Java Programming for Testing Automation

Student Guide

27698-54919-01

Table of content

Java Technology

Portability

Security

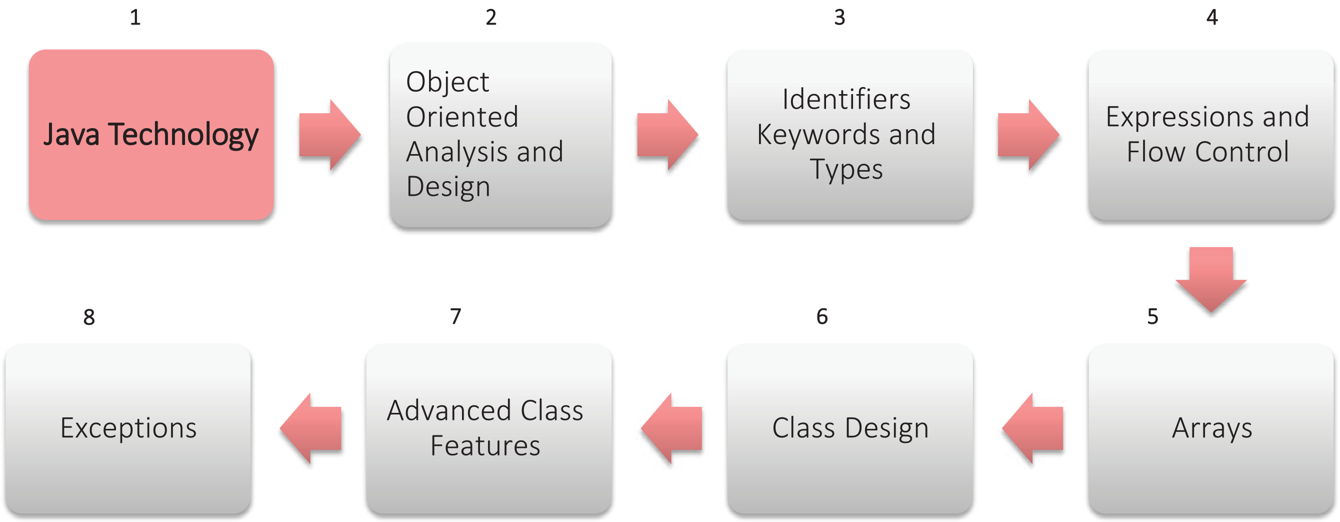
Complete APIs support

JOHN BRYCE
Leading in IT Education
matrix company

John Bryce Training, 29 Homa Umigdal St, Tel Aviv 6777129 // www.johnbryce.com

Objectives

JOHN BRYCE
Leading in IT Education
matrix company



1
Java Technology
2
Object Oriented Analysis and Design
3
Identifiers Keywords and Types
4
Expressions and Flow Control
5
Arrays
6
Class Design
7
Advanced Class Features
8
Exceptions

Objectives

By the end of this session

- You'll get familiar with Java language
- You'll be able to install Java and Eclipse

What Is Java ?

JavaZone 2013: Javapocalypse

- Framework for applications development
- Code portability. Java executables could run on all operating systems
- Rich libraries support
- Increases both security and code integrity by its runtime mechanism

Elements of The Java Platform

- Java programming language
- Software libraries accompanying the system
- Java Virtual Machine (interpreter)
- Compatibility to the Web

The Java Programming Language

- Easy to use, pure object oriented language
- Overwhelming set of rich libraries that makes java a suitable language for a large scale of programming fields
- An exception handling mechanism that improves bugs handling

Java Portability

JOHN BRYCE
Leading in IT Education
matrix company

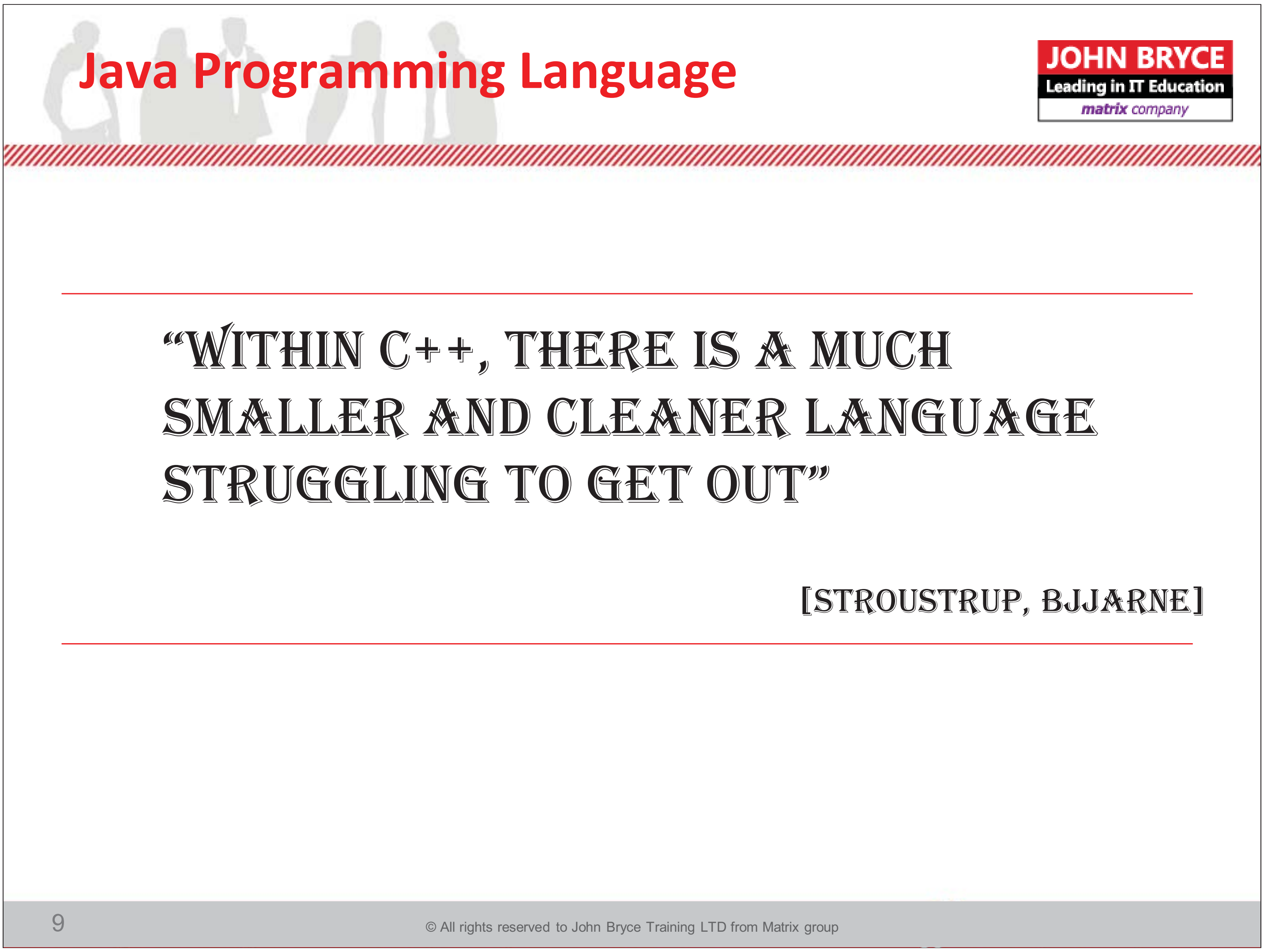
- All java code is cross platforms
- Rich set of libraries classes are identical on all java implementation no matter which platform you use
- A java program is compiled once on any OS and can run anywhere
- The way this portability is achieved is explained later in this module

Java Programming Language

JOHN BRYCE
Leading in IT Education
matrix company

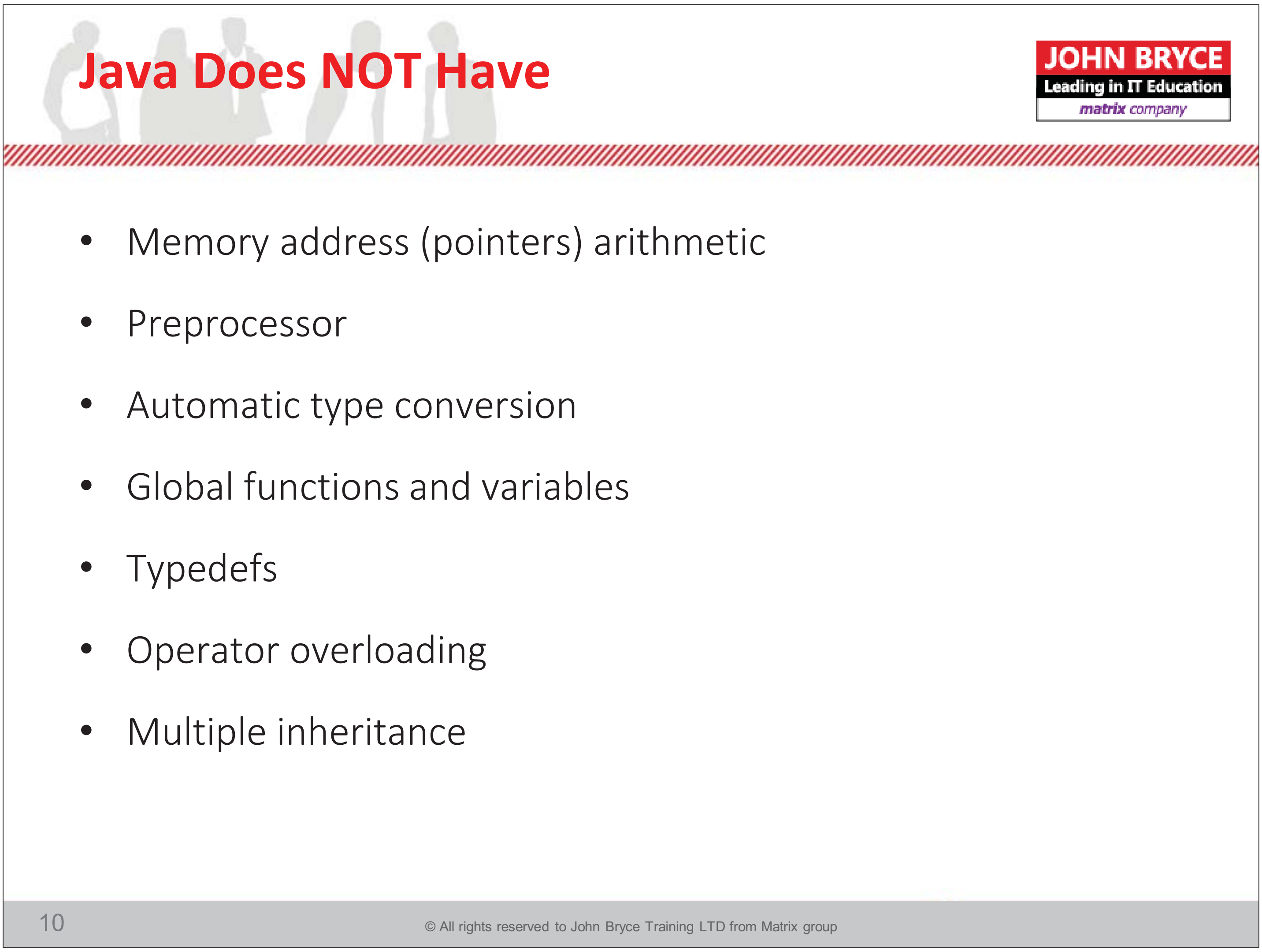
Characteristics

- Syntax - similar to C++
- Strongly typed
- Pure object oriented
- Size of primitives is laid down in the language and is platform independent
- Has garbage collector
- Multi threaded language
- Support exceptions



Java Programming Language

> “WITHIN C++, THERE IS A MUCH SMALLER AND CLEANER LANGUAGE STRUGGLING TO GET OUT”
>
> [STROUSTRUP, BJJARNE]



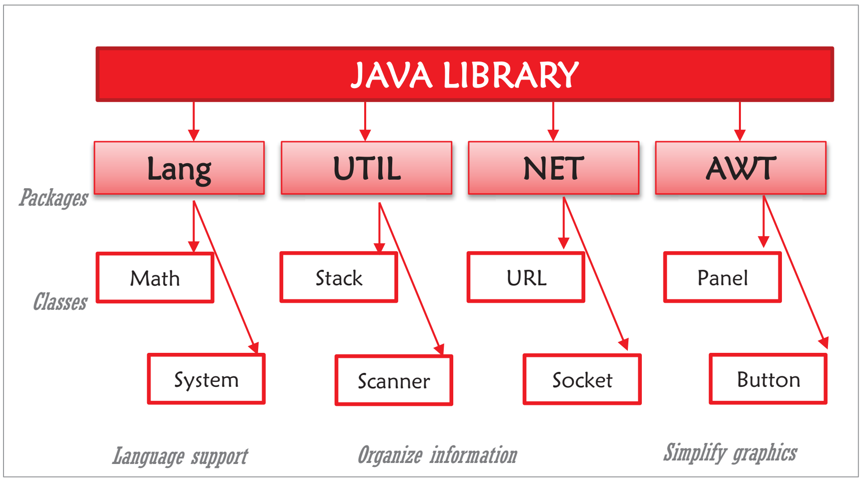
Java Does NOT Have

- Memory address (pointers) arithmetic
- Preprocessor
- Automatic type conversion
- Global functions and variables
- Typedefs
- Operator overloading
- Multiple inheritance

Java Libraries

JOHN BRYCE
Leading in IT Education
matrix company

- Huge amount of cross-platform APIs
- Easy to use APIs
- GUI handling
- Network handling
- Database handling
- I/O handling



JAVA LIBRARY

Packages: Lang, UTIL, NET, AWT

Classes: Math, System, Stack, Scanner, URL, Socket, Panel, Button

Language support

Organize information

Simplify graphics

Java APIs

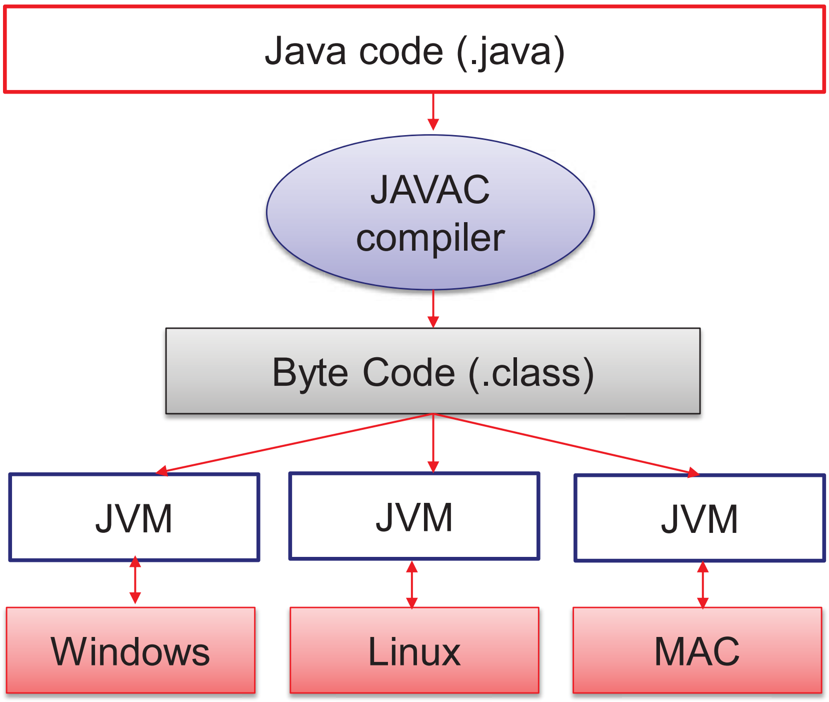
JOHN BRYCE
Leading in IT Education
matrix company

- Java runtime
- JFC
- Security
- JDBC
- JavaBeans
- Java RMI
- Java Communications
- JavaMail
- Java media
- JNDI

Write Once – Run Everywhere

- A java program may be compiled on any computer that has a JVM installed
- A java executable is a binary file that runs on every processor that has a JVM installed
- Recompilation is never required when replacing a platform

Java Virtual Machine



Java class is written in Unicode characters

Java compiler convert these Unicode characters into byte code

Java byte code can only be understandable by JVM

JVM is native code and specific to OS

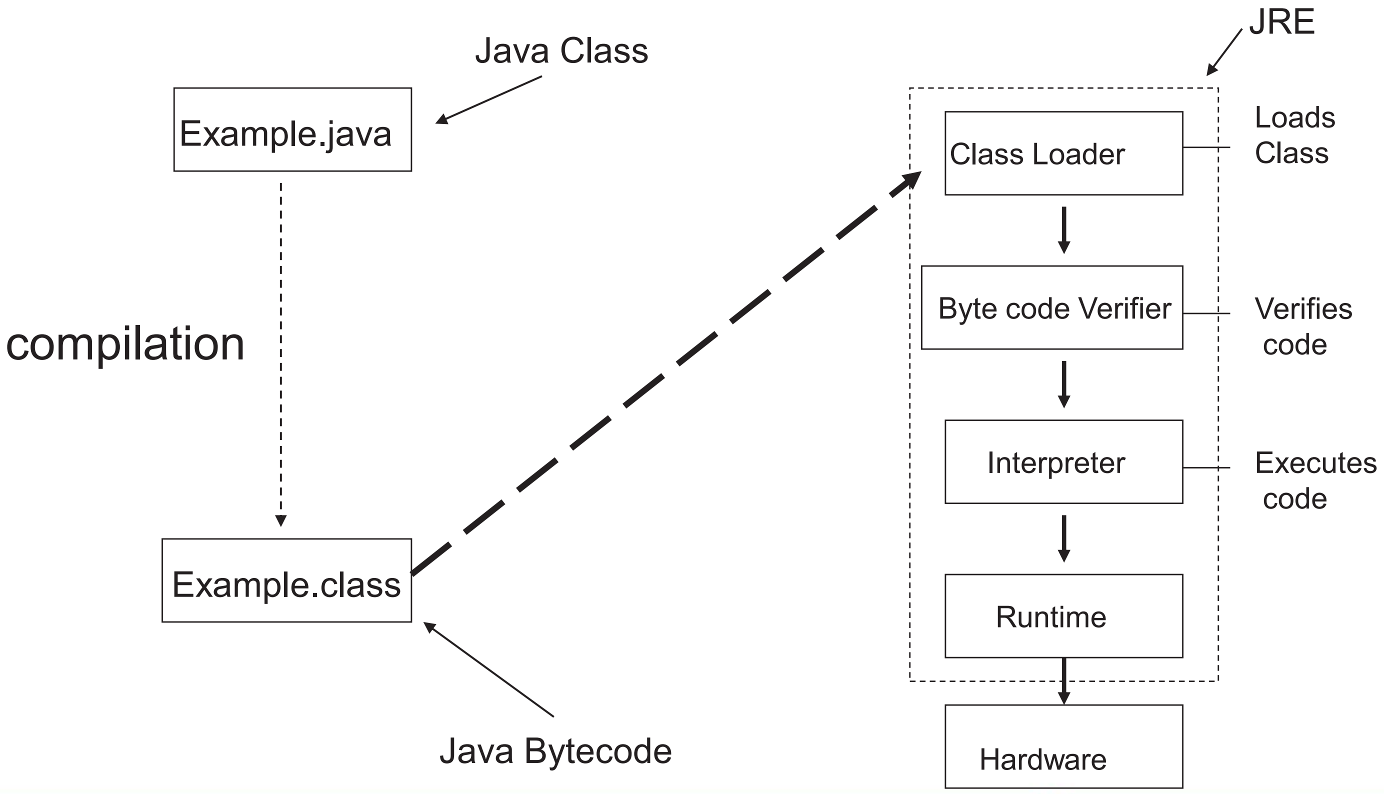
Java Runtime Environment

- Java source files are compiled into bytecode
- The bytecode is stored in a .class file
- At runtime, the bytecode is loaded, verified and ran by the interpreter

Java Runtime Environment

- Is what needed to run Java programs
- Free download from www.java.sun.com
- Can be installed as a browser plug-in
- Contains:
  - JVM
  - Runtime classes (Matching the specific OS)
  - Java Application Launcher

Java Runtime Environment



Java Class

Example.java

compilation

Example.class

Java Bytecode

JRE

Class Loader

Loads Class

Byte code Verifier

Verifies code

Interpreter

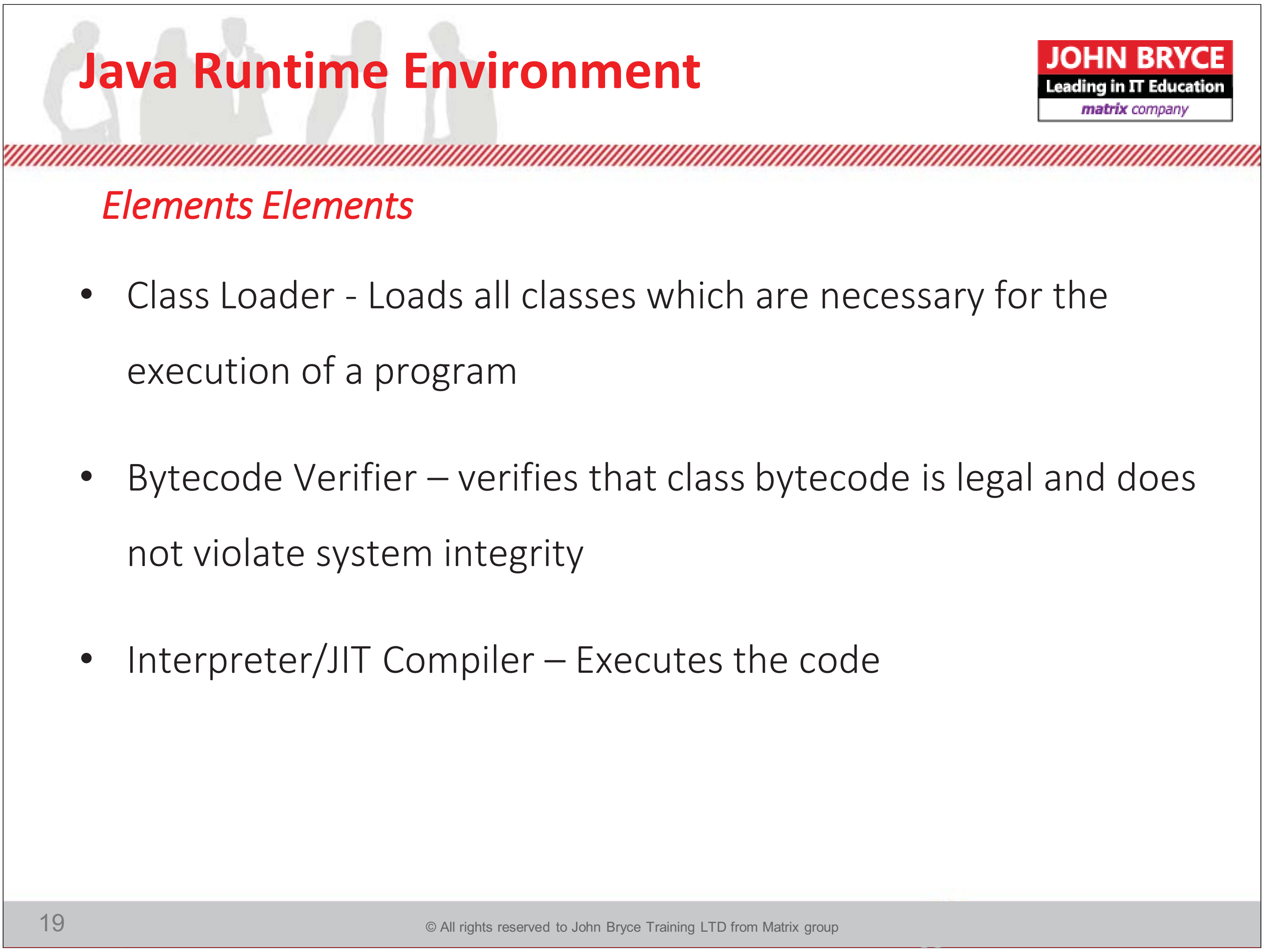
Executes code

Runtime

Hardware

Java Runtime Environment

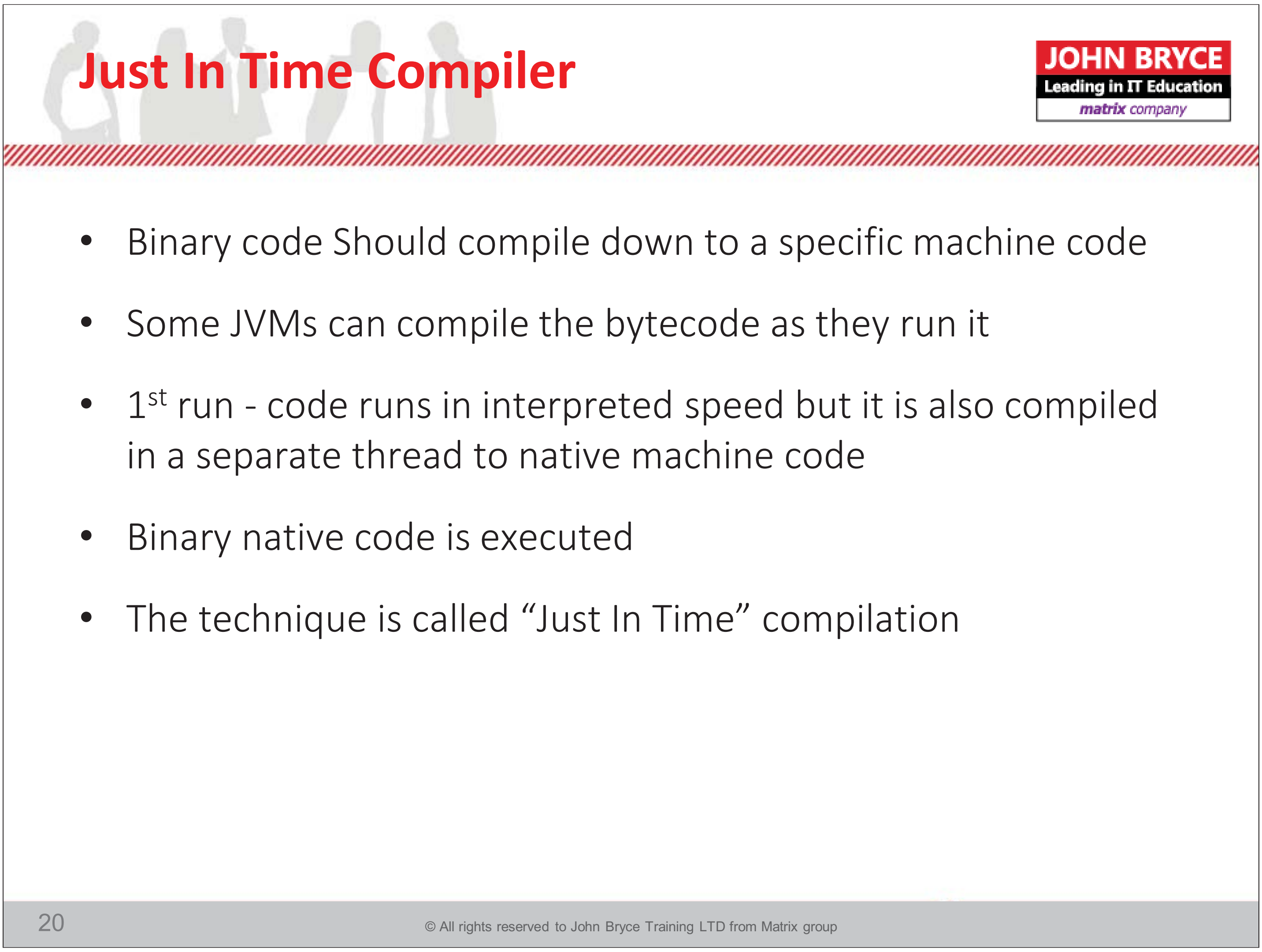
- A Java public class will be written in a file named after the class and has a ".java" extension (e.g. String.java).
- The java compiler (in the JDK, invoked by "javac className.java") compiles the Java source code into bytecode. This will result in a file that has the class name and the extension ".class".
- The Java Runtime Environment loads the class, verifies that its bytecode is compliment to the JVM specifications and then runs the bytecode.



Java Runtime Environment

Elements Elements

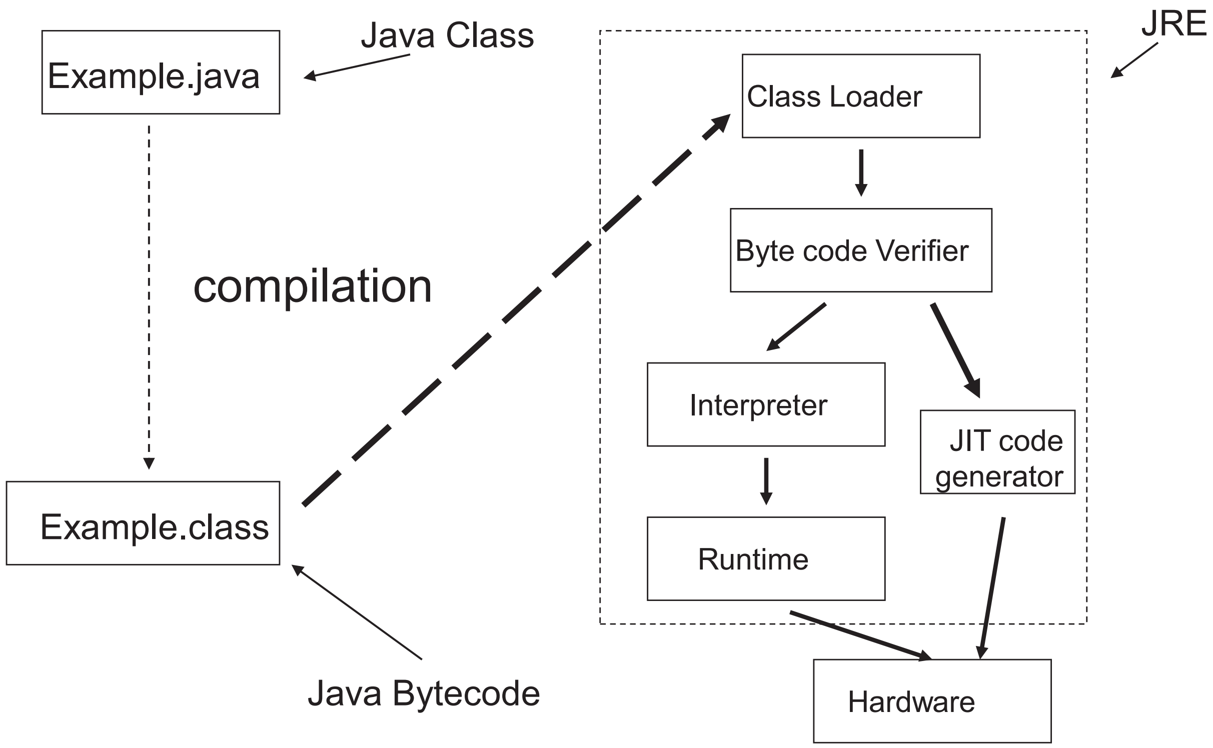
- Class Loader - Loads all classes which are necessary for the execution of a program
- Bytecode Verifier – verifies that class bytecode is legal and does not violate system integrity
- Interpreter/JIT Compiler – Executes the code



Just In Time Compiler

- Binary code Should compile down to a specific machine code
- Some JVMs can compile the bytecode as they run it
- 1st run - code runs in interpreted speed but it is also compiled in a separate thread to native machine code
- Binary native code is executed
- The technique is called “Just In Time” compilation

JIT Compiler



Hot-Spot

Java is a tool that:

- Improves caching of objects & methods in a distributed environments
- Has a fast and fully accurate Garbage Collection
- Offers more GC algorithms (see next slide)
- Has method inlining
- Has compiler that does not use the interpreter – much faster
- Has an advanced logging when handling native crashes
- Has smart thread queue management (all gets about 5% CPU time)
- Has fast thread synchronization
- Produces statistics between program executions to improve performance

Supports new I/O	Better buffering management Scalable network (not one thread per connection) Character-set support

Portability in Java

Portability achieved by the following Java language characteristics:

- Java operands evaluated in left-to-right order
- Primitives have definite size
- Same API on all platforms
- Unicode character set, which is a 16-bit superset of ASCII

Web Compatibility

- Applets - An application program that runs inside a browser
- Servlets - An application program that runs on demand by a web server

Memory Management

- Objects created on the heap using the new keyword
- The heap tends to be fragmented from time to time

Garbage Collection

- The garbage collector frees all dynamically allocated memory that is no longer referenced
- An object located on the heap that is no longer referenced should be regarded as “garbage” and has to be removed
- Garbage collection is actually memory recycling

The Java Development Kit

- The JDK is the software and tools required to compile, debug and run applications written in Java.
- The major releases are:
  - JDK 1.0.2 – May 1996.
  - JDK 1.1 – February 1997.
  - JDK 1.2 – December 1998. Its main contribution is the improved GUI support achieved by introducing the “swing” package and the support for CORBA.
  - Java 2 SDK v. 1.3 – 1.4 – Software Development Kit.
  - Java 2 SDK – v. 5.0 – Tiger – Syntax enhancements
  - Java 2 SDK – v. 6.0 – Mustang – More APIs & utilities
  - Java 2 SDK – v 7.0 – Dolphin
  - Java 2 SDK – v 8.0

The Java Development Kit

- Full kit for developing java applications
- Contains:
  - J2SE
  - JVM
  - Compiler, RMI-Compiler, IDL converter (bin directory)
  - API Documentation generating tool
  - Code examples
  - J2SE code sources
- Is extensible

Environment Variables

- PATH
  - OS variable to determine where to run the script from
  - Should point to the bin directory:

```
path=c:\jdk1.4\bin
```

- CLASSPATH
  - JVM variable used to locate the compiled and executed classes
  - Should point to the same directory or to added classes

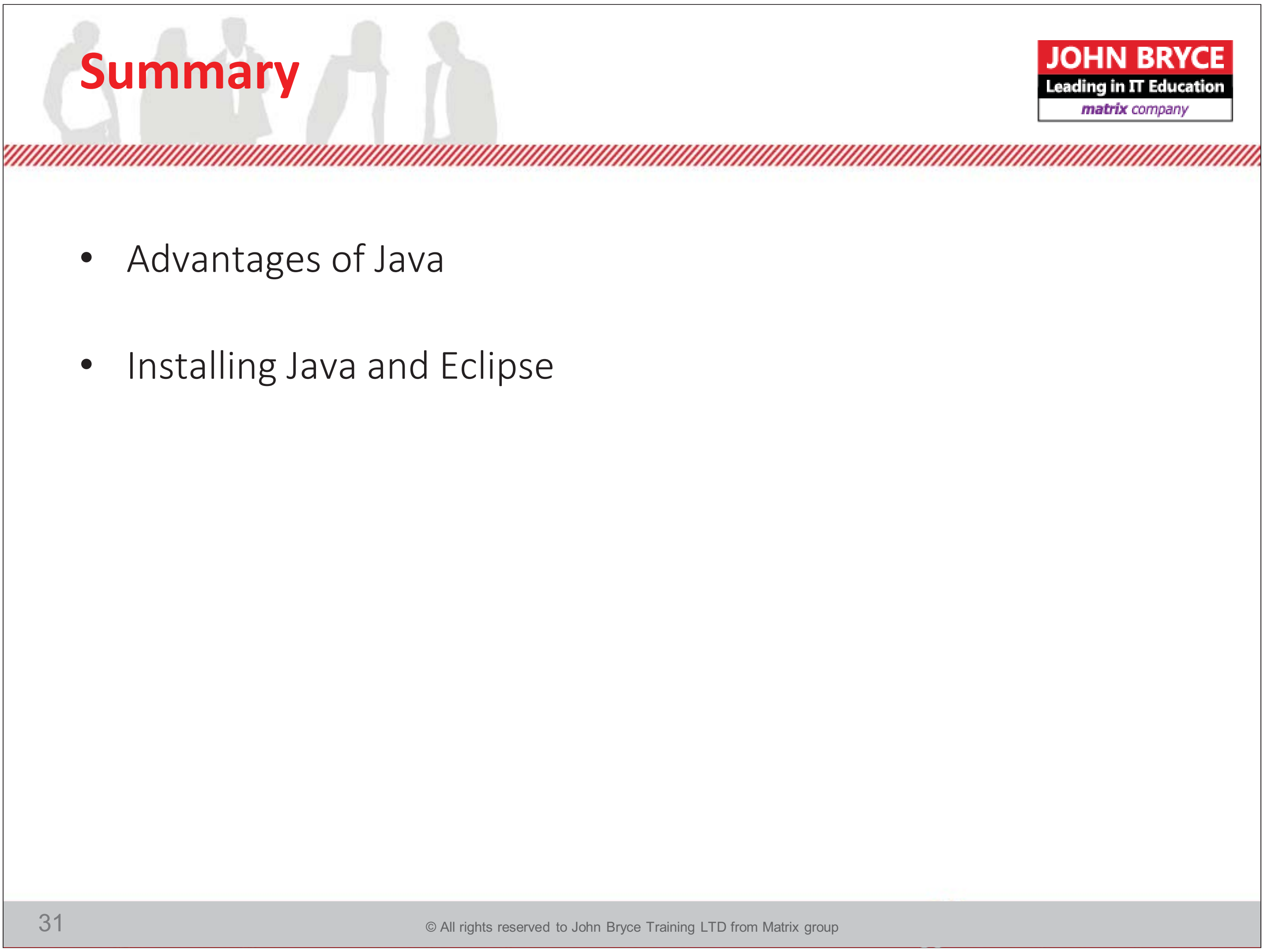
- JAVA_HOME
  - Java Servers variable, specify the Java main directory location
  - Should point to the JDK directory:

```
java_home=c:\jdk1.4
```

Installation

JDK install

Eclipse install



Summary

- Advantages of Java
- Installing Java and Eclipse

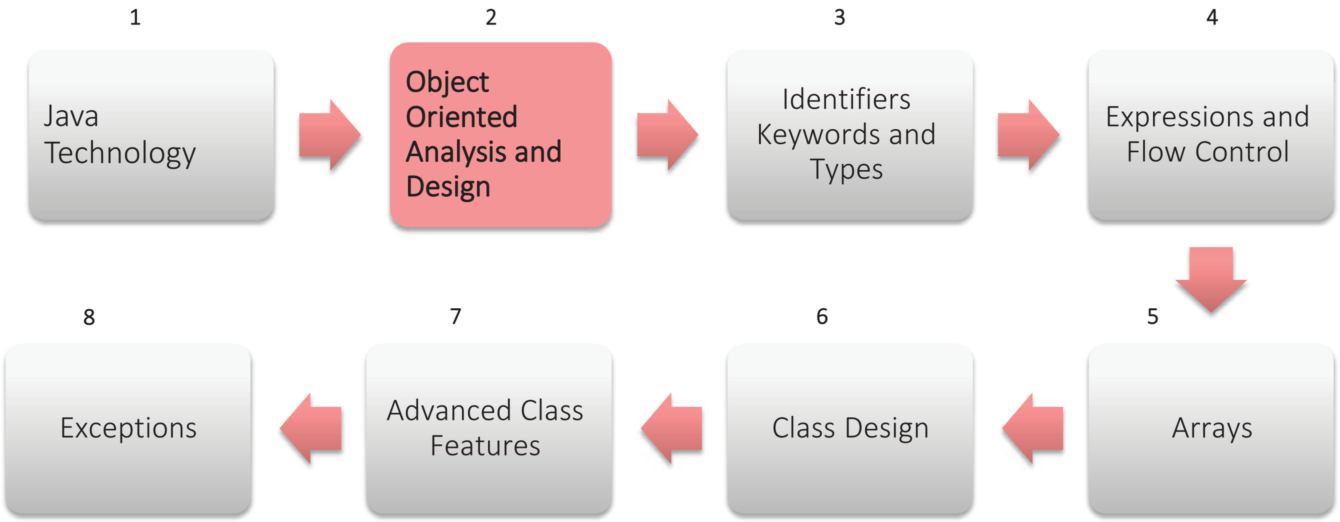
Object Oriented Analysis and Design

JOHN BRYCE
Leading in IT Education
matrix company

John Bryce Training, 29 Homa Umigdal St, Tel Aviv 6777129 // www.johnbryce.com

Objectives

JOHN BRYCE
Leading in IT Education
matrix company



1 Java Technology

2 Object Oriented Analysis and Design

3 Identifiers Keywords and Types

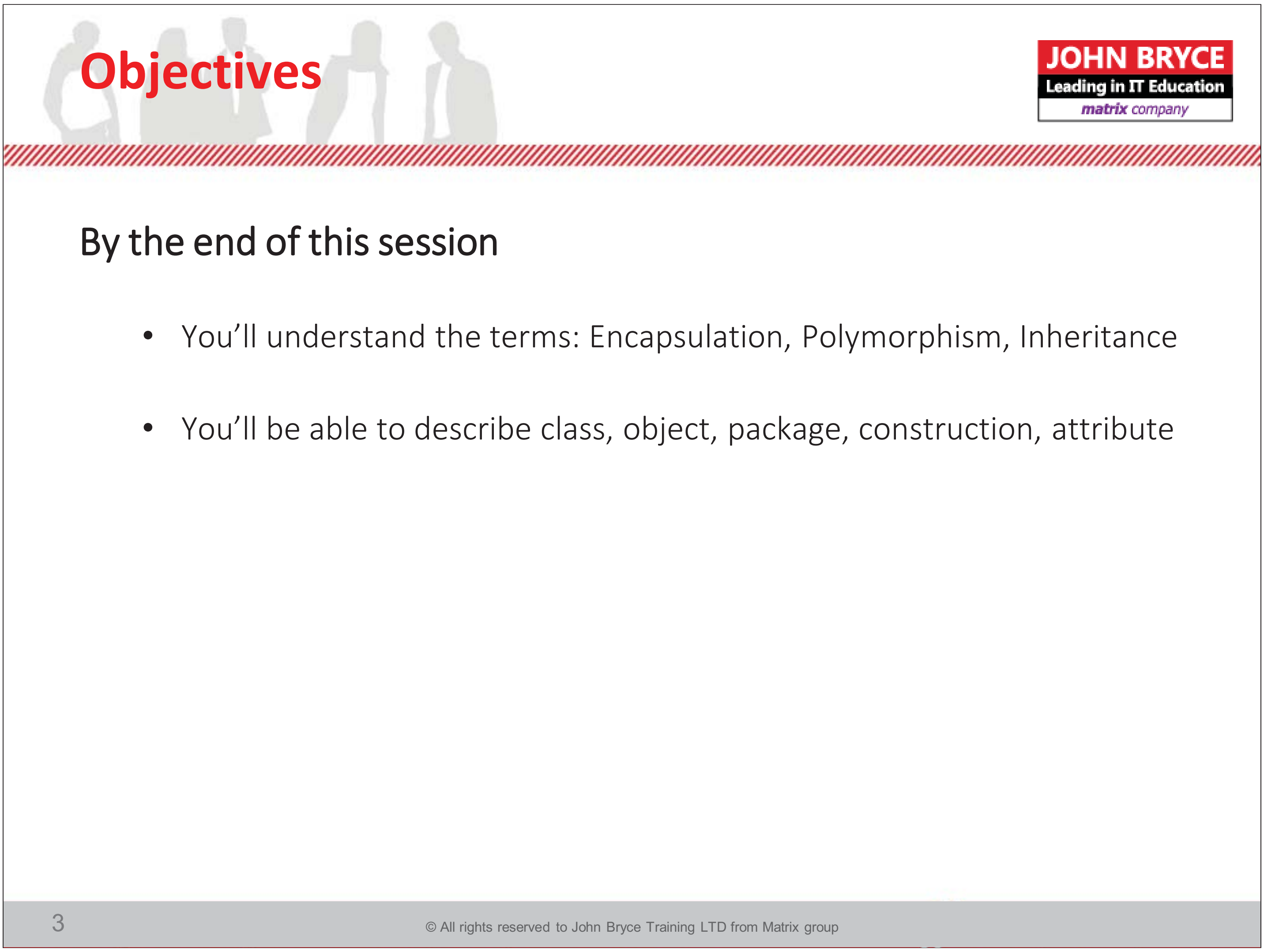
4 Expressions and Flow Control

5 Arrays

6 Class Design

7 Advanced Class Features

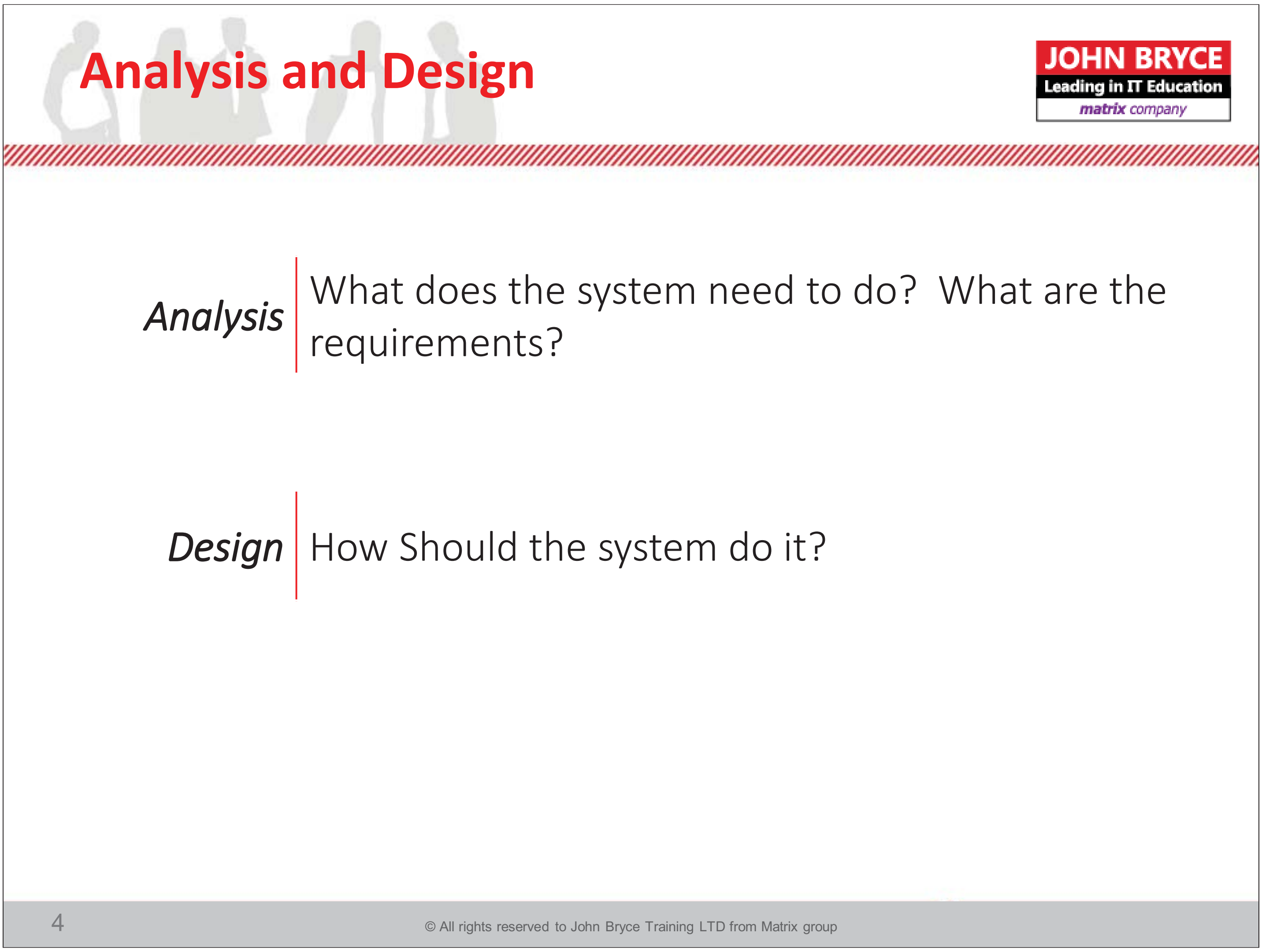
8 Exceptions



Objectives

By the end of this session

- You’ll understand the terms: Encapsulation, Polymorphism, Inheritance
- You’ll be able to describe class, object, package, construction, attribute

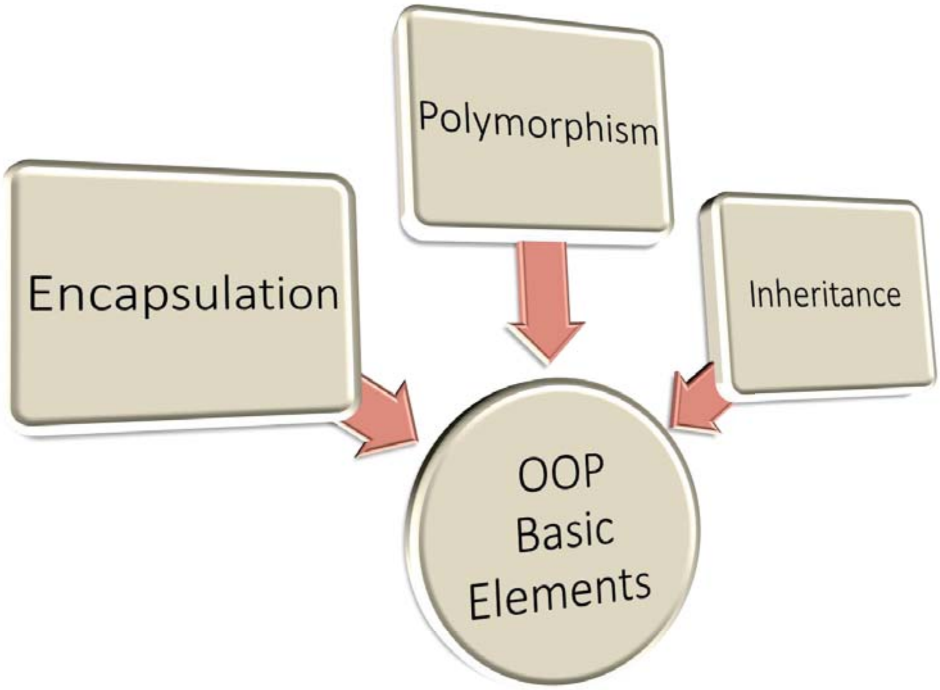


Analysis and Design

Analysis | What does the system need to do? What are the requirements?

Design | How Should the system do it?

Object Oriented Programming



OOP - Encapsulation

- Dividing the application into several entities
- Each entity has a well defined role
- Each entity encapsulates all data and functionality regarding its role within it

OOP - Inheritance

- Creates a new entity which is an extension of an existing one
- Reflects an "is a" relationship
- A derived (inherited) class is actually its ancestor plus additional data and/or functionality
- Inheritance enables to change simultaneously the basic structure of several different entities

OOP - Polymorphism

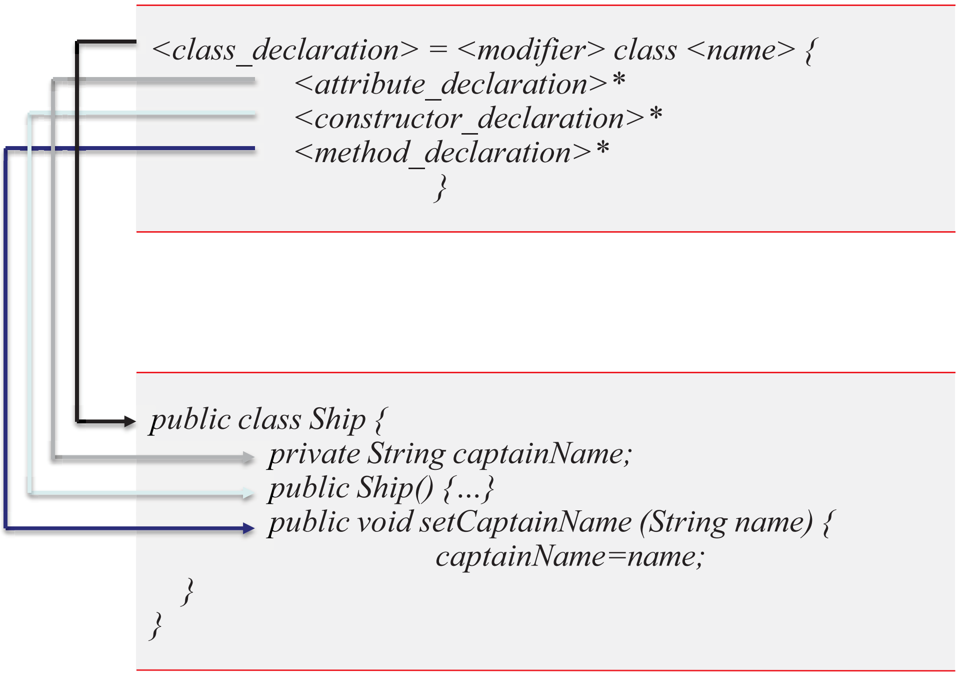
- Reference act differently on different occasions
- References may be assigned to objects of either the reference class or to objects which are descendants of that class
- A reference of a certain class might change its behavior according to the object type it references

Classes as Blueprints for Objects

- In manufacturing, a blueprint is a description of a device from which many physical devices are constructed
- In software, a class is a description of an object
- The class describes both the data (data members) of an object and its behavior (the methods it holds)
- Objects are instances of a class

Declaring Java Classes

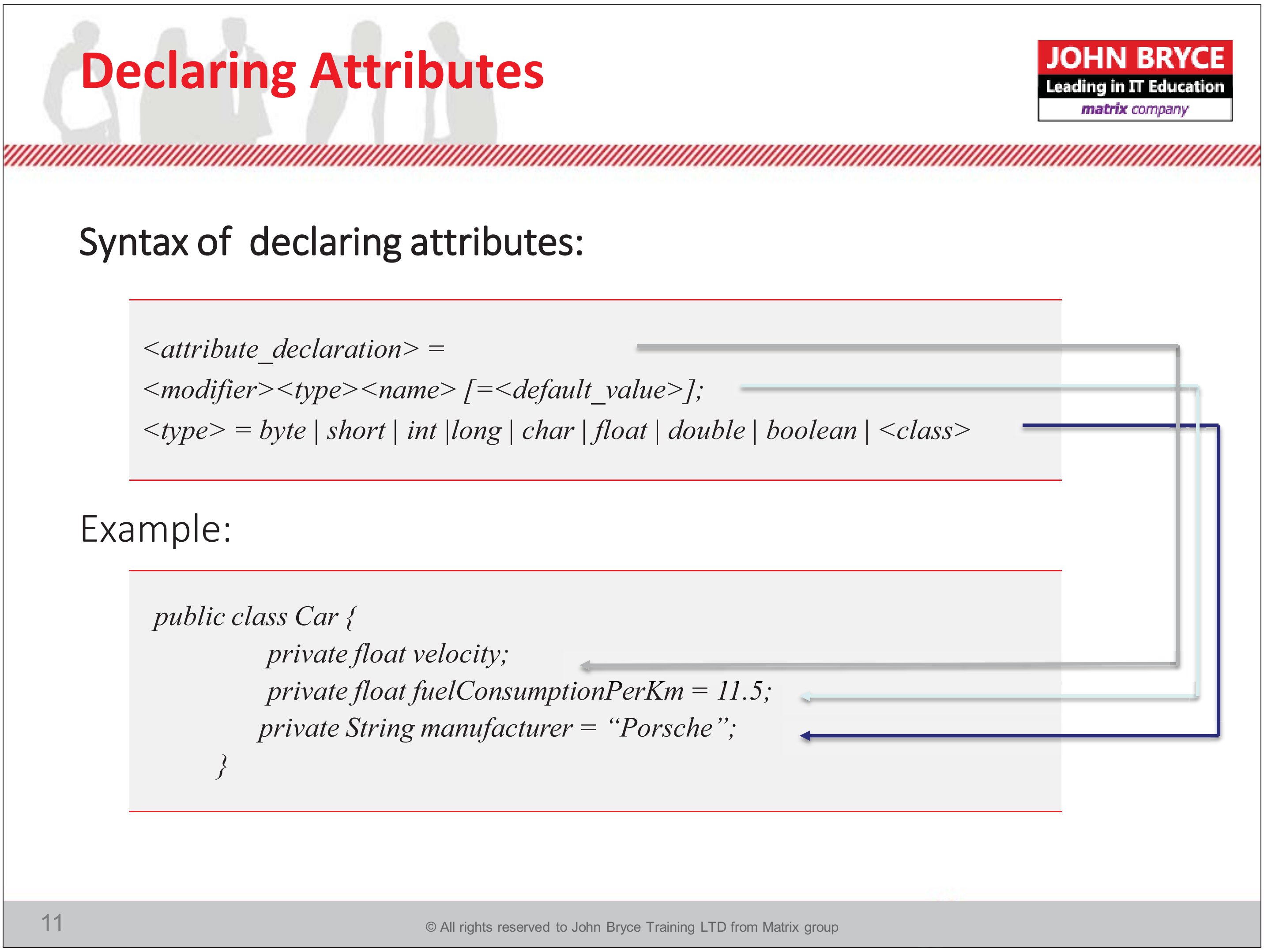
Basic syntax of a java class:



```
<class_declaration> = <modifier> class <name> {
            <attribute_declaration>*
            <constructor_declaration>*
            <method_declaration>*
                        }
```

Example:

```
public class Ship {
          private String captainName;
          public Ship() {…}
          public void setCaptainName (String name) {
                        captainName=name;
     }
}
```



Declaring Attributes

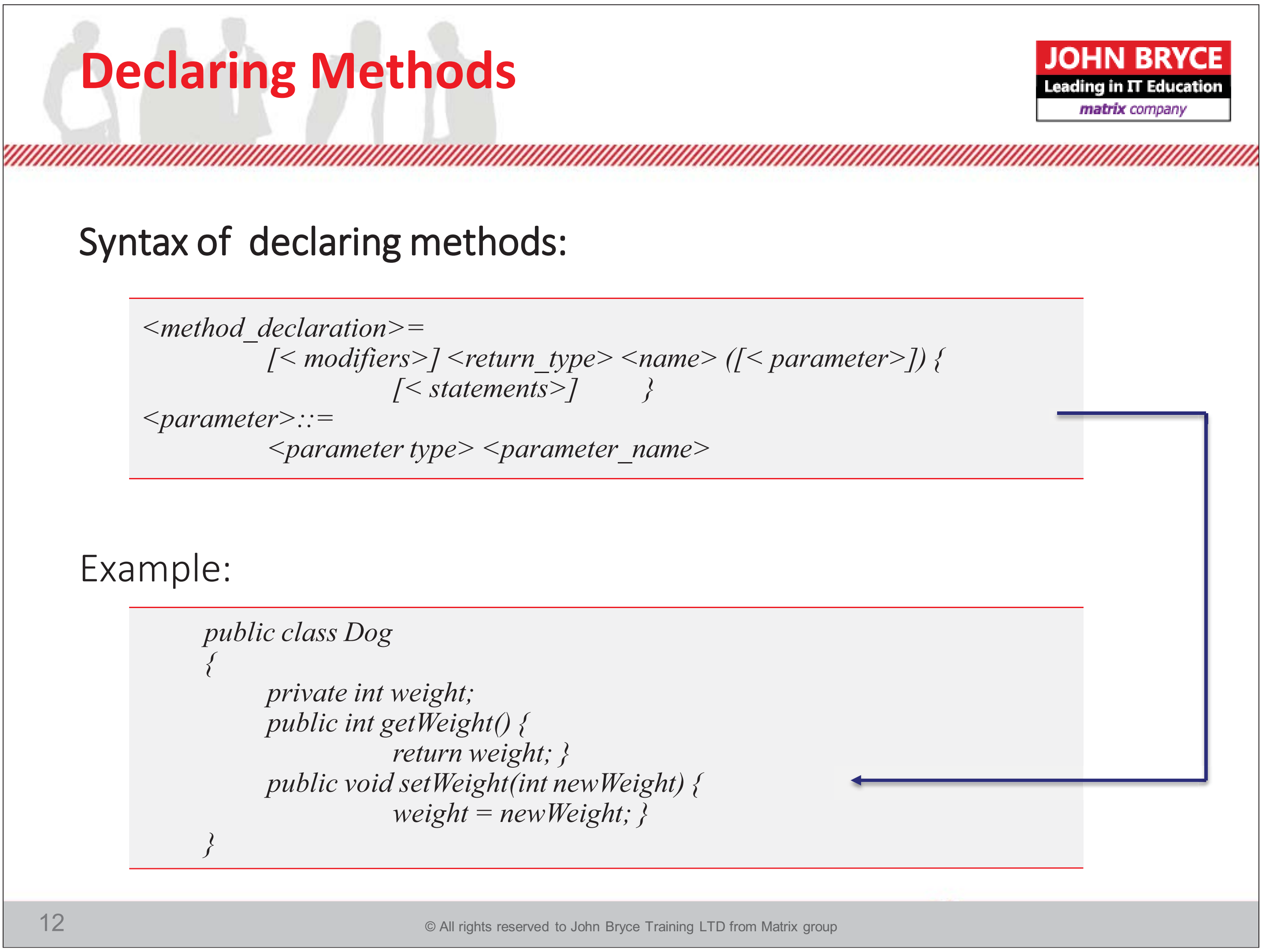
JOHN BRYCE
Leading in IT Education
matrix company

Syntax of declaring attributes:

```
<attribute_declaration> =
<modifier><type><name> [=<default_value>];
<type> = byte | short | int |long | char | float | double | boolean | <class>
```

Example:

```
public class Car {
            private float velocity;
            private float fuelConsumptionPerKm = 11.5;
           private String manufacturer = "Porsche";
       }
```



Declaring Methods

JOHN BRYCE
Leading in IT Education
matrix company

Syntax of declaring methods:

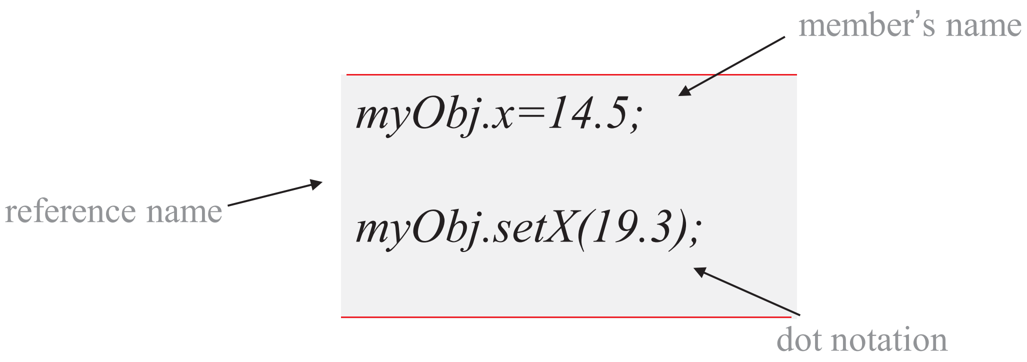
```
<method_declaration>=
             [< modifiers>] <return_type> <name> ([< parameter>]) {
                          [< statements>]        }
<parameter>::=
             <parameter type> <parameter_name>
```

Example:

```
      public class Dog
      {
            private int weight;
            public int getWeight() {
                          return weight; }
            public void setWeight(int newWeight) {
                          weight = newWeight; }
      }
```

Access to Object Members

Examples

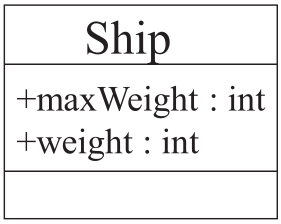


```
myObj.x=14.5;

myObj.setX(19.3);
```

Note: access is allowed according to the member's modifier

Information Hiding

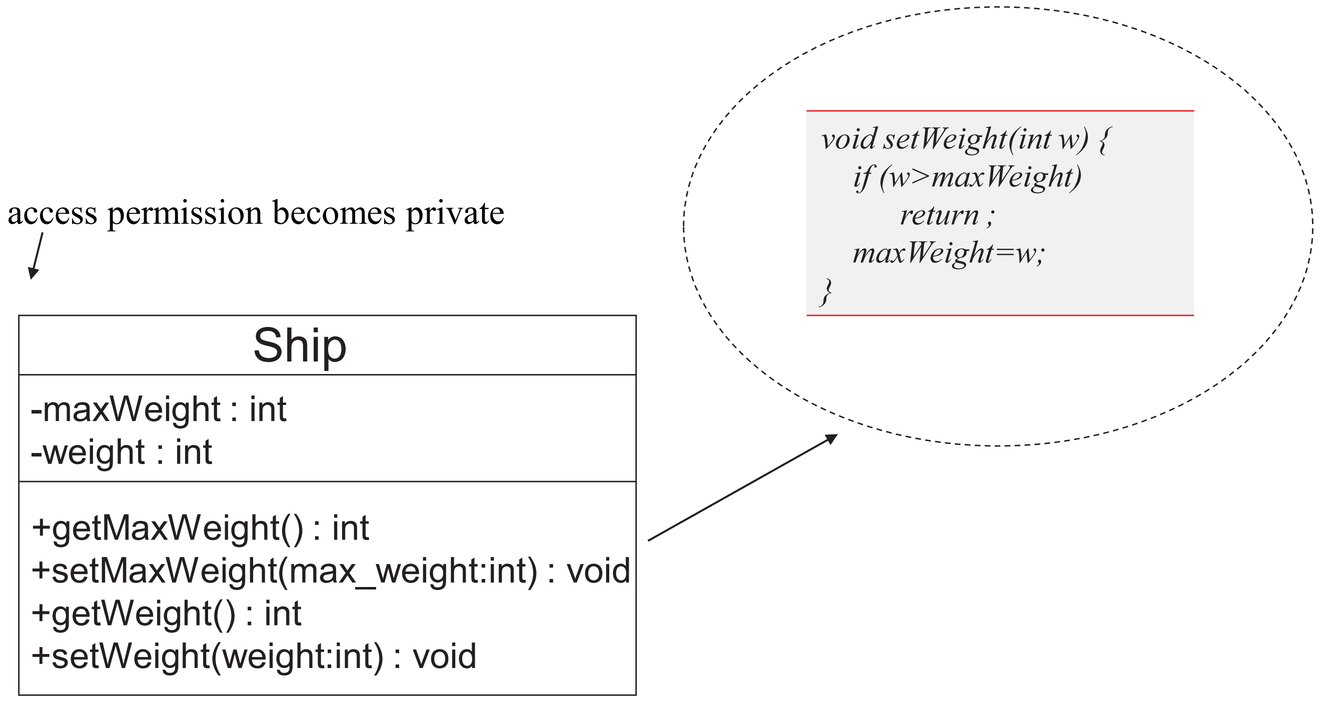
The Problem



```
Ship s;
s.maxWeight =340;                 //Max allowed weight is 200.
s. weight =s.maxWeight +100;      //No checking is done.
```

Information Hiding

Solution



access permission becomes private

Ship

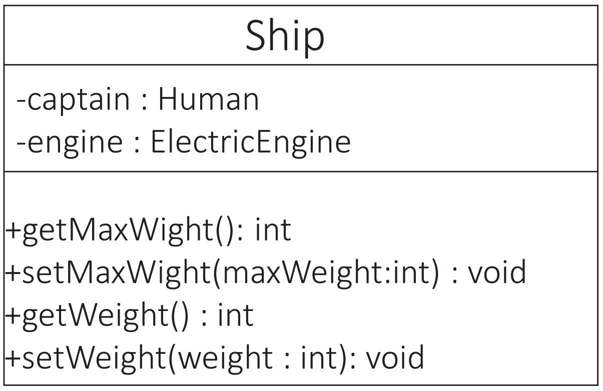
-maxWeight : int
-weight : int

+getMaxWeight() : int
+setMaxWeight(max_weight:int) : void
+getWeight() : int
+setWeight(weight:int) : void

```
void setWeight(int w) {
   if (w>maxWeight)
        return ;
   maxWeight=w;
}
```

Encapsulation

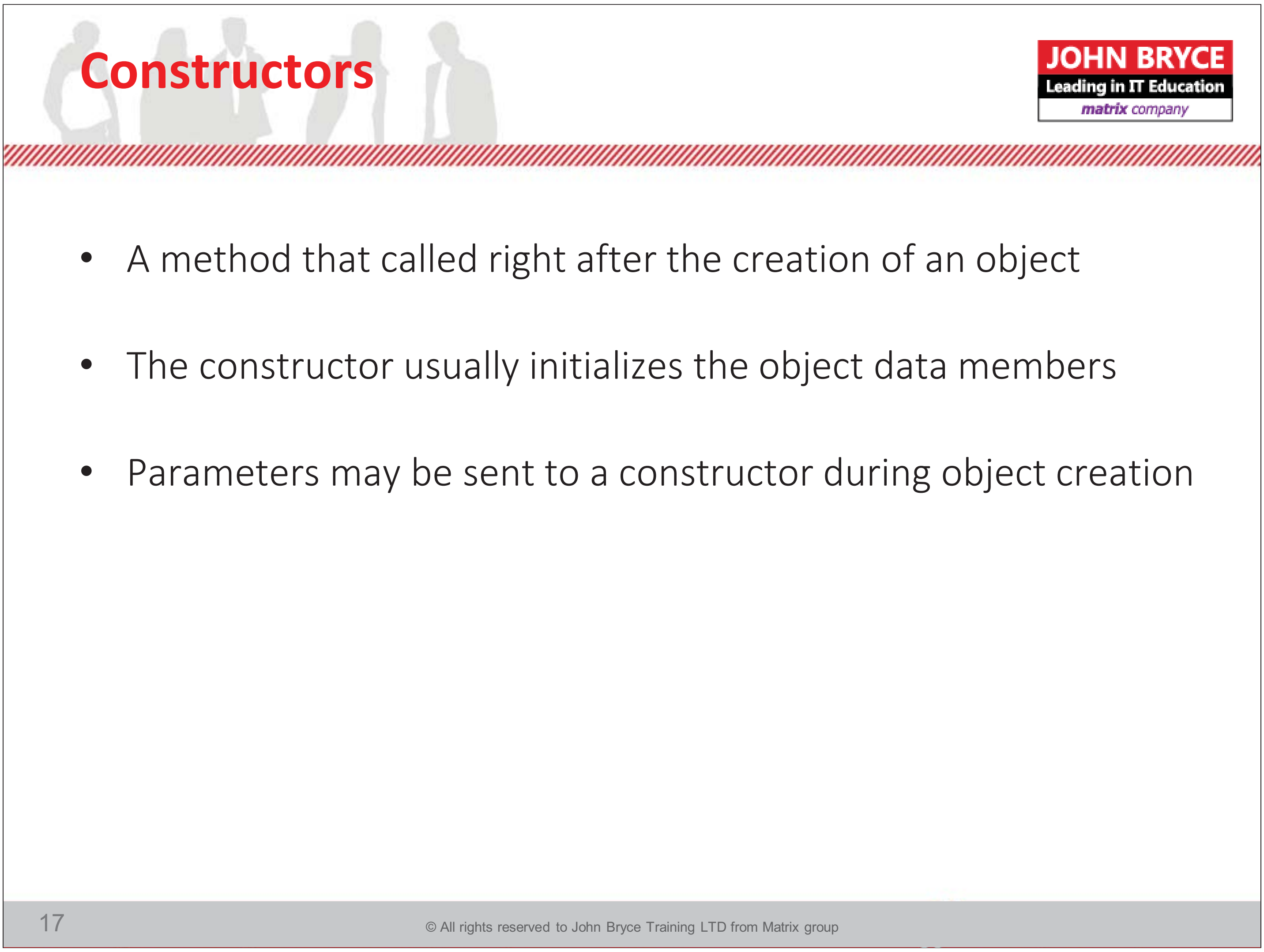
- Encapsulation of related data and functionality within one class (improves maintainability)
- Hides the implementation details of the class
- Forces the client to use interfaces for accessing data



Ship

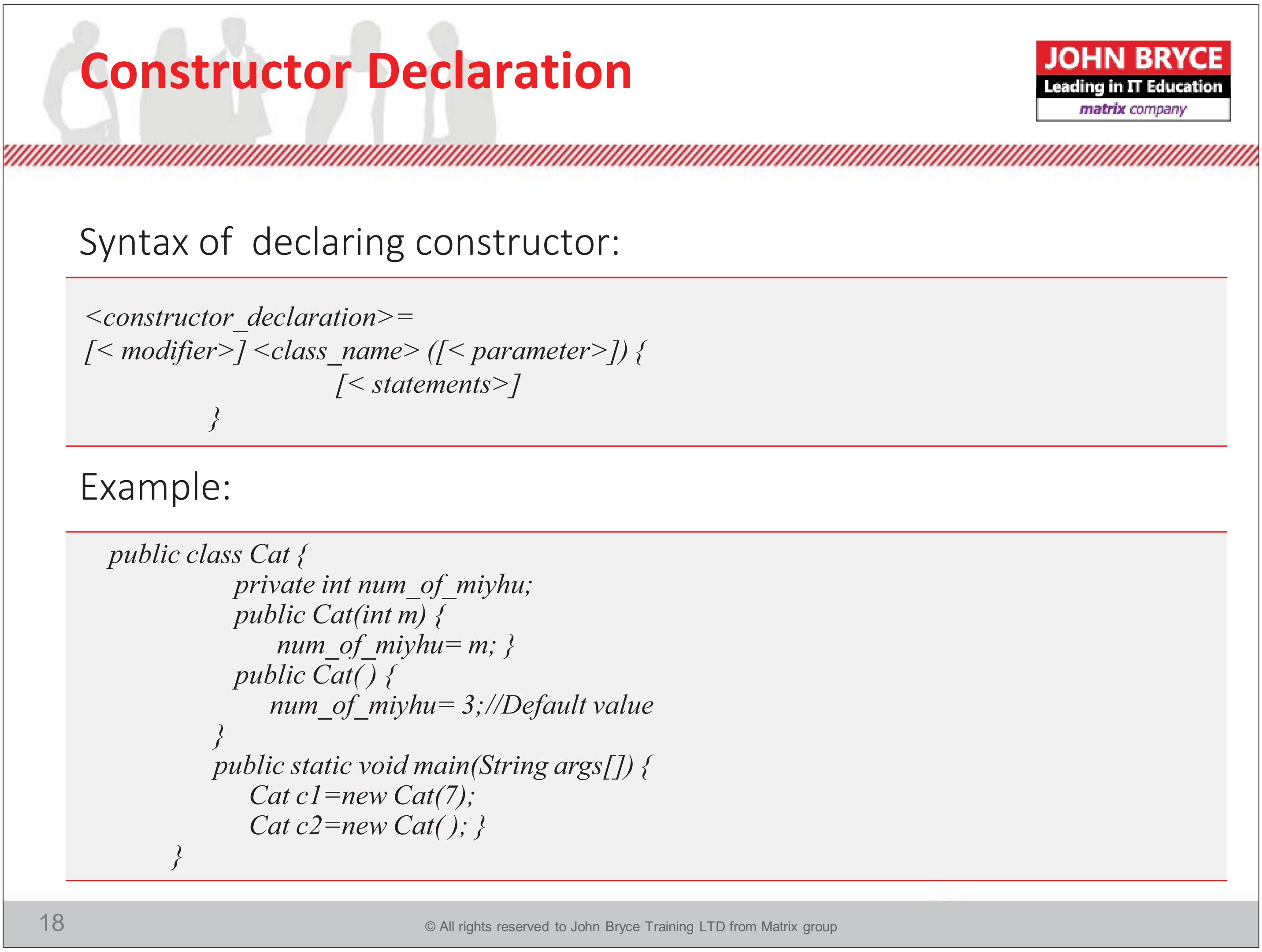
-captain : Human
-engine : ElectricEngine

+getMaxWight(): int
+setMaxWight(maxWeight:int) : void
+getWeight() : int
+setWeight(weight : int): void



Constructors

- A method that called right after the creation of an object
- The constructor usually initializes the object data members
- Parameters may be sent to a constructor during object creation



Constructor Declaration

Syntax of declaring constructor:

```
<constructor_declaration>=
[< modifier>] <class_name> ([< parameter>]) {
                              [< statements>]
               }
```

Example:

```
  public class Cat {
                 private int num_of_miyhu;
                 public Cat(int m) {
                      num_of_miyhu= m; }
                 public Cat( ) {
                     num_of_miyhu= 3;//Default value
               }
               public static void main(String args[]) {
                   Cat c1=new Cat(7);
                   Cat c2=new Cat( ); }
          }
```

Default Constructor

- Every class has a constructor
- Default constructor will be supplied automatically
- When programmer adds a constructor, the default constructor vanishes
- The default constructor takes no arguments and has no body. It only exists so the call for a constructor made during objects creation will be supplied

Package

- Java classes organized into packages
- Every package should contain classes that relates to the same subject
- Enables access privileges restricted to the package classes
- Default package will be used in case no package statement appears in the file
- Packages stored in the directory tree containing the package name

Define a Package

Syntax:

```
<package_declaration> =
            package <top_package_name>[.<sub_package_name];
```

Example:

```
package building.construction.house ;
```

Package declaration should appear only once, at the beginning of a file. Packages are hierarchical and are separated by dots.

Compiling into Package

- Compiling:

```
javac –d <root-location> <sources>
```

Example:

```
javac –d mainDir *.java
```

- Archiving classes / packages into jars:

```
jar –cf <result-jar-file-name> <sources>
```

Example:

```
jar –cf  application.jar  mainDir
```

Import Declarations

JOHN BRYCE
Leading in IT Education
matrix company

Basic syntax:

```
<import_declaration> =
        import <pkg_name>[.<sub_pkg_name>]*.<class_name | *>;
```

Examples:

```
import shapes.rectangles.*;          //define a path to all classes
                                     //in that package.
import java.lang.*;                  //imported always by default
import java.util.List;               //define a path to the List class.
```

Source File Layout

JOHN BRYCE
Leading in IT Education
matrix company

Basic syntax of a java source file:

```
<source_declaration>=
                              [<package_declaration>]
                              <import_declaration>*
                    <class_declaration>+
```

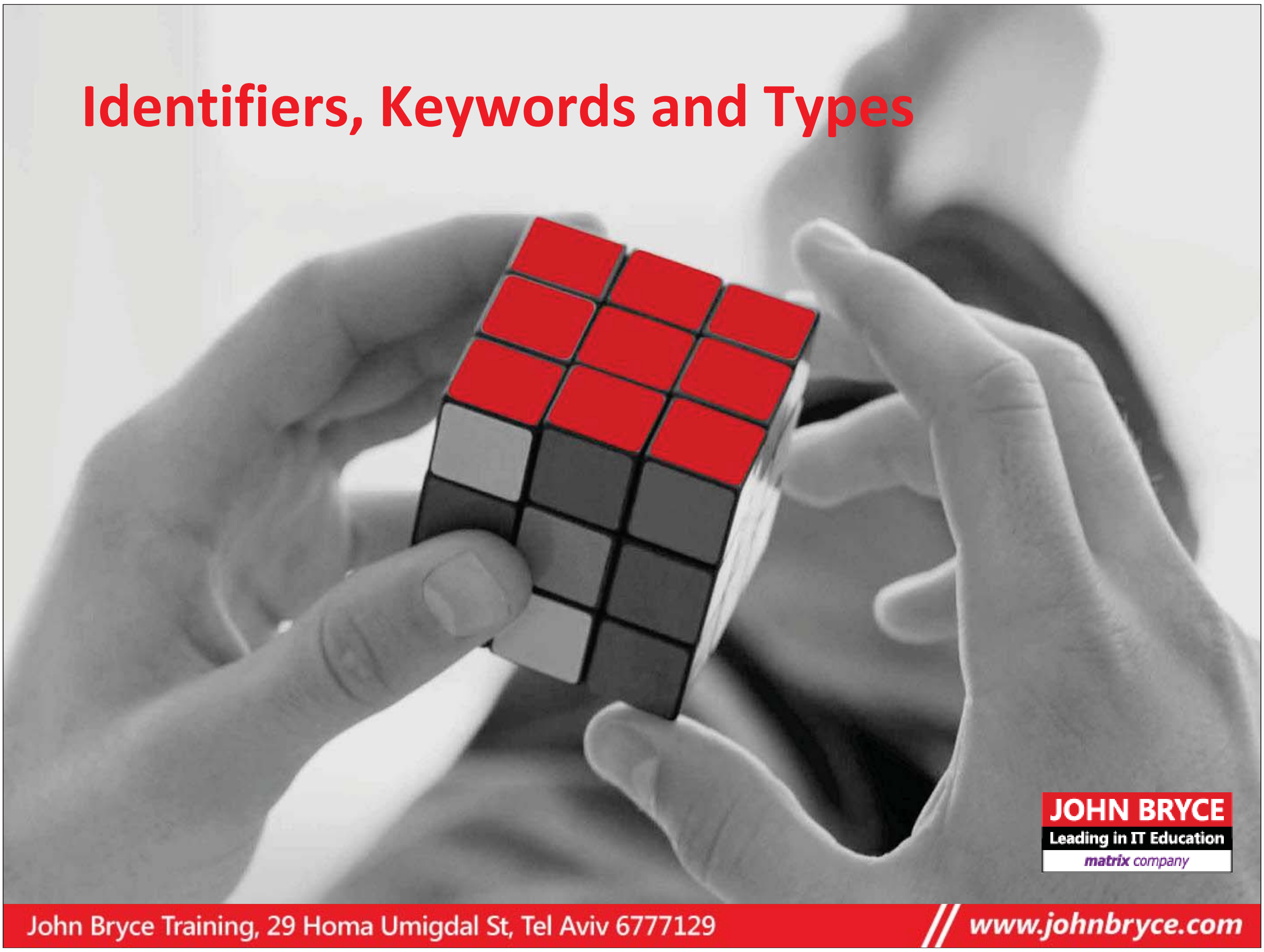
Example, the Box.java file:

```
package shapes.rectangles;
import java.util.Map;
import java.io.*;
            public class Box {
                      // Class definition goes here.
}
```

Compile and run

Summary

Class	A blueprint source code for instantiating objects
Object	An instance of a class
Attribute (Data Member, Instance Variable, Data Field)	A data element of an object
Method (Class Function)	A behavioral element of a class
Constructor	A method that is called whenever an object of a specific class is instantiated. Used to initialize the data members
Package	A grouping of classes (library)



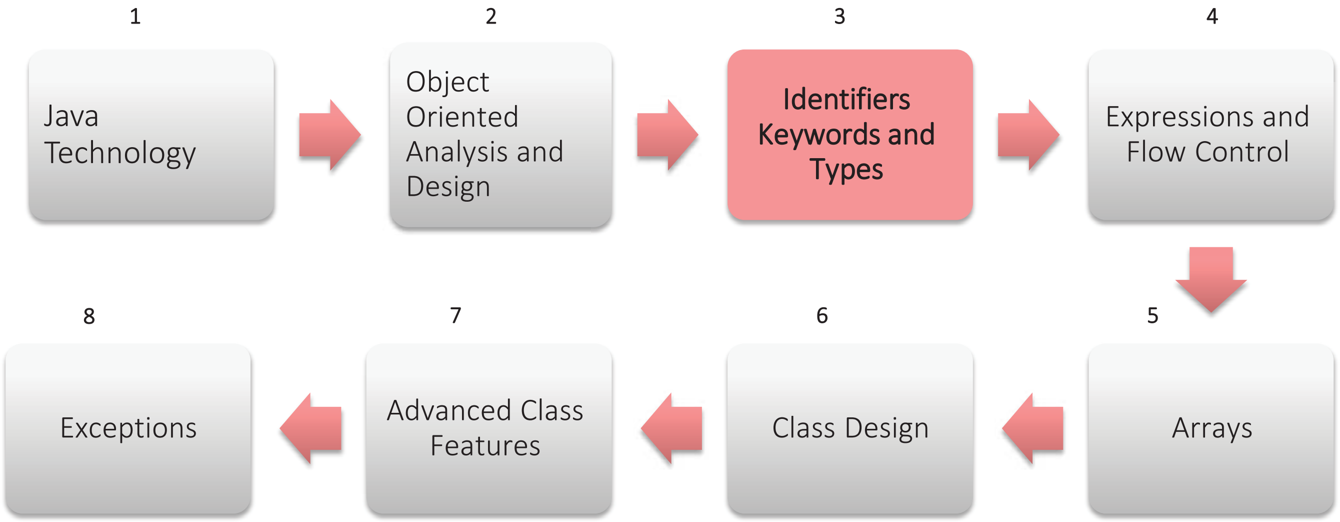
Identifiers, Keywords and Types

JOHN BRYCE
Leading in IT Education
matrix company

John Bryce Training, 29 Homa Umigdal St, Tel Aviv 6777129 // www.johnbryce.com

Objectives

JOHN BRYCE
Leading in IT Education
matrix company



1 Java Technology

2 Object Oriented Analysis and Design

3 Identifiers Keywords and Types

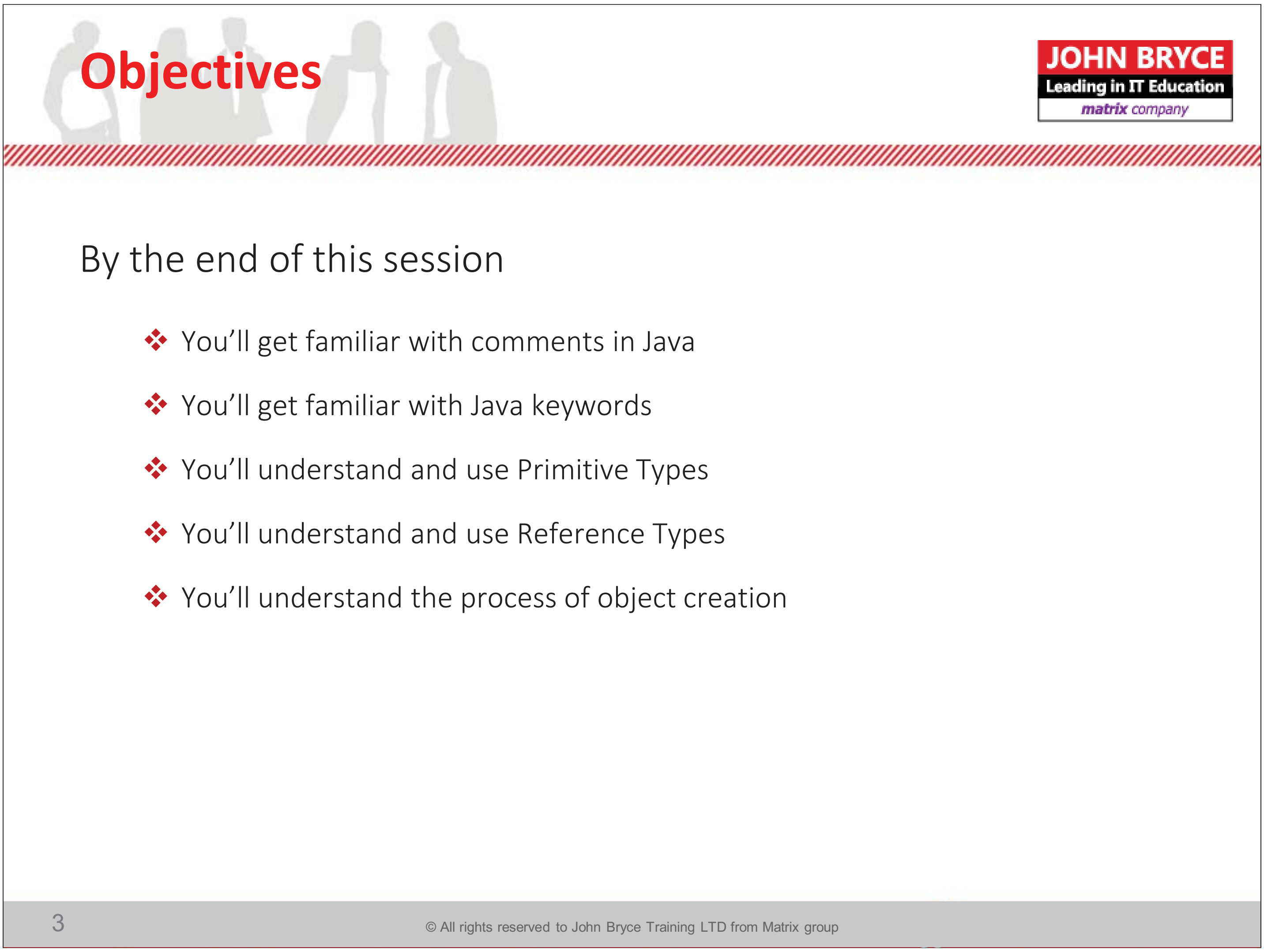
4 Expressions and Flow Control

5 Arrays

6 Class Design

7 Advanced Class Features

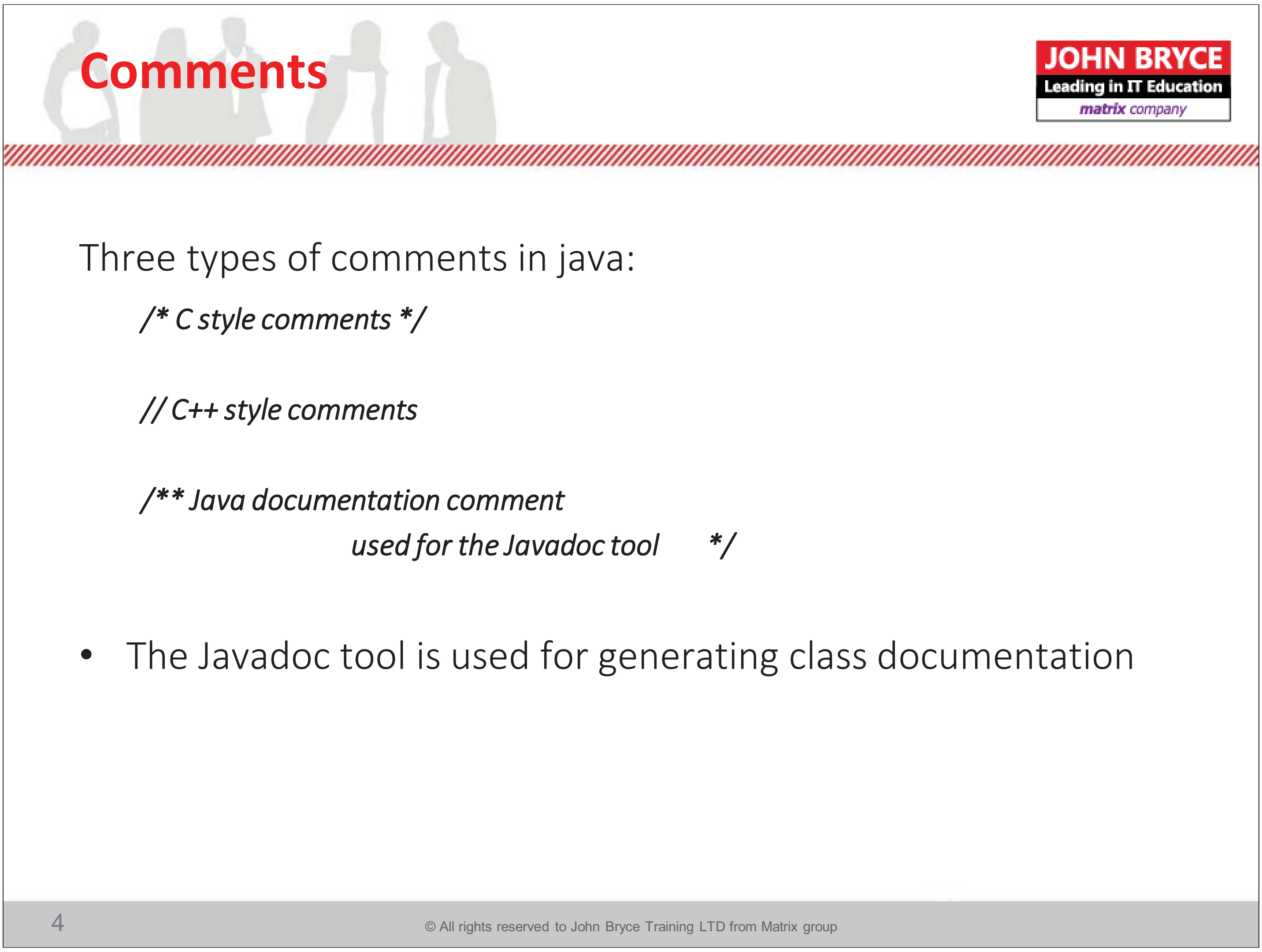
8 Exceptions



Objectives

By the end of this session

- You'll get familiar with comments in Java
- You'll get familiar with Java keywords
- You'll understand and use Primitive Types
- You'll understand and use Reference Types
- You'll understand the process of object creation



Comments

Three types of comments in java:

```
/* C style comments */

// C++ style comments

/** Java documentation comment
                    used for the Javadoc tool     */
```

- The Javadoc tool is used for generating class documentation

A Statement

JOHN BRYCE
Leading in IT Education
matrix company

- Every sentence (one or more lines of code) must terminate with a semicolon (;)
- White spaces may be entered between statement's parts

Example:

```
result = num1 + num2 + num3 – num4 ;
```

Blocks

JOHN BRYCE
Leading in IT Education
matrix company

- Every new scope (e.g., the beginning of a method) must starts with an opening brace ({) and ends with a closing one (})
- White spaces may be used inside blocks
- Blocks can be nested

Example:

```
public class Car {
    private int max_speed;
                 private String model;
                              . . .
                 public float get_fuel_consumption() {
                              . . .
                   }
 }
```

Identifiers

- Are names that we give to variables, classes, or methods
- Can start with a Unicode letter, underscore (_) or a dollar sign ($)
- Are case-sensitive and have no maximum length

Examples:

UserName

userName

user_name

_sys_var1

$change

Java Keywords

abstract
boolean
byte
break
char
case
catch
class
continue
double
default
do
extends
else
false
final
finally
float
for
interface
if
implements
import
instanceof
int
long
null
native
new
package
private
protected
public
return
short
static
strictfp
super
switch
synchronized
this
throw
throws
transient
try
true
void
volatile
while

Primitive Types

JOHN BRYCE
Leading in IT Education
matrix company

- *boolean* – This is a logical type that may hold either “true” or “false”

- *char* – A textual type that may hold a 16-bit unicode character

```
char ch = ‘a’;
```

Primitives

JOHN BRYCE
Leading in IT Education
matrix company

Integral Types

1. *byte* – A 8 bit integer, ranges: -2^{7} to 2^{7}-1
2. *short* – A 16 bit integer, ranges: -2^{15} to 2^{15}-1
3. *int* – A 32 bit integer, ranges: -2^{31} to 2^{31}-1
4. *long* – A 64 bit integer, ranges: -2^{63} to 2^{63}-1

Primitives

JOHN BRYCE
Leading in IT Education
matrix company

Floating Types

1. *float* – A 32 bit floating point value
2. *double* - A 64 bit floating point value

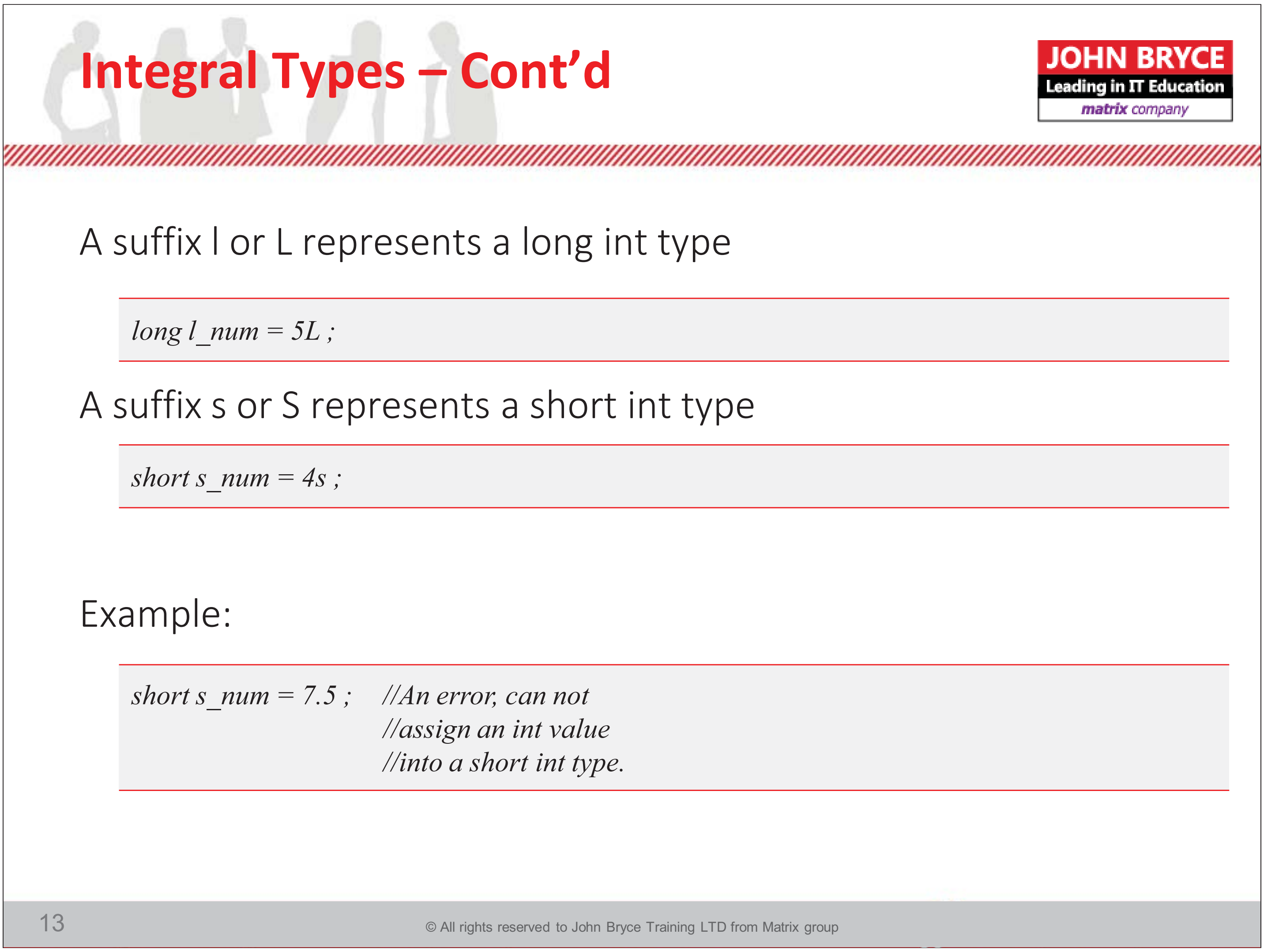
Integral Types – Cont'd

JOHN BRYCE
Leading in IT Education
matrix company

- A leading 0 indicates an octal value
- A leading 0x indicates a hexadecimal value
- Integrals have a default type of int
- All integrals types in java are signed numbers

Example:

```
indecVal = 26; // The number 26, in decimal
int octVal = 032; // The number 26, in octal
int ht exVal = 0x1a; // The number 26, in hexadecimal
```



Integral Types – Cont'd

A suffix l or L represents a long int type

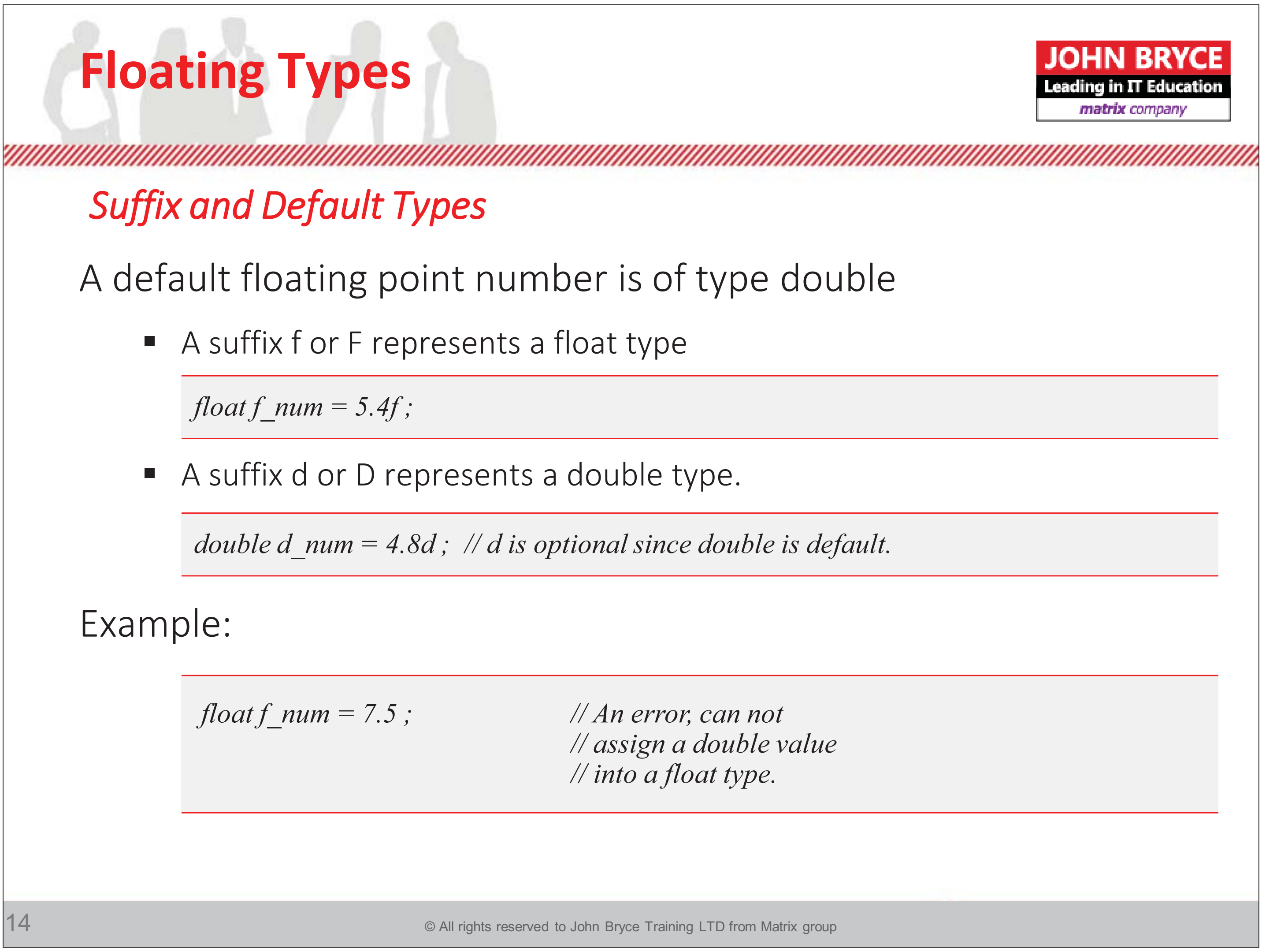
```
long l_num = 5L ;
```

A suffix s or S represents a short int type

```
short s_num = 4s ;
```

Example:

```
short s_num = 7.5 ;    //An error, can not
                       //assign an int value
                       //into a short int type.
```



Floating Types

Suffix and Default Types

A default floating point number is of type double

- A suffix f or F represents a float type

```
float f_num = 5.4f ;
```

- A suffix d or D represents a double type.

```
double d_num = 4.8d ;  // d is optional since double is default.
```

Example:

```
float f_num = 7.5 ;            // An error, can not
                               // assign a double value
                               // into a float type.
```

References to Objects

- Objects are created dynamically on the heap
- References (which may be local or global) are used as handles to objects
- A reference holds an object address which is used without pointers notation

References Vs. Primitives

```
public class Example {
    private int counter=0;                                  //counter is a primitive.
    private Box b1;                                         //b1 is a null reference
                                                            //to a Box object.

    public Example(int height, int width, int length)  {
      b1= new Box(height,width,length);                     //b1 is now
                                                            //initialized and points
                                                            //to the Box object
                                                            //located on the heap.
 }
 public static void main(String args[ ]) {
    Example e1=new Example(3,6,5);                          //e1 is a reference.
   …
 }
  …
}
```

Instantiation Of an Object

- Instantiation of an object is done using the operator *new*

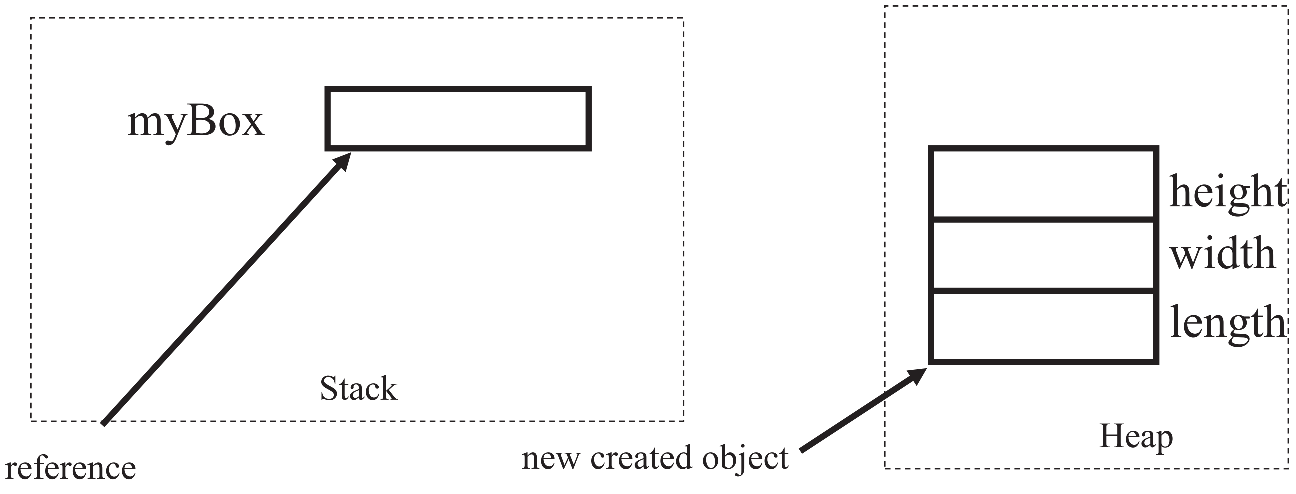
```
new MyClass( firstArg, SecArg);
```

- This will result in the following sequence:
  - Memory is allocated for the new object on the heap
  - Data members are initialized to their default values
  - A constructor is executed
  - The object is assigned to the reference that from now on will be used as its handle

Step 1

Memory allocation

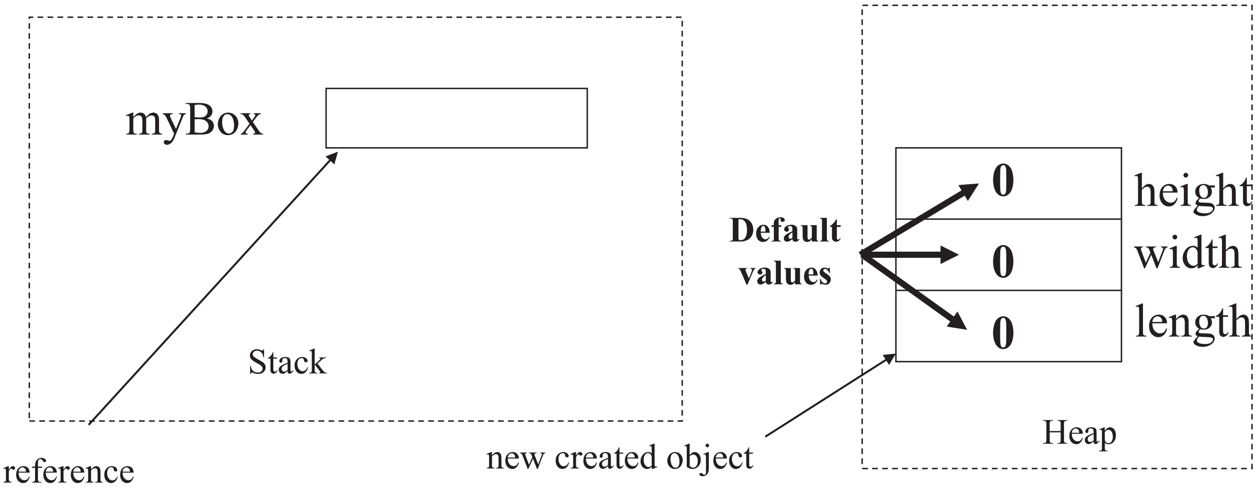
Box myBox=new Box(5,6,7);



Step 2

Data members are initialized to their default values

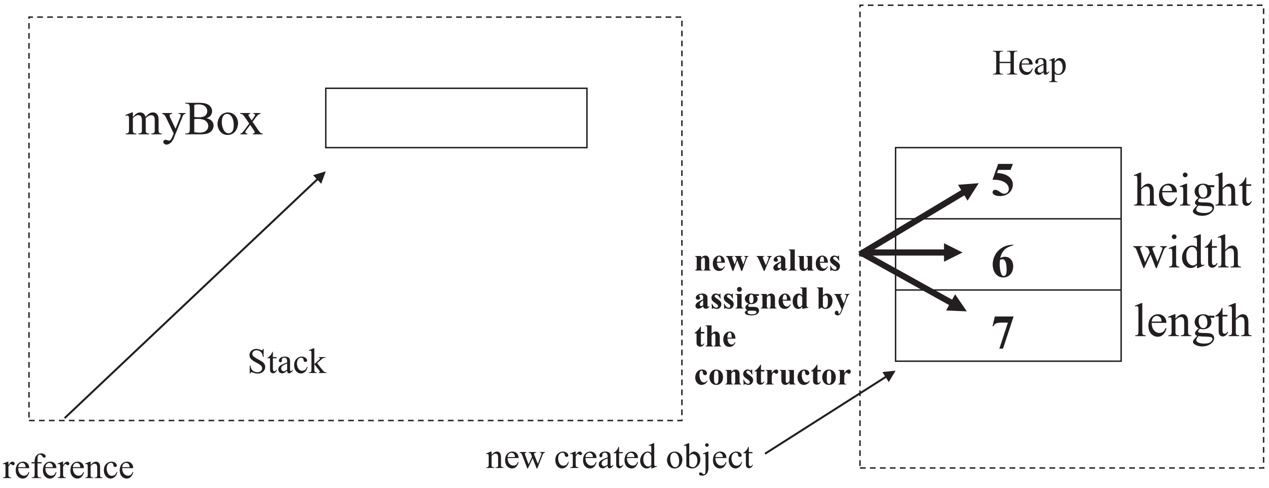
Box myBox=new Box(5,6,7);



Step 3

Constructor is executed

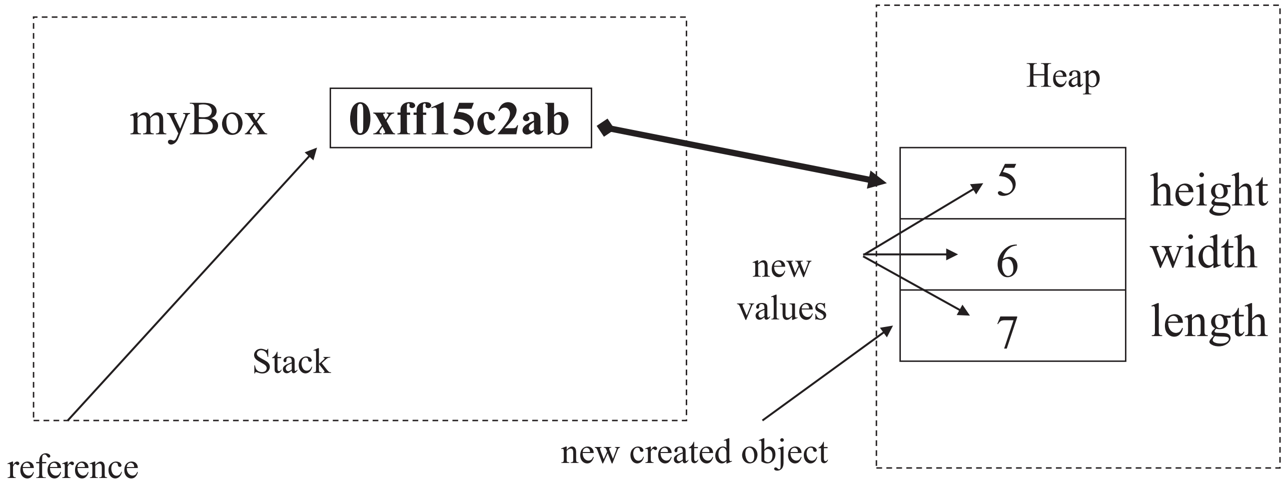
Box myBox=new Box(5,6,7);



Step 4

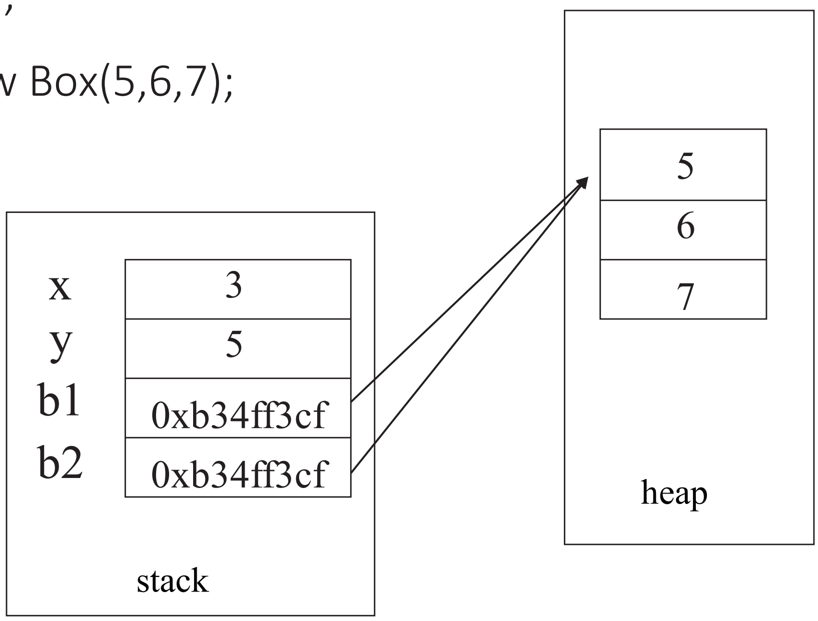
The newly created object is assigned to its reference

```
Box myBox=new Box(5,6,7);
```



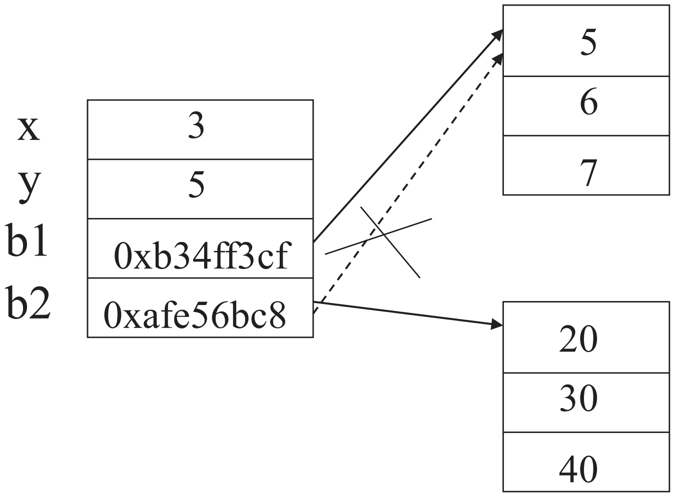
References in Action

```
int x=3, y=5;

Box b1=new Box(5,6,7);

Box b2=b1;
```



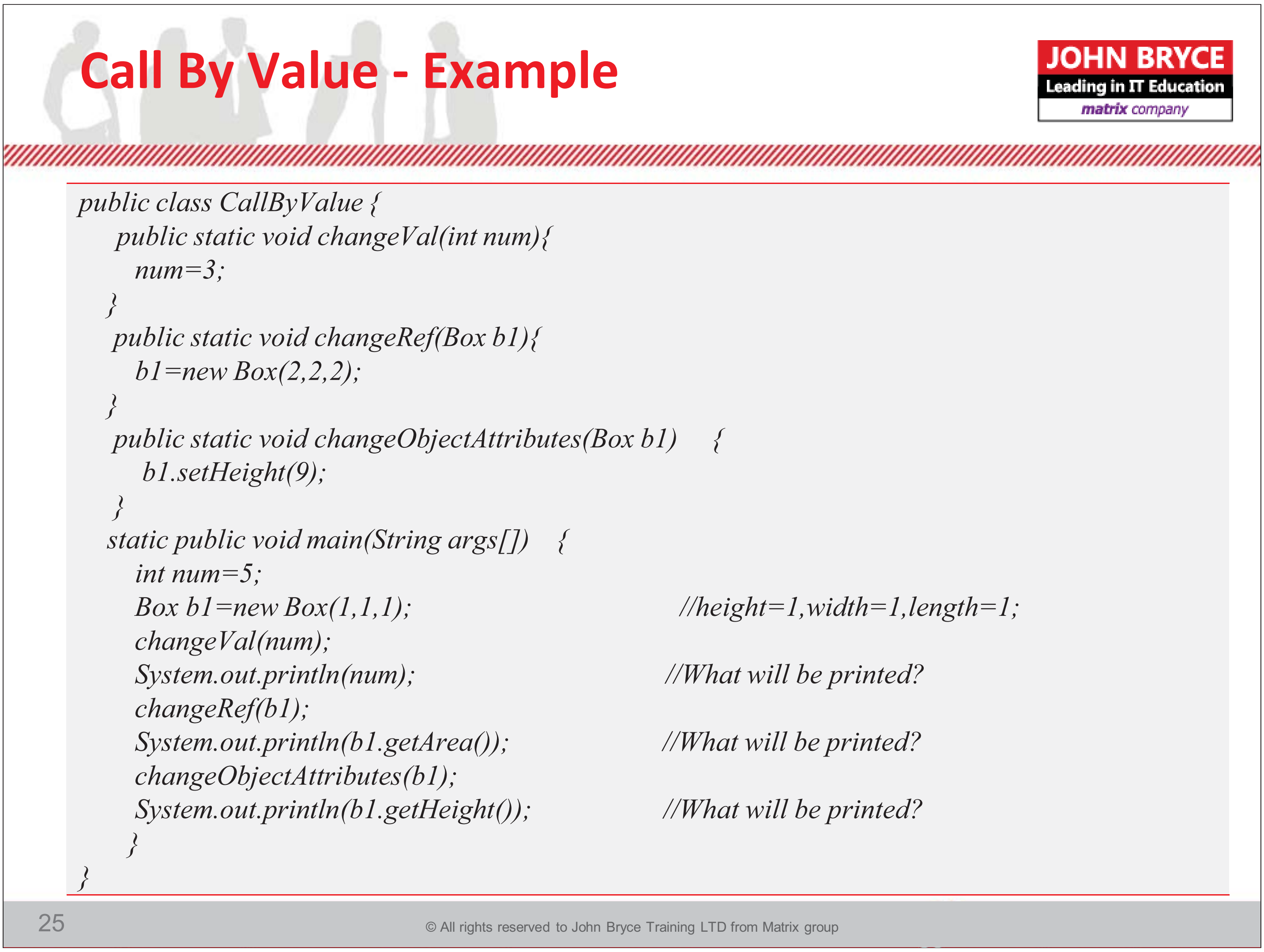
Reassignment of a Reference

```
int x=3, y=5;

Box b1=new Box(5,6,7);

Box b2=b1;

      …

b2=new Box(20,30,40);
```



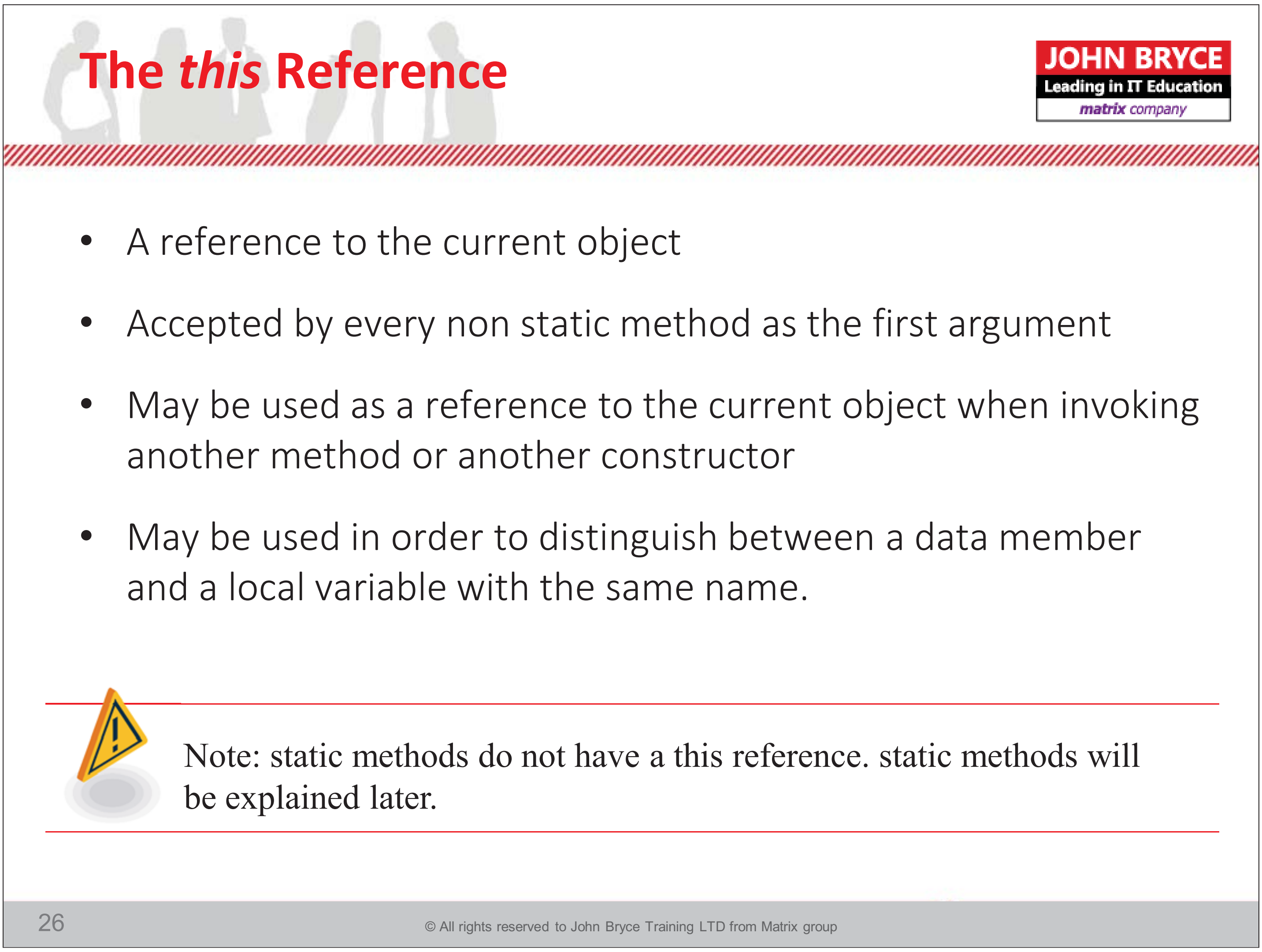
Methods Arguments

- **Call by value -** arguments to methods
- If primitive is sent to a method as an argument, a copy of it is created and passed by value to the method
- If reference is sent to a method as an argument, a copy of it is created and passed by value to the method
- The method’s argument is another reference to the original object. Both references may affect the same object
- Objects can not be passed to methods



Call By Value - Example

```
public class CallByValue {
    public static void changeVal(int num){
      num=3;
   }
   public static void changeRef(Box b1){
      b1=new Box(2,2,2);
   }
   public static void changeObjectAttributes(Box b1)     {
       b1.setHeight(9);
    }
   static public void main(String args[])    {
      int num=5;
      Box b1=new Box(1,1,1);                           //height=1,width=1,length=1;
      changeVal(num);
      System.out.println(num);                        //What will be printed?
      changeRef(b1);
      System.out.println(b1.getArea());               //What will be printed?
      changeObjectAttributes(b1);
      System.out.println(b1.getHeight());             //What will be printed?
     }
}
```



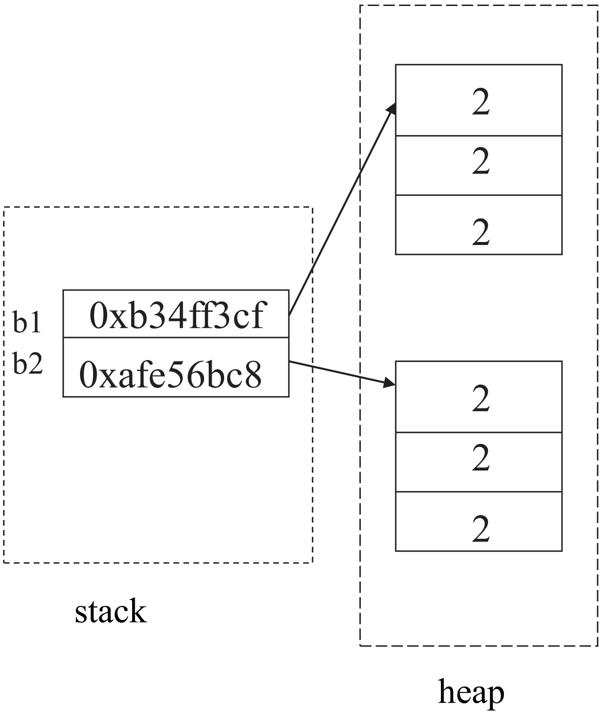
The *this* Reference

- A reference to the current object
- Accepted by every non static method as the first argument
- May be used as a reference to the current object when invoking another method or another constructor
- May be used in order to distinguish between a data member and a local variable with the same name.

Note: static methods do not have a this reference. static methods will be explained later.

this In Action

```
public class Box {
    private int height;
    private int width;
    private int length;
    public Box(int height,int width,int length)
      {
        this.height=height;
        this.width=width;
        this.length=length;
    }
     public Box(Box bArg) {
        this.height=bArg.height;
        this.width=bArg.width;
        this.length=bArg.length;
    }
    public Box replicateBox(){
        Box tmpBox=new Box(this);
        return tmpBox;
    }
  public static void main(String args[ ]) {
      Box b1=new Box(2,2,2);
      Box b2=b1.replicateBox();
      //. . . rest of main.
   }
}
```



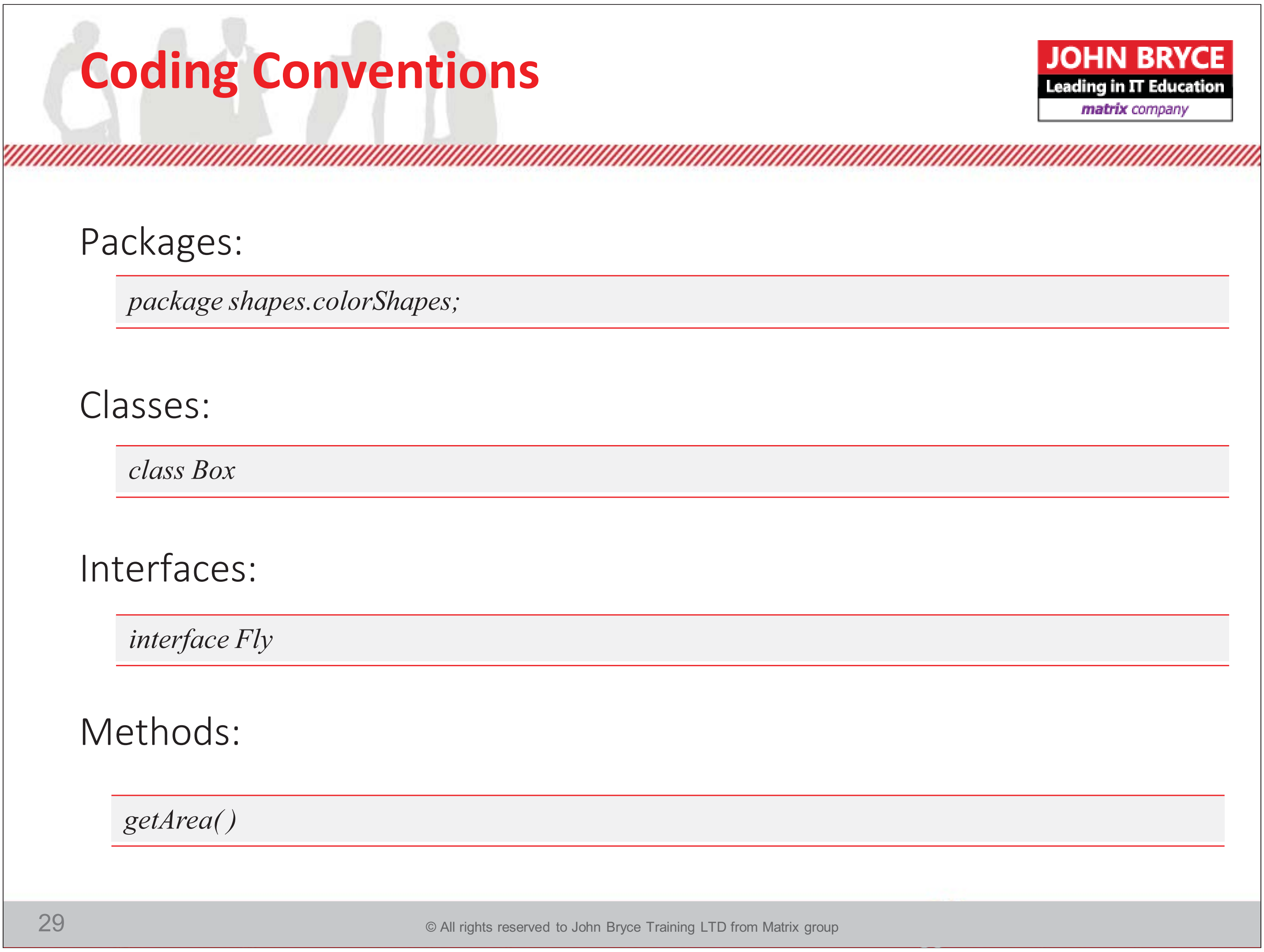
The *this* Reference-cont'd

Original code

```
public class ThisExample {
    int num;
    public void chgNum(int num)
    {
        this.num=num;
    }
    static public void main(String args[])
   {
        ThisExample te=new ThisExample();
        te.chgNum(12);
    }
}
```

Code converted by the compiler

```
public class ThisExample {
    int num;
    public void chgNum(ThisExample this, int num)
    {
        this.num=num;
    }
    static public void main(String args[])
     {
        ThisExample te=new ThisExample( );
        chgNum(te, 12);
     }
}
```



Coding Conventions

Packages:

```
package shapes.colorShapes;
```

Classes:

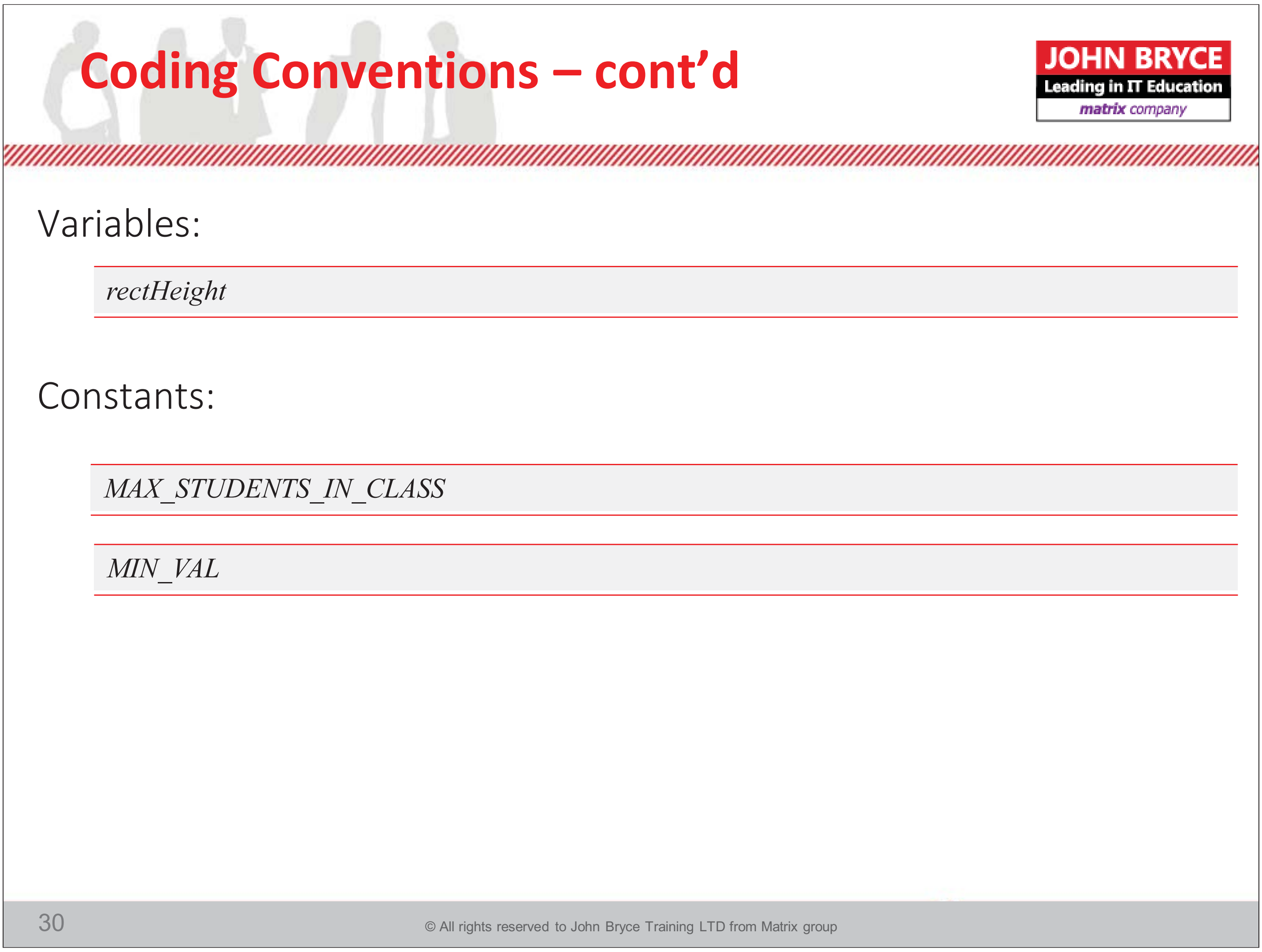
```
class Box
```

Interfaces:

```
interface Fly
```

Methods:

```
getArea( )
```



Coding Conventions – cont’d

Variables:

```
rectHeight
```

Constants:

```
MAX_STUDENTS_IN_CLASS
```

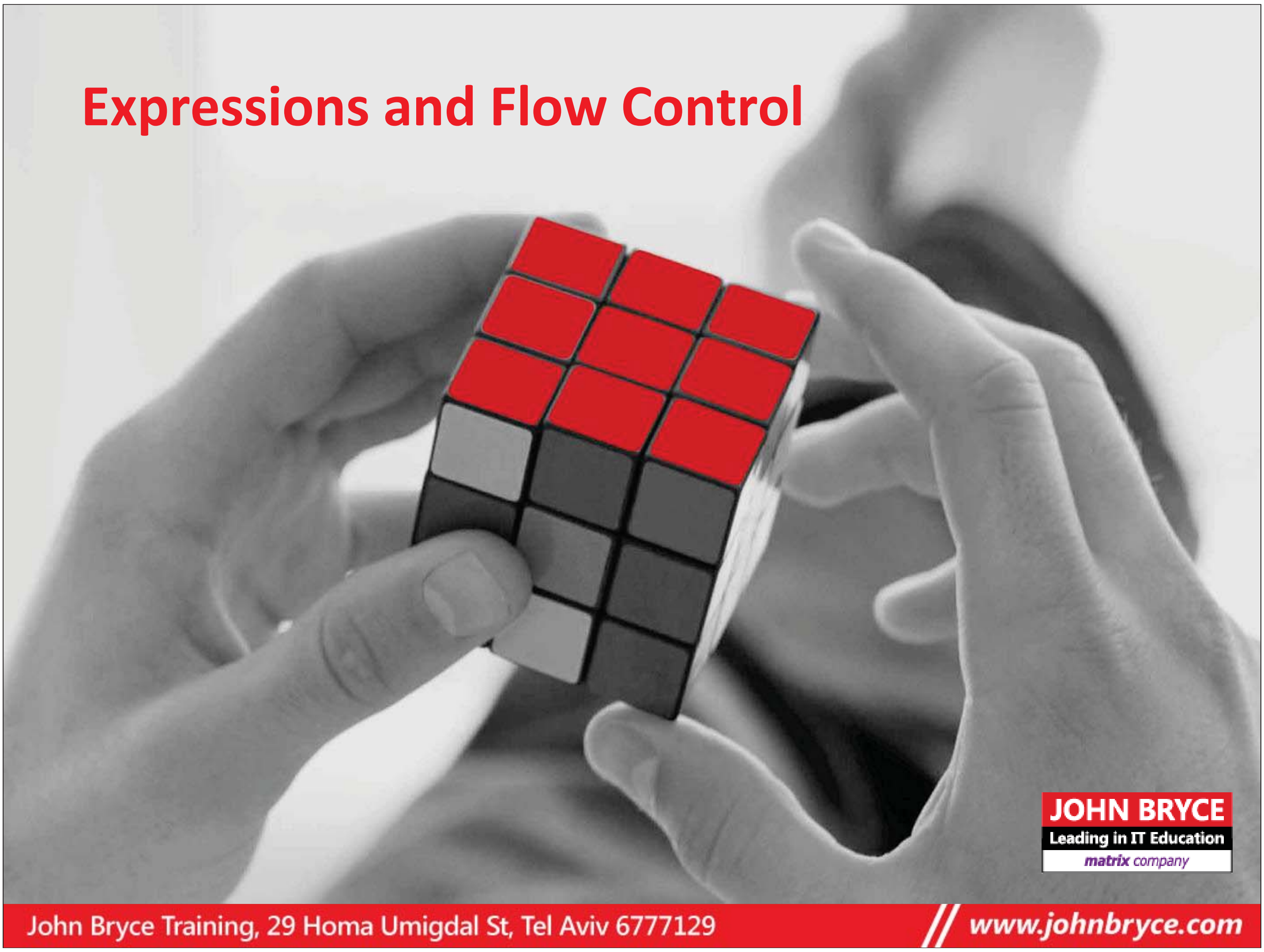
```
MIN_VAL
```

First program with eclipse

Summary

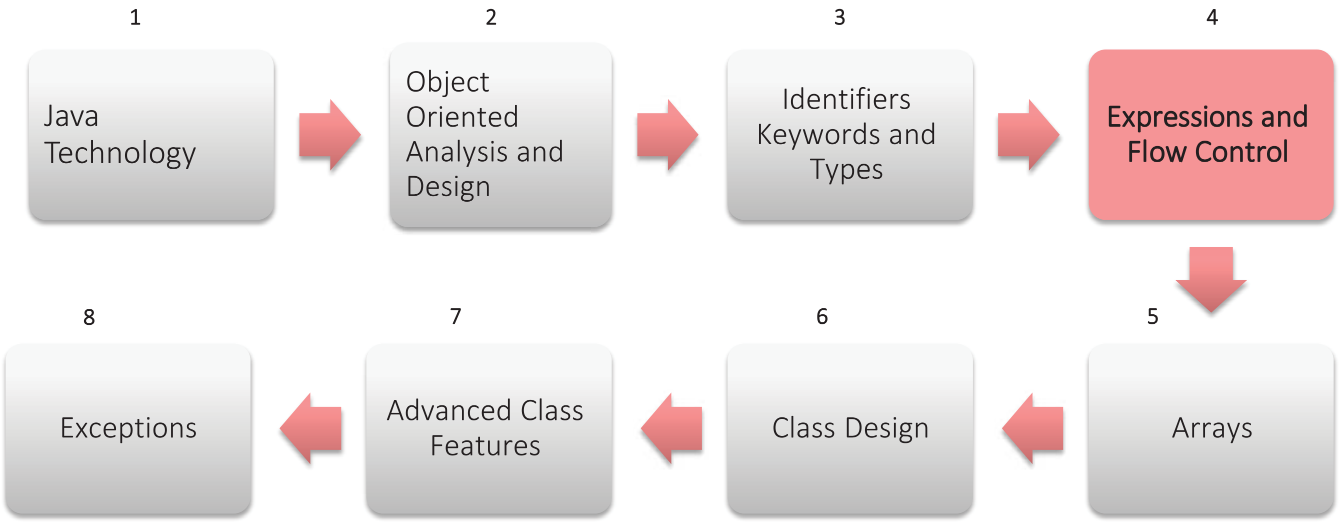
- Basic syntax
  - Statements, blocks
  - Comments
- Primitive Type
- Reference Type

Expressions and Flow Control



John Bryce Training, 29 Homa Umigdal St, Tel Aviv 6777129 // www.johnbryce.com

Objectives



1 Java Technology

2 Object Oriented Analysis and Design

3 Identifiers Keywords and Types

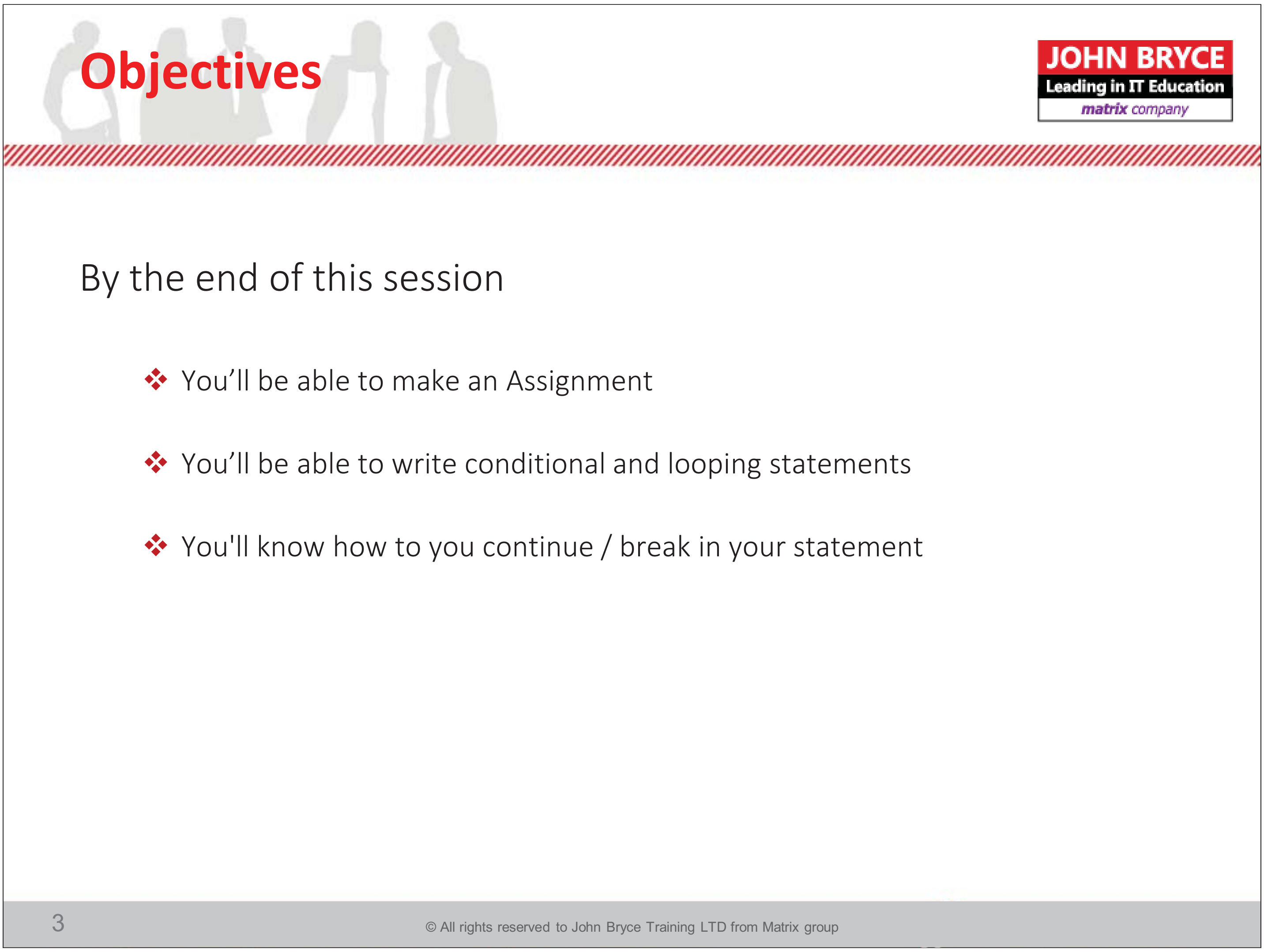
4 Expressions and Flow Control

5 Arrays

6 Class Design

7 Advanced Class Features

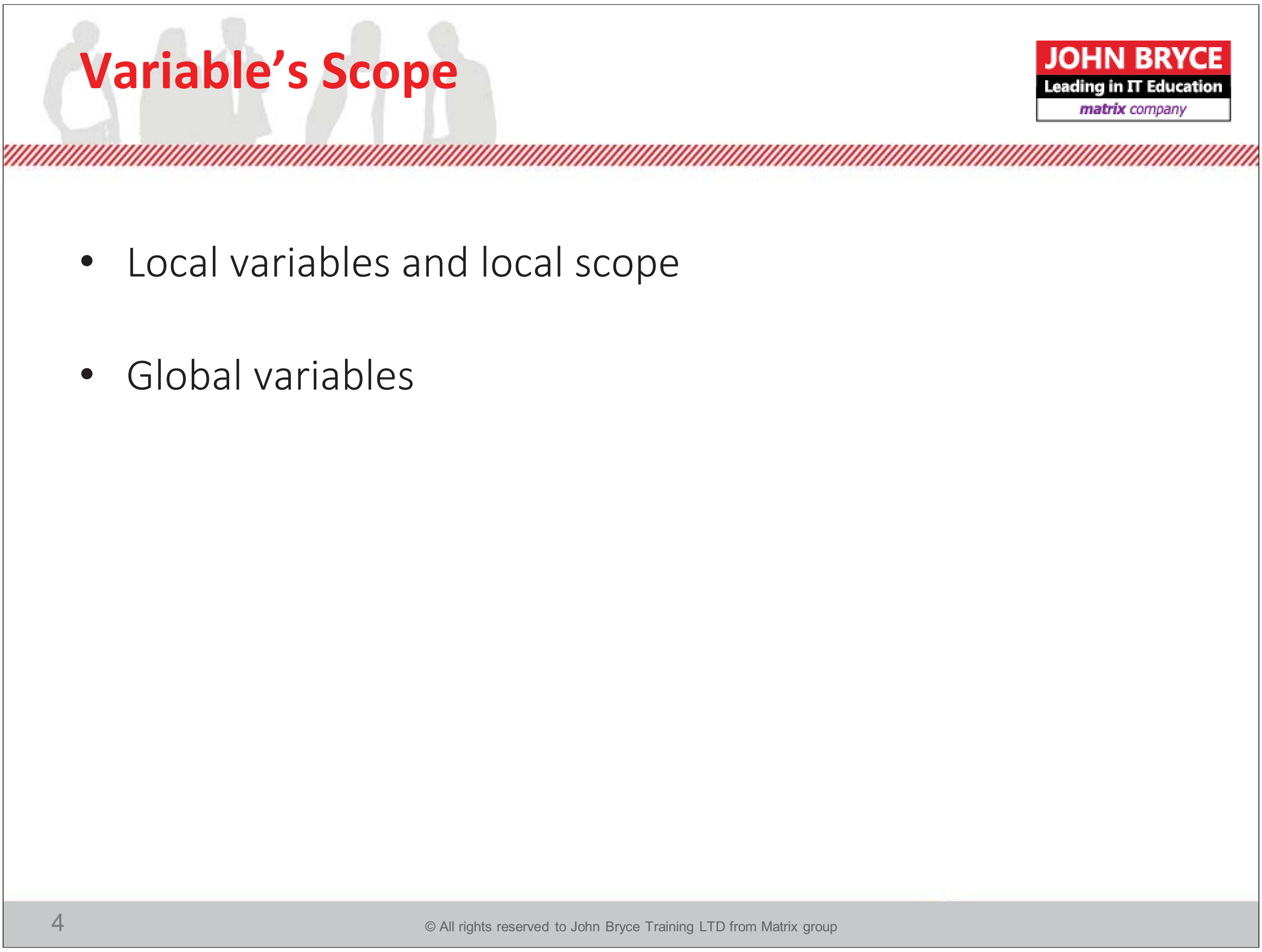
8 Exceptions



Objectives

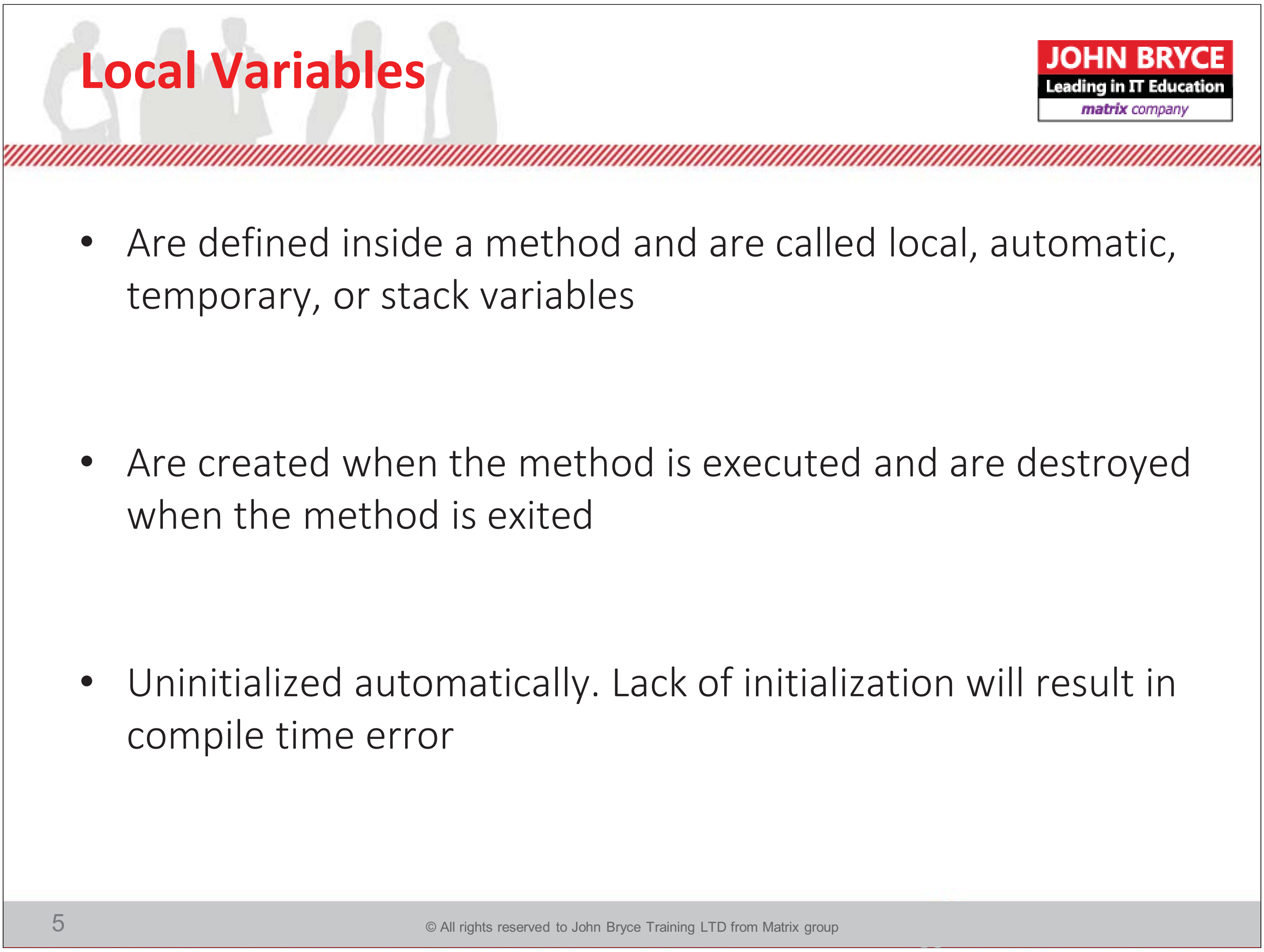
By the end of this session

- You’ll be able to make an Assignment
- You’ll be able to write conditional and looping statements
- You'll know how to you continue / break in your statement



Variable’s Scope

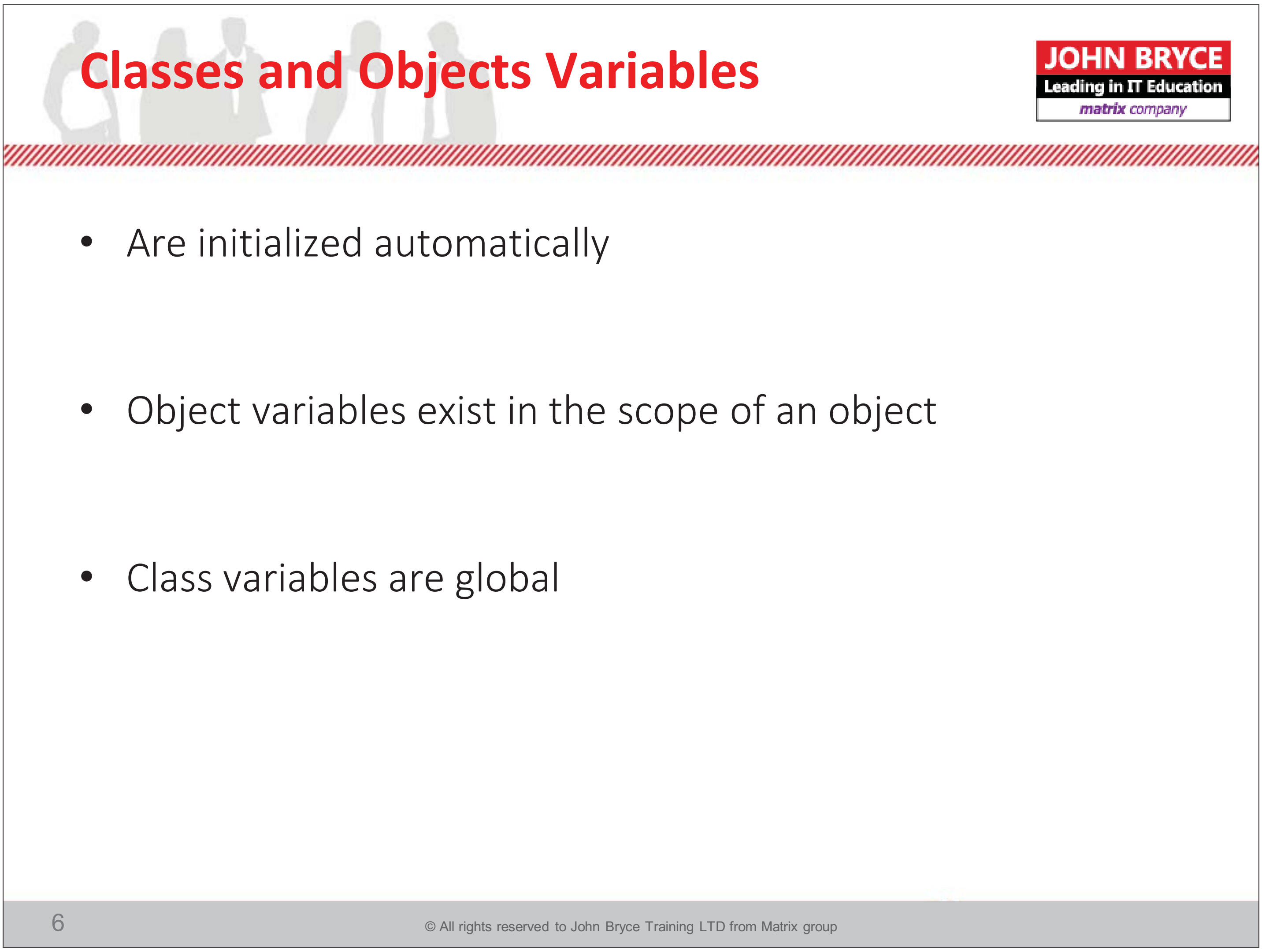
- Local variables and local scope
- Global variables



Local Variables

JOHN BRYCE Leading in IT Education matrix company

- Are defined inside a method and are called local, automatic, temporary, or stack variables
- Are created when the method is executed and are destroyed when the method is exited
- Uninitialized automatically. Lack of initialization will result in compile time error



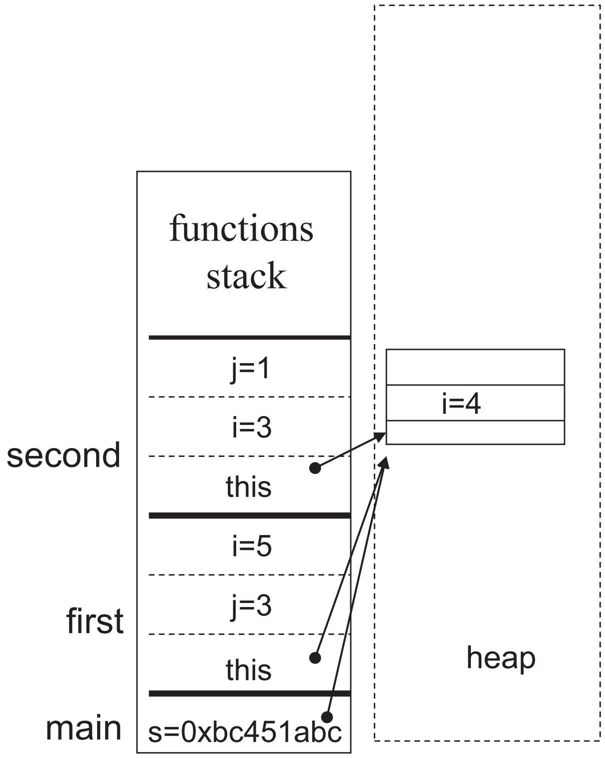
Classes and Objects Variables

JOHN BRYCE Leading in IT Education matrix company

- Are initialized automatically
- Object variables exist in the scope of an object
- Class variables are global

Scope Example

```
public class Scope {
    private int i=9;
    public void first()    {
        int j=3,i;
        i=5;
        this.i=i+j;
        second(j);
    }
    public void second (int i)    {
        int j=1;
        this.i=i+j;
    }
    public static void main(String args[])    {
        Scope s=new Scope();
        s.first();
    }
}
```



Precedence of Operators

Associative	Operators
R to L	++ -- + - ~ ! (data type)
L to R	* / %
L to R	+ -
L to R	<< >> >>>
L to R	< > <= >= instanceof
L to R	== !=
L to R	&
L to R	^
L to R	|
L to R	&&
L to R	||
R to L	?:
R to L	= *= /= %= += -= <<= >>= >>>= &= ^= |=

Logical Operators

NOT - !   OR - || AND - &&

Bits Operators

OR - | XOR - ^   AND - &

ONE'S COMPLEMENT - ~

Bitwise Logical Operators

Example

```
                                0 0 1 0 1 1 0 1
~  0 1 0 0 1 1 1 1           &  0 1 0 0 1 1 1 1
------------------           ------------------
   1 0 1 1 0 0 0 0              0 0 0 0 1 1 0 1


   0 0 1 0 1 1 0 1              0 0 1 0 1 1 0 1
^  0 1 0 0 1 1 1 1           |  0 1 0 0 1 1 1 1
------------------           ------------------
   0 1 1 0 0 0 1 0              0 1 1 0 1 1 1 1
```

Shift Operators

>> - Signed right shift.
>>> - Unsigned right shift.
<< - Left shift.

Syntax:

num >> no_of_right_shifts

num >>> no_of_right_shifts

num << no_of_left_shifts

Example:

$30>>4 = 30/2^4$

$30<<4 = 30*2^4$

$-30>>>4 = 30/2^4$

Shifts Examples

1357 =	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	0	1	1	0	1
-1357 =	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	1	0	1	1	0	0	1	1
1357 >> 5 =	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0
-1357 >> 5 =	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	1	0	1
1357 >>> 5 =	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0
-1357 >>> 5 =	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	1	0	1
1357 << 5 =	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	0	1	1	0	1	0	0	0	0	0
-1357 << 5 =	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	1	0	1	1	0	0	1	1	0	0	0	0	0

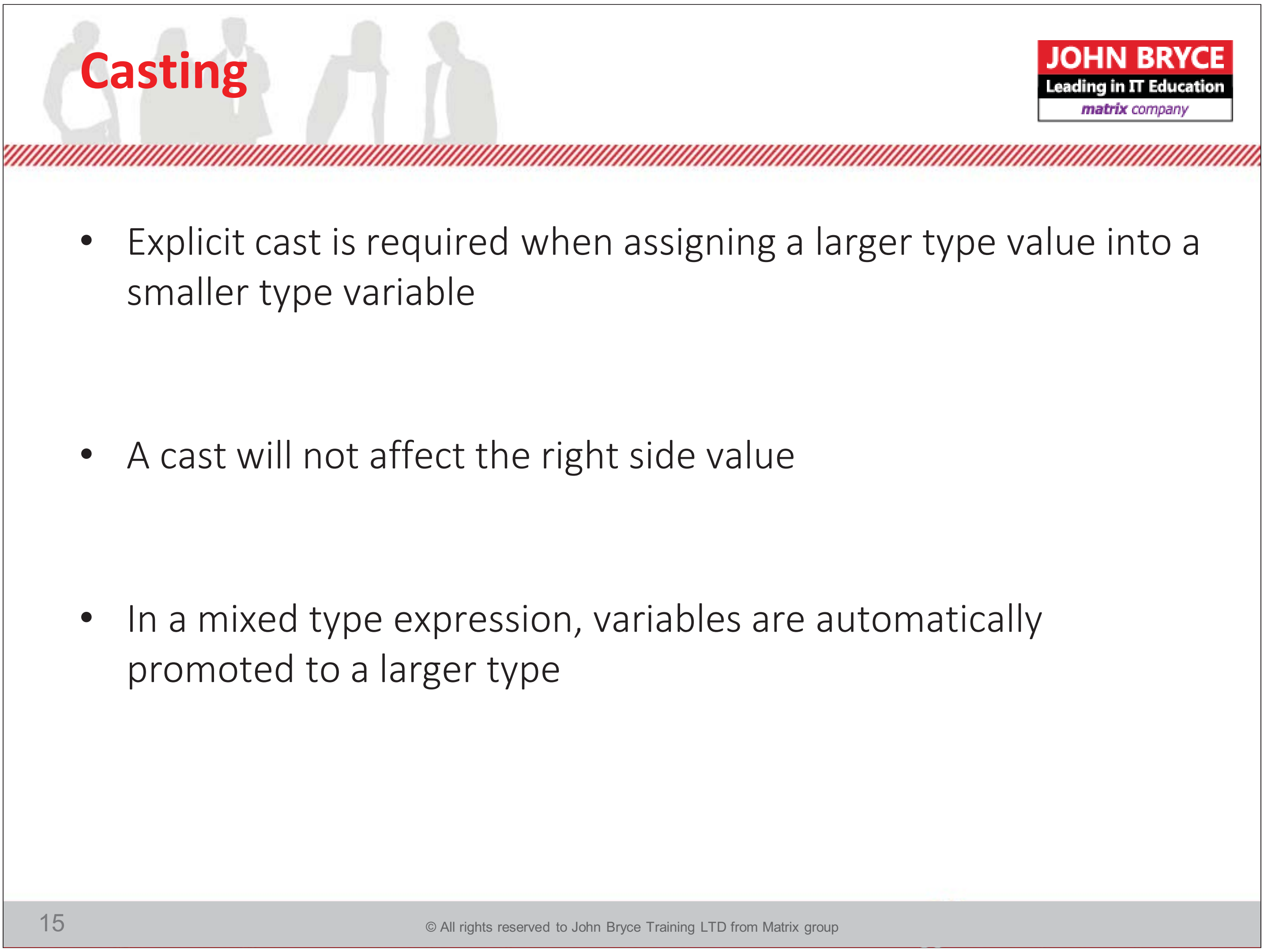
String Concatenation

- String objects are concatenated using the ‘+’ operator
- One of the objects must be a string and the other one will be converted automatically to a string
- An object may be cast to a string using the toString( ) method (will be explained in a later module)

String Concatenation

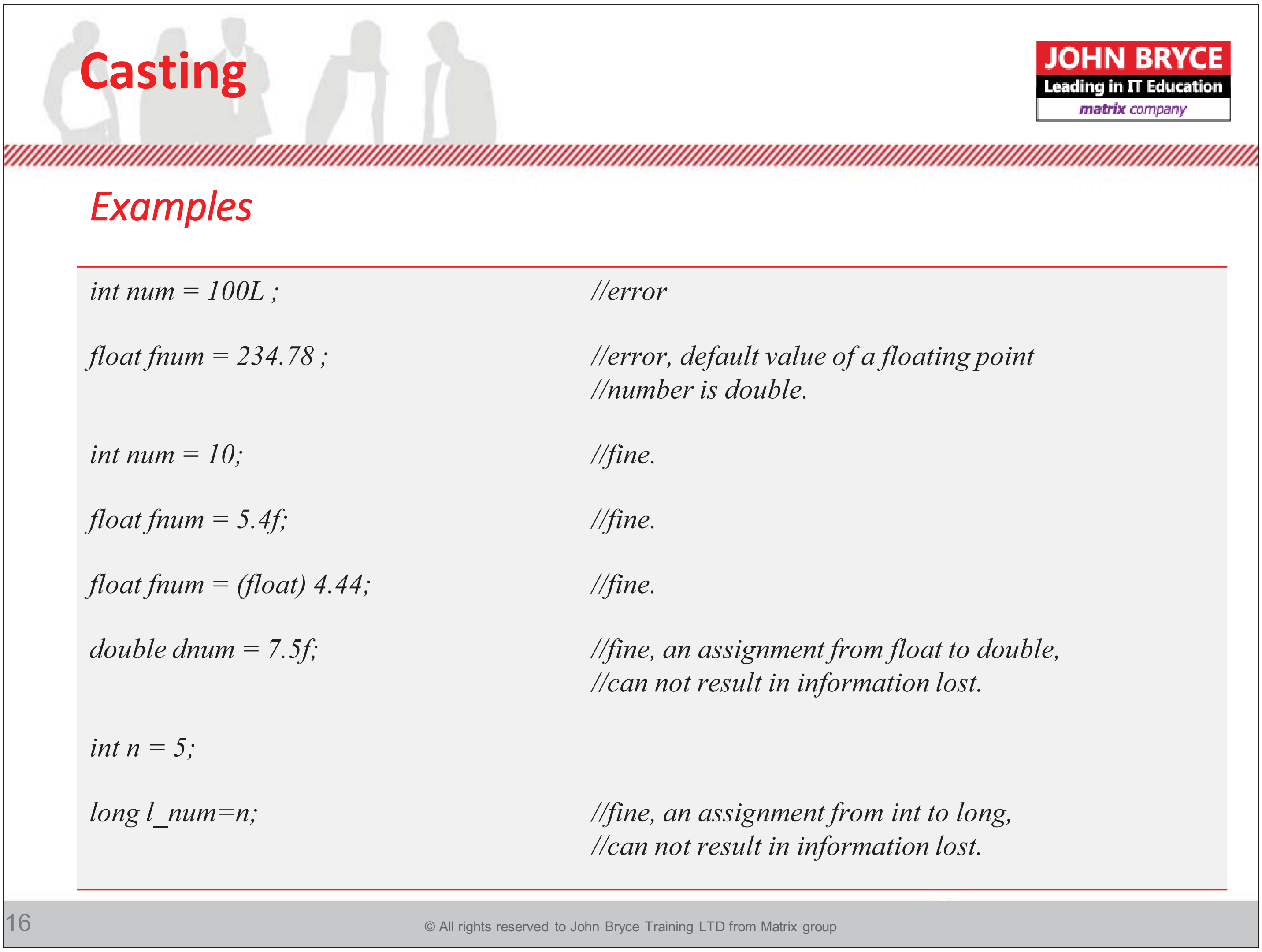
Example

```
String s = “Hello ”;

String name= “kuky”;

int num = 3;

s = s+”to “+name+” and his “+num+” dogs.”;

System.out.println(s);

            /* “Hello to kuky and his 3 dogs.”

            will be printed.  */
```



Casting

- Explicit cast is required when assigning a larger type value into a smaller type variable
- A cast will not affect the right side value
- In a mixed type expression, variables are automatically promoted to a larger type



Casting

Examples

```
int num = 100L ;              //error

float fnum = 234.78 ;         //error, default value of a floating point
                              //number is double.

int num = 10;                 //fine.

float fnum = 5.4f;            //fine.

float fnum = (float) 4.44;    //fine.

double dnum = 7.5f;           //fine, an assignment from float to double,
                              //can not result in information lost.

int n = 5;

long l_num=n;                 //fine, an assignment from int to long,
                              //can not result in information lost.
```

Branching Statements - *If*

An *if* statement:

```
if (boolean expression) {
                                        statement or block;
                           }
```

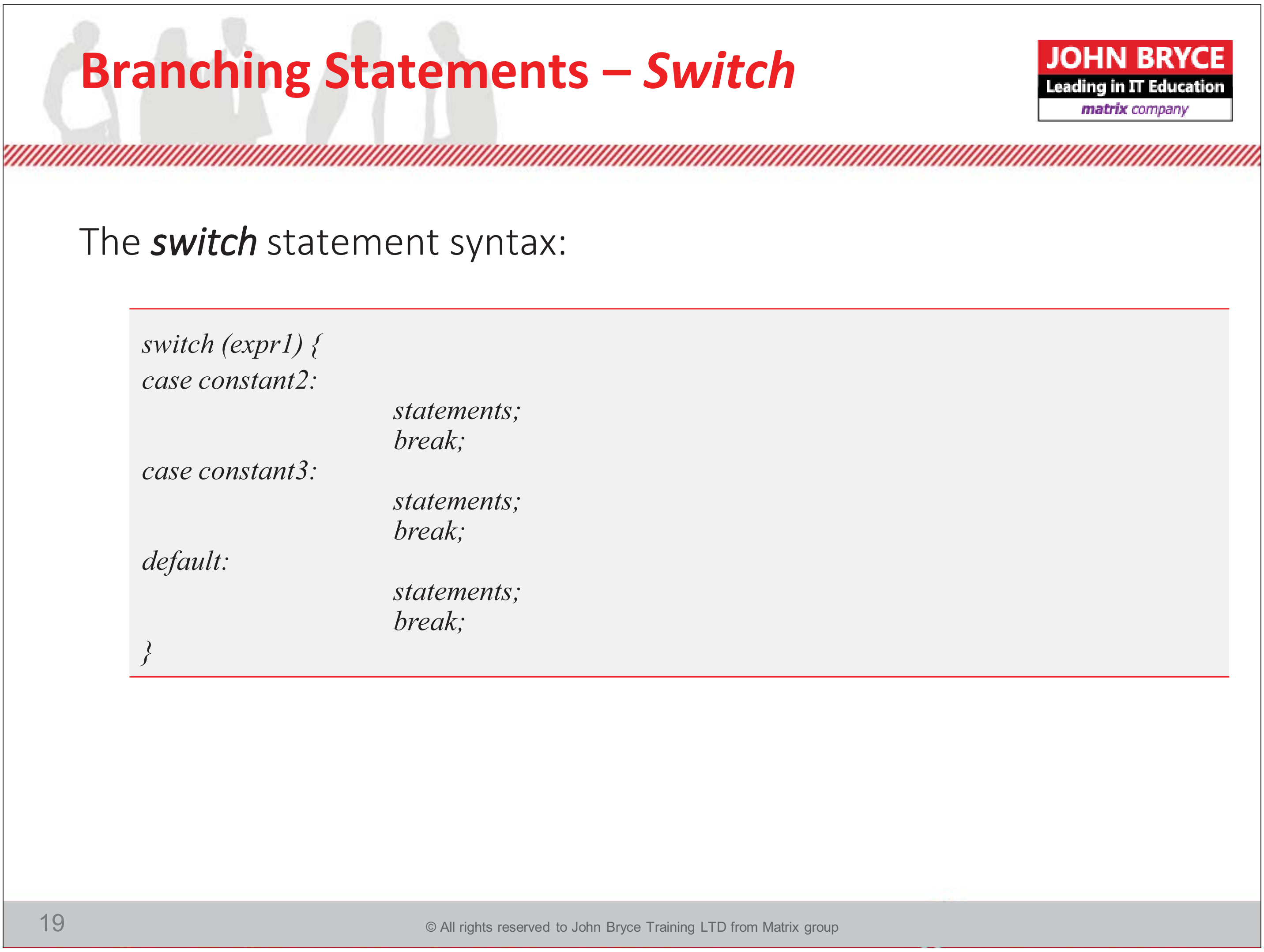
An *if-else* statement:

```
if ( boolean expression) {
                                        statement or block;
                           }
                           else {
                                        statement or block;
                           }
```

An *If-Else*

Example

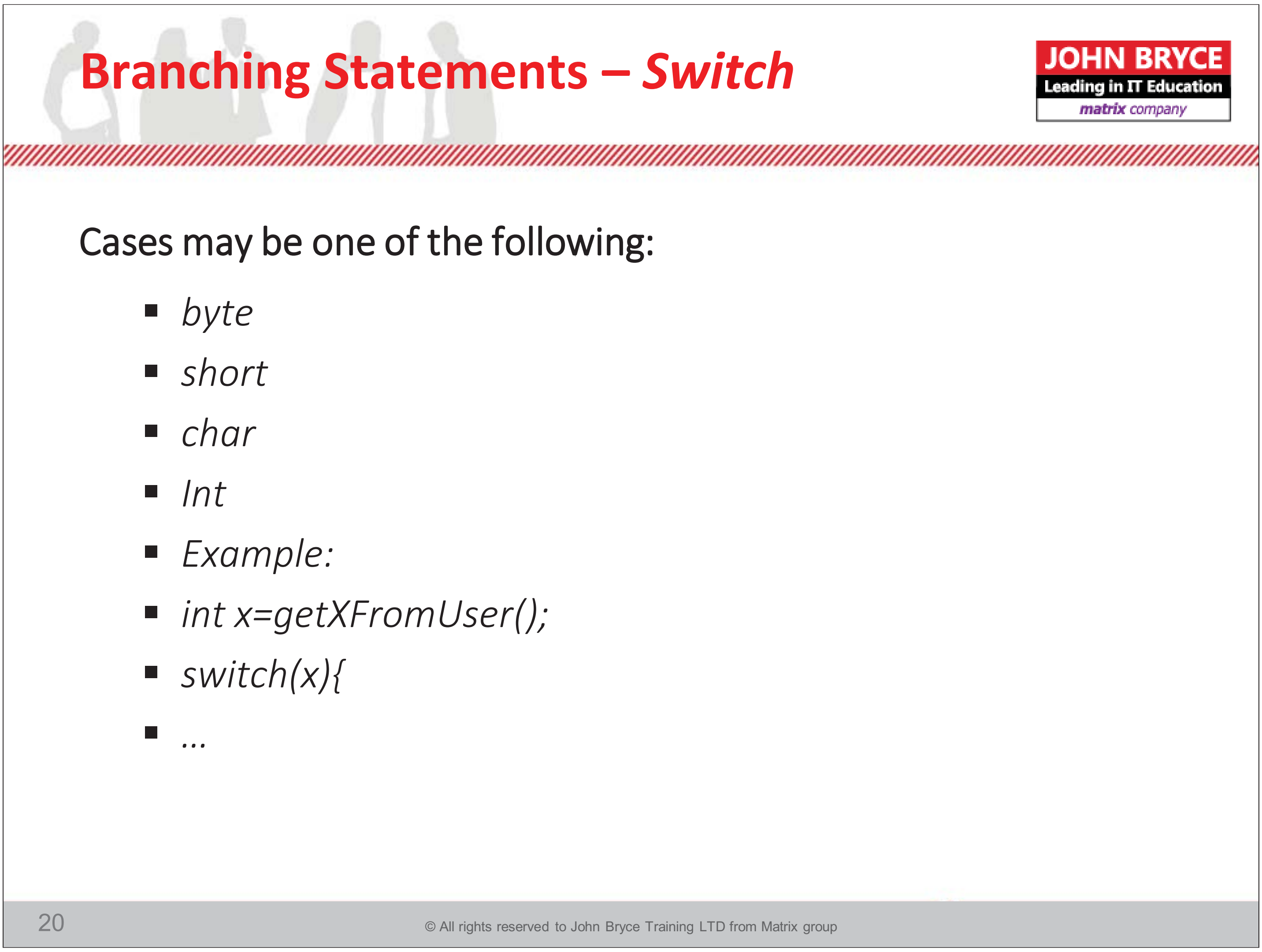
```
If (no_of_student > MAX_STUDENTS)
             openAnotherClass( );
else if (no_of_student < MIN_STUDENTS)
  closeClass( );
else
             enterClass( );
```



Branching Statements – *Switch*

The ***switch*** statement syntax:

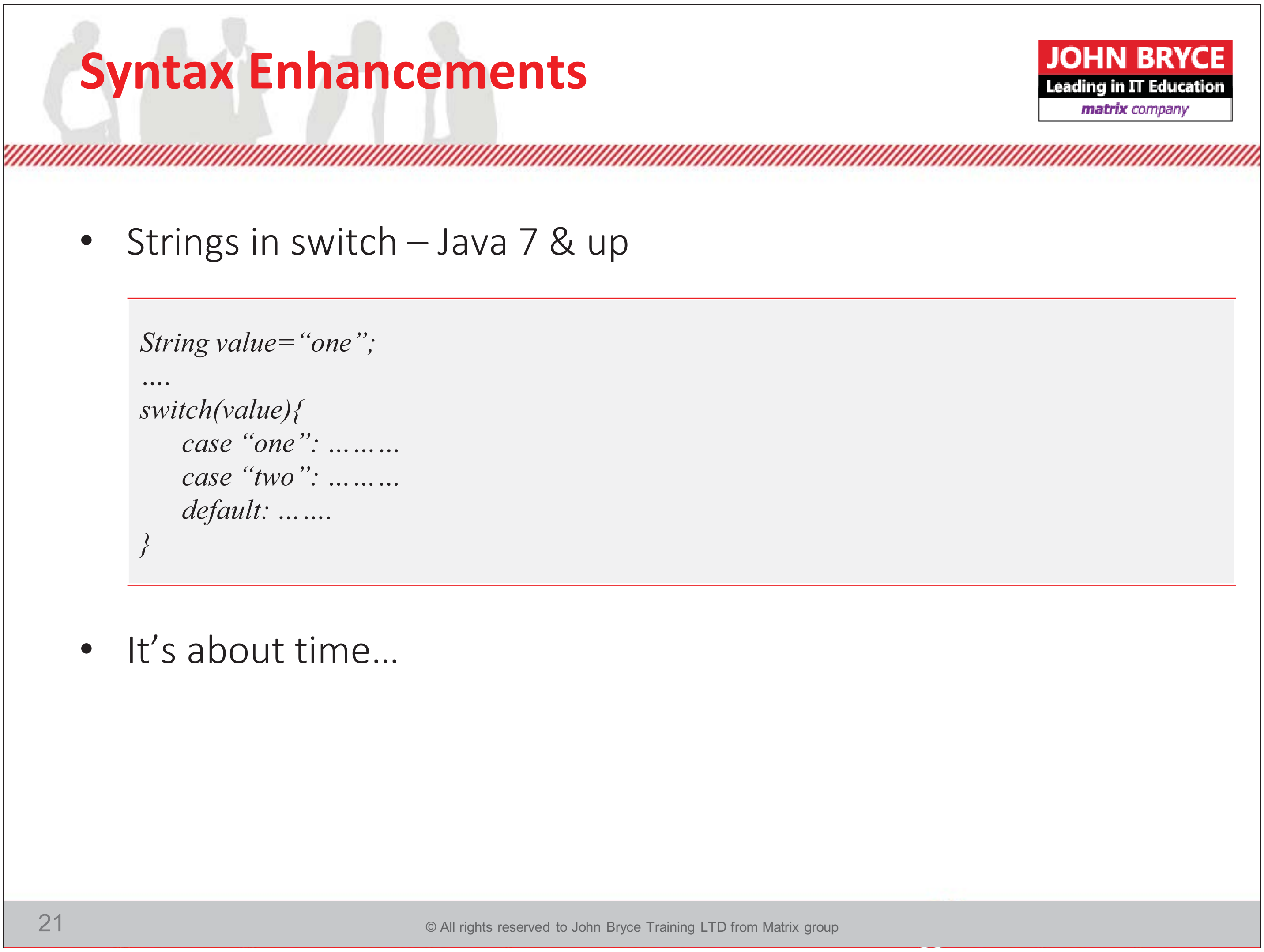
```
switch (expr1) {
case constant2:
                    statements;
                    break;
case constant3:
                    statements;
                    break;
default:
                    statements;
                    break;
}
```



Branching Statements – *Switch*

Cases may be one of the following:

- *byte*
- *short*
- *char*
- *Int*
- *Example:*
- *int x=getXFromUser();*
- *switch(x){*
- *…*



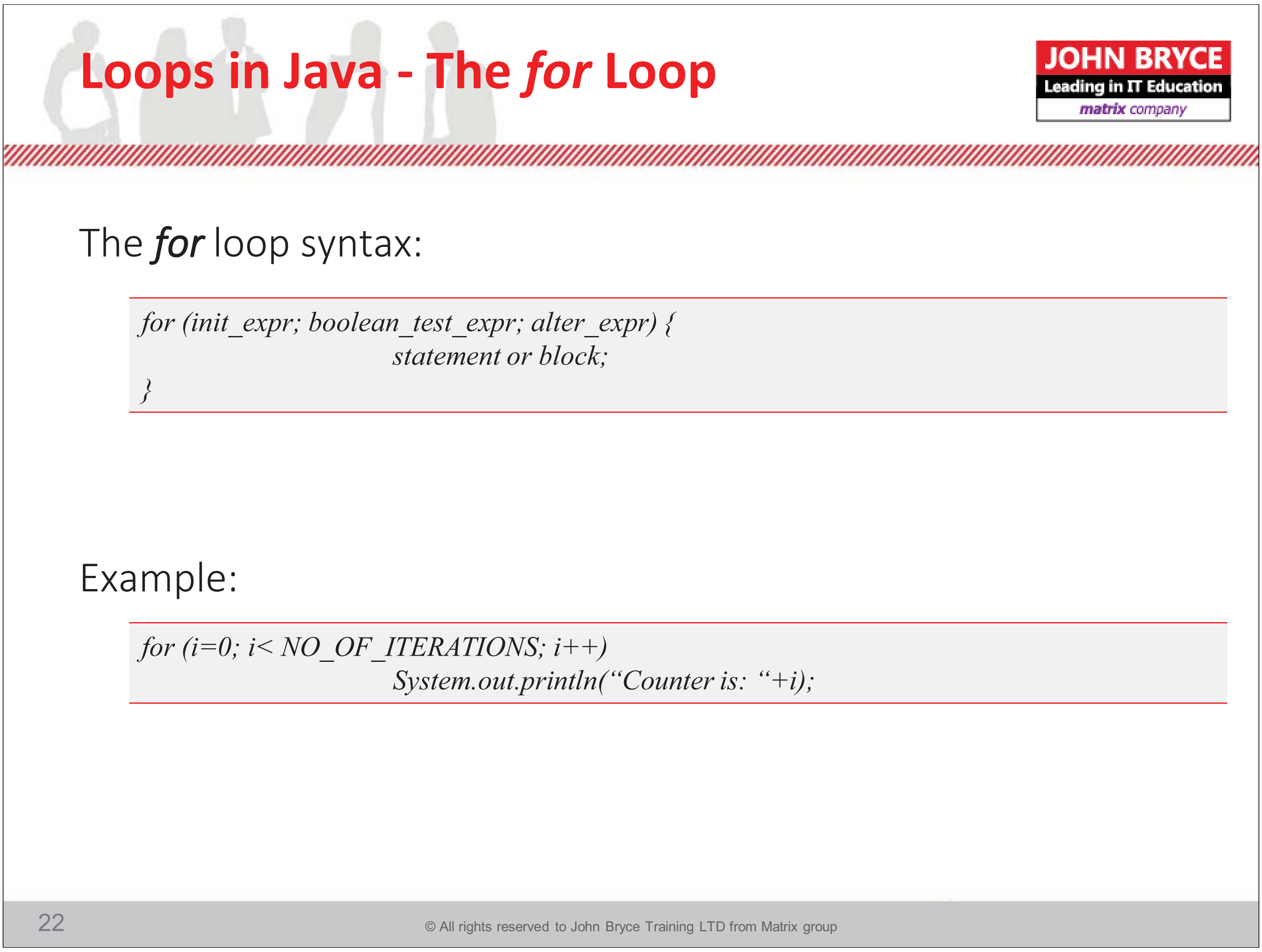
Syntax Enhancements

JOHN BRYCE Leading in IT Education matrix company

- Strings in switch – Java 7 & up

```
String value="one";
....
switch(value){
    case "one": ………
    case "two": ………
    default: …….
}
```

- It's about time…



Loops in Java - The *for* Loop

JOHN BRYCE Leading in IT Education matrix company

The ***for*** loop syntax:

```
for (init_expr; boolean_test_expr; alter_expr) {
                        statement or block;
}
```

Example:

```
for (i=0; i< NO_OF_ITERATIONS; i++)
                        System.out.println("Counter is: "+i);
```

Loops in Java -The *while* Loop

The ***while*** loop syntax:

```
while (boolean_test_expr) {
                              statement or block ;
   }
```

Example:

```
int i=0;
while ( i< NO_OF_ITERATIONS) {
                              System.out.println("Counter is:  "+i);
                              i++;
}
```

Loops in Java -The *do-while* Loop

The ***do-while*** loop syntax:

```
do {
                              statement or block ; }
  while (boolean_test_expr) ;
```

Example:

```
int i=0;
do {
                              System.out.println("Counter is:  "+i);
                              i++; }
while ( i< NO_OF_ITERATIONS) ;
```

Loop Flow Control

- break [label];
- continue [label];
- label: loop;

Loop Flow Control

break statement

will lead to permanently exit from the loop (“break” the loop)

continue statement

will lead to exit the current iteration and to continue the flow of the loop from the beginning of the next iteration

label statement

identifies any valid statement to which the control must be transferred

- For a ***break*** statement, a valid label may be any legal statement
- For a ***continue*** statement, a valid label must identify a loop

The *break* statement

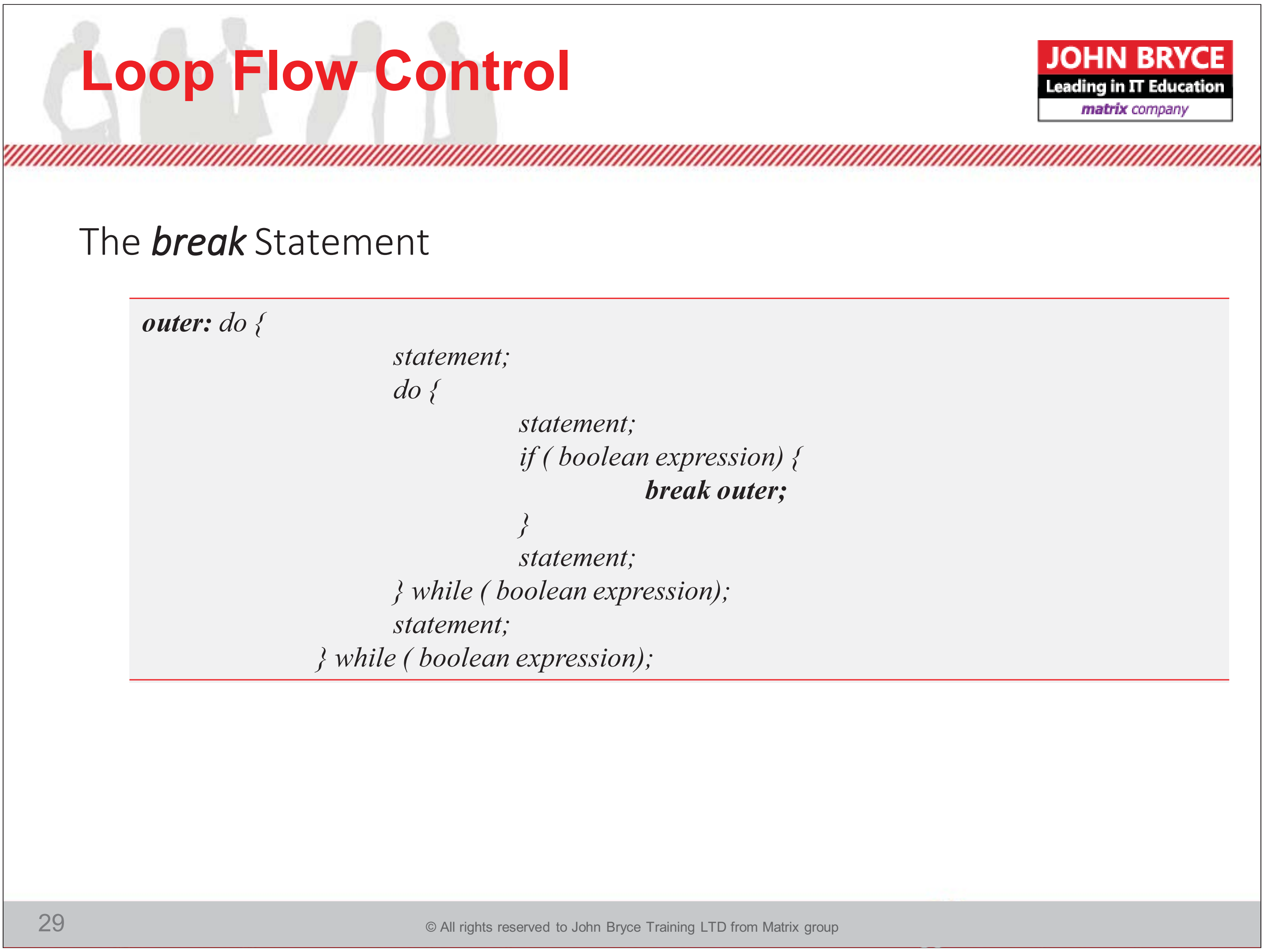
The ***break*** statement syntax:

```
do {
            statement;
            if ( condition is true) {
                        break;
            }
            statement;
} while ( boolean expression);
```

The *continue* statement

The ***continue*** statement syntax:

```
do {
            statement;
            if ( condition is true) {
                        continue;
            }
            statement;
} while ( boolean expression);
```

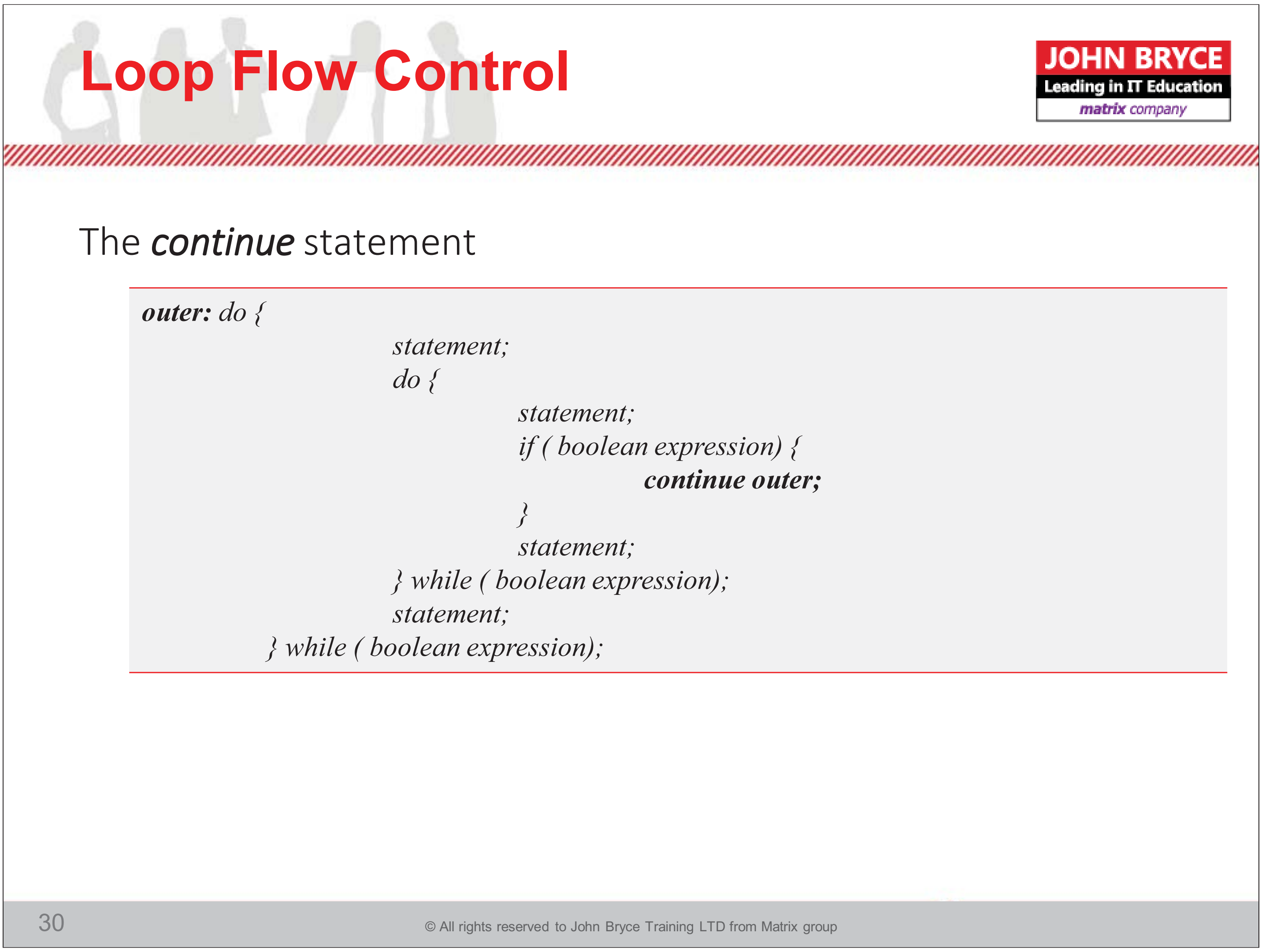


Loop Flow Control

JOHN BRYCE
Leading in IT Education
matrix company

The *break* Statement

```
outer: do {
            statement;
            do {
                  statement;
                  if ( boolean expression) {
                        break outer;
                  }
                  statement;
            } while ( boolean expression);
            statement;
        } while ( boolean expression);
```



Loop Flow Control

JOHN BRYCE
Leading in IT Education
matrix company

The *continue* statement

```
outer: do {
            statement;
            do {
                  statement;
                  if ( boolean expression) {
                        continue outer;
                  }
                  statement;
            } while ( boolean expression);
            statement;
      } while ( boolean expression);
```

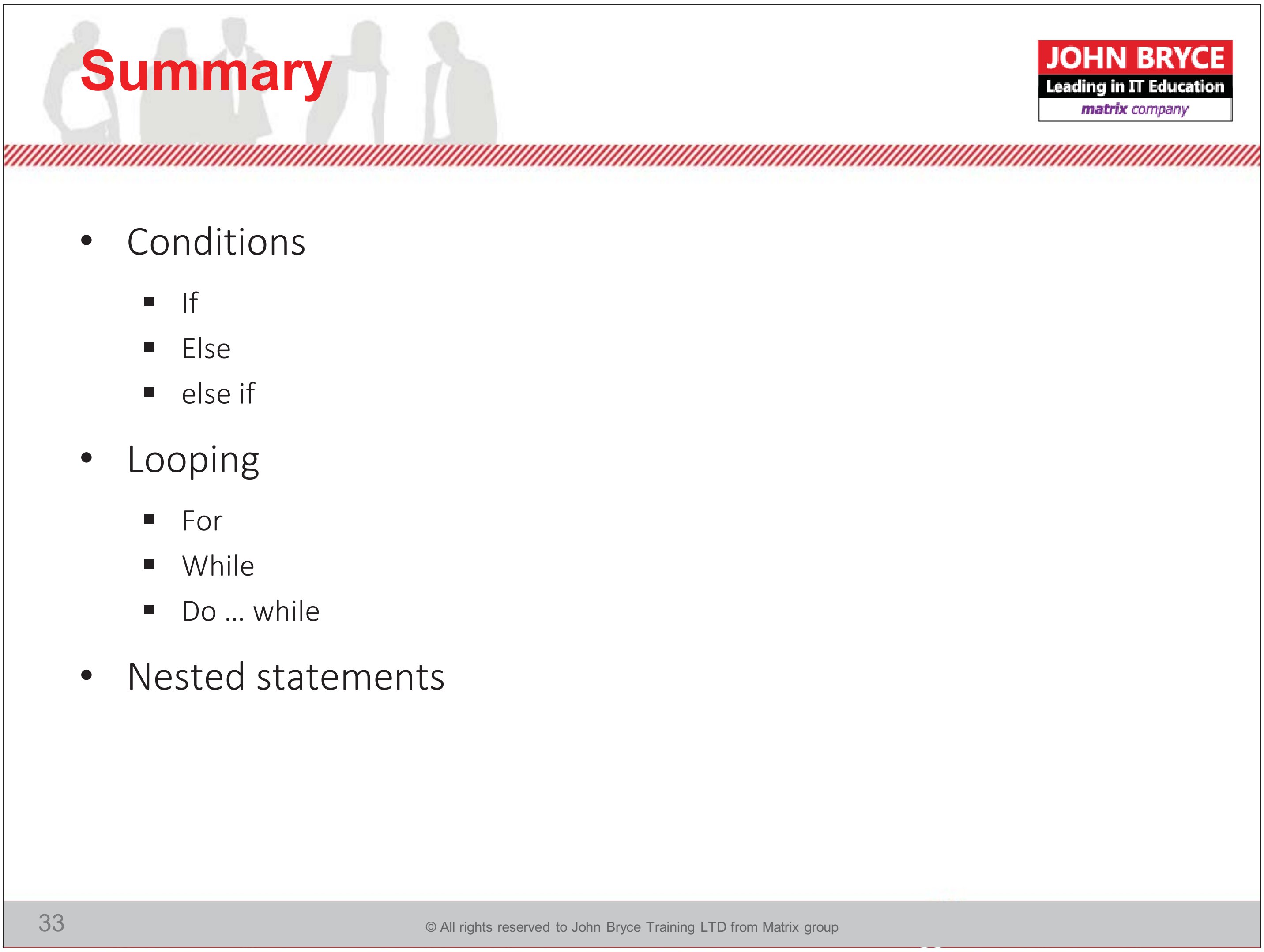
For-Each Loop for Arrays

Array of objects or primitives

```
public int sumArray (int[] nums){
        int sum=0;
        for (int i : nums)
                sum+=i;
        return sum;
}
```

```
public double concat (String[] words){
        String sentence=“”;
        for (String curr : words)
                sentence+=curr+” “;
        return sentence;
}
```

Conditions and loops



Summary

JOHN BRYCE
Leading in IT Education
matrix company

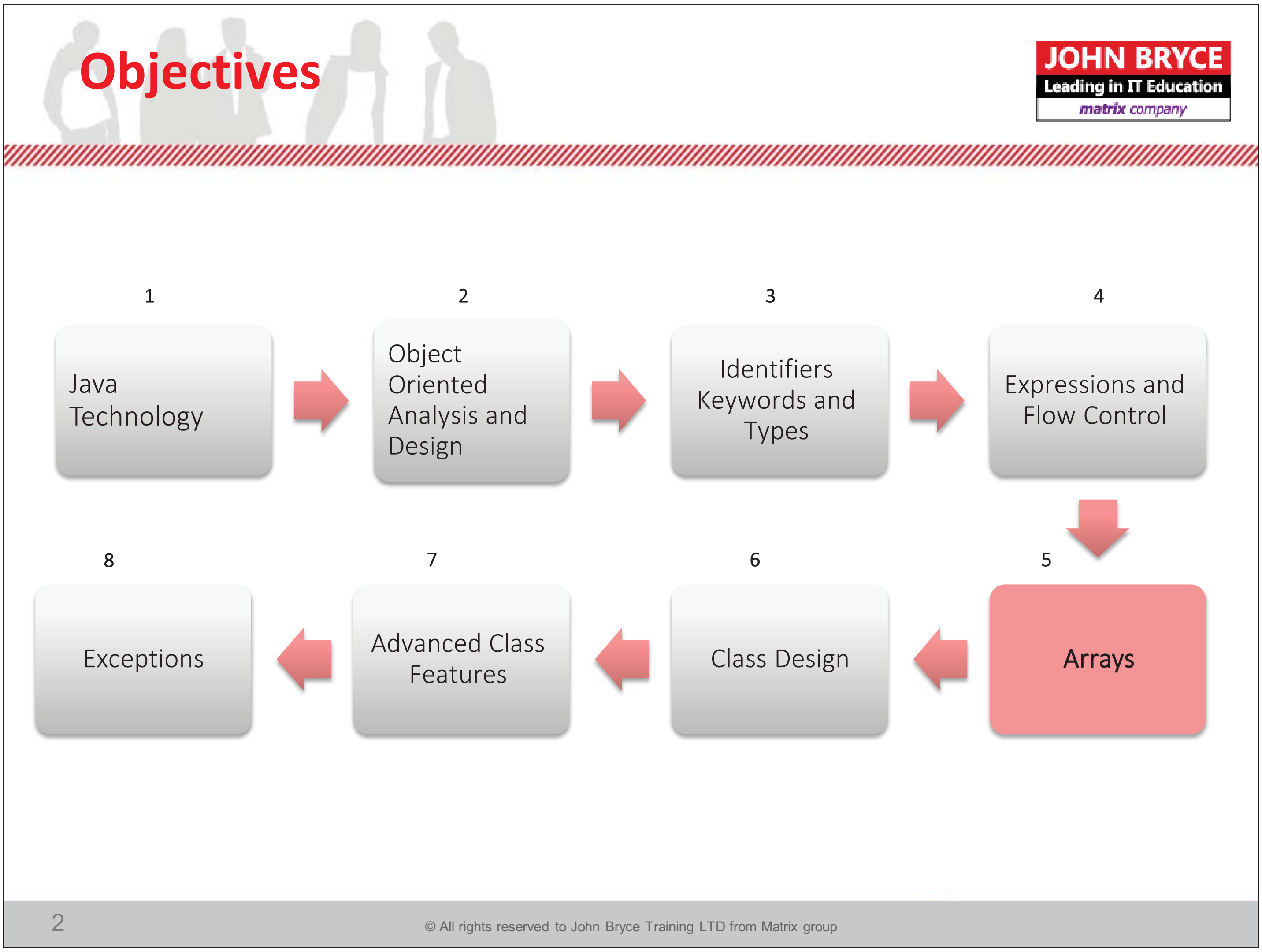
- Conditions
  - If
  - Else
  - else if
- Looping
  - For
  - While
  - Do … while
- Nested statements

33

Arrays

JOHN BRYCE
Leading in IT Education
matrix company

John Bryce Training, 29 Homa Umigdal St, Tel Aviv 6777129 // www.johnbryce.com



Objectives

JOHN BRYCE
Leading in IT Education
matrix company

1
Java Technology
2
Object Oriented Analysis and Design
3
Identifiers Keywords and Types
4
Expressions and Flow Control
5
Arrays
6
Class Design
7
Advanced Class Features
8
Exceptions

Objectives

By the end of this session

- ❖ You’ll be able to use arrays
- ❖ You’ll be able to use multidimensional arrays
- ❖ You’ll be able to user varargs

Java Arrays

- Arrays group objects or primitives of the same type together
- In Java, an array is an object
- Memory for the array reference is allocated on the stack
- Memory for the array object is allocated dynamically on the heap

Declaring Arrays References

An array reference is declared as follows:

```
element_type  arr_ref_name [ ] ;
```

OR:

```
element_type[]  arr_ref_name;
```

Examples:

```
char c_arr [ ] ;
```

```
Point p_arr [ ];            //p_arr is a null reference to an array
                            //of references to objects of
                            //class point.
```

```
Box boxArray [ ] ;
```

Creating Arrays

An array, like any other object, is instantiated using the *new* keyword

Examples:

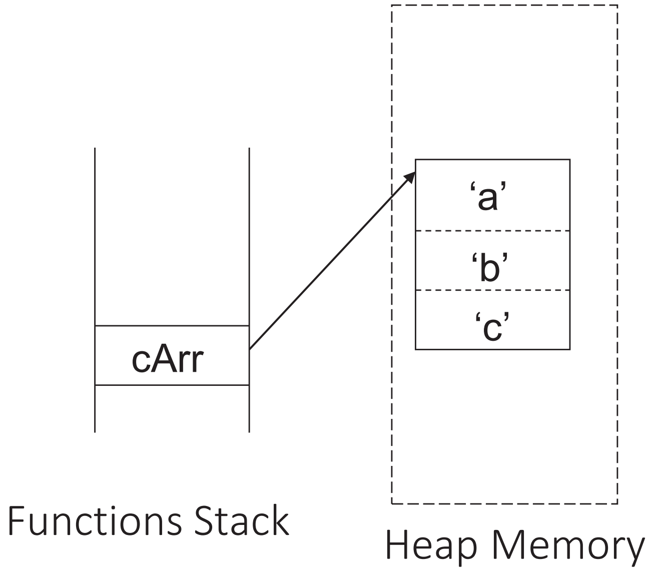
```
            char c_arr[ ] = new char[100];  //This will create an (array)
                                            //object that holds 100 chars.
  int i_arr[ ] ;
  …
  i_arr = new int [MAX];
```

```
Point pArr[ ] = new Point [ 200 ];          //This will create an (array)
                                            //object that holds 200 null
                                            //references to objects of
                                            //class Point.
```

Primitives Arrays

Memory allocation

```
char cArr [ ] = new char [3] ;
cArr[0]='a';
cArr[1]='b';
cArr[2]='c';
```

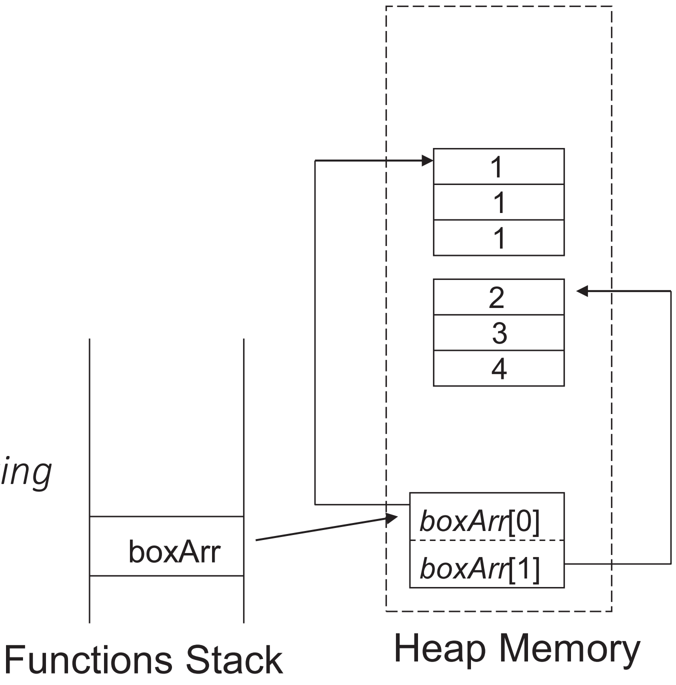


Arrays of Objects

Memory Allocation

```
Box boxArr [ ] = new Box [2] ;
boxArr[0]= new Box(1,1,1);
boxArr[1]= new Box(2,3,4);
```

Note: boxArr is a reference to an array of 2 references, each pointing at another Box object.



Arrays Initialization

JOHN BRYCE
Leading in IT Education
matrix company

Arrays may be initialized during declaration or assigned values after it

```
String names[ ] =new String[3];
names[0]=new String ("John");
names[1]=new String("Bryce");
names[2]=new String("Levy");

String names[ ] ={
  “John”,
  “Bryce”,
 “Levy”
};
```

```
Box boxes[ ] =new Box[3];
boxes[0]=new Box (10,20,10);
boxes[1]=new Box (3,5,13);
boxes[2]=new Box (8,6,11);

Box boxes[ ] ={
 new Box (10,20,10),
 new Box (3,5,13),
 new Box (8,6,11)
};
```

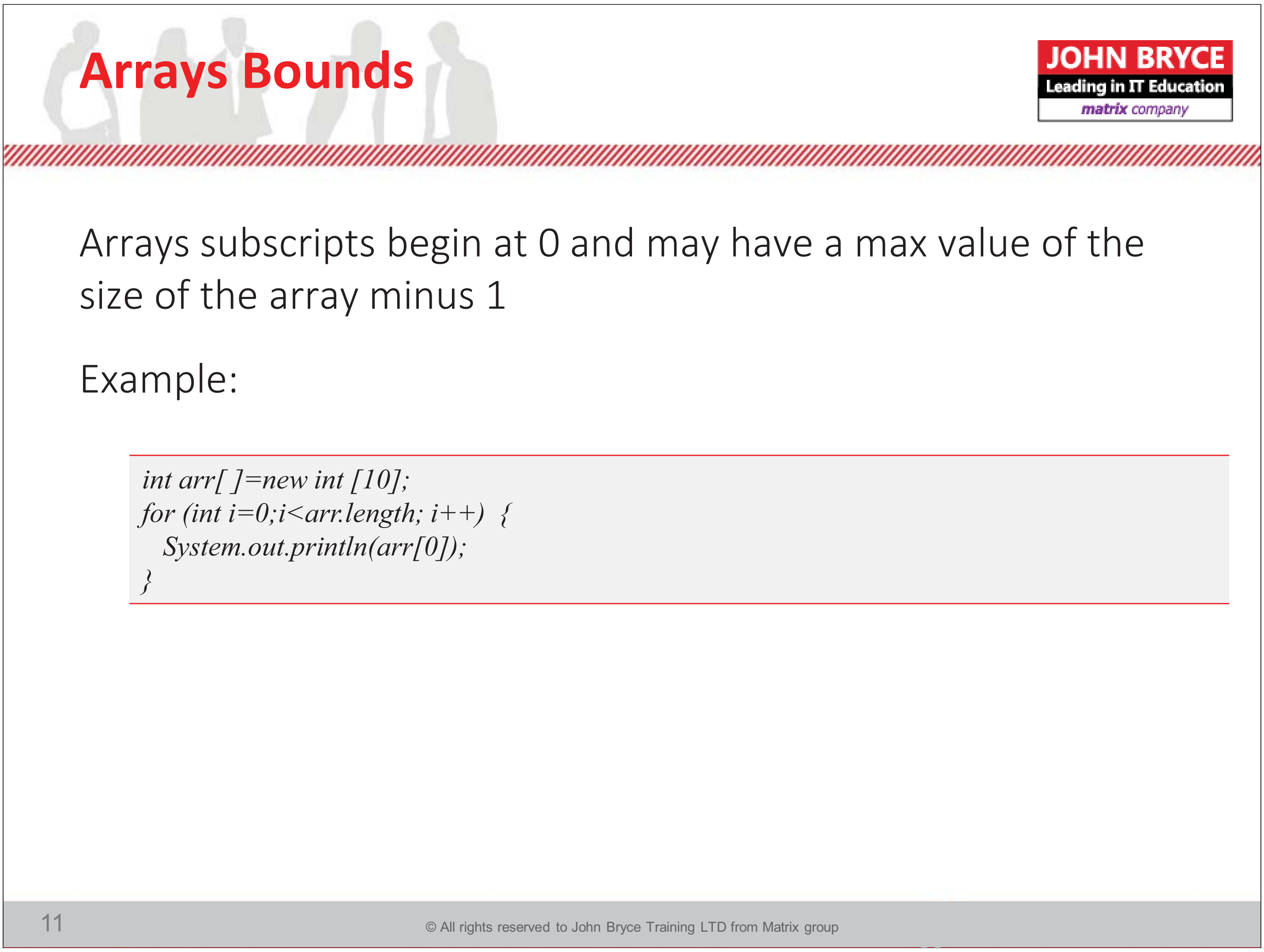
MultiDimensional Arrays

JOHN BRYCE
Leading in IT Education
matrix company

Array of arrays

Example:

```
short twoDim[ ][ ];
                      twoDim = new short[4][ ];        //twoDim is a reference to
                                                       //array of four elements (each
                                                       //element is of type array of short).
                      twoDim[0]=new short[9];
                      twoDim[1]=new short[3];

                      short twoDim[ ][ ] = new short [ ] [9] ;       //illegal
                      twoDim = new short[4][5];                      //legal
```

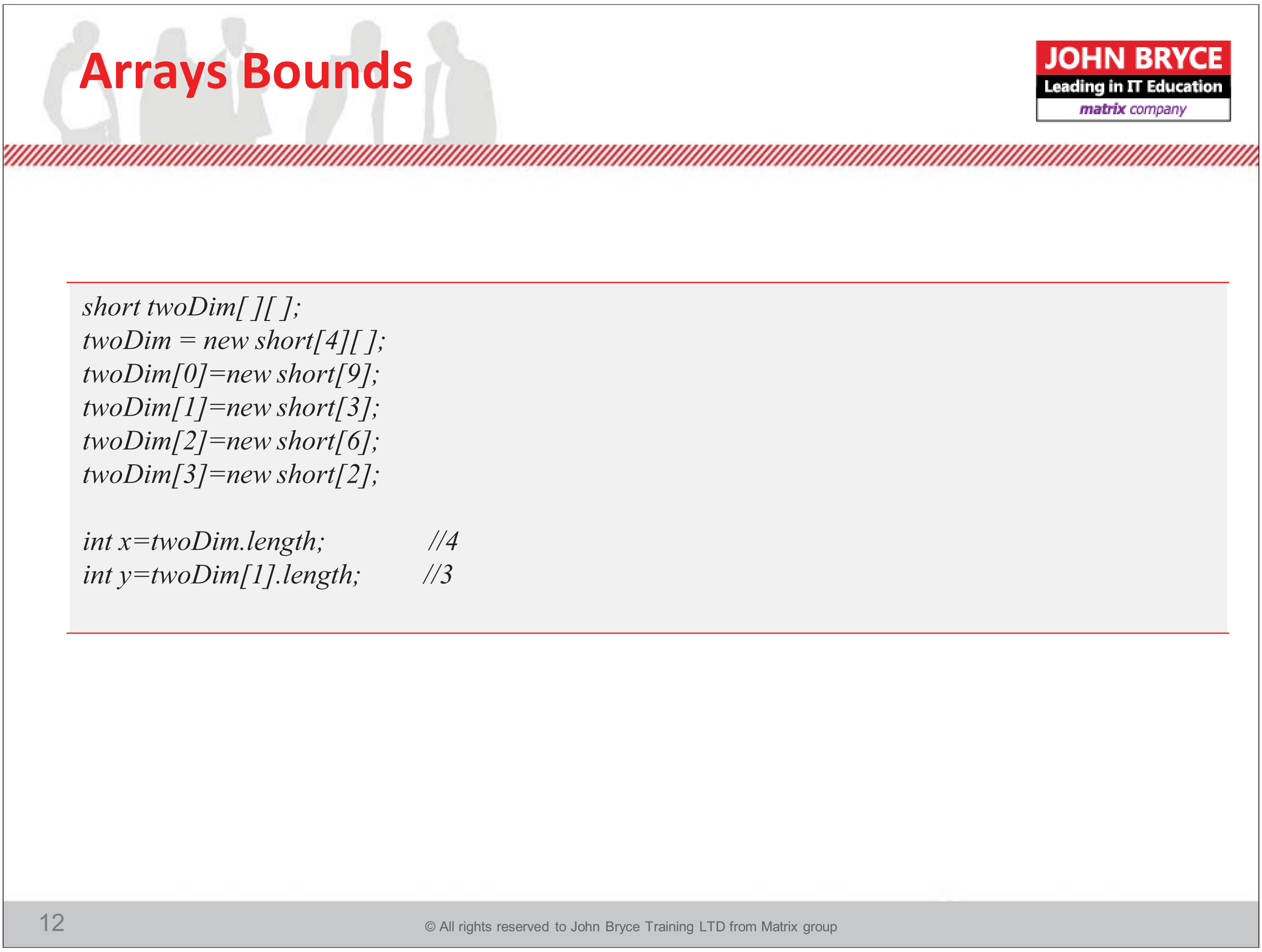


Arrays Bounds

Arrays subscripts begin at 0 and may have a max value of the size of the array minus 1

Example:

```
int arr[ ]=new int [10];
for (int i=0;i<arr.length; i++)  {
   System.out.println(arr[0]);
}
```



Arrays Bounds

```
short twoDim[ ][ ];
twoDim = new short[4][ ];
twoDim[0]=new short[9];
twoDim[1]=new short[3];
twoDim[2]=new short[6];
twoDim[3]=new short[2];

int x=twoDim.length;              //4
int y=twoDim[1].length;         //3
```

Array Resizing

An array can not be resized

The same array reference may be reinitialized to another array

Example:

```
int arr[ ]=new int [10];
    …
    arr = new int [4];            //unless another reference to the
                                  //first array exist elsewhere, the first
                                  //array is lost and may be garbage
                                  //collected.
```

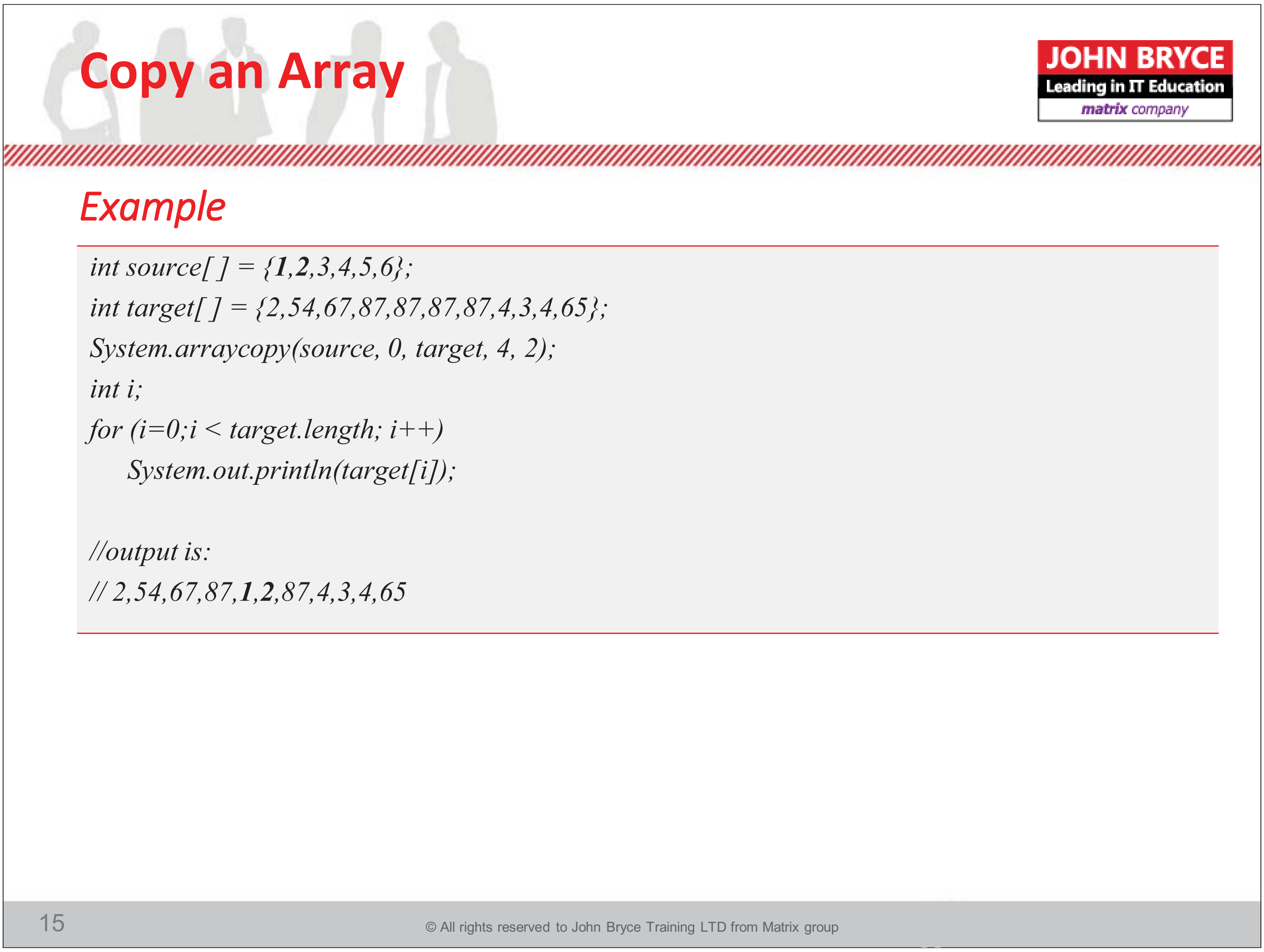
Copy an Array

Use System.arraycopy(. . .) method to copy arrays

Syntax:

```
System.arraycopy( sourceArr,src_starting_ind, target, target_starting_ind, sourceArr.length);
```

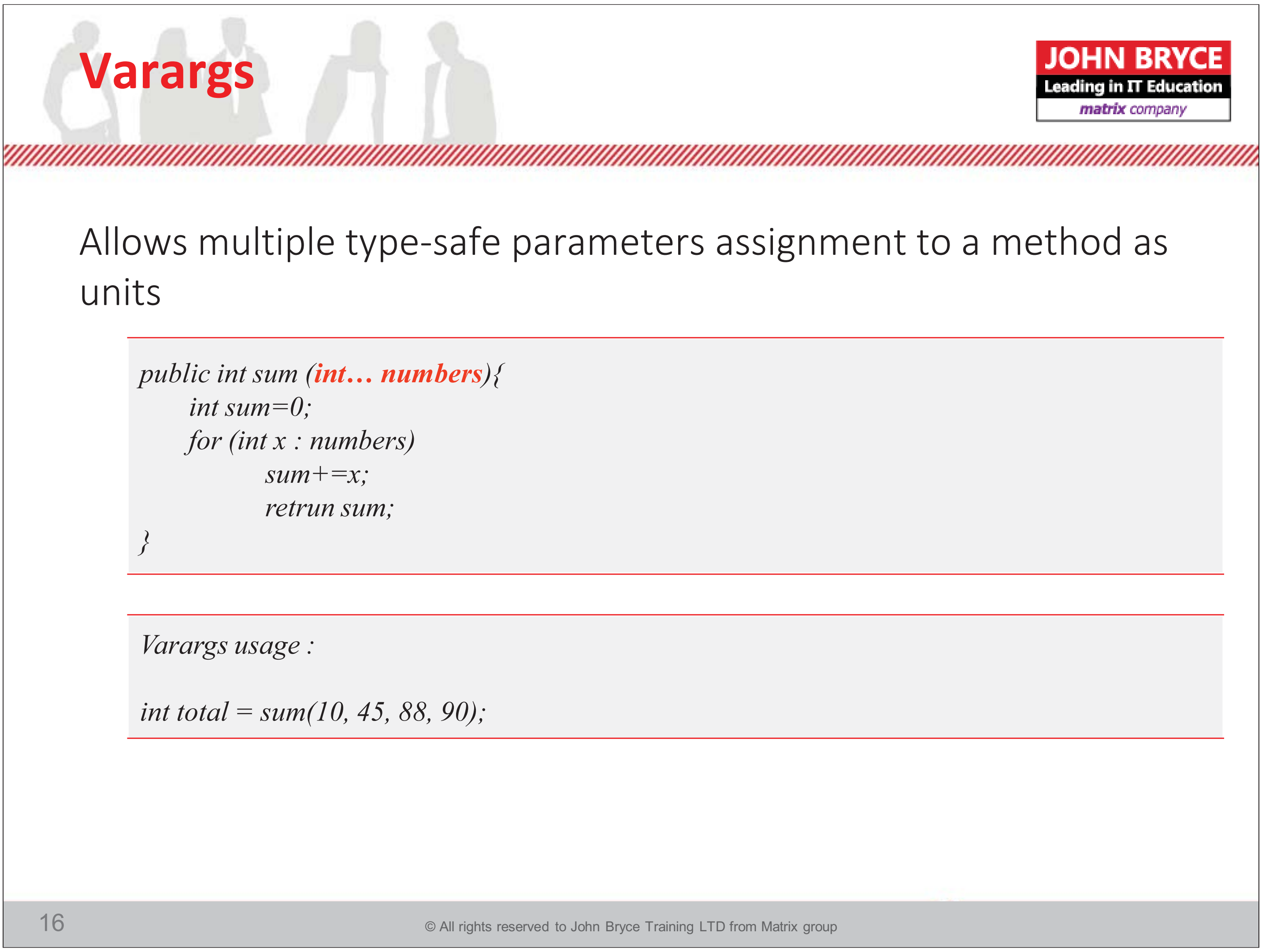
Note: System.arraycopy( ) copies primitives or references, not objects



Copy an Array

Example

```
int source[ ] = {1,2,3,4,5,6};
int target[ ] = {2,54,67,87,87,87,87,4,3,4,65};
System.arraycopy(source, 0, target, 4, 2);
int i;
for (i=0;i < target.length; i++)
    System.out.println(target[i]);

//output is:
// 2,54,67,87,1,2,87,4,3,4,65
```



Varargs

Allows multiple type-safe parameters assignment to a method as units

```
public int sum (int… numbers){
        int sum=0;
        for (int x : numbers)
                sum+=x;
                retrun sum;
}
```

```
Varargs usage :

int total = sum(10, 45, 88, 90);
```

Varargs

Method overloading issue

- Varargs equals to an array

  Therefore:

  Cannot be overloaded with a method that takes an array

  If it is not the only parameter – varargs must be the last one

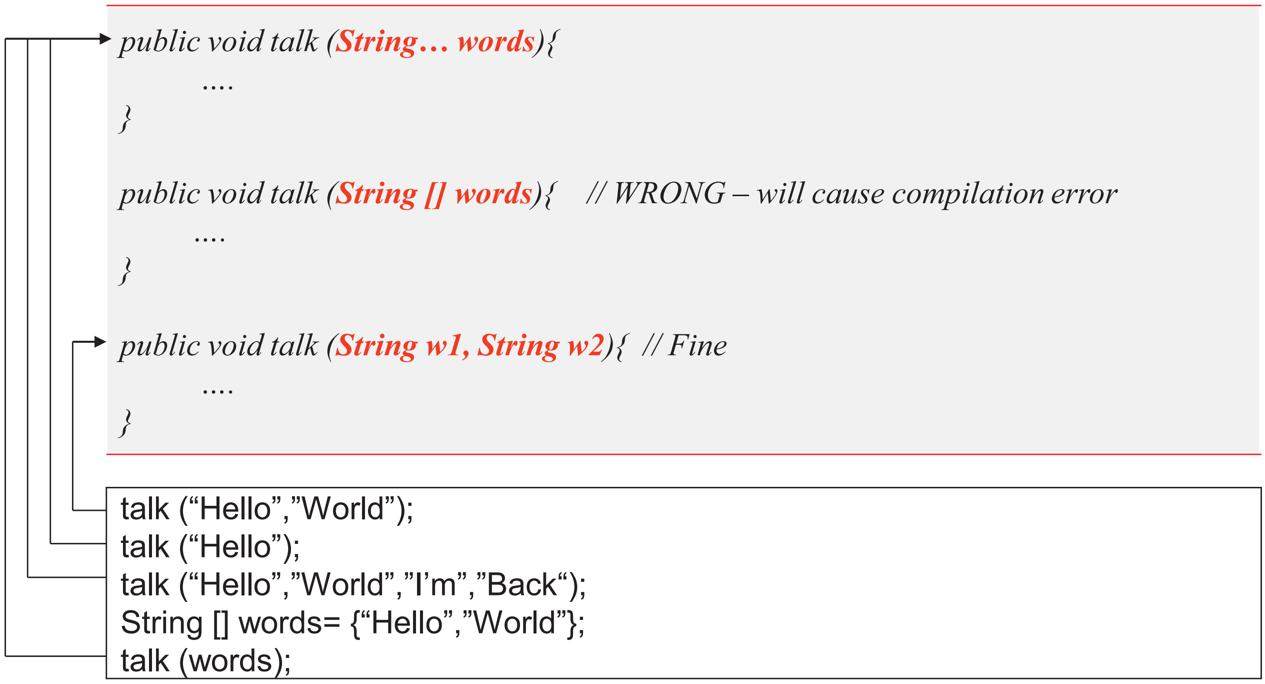
  Arrays can be also assigned as a varargs

  main method – new look:

```
public static void main (String… args){
        …..
}
```

Varargs

Examples:



```
public void talk (String… words){
        ….
}

public void talk (String [] words){    // WRONG – will cause compilation error
        ….
}

public void talk (String w1, String w2){  // Fine
        ….
}
```

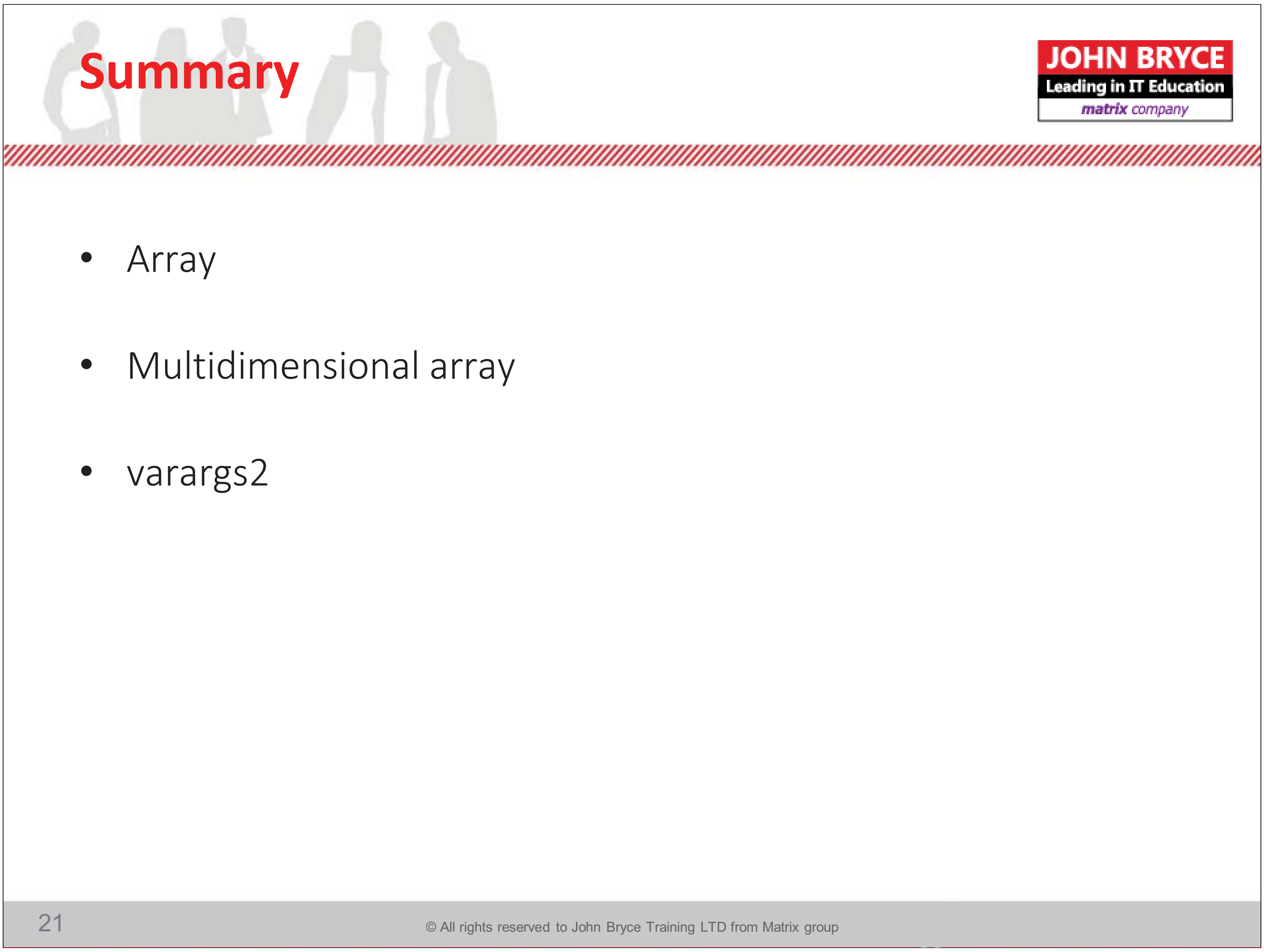
```
talk (“Hello”,”World”);
talk (“Hello”);
talk (“Hello”,”World”,”I’m”,”Back“);
String [] words= {“Hello”,”World”};
talk (words);
```

Varargs

Examples

```
public void talk (String… words){
        ….
}

public void talk (int x, String word, String… words){  // Fine
        ….
}

public void talk (String… words, String word){  // WRONG  - will cause compilation error
        ….
}
```

Arrays



Summary

JOHN BRYCE
Leading in IT Education
matrix company

- Array
- Multidimensional array
- varargs2

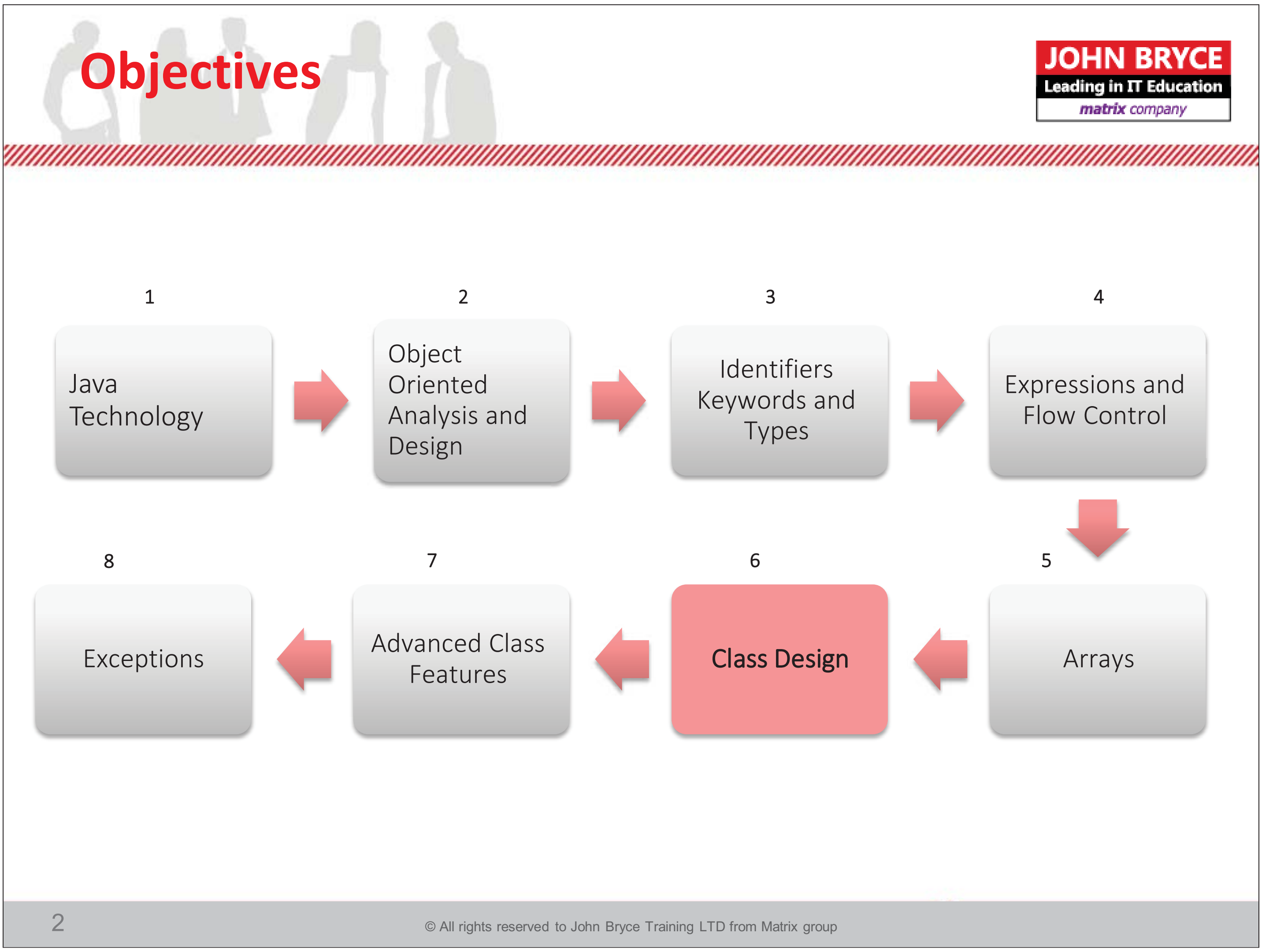
21

Class Design

Inheritance and Polymorphism

JOHN BRYCE
Leading in IT Education
matrix company

John Bryce Training, 29 Homa Umigdal St, Tel Aviv 6777129 // www.johnbryce.com



Objectives

JOHN BRYCE
Leading in IT Education
matrix company

1 Java Technology

2 Object Oriented Analysis and Design

3 Identifiers Keywords and Types

4 Expressions and Flow Control

5 Arrays

6 Class Design

7 Advanced Class Features

8 Exceptions

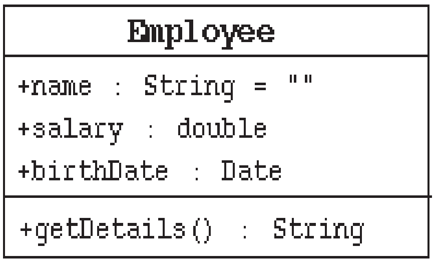
2

Objectives

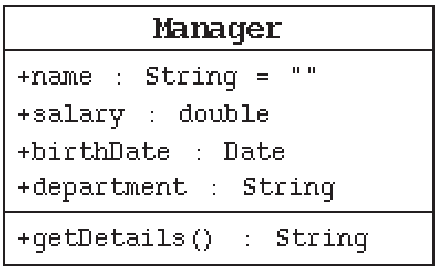
By the end of this session

- You'll be able to implement capsulation
- You'll be able to implement inheritance

The "*is a*" Relationship

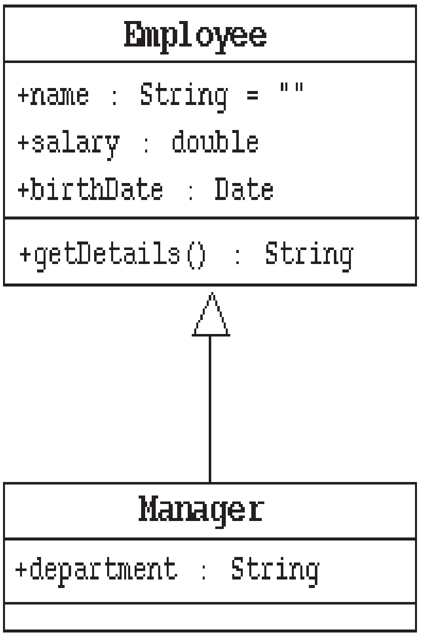


```
public class Employee {
  public String name = "";
  public double salary;
  public Date birthDate;

  public String getDetails() {...}
}
```



```
public class Manager {
  public String name = "";
  public double salary;
  public Date birthDate;
  public String department;

  public String getDetails() {...}
}
```

The "*is a*" Relationship

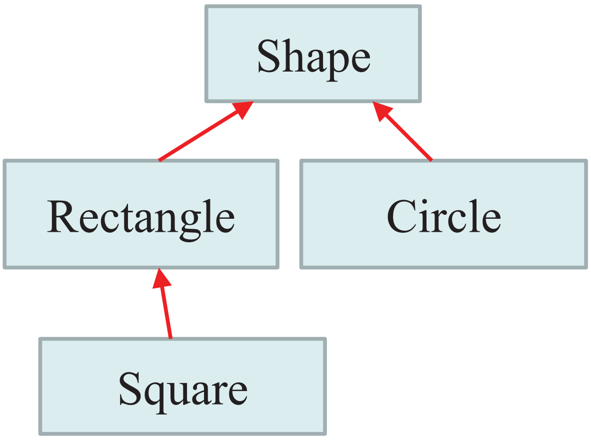


```
public class Employee {
  public String name = "";
  public double salary;
  public Date birthDate;

  public String getDetails() {...}
}
```

```
public class Manager extends Employee {
  public String department;
}
```

Class Inheritance

- Java implements single inheritance (unlike C++ that has multiple inheritance) among classes
- Single inheritance makes code more reliable
- Interfaces provide the benefits of multiple inheritance without drawbacks



Inheritance

Example

```
public class Shape {
        protected double area;
       public double getArea() { return area; }

}

public class Circle extends Shape {

   private double radius;

   public Circle(double radius) {

       this.radius = radius;

       area = Math.PI * radius * radius;

   }

}
```

Inheritance

- Keyword extends is used
- If the extend superclass clause is omitted, the class implicitly extends the class java.lang.Object
- Thus, java.lang.Object is the root of the class hierarchy, since every class is its subclass either directly or indirectly

Inheritance

- A subclass inherits all members of its superclass, except those who are invisible to the subclass (private)

- Attributes of the subclass can hide members of the superclass. In this case the super pseudo variable is used to access those members

Access Modifiers Summary

Modifier	Same Class	Same Package	Sub-Class	Universe
public	Yes	Yes	Yes	Yes
protected	Yes	Yes	Yes	
default	Yes	Yes		
private	Yes			

Constructors

this can be used to explicitly specify which constructor of the same class is to be called

super can be used to explicitly specify which constructor of the superclass is to be called

this

Example

```
public class Rectangle extends Shape {


        private double a,b;
        public Rectangle(double a, double b) {

        this.a = a; this.b = b;

                   area = a*b;

    }

        public Rectangle(double a) {
                this(a,a);

        }

}
```

super

Example

```
public class Rectangle extends Shape {
        private double a,b;
        public Rectangle(double a, double b) {

                this.a = a; this.b = b;

              area = a*b;

         } // implicitly calling Shape()constructor

}

public class Square extends Rectangle {

    public Square(double a) {

                super(a, a);

         } // explicitly calling Rectangle(a,a) constructor.

}
```

Method Overloading

Methods with the same name may be written in one class or its subclasses as long as:

Arguments types are different

or

Number of arguments is different

Example

```
public class Printer {

      public void print(int x){
                      System.out.println(x);
   }
   public void print(int x,int y){
                      System.out.println(x+y);
   }
   public void print(String str){
                      System.out.println(str);
   }
   public String print(String str1, int x){
                      System.out.println(str1+x);
         return str1+x;
   }
}
```

Method Overriding

- Inherited methods can be overriden:
  subclass re-implements a method's body
- Dynamic binding is used to call such methods
- Class (static) members (neither variables nor methods) are NEVER inherited
- Class methods are statically bound
- Instance attributes are also hidden during inheritance - this can cause non-trivial errors

Example

```
public class Shape {
        public void paint() { // do nothing }
}
public class Circle extends Shape {
    public void paint() { // overriding
        … draw a circle
    }
}


…
Shape s = new Circle(5.0); // assume such constr.
s.paint() // a circle is painted
…
```

Method Overriding

- The signature of the overriding method MUST be identical to that of the overridden one
- The subclass can declare the method with the same or less restrictive accessibility
- For the same reason at most the exceptions declared in the superclass can be declared

super

Example

```
public class Employee {
       ...
       public String getDetails() {
        return name+" "+salary;
    }
}

public class Manager extends Employee{
    ...
       public String getDetails() {
        return super.getDetails()
                                            +" "+dept;
    }
}
```

Rules About Overridden Methods

```
public class Parent {
  public void doSomething() {}
}
public class Child extends Parent {
  private void doSomething() {}
}
public class UseBoth {
  public void doOtherThing() {
    Parent p1 = new Parent();
    Parent p2 = new Child();
    p1.doSomething();
    p2.doSomething();
}
```

Final Classes and Methods

JOHN BRYCE
Leading in IT Education
matrix company

- Class which declared to be final can not be extended

  e.g. java.lang.String

- Final methods can not be overridden

References Types and Inheritance

JOHN BRYCE
Leading in IT Education
matrix company

- A class-reference type variable can be assigned a reference to an instance of the declared class including to those that are instances of any sublclasses

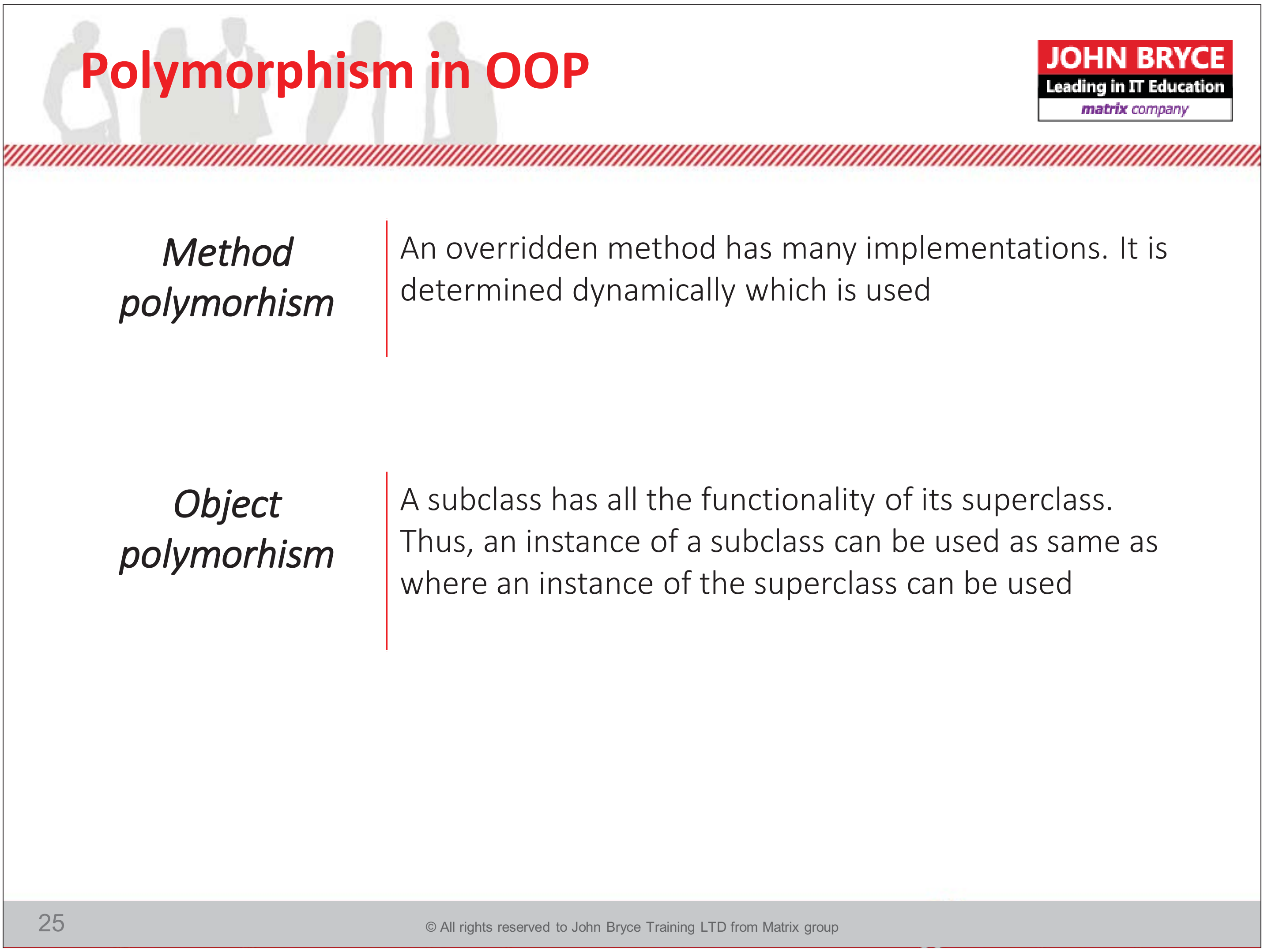
```
Shape s1 = new Shape();
Shape s2 = new Circle(5.0);
```

- As a result, Object type references can be assigned any instance reerences.

Polymorphism

- Polymorphism is the ability to have many different ***forms***
- An *object* has only one form
- A reference variable can refer to objects of different forms

Inheritance



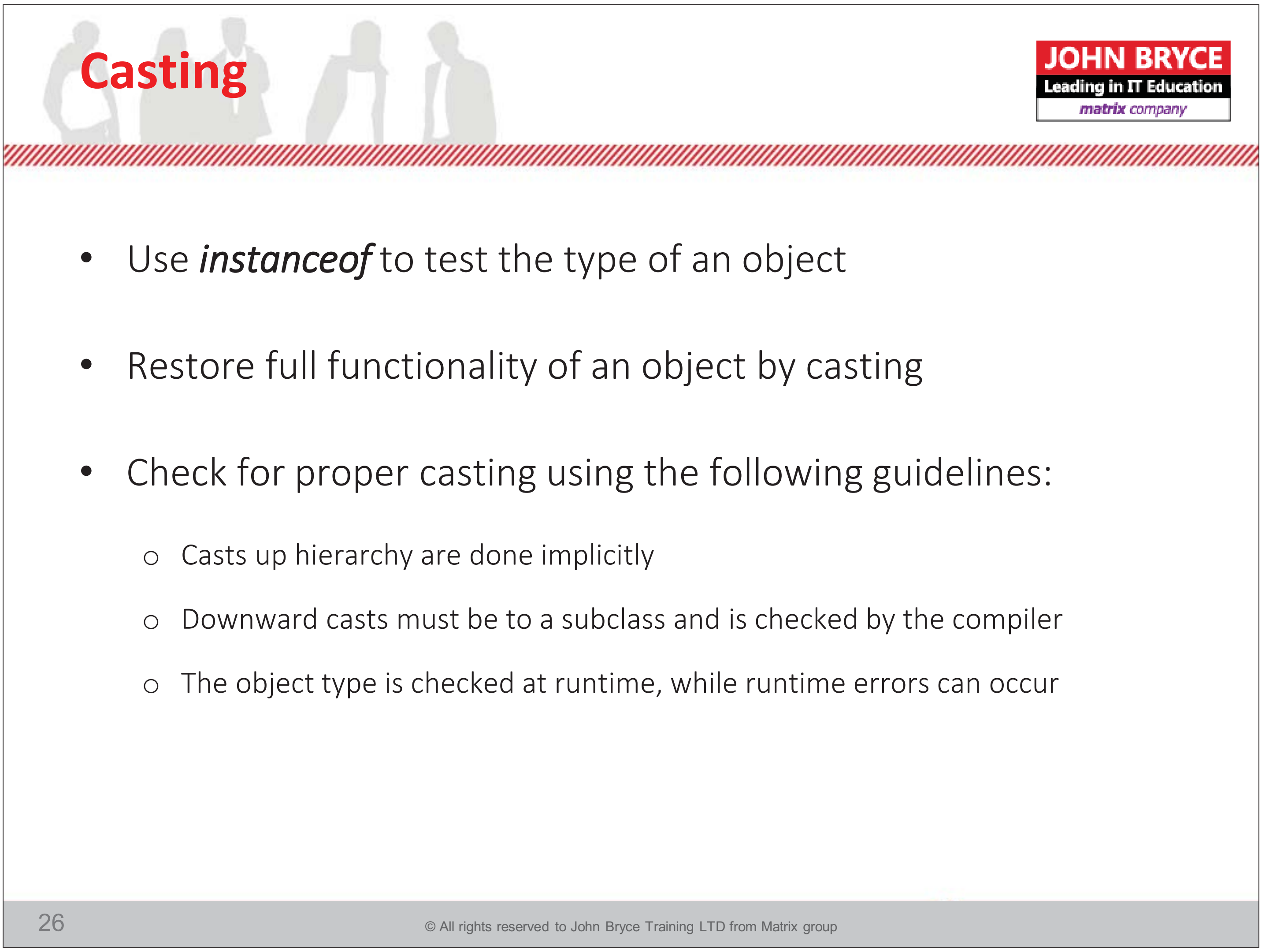
Polymorphism in OOP

Method polymorhism

An overridden method has many implementations. It is determined dynamically which is used

Object polymorhism

A subclass has all the functionality of its superclass. Thus, an instance of a subclass can be used as same as where an instance of the superclass can be used



Casting

- Use ***instanceof*** to test the type of an object
- Restore full functionality of an object by casting
- Check for proper casting using the following guidelines:
  - Casts up hierarchy are done implicitly
  - Downward casts must be to a subclass and is checked by the compiler
  - The object type is checked at runtime, while runtime errors can occur

Downcasting

```
Shape s;
Circle c = new Circle(1);
Rectangle r = new Rectangle(1.0, 2.0);
s = c; // polymorphism
s = (Shape) c; // needless casting

if(s instanceof Circle){
  Circle c1 = (Circle) s; // downcasting
  Circle c2 = (Circle) r; // illegal –
     //impossible
}
```

Heterogeneous Collections

Collections of objects with the same class type are called homogenous collections

```
MyDate[] dates = new MyDate[2];
dates[0] = new MyDate(22, 12, 1964);
dates[1] = new MyDate(22, 7, 1964);
```

Collections of objects with different class types are called heterogeneous collections

```
Employee [] staff = new Employee[1024];
staff[0] = new Manager();
staff[1] = new Employee();
staff[2] = new Engineer();
```

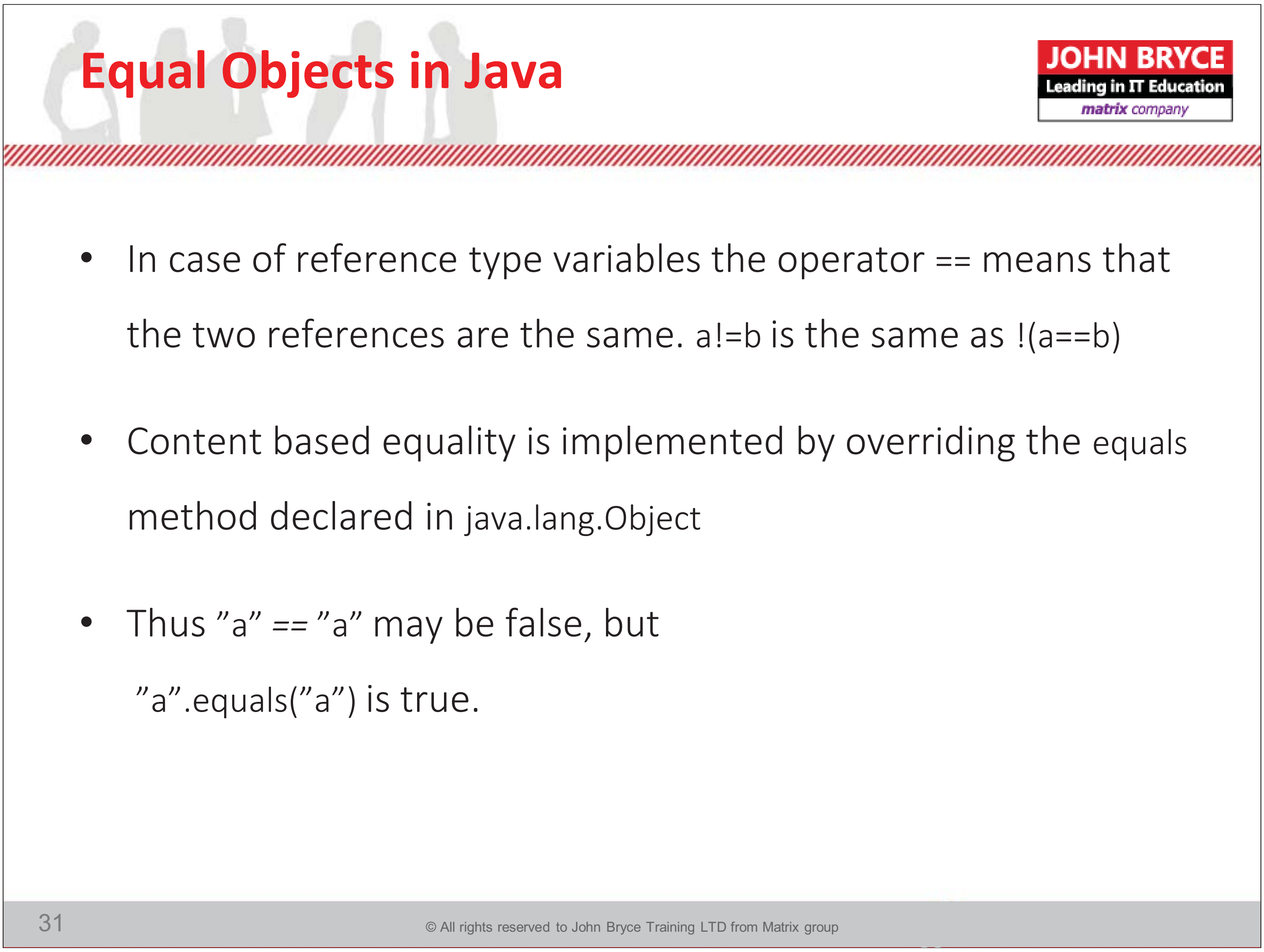
Object Methods Overridden

Object Methods Frequently Being Overridden

- Recall that the Object class is the root of all classes in Java
- A class declaration with no extends clause, implicitly uses "extends Object"
- Object’s methods that should be overridden
  - equals
  - toString

The *equals* Method

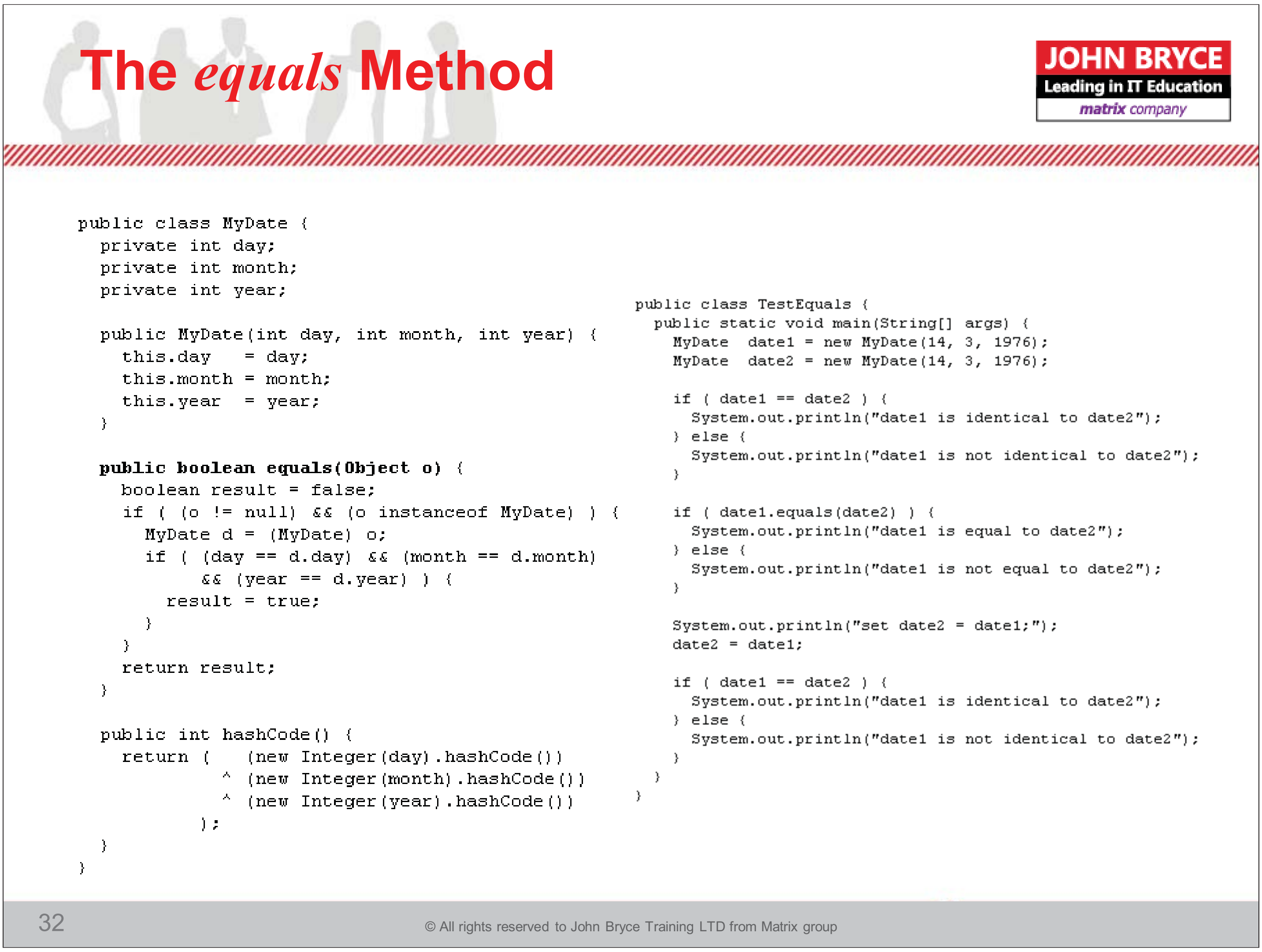
- == operator determines if two references are identical to each other
- The equals method determines if objects are "equal" by their contents, but not necessarily identical
- The Object implementation of the equals method uses the == operator
- User classes can override the equals method to implement a domain-specific test for equality



Equal Objects in Java

JOHN BRYCE
Leading in IT Education
matrix company

- In case of reference type variables the operator == means that the two references are the same. a!=b is the same as !(a==b)
- Content based equality is implemented by overriding the equals method declared in java.lang.Object
- Thus ”a” == ”a” may be false, but ”a”.equals(”a”) is true.



The *equals* Method

JOHN BRYCE
Leading in IT Education
matrix company

```
public class MyDate {
  private int day;
  private int month;
  private int year;

  public MyDate(int day, int month, int year) {
    this.day   = day;
    this.month = month;
    this.year  = year;
  }

  public boolean equals(Object o) {
    boolean result = false;
    if ( (o != null) && (o instanceof MyDate) ) {
      MyDate d = (MyDate) o;
      if ( (day == d.day) && (month == d.month)
           && (year == d.year) ) {
        result = true;
      }
    }
    return result;
  }

  public int hashCode() {
    return (    (new Integer(day).hashCode())
              ^ (new Integer(month).hashCode())
              ^ (new Integer(year).hashCode())
           );
  }
}
```

```
public class TestEquals {
  public static void main(String[] args) {
    MyDate  date1 = new MyDate(14, 3, 1976);
    MyDate  date2 = new MyDate(14, 3, 1976);

    if ( date1 == date2 ) {
      System.out.println("date1 is identical to date2");
    } else {
      System.out.println("date1 is not identical to date2");
    }

    if ( date1.equals(date2) ) {
      System.out.println("date1 is equal to date2");
    } else {
      System.out.println("date1 is not equal to date2");
    }

    System.out.println("set date2 = date1;");
    date2 = date1;

    if ( date1 == date2 ) {
      System.out.println("date1 is identical to date2");
    } else {
      System.out.println("date1 is not identical to date2");
    }
  }
}
```

The *toString* Method

- Converts an object to a String
- Used during string concatenation
- Override this method to provide information about a user-defined object in a readable format
- Primitive types are converted to a String using the wrapper class's *toString* static method

Polymorphism

Wrapper Classes

JOHN BRYCE
Leading in IT Education
matrix company

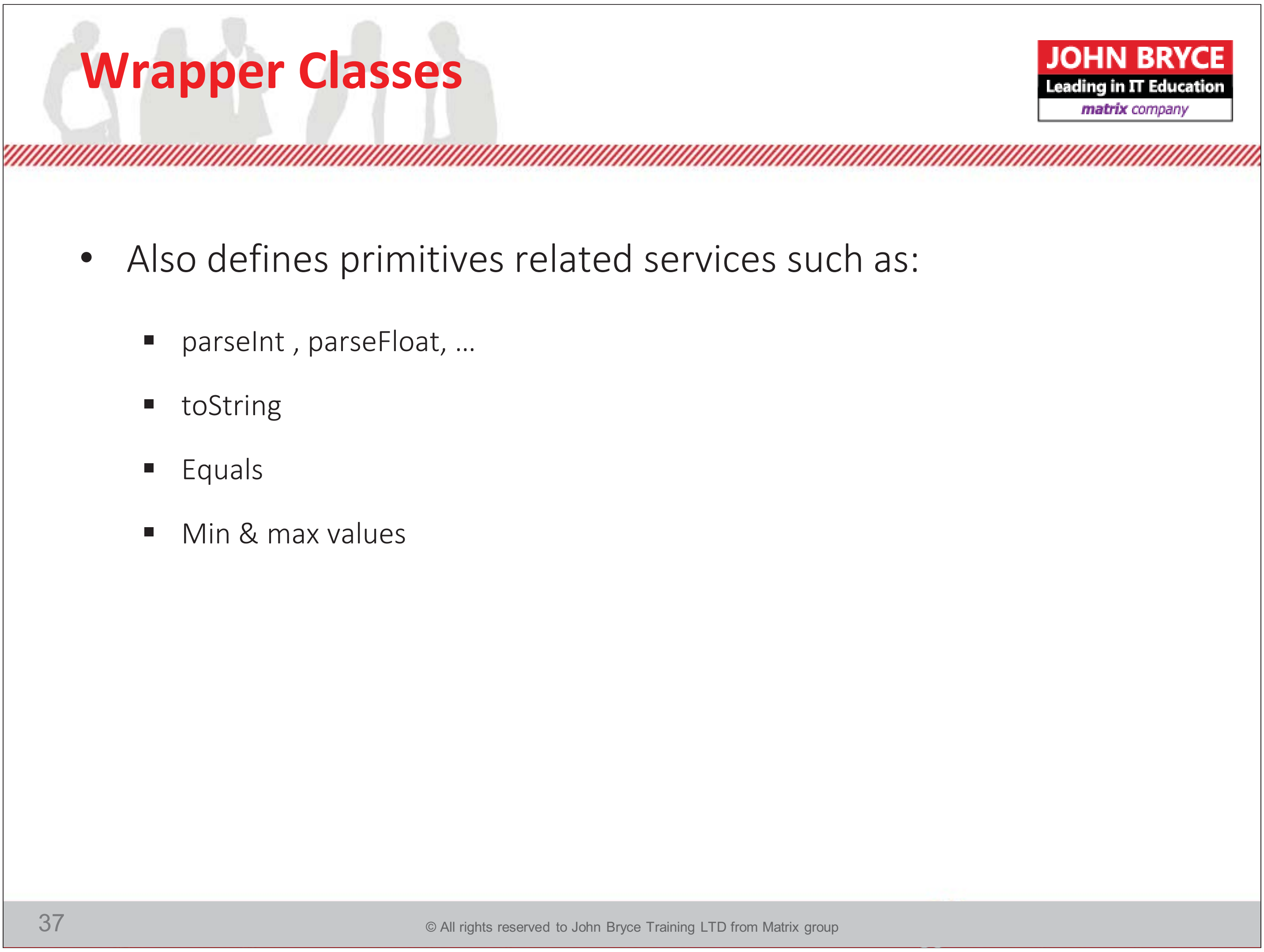
- Look at primitive data elements as objects
- Especially useful in heterogeneous collections
- Java ArrayList contains only Object s, so the only way to store *int*s in it, is to wrap them - treat them like objects

Primitive Data Type	Wrapper Class
boolean	Boolean
int	Integer
byte	Byte
char	Character
short	Short
long	Long
float	Float
double	Double

Wrapper Classes

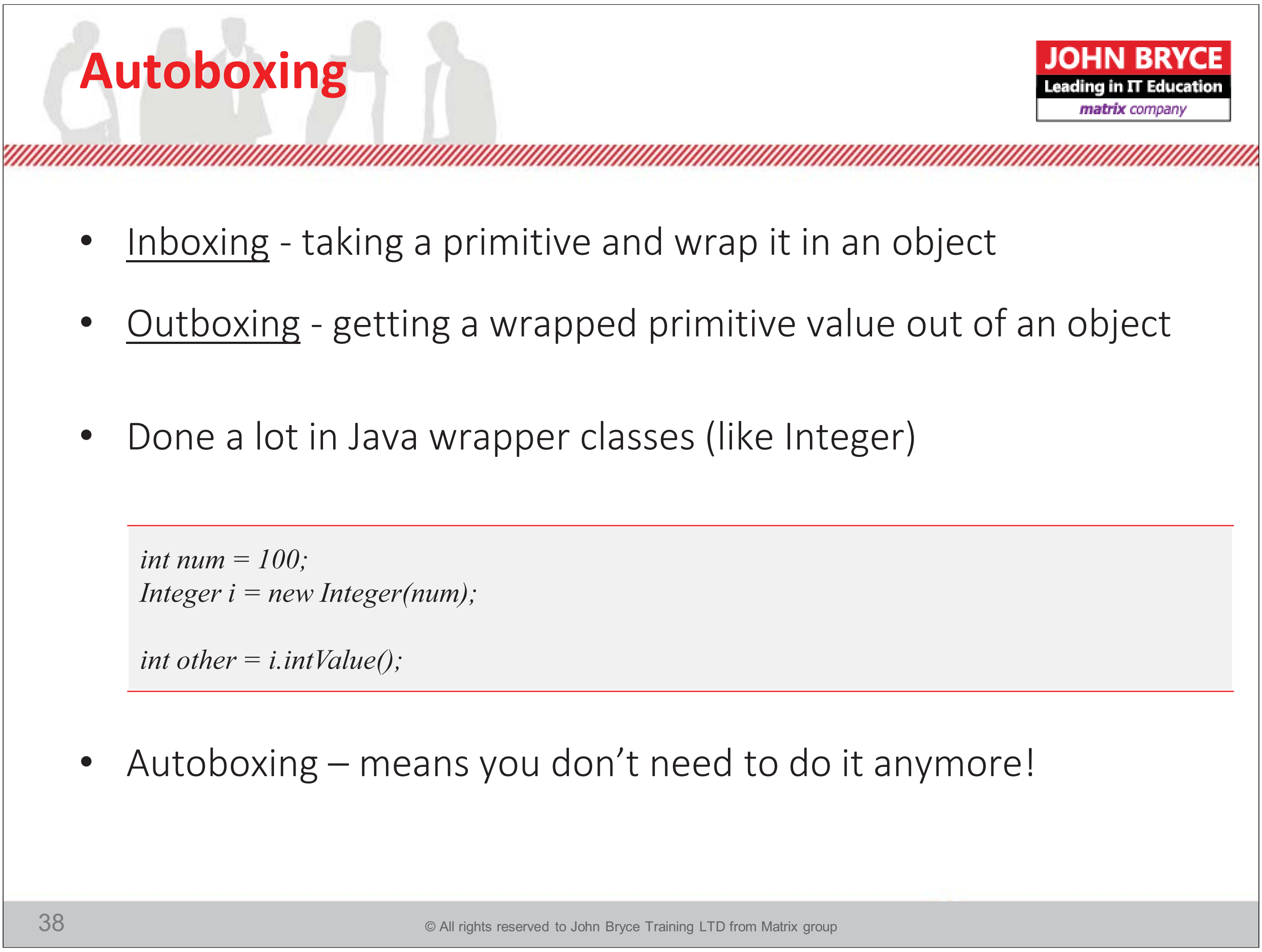
JOHN BRYCE
Leading in IT Education
matrix company

- ArrayList list = new ArrayList();
- list.add (new Integer(5));
- list.add (new Integer(7));
- int k = ((Integer)list.get(0)).intValue();



Wrapper Classes

- Also defines primitives related services such as:
  - parseInt , parseFloat, …
  - toString
  - Equals
  - Min & max values



Autoboxing

- Inboxing - taking a primitive and wrap it in an object
- Outboxing - getting a wrapped primitive value out of an object
- Done a lot in Java wrapper classes (like Integer)

```
int num = 100;
Integer i = new Integer(num);

int other = i.intValue();
```

- Autoboxing – means you don’t need to do it anymore!

Autoboxing

Example

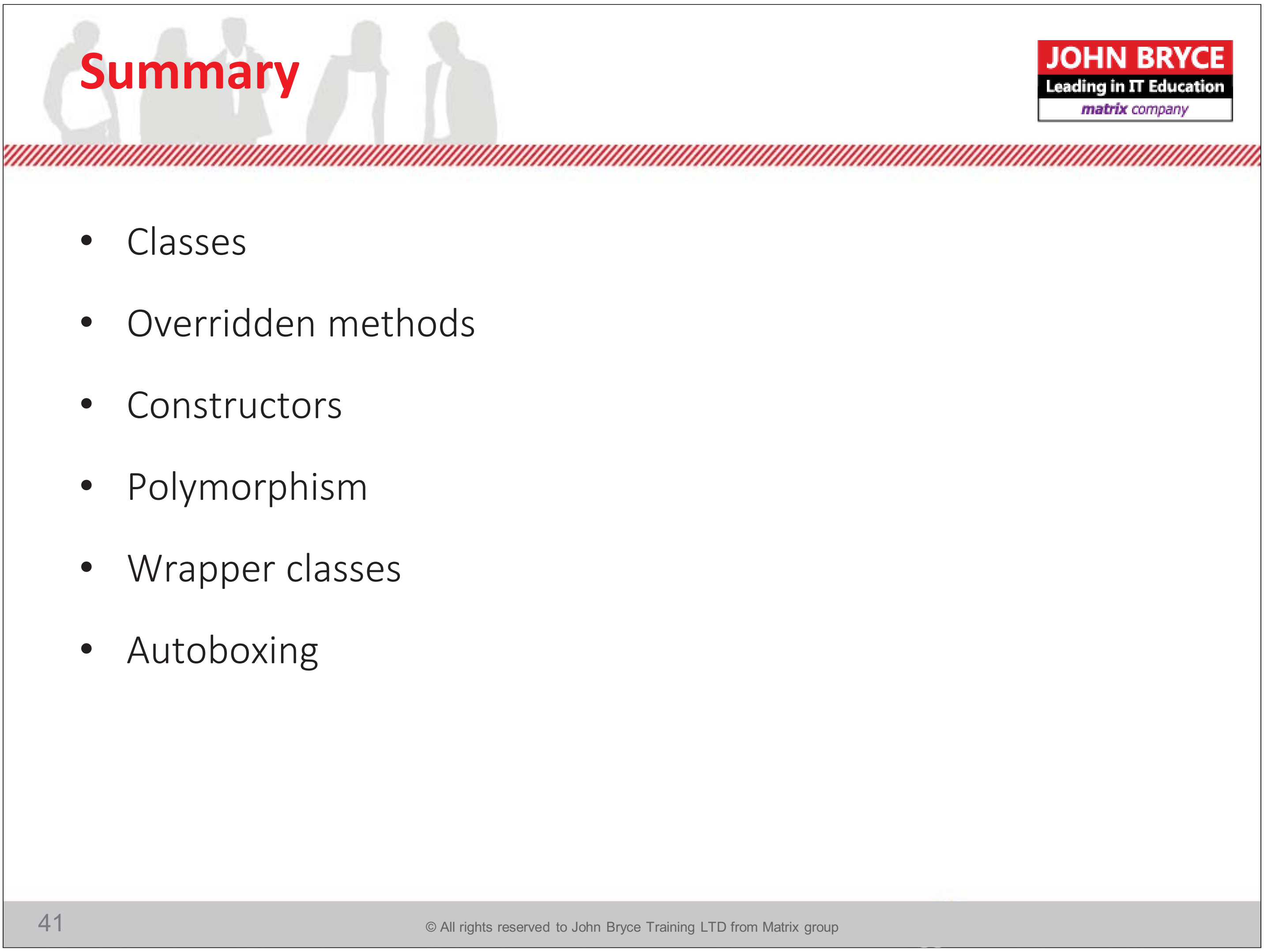
```
public class IntMaster {

      private int[] nums = {1,2,3,4,5,6,7,8,9,10};

      public Integer getInt(int index){
               return nums[index];
      }

      public void setInteger (Integer toReplace, int index){
               nums[index] = toReplace;
      }
}
```

Autoboxing

⚠ *Boxing is far from being efficient.*

Use it only to contain primitives in an object Collection.

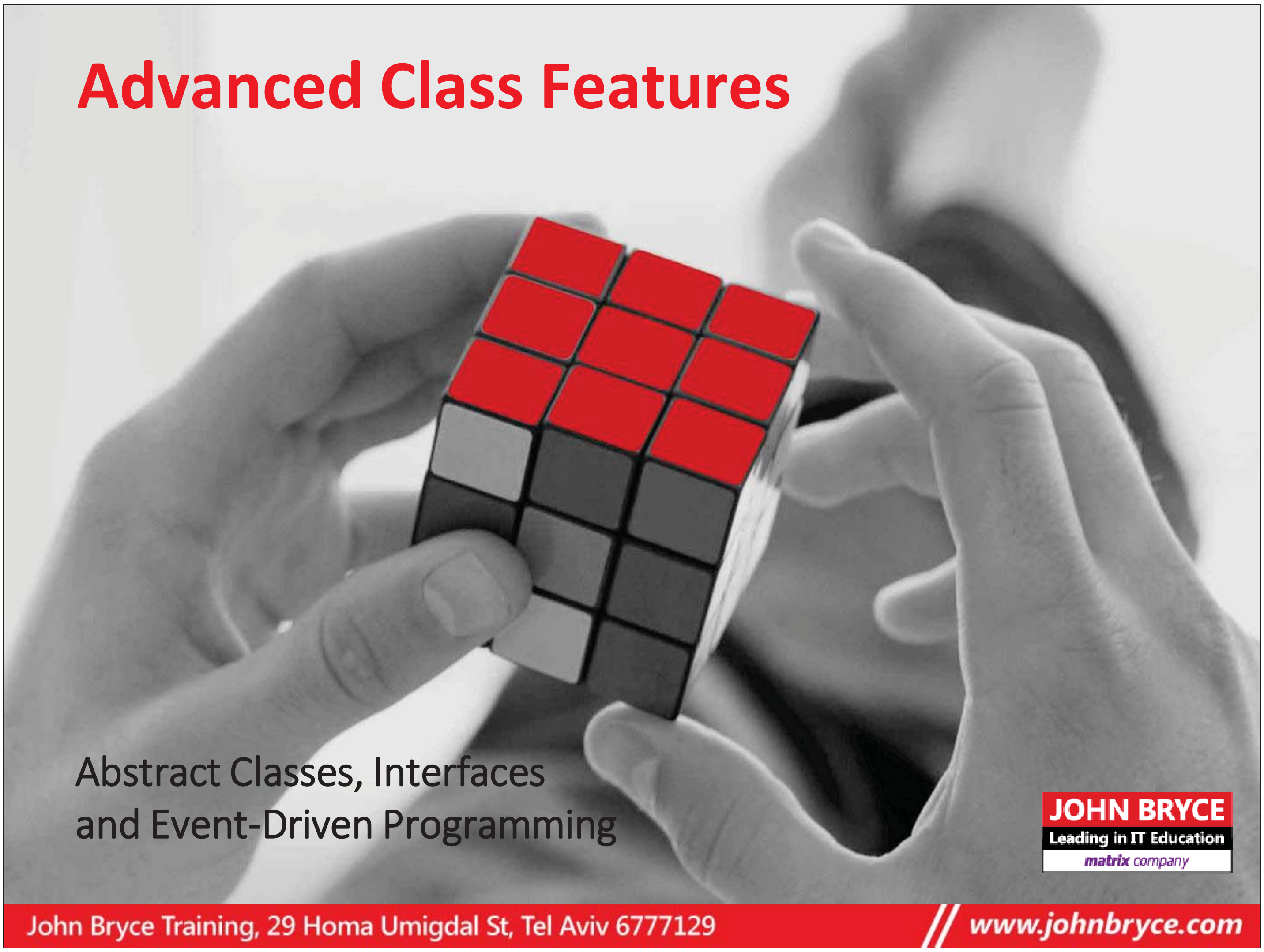
Never use it for scientific calculations.



Summary

- Classes
- Overridden methods
- Constructors
- Polymorphism
- Wrapper classes
- Autoboxing

Advanced Class Features



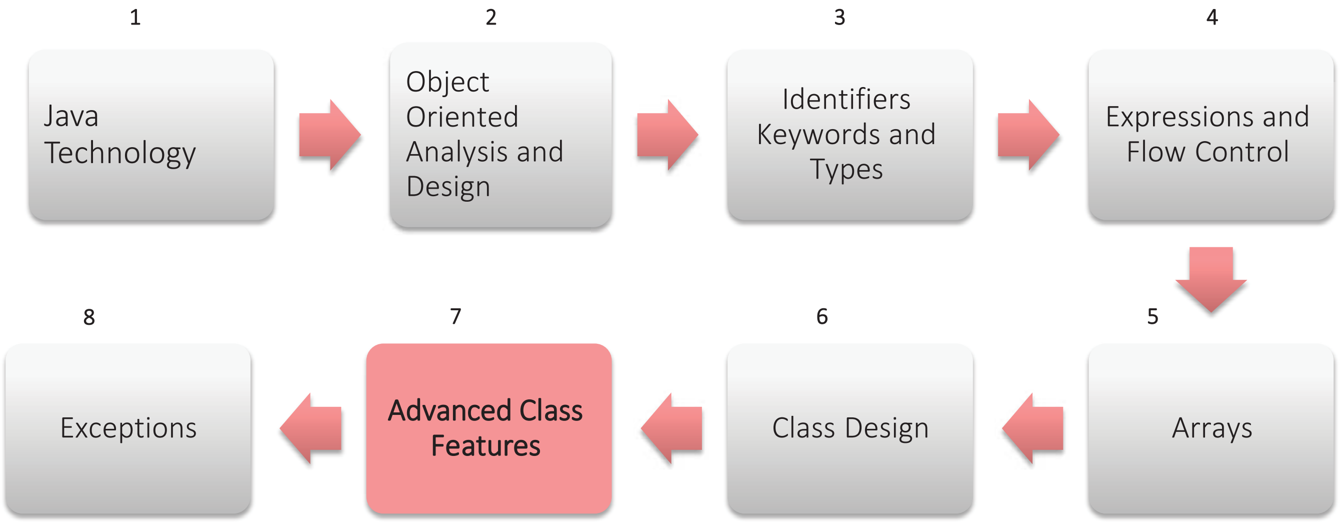
Abstract Classes, Interfaces
and Event-Driven Programming

JOHN BRYCE
Leading in IT Education
matrix company

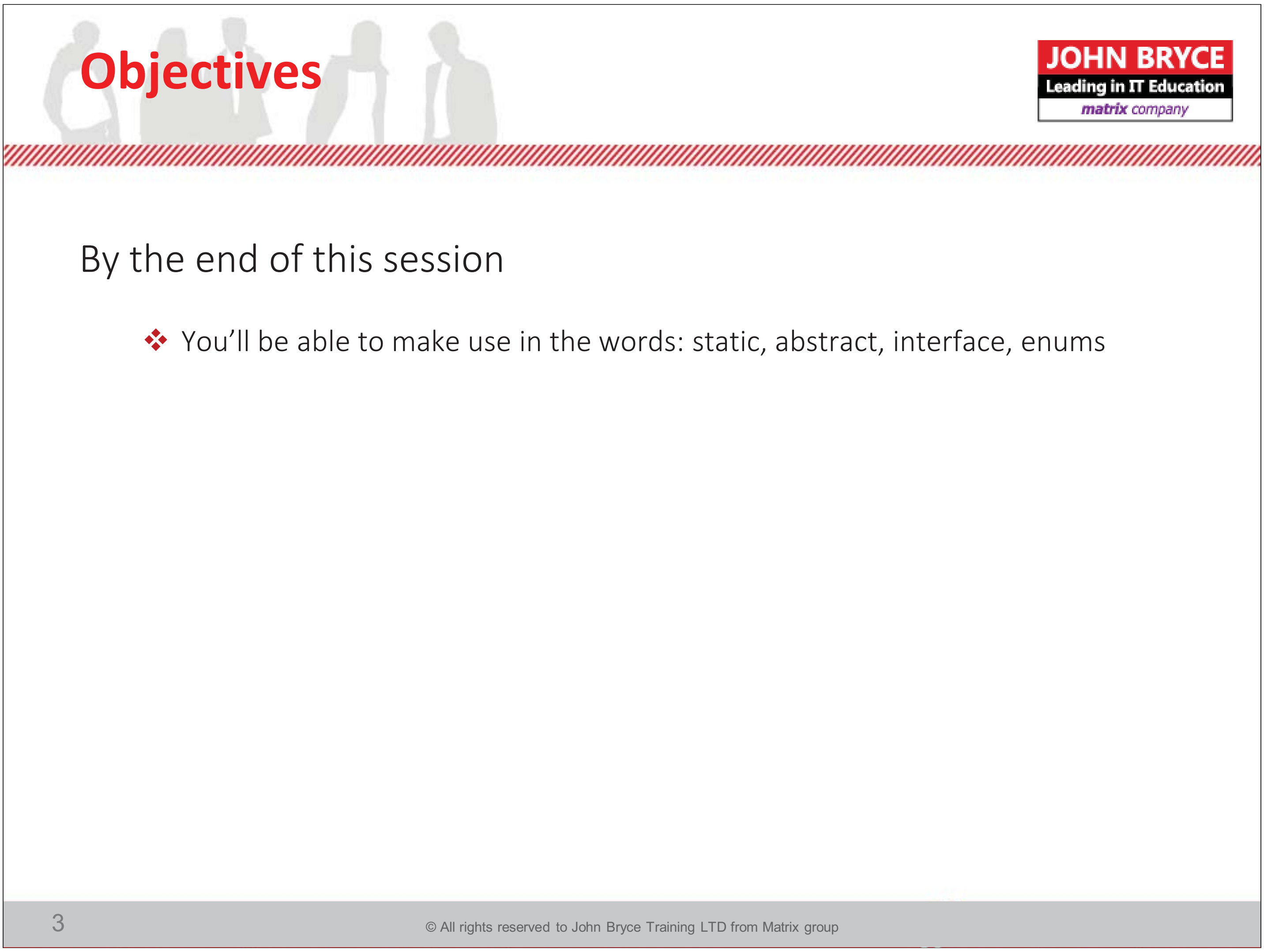
John Bryce Training, 29 Homa Umigdal St, Tel Aviv 6777129 // www.johnbryce.com

Objectives

JOHN BRYCE
Leading in IT Education
matrix company



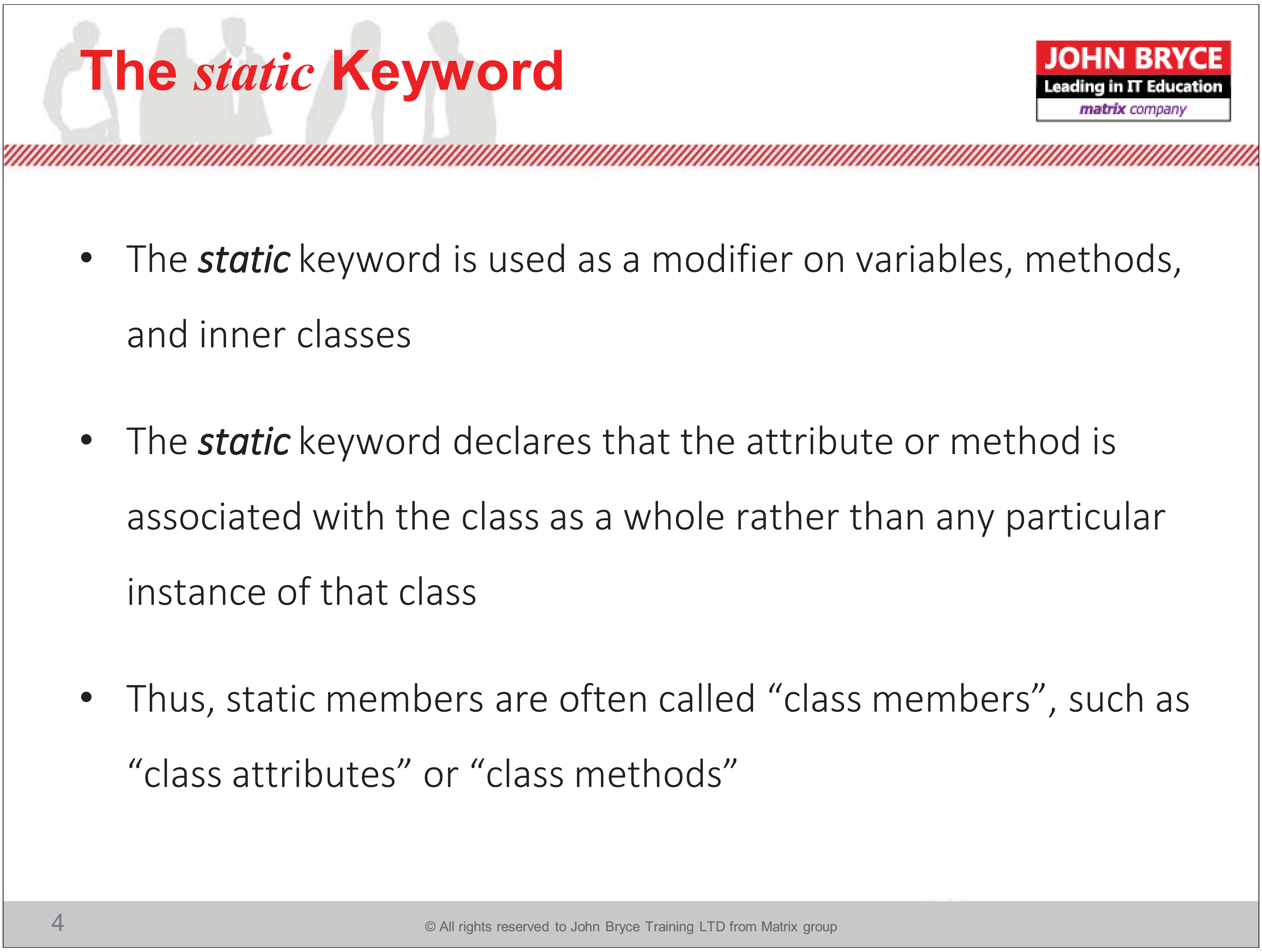
1 Java Technology → 2 Object Oriented Analysis and Design → 3 Identifiers Keywords and Types → 4 Expressions and Flow Control → 5 Arrays → 6 Class Design → 7 Advanced Class Features → 8 Exceptions



Objectives

By the end of this session

- You'll be able to make use in the words: static, abstract, interface, enums



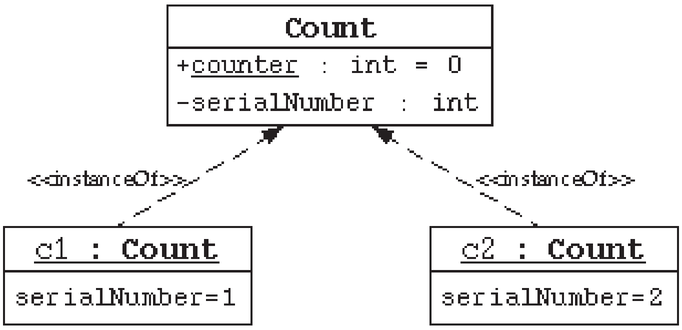
The *static* Keyword

- The ***static*** keyword is used as a modifier on variables, methods, and inner classes
- The ***static*** keyword declares that the attribute or method is associated with the class as a whole rather than any particular instance of that class
- Thus, static members are often called "class members", such as "class attributes" or "class methods"

The *static* Keyword

Are shared among all instances of a class

```
public class Count {
  private int serialNumber;
  public static int counter = 0;

  public Count() {
    counter++;
    serialNumber = counter;
  }
}
```



The *static* Keyword

static attribute can be accessed from outside the class if marked as public.

```
public class OtherClass {
  public void incrementNumber() {
    Count.counter++;        // without an instance of the class
  }
}
```

The *static* Keyword

```
public class Count {
  private int serialNumber;
  private static int counter = 0;

  public static int getTotalCount()        {
  return counter;
  }

  public Count() {
    counter++;
    serialNumber = counter;
  }
}
```

```
public class TestCounter {
public static void main(String[] args) {
System.out.println("Number of counter is "
+ Count.getTotalCount());
Count count1 = new Count();
System.out.println("Number of counter is "
+ Count.getTotalCount());
 }
}
```

The *static* Keyword

- *Static block* code (static initializer, or static constructor) executes only once, when the class is loaded
- A *static block* is usually used to initialize static (class) attributes

```
public class Count {
  public static int counter;
  static {
    counter =      Integer.getInteger("myApp.Count.counter").intValue();
  }
}
```

The *singleton design* Pattern

May be instantiated only once

- The client should not be able to instantiate it:

  private constructor

- The class stores its only instance

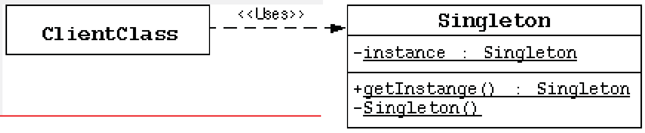
  static (class) variable

- The class lets users to get that only instance:

  static (class) method

The *singleton design* Pattern

```
public class Company {
  private static Company instance = new    Company();

  public static Company getCompany() {
    return instance;
  }

  private Company() {...}

}
```



Client usage:

Company c = Company.getCompany();

Static Import

Instead of doing that:

```
public double calculate(double startValue){
        return startValue*Math.PI+100/Math.E;
}
```

Programmers prefer Constant Interfaces:

```
public interface MyConstants{
      public double PI = 3.141592653589793;
      public double E= 2.718281828459045;
}
```

```
public class MyClass implements MyConstants{
…
public double calculate(double startValue){
        return startValue*PI+100/E;
}
```

Static Import

Constant Interface Anti-pattern

- Ease of use shouldn’t have structural influence
- Class that implements an interface must take it all
- The polymorphic ability that gained is irrelevant

Static Import

The solution – Static Imports

- Import static members and static methods only
- Allows unqualified access to static member of other class/interface
- Done without inheriting the content of the other class/interface

```
import static java.lang.Math.*;
or
import static java.lang.Math.PI;
import static java.lang.Math.E;

public class MyClass{
…
public double calculate(double startValue){
        return startValue*PI+100/E;
}
```

Destructor-like Methods

- An object is destructed automatically when it can not be accessed by any reference anymore (its RefCount=0)
- Then, some pre-specified methods are called:
  - finalize – instance method; runs before GC on the instance

STATIC

Static & *final* Keywords Combination

- Constants in Java are specified as ***final***
- Usually, they are also ***static***
    - Math.PI
    - Math.E
- Globally accessible mathematical functions are also ***static***:
    - Math.sin()
    - Math.cos()

The *final* Keyword

- Constants:

  private static final double   DEFAULT_INTEREST_RATE=3.2;

- Blank **final** Instance Attribute:

```
public class Customer {
  private final long customerID;
  public Customer() {
    customerID = createID();
  }
  public long getID() {
    return customerID;
  }
  private long createID() {
    return ... // generate new ID
  }
... // more declarations }
```

The *final* Keyword Summary

- You can not subclass a final class
- You can not override a final method
- A final variable is a constant
- You can set a final variable only once, but that assignment can occur independently of the declaration; this is called "blank final variable"
  - A blank final instance attribute must be set in every constructor
  - A blank final method variable must be set in the method body before being used

Abstract Class

JOHN BRYCE
Leading in IT Education
matrix company

- A class that can not be instantiated
- A class that declared as abstract:

```
public abstract class Shape {…
```

- We declare a class as abstract because we:
  - want to prohibit it from being instantiated
  - lack the functionality to implement some methods
- Can not be both final and abstract at the same time

Abstract Class

JOHN BRYCE
Leading in IT Education
matrix company

Example

```
public abstract class Shape {
  public abstract void paint();//no body
}
public class Circle extends Shape {
    public void paint(){   //implement the
                           //body
       … draw a circle
    }
}
```

- For an undefined Shape we don't know how to paint it!

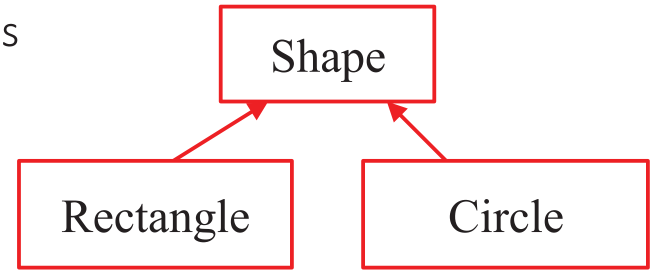
Abstract Methods

- *Abstract* classes may (but don't have to) have *abstract* methods
- These methods must be overridden in non-abstract subclasses to provide an implementation
- A method can not be both final and *abstract* at the same time

Abstract Classes Usage

Recommendations

- Instead of mixing different levels of inheritance:
  - Use only Rectangle and Circle (both – 2nd depth)
  - Do not use Shape directly



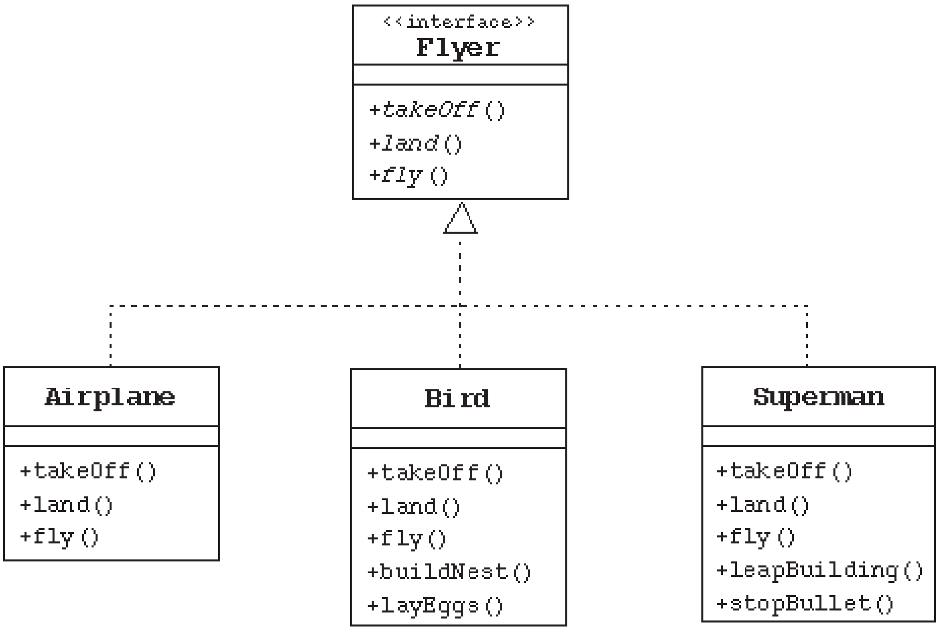
- Importance in superclassing for heterogeneous collections
  - Define a list of different shapes, with the same methods, may be implemented differently
  - getPerimeter() is undefined at the higher level

ABSTRACT

Interfaces

- A "public interface" is a contract between client code and the class that implements that interface
- A Java *interface* is a formal declaration of such contract in which all methods contain no implementation
- Many, unrelated classes can implement the same *interface*
- A class can implement many, unrelated interfaces

Interfaces



<<interface>>
Flyer

+takeOff()
+land()
+fly()

Airplane

+takeOff()
+land()
+fly()

Bird

+takeOff()
+land()
+fly()
+buildNest()
+layEggs()

Superman

+takeOff()
+land()
+fly()
+leapBuilding()
+stopBullet()

Interfaces

- An interface resembles to an abstract class but it is not a class
- An interface can not have runnable code inside
- All interfaces and their members are public and abstract by default, thus, it is deprecated to state these explicitly

Interface Members

JOHN BRYCE
Leading in IT Education
matrix company

- Interfaces can contain:
  - final variables (constants)
  - abstract methods
- A non-abstract class implements an interface if and only if:
  - it declares that it implements the interface
  - it implements all of the abstract methods of the interface

Interface References

JOHN BRYCE
Leading in IT Education
matrix company

- An interface type variable can be assigned instances of classes implementing the interface
- An abstract class can implement an interface. Hence, it declares some or all of the interface’s methods as abstract

Considerable

Example

```
public interface Paintable {
   public void paint();
}
public abstract class Shape implements Paintable {
  public abstract void paint(); // no body
}
…
Shape s = new Circle(5.0);
Paintable p = s;
p = new Circle(5.0);//Circle implements Paintable
```

Interfaces and Inheritance

- A subclass implements all the interfaces that its superclass implements
- A class can implement more than one interface
- An interface can extend multiple interfaces
- A class implementing an interface A implements all the interfaces that A extends

Interface Inheritance

```
public interface CanSayYes {
   public void sayYes();
}
public interface CanSayNo {
   public void sayNo();
}
public interface CanSayYesOrNo extends CanSayYes, CanSayNo{
}
public class Politician implements CanSayYesOrNo {
   public void sayYes() {System.out.println(”yes”);}
   public void sayNo()  {System.out.println(”no”);}
}
```

Uses of *interfaces*

- Declaring methods which one or more classes are expected to implement
- Determining an object's programming interface without revealing the actual body of the class
- Capturing **similarities** between unrelated classes without forcing a class relationship (has a relationship)
- Simulating **multiple inheritance** by declaring a class that implements several interfaces

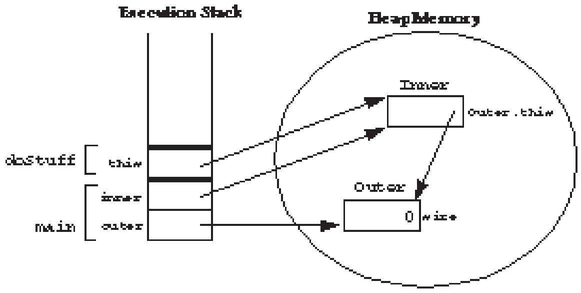
Interface

Inner Classes

- Allow a class definition to be placed inside another class definition
- Group classes that logically belong together
- Have an access to their enclosing class's scope

Inner Classes

```
public class Outer2 {
  private int size;

  public class Inner {
    public void doStuff() {
      size++;
    }
  }
}
```



```
public class TestInner {
  public static void main(String[] args) {
    Outer2 outer = new Outer2();

    // Must create an Inner object relative to an Outer
    Outer2.Inner inner = outer.new Inner();
    inner.doStuff();
  }
}
```

Inner Classes

- You can use the class name only within the defined scope, except when used in a qualified name.

  The name of the inner class must differ from the enclosing class

- The inner class can be defined inside a method.

  Only local variables marked as final can be accessed by methods within an inner class

Inner Classes

JOHN BRYCE
Leading in IT Education
matrix company

- The inner class can use both class and instance variables of the enclosing classes and local variables of enclosing blocks
- The inner class can be defined as abstract
- The inner class can have any access mode
- The inner class can act as an interface implemented by another inner class

Inner Classes

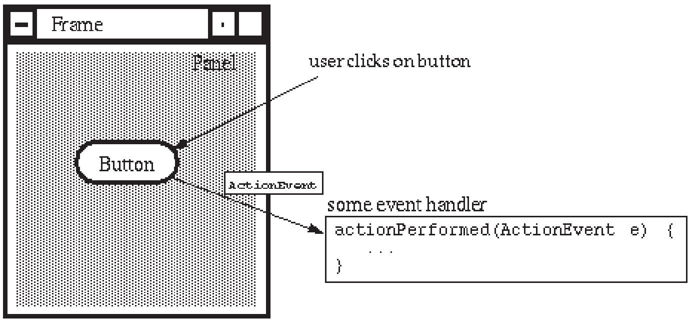
JOHN BRYCE
Leading in IT Education
matrix company

- Inner classes that are declared as static automatically become top-level classes
- Inner classes can not declare any static members; only top-level classes can declare static members
- An inner class wanting to use a static member must be declared static

Event-driven

Programming and Listeners

- Events - Objects that describe what happened
- Event sources - The generator of an event
- Event handlers - A method that receives an event object, deciphers it, and processes the user's interaction



```
actionPerformed(ActionEvent e) {
   ...
}
```

- Event handlers register with components when they are interested in events generated by that component

Event-driven

Programming and Listeners

```
public class TestButton {
  private Button b;

  public void launchFrame() {
   b.addActionListener(new ButtonHandler());
  }

  public class ButtonHandler implements
ActionListener {
    public void actionPerformed(ActionEvent e) {
      System.out.println("Action occurred");
      System.out.println("Button's command is: "
        + e.getActionCommand());
    }
  } //inner class

} //outer class
```

```
public class TestButton {
private Button b;

public void launchFrame() {
 b.addActionListener(new ActionListener()
  {
    public void actionPerformed(ActionEvent e) {
      System.out.println("Action occurred");
      System.out.println("Button's command is: "
        + e.getActionCommand());
    }
  } //anonymous inner class
);
}

 } //outer class
```

Event-driven

Programming and Listeners

Listener

interface declaring the methods to be called when some events occur

- After a specific class implementing this interface, is registered, its methods will be called b.addActionListener(…)

Event adapter

specific (non-abstract) classes implementing the listener interfaces in an empty way

- No need to implement ALL the interface methods
- WindowListener has 7 methods, when you may be interested in listening to only one event: windowClosing
- Without an adapter, your class could be abstract!

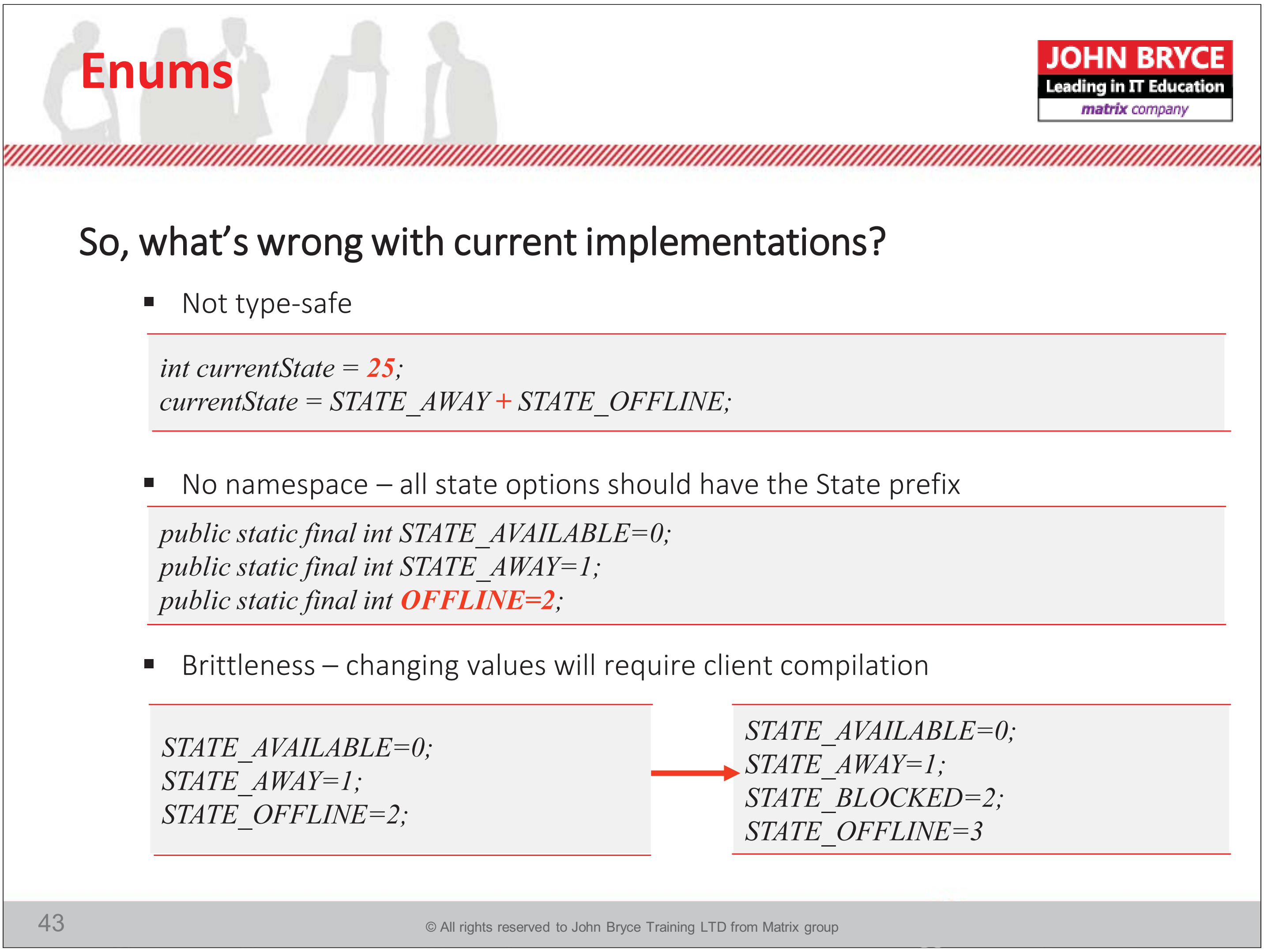
Enums

Understanding enumeration types:

- Specify customized types
- Define optional values

- Currently done like that:

```
public static final int STATE_AVAILABLE=0;
public static final int STATE_AWAY=1;
public static final int STATE_OFFLINE=2;
```



Enums

So, what's wrong with current implementations?

- Not type-safe

```
int currentState = 25;
currentState = STATE_AWAY + STATE_OFFLINE;
```

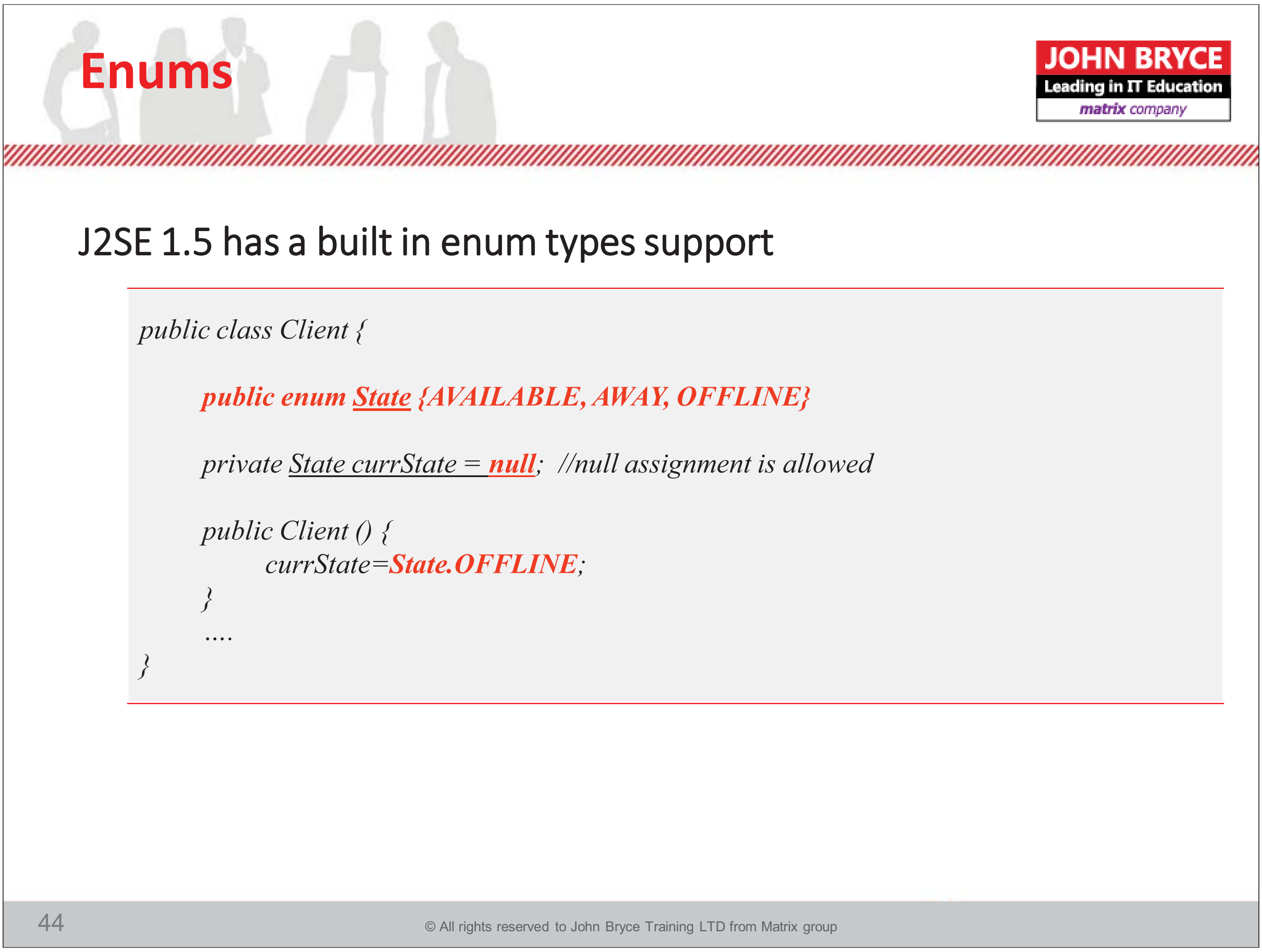
- No namespace – all state options should have the State prefix

```
public static final int STATE_AVAILABLE=0;
public static final int STATE_AWAY=1;
public static final int OFFLINE=2;
```

- Brittleness – changing values will require client compilation

```
STATE_AVAILABLE=0;
STATE_AWAY=1;
STATE_OFFLINE=2;
```

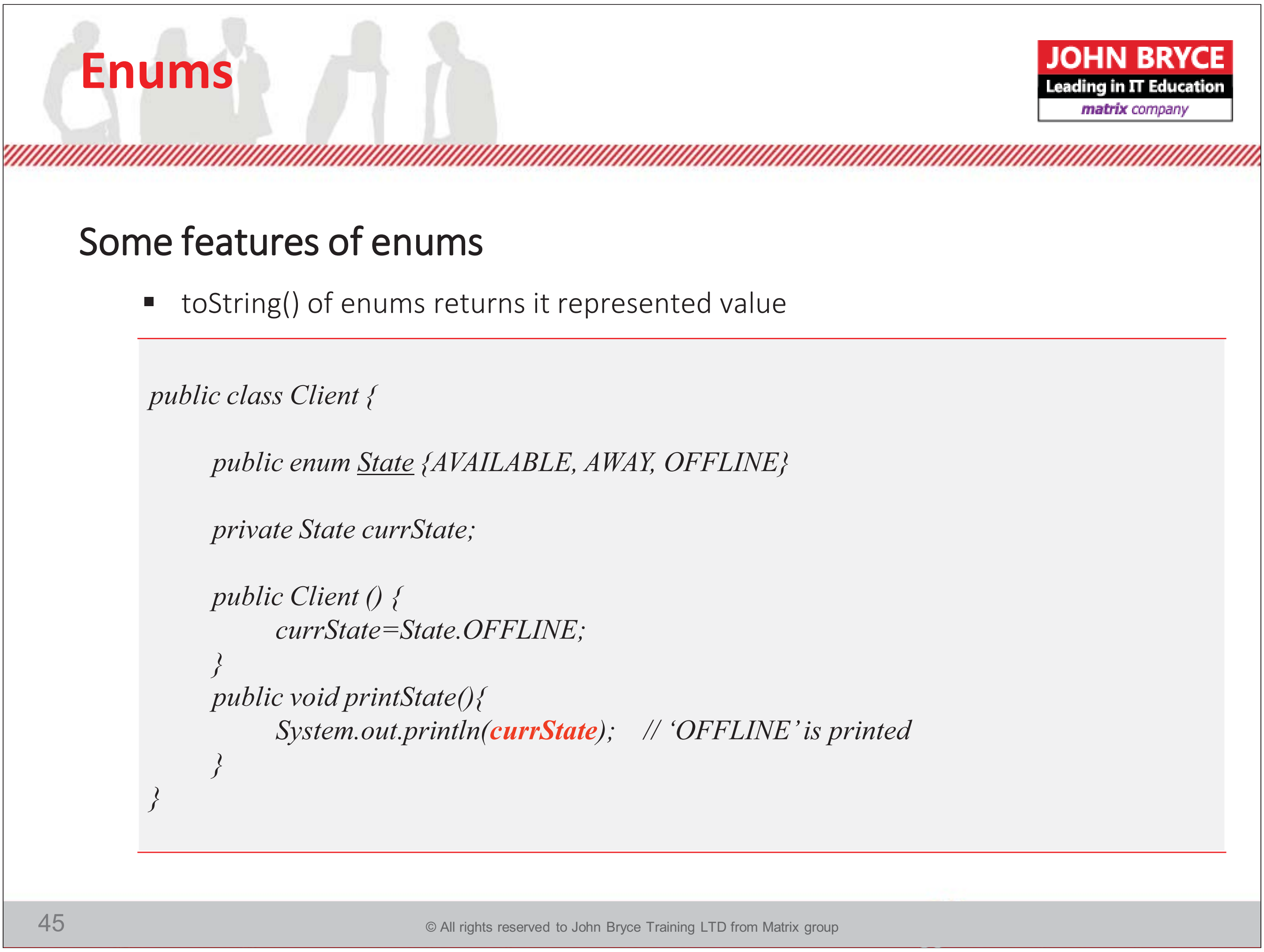
```
STATE_AVAILABLE=0;
STATE_AWAY=1;
STATE_BLOCKED=2;
STATE_OFFLINE=3
```



Enums

J2SE 1.5 has a built in enum types support

```
public class Client {

        public enum State {AVAILABLE, AWAY, OFFLINE}

        private State currState = null;  //null assignment is allowed

        public Client () {
                currState=State.OFFLINE;
        }
        ....
}
```

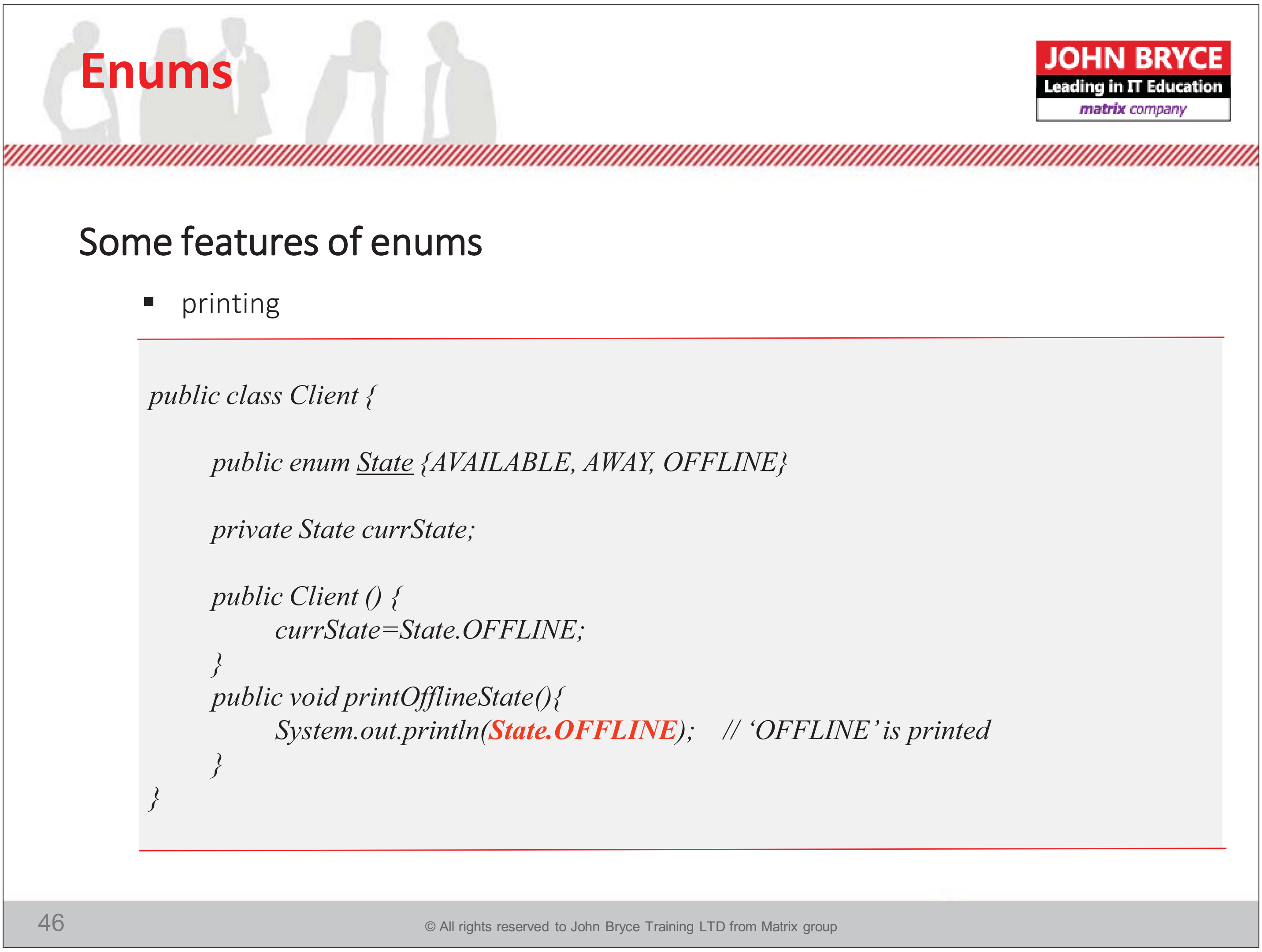


Enums

Some features of enums

- toString() of enums returns it represented value

```
public class Client {

        public enum State {AVAILABLE, AWAY, OFFLINE}

        private State currState;

        public Client () {
                currState=State.OFFLINE;
        }
        public void printState(){
                System.out.println(currState);    // ‘OFFLINE’ is printed
        }
}
```

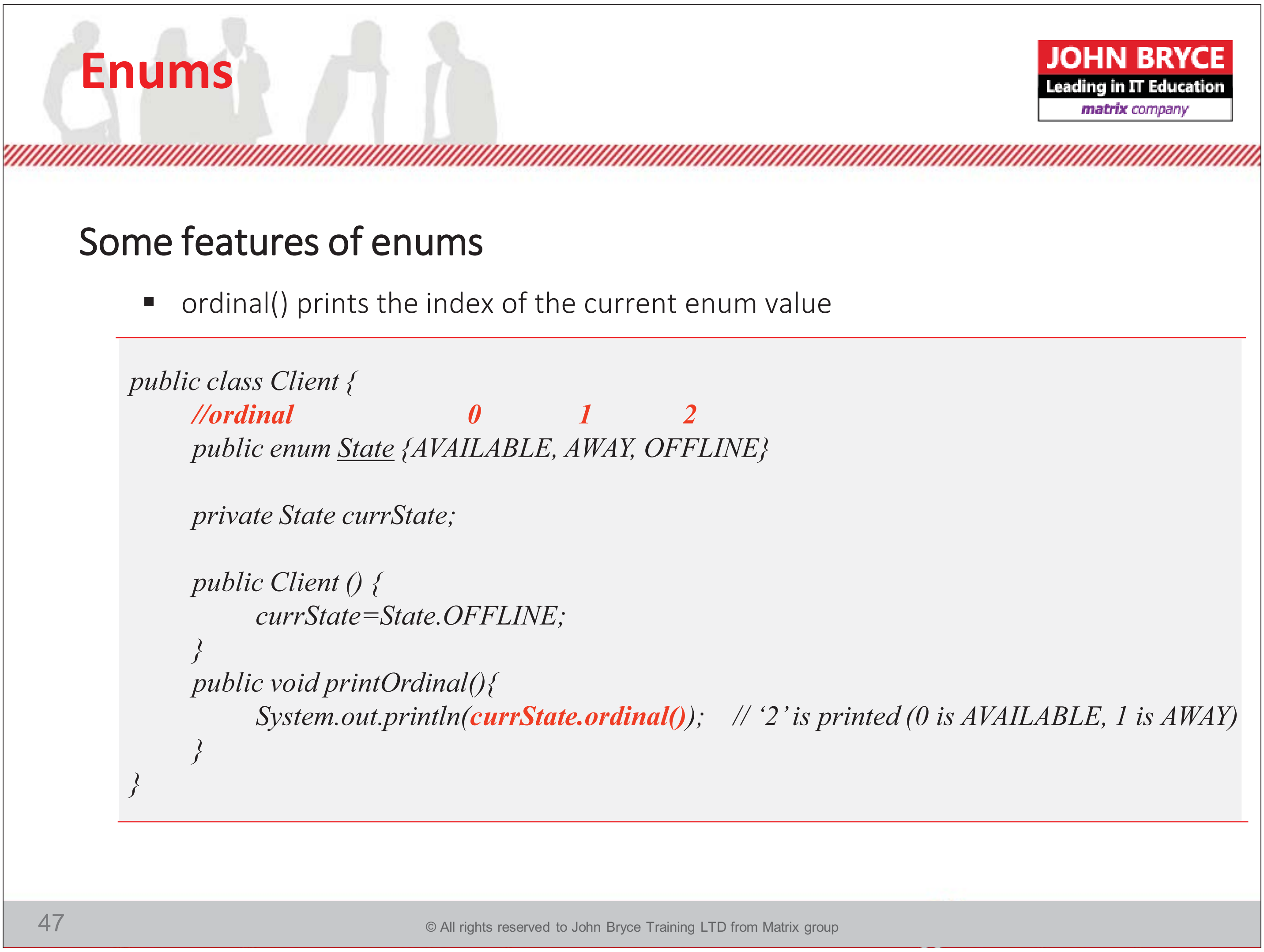


Enums

Some features of enums

- printing

```
public class Client {

        public enum State {AVAILABLE, AWAY, OFFLINE}

        private State currState;

        public Client () {
                currState=State.OFFLINE;
        }
        public void printOfflineState(){
                System.out.println(State.OFFLINE);    // ‘OFFLINE’ is printed
        }
}
```



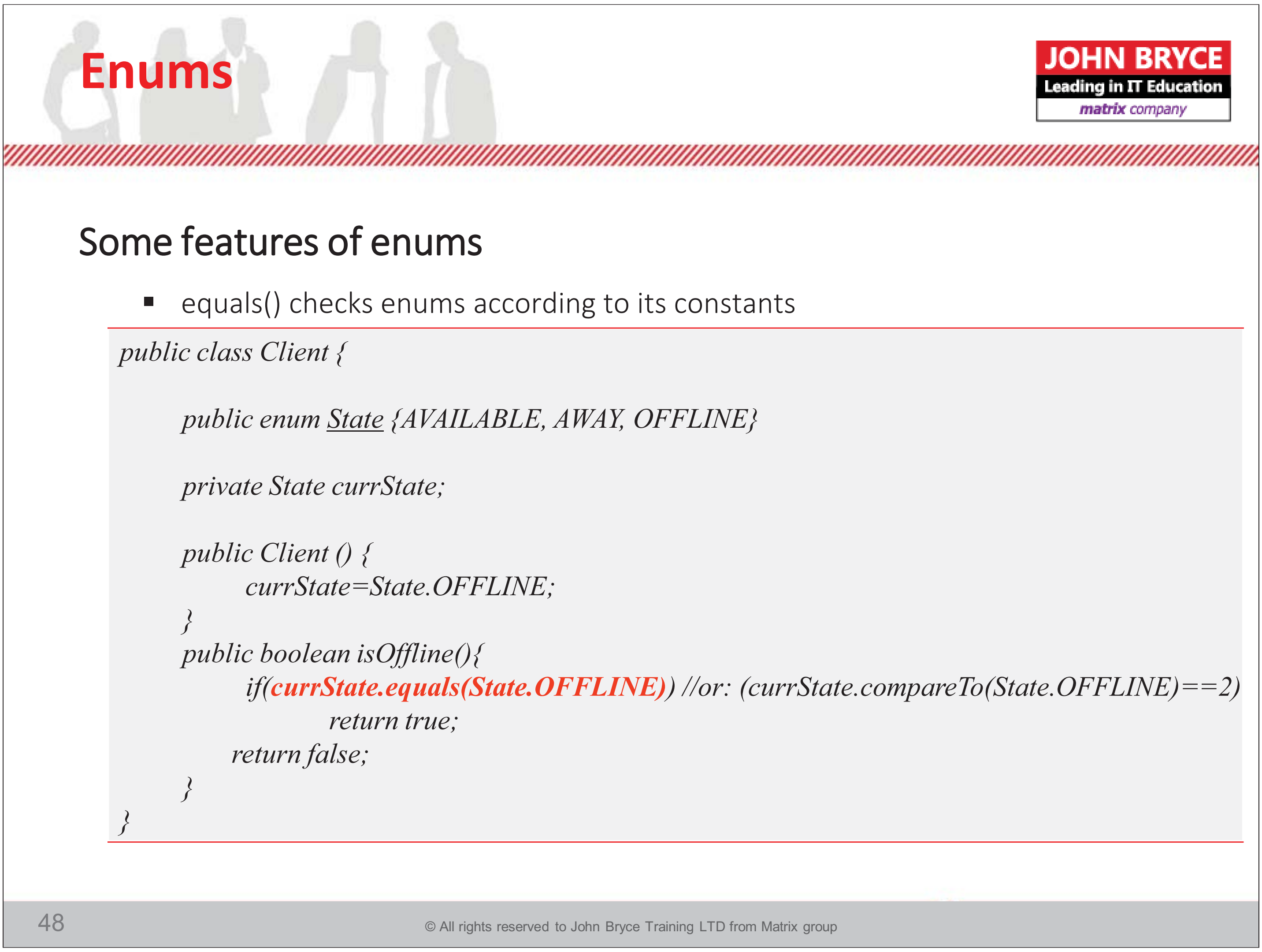
Enums

JOHN BRYCE
Leading in IT Education
matrix company

Some features of enums

- ordinal() prints the index of the current enum value

```
public class Client {
        //ordinal                      0            1           2
        public enum State {AVAILABLE, AWAY, OFFLINE}

        private State currState;

        public Client () {
                currState=State.OFFLINE;
        }
        public void printOrdinal(){
                System.out.println(currState.ordinal());    // '2' is printed (0 is AVAILABLE, 1 is AWAY)
        }
}
```



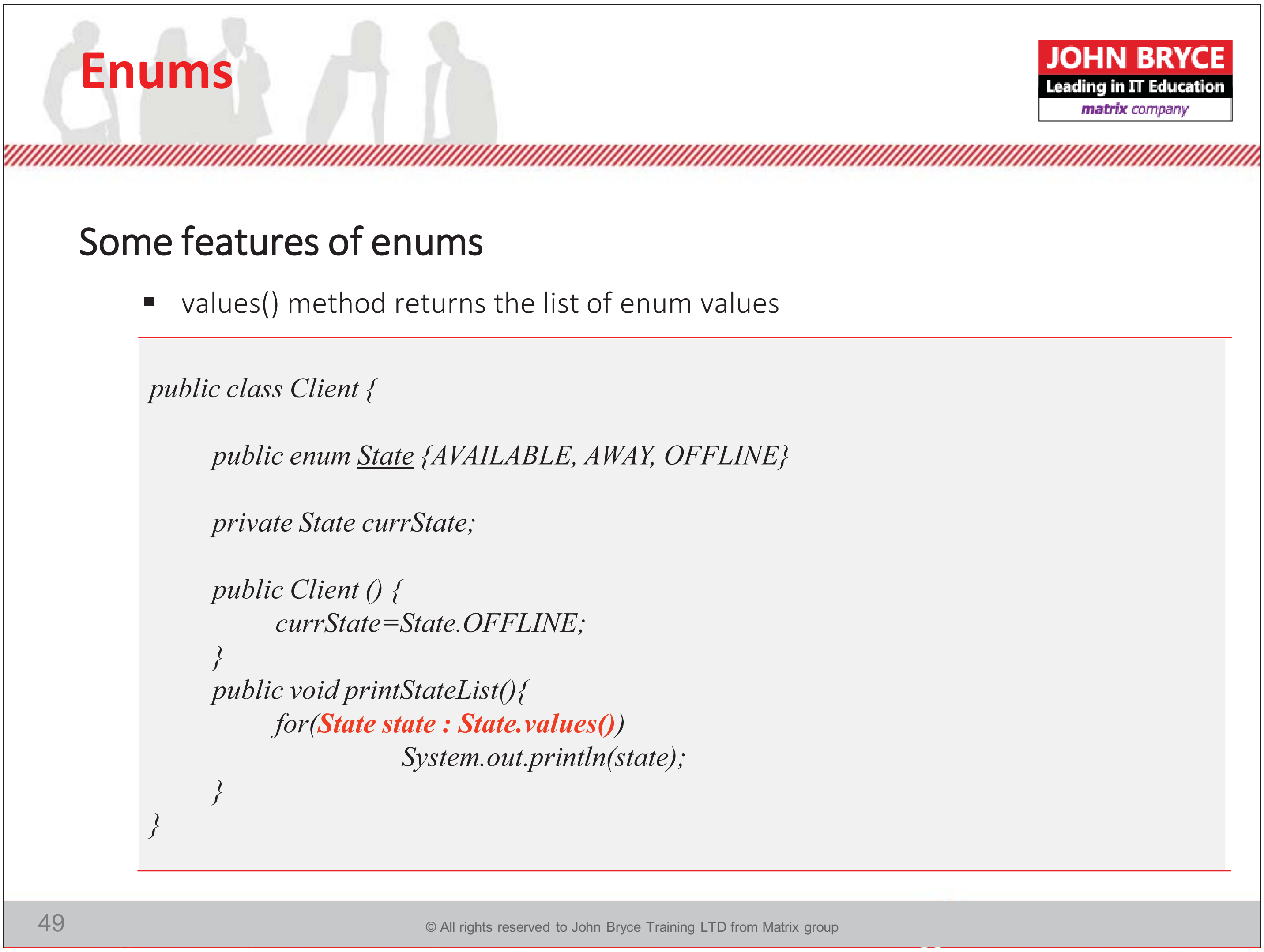
Enums

JOHN BRYCE
Leading in IT Education
matrix company

Some features of enums

- equals() checks enums according to its constants

```
public class Client {

        public enum State {AVAILABLE, AWAY, OFFLINE}

        private State currState;

        public Client () {
                currState=State.OFFLINE;
        }
        public boolean isOffline(){
                if(currState.equals(State.OFFLINE)) //or: (currState.compareTo(State.OFFLINE)==2)
                        return true;
                return false;
        }
}
```

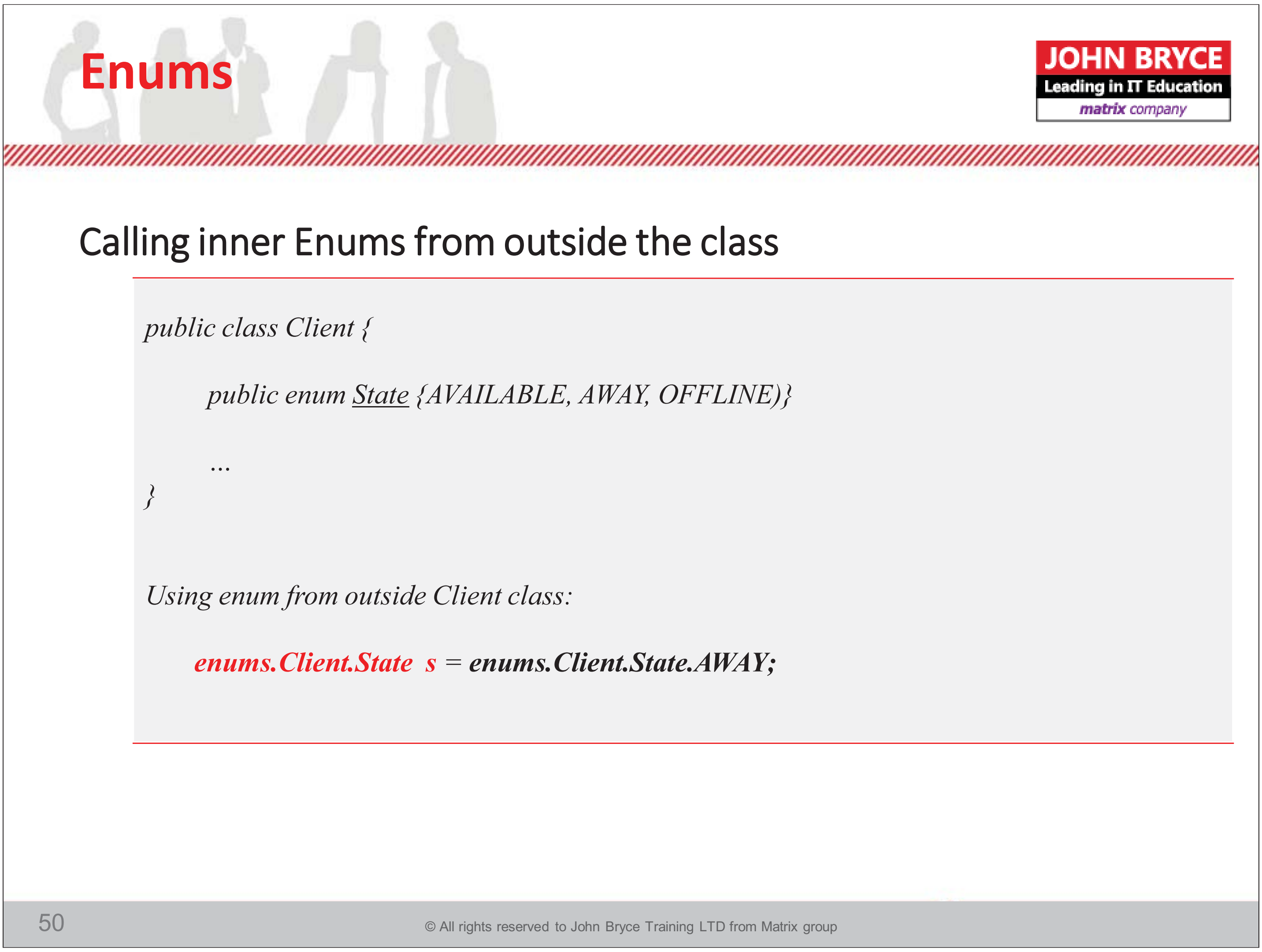


Enums

Some features of enums

- values() method returns the list of enum values

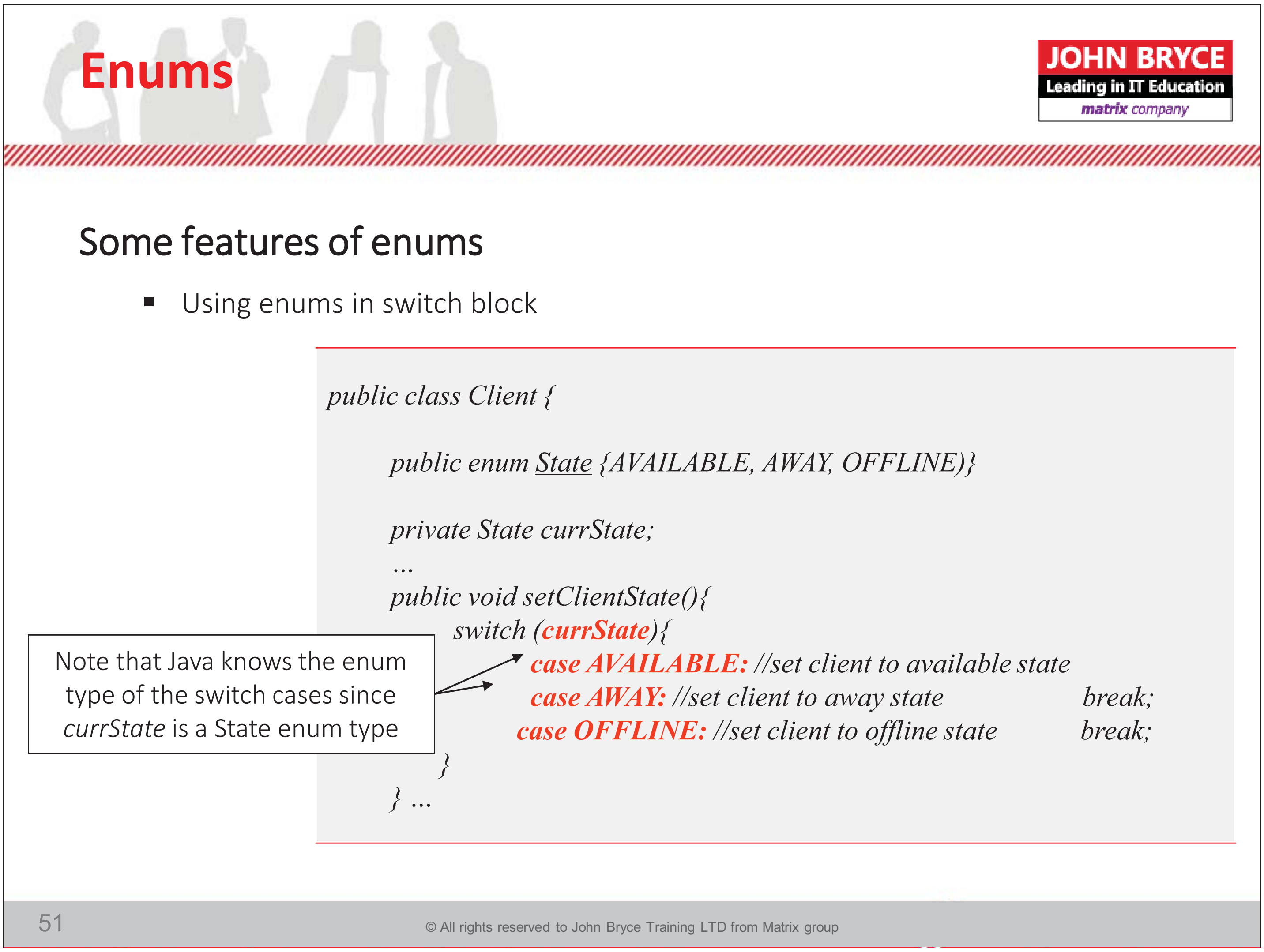
```
public class Client {

        public enum State {AVAILABLE, AWAY, OFFLINE}

        private State currState;

        public Client () {
                currState=State.OFFLINE;
        }
        public void printStateList(){
                for(State state : State.values())
                                System.out.println(state);
        }
}
```



Enums

Calling inner Enums from outside the class

```
public class Client {

        public enum State {AVAILABLE, AWAY, OFFLINE)}

        …
}


Using enum from outside Client class:

      enums.Client.State  s = enums.Client.State.AWAY;
```



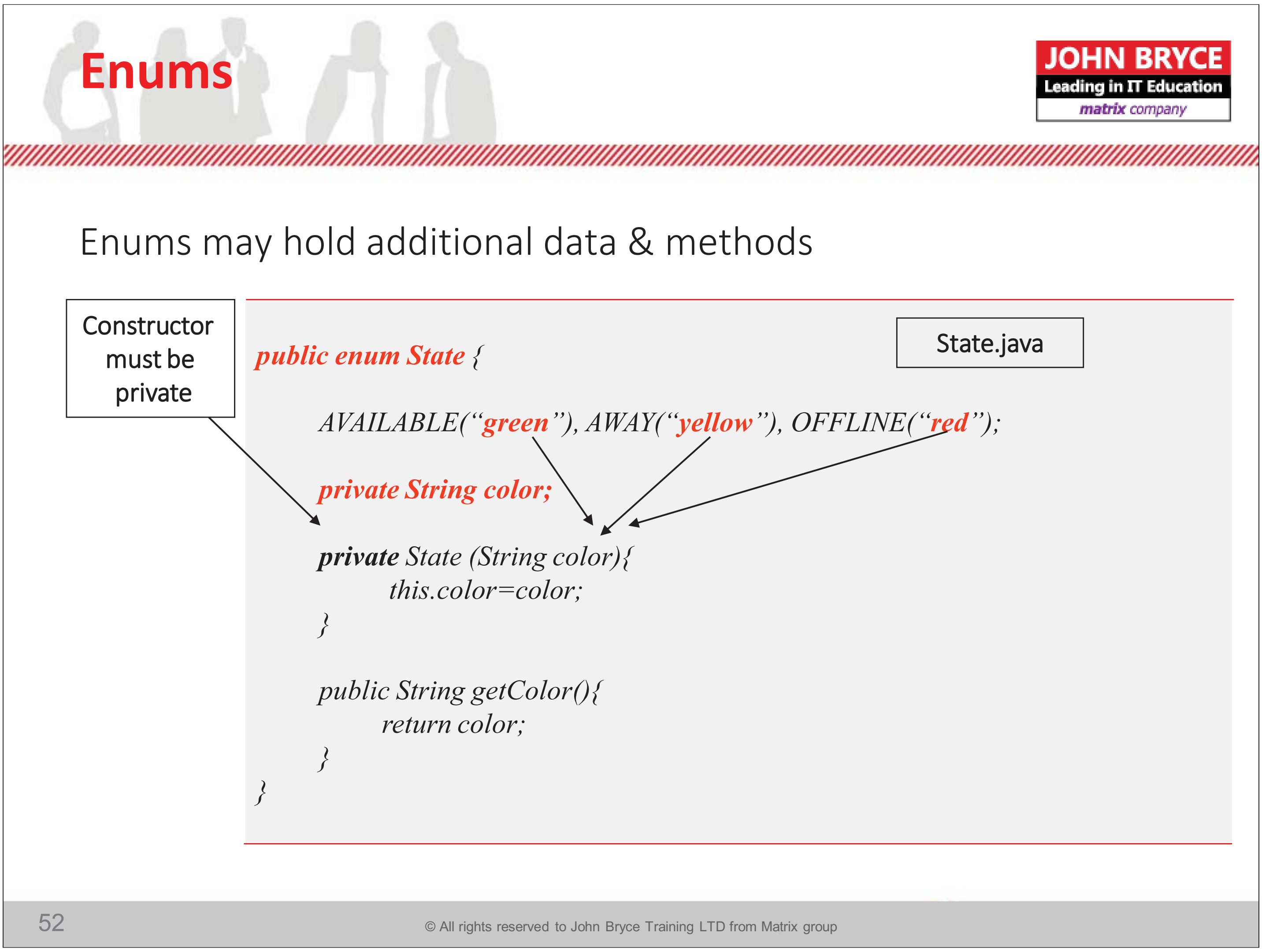
Enums

Some features of enums

- Using enums in switch block

```
public class Client {

        public enum State {AVAILABLE, AWAY, OFFLINE)}

        private State currState;
        …
        public void setClientState(){
                switch (currState){
                        case AVAILABLE: //set client to available state
                        case AWAY: //set client to away state              break;
                      case OFFLINE: //set client to offline state          break;
                }
        } …
```

Note that Java knows the enum type of the switch cases since *currState* is a State enum type



Enums

Enums may hold additional data & methods

State.java

```
public enum State {

        AVAILABLE("green"), AWAY("yellow"), OFFLINE("red");

        private String color;

        private State (String color){
                this.color=color;
        }

        public String getColor(){
                return color;
        }
}
```

Constructor must be private

Enums

Some points to remember:

- Enums cannot be inherited
- Enums constructor cannot be invoked programmatically (Done only by the compiler)
- All Enums are of type java.lang.Enum

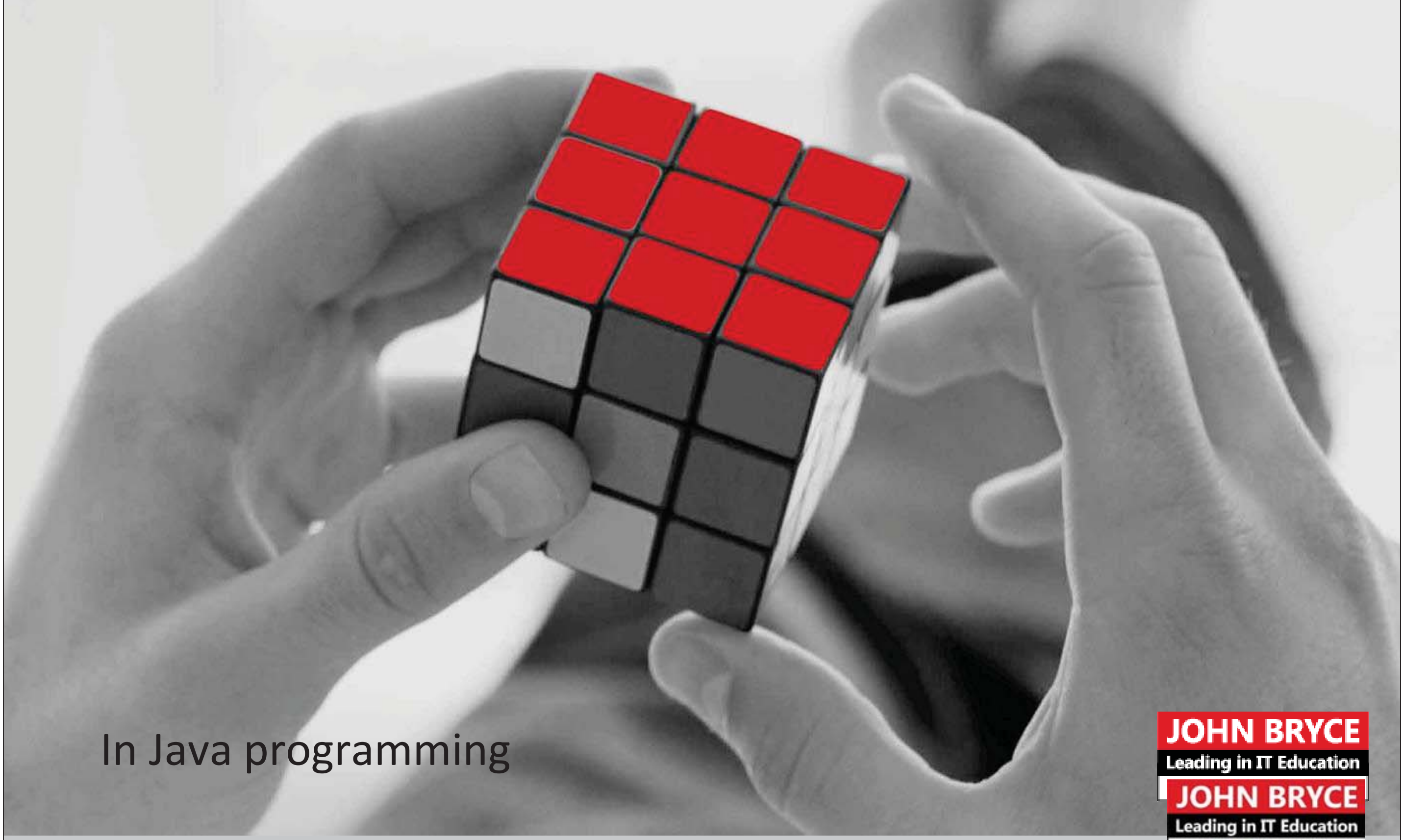
```
….
Enum e = State.AWAY;
…
String name=e.name();
int index=e.ordinal();
Class<State> class=e.getDeclaredClass();
…
```

- clone() isn’t supported – throws CloneNotSupportedException

Summary

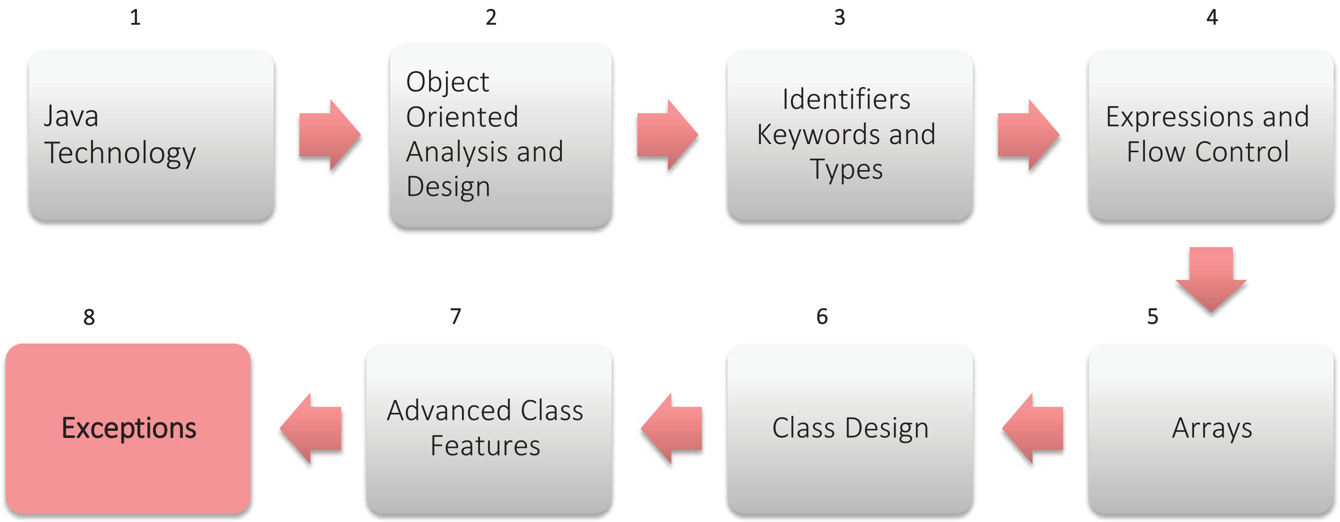
- Static
- Final
- Abstract
- Interface
- Enums

Exceptions Handling



In Java programming

Objectives



1 Java Technology

2 Object Oriented Analysis and Design

3 Identifiers Keywords and Types

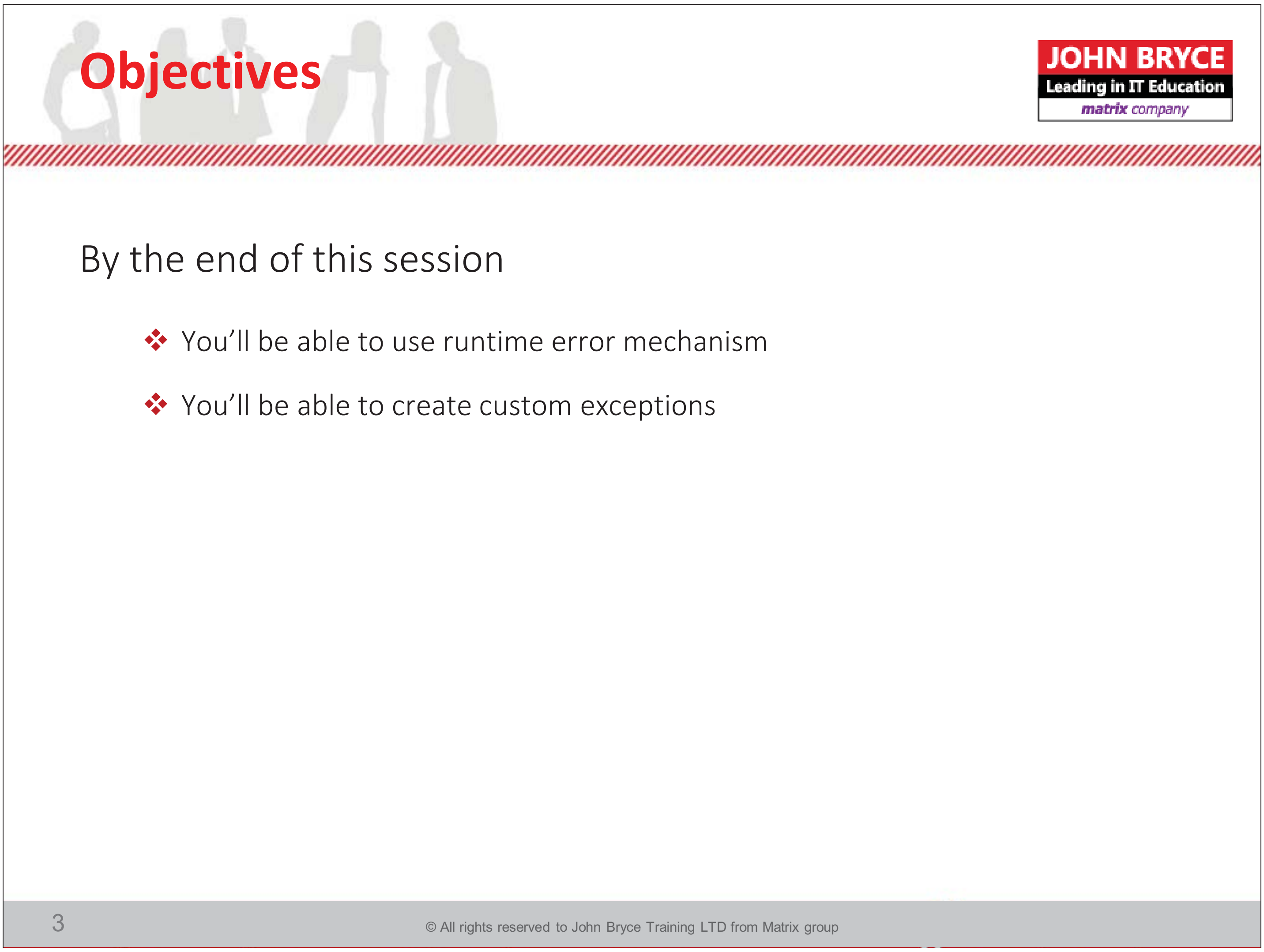
4 Expressions and Flow Control

5 Arrays

6 Class Design

7 Advanced Class Features

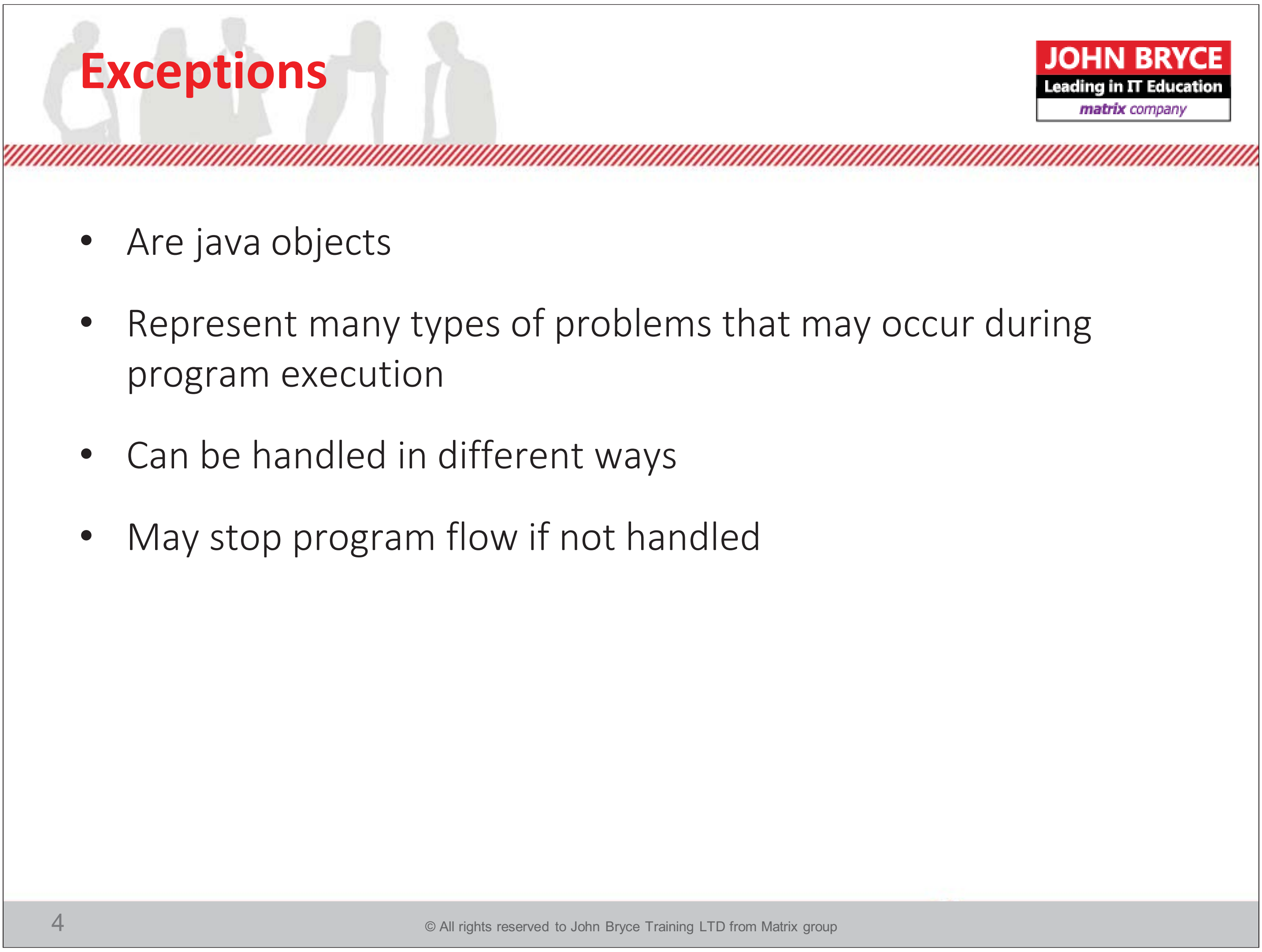
8 Exceptions



Objectives

By the end of this session

- You'll be able to use runtime error mechanism
- You'll be able to create custom exceptions



Exceptions

- Are java objects
- Represent many types of problems that may occur during program execution
- Can be handled in different ways
- May stop program flow if not handled

Type of Exceptions

Error | defines serious error conditions

Exception | – program errors

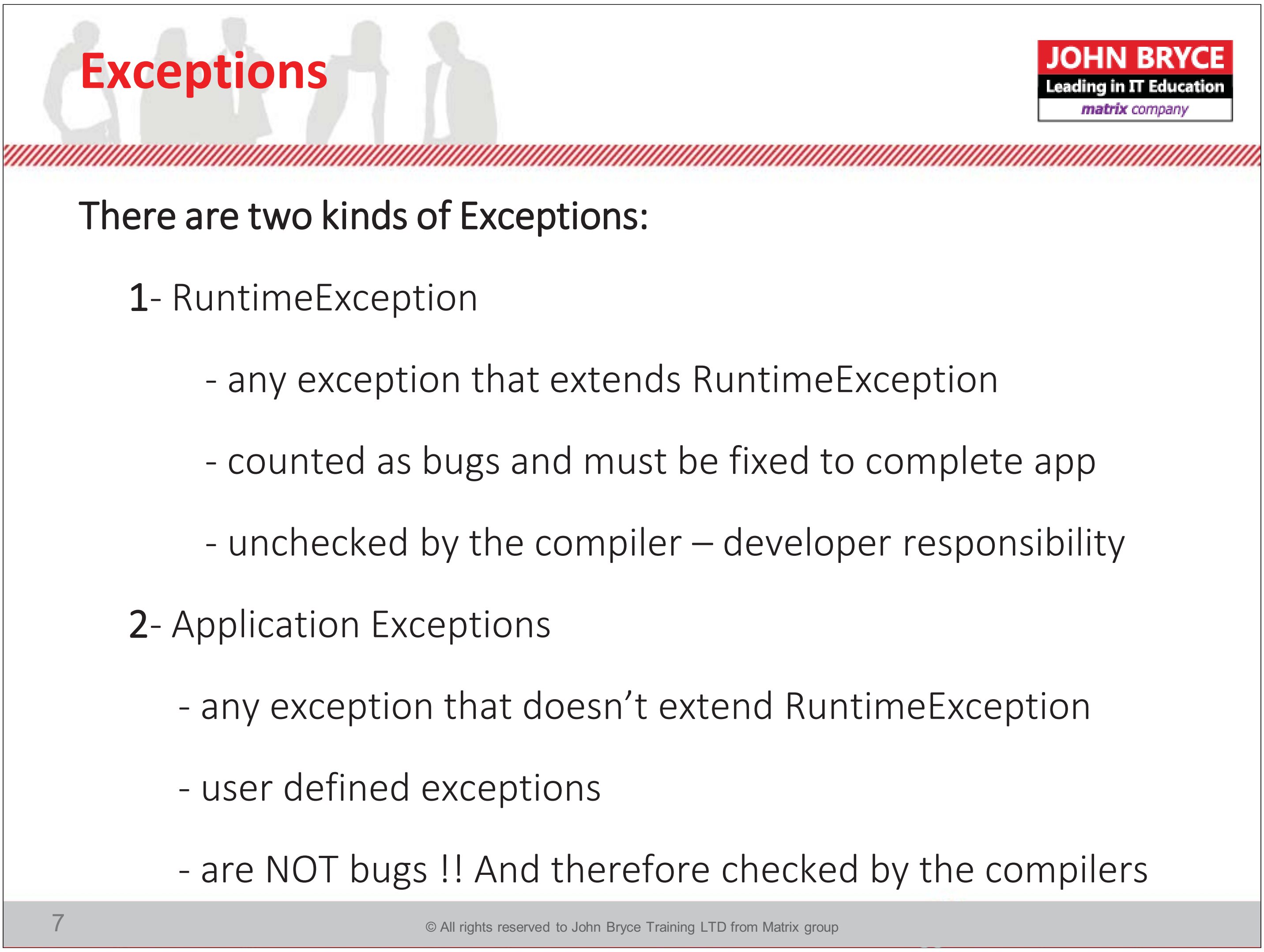
Run-time Exception

Others - I/O

- SQL
- other customized application exception

Errors

- Used to indicated problems that mostly cannot be fixed in runtime
- AWTError
- VirtualMachineError
    - StackOverflowError
    - OutOfMemoryError
    - InternalError (unexpected problem in the VM)



Exceptions

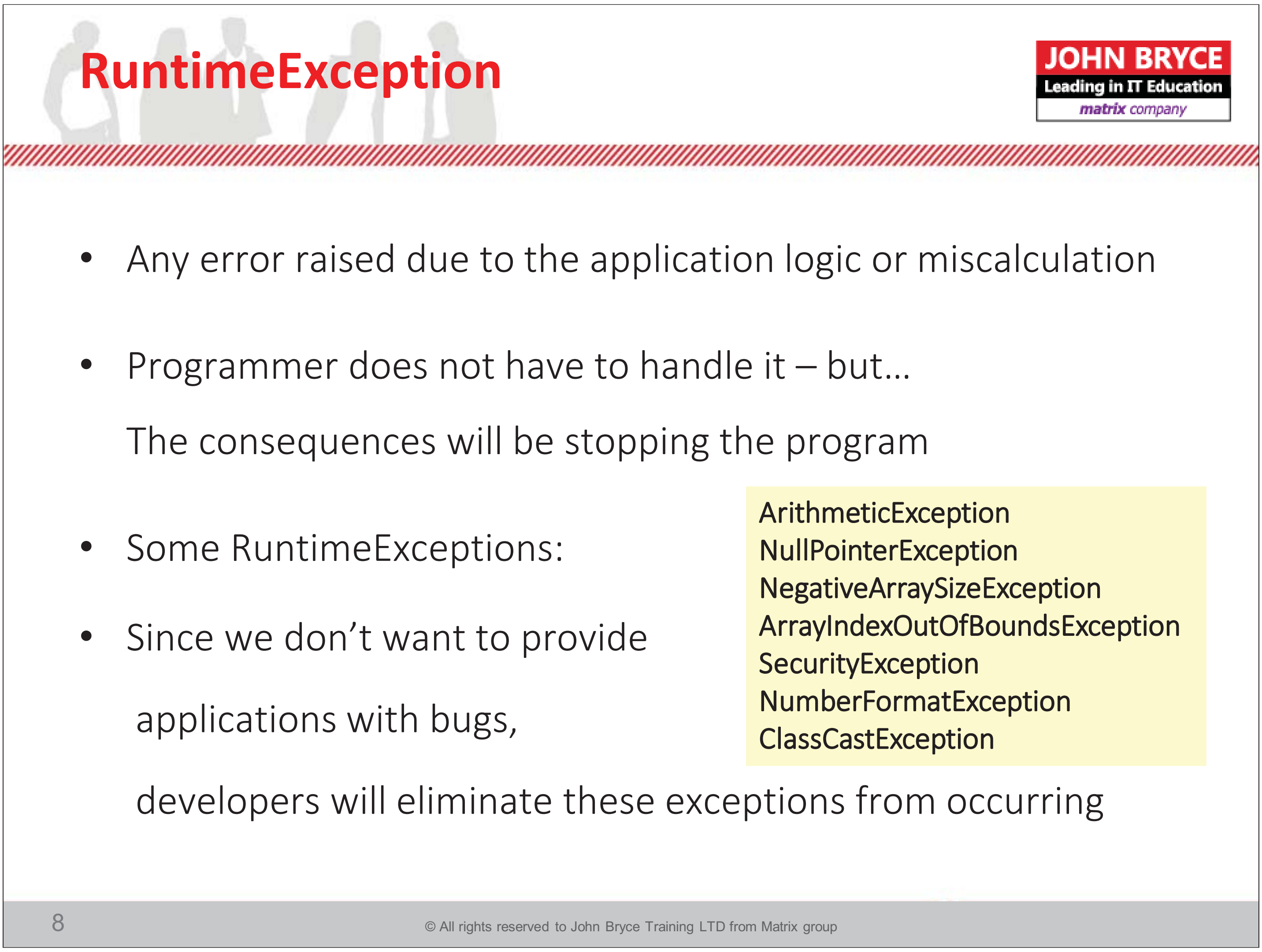
There are two kinds of Exceptions:

1- RuntimeException

- any exception that extends RuntimeException
- counted as bugs and must be fixed to complete app
- unchecked by the compiler – developer responsibility

2- Application Exceptions

- any exception that doesn’t extend RuntimeException
- user defined exceptions
- are NOT bugs !! And therefore checked by the compilers



RuntimeException

- Any error raised due to the application logic or miscalculation
- Programmer does not have to handle it – but…

  The consequences will be stopping the program
- Some RuntimeExceptions:
- Since we don’t want to provide

  applications with bugs,

  developers will eliminate these exceptions from occurring

ArithmeticException
NullPointerException
NegativeArraySizeException
ArrayIndexOutOfBoundsException
SecurityException
NumberFormatException
ClassCastException

Application Exception

- Usually thrown when any external implementation is involved – such as:

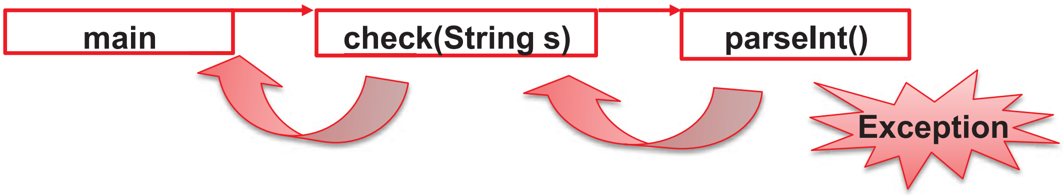
File System
JDBC Drivers
XML Parsers

- Programmer must handle it (compilation error)
- Some ApplicationExceptions:

IOException [EOFException, FileNotFoundException…]
SQLException
DOMException, SAXException
ClassNotFoundException
RemoteException
AWTException

Call Stack Mechanism

- both exceptions and errors extend Throwable
- It allows a method that generates exception to ‘throw’ it up the stack
- If the thrown exception reaches main() without being handled – the program stops

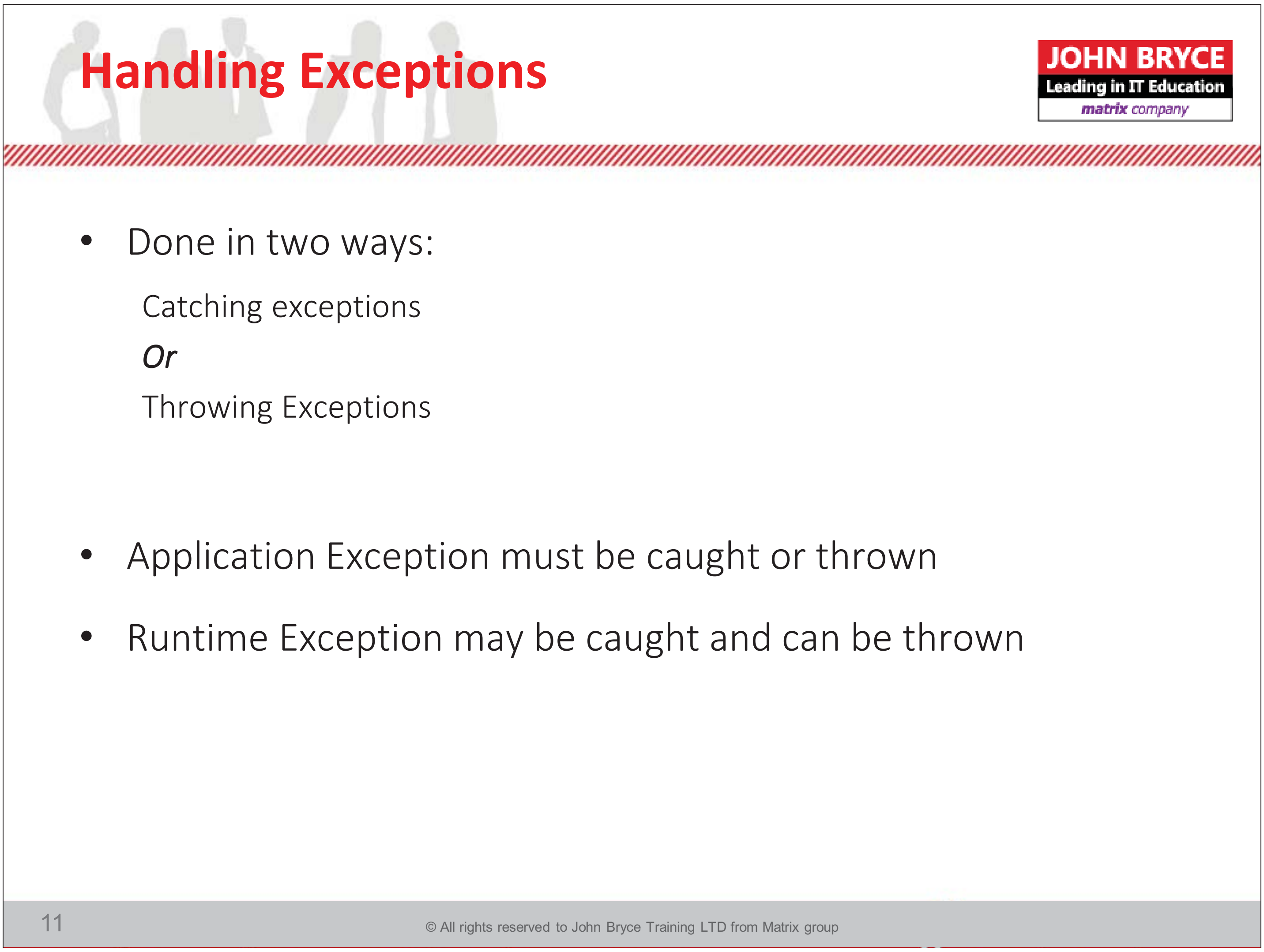


main

check(String s)

parseInt()

Exception



Handling Exceptions

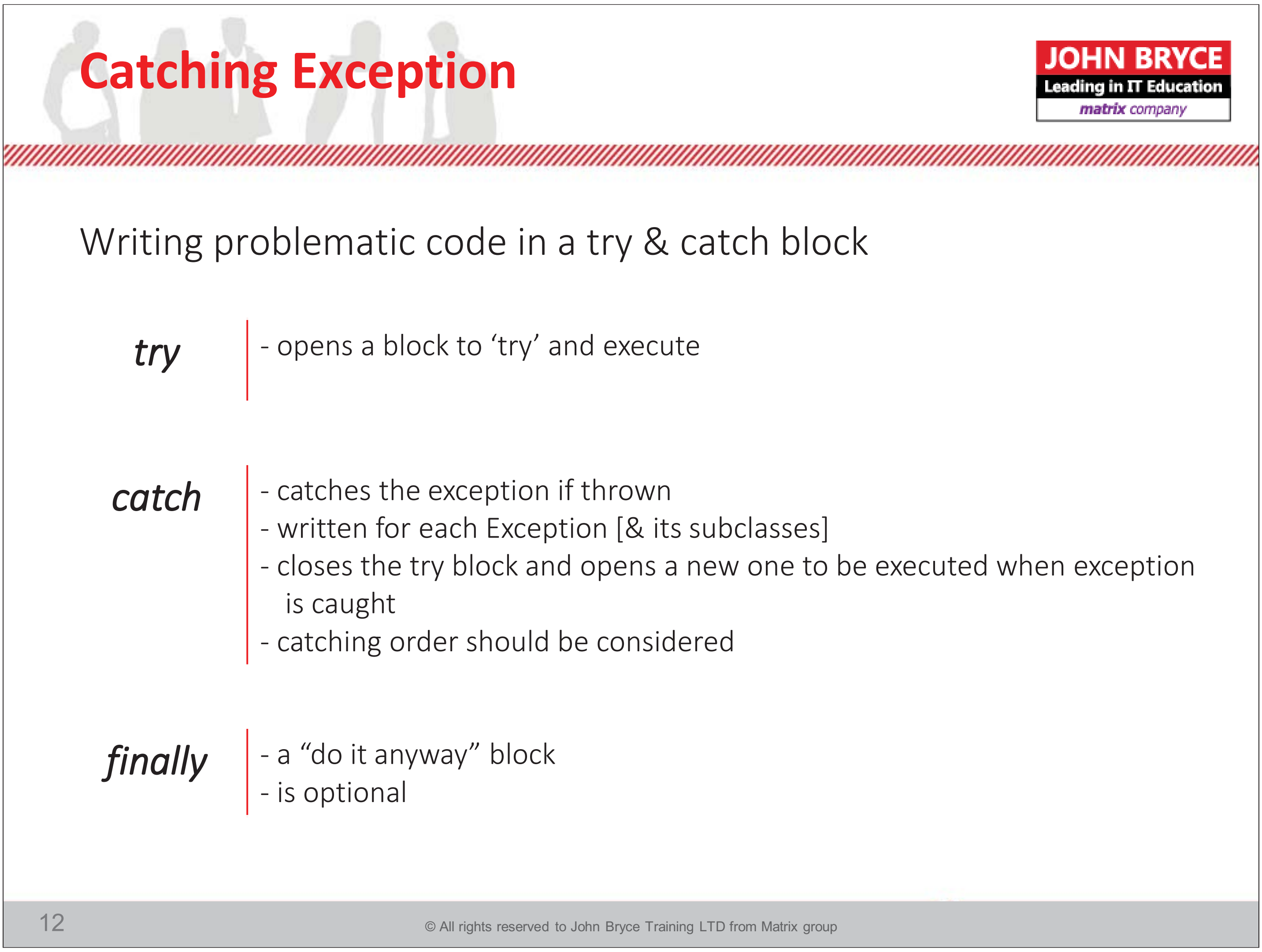
- Done in two ways:

  Catching exceptions

  Or

  Throwing Exceptions

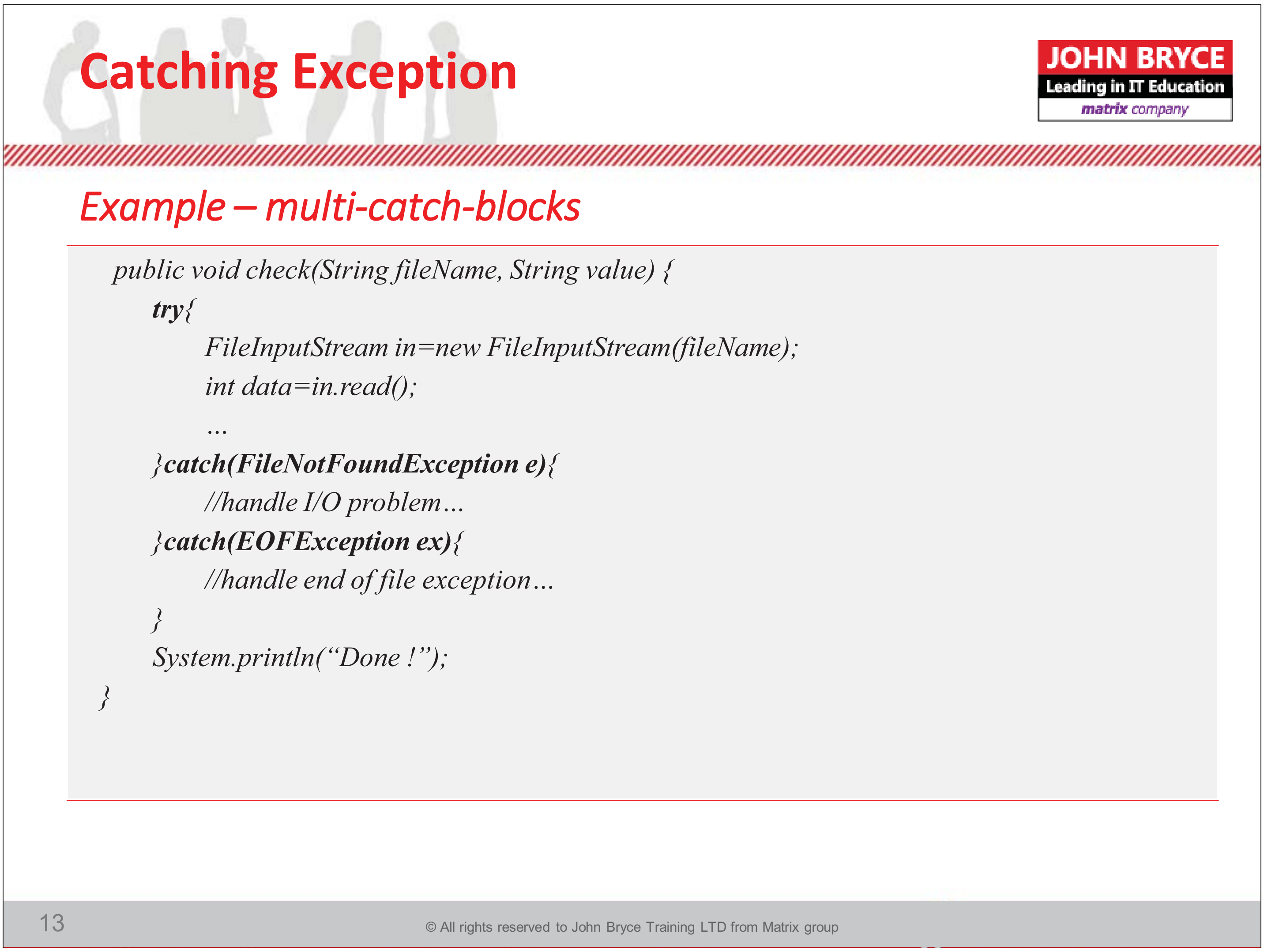
- Application Exception must be caught or thrown
- Runtime Exception may be caught and can be thrown



Catching Exception

Writing problematic code in a try & catch block

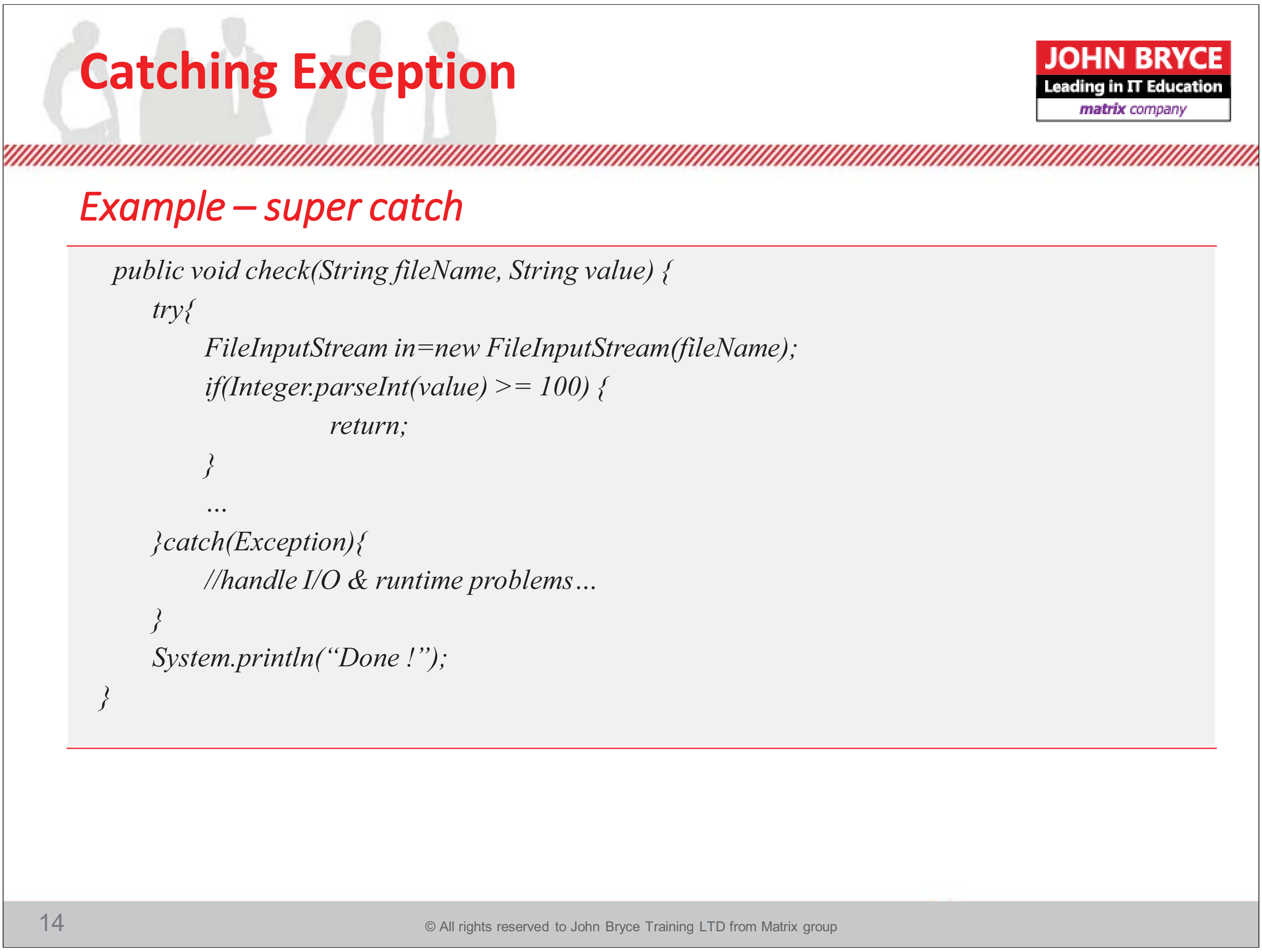
try	- opens a block to ‘try’ and execute
catch	- catches the exception if thrown - written for each Exception [& its subclasses] - closes the try block and opens a new one to be executed when exception is caught - catching order should be considered
finally	- a “do it anyway” block - is optional



Catching Exception

Example – multi-catch-blocks

```
    public void check(String fileName, String value) {
        try{
            FileInputStream in=new FileInputStream(fileName);
            int data=in.read();
            …
        }catch(FileNotFoundException e){
            //handle I/O problem…
        }catch(EOFException ex){
            //handle end of file exception…
        }
        System.println("Done !");
  }
```



Catching Exception

Example – super catch

```
    public void check(String fileName, String value) {
        try{
            FileInputStream in=new FileInputStream(fileName);
            if(Integer.parseInt(value) >= 100) {
                        return;
            }
            …
        }catch(Exception){
            //handle I/O & runtime problems…
        }
        System.println("Done !");
  }
```

Catching Exception

Example – finally

```
    public void check(String fileName, String value) {
        try{
            FileInputStream in=new FileInputStream(fileName);
            if(Integer.parseInt(value) >= 100) {
                    return;
            }
            …
        }catch(FileNotFoundException){
            //handle I/O problem…
            return;
        }catch(NumberFormatException){
            //handle runtime exception…
        }finally{
            System.println(“This is printed in any case….”);
        }
        System.println(“Done !”);
   }
```

Catching Exception

- Should be written as close to the origin throwing point as possible
- Catching `java.lang.Exception` will catch all types of exceptions
- Use `java.lang.Exception` methods to get information:

Exception

• getMessage()	Returns a message describes this exception
• printStackTrace (out)	Prints the stack trace – good for debugging

```
}catch(Exception e){  System.out.println(e.getMessage());
        e.printStackTrace(System.out);
}
```

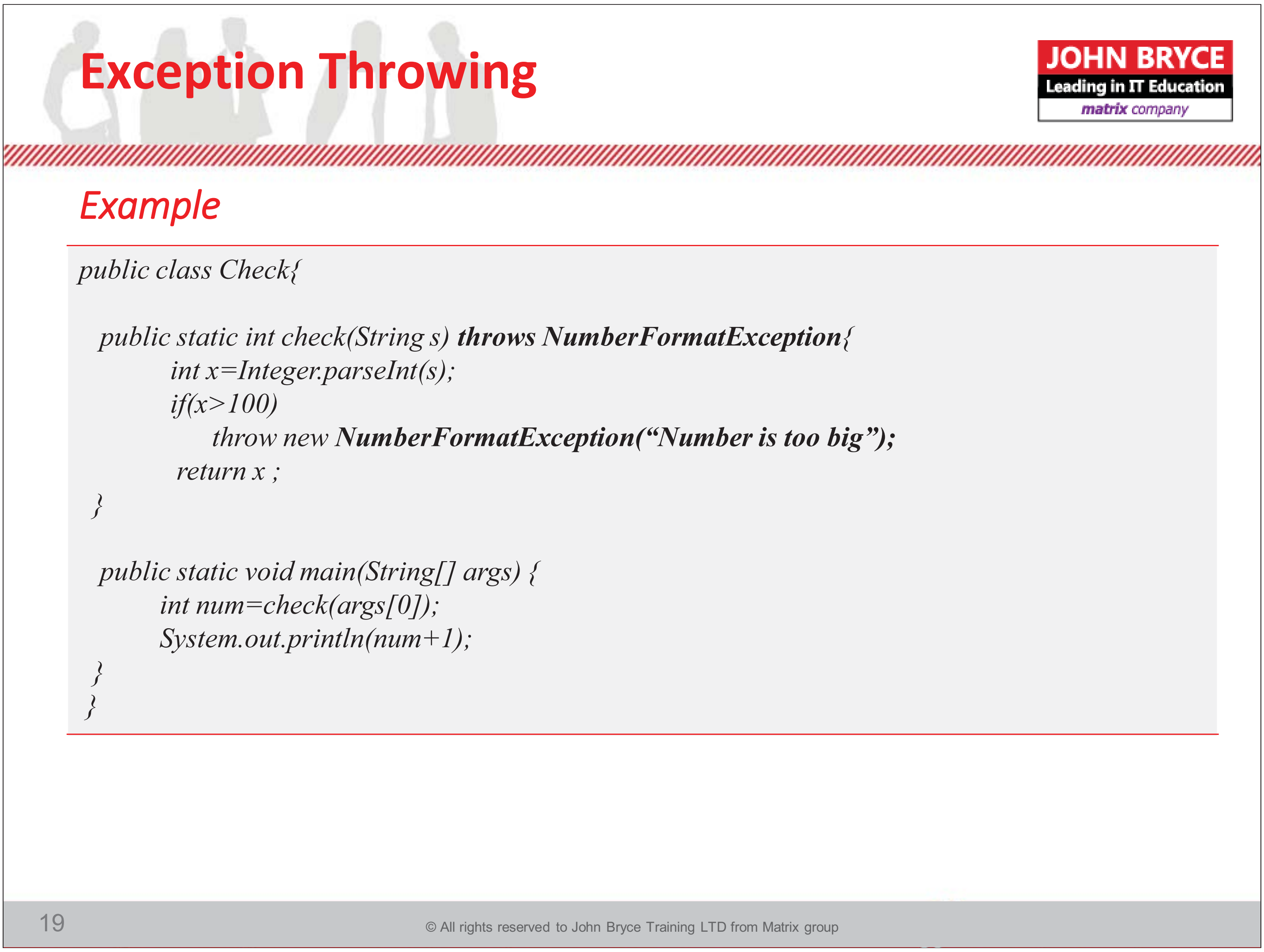
Exception Throwing

- Any method can delegate exceptions to the caller
- A method must declare any thrown Exception as part of its signature
- Throwing Runtime Exceptions is allowed but not always necessary
- **throws** - declares all thrown exceptions
- **throw** - actually creates an exception and throw it

Exception Throwing

Example

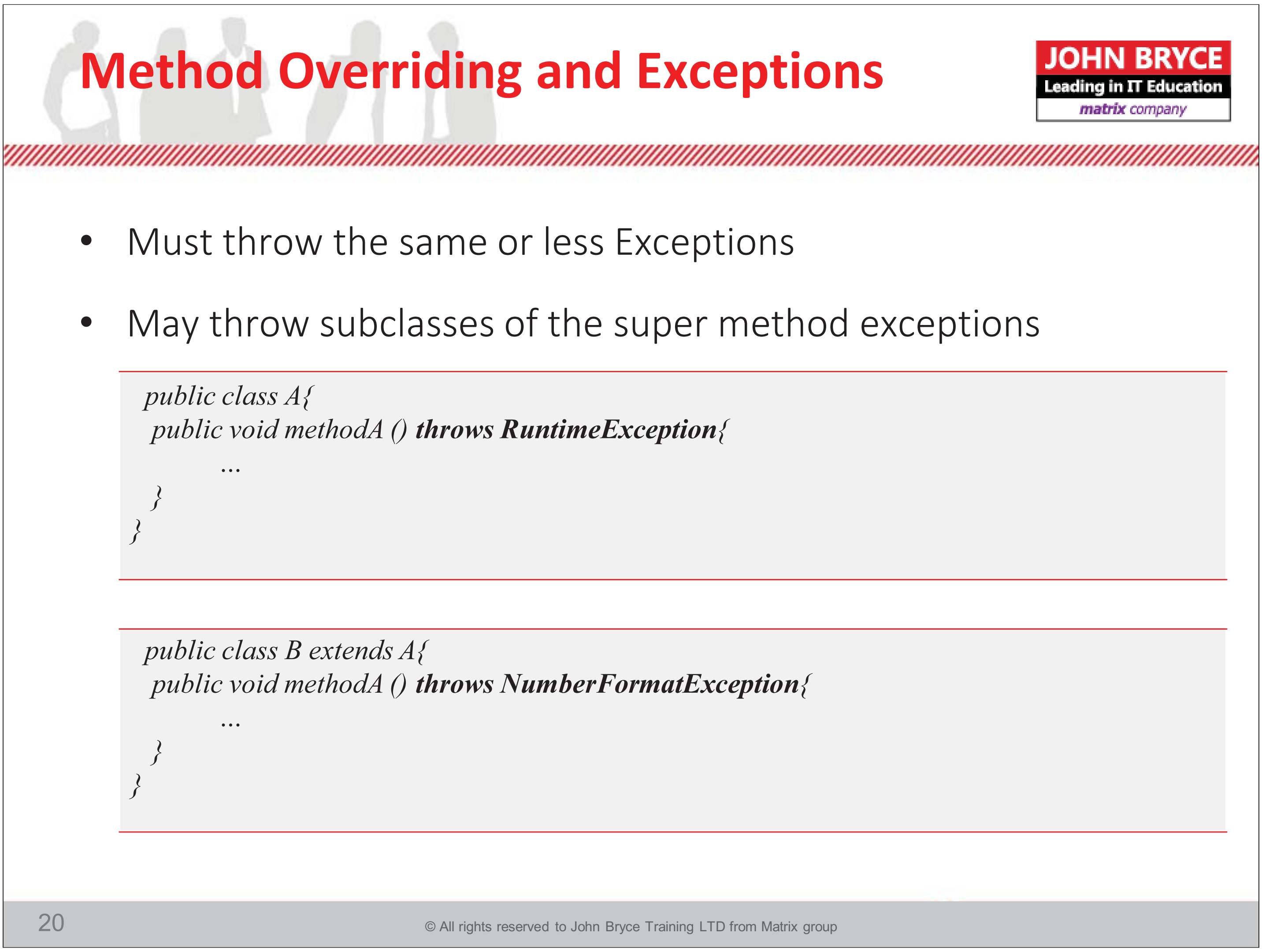
```
public class Check{

 public static int check(String s) throws NumberFormatException{
          return Integer.parseInt(s);
 }

 public static void main(String[] args) {
         int num=check(args[0]);
         System.out.println(num+1);
 }
}
```



Exception Throwing

Example

```
public class Check{

  public static int check(String s) throws NumberFormatException{
          int x=Integer.parseInt(s);
          if(x>100)
               throw new NumberFormatException("Number is too big");
           return x ;
 }

  public static void main(String[] args) {
         int num=check(args[0]);
         System.out.println(num+1);
 }
}
```



Method Overriding and Exceptions

- Must throw the same or less Exceptions
- May throw subclasses of the super method exceptions

```
public class A{
 public void methodA () throws RuntimeException{
          …
  }
}
```

```
public class B extends A{
 public void methodA () throws NumberFormatException{
          …
  }
}
```

Method Overriding and Exceptions

More examples

```
public class A{
  public void methodA () throws RuntimeException{
          …
  }
}
```

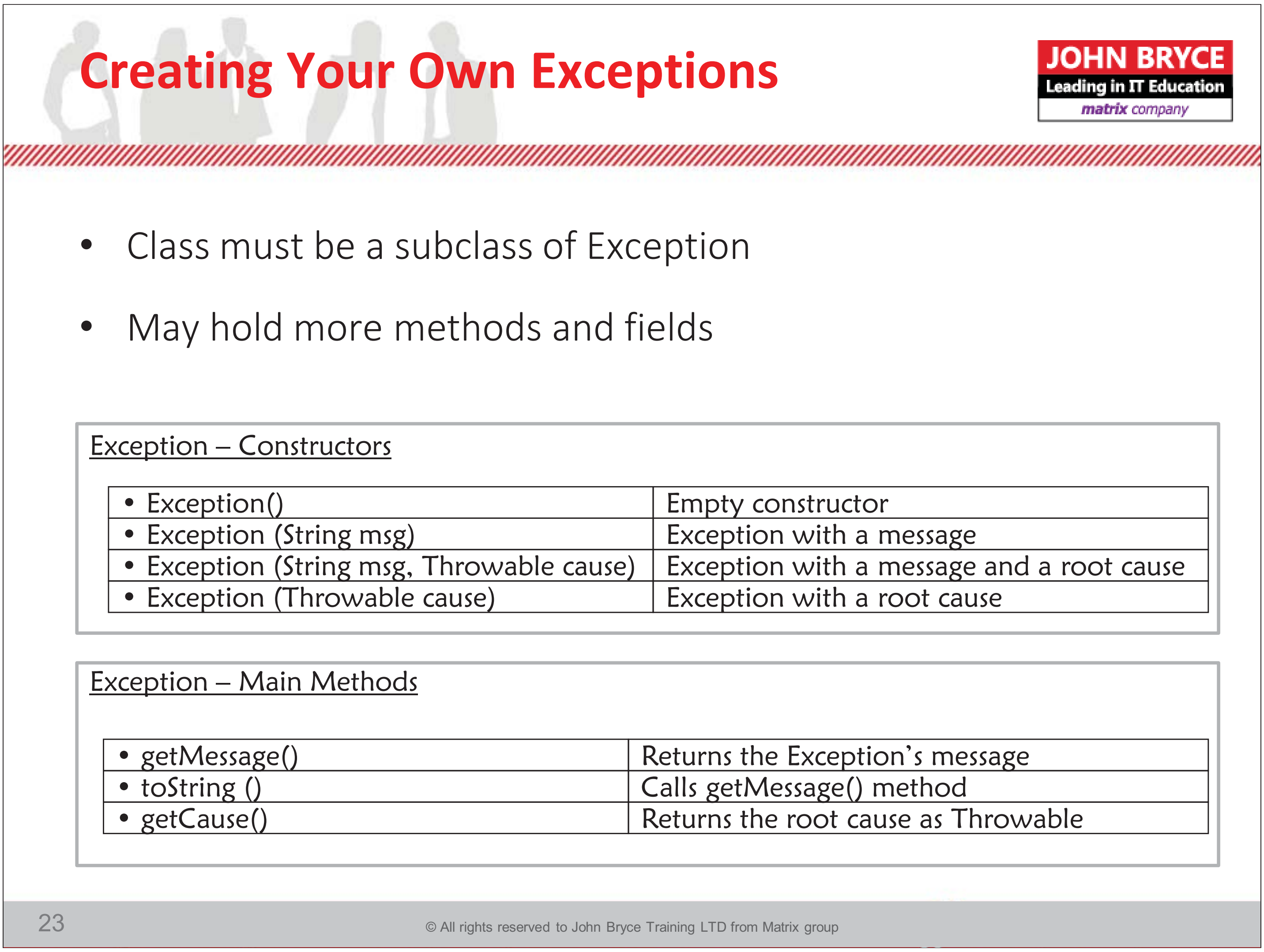
```
public class B extends A{
  public void methodA () throws NumberFormatException, SecurityException{
          …
  }
}
```

Method Overriding and Exceptions

More examples

```
public class A{
  public void methodA () throws RuntimeException, IOException{
          …
  }
}
```

```
public class B extends A{
  public void methodA () throws EOFException{
          …
  }
}
```



Creating Your Own Exceptions

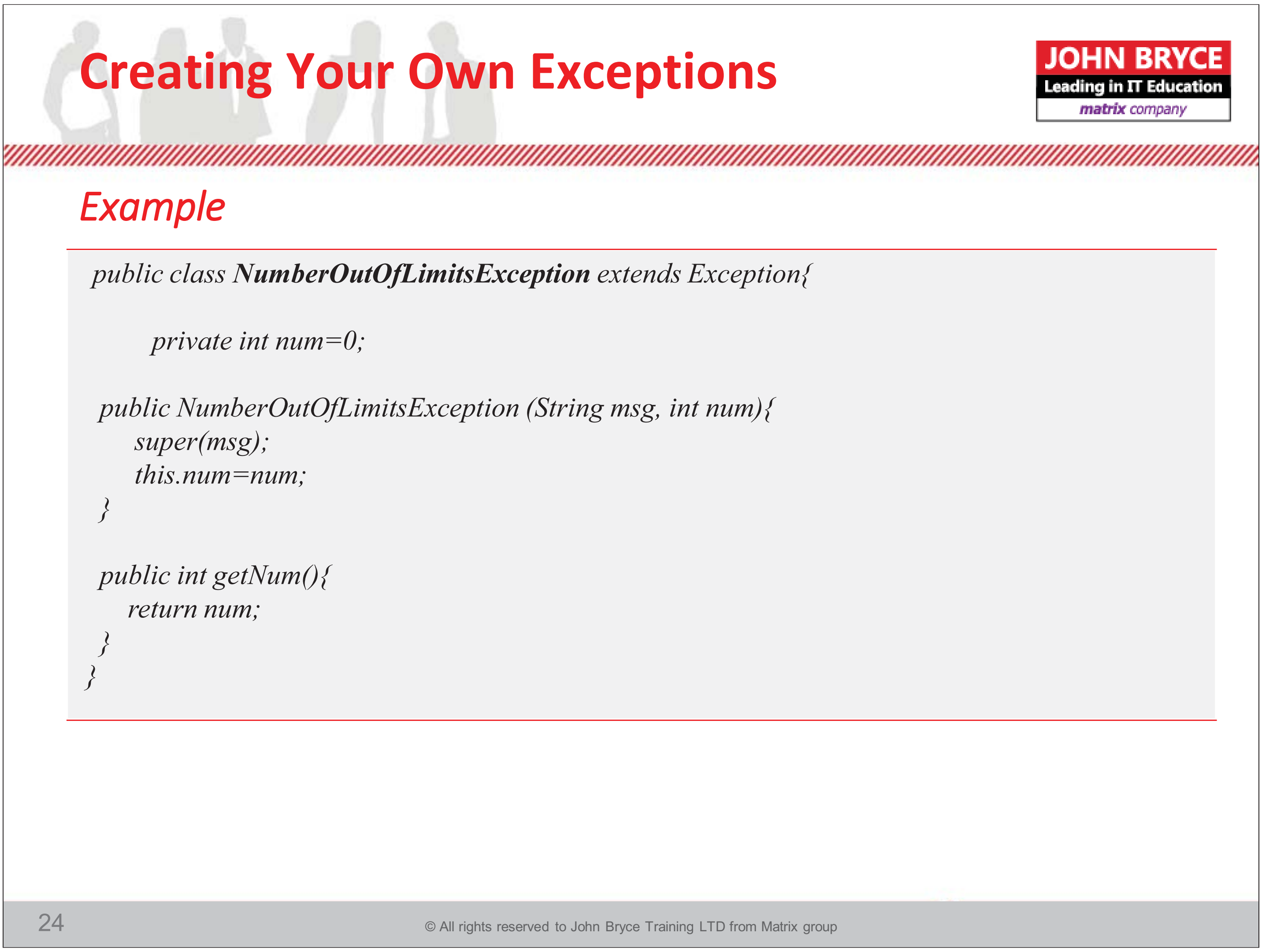
- Class must be a subclass of Exception
- May hold more methods and fields

Exception – Constructors

• Exception()	Empty constructor
• Exception (String msg)	Exception with a message
• Exception (String msg, Throwable cause)	Exception with a message and a root cause
• Exception (Throwable cause)	Exception with a root cause

Exception – Main Methods

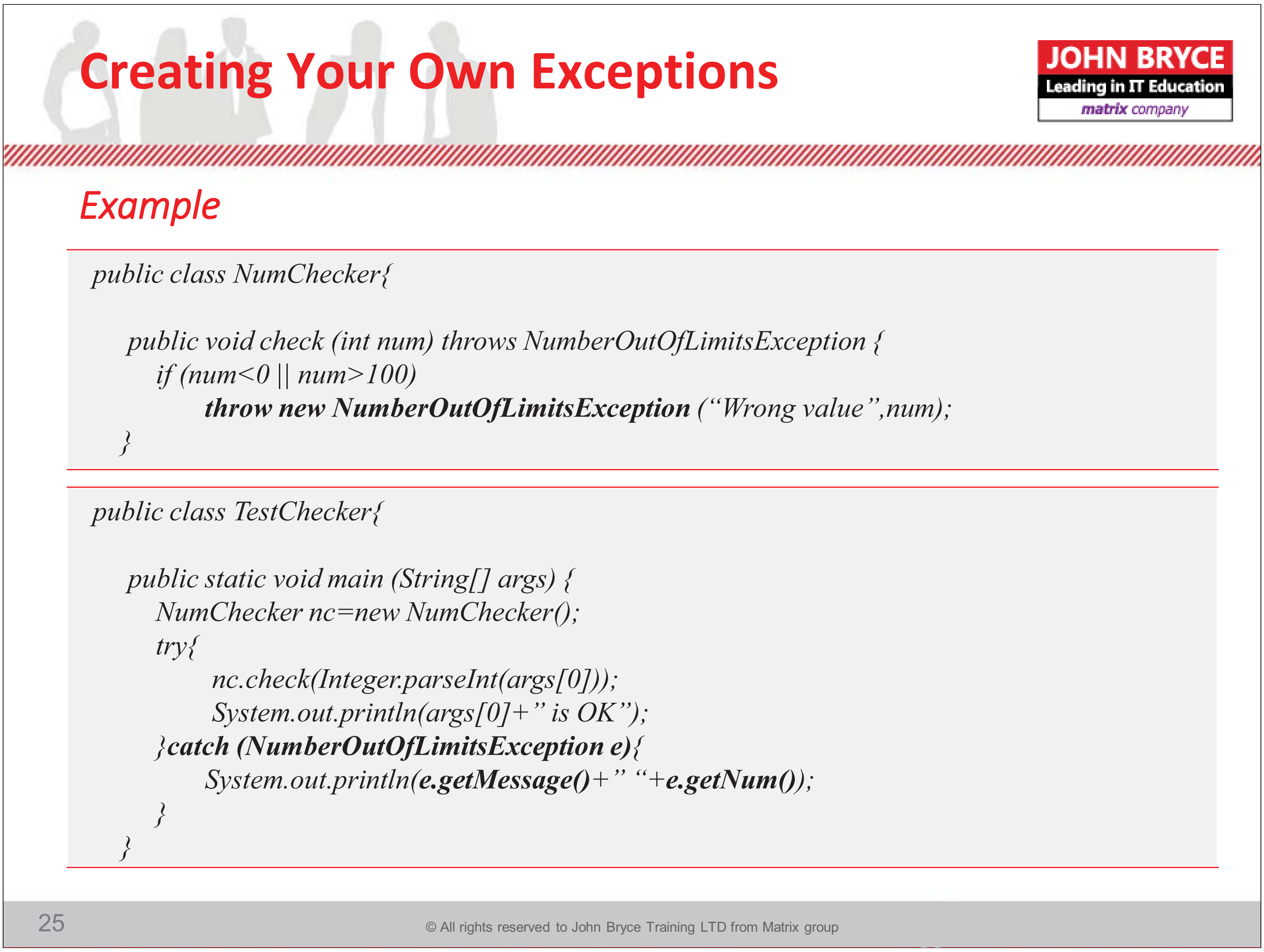
• getMessage()	Returns the Exception's message
• toString ()	Calls getMessage() method
• getCause()	Returns the root cause as Throwable



Creating Your Own Exceptions

Example

```
public class NumberOutOfLimitsException extends Exception{

        private int num=0;

  public NumberOutOfLimitsException (String msg, int num){
       super(msg);
       this.num=num;
  }

  public int getNum(){
      return num;
  }
}
```

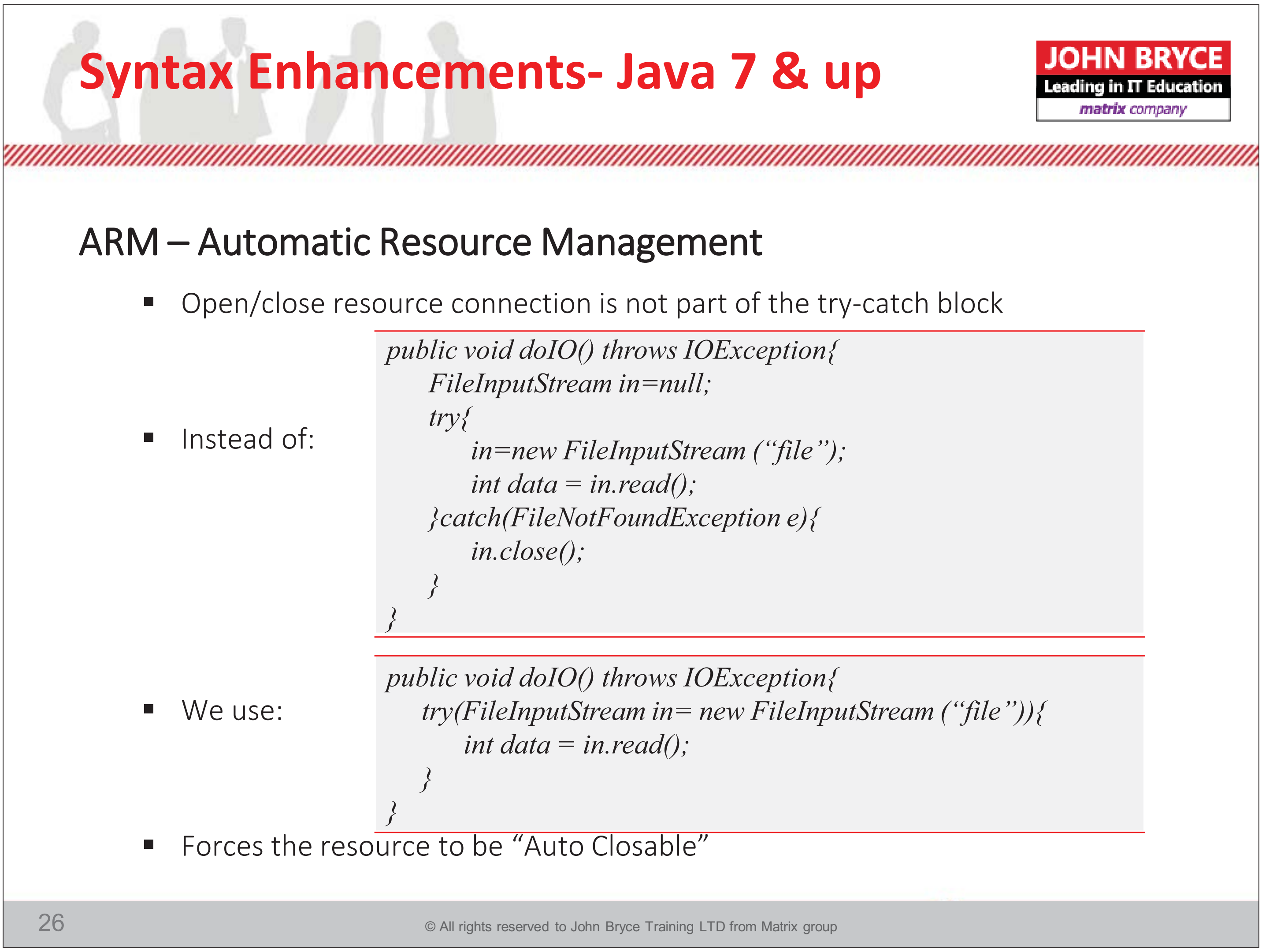


Creating Your Own Exceptions

Example

```
public class NumChecker{

    public void check (int num) throws NumberOutOfLimitsException {
       if (num<0 || num>100)
            throw new NumberOutOfLimitsException (“Wrong value”,num);
    }
```

```
public class TestChecker{

    public static void main (String[] args) {
       NumChecker nc=new NumChecker();
       try{
            nc.check(Integer.parseInt(args[0]));
            System.out.println(args[0]+” is OK”);
       }catch (NumberOutOfLimitsException e){
            System.out.println(e.getMessage()+” “+e.getNum());
       }
    }
```



Syntax Enhancements- Java 7 & up

ARM – Automatic Resource Management

- Open/close resource connection is not part of the try-catch block
- Instead of:

```
public void doIO() throws IOException{
     FileInputStream in=null;
     try{
          in=new FileInputStream (“file”);
          int data = in.read();
     }catch(FileNotFoundException e){
          in.close();
     }
}
```

- We use:

```
public void doIO() throws IOException{
     try(FileInputStream in= new FileInputStream (“file”)){
          int data = in.read();
     }
}
```

- Forces the resource to be “Auto Closable”

Syntax Enhancements

ARM – Automatic Resource Management

- Closable.close() method throws IO exception
- In order to use ARM for other APIs as well – an AutoClosable super interface was created

AutoCloseable close() method throws a generat Exception

Closeable now extends it

```
public  interface AutoClosable {
            public void close () throws Exception;
}
```

```
public  interface Closable extends AutoClosable {
            public void close () throws IOException;
}
```

Syntax Enhancements

More on ARM

- Manages “AutoCloseable” implementations only(!)
- Whether try block pass or fails – close() will be invoked
- Can declare and use more than one resource:

```
try(FileInputStream in= new FileInputStream (“file1”);
    FileOutputStream out= new FileOutputStream(“file2”) ){
          int data = in.read();
          out.write(data);
}
```

- Close() method is called according to resource declaration order in the try clause

Syntax Enhancements

Multi-catch

- Relating to different exceptions in a single catch block

Instead of:

```
try{
        FileInputStream in=new FileInputStream (“file”);
        Connection con = DriverManager.getConnection(….);
        ….in.read();
        ….con.createStatement();
}catch(IOException e){
        ….
}catch(SQLException e){
        ….
}
```

We use:

```
try{
        …..
}catch(IOException | SQLException e){ …. }
```

Summary

- Exceptions type
    - Runtime
    - Exceptions
- Handling exceptions
    - Catching
    - Throwing
- Method override and exceptions
- Create exception
- Use ARM & Multicatch

Exceptions