Deutsche Bank
Corporate Division

Day 3

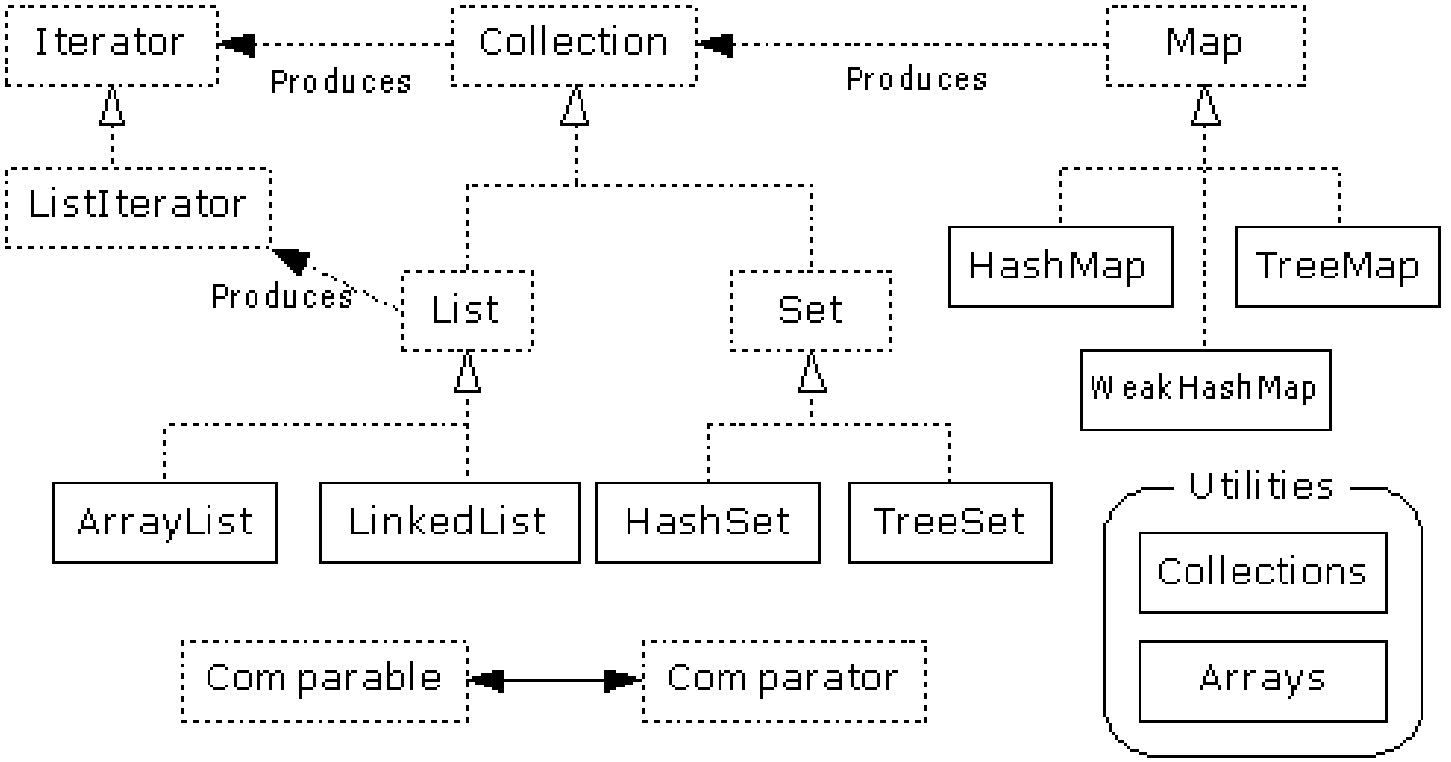
INTRODUCTION IN JAVA - III

PPT-Master Version 01. There will be an extensive update with the new Deutsche Bank font and sample charts at the end of August 2024.

8 May 2024, Speaker name


Deutsche Bank
Corporate Division
Collections Framework
Exploring List, Set, and Map Implementations for
Organized Data Storage and Retrieval.

Collections Framework Diagram

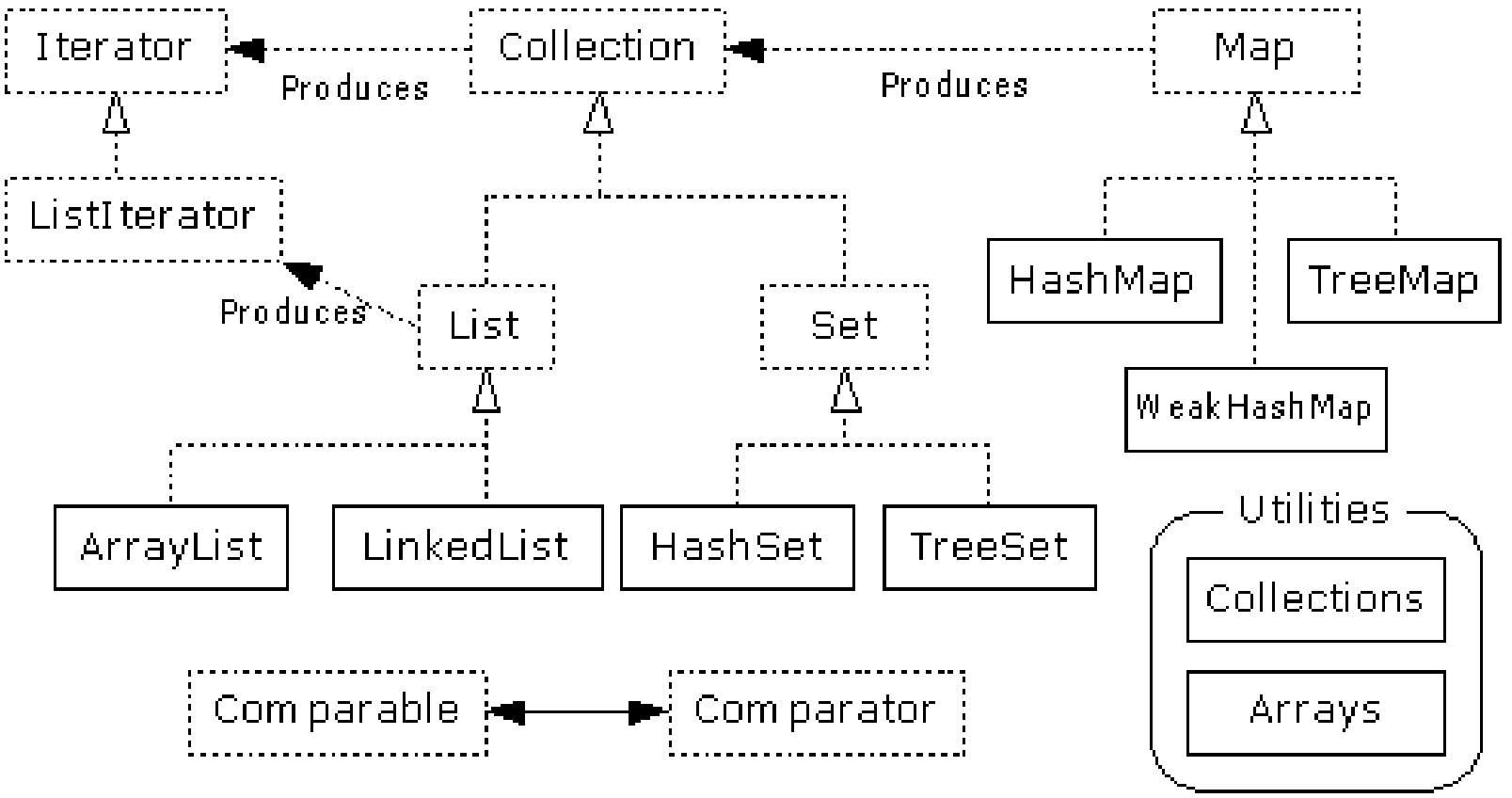


Interfaces, Implementations, and Algorithms

Collection Interface

- Defines fundamental methods
  - `int size();`
  - `boolean isEmpty();`
  - `boolean contains(Object element);`
  - `boolean add(Object element);    // Optional`
  - `boolean remove(Object element); // Optional`
  - `Iterator iterator();`
- These methods are enough to define the basic behavior of a collection
- Provides an Iterator to step through the elements in the Collection

List Interface Context



List Interface

- The List interface adds the notion of order to a collection
- The user of a list has control over where an element is added in the collection
- Lists typically allow duplicate elements
- Provides a List Iterator to step through the elements in the list.

List Implementations

- ArrayList
  - low cost random access
  - high cost insert and delete
  - array that resizes if need be
- LinkedList
  - sequential access
  - low cost insert and delete
  - high cost random access

ArrayList overview

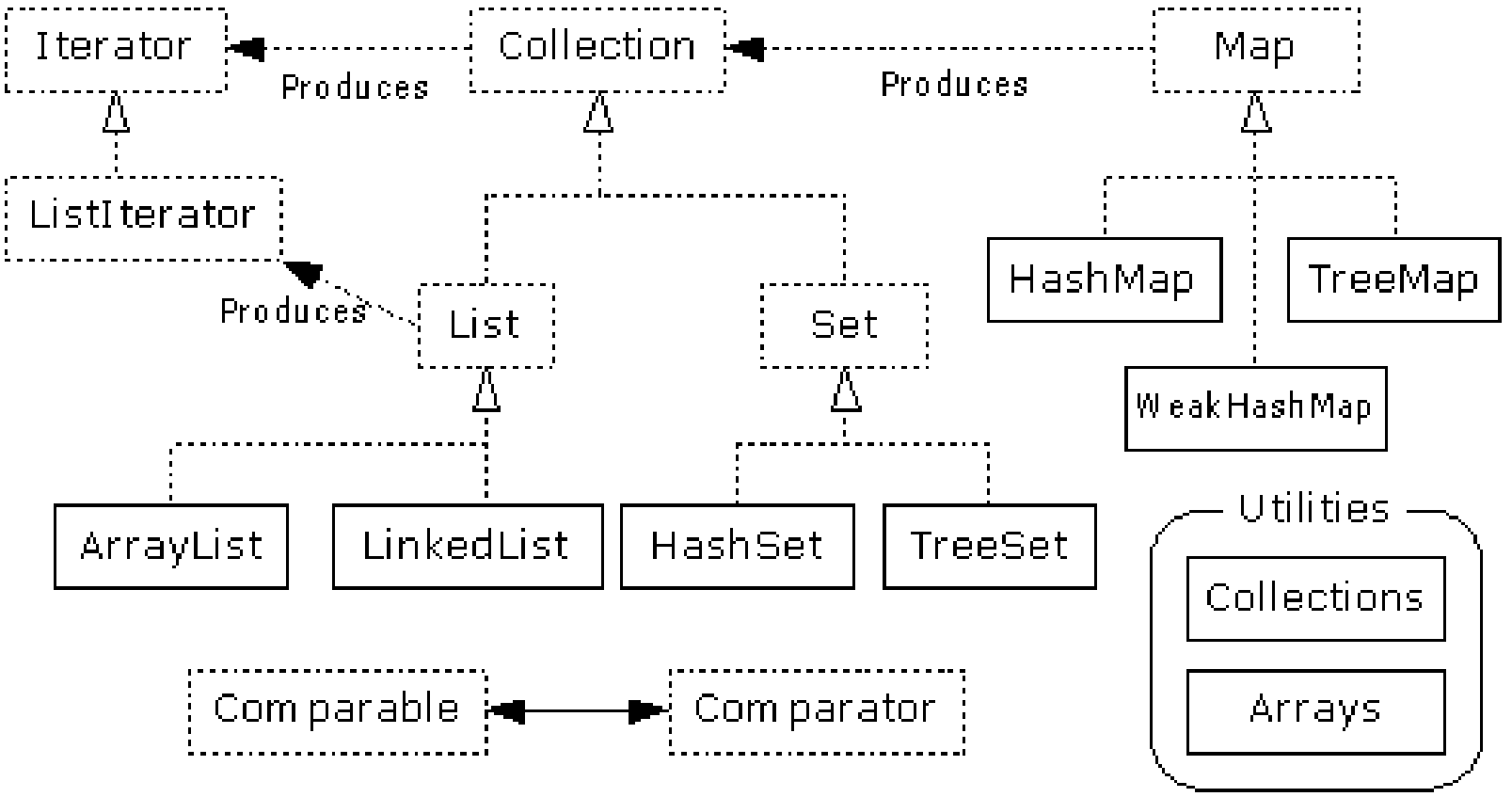
- Constant time positional access (it's an array)
- One tuning parameter, the initial capacity

```
public ArrayList(int initialCapacity) {
        super();
        if (initialCapacity < 0)
                throw new IllegalArgumentException(
                        "Illegal Capacity:
"+initialCapacity);
        this.elementData = new
Object[initialCapacity];
}
```

ArrayList Methods

- The indexed get and set methods of the List interface are appropriate to use since ArrayLists are backed by an array
  - Object get(int index)
  - Object set(int index, Object element)
- Indexed add and remove are provided, but can be costly if used frequently
  - void add(int index, Object element)
  - Object remove(int index)
- May want to resize in one shot if adding many elements
  - void ensureCapacity(int minCapacity)

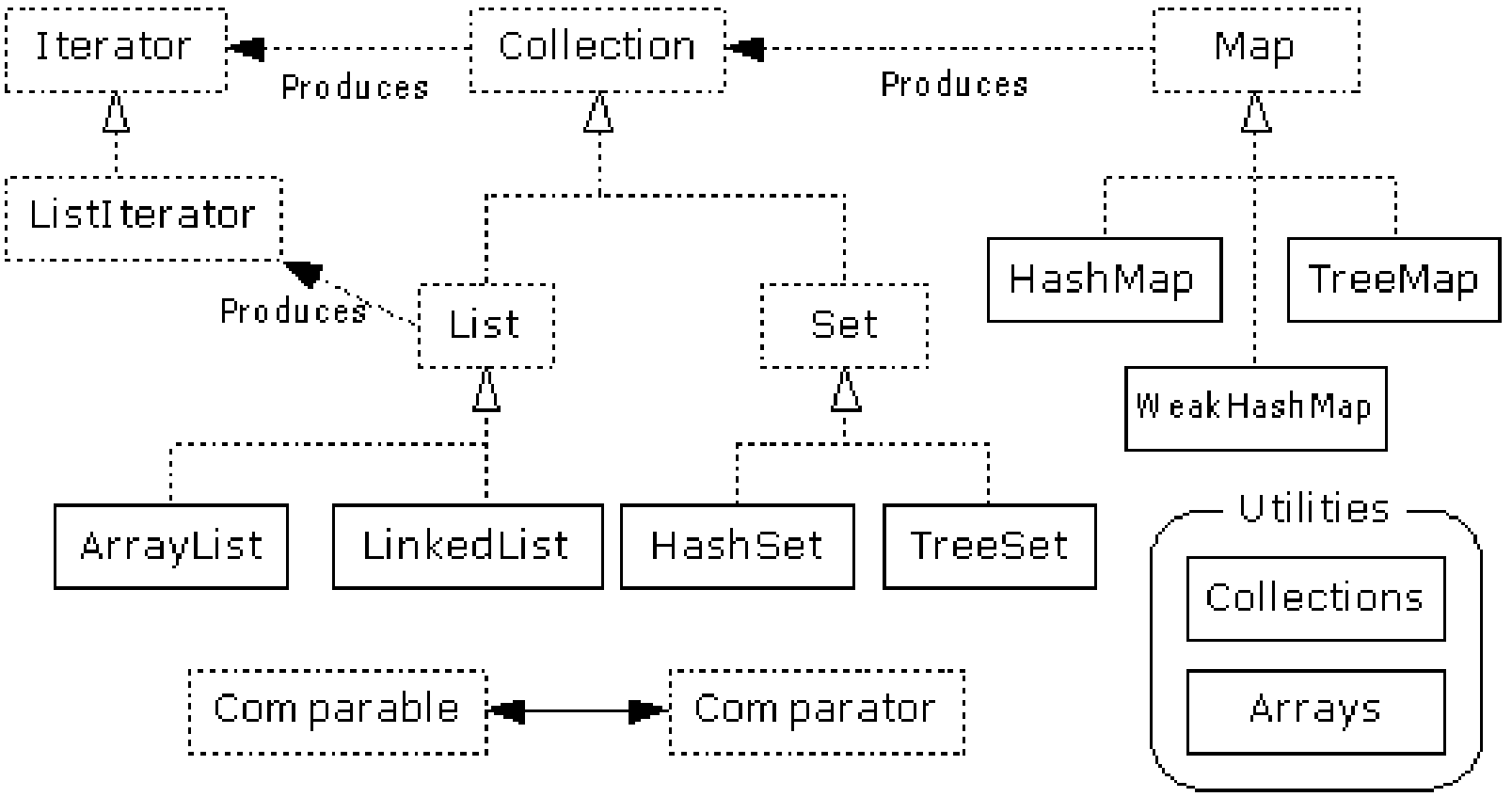
Set Interface Context



Set Interface

- Same methods as Collection
  - different contract - no duplicate entries
- Defines two fundamental methods
  - boolean add(Object o) - reject duplicates
  - Iterator iterator()
- Provides an Iterator to step through the elements in the Set
  - No guaranteed order in the basic Set interface
  - There is a SortedSet interface that extends Set

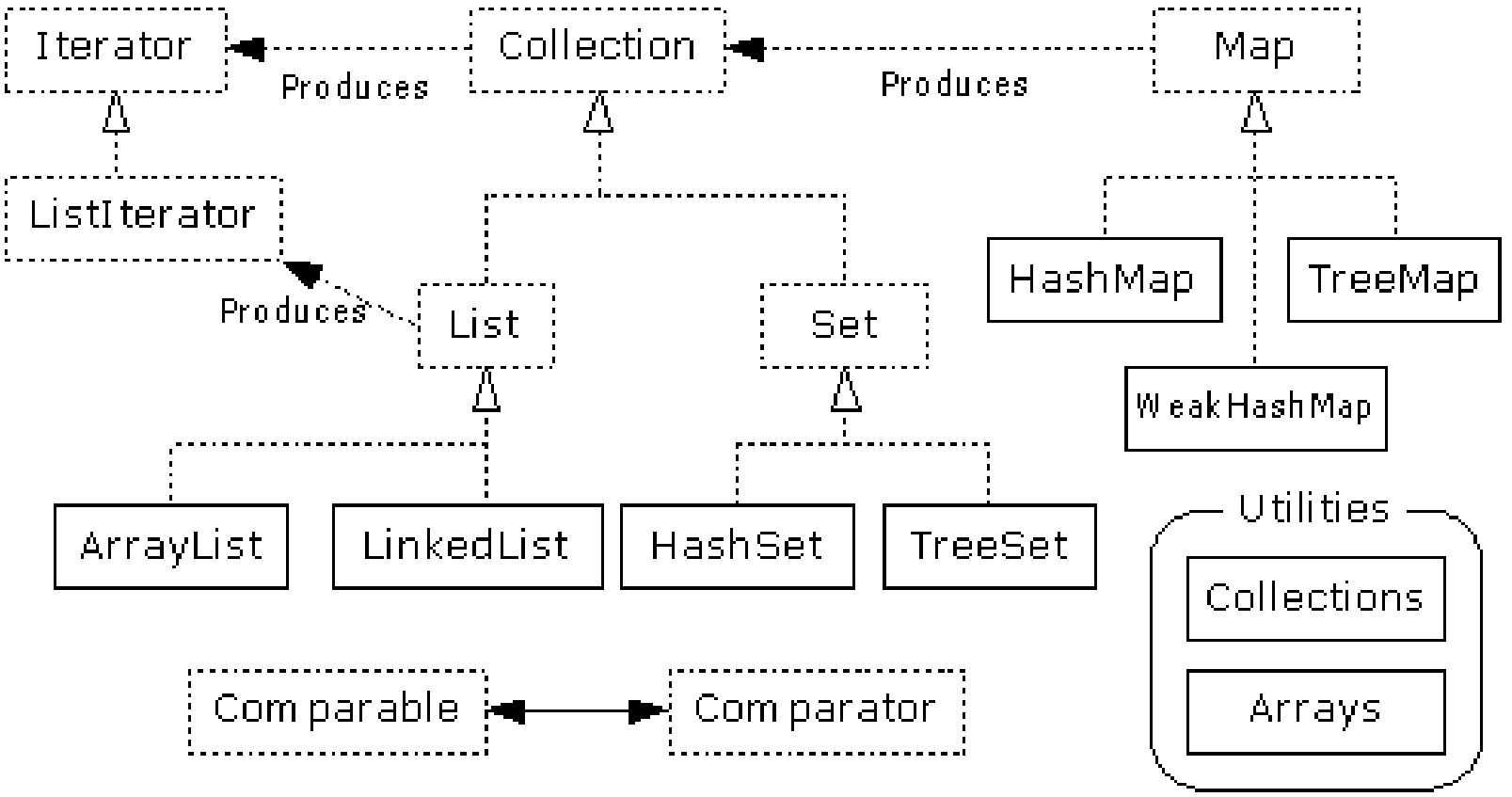
HashSet and TreeSet Context



HashSet

- Find and add elements very quickly
  - uses hashing implementation in HashMap
- Hashing uses an array of linked lists
  - The hashCode() is used to index into the array
  - Then equals() is used to determine if element is in the (short) list of elements at that index
- No order imposed on elements
- The hashCode() method and the equals() method must be compatible
  - if two objects are equal, they must have the same hashCode() value

Map Interface Context



Map Interface

- Stores key/value pairs
- Maps from the key to the value
- Keys are unique
  - a single key only appears once in the Map
  - a key can map to only one value
- Values do not have to be unique

Map Methods

```
Object put(Object key, Object value)
Object get(Object key)
Object remove(Object key)
boolean containsKey(Object key)
boolean containsValue(Object value)
int size()
boolean isEmpty()
```

Map Views

- A means of iterating over the keys and values in a Map
- Set keySet()
  - returns the Set of keys contained in the Map
- Collection values()
  - returns the Collection of values contained in the Map. This Collection is not a Set, as multiple keys can map to the same value.
- Set entrySet()
  - returns the Set of key-value pairs contained in the Map. The Map interface provides a small nested interface called Map.Entry that is the type of the elements in this Set.

Deutsche Bank
Corporate Division

EXAMPLE

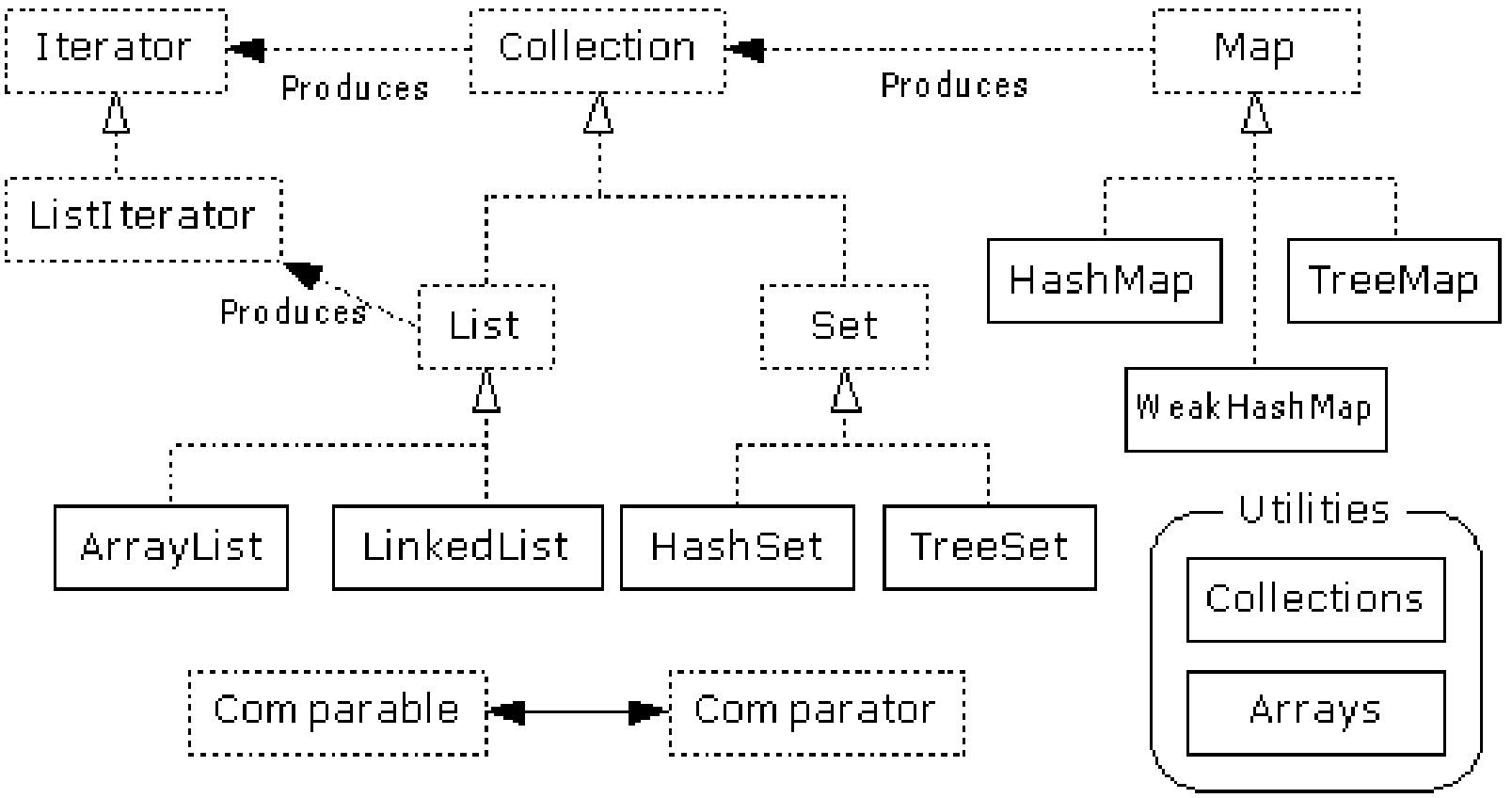
8 May 2024, Speaker name

Bulk Operations

- In addition to the basic operations, a Collection may provide “bulk” operations

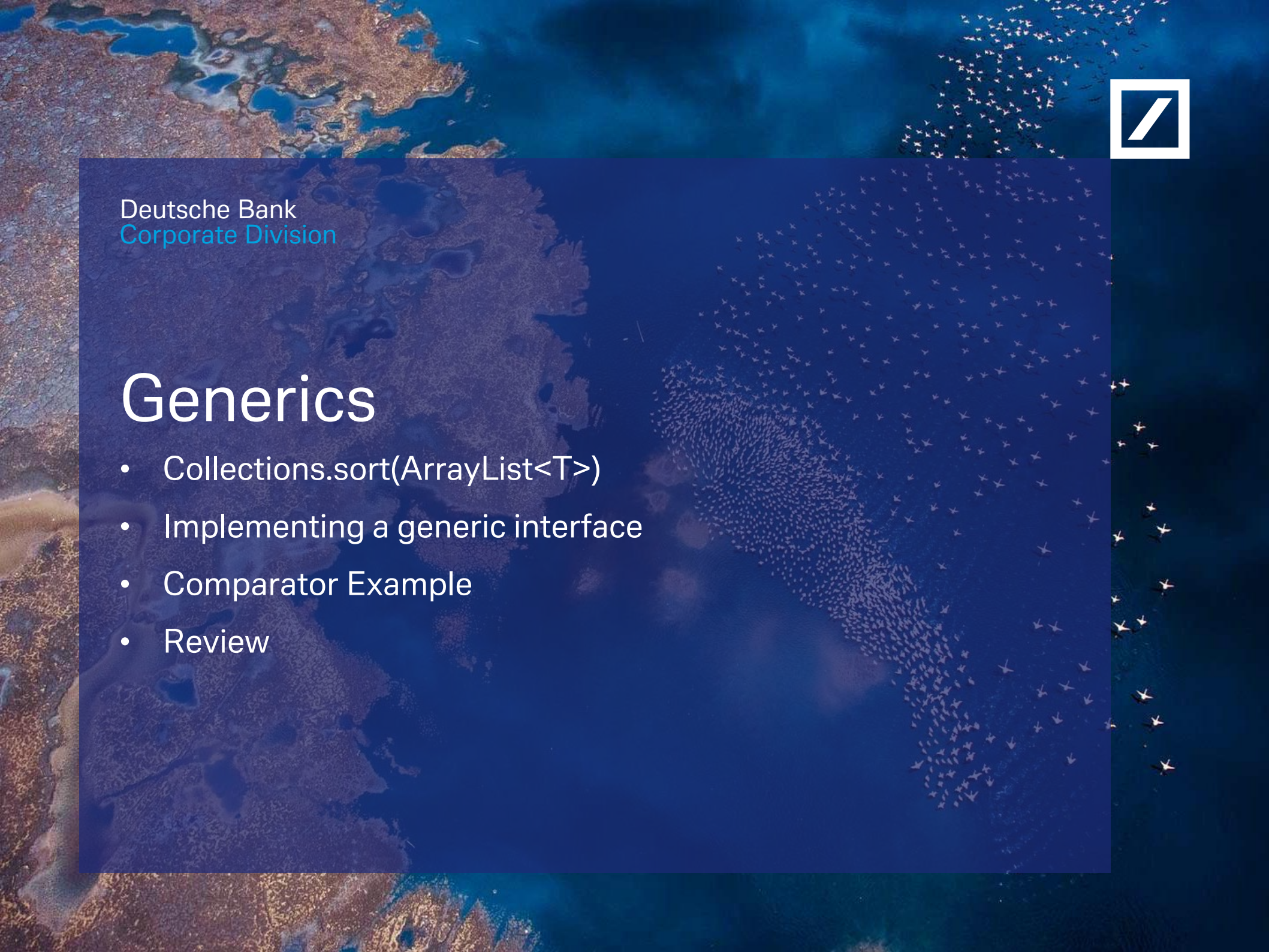
```
boolean containsAll(Collection c);
boolean addAll(Collection c);    // Optional
boolean removeAll(Collection c); // Optional
boolean retainAll(Collection c); // Optional
void clear();                    // Optional
Object[] toArray();
Object[] toArray(Object a[]);
```

Utilities Context



Utilities Context

- The Collections class provides a number of static methods for fundamental algorithms
- Most operate on Lists, some on all Collections
  - Sort, Search, Shuffle
  - Reverse, fill, copy
  - Min, max
- Wrappers
  - synchronized Collections, Lists, Sets, etc
  - unmodifiable Collections, Lists, Sets, etc



Generics

- Collections.sort(ArrayList<T>)
- Implementing a generic interface
- Comparator Example
- Review

Collections.sort(ArrayList<T>) – how can it work?

- Framework's `Collections` class can sort known types

```
ArrayList<Integer> ages =
                          new ArrayList<Integer>(Arrays.asList(30,22,13,18));

Collections.sort(ages);

for (int no : ages) {
    System.out.println(no);
}
```

```
13
18
22
30
```

```
ArrayList<String> names =
                                    new
ArrayList<String>(Arrays.asList("zak","Fred","john","bob"));

Collections.sort(names, Collections.reverseOrder() );

for (String name : names) {
    System.out.println(name);
}
```

```
zak
john
bob
Fred
```

sort() does not sort user defined types (Java or C#)

```
class Account {
  int id, balance;
  String name;

  public Account(int id, int balance, String name) {
    this.id = id;
    this.balance = balance;
    this.name = name;
  }
}
```

```
ArrayList<Account> accounts = new ArrayList<Account> (
                          Arrays.asList(new Account(111, 1000,
"Bob"),
                                        new Account(222, 5000, "Wilma"),
                                         new Account(333, 2000,
"Abby")));
```

```
Compilation error: The method sort(List<T>) in the type Collections
is not applicable for the arguments (ArrayList<Account>)
```

Implementing a generic interface

- Interface `Comparable<T>`

  If the Collections.sort() method cannot receive a unknown type

  What if could receive objects of a known interface

  The kind of interface that would enable it to compare two references

Java: Class must implement Comparable

```
class Account implements Comparable{
    public int id, balance;
    public String name;

    public Account(int id, int balance, String name) {
          this.id = id;
          this.balance = balance;
          this.name = name;
    }

    @Override
    public int compareTo(Object other) {
          return this.balance - ((Account)other).balance;
    }
}
```

Used by the sort method to sort the accounts
Offers only one way of sorting accounts

Comparator example

```
class AccountBalanceComparer implements Comparator<Account> {
        public int compare(Account a1, Account a2) {
                return a1.balance - a2.balance;
        }
}

class AccountIDComparer implements Comparator<Account> {
        public int compare(Account a1, Account a2) {
                return a1.id - a2.id;
        }
}
```

```
ArrayList<Account> accounts = new ArrayList<Account>();
accounts.add(new Account(111, 1000, "Bob"));
accounts.add(new Account(222, 5000, "Wilma"));
accounts.add(new Account(333, 2000, "Abby"));

Collections.sort(accounts, new AccountIDComparer());

Collections.sort(accounts, new AccountBalanceComparer());
```

Review

- The need for generic interfaces

- Framework interfaces

  Comparable<T> – to define a natural sort sequence

  Comparator<T> – to define alternative sort sequence(s)

- Many sort() routines (or ctors of objects with sort()) are overloaded to (optionally) receive a Comparator<T>


Deutsche Bank
Corporate Division
Working with Lambda Expressions and Functional Interfaces for Concise Code.

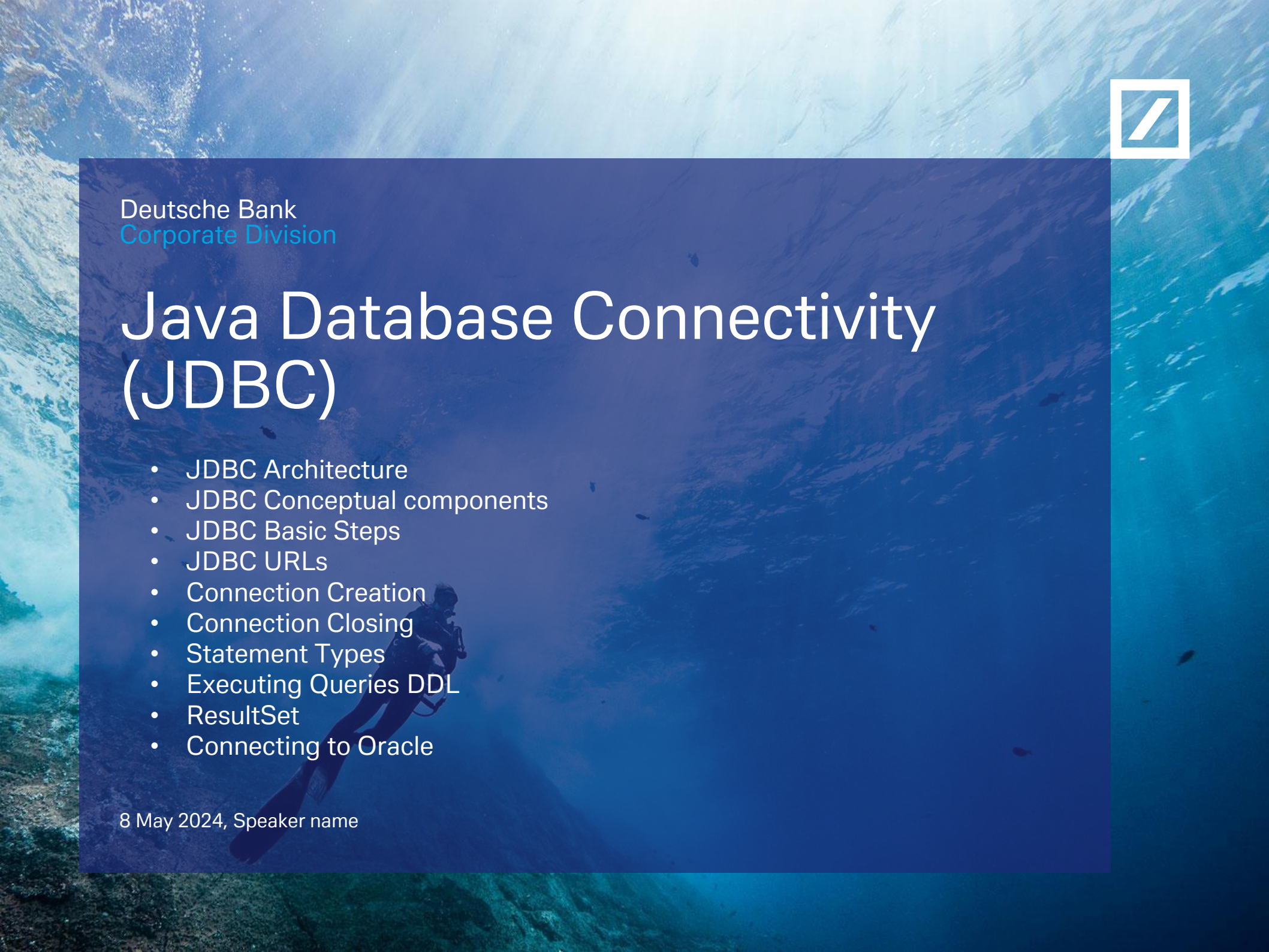
LAMBDA

```
public class LambdaExample {
    public static void main(String[] args) {
     // Lambda expression to add two integers
     MathOperation addition = (a, b) -> a +
     b;

     // Lambda expression to multiply two
     integers
     MathOperation multiplication = (int a,
     int b) -> { return a * b; };

     int result1 = addition.operation(10, 5);
     int result2 =
     multiplication.operation(7, 3);

     System.out.println("Addition Result: " +
     result1);
     System.out.println("Multiplication
     Result: " + result2);
     }
    interface MathOperation {
     int operation(int a, int b);
    }
}
```

In this example:

MathOperation is a functional interface with a single abstract method operation(int a, int b).The lambda expressions (a, b) -> a + b and (int a, int b) -> { return a * b; } implement this interface.We create instances of these lambdas (addition and multiplication) and call their operation method to perform addition and multiplication.


Deutsche Bank
Corporate Division
JAVA DATE TIME (DEMO)



Deutsche Bank
Corporate Division

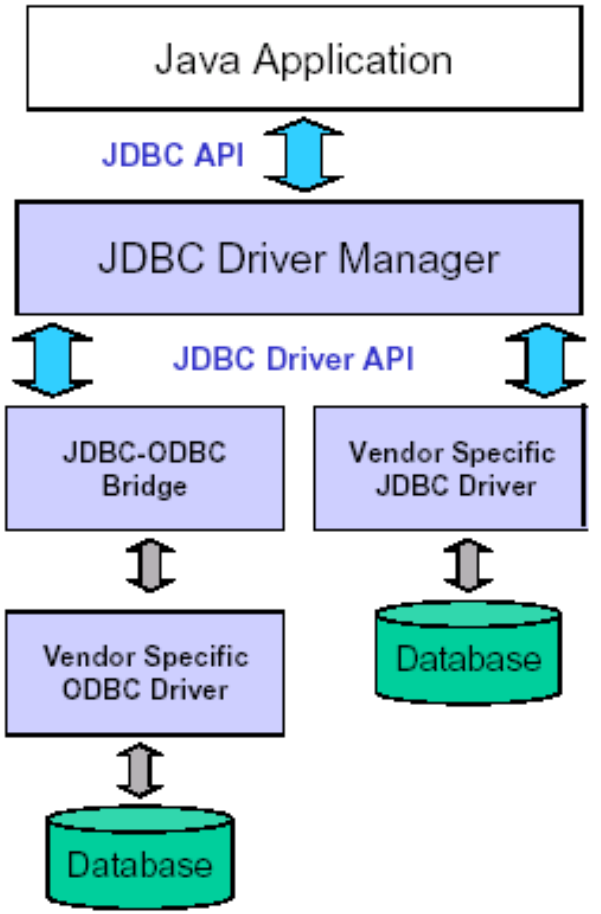
Java Database Connectivity (JDBC)

- JDBC Architecture
- JDBC Conceptual components
- JDBC Basic Steps
- JDBC URLs
- Connection Creation
- Connection Closing
- Statement Types
- Executing Queries DDL
- ResultSet
- Connecting to Oracle

8 May 2024, Speaker name

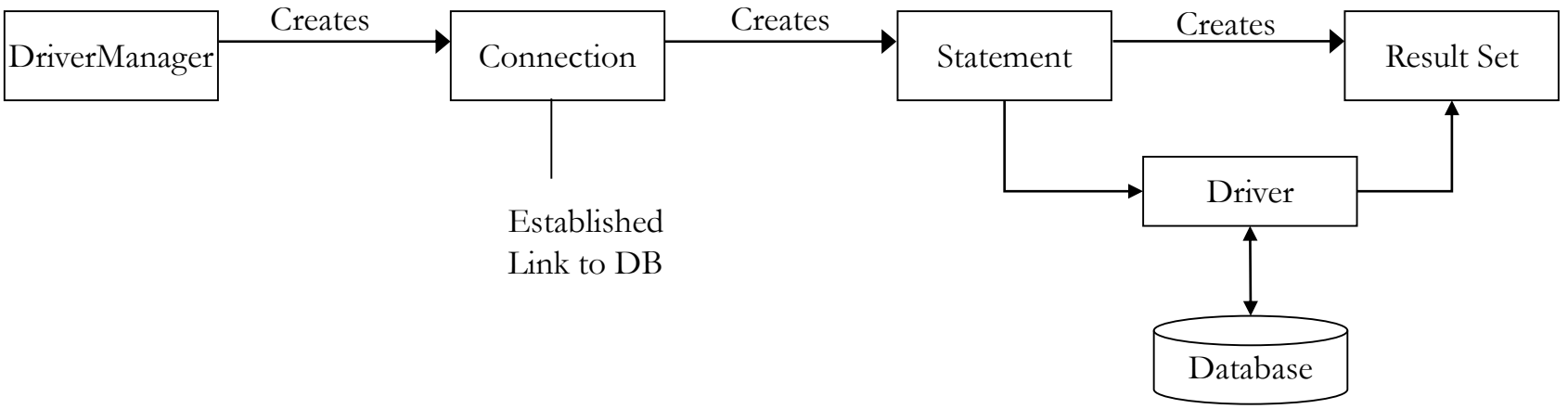
JDBC Architecture

- JDBC Consists of two parts:
  - JDBC API, a purely Java-based API
  - JDBC Driver Manager, which communicates with vendor-specific drivers that perform the real communication with the database
- Translation to the vendor format occurs on the client
  - No changes needed to the server
  - Driver (translator) needed on client



JDBC Conceptual Components

- **Driver Manager:** Loads database drivers and manages connections between the application and the driver
- **Driver:** Translates API calls into operations for specific database
- **Connection:** Session between application and data source
- **Statement:** SQL statement to perform query or update
- **Metadata:** Information about returned data, database, & driver
- **Result Set:** Logical set of columns and rows of data returned by executing a statement



JDBC Basic Steps

- Import the necessary classes
- Load the JDBC driver
- Identify the data source (Define the Connection URL)
- Establish the Connection
- Create a Statement Object
- Execute query string using Statement Object
- Retrieve data from the returned ResultSet Object
- Close ResultSet & Statement & Connection Object in order

JDBC URLs

- JDBC Urls provide a way to identify a database
- Syntax:

  <protocol>:<subprotocol>:<protocol>

  - Protocol: Protocol used to access database (jdbc here)
  - Subprotocol: Identifies the database driver
  - Subname: Name of the resource
- Example
  - Jdbc:cloudscape:Movies
  - Jdbc:odbc:Movies

Connection Creation

- Required to communicate with a database via JDBC
- Three separate methods:

  public static Connection getConnection(String url)

  public static Connection getConnection(String url, Properties info)

  public static Connection getConnection(String url, String user, String password)
- Code Example (Access)

```
try {// Load the driver class
    System.out.println("Loading Class driver");
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    // Define the data source for the driver
    String sourceURL = "jdbc:odbc:music";
    // Create a connection through the DriverManager class
    System.out.println("Getting Connection");
    Connection databaseConnection = DriverManager.getConnection(sourceURL);
    }
catch (ClassNotFoundException cnfe) {
        System.err.println(cnfe); }
 catch (SQLException sqle) {
        System.err.println(sqle);}
```

Connection Creation

- Code Example (Oracle)

```
try {
        Class.forName("oracle.jdbc.driver.OracleDriver");
        String sourceURL = "jdbc:oracle:thin:@delilah.bus.albany.edu:1521:databasename";
        String user = "goel";
        String password = "password";
        Connection databaseConnection=DriverManager.getConnection(sourceURL,user,
    password );
        System.out.println("Connected Connection"); }
catch (ClassNotFoundException cnfe) {
        System.err.println(cnfe); }
catch (SQLException sqle) {
        System.err.println(sqle);}
```

Connection Closing

- Each machine has a limited number of connections (separate thread)
- If connections are not closed the system will run out of resources and freeze
- Syntax: public void close() throws SQLException

- Naïve Way:

```
try {
    Connection conn
    = DriverManager.getConnection(url);
     // Jdbc Code
    …
    } catch (SQLException sqle) {
      sqle.printStackTrace();
    }
    conn.close();
```

- SQL exception in the Jdbc code will prevent execution to reach conn.close()

- Correct way (Use the finally clause)

```
try{
Connection conn =
    Driver.Manager.getConnection(url);
    // JDBC Code
    } catch (SQLException sqle) {
        sqle.printStackTrace();
    } finally {
        try {
            conn.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
}
```

Statement Types

- Statements in JDBC abstract the SQL statements
- Primary interface to the tables in the database
- Used to create, retrieve, update & delete data (CRUD) from a table
  - Syntax: Statement statement = connection.createStatement();
- Three types of statements each reflecting a specific SQL statements
  - Statement
  - PreparedStatement
  - CallableStatement

Statement Syntax

- Statement used to send SQL commands to the database
  - Case 1: ResultSet is non-scrollable and non-updateable

    public Statement createStatement() throws SQLException

    Statement statement = connection.createStatement();
  - Case 2: ResultSet is non-scrollable and/or non-updateable

    public Statement createStatement(int, int) throws SQLException

    Statement statement = connection.createStatement();
  - Case 3: ResultSet is non-scrollable and/or non-updateable and/or holdable

    public Statement createStatement(int, int, int) throws SQLException

    Statement statement = connection.createStatement();
- PreparedStatement

  public PreparedStatement prepareStatement(String sql) throws SQLException

  PreparedStatement pstatement = prepareStatement(sqlString);
- CallableStatement used to call stored procedures

  public CallableStatement prepareCall(String sql) throws SQLException

Executing Queries Data Definition Language (DDL)

- Data definition language queries use executeUpdate
- Syntax: int executeUpdate(String sqlString) throws SQLException
  - It returns an integer which is the number of rows updated
  - sqlString should be a valid String else an exception is thrown
- Example 1: Create a new table
- Statement statement = connection.createStatement();

```
String sqlString =
"Create Table Catalog"
+ "(Title Varchar(256) Primary Key Not Null,"+
+ "LeadActor Varchar(256) Not Null, LeadActress Varchar(256) Not Null,"
+ "Type Varchar(20) Not Null, ReleaseDate Date Not NULL )";
Statement.executeUpdate(sqlString);
```

  - executeUpdate returns a zero since no row is updated

Executing Queries Data Definition Language (DDL) EXAMPLE

- Example 2: Update table

```
Statement statement = connection.createStatement();
String sqlString =
 "Insert into Catalog"
+ "(Title, LeadActor, LeadActress, Type, ReleaseDate)"
+ "Values('Gone With The Wind', 'Clark Gable', 'Vivien Liegh',"
+ "'Romantic', '02/18/2003' "
Statement.executeUpdate(sqlString);
```

  - executeUpdate returns a 1 since one row is added

Executing Queries Data Manipulation Language (DML)

- Data definition language queries use executeQuery
- Syntax

  ResultSet executeQuery(String sqlString) throws SQLException

  - It returns a ResultSet object which contains the results of the Query
- Example 1: Query a table

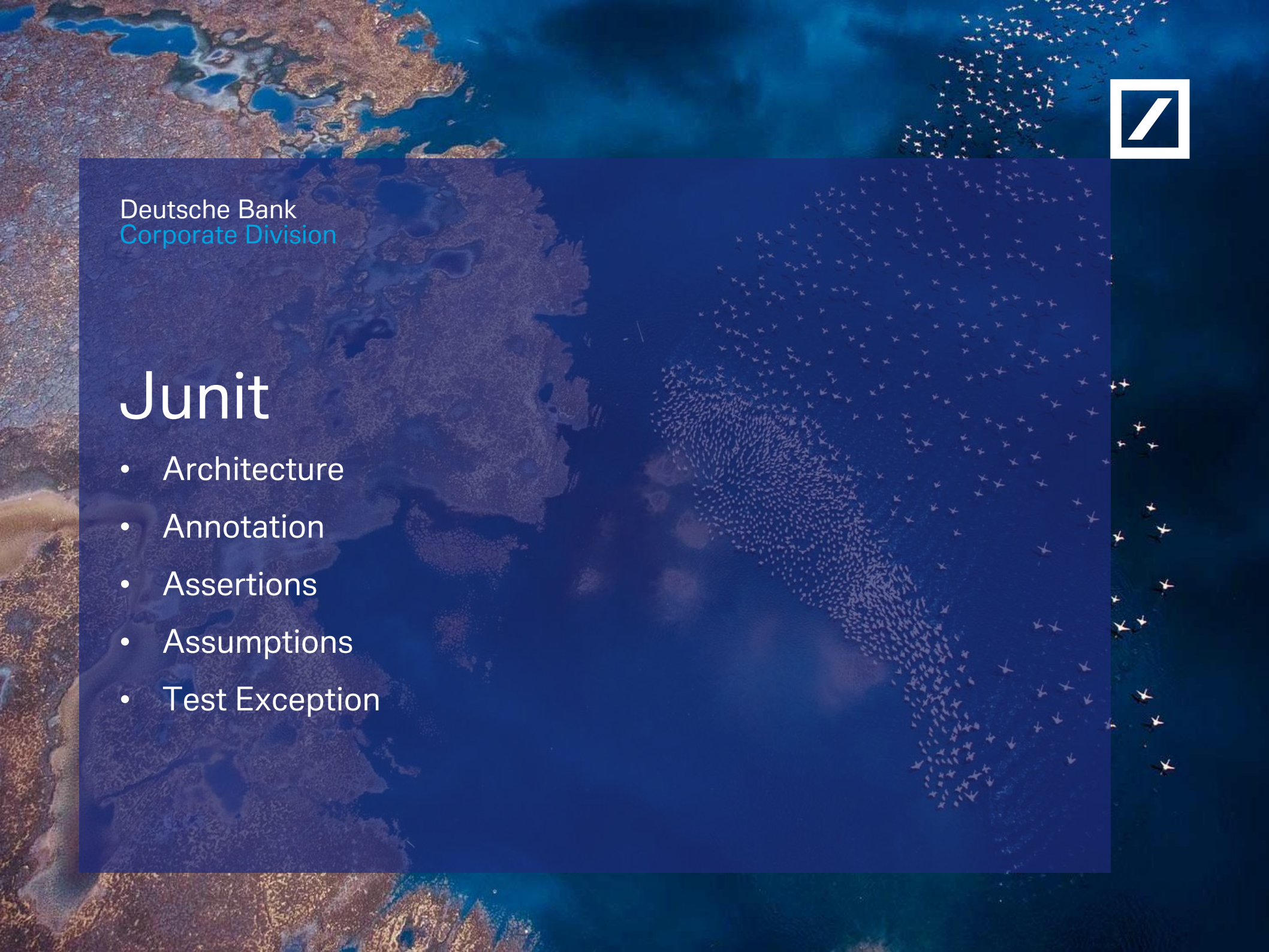
```
Statement statement = connection.createStatement();
String sqlString = “Select Catalog.Title, Catalog.LeadActor, Catalog.LeadActress,” +
                                        “Catalog.Type, Catalog.ReleaseDate From
Catalog”;
ResultSet rs = statement.executeQuery(sqlString);
```

ResultSet Definition

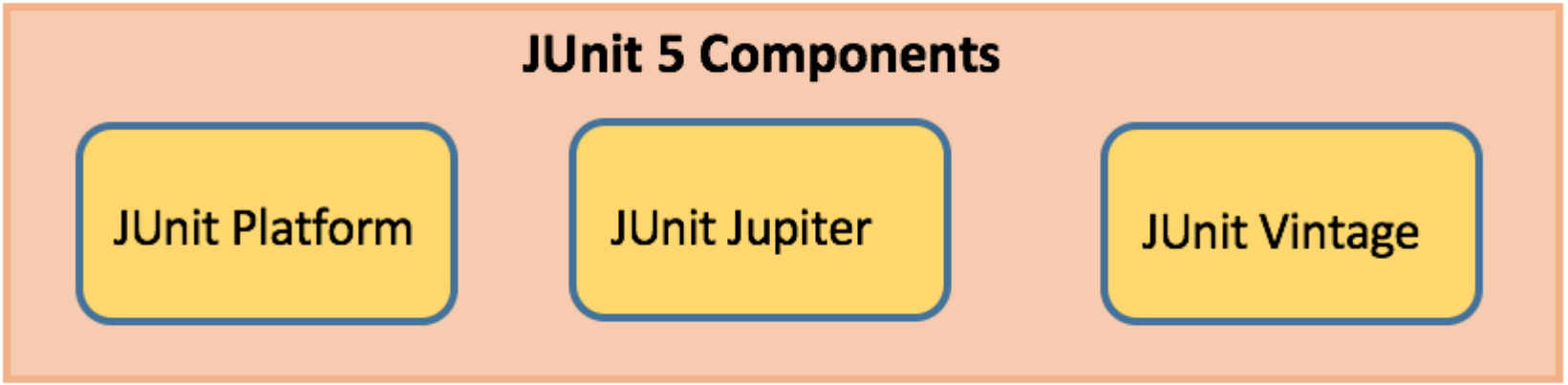
- ResultSet contains the results of the database query that are returned
- Allows the program to scroll through each row and read all columns of data
- ResultSet provides various access methods that take a column index or column name and returns the data
  - All methods may not be applicable to all resultsets depending on the method of creation of the statement.
- When the executeQuery method returns the ResultSet the cursor is placed before the first row of the data
  - Cursor refers to the set of rows returned by a query and is positioned on the row that is being accessed
  - To move the cursor to the first row of data next() method is invoked on the resultset
  - If the next row has a data the next() results true else it returns false and the cursor moves beyond the end of the data
- First column has index 1, not 0

Connecting to Oracle

```
import java.sql.*;

public class ConnectOracle {

    /**
     * This is the main function which connects to the
        Oracle database
     * and executes a sample query
     *
     * @param String[] args - Command line arguments
        for the program
     * @return void
     * @exception none
     *
     */
    public static void main(String[] args) {

// Load the driver
        try {
            // Load the driver class

        Class.forName("oracle.jdbc.driver.OracleDriver"
        );

            // Define the data source for the driver
            String sourceURL
             =
        "jdbc:oracle:thin:@delilah.bus.albany.edu:1521:
        bodb01";

            // Create a connection through the
        DriverManager class
            String user = "goel";
            String password = "goel";
            Connection databaseConnection
             = DriverManager.getConnection(sourceURL,
        user, password);
            System.out.println("Connected to Oracle");

            // Create a statement
            Statement statement =
        databaseConnection.createStatement();

            // Create a query String
            String sqlString = "SELECT artistid,
        artistname FROM artistsandperformers";

            // Close Connection
            databaseConnection.close();
        }
        catch (ClassNotFoundException cnfe) {
            System.err.println(cnfe);
        }
        catch (SQLException sqle) {
            System.err.println(sqle);
        }
        }
}
```


Deutsche Bank
Corporate Division
Junit
• Architecture
• Annotation
• Assertions
• Assumptions
• Test Exception

JUNIT ARCHITECTURE



JUnit Platform

- Launches testing frameworks on the JVM
- Has TestEngine API used to build a testing framework that runs on the JUnit platform

JUnit Jupiter

- Blend of new programming model for writing tests and extension model for extensions
- Addition of new annotations like @BeforeEach, @AfterEach, @AfterAll, @BeforeAll etc.

JUnit Vintage

- Provides support to execute previous JUnit version 3 and 4 tests on this new platform

JUNIT MAVEN DEPENDENCIES

```
<dependency>
     <groupId>org.junit.jupiter</groupId>
     <artifactId>junit-jupiter-engine</artifactId>
     <version>5.1.1</version>
     <scope>test</scope>
</dependency>
<dependency>
     <groupId>org.junit.platform</groupId>
     <artifactId>junit-platform-runner</artifactId>
     <version> 1.1.1</version>
     <scope>test</scope>
</dependency>
```

Junit Annotations

Annotation	Description
@Test	Denotes a test method
@DisplayName	Declares a custom display name for the test class or test method
@BeforeEach	Denotes that the annotated method should be executed before each test method
@AfterEach	Denotes that the annotated method should be executed after each test method
@BeforeAll	Denotes that the annotated method should be executed before all test methods
@AfterAll	Denotes that the annotated method should be executed after all test methods
@Disable	Used to disable a test class or test method
@Nested	Denotes that the annotated class is a nested, non-static test class
@Tag	Declare tags for filtering tests
@ExtendWith	Register custom extensions

JUnit Assertions

Assertion	Description
`assertEquals(expected, actual)`	Fails when expected does not equal actual
`assertFalse(expression)`	Fails when expression is not false
`assertNull(actual)`	Fails when actual is not null
`assertNotNull(actual)`	Fails when actual is null
`assertAll()`	Group many assertions and every assertion is executed even if one or more of them fails
`assertTrue(expression)`	Fails if expression is not true
`assertThrows()`	Class to be tested is expected to throw an exception

JUnit5 Assumptions

Assertion	Description
assumeTrue	Execute the body of lamda when the positive condition hold else test will be skipped
assumeFalse	Execute the body of lamda when the negative condition hold else test will be skipped
assumingThat	Portion of the test method will execute if an assumption holds true and everything after the lambda will execute irrespective of the assumption in assumingThat() holds

Deutsche Bank
Corporate Division

JUnit Nested Test Classes (DEMO)

8 May 2024, Speaker name

JUnit Test Exception

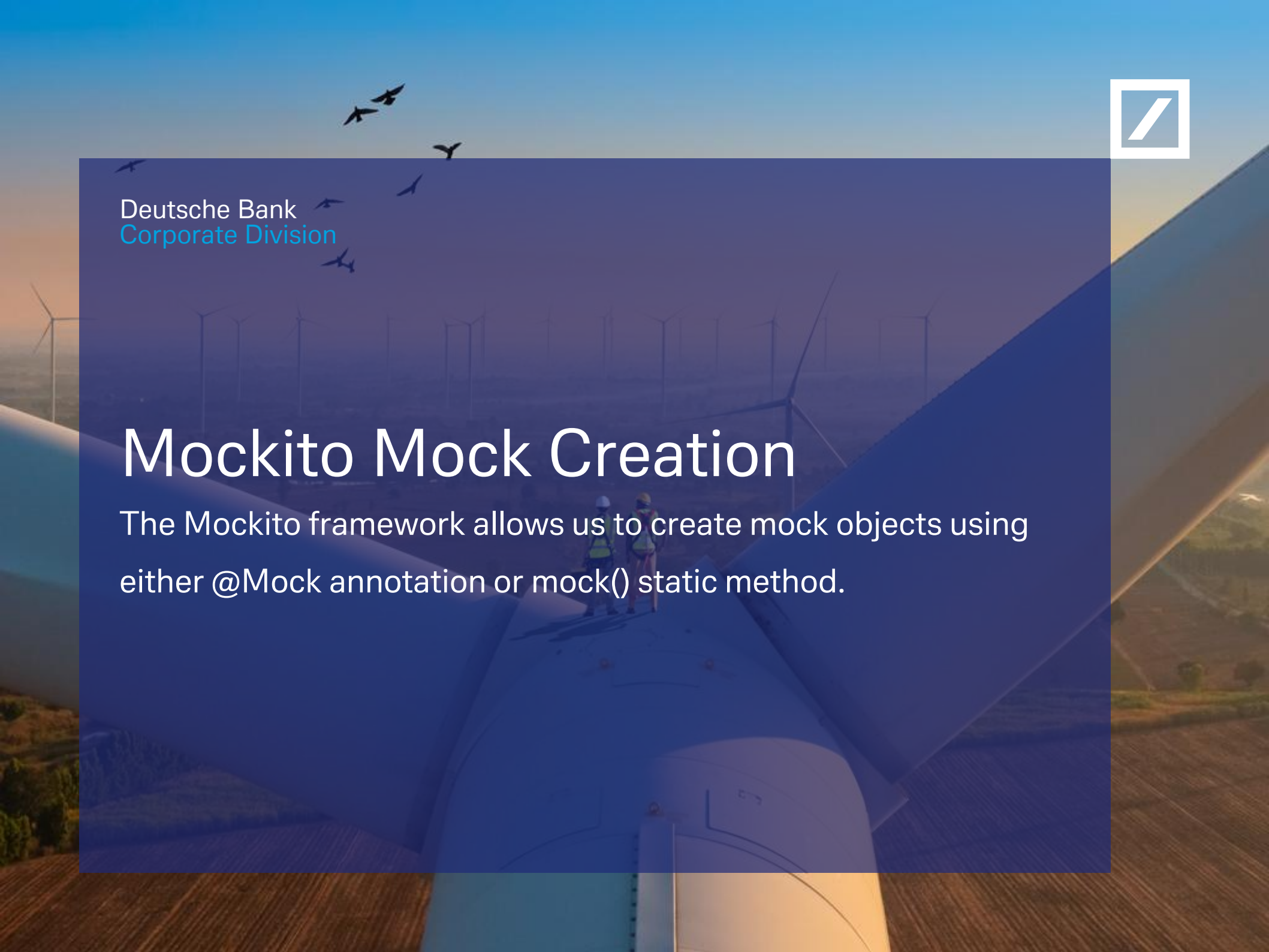
```
Throwable exception = assertThrows(IllegalArgumentException.class, () -> {
     throw new IllegalArgumentException("Illegal Argument Exception
occured");
});
assertEquals("Illegal Argument Exception occured", exception.getMessage());
```

Mockito

- Mockito Maven Dependecies
- Mockito Mock Creation
- Mockito Mock() Annotation
- Mockito Verify Interaction

MOCKITO MAVEN DEPENDENCIES

```
<dependency>
     <groupId>org.mockito</groupId>
     <artifactId>mockito-core</artifactId>
     <version>2.19.0</version>
     <scope>test</scope>
</dependency>
<dependency>
     <groupId>org.mockito</groupId>
     <artifactId>mockito-junit-jupiter</artifactId>
     <version>2.19.0</version>
     <scope>test</scope>
</dependency>
```



Mockito Mock Creation

The Mockito framework allows us to create mock objects using either @Mock annotation or mock() static method.

Deutsche Bank
Corporate Division

Mockito Mock() Method (DEMO)

Deutsche Bank
Corporate Division

Mockito Mock() Annotation (DEMO)

Mockito Verify Interaction

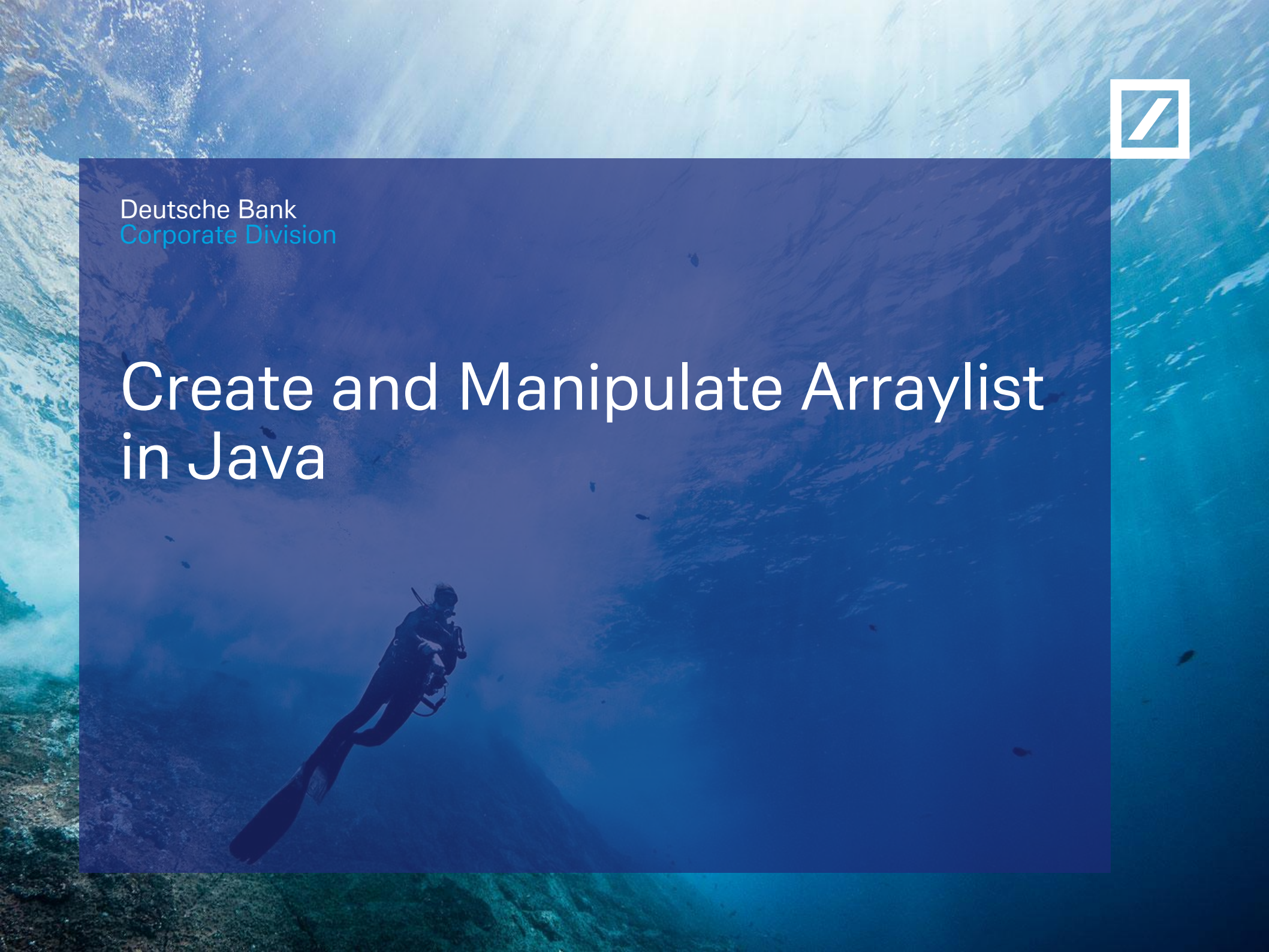
The Mockito framework records all method calls and their parameters made to the mock object. By using Mockito's `verify()` method on the mock object, you can confirm that a method was called with specific parameters. Additionally, you can define the number of invocations, such as an exact number of times, at least a certain number of times, or fewer than a certain number of times, using the `VerificationModeFactory`. The `verify()` method ensures a method was called with the correct parameters, but it does not verify the result of the method call like an assertion would.

Deutsche Bank
Corporate Division

Mockito Verify Interaction (DEMO)

JavaNista


Deutsche Bank
Corporate Division
Community News Jdbc


Deutsche Bank
Corporate Division
Create and Manipulate Arraylist in Java

Deutsche Bank
Corporate Division

Ice Cream Inventory

(Optional)

Deutsche Bank
Corporate Division

Use a Lambda Expression in Java