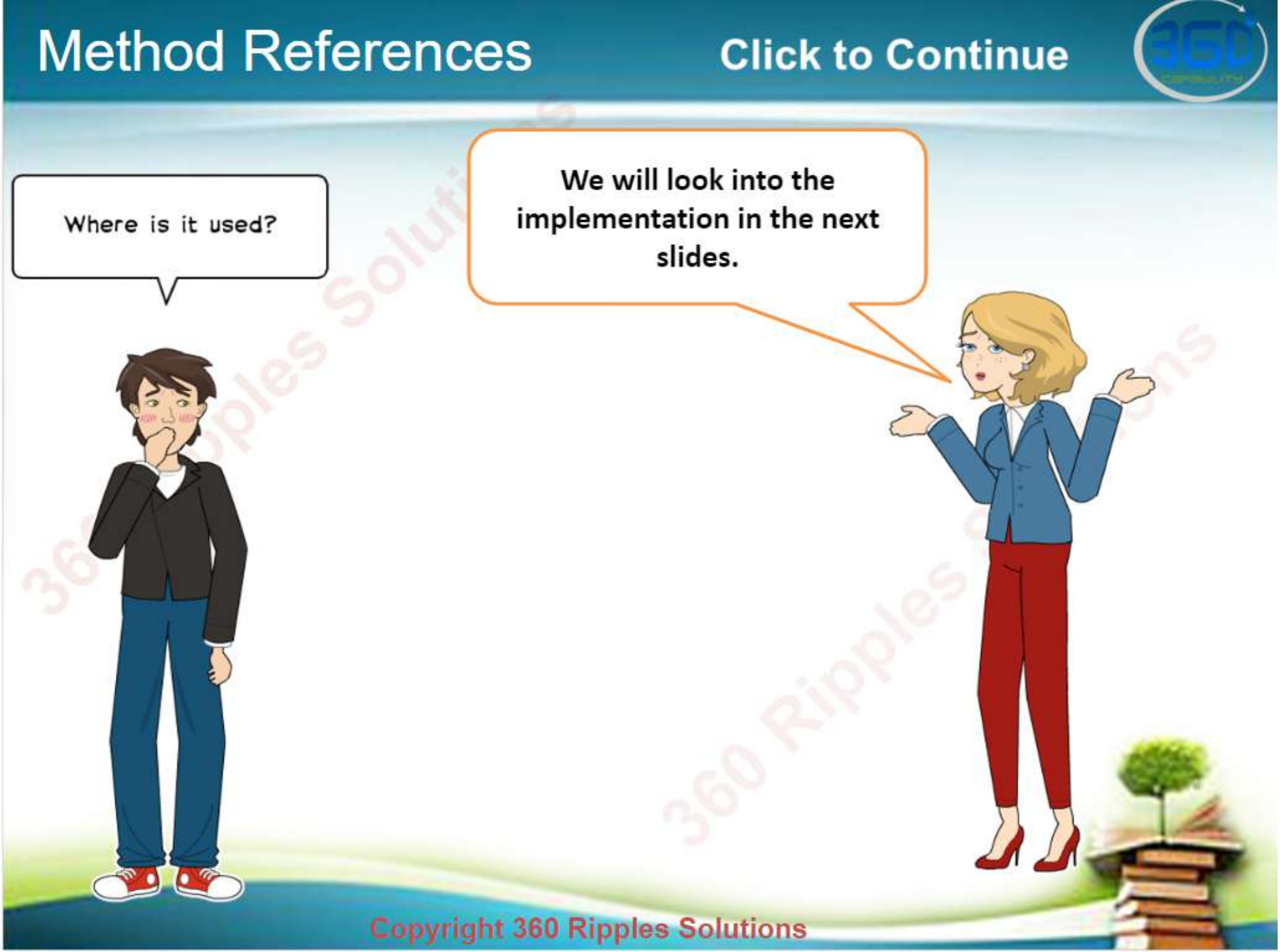
It is used to refer methods of functional interface.

```
ICalculator M = <Method injected>
```

Here M is the method reference.

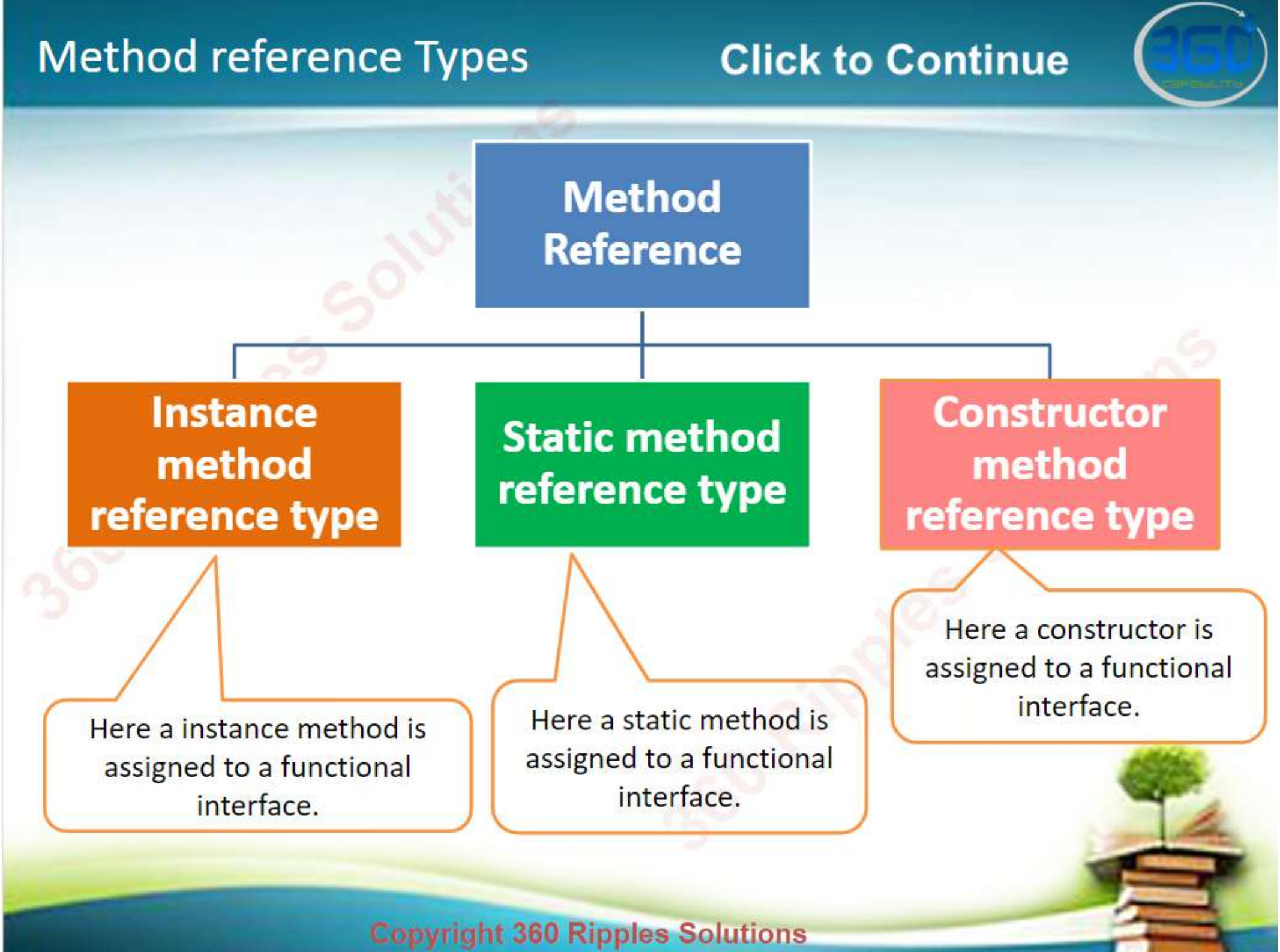
Method References

Click to Continue



Method reference Types

Click to Continue



Try it out - Instance method reference

Click to Continue

Assume there is a functional interface with a method

```
@FunctionalInterface
interface Say {
public void wish();
}
```

```
public class JavaMethodReference
{
public void sayHello() {
System.out.println("Say hello");
}
public static void main(String[]
args) {
JavaMethodReference ref = new
JavaMethodReference();
Say say = ref::sayHello;
say.wish();
}
}
```

Here the sayHello() instance method of the class is injected using :: (method reference operator) as the implementation of the functional interface Say's wish() method.

Output: The program will print "Say Hello"

NOTE: Here instead of using a lambda expression to provide the functional interface implementation. A method reference is used.

Try it out - Static method reference

Click to Continue

Assume there is a functional interface with a method

```
@FunctionalInterface
interface Say {
public void wish();
}
```

```
public class JavaMethodReference
{
public static void sayHello() {
System.out.println("Say hello
Static");
}
public static void main(String[]
args) {
Say say = JavaMethodReference
::sayHello;
say.wish();
}
}
```

Here the sayHello() static method is referred by using the Class name, rather than the object of the class.

Try it out - Constructor reference

Click to Continue

Assume there is a functional interface with a method

```
@FunctionalInterface
interface Say {
public void wish();
}
```

```
public class JavaMethodReference
{

public JavaMethodReference() {
System.out.println("Say hello
Constructor");
}
public static void main(String[]
args) {
Say say =
JavaMethodReference::new;
say.wish();
}
}
```

Here the constructor of the class "JavaMethodReference" itself is injected as functional interface implementation.

Output: The program will print "Say Hello Constructor"

Try it out – Println as method reference Click to Continue

In the below code, the **println** method of **System.out** class is injected into the for each method as method reference.

```
List<String> list = new ArrayList<String>();
list.add("apple");
list.add("orange");
list.add("grapes");
list.add("Pineapple");
list.forEach(System.out::println);
```

The same code can be written using lambda expression as below,

```
List<String> list = new ArrayList<String>();
list.add("apple");
list.add("orange");
list.add("grapes");
list.add("Pineapple");
list.forEach((s)->System.out.println(s));
```

Lambda expression

Try it out - Vargs

Click to Continue

Assume we have a method which can accept one or more parameters, this can be achieved by using an variable arguments as below,

```
public class VargsDemo {
public void add(int... a) {
int sum = 0;
for (int b : a) {
sum = sum + b;
}
System.out.println(sum);
}
public static void main(String args[]) {
VargsDemo demo = new VargsDemo();
demo.add(1, 1, 1);
demo.add(1, 1, 1, 2, 2);
}
}
```

Here the arguments is specified with .... This means the method can accept any number of int arguments.

Method is invoked by passing three values

Method is invoked by passing five values.

IMPORTANT: We cannot have more than one **vargs** as a method parameter.
Also vargs should always be the last method parameter.

Try it out - String Joiner

Click to Continue

StringJoiner is a new class introduced, to join one or more strings delimited by delimiter.

Method 1: Joins the strings with delimiter hyphen.

```
StringJoiner sj = new StringJoiner("-");
sj.add("Apple");
sj.add("Guava");
sj.add("Cherry");
sj.add("Banana");
sj.add("Avocado");
System.out.println("joined String -->" + sj);
```

Output: joined String -->Apple-Guava-Cherry-Banana-Avocado

Method 2: Joins the strings with delimiter hyphen with prefix and suffix.

```
StringJoiner sj = new StringJoiner("-", "[FRUITS:", "]");
sj.add("Apple");
sj.add("Guava");
sj.add("Cherry");
sj.add("Banana");
sj.add("Avocado");
System.out.println("joined String -->" + sj);
```

Output: joined String -->[FRUITS:Apple-Guava-Cherry-Banana-Avocado]

Method 3: This adds the new string to existing joined string.

```
sj.add("Papaya");
System.out.println("Joined String after add-->" + sj);
```

Output: Joined String after add-->[FRUITS:Apple-Guava-Cherry-Banana-Avocado-Papaya]

Method 4: This merges two string joiner. In the below example the fruits joined string is merged with the vegetables

```
StringJoiner sj1 = new StringJoiner("::", "[VEGETABLES:", "]");
sj1.add("Tomato");
sj1.add("Beans");
sj1.add("Broccoli");
sj1.add("Carrot");
sj1.merge(sj);
System.out.println("joined String after merge-->" + sj1);
```

Output: joined String after merge-->[VEGETABLES:Tomato::Beans::Broccoli::Carrot::Apple-Guava-Cherry-Banana-Avocado-Papaya]

Click to Continue

Method overloading type promotion is the technique where one type is promoted to another implicitly if no matching datatype is found in overloaded method.

```
public class MethodOverloadingPromotion {

public void divide(int a, long d) {
System.out.println("int invoked");
}
public void divide(int a, double d) {
System.out.println("double invoked");
}

public static void main(String args[]) {
MethodOverloadingPromotion e = new
MethodOverloadingPromotion();
e.divide(10, 100);

e.divide(10, 100.888f);
}
}
```

Here two int value is passed to the divide method. The second int will be promoted to long automatically and method `divide(int a, long d)` will be triggered.

Similarly here the float will automatically be promoted to double `divide(int a, double d)` will be triggered..

Click to Continue

IMPORTANT: Float can be automatically converted to double. But vice versa is not possible. So a big data type cannot be promoted to smaller data type like long to int.

Try it out – Multiple Catch block **Click to Continue**

Java 7 introduced a feature where multiple exceptions can be caught in a single catch block.

```
try (FileOutputStream os = new FileOutputStream("")) {
} catch (FileNotFoundException | NullPointerException  |
ArithmeticException e ) {
e.printStackTrace();
}
catch( Exception e)
{
e.printStackTrace();
}
```

Here three exceptions are caught in single exception block and handled. | symbol used to separate the exceptions caught

IMPORTANT: The order in which you catch exceptions is critical. You cannot catch a parent exception followed by the child exception.

```
catch (Exception | NullPointerException  | ArithmeticException e )
```

This throws a compilation error as Exception (which is the parent of all exceptions) is caught first followed by other exceptions.

Try it out – Try block with references

Click to Continue

```
try {
connection = DriverManager.getConnection();
stmt = connection.createStatement();
String query = "INSERT INTO student VALUES"
stmt.executeUpdate(query);
} catch (SQLException e) {
        // handle exception
} finally {
        stmt.close();
        connection.close();
}
```

Here the connection and statement should be closed in finally block by the programmer. Else connection leak will happen.

To avoid this java introduced , try with references.

This is called try with references.

```
try (Connection connection =
DriverManager.getConnection();
Statement stmt = connection.createStatement();)
{
String query = "INSERT INTO
stmt.executeUpdate(query);
} catch (SQLException e) {
        // handle exception
}
```

Here the connection and statement objects are initialized inside the try statement.
Java will automatically close the statements and connection object after program execution. Finally block not needed.