Rounding Out Your Java Knowledge

Object Oriented Programming
2016375 - 5

Camilo López

Outline

- The Object Nature of Strings
- Message Chains
- Enums (Enumerations)
- Features of the Object Class
- Java Exception Handling
- Behind the Scenes of the JVM
- Javadoc Comments
- Java Archive (JAR) Files

Java-Specific Terminology

Generic OO Terminology Used in this Book	Formal Java-Specific Terminology Used by Sun Microsystems	Used to Describe the Following Notion
attribute	field, instance variable	A variable that is created once per object—that is, per each instance of a class. Each object has its own separate set of instance variables.
static variable (informal: static attribute)	static field, class variable	A variable that exists only once per class.
method	instance method	A function that is invoked on an object.
static method	class method	A function that can be called on a class as a whole, without reference to a specific object. Class methods can neither call instance methods nor access instance variables.
feature	member	Those components of a class that can potentially be inherited—for example, instance/class variables and instance/class methods, but ***not*** constructors.

The Object Nature of Strings

```
String x = “Foo”;
String y = “bar”;
String z = x + y + “!”;
```

- String is a reference type

- Variables x, y, and z refers to Objects of type String

How can we take advantage of this?

Think about the numerous methods that are declared by the String class for manipulating Strings

The Object Nature of Strings

Features

```
String x = "Foo";
String y = x + "bar";
```

- int length()

```
int tres = x.length();
```

- boolean startsWith(String s)
- boolean endsWith(String s)

```
if (x.startsWith("fo")){…}
if (x.endsWith("o")){…}

if (y.startsWith('b')){…}
```

compiler ERROR!!! 

The Object Nature of Strings

Features

```
String x = “Foo”;
String y = x + “bar”;
```

- boolean contains(String s)

```
if (y.contains(x){…}
if (y.contains(“oob”)){…}
```

- int indexOf(String s)

```
int posOob = s.indexOf(“oob”);
```

- String replace(char old, char new)

```
String z = y.replace(‘o’, ‘a’);
```

The Object Nature of Strings

Features

```
String x = “Foo”;
String y = x + “bar”;
```

- String substring(int i)

```
String z = y.substring(3);
```

- String substring(int I, int j)

```
String z = y.substring(2,3);
```

- char charAt(int index)

```
for (int i = 0; i < y.length(); i++) {
    System.out.println(y.charAt(i));
}
```

The Object Nature of Strings

Features

```
String x = “Foo”;
String y = x + “bar”;
```

- boolean equals(String s)

```
String z = “Foo”;

if (x.equals(y)){…}
if (x.equals(z)){…}

if (x == z) {…}
```

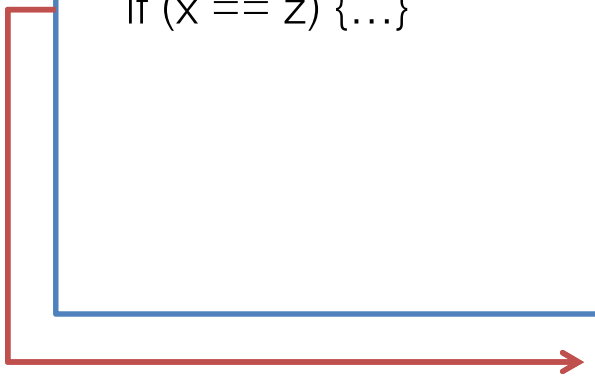
The Object Nature of Strings

Features

```
String x = “Foo”;
String y = x + “bar”;
```

- boolean equals(String s)

```
String z = “Foo”;

if (x.equals(y)){…}
if (x.equals(z)){…}

if (x == z) {…}
```



These are references

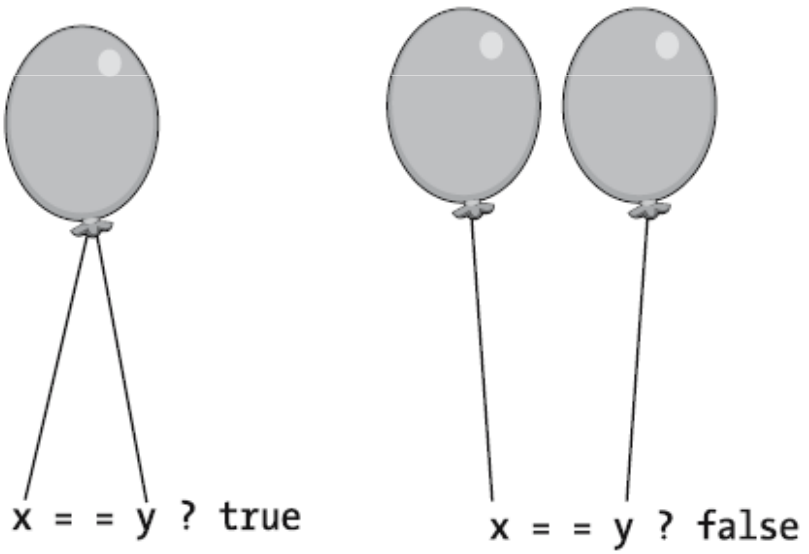
The Object Nature of Strings

Features

```
String x = “Foo”;
String y = x + “bar”;
```

- boolean equals(String s)

```
String z = “Foo”;

if (x.equals(y)){…}
if (x.equals(z)){…}

if (x == z) {…}
```



These are references

The Object Nature of Strings

Strings are Immutable

```
String x = “Foo”;
```

The Object Nature of Strings

Strings are Immutable

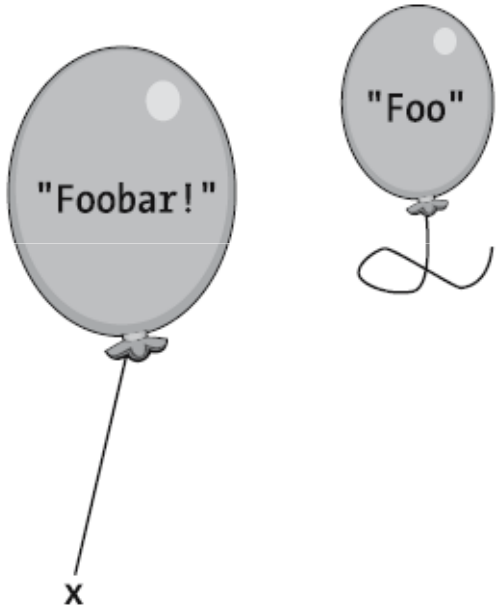
```
String x = “Foo”;
```


```
String x = x + “bar!”;
```

The Object Nature of Strings

Strings are Immutable

String x = “Foo”;



String x = x + “bar!”;

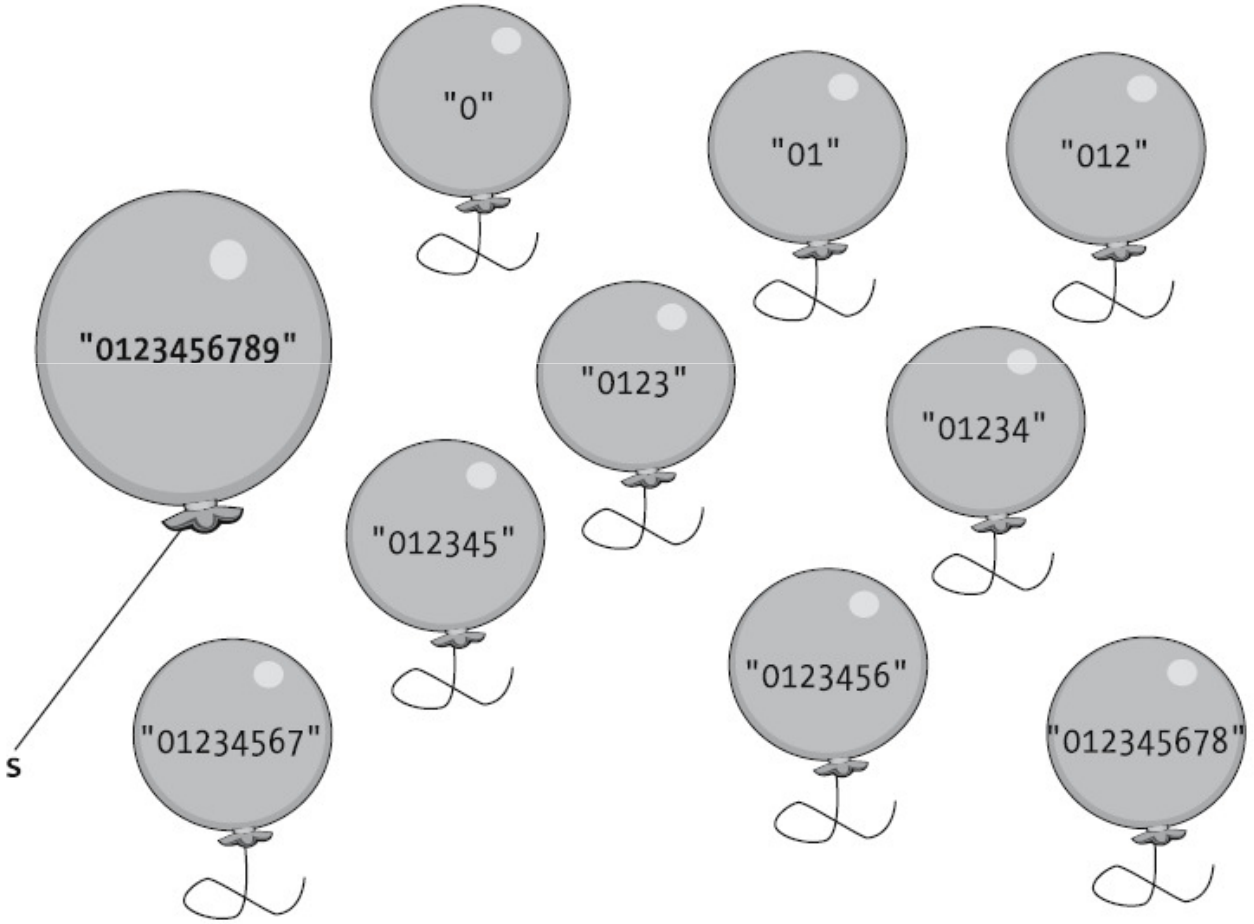
The Object Nature of Strings

Strings are Immutable

```
String s = "";
for (int x = 0; x < 10; x++) {
    // Append another digit to s.
    s = s + x;
}

System.out.println(s);
```

The Object Nature of Strings

Strings are Immutable

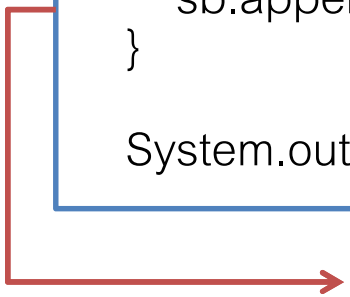


The Object Nature of Strings

The StringBuffer class

```
import java.util.StringBuffer;
```

```
StringBuffer sb = new StringBuffer();
for (int x = 0; x < 10; x++) {
    // Append another digit to s.
    sb.append(x);
}

System.out.println(sb);
```



Only one object was instantiated

The Object Nature of Strings

The StringTokenizer class

```
import java.util.StringTokenizer;
```

```
String s = "This is a test.";
StringTokenizer st = new StringTokenizer(s);
while (st.hasMoreTokens()) {
    System.out.println(st.nextToken());
}
```

```
String date = "11/17/1985";
StringTokenizer st = new StringTokenizer(date, "/");
```

An overloaded form of the constructor,

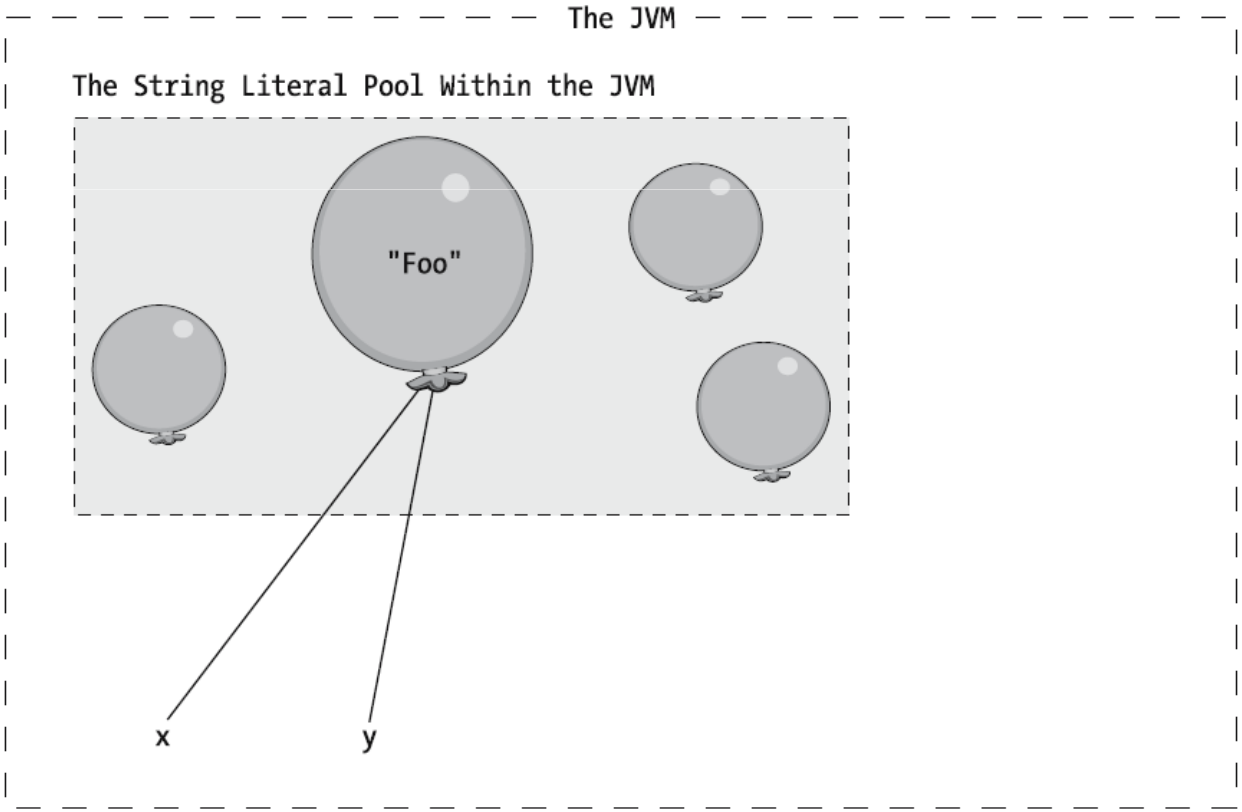
StringTokenizer(String s, String delimiter),

can be used if we want to specify a specific delimiter to be used when parsing a String.

The Object Nature of Strings

Instantiating Strings and the String Literal Pool

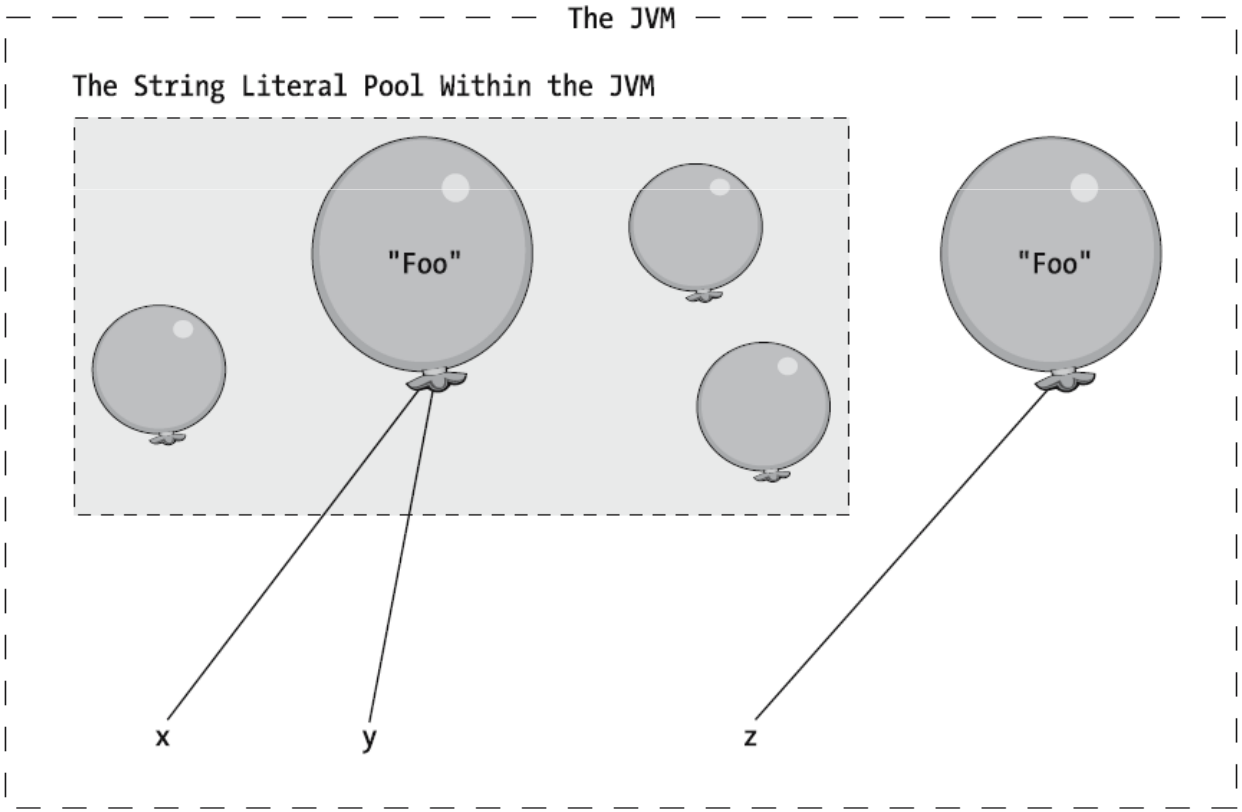
```
String x = “Foo";
String y = “Foo";
```



The Object Nature of Strings

Instantiating Strings and the String Literal Pool

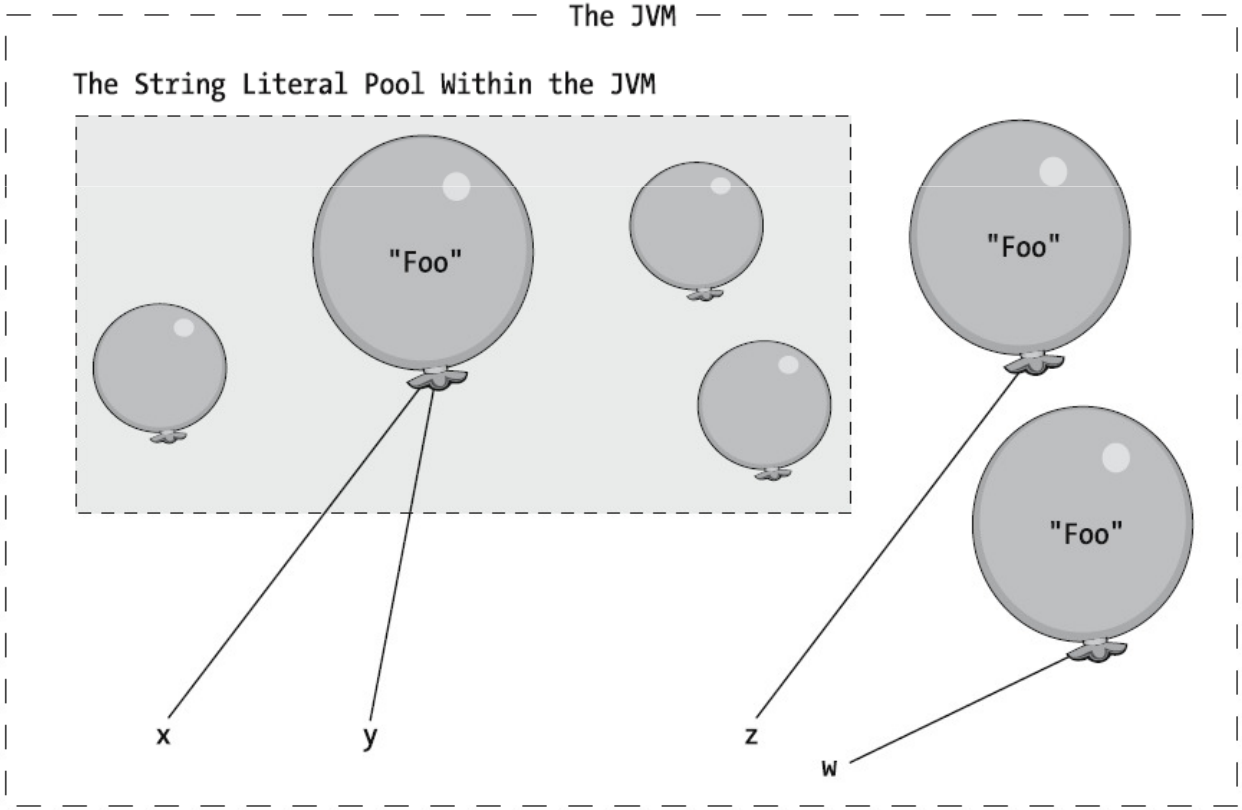
String z = new String(“Foo");



The Object Nature of Strings

Instantiating Strings and the String Literal Pool

```
String z = new String("Foo");
String w = new String("Foo");
```



Message Chains

- A “chain” of two or more messages concatenated by periods (“dots”): p.getName().length()

Another form of expression

```
Student s = new Student();
Professor p = new Professor();
Department d = new Department();

d.setName("MATH");
p.setDepartment(d);
s.setAdvisor(p);
```

- Let’s evaluate the following line of code

```
s.setMajor(s.getAdvisor().getDepartment().getName());
```

Enums (Enumerations)

J2SE 5.0 and above

```
public enum EnumName {
    symbolicName1(value1),
    symbolicName2(value2),
    //<...>
    symbolicNameN(valueN);

    private final type value;

    EnumName(type v) {
        value = v;
    }

    public type value() {
        return value;
    }
}
```

Comma-separated list of name/value pairs.

A single attribute.

A (nonpublic) constructor

An accessor method

Enums (Enumerations)

J2SE 5.0 and above

```
public enum Major {
    Math("Mathematics"),
    Bio("Biology"),
    CS("Computer Science"),
    Chem("Chemistry");

    private final String value;

    EnumName(String v) {
        value = v;
    }

    public String value() {
        return value;
    }
}
```

Comma-separated list of name/value pairs.

A single attribute.

A (nonpublic) constructor

An accessor method

Enums (Enumerations)

J2SE 5.0 and above

```
public class Student {
    private String name;
    private Major major;
    // ...

    public Student(String name, Major major) {
        this.setName(name);
        this.setMajor(major);
    }

    public void setName(String n) {
        name = n;
    }
    public void setMajor(Major m) {
        major = m;
    }
}
```

It's not a String

Accessor methods

Enums (Enumerations)

J2SE 5.0 and above

```
Major m = Major.CS;
System.out.println(m);
System.out.println(m.value());

Major n = "Computer Engineering";

Major n = Major.Computer;
```

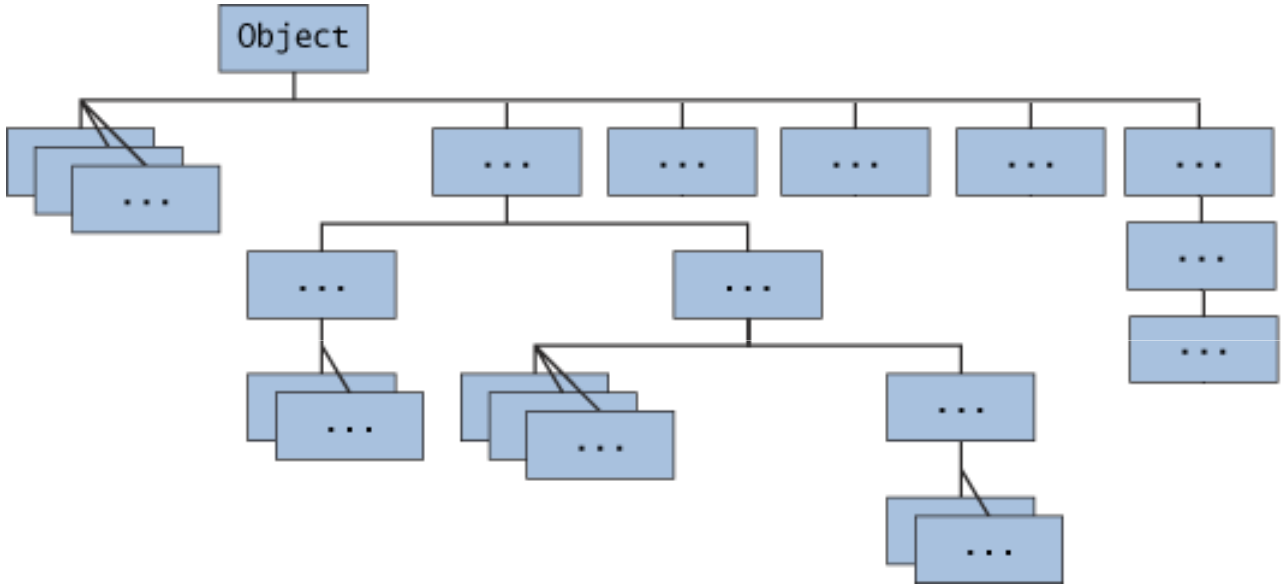
Access the symbol
Access the value

compiler ERROR!!! 

compiler ERROR!!! 

- Prior to J2SE 5.0
  - What was the option for enumerate possible values?
  - What are the characteristics of this values?

Features of the Object Class



For more info:

http://java.sun.com/javase/6/docs/api/java/lang/Object.html

Features of the Object Class

```
Professor pr = new Professor();
```

- Class getClass() --- String getName()

```
pr.getClass();
pr.getClass().getName();
```

We can Test to see if x is referring to a Professor object.

```
if (x != null && x.getClass().getName().equals("Professor")) {...}
```

Another way to do this – referenceVariable instanceof ClassName

```
if (x instanceof Professor) {…}
```

Features of the Object Class

Overriding the toString Method

Recall that the (overloaded) **print** and **println** methods do their best to render whatever expression is passed in as an argument into an equivalent String representation.

```
int x = 7;
double y = 3.8;
boolean z = false;
System.out.println(x);
System.out.println(y);
System.out.println(z);
```

and the code for expressions that **resolve** to one of these types:

```
System.out.println(x + y);
System.out.println(x == y);
```

Features of the Object Class

Overriding the toString Method

If we were to try to print the value of an expression that resolves to an **object reference.**

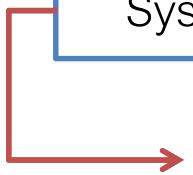
```
Student s = new Student("Harvey", "123-45-6789");
System.out.println(s);
```

Features of the Object Class

Overriding the toString Method

If we were to try to print the value of an expression that resolves to an **object reference.**

```
Student s = new Student("Harvey", "123-45-6789");
System.out.println(s);
```



This prints something similar to Student@71f71130, which represents an internal objectID relevant only to the JVM

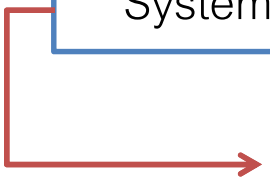
The String representation of an object is obtained with the toString() method.

```
System.out.println(s.toString());
```

Features of the Object Class

Overriding the toString Method

```
Student s = new Student("Harvey", "123-45-6789");
System.out.println(s);
```

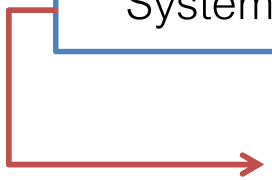


So, what do we need to print the Student object's attributes as a representation of the object? this is, by using the println() method?

Features of the Object Class

Overriding the toString Method

```
Student s = new Student("Harvey", "123-45-6789");
System.out.println(s);
```



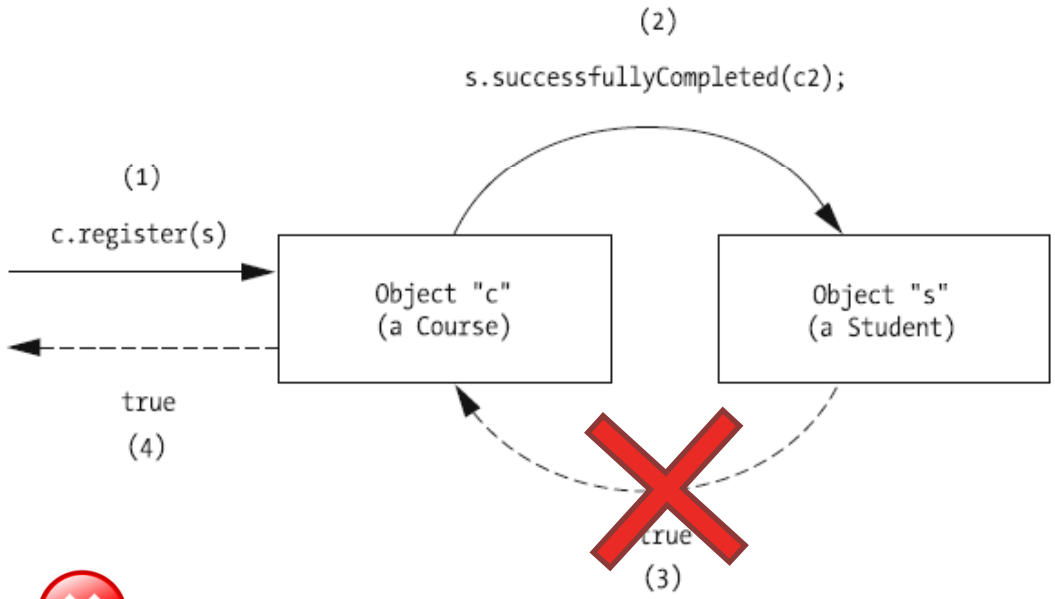
So, what do we need to print the Student object's attributes as a representation of the object? this is, by using the println() method?

- Let's override the toString() method, for instance:

```
public class Student {
    private String name;

    //accessor methods
    public String toString() {
        return this.getName() + " (" + this.getSsn() + ")");
    }
}
```

Java Exception Handling

- The problem may be something as simple as a logic error that the compiler wasn't able to detect.



runtime ERROR!!!

Java Exception Handling

Unexpected problems can arise as the JVM interprets/executes a Java program, for example:

- A program may be **unable to open a data file** due to inappropriate permissions.
- A program may have **trouble establishing a connection to a database management system** (DBMS) because a user has supplied an invalid password.
- A user may **supply inappropriate data via** an application's user interface—for example, a non-numeric value where a numeric value is expected.

Java Exception Handling

```
public class Problem {
    public static void main(String[] args) {
        Student s1 = null;
        Student s2 = null;
        // ...

        s1 = new Student();
        // ...

        s1.setName("Fred");

        s2.setName("Mary");
    }
}
```

Later on, we instantiate an object for s1 to refer to, but forget to do so for s2.

Still later in our program, we attempt to assign names to both Students.

Java Exception Handling

```
public class Problem {
    public static void main(String[] args) {
        Student s1 = null;
        Student s2 = null;
        // ...

        s1 = new Student();
        // ...

        s1.setName("Fred");

        s2.setName("Mary");
    }
}
```

Later on, we instantiate an object for s1 to refer to, but forget to do so for s2.

Still later in our program, we attempt to assign names to both Students.

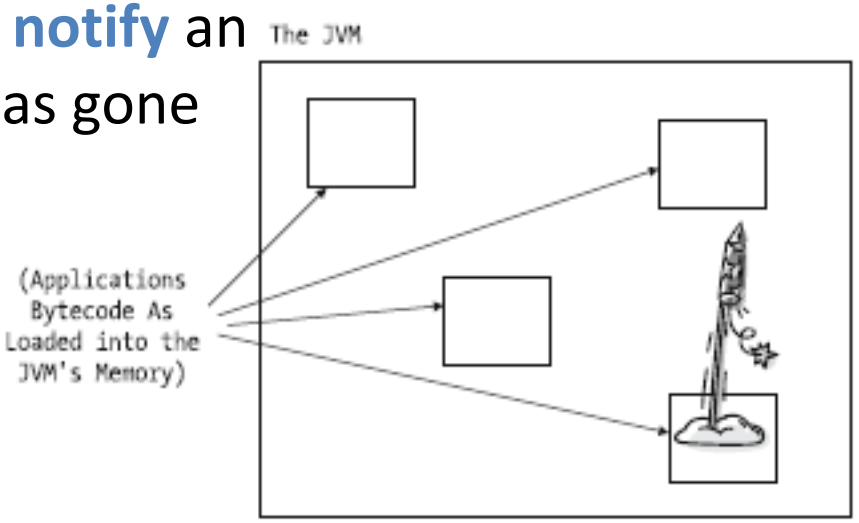
run-time ERROR!!!

Exception in thread "main" java.lang.NullPointerException

at Problem.main(Problem.java:22)

Java Exception Handling

- An **Exception** is a (**recoverable**) **Java run-time error.**
- The process whereby the **JVM reports that a run-time error has arisen** is referred as **throwing an exception.**
- When the JVM throws an exception, **it's as if the JVM is shooting off a signal flare to notify** an application that something has gone wrong.



Java Exception Handling

The Mechanics of Exception Handling

- The try block

  We enclose code that is likely to throw an exception. This indicates our intention to **catch** (i.e. to detect and respond to) any exceptions that might be thrown by the JVM.

- The catch block

  Presents the specific type of exception that is to be caught. We enclose the code to be used in recovering from the exception.

- The finally block

  The code within a finally block is **guaranteed** to execute no matter what happens in the try/catch code that precedes it

Java Exception Handling

The Mechanics of Exception Handling

```
try {
    code liable to throw exception(s) goes here ...
}
catch (ExceptionType1 variableName1) {
    recovery code for ExceptionType1 goes here ...
}
catch (ExceptionType2 variableName2) {
    recovery code for ExceptionType2 goes here ...
}
//…
finally {
    This code will be executed no matter what ...
}
```

The finally block is optional

The catch blocks are unnecessary if a finally block is present

Java Exception Handling

The Mechanics of Exception Handling

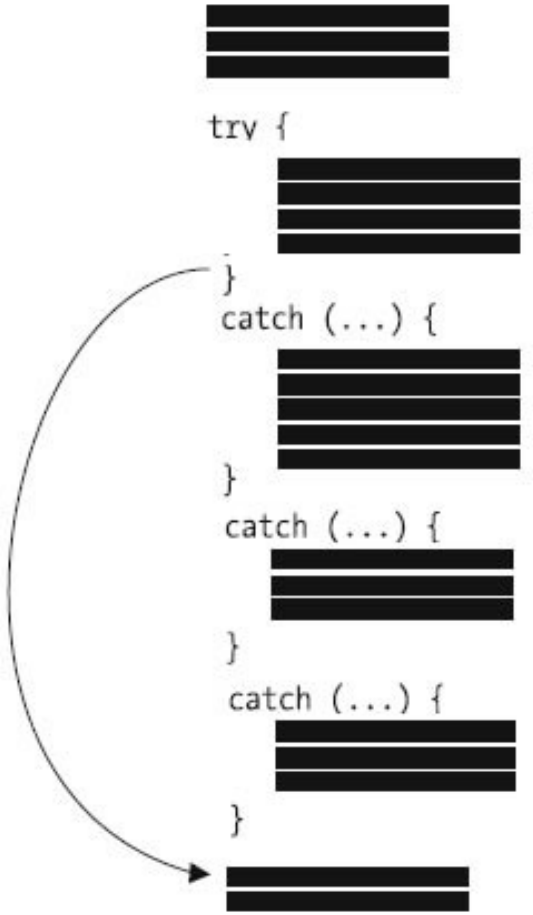
```
public class Problem {
    public static void main(String[] args) {
        Student s1 = null;
        Student s2 = null;

        s1 = new Student();
        // ...
        try {
            s1.setName("Fred");
            s2.setName("Mary");
        }
        catch (ArithmeticException e) {…}
        catch (NullPointerException e2) {
            System.out.println("Darn! We forgot to initialize all of the students!");
        }
        catch (ArrayIndexOutOfBoundsException e3) {…}
    }
}
```

Recovery code for an ArithmeticException goes here …

Recovery code for an ArrayIndexOutOfBoundsException goes here …

Java Exception Handling

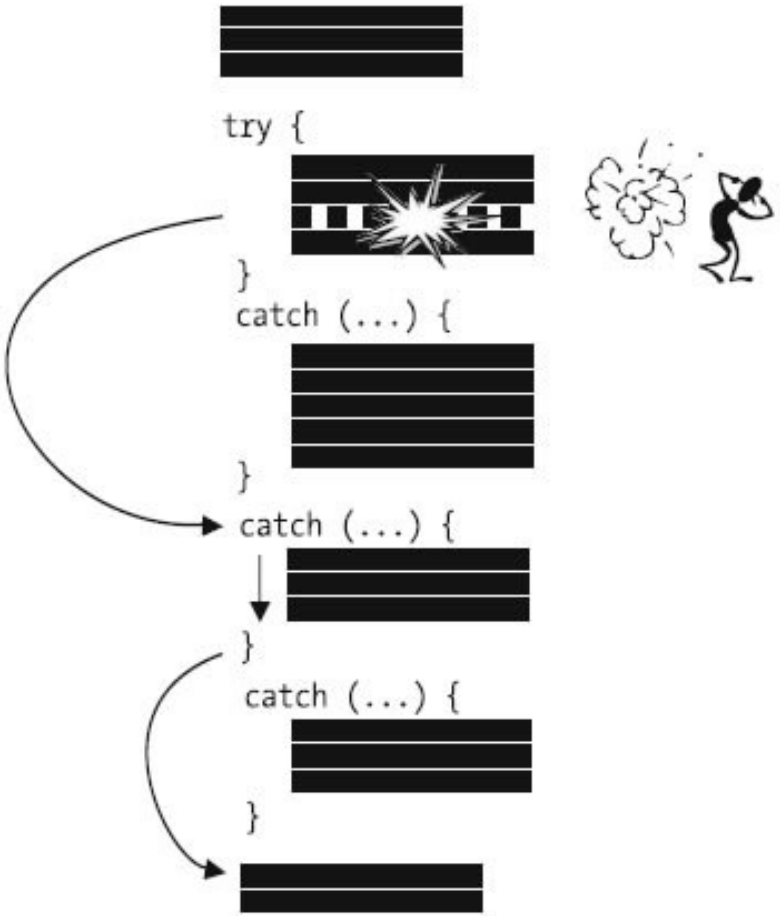
The Mechanics of Exception Handling



If no exceptions are thrown in the try block, all catch blocks are bypassed.

Java Exception Handling

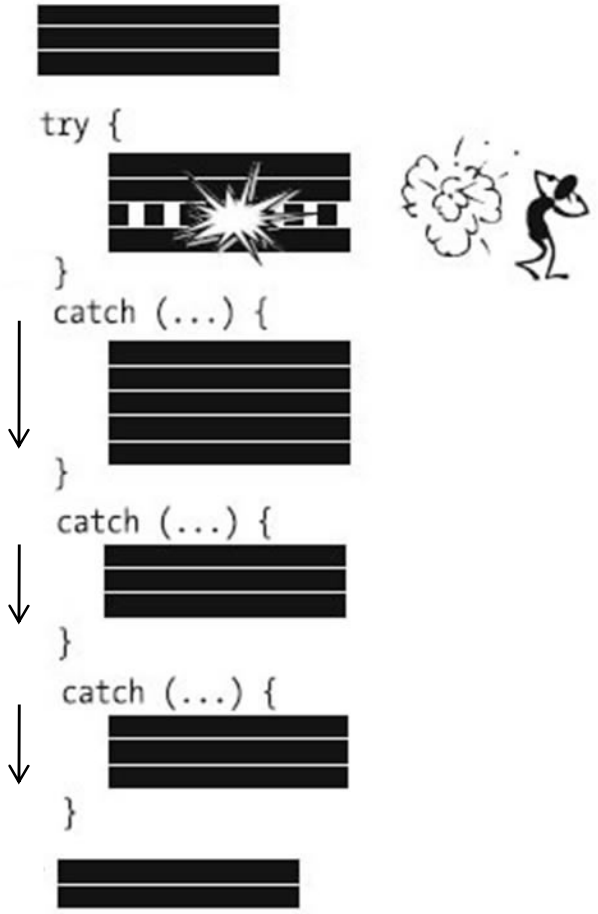
The Mechanics of Exception Handling



If an exception arises, the first matching catch block, if any, is executed, and the rest are skipped.

Java Exception Handling

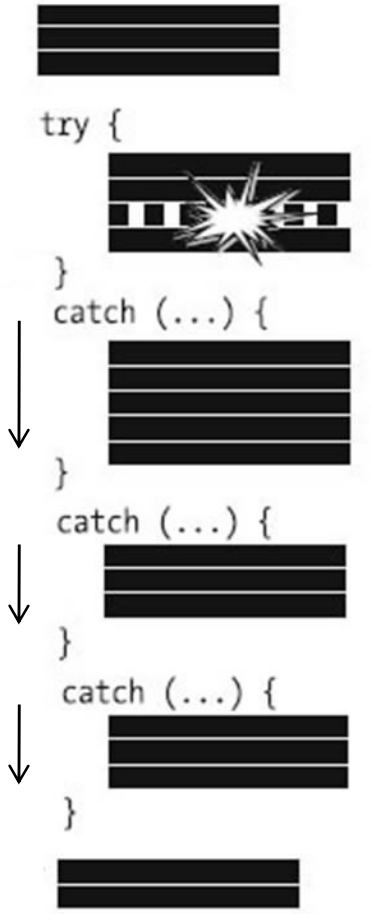
The Mechanics of Exception Handling



If no matching catch clause is found, the exception is said to be uncaught

Java Exception Handling

The Mechanics of Exception Handling



run-time ERROR!!!

If no matching catch clause is found, the exception is said to be uncaught

Java Exception Handling

The Mechanics of Exception Handling

```
public class Problem {
    public static void main(String[] args) {
        Student s1 = null;
        Student s2 = null;

        s1 = new Student();
        // ...
        System.out.println("We're about to enter the try block ...");
        try {
            System.out.println("We're about to call s1.setName(...)");
            s1.setName("Fred");
            System.out.println("We're about to call s2.setName(...)");
            s2.setName("Mary");
            System.out.println("We've reached the end of the try block ...");
        }
        //…
```

Java Exception Handling

The Mechanics of Exception Handling

```
        //…Here are our catch blocks (three in total).
        catch (ArithmeticException e) {
            System.out.println("Executing the first catch block ...");
        }
        catch (NullPointerException e2) {
            System.out.println("Executing the second catch block ...");
        }
        catch (ArrayIndexOutOfBoundsException e3) {
            System.out.println("Executing the third catch block ...");
        }



        System.out.println("We're past the last catch block ...");
    }
}
```

Java Exception Handling

The Mechanics of Exception Handling

```
        //…Here are our catch blocks (three in total).
        catch (ArithmeticException e) {
            System.out.println("Executing the first catch block ...");
        }
        catch (NullPointerException e2) {
            System.out.println("Executing the second catch block ...");
        }
        catch (ArrayIndexOutOfBoundsException e3) {
            System.out.println("Executing the third catch block ...");
        }
        finally {
            System.out.println("Executing the finally block ...");
        }
        System.out.println("We're past the last catch block ...");
    }
}
```

Let's try the finally block

Java Exception Handling

The Mechanics of Exception Handling

- The finally block [is optional as you have seen]

  The code within a finally block is **guaranteed** to execute no matter what happens in the try/catch code that precedes it—that is, whether any of the following happen:

  - The try block executes to completion without throwing any exceptions whatsoever.
  - The try block throws an exception that is handled by one of the catch blocks.
  - The try block throws an exception that is **not handled** by any of the catch blocks.

Java Exception Handling

The Mechanics of Exception Handling

- Two more changes in our example:
  - Let's now **repair the code** in our try block so that it doesn't throw any exceptions, to see what output our program will produce.
  - Once again **let the program throws a NullPointerException**, but **we are not going to catch it**.

Java Exception Handling

Catching Exceptions

Consider the following three-class example:

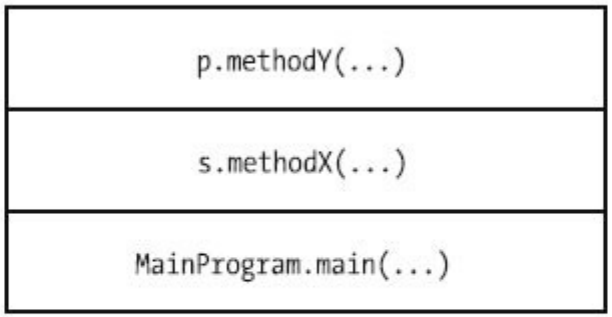
```
public class MainProgram {
    public static void main(String[] args) {
        Student s = new Student();
            s.methodX();
    }
}
```

```
public class Student {
    public void methodX() {
        Professor p = new Professor();
        p.methodY();
    }
}
```

```
public class Professor {
    public void methody() {
        //…
    }
}
```

Java Exception Handling

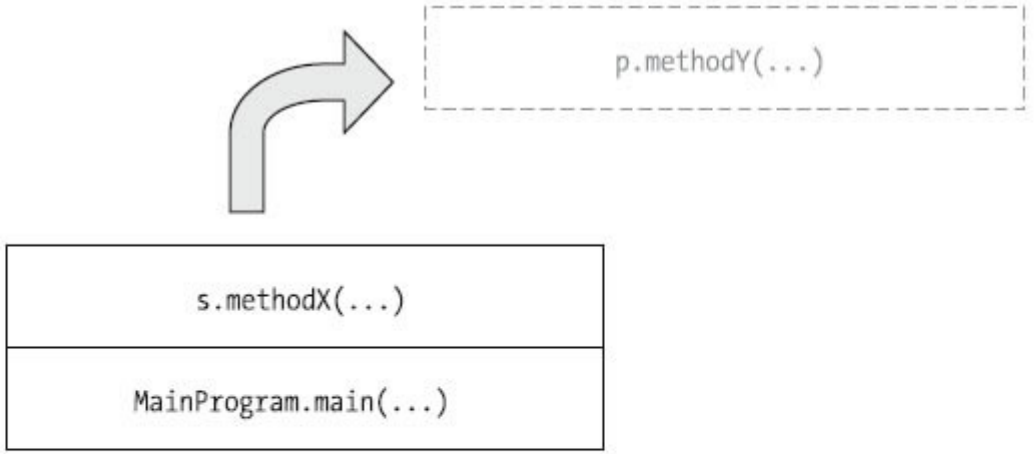
Catching Exceptions



- When the JVM executes our MainProgram, its main method is invoked, which in turn invokes s.methodX(), which in turn invokes p.methodY();—this produces what is known as a **call stack** at run time.

Java Exception Handling

Catching Exceptions

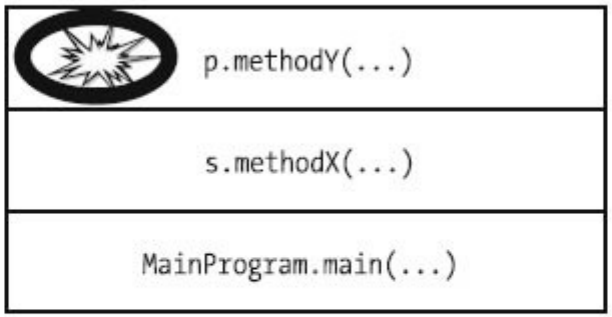


- A **stack** is a **last-in, first-out (LIFO)** data structure; the most recent method call is pushed onto the top of the call stack, and when that method exits, it is removed from (popped off of) the call stack.

Java Exception Handling

Catching Exceptions

A NullPointerException is thrown while executing methodY. If the appropriate try/catch logic is incorporated within the body of methodY then neither the Student nor MainProgram classes will be aware that such an exception was ever thrown.

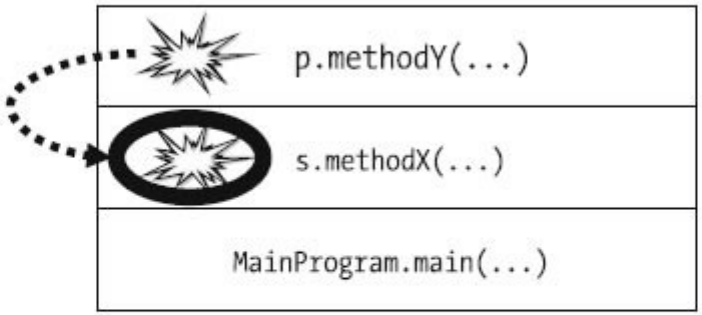



```
public class Professor {
    public void methody() {
        try {…}
        catch (NullPointerException e) {…}
    }
}
```

Java Exception Handling

Catching Exceptions

methodY does not catch/handle NullPointerException.

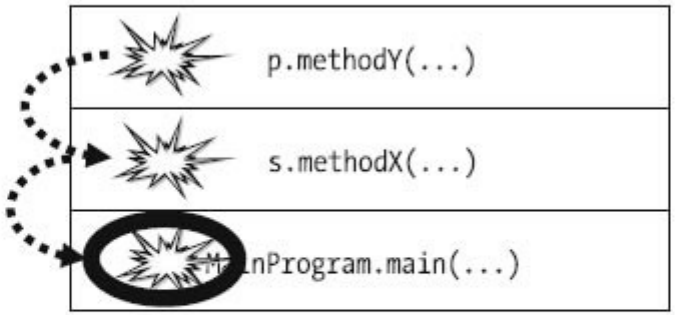



```
public class Professor {
    public void methody() {
        // A NullPointerException is thrown here, but
        // is NOT caught/handled.
    }
}
```

Java Exception Handling

Catching Exceptions

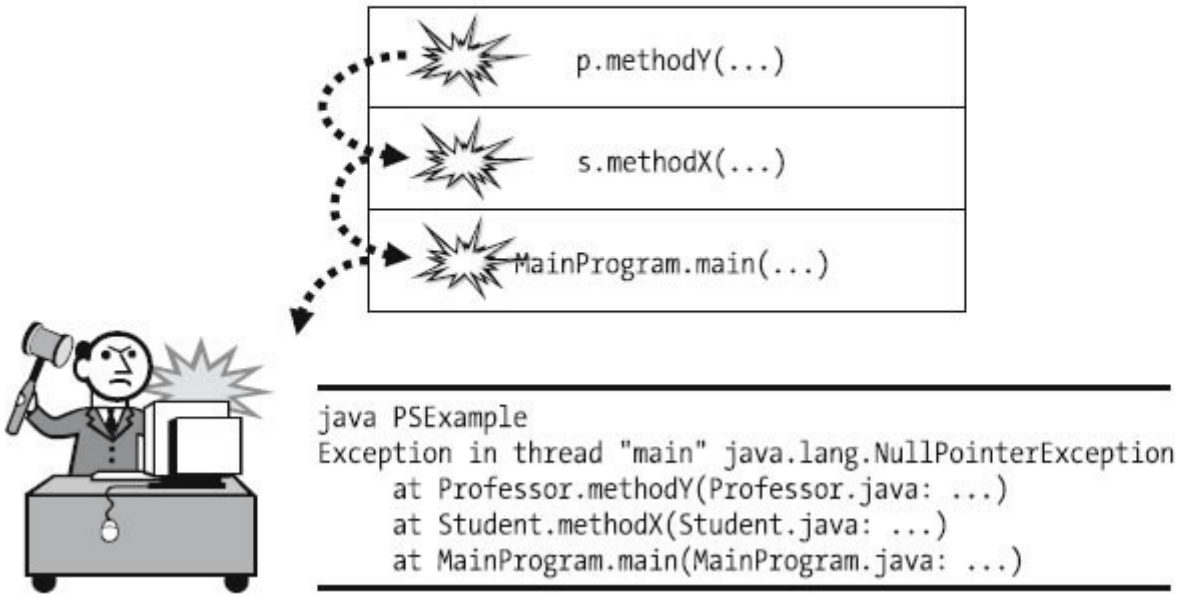
The NullPointerException makes its way to the main method.



Finally, what happened if the main method doesn't handle the exception?

Java Exception Handling

Catching Exceptions




```
java PSExample
Exception in thread "main" java.lang.NullPointerException
     at Professor.methodY(Professor.java: ...)
     at Student.methodX(Student.java: ...)
     at MainProgram.main(MainProgram.java: ...)
```

run-time ERROR!!!

Java Exception Handling

Interpreting Exception Stack Traces

```
java PSExample
Exception in thread "main" java.lang.NullPointerException
    at Student.print(Student.java:10)
    at Professor.printAdviseeInfo(Professor.java:11)
    at PSExample.main(PSExample.java:12)
```

Reading the stack trace from top to bottom:

- The actual NullPointerException arose on line 10 of the Student class:
- That line of code is within the body of the print method of the Student class, which was invoked from line 11 of the Professor class:
- And, that line of code is within the body of the printAdviseeInfo method of the Professor class, which was in turn invoked from the PSExample class's main method on line 12:

Java Exception Handling

The Exception Class Hierarchy

Overview Package Class Use Tree Deprecated Index Help

Java™ Platform Standard Ed. 6

PREV CLASS NEXT CLASS | FRAMES NO FRAMES All Classes

SUMMARY: NESTED | FIELD | CONSTR | METHOD | DETAIL: FIELD | CONSTR | METHOD

java.lang

Class Exception

```
java.lang.Object
  └ java.lang.Throwable
      └ java.lang.Exception
```

All Implemented Interfaces:

Serializable

Direct Known Subclasses:

AclNotFoundException, ActivationException, AlreadyBoundException, ApplicationException, AWTException, BackingStoreException, BadAttributeValueExpException, BadBinaryOpValueExpException, BadLocationException, BadStringOperationException, BrokenBarrierException, CertificateException, ClassNotFoundException, CloneNotSupportedException, DataFormatException, DatatypeConfigurationException, DestroyFailedException, ExecutionException, ExpandVetoException, FontFormatException, GeneralSecurityException, GSSException, IllegalAccessException, IllegalClassFormatException, InstantiationException, InterruptedException, IntrospectionException, InvalidApplicationException, InvalidMidiDataException, InvalidPreferencesFormatException, InvalidTargetObjectTypeException, InvocationTargetException, IOException, JAXBException, JMException, KeySelectorException, LastOwnerException, LineUnavailableException, MarshalException, MidiUnavailableException, MimeTypeParseException, MimeTypeParseException, NamingException, NoninvertibleTransformException, NoSuchFieldException, NoSuchMethodException, NotBoundException, NotOwnerException, ParseException, ParserConfigurationException, PrinterException, PrintException, PrivilegedActionException, PropertyVetoException, RefreshFailedException, RemarshalException, RuntimeException, SAXException, ScriptException, ServerNotActiveException, SOAPException, SQLException, TimeoutException, TooManyListenersException, TransformerException, TransformException, UnmodifiableClassException, UnsupportedAudioFileException, UnsupportedCallbackException, UnsupportedFlavorException, UnsupportedLookAndFeelException, URIReferenceException, URISyntaxException, UserException, XAException, XMLParseException, XMLSignatureException, XMLStreamException, XPathException

---

```
public class Exception
extends Throwable
```

The class `Exception` and its subclasses are a form of `Throwable` that indicates conditions that a reasonable application might want to catch.

Since:

JDK1.0

See Also:

`Error`, Serialized Form

Java Exception Handling

The Exception Class Hierarchy

Overview Package **Class** Use Tree Deprecated Index Help

PREV CLASS NEXT CLASS | FRAMES NO FRAMES All Classes

SUMMARY: NESTED | FIELD | CONSTR | METHOD | DETAIL: FIELD | CONSTR | METHOD

Java™ Platform Standard Ed. 6

java.sql

Class DataTruncation

```
java.lang.Object
  └ java.lang.Throwable
      └ java.lang.Exception
          └ java.sql.SQLException
              └ java.sql.SQLWarning
                  └ java.sql.DataTruncation
```

All Implemented Interfaces:

Serializable, Iterable<Throwable>

```
public class DataTruncation
extends SQLWarning
```

A catch clause for a given exception type X will catch that specific type of exception or any of its subtypes, by virtue of the "is a" nature of inheritance.

List catch clauses in most specific to least specific order after a try block.

Do you know why?

Java Exception Handling

The Mechanics of Exception Handling

```
try {
    // attempt to write data to a database
}
catch (DataTruncation e1) {
    // Catch the most specific exception type first;
}
catch (SQLWarning e2) {
    // ... then, the next most specific ...
}
catch (SQLException e3) {
    // ... working our way up to the most general.
}
```

Java Exception Handling

The Mechanics of Exception Handling

```
try {
    // attempt to write data to a database
}
catch (DataTruncation e1) {
    // Catch the most specific exception type first;
}
catch (SQLWarning e2) {
    // ... then, the next most specific ...
}
catch (SQLException e3) {
    // ... working our way up to the most general.
}
```

```
try { // database access operation… }
catch (SQLException e) {…}
```

This will catch DataTruncation exceptions, so, why bother?

Java Exception Handling

The Mechanics of Exception Handling

```
try {
    // attempt to write data to a database
}
catch (DataTruncation e1) {
    // respond SPECIFICALLY to data truncation issues ...
}
catch (SQLWarning e2) {…}
catch (SQLException e3) {…}
```

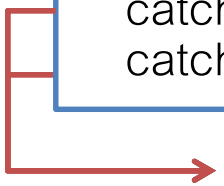
```
try {
    // attempt to write data to a database
}
catch (SQLException e3) {…}
catch (SQLWarning e2) {…}
catch (DataTruncation e1) {
    // respond SPECIFICALLY to data truncation issues ...
}
```

Java Exception Handling

The Mechanics of Exception Handling

```
try {
    // attempt to write data to a database
}
catch (DataTruncation e1) {
    // respond SPECIFICALLY to data truncation issues ...
}
catch (SQLWarning e2) {…}
catch (SQLException e3) {…}
```

```
try {
    // attempt to write data to a database
}
catch (SQLException e3) {…}
catch (SQLWarning e2) {…}
catch (DataTruncation e1) {…}
```



These catch clauses are wasted – they can never be reached

Java Exception Handling

Catching the Generic Exception Type

- Some programmers use the "lazy" approach of catching the most generic Exception type and then doing **nothing to recover**, just to silence the compiler.

```
try {
    // anything…
}
catch (Exception e) {}
```

Empty braces → do NOTHING to recover!

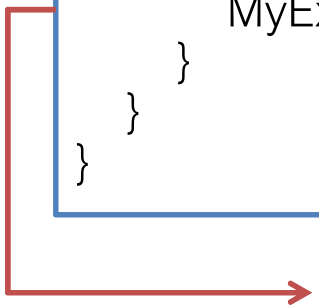
- **This is not a good practice!** We're masking the fact that an exception has occurred.
  Our program may be in a serious state of dysfunction, perhaps coming to a screeching halt (!), but it will remain silent.

Java Exception Handling

Catching the Generic Exception Type

- One legitimate case in which we might wish to do so is if we are writing a special-purpose error-handling subsystem for an application, as suggested by the following pseudocode:

```
public class Example {
    public static void main(String[] args) {
        try {
            // all of our main application logic is here ...
        }
        catch (Exception e) {
            MyExceptionHandler.handleException(e);
        }
    }
}
```



What can you say about the MyExceptionHandler.handleException() method?

Java Exception Handling

Compiler Enforcement of Exception Handling

- Generally speaking, the Java compiler will force us to enclose code that is liable to throw an exception in a try block with an appropriate catch block(s).

> Unreported exception java.io.FileNotFoundException; must be caught or declared to be thrown

- In such a situation, we have two choices:
  - Ideally, we'd enclose the code in question in a try block with an appropriate catch block (or blocks):
  - Alternatively, we may add a throws clause to the header of the method in which the uncaught exception might arise.

Java Exception Handling

Compiler Enforcement of Exception Handling

```
public class FileIOExample {
    public static void main(String[] args) {
        try {
            // open the file of interest;
            // while (end of file not yet reached)
                // read next line from file;
        }
        catch (FileNotFoundException e) { ...}
    }
}
```

```
public class FileIOExample {
    public static void main(String[] args) throws FileNotFoundException {
        // open the file of interest;
        // while (end of file not yet reached)
            // read next line from file;
    }
}
```

Java Exception Handling

Compiler Enforcement of Exception Handling

- The only exception types that the compiler doesn't mandate catching are those derived from the RuntimeException class.
  - NullPointerException
  - ArithmeticException
  - ClassCastException
  - IndexOutOfBoundsException

Java Exception Handling

Taking Advantage of the Exception That We've "Caught"

```
try {
    // try to open a nonexistent file named Foo.dat ...
}
catch (FileNotFoundException e) {
    System.out.println("Error opening file " + e.getMessage());
}
```

```
try {
    // try to open a nonexistent file named Foo.dat ...
}
catch (FileNotFoundException e) {
    e.printStackTrace();
}
```

Java Exception Handling

Nesting of try/catch Blocks

- A try statement may be nested inside either the try or catch block of another try statement.

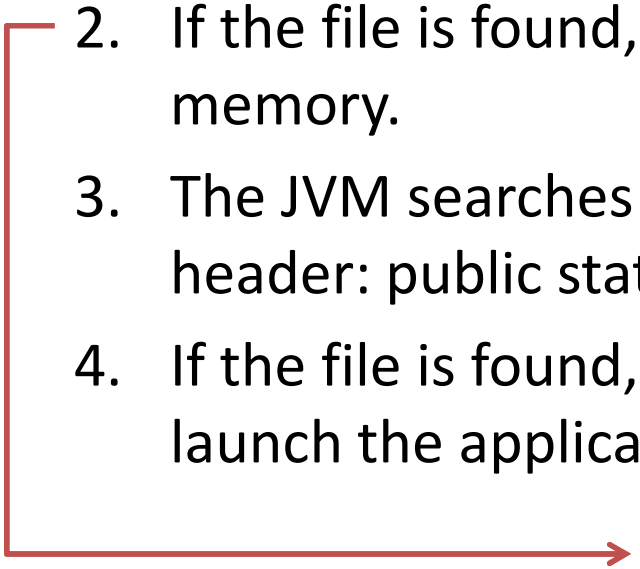
```
try {
    // open a user-specified file
}
catch (FileNotFoundException e) {
    try {
        // open a DEFAULT file instead ...
    }
    catch (FileNotFoundException e2) {
        // attempt to recover ...
    }
}
```

We couldn't find the user-specified file, perhaps we might want to open a DEFAULT file... but what if the DEFAULT file cannot be found, either?

Behind the Scenes of the JVM

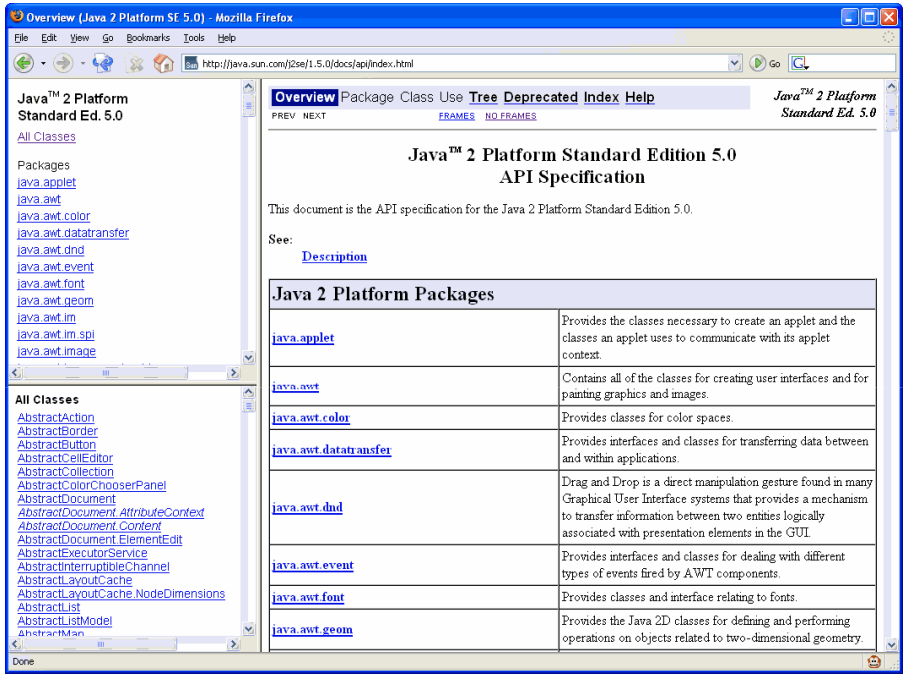
- When you run a Java program, you're actually launching the JVM, which in turn goes through the following process:

1. The JVM searches for the specified bytecode file in its classpath.
2. If the file is found, the JVM loads the bytecode into its memory.
3. The JVM searches the bytecode for the official main method header: public static void main(String[] args).
4. If the file is found, the JVM executes the main method to launch the application.



The JVM's Class Loader

Javadoc Comments



@see

http://java.sun.com/j2se/javadoc/writingdoccomments/index.html

In Eclipse: **Project>Generate Javadoc…**

Java Archive (JAR) Files

- The Java bytecode comprising an application is commonly bundled and delivered in the form of a **Java Archive (JAR) file.**

- In Eclipse:
  Right click on the Project > Export > JAR File

  Right click on the Project > Build Path > Add External Archives...

More info

http://java.sun.com/developer/Books/javaprogramming/JAR/basics/

UML

1. Introduction
2. Class Diagrams – Classes, interfaces, and collaborations
3. Package Diagrams
4. Use Cases – Organizes the behaviors of the system
5. Sequence Diagrams – Focuses on the time ordering of messages
6. UML Tools {Eclipse, NetBeans, ArgoUML, Umbrello,…}

UML

S.W. Ambler, The Elements of UML(TM) 2.0 Style, Cambridge University Press, 2005.

M. Fowler, UML Distilled: A Brief Guide to the Standard Object Modeling Language (3rd Edition), Addison-Wesley Professional, 2003.

G. Booch, J. Rumbaugh, and I. Jacobson, Unified Modeling Language User Guide, The (2nd Edition), Addison-Wesley Professional, 2005.

References

- J. Barker, *Beginning Java Objects: From Concepts To Code, Second Edition*, Apress, 2005.
- Java™ Platform, Standard Edition 6 – The Collection Framework. Available online at: http://java.sun.com/javase/6/docs/technotes/guides/collections/index.html