Deutsche Bank
Corporate Division

Day 2

INTRODUCTION IN JAVA - II

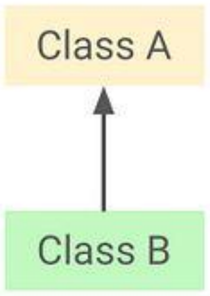
8 May 2024, Speaker name

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.

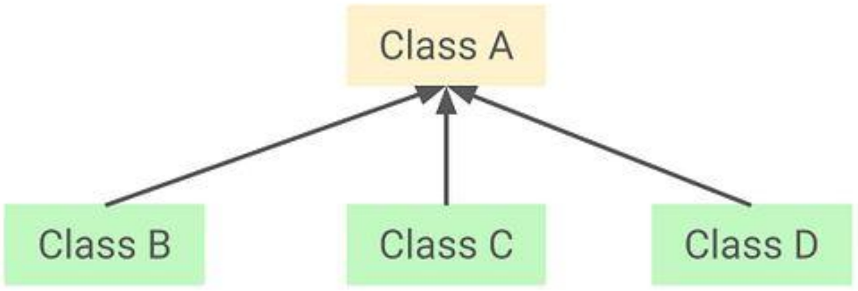
FOUR PILLARS OF OBJECT ORIENTED PROGRAMMING

- **Inheritance** is a mechanism where a new class (child or subclass) inherits the properties and behavior (methods) of an existing class (parent or superclass). It promotes code reusability and establishes a natural hierarchical relationship between classes.
- **Polymorphism** means "many shapes" and allows objects of different classes to be treated as objects of a common superclass. It enables a single interface to be used for a general class of actions, making it easier to manage and scale the code.
- **Abstraction** is the concept of hiding the complex implementation details and showing only the essential features of the object. It simplifies the complexity by providing a simplified model of the system.
- **Encapsulation** is the mechanism of wrapping the data (variables) and the code (methods) that manipulates the data into a single unit called a class. It restricts direct access to some of the object's components, which can prevent the accidental modification of data.

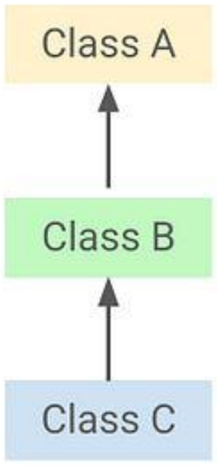
INHERITENCE



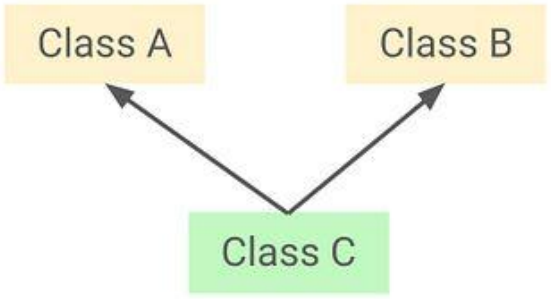
Single Inheritance



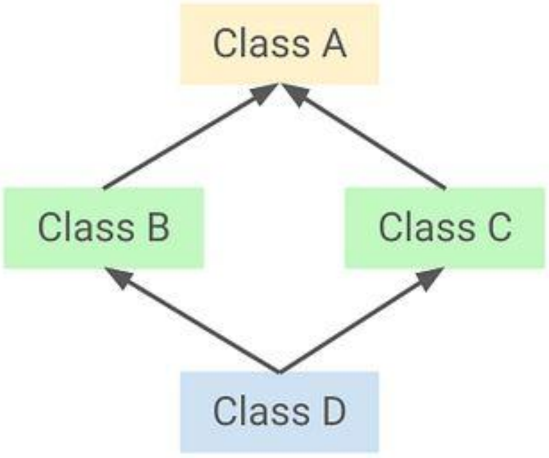
Hierarchical inheritance



Multilevel Inheritance



Multiple Inheritance



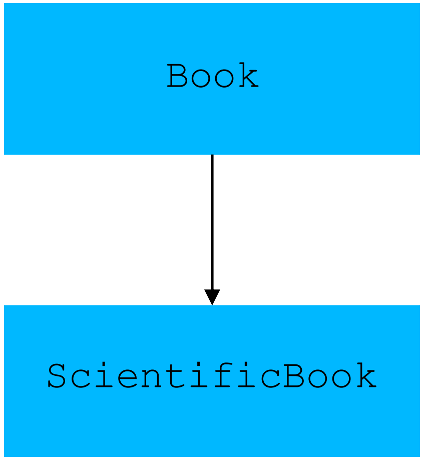
Hybrid Inheritance

INHERITANCE

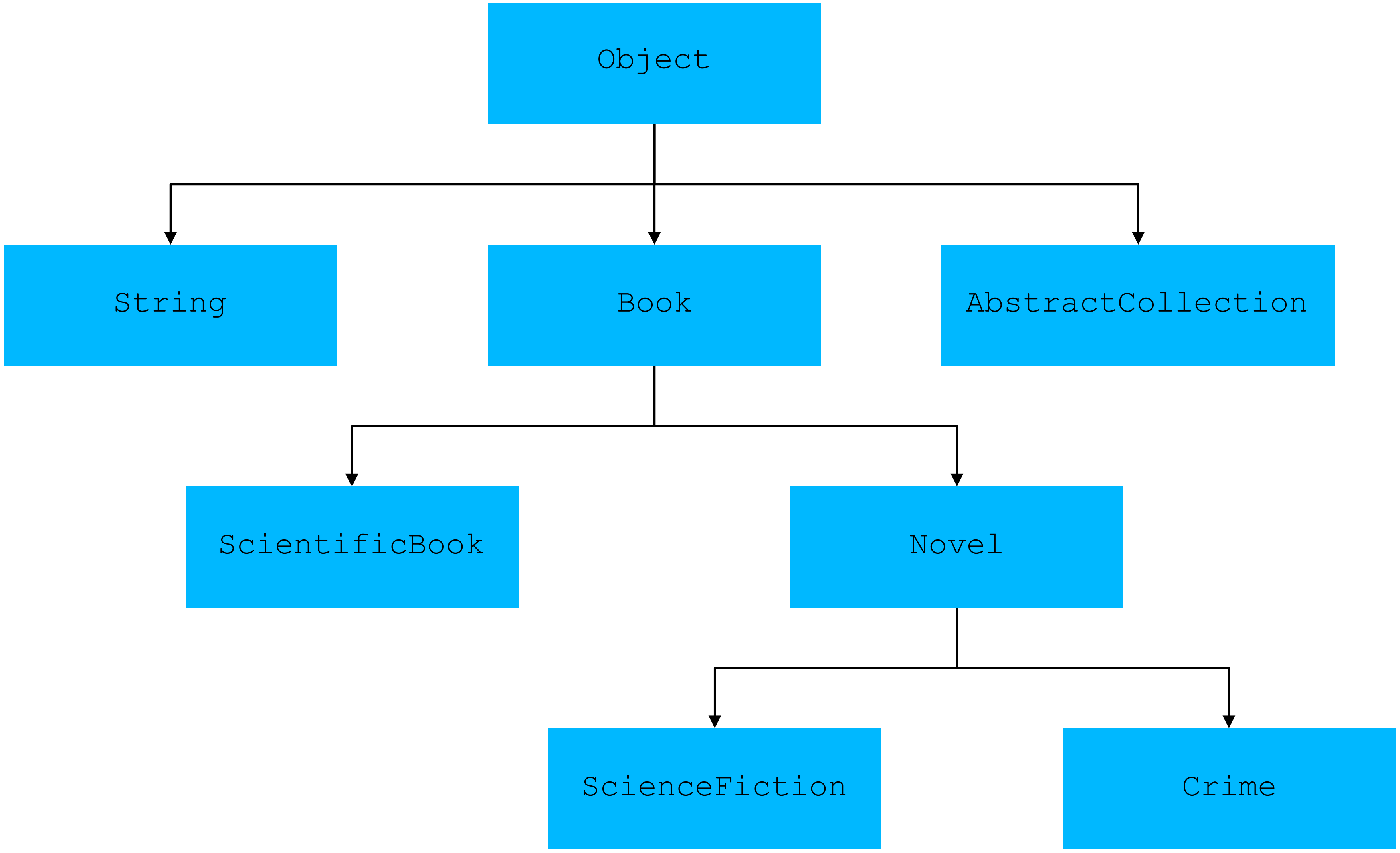
- Inheritance allows to define new classes by reusing other classes, specifying just the differences.

- It is possible to define a new class (subclass) by saying that the class must be like other class (superclass):

```
class ScientificBook extends Book {
  String area;
  boolean proceeding = false;
}
```

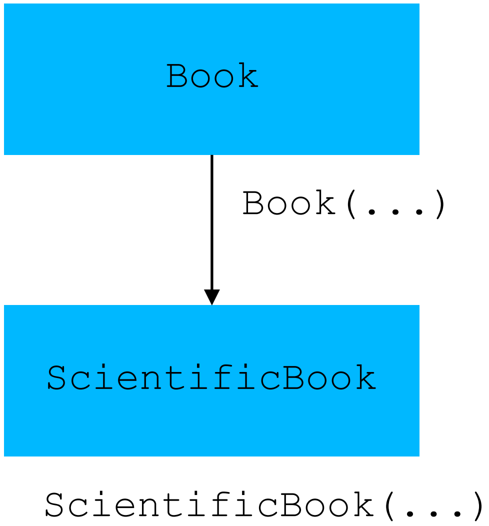


INHERITANCE (HIERARCHY)



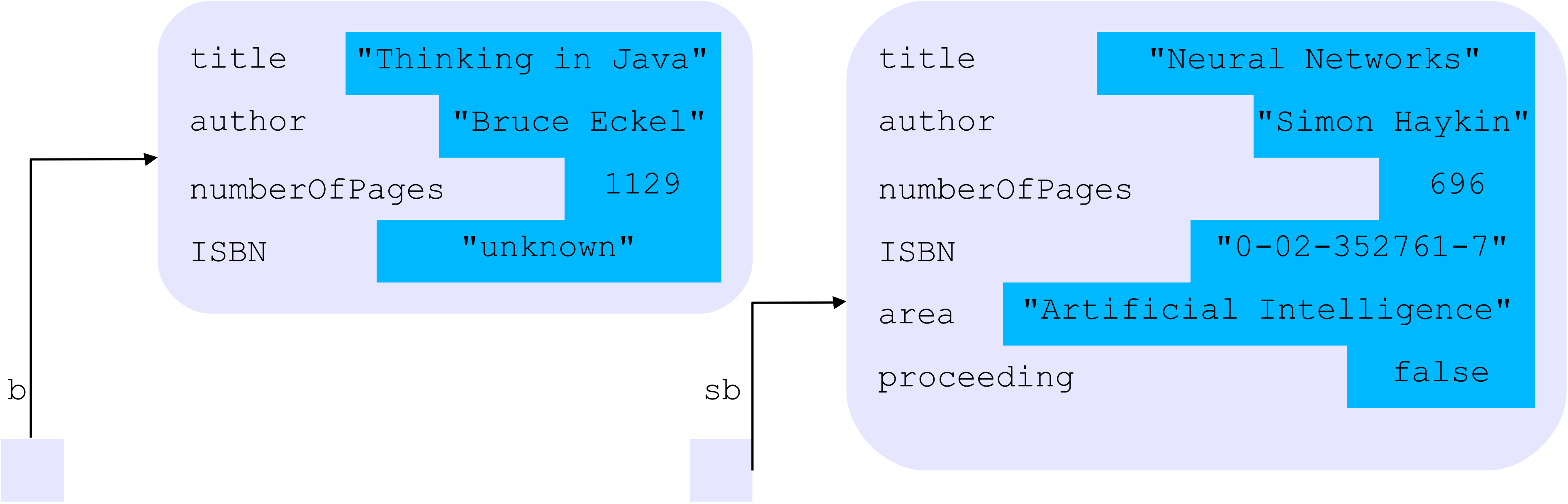
INHERITANCE (CONSTRUCTORS)

```
class ScientificBook extends Book {
  String area;
  boolean proceeding = false;

  ScientificBook(String tit, String aut,
       int num, String isbn, String a) {
    super(tit,aut,num,isbn);
    area = a;
  }
}
```

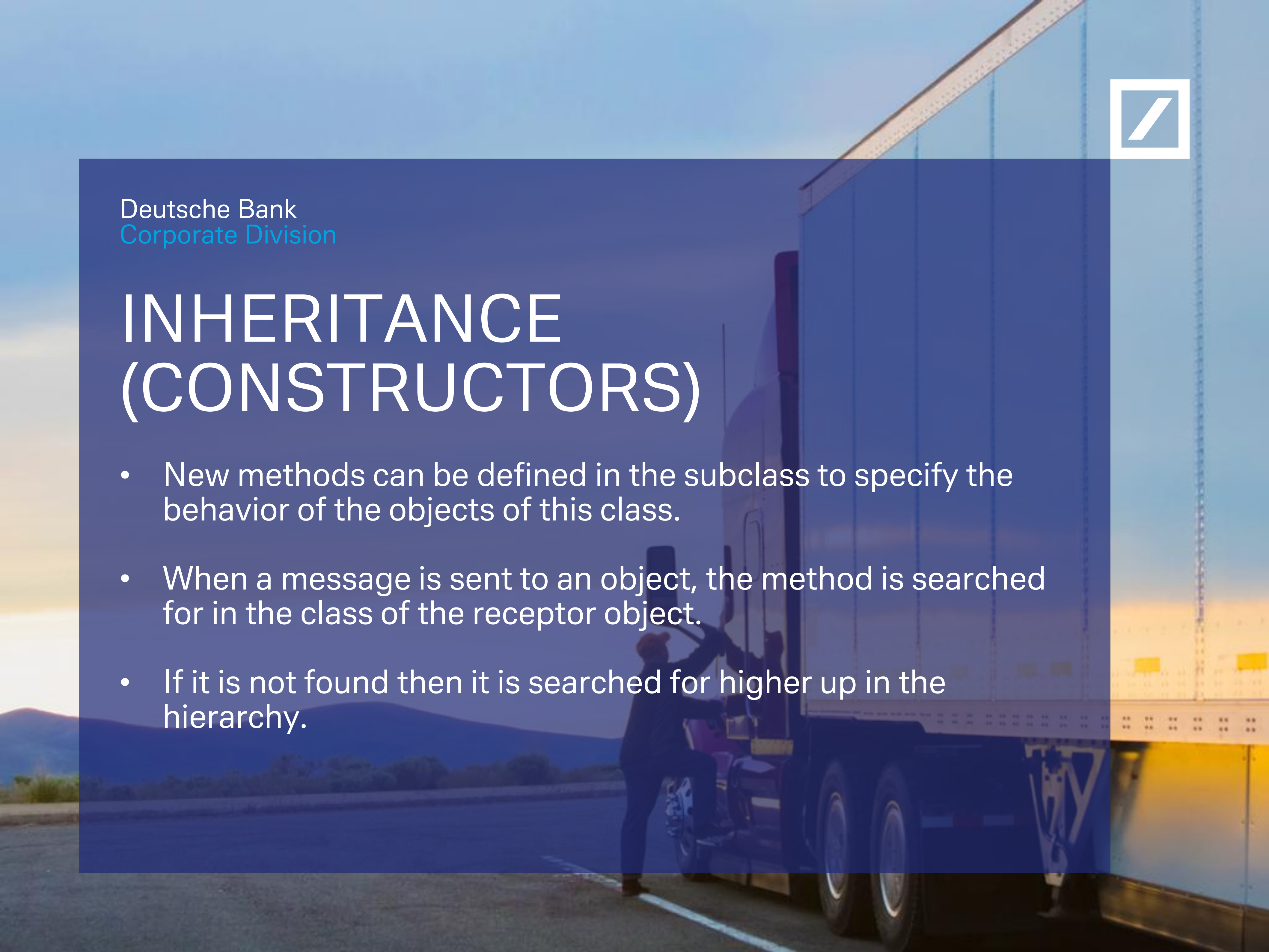


```
ScientificBook sb;

sb = new ScientificBook(
           "Neural Networks",
           "Simon Haykin",696,"0-02-352761-7",
           "Artificial Intelligence");
```

INHERITANCE (CONSTRUCTORS)



```
Book b = new Book("Thinking in Java","Bruce Eckel",1129);

ScientificBook sb = new ScientificBook(
           "Neural Networks",
           "Simon Haykin",696,"0-02-352761-7",
           "Artificial Intelligence");
```



Deutsche Bank
Corporate Division

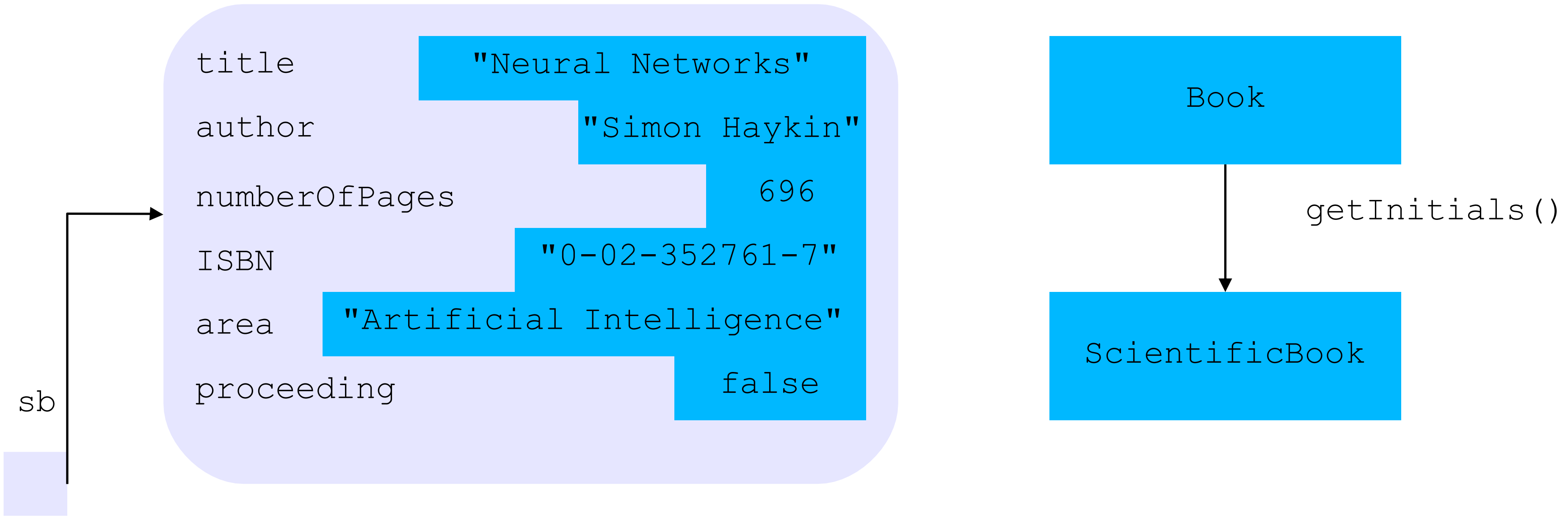
INHERITANCE (CONSTRUCTORS)

- New methods can be defined in the subclass to specify the behavior of the objects of this class.

- When a message is sent to an object, the method is searched for in the class of the receptor object.

- If it is not found then it is searched for higher up in the hierarchy.

INHERITANCE (INHERITING METHODS)



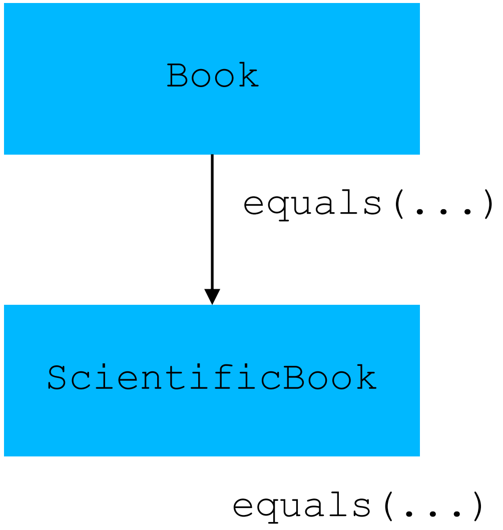
title	"Neural Networks"
author	"Simon Haykin"
numberOfPages	696
ISBN	"0-02-352761-7"
area	"Artificial Intelligence"
proceeding	false

```
ScientificBook sb;

sb = new ScientificBook("Neural Networks","Simon Haykin", 696,
            "0-02-352761-7","Artificial Intelligence");
System.out.println(sb.getInitials());
```

```
S.H.
```

INHERITANCE (OVERRIDING METHODS)

```
   ScientificBook(String tit, String aut,
        int num, String isbn, String a) {
     super(tit,aut,num,isbn);
     area = a;
   }

   public boolean equals(ScientificBook b){
     return super.equals(b) &&
            area.equals(b.area) &&
            proceeding == b.proceeding;
   }
 }
```



INHERITANCE (OVERRIDING METHODS)

Two possible solutions:

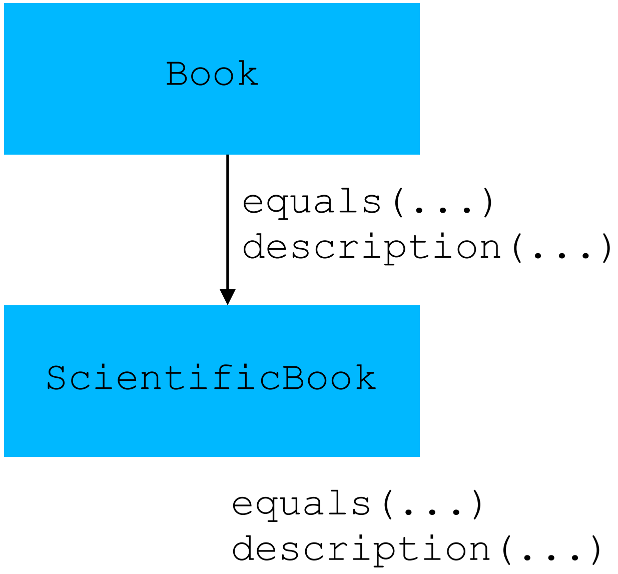
```
public boolean equals(ScientificBook b){
  return super.equals(b) && area.equals(b.area)
         && proceeding == b.proceeding;
}
```

```
public boolean equals(ScientificBook b) {
  return (title.equals(b.title) && author.equals(b.author)
         && numberOfPages == b.numberOfPages
         && ISBN.equals(b.ISBN) && area.equals(b.area)
         && proceeding == b.proceeding;
}
```

Which one is better ?

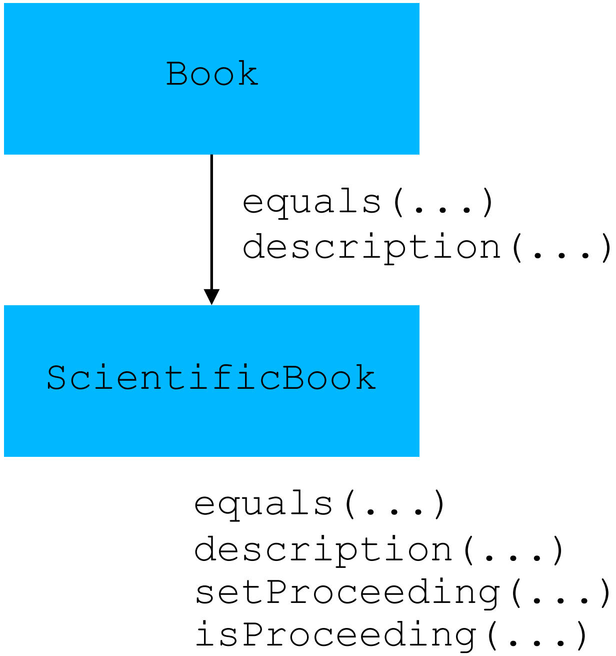
INHERITANCE (OVERRIDING METHODS)

```
class ScientificBook extends Book {
  String area;
  boolean proceeding = false;

  ScientificBook(String tit, String aut,
       int num, String isbn, String a) {
    ...
  }

  public boolean equals(ScientificBook b){
    ...
  }

  public static String description() {
    return "ScientificBook instances can" +
           " store information on " +
           " scientific books";
  }
}
```



INHERITANCE (NEW METHODS)

```java
class ScientificBook extends Book {
  String area;
  boolean proceeding = false;

  ScientificBook(String tit, String aut,
       int num, String isbn, String a) {
    super(tit,aut,num,isbn);
    area = a;
  }
  ...

  public void setProceeding() {
    proceeding = true;
  }

  public boolean isProceeding() {
    return proceeding;
  }
}
```



INHERITANCE (NEW METHODS)

```
class TestScientificBooks {
  public static void main(String[] args) {
    ScientificBook sb1,sb2;

    sb1 = new ScientificBook("Neural Networks","Simon Haykin",
                             696,"0-02-352761-7",
                             "Artificial Intelligence");
    sb2 = new ScientificBook("Neural Networks","Simon Haykin",
                             696,"0-02-352761-7",
                             "Artificial Intelligence");
    sb2.setProceeding();
    System.out.println(sb1.getInitials());
    System.out.println(sb1.equals(sb2));
    System.out.println(sb2.description());
  }
}
```

```
$ java TestScientificBooks
S.H.     false
ScientificBook instances can store information on scientific books
```

getClass()

getClass() returns the runtime class of an object:

```
Book b1 = new Book("Thinking in Java","Bruce Eckel",1129);

System.out.println(b1.getClass().getName());
```

```
Book
```

Instanceof and getClass()

```
class TestClass {
  public static void main(String[] args) {
    Book b1 = new Book("Thinking in Java","Bruce Eckel",1129);
    ScientificBook sb1 = new ScientificBook("Neural Networks",
                            "Simon Haykin",696,"0-02-352761-7",
                            "Artificial Intelligence");

    System.out.println(b1.getClass().getName());
    System.out.println(sb1.getClass().getName());
    System.out.println(b1 instanceof Book);
    System.out.println(sb1 instanceof Book);
    System.out.println(b1 instanceof ScientificBook);
    System.out.println(sb1 instanceof ScientificBook);
  }
}
```

```
$ java TestClass
class Book
class ScientificBook
true true false true
```

POLYMORPHISM

```
// Base class
class Animal {
    public void speak() {
        System.out.println("Animal makes a sound");
    }
}

// Derived classes
class Dog extends Animal {
    @Override
    public void speak() {
        System.out.println("Dog barks");
    }
}

class Cat extends Animal {
    @Override
    public void speak() {
        System.out.println("Cat meows");
    }
}

class Cow extends Animal {
    @Override
    public void speak() {
        System.out.println("Cow moos");
    }
}
```

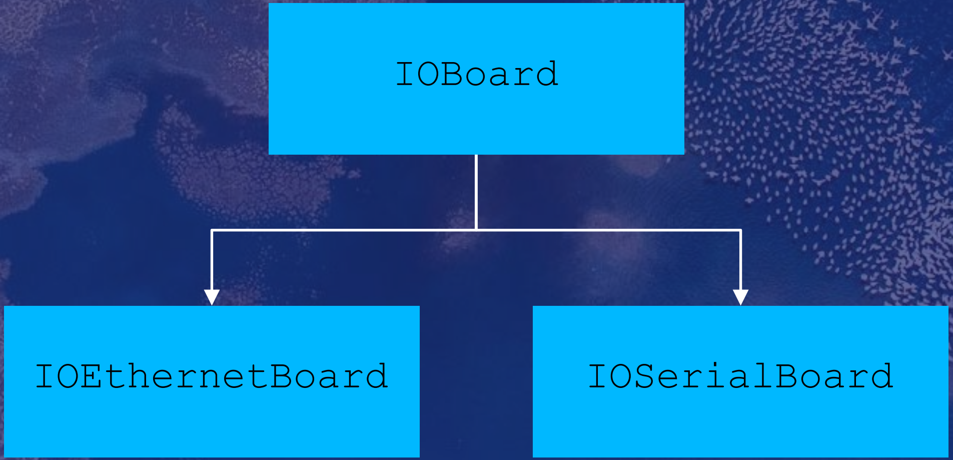
```
public class Main {
public static void animalSound(Animal animal)
{ animal.speak(); }

public static void main(String[] args) {
 Animal dog = new Dog();
Animal cat = new Cat();
Animal cow = new Cow();
 animalSound(dog);

           }
}
```

FINAL AND ABSTRACT

- The modifiers **final** and **abstract** can be applied to classes and methods:
  - **final:**
    - A final class does not allow subclassing.
    - A final method cannot be redefined in a subclass.
  - **abstract:**
    - An abstract class is a class that cannot be instantiated.
    - An abstract method has no body, and it must be redefined in a subclass.

FINAL AND ABSTRACT

An example: the class IOBoard and its subclasses.



IOBoard

IOEthernetBoard

IOSerialBoard

FINAL AND ABSTRACT

```
abstract class IOBoard {
  String name;
  int numErrors = 0;

  IOBoard(String s) {
    System.out.println("IOBoard constructor");
    name = s;
  }
  final public void anotherError() {
    numErrors++;
  }
  final public int getNumErrors() {
    return numErrors;
  }
  abstract public void initialize();
  abstract public void read();
  abstract public void write();
  abstract public void close();
}
```

FINAL AND ABSTRACT

```
class IOSerialBoard extends IOBoard {
  int port;

  IOSerialBoard(String s,int p) {
    super(s); port = p;
    System.out.println("IOSerialBoard constructor");
  }
  public void initialize() {
    System.out.println("initialize method in IOSerialBoard");
  }
  public void read() {
    System.out.println("read method in IOSerialBoard");
  }
  public void write() {
    System.out.println("write method in IOSerialBoard");
  }
  public void close() {
    System.out.println("close method in IOSerialBoard");
  }
}
```

FINAL AND ABSTRACT

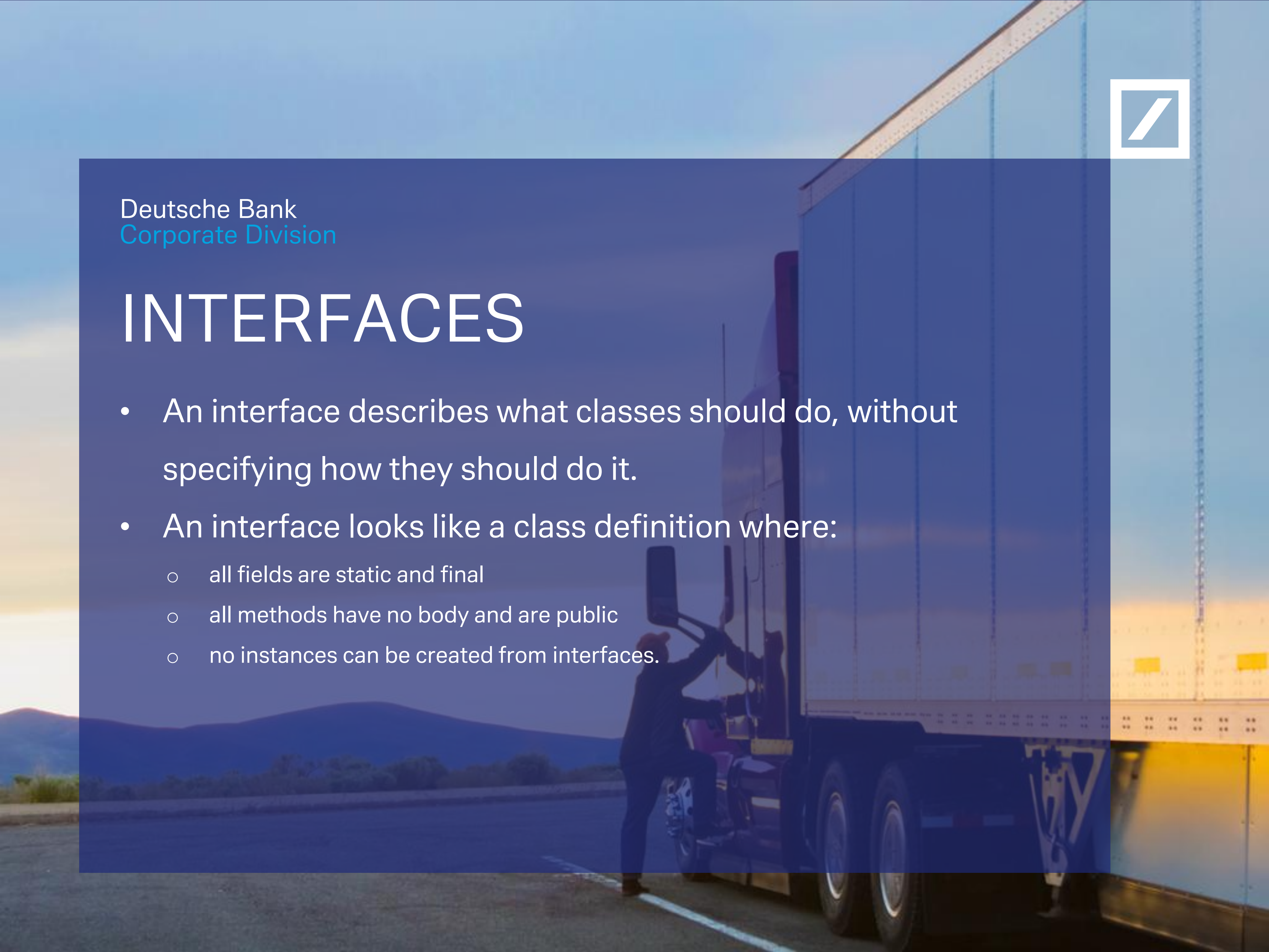
```
class IOEthernetBoard extends IOBoard {
  long networkAddress;

  IOEthernetBoard(String s,long netAdd) {
    super(s); networkAddress = netAdd;
    System.out.println("IOEthernetBoard constructor");
  }
  public void initialize() {
    System.out.println("initialize method in IOEthernetBoard");
  }
  public void read() {
    System.out.println("read method in IOEthernetBoard");
  }
  public void write() {
    System.out.println("write method in IOEthernetBoard");
  }
  public void close() {
    System.out.println("close method in IOEthernetBoard");
  }
}
```

FINAL AND ABSTRACT

Creation of a serial board instance:

```
class TestBoards1 {
  public static void main(String[] args) {
    IOSerialBoard serial = new IOSerialBoard("my first port",
                                              0x2f8);
    serial.initialize();
    serial.read();
    serial.close();
  }
}
```

```
$ java TestBoards1
IOBoard constructor
IOSerialBoard constructor
initialize method in IOSerialBoard
read method in IOSerialBoard
close method in IOSerialBoard
```



Deutsche Bank
Corporate Division

INTERFACES

- An interface describes what classes should do, without specifying how they should do it.
- An interface looks like a class definition where:
  - all fields are static and final
  - all methods have no body and are public
  - no instances can be created from interfaces.

INTERFACES

- An interface for specifying IO boards behavior:

```
interface IOBoardInterface {
  public void initialize();
  public void read();
  public void write();
  public void close();
}
```

- An interface for specifying nice behavior:

```
interface NiceBehavior {
  public String getName();
  public String getGreeting();
  public void sayGoodBye();
}
```

INTERFACES

```
class IOSerialBoard2 implements IOBoardInterface {
  int port;

  IOSerialBoard(String s,int p) {
    super(s); port = p;
    System.out.println("IOSerialBoard constructor");
  }
  public void initialize() {
    System.out.println("initialize method in IOSerialBoard");
  }
  public void read() {
    System.out.println("read method in IOSerialBoard");
  }
  public void write() {
    System.out.println("write method in IOSerialBoard");
  }
  public void close() {
    System.out.println("close method in IOSerialBoard");
  }
}
```

INTERFACES

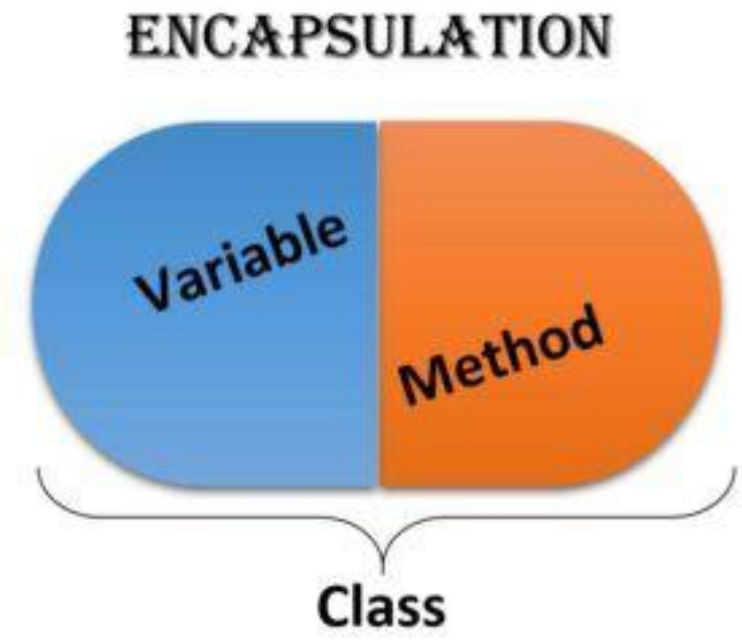
A class can implement more than one interface.

```
class IOSerialBoard2 implements IOBoardInterface,
                                NiceBehavior {

....
}
```

Which methods should it implement ?

ENCAPSULATION

Encapsulation in Java is the principle of bundling the data (attributes) and methods (behaviors) that operate on the data into a single unit called a class. It helps in hiding the internal state of an object from outside interference and manipulation.



ENCAPSULATION

```
// Class with encapsulated data and methods
class Person {
    private String name;
    private int age;
    // Constructor
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    // Getter for name    public String getName() {
     return name;
    }
    // Setter for name    public void
     setName(String name) {
        this.name = name;
    }
    // Getter for age    public int getAge() {
        return age;
    }
```

```
    // Setter for name
    public void setName(String name) {
        this.name = name;
    }
    // Getter for age
    public int getAge() {
        return age;
    }
    // Setter for age
    public void setAge(int age) {
        this.age = age;
    }
}
// Main class to demonstrate encapsulation
public class Main {
    public static void main(String[] args) {
        // Create an object of Person
        Person person = new
Person("Alice", 30);
```

ENCAPSULATION

```
    // Access and modify attributes using
     getter and setter methods
    System.out.println("Name: " +
     person.getName() + ", Age: " +
     person.getAge());
    person.setName("Bob");
    person.setAge(25);
    System.out.println("Updated Name: " +
     person.getName() + ", Updated Age: " +
     person.getAge());
    }
}
```

EXCEPTIONS

The usual behavior on runtime errors is to abort the execution:

```java
class TestExceptions1 {
  public static void main(String[] args) {

    String s = "Hello";
    System.out.print(s.charAt(10));
  }
}
```

```
$ java TestExceptions1
Exception in thread "main"
java.lang.StringIndexOutOfBoundsException:
String index out of range: 10
at java.lang.String.charAt(String.java:499)
at TestExceptions1.main(TestExceptions1.java:11)
```

EXCEPTIONS

The exception can be trapped:

```
class TestExceptions2 {
  public static void main(String[] args) {

    String s = "Hello";
    try {
      System.out.print(s.charAt(10));
    } catch (Exception e) {
      System.out.println("No such position");
  }
}
```

```
$ java TestExceptions2
No such position
```

EXCEPTIONS

It is possible to specify interest on a particular exception:

```
class TestExceptions3 {
  public static void main(String[] args) {

    String s = "Hello";
    try {
      System.out.print(s.charAt(10));
    } catch (StringIndexOutOfBoundsException e) {
      System.out.println("No such position");
  }
}
```

```
$ java TestExceptions3
No such position
```

EXCEPTIONS

It is possible to send messages to an exception object:

```
 class TestExceptions4 {
   public static void main(String[] args) {

    String s = "Hello";
    try {
      System.out.print(s.charAt(10));
    } catch (StringIndexOutOfBoundsException e) {
      System.out.println("No such position");
      System.out.println(e.toString());
  }
}
```

```
$ java TestExceptions4
No such position
java.lang.StringIndexOutOfBoundsException:
String index out of range: 10
```

EXCEPTIONS

We can add multiple catch blocks and a finally clause:

```
class MultipleCatch {
  public void printInfo(String sentence) {
    try {
      // get first and last char before the dot
      char first = sentence.charAt(0);
      char last = sentence.charAt(sentence.indexOf(".") - 1);
      String out = String.format("First: %c Last: %c",first,last);
      System.out.println(out);
    } catch (StringIndexOutOfBoundsException e1) {
      System.out.println("Wrong sentence, no dot?");
    } catch (NullPointerException e2) {
      System.out.println("Non valid string");
    } finally {
      System.out.println("done!");
    }
  }
}
```

EXCEPTIONS

```java
class MultipleCatch {
  public void printInfo(String sentence) {
    try {
      // get first and last char before the dot
      char first = sentence.charAt(0);
      char last = sentence.charAt(sentence.indexOf(".") - 1);
      String out = String.format("First: %c Last: %c",first, last);
      System.out.println(out);
    } catch (StringIndexOutOfBoundsException e1) {
      System.out.println("Wrong sentence, no dot?");
    } catch (NullPointerException e2) {
      System.out.println("Non valid string");
    } finally {
      System.out.println("done!");
    }
  }
```

```java
String sentence = "A test sentence.";
MultipleCatch mc = new MultipleCatch();
mc.printInfo(sentence);
```

```
First: A Last: e
done!
```

EXCEPTIONS

```
class MultipleCatch {
  public void printInfo(String sentence) {
    try {
      // get first and last char before the dot
      char first = sentence.charAt(0);
      char last = sentence.charAt(sentence.indexOf(".") - 1);
      String out = String.format("First: %c Last: %c",first, last);
      System.out.println(out);
    } catch (StringIndexOutOfBoundsException e1) {
      System.out.println("Wrong sentence, no dot?");
    } catch (NullPointerException e2) {
      System.out.println("Non valid string");
    } finally {
      System.out.println("done!");
    }
  }
```

```
String sentence = "A test sentence";
MultipleCatch mc = new MultipleCatch();
mc.printInfo(sentence);
```

```
Wrong sentence, no dot?
done!
```

EXCEPTIONS

```
class MultipleCatch {
  public void printInfo(String sentence) {
    try {
      // get first and last char before the dot
      char first = sentence.charAt(0);
      char last = sentence.charAt(sentence.indexOf(".") - 1);
      String out = String.format("First: %c Last: %c",first, last);
      System.out.println(out);
    } catch (StringIndexOutOfBoundsException e1) {
      System.out.println("Wrong sentence, no dot?");
    } catch (NullPointerException e2) {
      System.out.println("Non valid string");
    } finally {
      System.out.println("done!");
    }
  }
```

```
String sentence = null;
MultipleCatch mc = new MultipleCatch();
mc.printInfo(sentence);
```

```
Non valid string
done!
```

EXCEPTIONS

- There exists a set of predefined exceptions that can be caught.
- In some cases it is compulsory to catch exceptions.
- It is also possible to express the interest to not to catch even compulsory exceptions.

Deutsche Bank
Corporate Division

STRING OPERATORS

Java provides many operators for Strings:

- Concatenation (+)
- many more…

IMPORTANT:

- If the expression begins with a string and uses the + operator, then the next argument is converted to a string.
- Strings cannot be compared with == and !=.

Use this light-coloured cover template to save on toner when printing.

8 May 2024, Speaker name

STRING OPERATORS

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

    String s1 = "Hello" + " World!";
    System.out.println(s1);

    int i = 35,j = 44;
    System.out.println("The value of i is " + i +
                       " and the value of j is " + j);
  }
}
```

```
$ java Strings
Hello World!
The value of i is 35 and the value of j is 44
```

STRING OPERATORS

```
class Strings2 {
  public static void main(String[] args) {

    String s1 = "Hello";
    String s2 = "Hello";

    System.out.println(s1.equals(s2));
    System.out.println(s1.equals("Hi"));
  }
}
```

```
$ java Strings2
true
false
```

CASTING

Java performs a automatic type conversion in the values when there is no risk for data to be lost.

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

    int a = ’x’;         // ’x’ is a character
    long b = 34;         // 34 is an int
    float c = 1002;      // 1002 is an int
    double d = 3.45F;    // 3.45F is a float
  }
}
```

CASTING

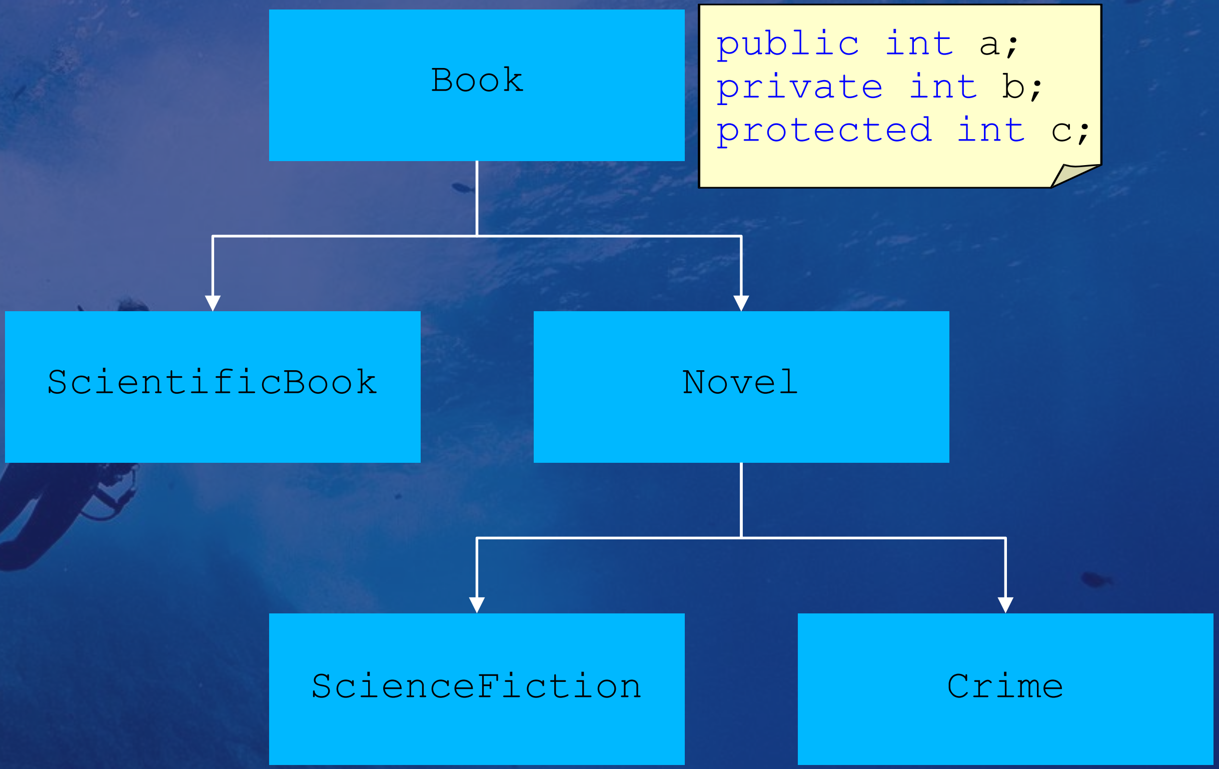
In order to specify conversions where data can be lost it is necessary to use the cast operator.

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

    long a = 34;
    int b = (int)a;          // a is a long
    double d = 3.45;
    float f = (float)d;      // d is a double
  }
}
```

ACCESS CONTROL

It is possible to control the access to methods and variables from other classes with the modifiers:

- public
- private
- protected



```
public int a;
private int b;
protected int c;
```

ACCESS CONTROL

- The default access allows full access from all classes that belong to the same package.
- For example, it is possible to set the proceeding condition of a scientific book in two ways:

```
sb1.setProceeding();
```

- or by just accessing the data member:

```
sb1.proceeding = true;
```

ACCESS CONTROL

- Usually we do not want direct access to a data member in order to guarantee encapsulation:

```
class ScientificBook extends Book {
  private String area;
  private boolean proceeding = false;
  ................
}
```

- Now, the proceeding condition can only be asserted with the message:

```
sb1.setProceeding();        // fine
sb1.proceeding = true;      // wrong
```

ACCESS CONTROL

The same access control can be applied to methods.

```
class ScientificBook extends Book {
  private String area;
  private boolean proceeding = false;
  ...............

  private boolean initialized() {

    return title != null && author != null &&
           numberOfPages != 0 && area != null;
  }
}
```

Where can initialized() be called from ?

PACKAGES

- A package is a structure in which classes can be organized.
- It can contain any number of classes, usually related by purpose or by inheritance.
- If not specified, classes are inserted into the *default*
- package.

PACKAGES

- The standard classes in the system are organized in packages:

```
import java.util.*; // or import java.util.Date

class TestDate {
  public static void main(String[] args) {
    System.out.println(new Date());
  }
}
```

```
$ java TestDate
Wed Oct 25 09:48:54 CEST 2006
```

PACKAGES

Package name is defined by using the keyword package as the first instruction:

```
package myBook;

class Book {
  String title;
  String author;
  int numberOfPages;
}
```

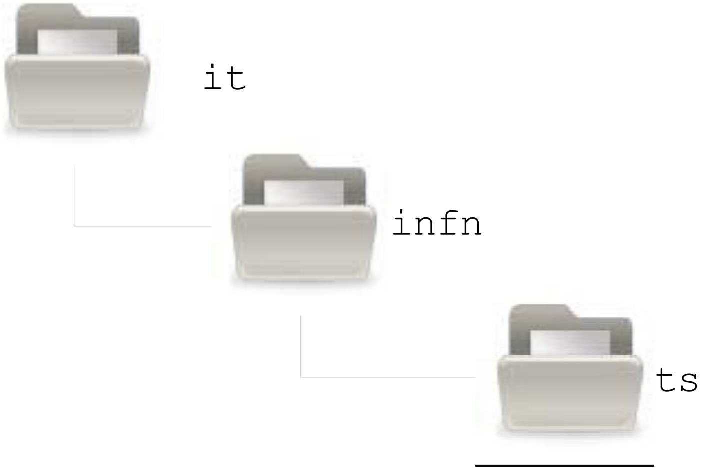
Book.java

ExampleBooks.java

```
package myBook;

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

    Book b = new Book();
    b.title = "Thinking in Java";
    b.author = "Bruce Eckel";
    b.numberOfPages = 1129;
    System.out.println(b.title + " : " +
      b.author + " : " + b.numberOfPages);
  }
}
```

PACKAGES

Files have to be stored in special directories accessible on the class path ($CLASSPATH):



```
package it.infn.ts;

class Book {
    ...
}
```

Example of use:

```
import it.infn.ts.Book;

class TestBook {
    ...
    Book b = new Book(...);
    ...
}
```

Streams and File I/O

- I/O Overview
- Text File I/O
- Text File Output
- Output File Streams + DEMO
- Exception Handling with File I/O

I/O Overview

- I/O = Input/Output
- In this context it is input to and output from programs
- Input can be from keyboard or a file
- Output can be to display (screen) or a file
- Advantages of file I/O
    - permanent copy
    - output from one program can be input to another
    - input can be automated (rather than entered manually)

Text File I/O

- Important classes for text file output (to the file)
  - PrintWriter
  - FileOutputStream [or FileWriter]
- Important classes for text file input (from the file):
  - BufferedReader
  - FileReader
- FileOutputStream and FileReader take file names as arguments.
- PrintWriter and BufferedReader provide useful methods for easier writing and reading.
- Usually need a combination of two classes
- To use these classes your program needs a line like the following:

  import java.io.*;

Every File Has Two Names

1. the stream name used by Java

   `outputStream` in the example

2. the name used by the operating system

   `out.txt` in the example

Text File Output

- To open a text file for output: connect a text file to a stream for writing

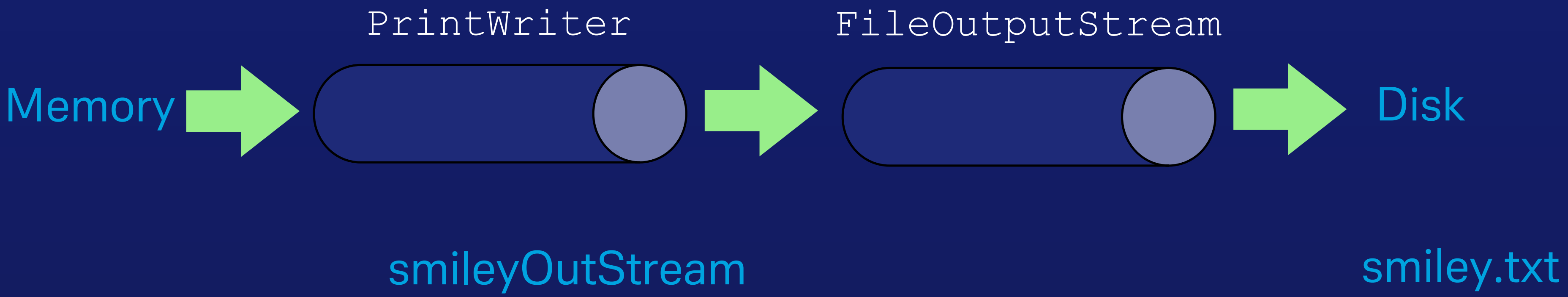
```
PrintWriter outputStream =
new PrintWriter(new FileOutputStream("out.txt"));
```

- Similar to the long way:

```
FileOutputStream s = new FileOutputStream("out.txt");
PrintWriter outputStream = new PrintWriter(s);
```

- Goal: create a `PrintWriter` object
  - which uses `FileOutputStream` to open a text file
- `FileOutputStream` “connects” `PrintWriter` to a text file.

Output File Streams



```
PrintWriter smileyOutStream = new PrintWriter( new FileOutputStream(“smiley.txt”)
```

Text File Output Demo - Part 1

```
public static void main(String[] args)
{
   PrintWriter outputStream = null;
   try
   {
       outputStream =
            new PrintWriter(new FileOutputStream("out.txt"));
   }
   catch(FileNotFoundException e)
   {
       System.out.println("Error opening the file out.txt. “
                              + e.getMessage());
       System.exit(0);
   }
```

A try-block is a block: outputStream would not be accessible to the rest of the method if it were declared inside the try-block

Opening the file

Creating a file can cause the FileNotFound-Exception if the new file cannot be made.

Text File Output Demo - Part 1

```
System.out.println("Enter three lines of text:");
String line = null;
int count;
   for (count = 1; count <= 3; count++)
   {
       line = keyboard.nextLine();
       outputStream.println(count + " " + line);
   }
   outputStream.close();
   System.out.println("... written to out.txt.");
}
```

Writing to the file

Closing the file

The `println` method is used with two different streams: `outputStream` and `System.out`

Exception Handling with File I/O

- Catching IOExceptions
- `IOException` is a predefined class
- File I/O might throw an `IOException`
- catch the exception in a catch block that at least prints an error message and ends the program
- `FileNotFoundException` is derived from `IOException`
    - therefor any catch block that catches `IOException`s also catches `FileNotFoundException`s
    - put the more specific one first (the derived one) so it catches specifically file-not-found exceptions
    - then you will know that an I/O error is something other than file-not-found

Example: Reading a File Name from the Keyboard

```
public static void main(String[] args)
  {
    String fileName = null;  // outside try block, can be used in catch
    try
    { Scanner keyboard = new Scanner(System.in);
      System.out.println("Enter file name:");
      fileName = keyboard.next();
      BufferedReader inputStream =
        new BufferedReader(new FileReader(fileName));
      String line = null;
      line = inputStream.readLine();
      System.out.println("The first line in " + filename + " is:");
      System.out.println(line);
      // . . . code for reading second line not shown here . . .
      inputStream.close();
    }
    catch(FileNotFoundException e)
    {
      System.out.println("File " + filename + " not found.");
    }
    catch(IOException e)
    {
      System.out.println("Error reading from file " + fileName);
    }
  }
```

reading a file name from the keyboard

using the file name read from the keyboard

reading data from the file

closing the file

Exception.getMessage()

```
try
{
   …
}
catch (FileNotFoundException e)
{
   System.out.println(filename + “ not found”);
   System.out.println(“Exception: “ +
                       e.getMessage());
   System.exit(-1);
}
```

Solid Properties

- Single Responsibility Principle
- Open-Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

SOLID Principles

Principle	Description
Single Responsibility Principle	Each class should be responsible for a single part or functionality of the system
Open-Closed Principle	Software components should be open for extension, but not for modification.
Liskov Substitution Principle	Objects of a superclass should be replaceable with objects of its subclasses without breaking the system.
Interface Segregation Principle	No client should be forced to depend on methods that it does not use.
Dependency Inversion Principle	High-level modules should not depend on low-level modules, both should depend on abstractions.

Single Responsibility Principle

```
public class Vehicle {
    public void printDetails() {}
    public double calculateValue() {}
    public void addVehicleToDB() {}
}
```

- The Vehicle class has three separate responsibilities: reporting, calculation, and database.
- By applying SRP, we can separate the above class into three classes with separate responsibilities.

Single Responsibility Principle

```
public class Vehicle {
    private String make;
    private String model;

    // Constructor
    public Vehicle(String make, String model) {
        this.make = make;
        this.model = model;
    }

    // Getters
    public String getMake() {
        return make;
    }

    public String getModel() {
        return model;
    }
```

Single Responsibility Principle

```
// Print vehicle details
    public void printDetails() {
        System.out.println("Make: " + make);
        System.out.println("Model: " + model);
    }

    public static void main(String[] args) {
        // Create a sample vehicle
        Vehicle firstCar = new Vehicle("Toyota", "Camry");

        // Print details
        firstCar.printDetails();
    }
}
```

Open-closed Principle

```
public class VehicleCalculations {
    public double calculateValue(Vehicle v) {
        if (v instanceof Car) {
            return v.getValue() * 0.8;
        if (v instanceof Bike) {
            return v.getValue() * 0.5;

    }
}
```

- Suppose we now want to add another subclass called Truck. We would have to modify the above class by adding another if statement, which goes against the Open-Closed Principle.

- A better approach would be for the subclasses Car and Truck to override the calculateValue method:

Open-closed Principle

```
// Make the Vehicle class as parent for Car and Truck subclasses
class Vehicle {
    private double value;

    public Vehicle(double value) {
        this.value = value;
    }

    public double getValue() {
        return value;
    }

    // Calculate vehicle value (base implementation)
    public double calculateValue() {
        return value; // No depreciation by default
    }
}
```

Open-closed Principle

```
// Subclass Car
class Car extends Vehicle {
    public Car(double value) {
        super(value);
    }

    // Override calculateValue for cars (80% depreciation)
    @Override
    public double calculateValue() {
        return super.calculateValue() * 0.8; // Apply 80%
depreciation
    }
}
// Subclass Truck
class Truck extends Vehicle {
    public Truck(double value) {
        super(value);
    }
```

Open-closed Principle

```
// Override calculateValue for trucks (90% depreciation)
    @Override
    public double calculateValue() {
        return super.calculateValue() * 0.9; // Apply 90%
depreciation
    }
}

public class Main {
    public static void main(String[] args) {
        Car myCar = new Car(25000.0);
        Truck myTruck = new Truck(35000.0);

        System.out.println("Car Value: $" +
myCar.calculateValue());
        System.out.println("Truck Value: $" +
myTruck.calculateValue());
    }
}
```

Liskov Substitution Principle

```
public class Rectangle {
    private double height;
    private double width;
    public void setHeight(double h) { height = h; }
    public void setWidht(double w) { width = w; }
    ...
}
public class Square extends Rectangle {
    public void setHeight(double h) {
        super.setHeight(h);
        super.setWidth(h);
    }
    public void setWidth(double w) {
        super.setHeight(w);
        super.setWidth(w);
    }
}
```

Liskov Substitution Principle

```
abstract class Shape {
    public abstract double getArea();
}

class Rectangle extends Shape {
    private double height;
    private double width;

    public Rectangle(double height, double width) {
        this.height = height;
        this.width = width;
    }

    @Override
    public double getArea() {
        return height * width;
    }
}
```

Liskov Substitution Principle

```
class Square extends Shape {
    private double side;

    public Square(double side) {
        this.side = side;
    }

    @Override
    public double getArea() {
        return side * side;
    }
}
```

Liskov Substitution Principle

```
public class Main {
    public static void main(String[] args) {
        Shape rectangle = new Rectangle(7.0, 5.0);
        Shape square = new Square(3.0);

        System.out.println("Rectangle Area: " +
rectangle.getArea());
        System.out.println("Square Area: " +
square.getArea());
    }
}
```

Interface Segregation Principle

```
public interface Vehicle {
    public void drive();
    public void stop();
    public void refuel();
    public void openDoors();
}
public class Bike implements Vehicle {

    // Can be implemented
    public void drive() {...}
    public void stop() {...}
    public void refuel() {...}

    // Can not be implemented
    public void openDoors() {...}
}
```

Dependency Inversion Principle

```
public class Car {
    private Engine engine;
    public Car(Engine e) {
        engine = e;
    }
    public void start() {
        engine.start();
    }
}
public class Engine {
   public void start() {...}
}
```

```
public interface Engine {
    public void start();
}
```

Dependency Inversion Principle

```
public class Car {
    private Engine engine;
    public Car(Engine e) {
        engine = e;
    }
    public void start() {
        engine.start();
    }
}
public class PetrolEngine implements Engine {
   public void start() {...}
}
public class DieselEngine implements Engine {
   public void start() {...}
}
```

Dependency Inversion Principle

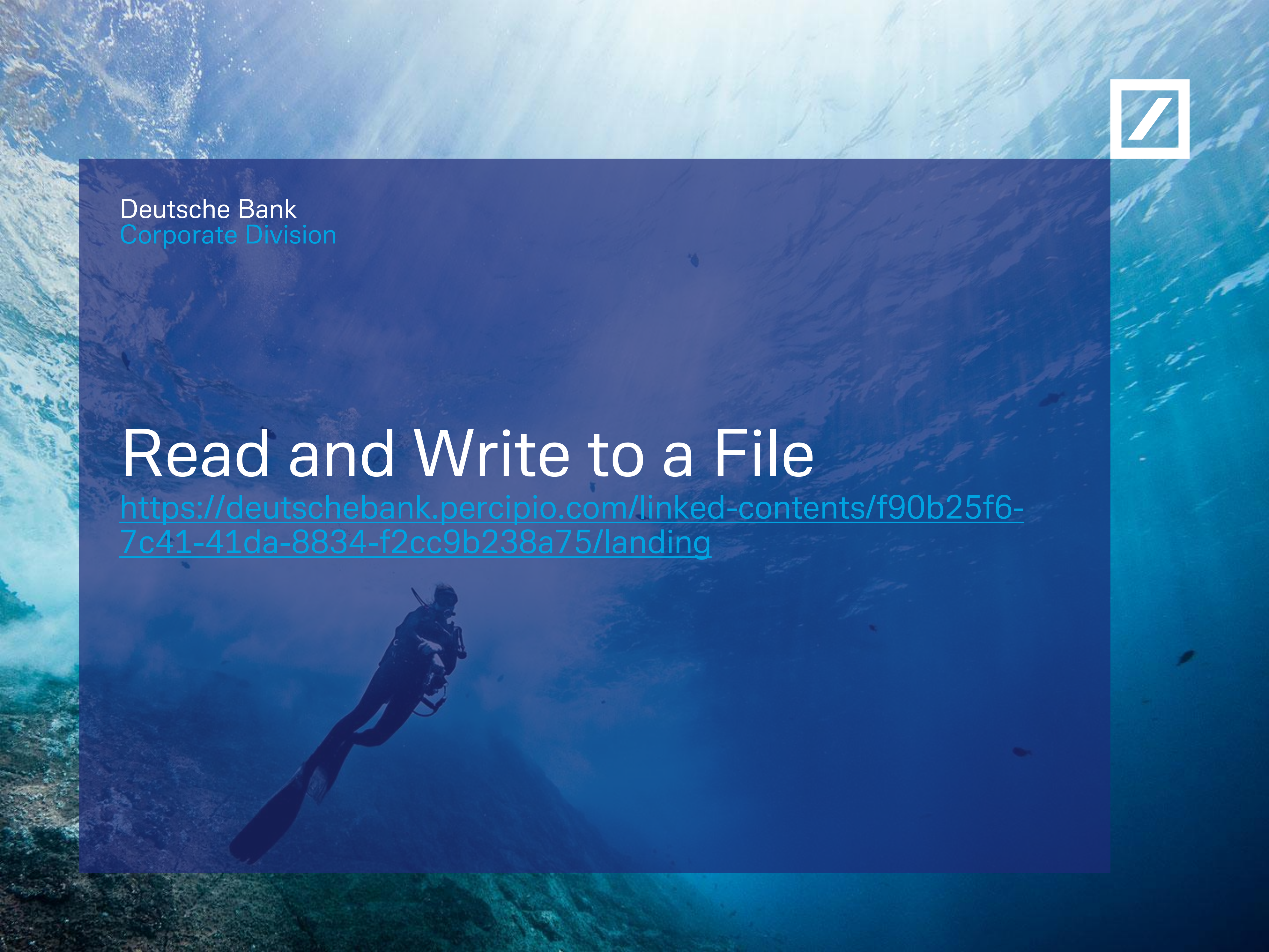
```
// Abstraction for Engine
interface Engine {
    void start();
}

// Class for PetrolEngine
class PetrolEngine implements Engine {
    public void start() {
        System.out.println("Petrol engine started.");
    }
}

// Class for PetrolEngine
class DieselEngine implements Engine {
    public void start() {
        System.out.println("Diesel engine started.");
    }
}
```

Dependency Inversion Principle

```
// Car class depends on Engine abstraction
class Car {
    private Engine engine;

    public Car(Engine e) {
        this.engine = e;
    }

    public void startCar() {
        engine.start();
    }
}
```

Dependency Inversion Principle

```
public class Main {
    public static void main(String[] args) {
        Engine petrolEngine = new PetrolEngine();
        Engine dieselEngine = new DieselEngine();

        Car petrolCar = new Car(petrolEngine);
        Car dieselCar = new Car(dieselEngine);

        petrolCar.startCar();
        dieselCar.startCar();
    }
}
```



Deutsche Bank
Corporate Division

Read and Write to a File

https://deutschebank.percipio.com/linked-contents/f90b25f6-7c41-41da-8834-f2cc9b238a75/landing

Java Novice

Deutsche Bank
Corporate Division

Handle Exception in Java

Deutsche Bank
Corporate Division

Threading in Gym

Deutsche Bank
Corporate Division

Medieval Serialization Lab (Optional)

(Optional)