CORE JAVA

By

Teknoturf Info Services Pvt. Ltd.

Chapter 1
Introduction to Java

Agenda – Day 1

- (00:30) Introduction to Java
- (00:30) Features of Java
- (00:30) Data types, Operators, Keywords, reserved words
- (00:30) Flow Control Mechanisms
- (00:15) Break
- (00:30) Classes & Objects
- (01:00) Strings
- (01:00) Lunch Break
- (01:00) Arrays
- (00:30) Command Line Arguments – Wrapper Classes
- (00:30) Constructors

Agenda – Day 2

- (00:45) Packages
- (01:00) Inheritance
- (00:15) Break
- (00:40) Abstract Classes
- (00:40) Interfaces
- (01:00) Lunch Break
- (01:00) Exception Handling & User defined exceptions
- (01:00) Files & Streams
- (00:15) Break

Agenda – Day 3

- (00:30) Collection API
- (00:45) Contd., Collection API
- (01:30) Reflection API
- (00:15) Break
- (01:30) JDBC
- (00:45) Introduction to MVC Pattern & Implementation

Agenda – Day 4

- Building a Salary Statement generator for Temporary and Permanent Employees of an organization. The Application includes inheritance concepts, Interfaces and packages.
- Creating an address software where the java program will communicate with data stored in tables in the Oracle Database using the concepts of JDBC
- Understanding the various Exceptions available in java (Team Work)
- Implementing the various classes analyzed for the banking application in a framework. (Team Work)

About Java...

- A modern object-oriented language
- Platform independence
- Lots of powerful built-in features, e.g. threads, networking, etc.
- Java can run:
  - in a browser as an applet
  - On the desktop as an application
  - In a server to provide, e.g., database access,
  - Embedded in a device

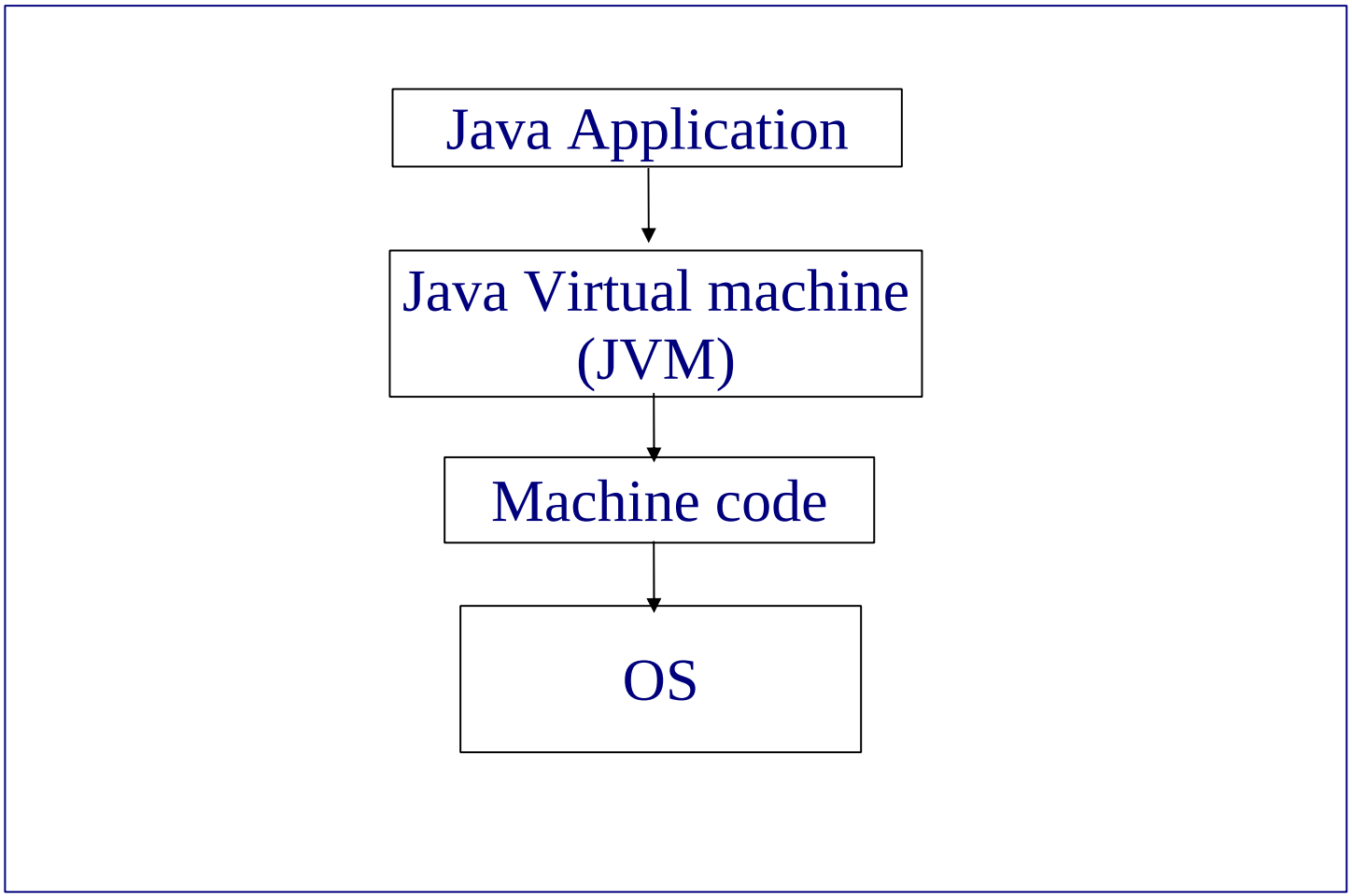
Features

- Object-oriented
- Compiled code runs in a Virtual Machine
- Platform independent
- Many security safeguards
- Fast development due to features like
  - Memory management (automatic garbage collection)
  - Array limit checking
  - No direct access to memory addresses
  - Easy to create threads for multi-processing

Java Editions

- Java Standard Edition
  - Basic tools and libraries to build applications and applets
- Java Enterprise Edition
  - Extra libraries for creating servlets, Java server pages, database interfaces, etc.
- Java Micro Edition
  - Fewer libraries, e.g. simplified graphics, and smaller VM for embedded apps such as PDAs and cellular phones.

Programming Environment



Execution of a java program

ProgrammingCode → Compiler → Bytecode instructions

Bytecodes → Virtual Machine executes instructions

Since the program runs inside the VM, it will run on any machine on which a VM has been created.

Execution of Java Program

- ***.class*** **file is not machine code. It is intermediate form called Java Byte code. Can run on any platform as long as platform has a Java Virtual Machine (JVM).**
- **The second step on previous slide invokes the JVM to interpret the byte code on the given platform.**
- **In theory, byte code can be moved to another platform and be run there without recompiling – this is the magic of applets.**
- **Leave off the *.class* part when invoking the JVM.**

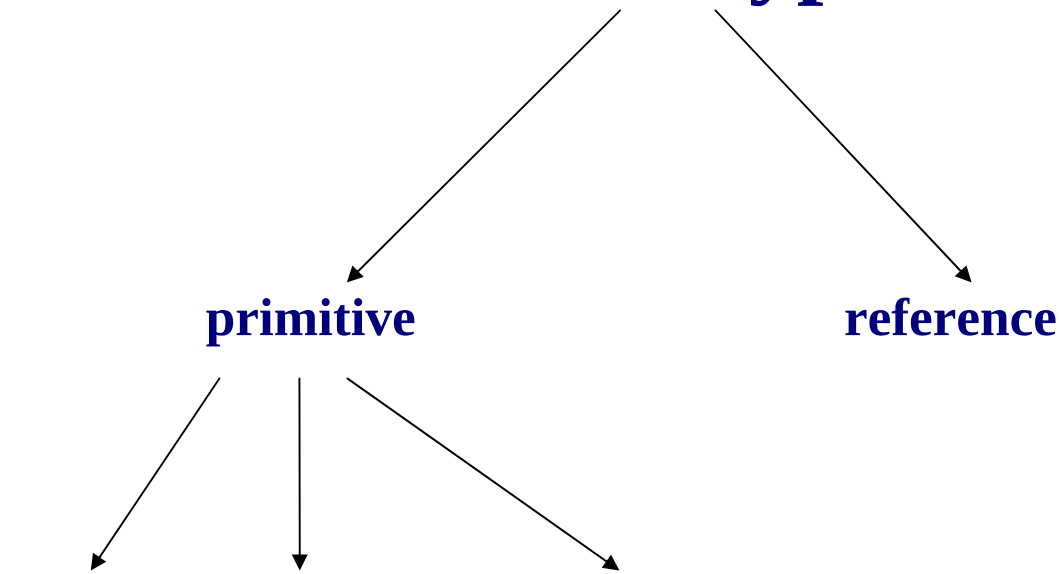
Keywords in Java 1.4

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

Reserved words

- goto
- const

Data types


primitive
reference
Integral
Floating point
Others
byte
short
int
long
float
double
char
boolean

Primitive data types

- **int** 4 bytes
- **short** 2 bytes
- **long** 8 bytes
- **byte** 1 byte
- **float** 4 bytes
- **double** 8 bytes
- **char** Unicode encoding (2 bytes)
- **boolean {true,false} (Size depends on JVM)**

Note:

Primitive type always begin with lower-case

Operators

- **Arithmetic Operators**

  +, -, *, /, %

- **Unary Arithmetic Operators**

  ++, --

- **Relational and Conditional Operators**

  >, >=, <, <=, ==, !=, &&, ||, !

- **Bitwise Operators**

  >>, <<, >>>, &, |, ^, ~

- **Ternary Operator: ()?():()**
- **Special Operator:** instanceof

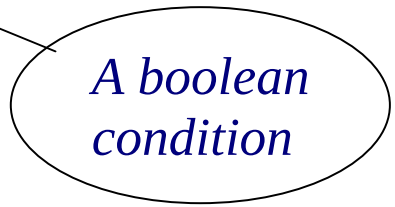
Flow control

- Decision making
  - if-else, switch-case
- Loop
  - for, while, do-while
- Exception
  - try-catch-finally, throw
- Miscellaneous
  - Break, continue, label:, return

if-else

Syntax:

```
if(condition)
{   //true part}
    else
{   //false part}
```



Nested 'if'

```
if(condition1)
{   if(condition2)
      {   //true part of both conditions
      }else
      {   //false part of condition2}
      else
      {   //false part of condition1}}
```

multiple 'if'

```
if(condition1)
{   //true part for condition1
}
    else if(condition2)
    {   //true part for condition2
        //also false part of condition1
        }
        else
        {
            //false part of condition2
            }
```

The switch-case

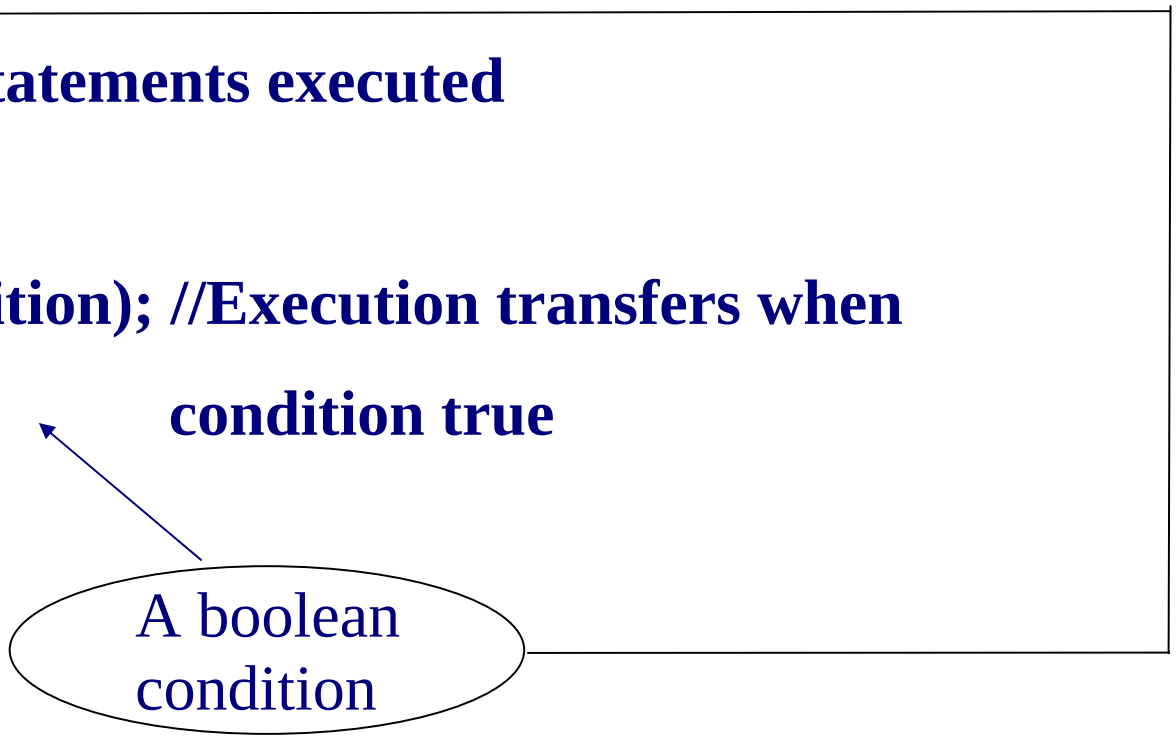
```
...
switch(int parameter)
{
case '1':
    ..// block executed when the int parameter value is '1'
    break;
case '3':
    ..// block executed when the int parameter value is '3'
    break;
default:
    ..// block executed when the int parameter value does not match any
       specific value
    break;
    }
```

The 'while' loop

```
while(condition)
{
//block of code executed when condition is 'true'
}
//execution reaches here when condition is false
```

'do-while' loop

```
do
{
//block of statements executed
}
while(condition); //Execution transfers when
                  condition true
```



'for' loop

...

for(initialization; condition; modifier)

{

block executed when *condition true*

}

....

Few Valid syntax:

```
for(;condition;)
{
.....
}
```

```
for(; ;)
{
......
}
```

```
for(initialization; condition;)
{
.....
}
```

Comments in Java

The javadoc program generates HTML API documentation from the "javadoc" style comments in your code.

```
/* This kind of comment can span multiple lines */

// This kind is to the end of the line

/**
  * This kind of comment is a special
  * 'javadoc' style comment
  */
```

Encapsulation in Java

- Class

- A class is a definition of a *type:*
  - Like a template, a class defines the characteristics and behaviors of the type.
- Creation of classes
  - Includes Variables, methods, classes

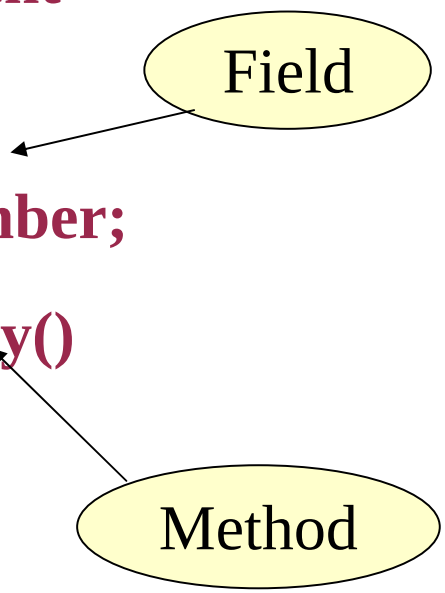
Objects

- Objects represent real-world things:
  - Ship
  - Wheel,etc
- Programatically, an object is an *instance* of a class:
  - Can be instantiated and manipulated
  - An object's characteristics are defined by the class that was used to create the object.
- Objects have
  - State
  - Behavior
  - Identity

Java syntax

- **Creating a class**

```
class Student
{
int rollnumber;
void display()
{...}
}
```



- **Creating an Object**

```
Student Tom=
new Student();
Tom.rollnumber=10092;
```

Fields

- Fields define the properties of a class.
  - Can be intrinsic types (`int`, `boolean`...)
  - Can be user-defined objects

State is the current value of a field in an object.

Methods

- Methods describe the capabilities of the class.
  - Every method must be called from another method. The only exception is `main()`, which is called by the OS.
  - Methods can accept parameters.

Compiling/Running first Java program

- Create source code file (call it for example MyFirstProgram.java).
- To compile:

  prompt >> javac MyFirstProgram.java
- This produces byte code file named MyFirstProgram.class
- To run:

  prompt >> java MyFirstProgram

Strings

- **Reference data type-**
- **`String` is a class in java.lang.String**
- **Important methods**
- **Creation of String**
  - **Programmatic syntax**

  Eg:

  String name=”Kiran”

  String name=new String(“Tarun”)

Strings

- Once a `String` object has been created, neither its value nor its length can be changed
- Thus we say that an object of the `String` class is *immutable*
- However, several methods of the `String` class return new `String` objects that are modified versions of the original

String indexes

- It is occasionally helpful to refer to a particular character within a string
- This can be done by specifying the character's numeric *index*
- The indexes begin at zero in each string
- In the string `"Hello"`, the character `'H'` is at index 0 and the `'o'` is at index 4

String operations

- **String concatenation**
  - **Use of '+' and API methods**

```
System.out.println(“Hai”+”friends”+123);
String s=”friend”;
s.concat(“hello”);
```

- **String copy**

Comparing Strings

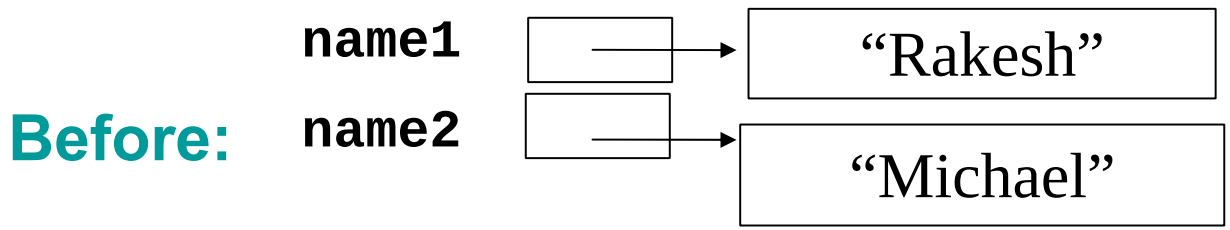
- Strings can be compared to see if they are equal:
  - `equals()` method is case sensitive
  - `equalsIgnoreCase()` method ignores case

String copy

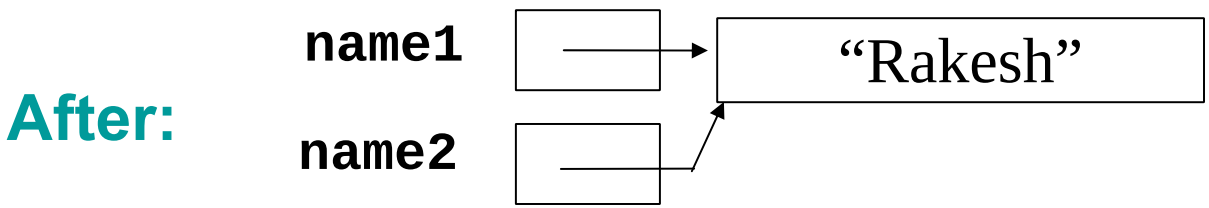
- For object references, assignment copies the address:

String name1=" Rakesh";

String name2= "Michael"



name2 = name1



StringBuffer class

- Java strings are immutable. Use `StringBuffer` object when you need to modify a string of characters.
  - Includes methods to:
    - *Set character at specific index:* `setCharAt()`
    - *Append characters:* `append()`
    - *Insert characters at a specific index:* `insert()`
    - *Reverse the characters:* `reverse()`

Garbage collection

- When an object no longer has any valid references to it, it can no longer be accessed by the program
- The object is useless, and therefore is called *garbage*
- Java performs *automatic garbage collection* periodically, returning an object's memory to the system for future use
- In other languages, the programmer is responsible for performing garbage collection

Arrays

- An array is an ordered collection that stores many elements of the same type within one variable name.
- Elements are the items in an array.
- Each element is identified with an index number.
- Every element is of identical data type
- Index numbers always start at 0 for the first element.

Arrays

- Reference data types
- 1-d,2-d,3-d,n-d arrays

```
int[] one=new int[3];

int[][] two =new int[2][2]

int[][][] three= new
   int[2][2][4]
```

Syntax

- **Declaration & initialization**

```
int[] arr={1,2,3,3}
int[] arr1=new
   int[]{2,3,4}
int[] arr=new int[10]
arr[0]=23;
arr[1]=45;
float[][]
   dim={{2,3},{4,6}}
```

Iterating a 1-D array

- Use the `length` property to find the number of elements in the array.

```
System.out.println(myArray.length);
```

  - *Displays the length of the array in the console*

```
for ( int i = 0; i < myArray.length;  i++ );
```

  - *Uses the `length` property to set up the `for` loop*

2-D Arrays

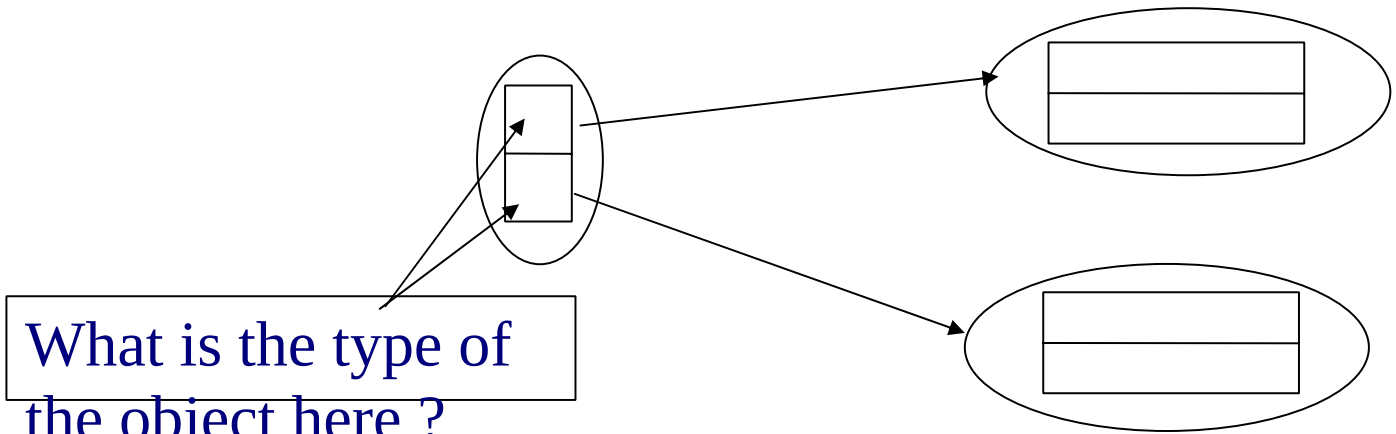
A two-dimensional array has “rows” and “columns,” and can be thought of as a series of one-dimensional arrays stacked on top of one another.

Declare a two-dimensional array:

```
int[][] anArray = new int[5][5];
```

Multi dimensional arrays

- In Java
- `Animal[][] arr= new Animal[2][2]`



Wrapper Classes

The java.lang package contains *wrapper classes* that correspond to each primitive type:

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

Wrapper Class Names

Except for int and char, the wrapper class name is exactly the same as the primitive type name EXCEPT that it starts with a capital letter.

double is a primitive	Double is a class.
boolean is a primitive	Boolean is a class.
long is a primitive	Long is a class.

The wrapper classes for int and char are different.

int is a primitive	Integer is a class.
char is a primitive	Character is a class.

Wrapper Classes

- An object of a wrapper class can be used in any situation where a primitive value will not suffice
- For example, some objects serve as containers of other objects. Primitive values could not be stored in such containers, but wrapper objects could be.
- Another example is that some methods take an Object as a parameter. So a primitive cannot be used, but an object of any class can be used.

Why Wrapper classes?

- It's easy to go back and forth between primitives and their wrappers and the wrappers add the needed functionality that is provided for objects.

- Why primitives?
  - *Some languages don't have them, but, in Java, primitives get assigned memory slots when they are declared and this memory stores the value, but when objects are created it is the reference that is manipulated*

Instantiating Wrapper Classes

- **All wrapper classes can be instantiated using the corresponding primitive type as an argument.**

```
Integer age = new Integer(40);
Double num = new Double(8.2);
Character ampersand = new Character('&');
Boolean isDone = new Boolean(false);
```

Instantiating Wrapper Classes

- **All wrapper classes EXCEPT Character and Void can be instantiated using a String argument.**

```
Integer age = new Integer("40");
Double num = new Double("8.2");
Boolean isDone = new Boolean("true");
```

- **The Void wrapper class cannot be instantiated.**

Wrapper Class Methods

- **Wrapper classes all contain various methods that help convert from the associated type to another type.**

```
Integer num = new Integer(4);
float flt = num.floatValue();
//stores 4.0 in flt

Double dbl = new Double(8.2);
int val = dbl.intValue(); //stores 8 in val
```

More Wrapper Class Methods

- **Some Wrapper classes also contain methods that will compare the value of two objects of that class.**

```
Integer num1 = new Integer(4);
Integer num2 = new Integer(11);
if(num1.compareTo(num2)< 0)
  //executes if num1 < num2
```

Wrapper Class Methods

- Wrapper classes also contain static methods that help manage the associated type
- Each numeric class contains a method to convert a representation in a `String` to the associated primitive:

```
int num1 = Integer.parseInt("3");
double num2 = Double.parseDouble("4.7");
long num3 = Long.parseLong("123456789");
```

Wrapper Class Constants

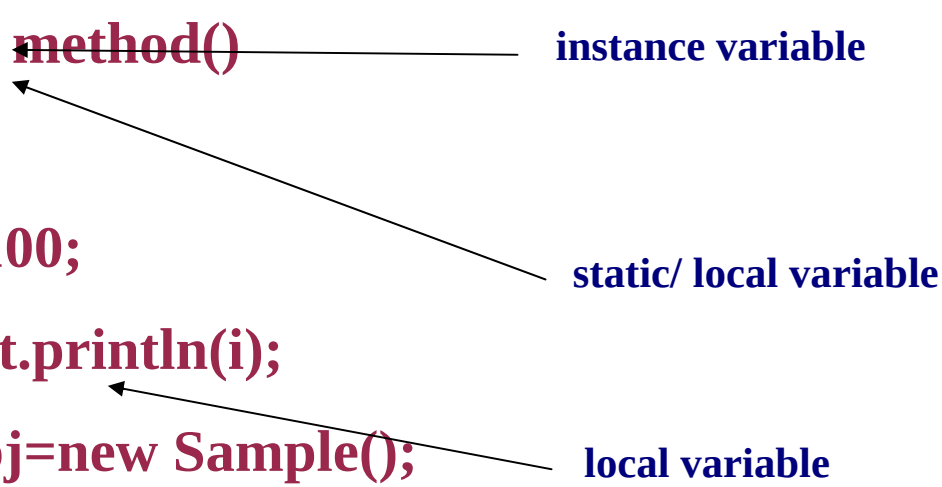
- The wrapper classes often contain useful constants as well
- The `Integer` class contains `MIN_VALUE` and `MAX_VALUE` which hold the smallest and largest `int` values
- Other numeric classes will also have `MIN_VALUE` and `MAX_VALUE` which hold the smallest and largest value for their corresponding types.

Variables in a class

- Static variables – class variables
  - storage
  - How to access
- Non-static variables
  - Instance variables
  - Local variables

Use of variables

```
class Sample
{   float j=45.5f;
    static int i=90;
    static void method()
    {
    int local=100;
    System.out.println(i);
    Sample obj=new Sample();
    System.out.printin(obj.j);
    System.out.println(local);
} }
```



Accessing static variables

```
class MyClass
{static int j=56;
public static void main(String ar[])
{
System.out.println(j);
}
public void method()
{
System.out.println(MyClass.j);
}
}
```

Use of 'this' keyword

- Used to refer to class members of the current object.
- Used within class definition ONLY.

Use of 'this'

```
class Sample
{
int i;
void myMethod(int i)
{
    this.i=i;
}
}
```

Command Line Arguments

- **Use of main method**

```
public static void main(String arg[])
{
    System.out.println(arg[0]);
    System.out.println(arg[1]);
}
```

- **Command line arguments**

```
java Student 153 Jerry
```

Constructors

- Use of constructors
- Rules of writing
  - No return type
  - same as class name
  - can be overloaded

Constructor..

```
class Student
{
int rno;
public Student()
{
    rno=100;
}
}
```

Overloading Constructors

```
class Student
{
int rno;
      public Student()
      {    rno=100;
      }
      public Student(int j)
      {
      rno=j;
      } }
```

- Creating objects of Student

....

....

```
Student s=new Student();
Student s2=new
    Student(234);
```

...

Calling a Constructor

```
class Student{
        int rno;
        public Student(){
                this(100);
        }
        private Student(int j){
                rno=j;
        }
    }
```

Packages

- A package is a collection of related classes and interfaces that provides access protection and namespace management.
- Packages are created using the *package* keyword.

Packages

§ The *package statement has to be the first statement of a program*

Eg:

```
package academics;

class Student

{

.....

}
```

Packages

- By convention, package names are in lower case
- Different packages can contain classes with the same name

Packages

- Classes belong to one package are stored in the subdirectory whose name is same as the package name.
- Package member can be accessed by using *import* keyword.
- Only public members of package are accessible to universe.

Core Java packages

java.lang

- **Provides classes that are fundamental to the design of the Java programming language**
  - ***Includes wrapper classes, String and StringBuffer, Object, …***
  - ***Imported implicitly into all packages.***

java.util

- **Contains the collections framework, event model, date and time facilities, internationalization, and miscellaneous utility classes**

java.io

- **Provides for system input and output through data streams, serialization and the file system.**

java.math

- **Provides classes for performing arbitrary-precision integer arithmetic (BigInteger) and arbitrary-precision decimal arithmetic (BigDecimal)**

java.sql

- **Provides the API for accessing and processing data stored in a data source (usually a relational database)**

java.text

- **Provides classes and interfaces for handling text, dates, numbers, and messages in a manner independent of natural languages**

The *import declaration*

- When you want to use a class from a package, you could use its *fully qualified name*

```
java.util.Scanner
```

- Or you can *import* the class, and then use just the class name

```
import java.util.Scanner;
```

- To import all classes in a particular package, you can use the * wildcard character

```
import java.util.*;
```

The *import declaration*

- All classes of the `java.lang` package are imported automatically into all programs
- It's as if all programs contain the following line:

```
import java.lang.*;
```

- That's why we didn't have to import the `System` or `String` classes explicitly in earlier programs
- The `Scanner` class, on the other hand, is part of the `java.util` package, and therefore must be imported

The default package

- All classes of the java.lang package are imported automatically into all programs
- It's as if all programs contain the following line:

```
import java.lang.*;
```

- That's why we didn't have to import the System or String classes explicitly in earlier programs

java.io package

- Streams
  - Byte Streams
  - Character Streams
- Abstract classes
  - Byte
    - *InputStream*
    - *OutputStream*
  - Character
    - *Reader*
    - *Writer*

Class visibility

- Classes
  - Can reference other classes within the same package by class name only
  - Must provide the fully qualified name (including package) for classes defined in a different package
- Include `import` statements to make other classes directly visible

Access specifiers

- Java provides four distinct access specifiers for class members.
  - private
  - protected
  - public
  - default / (package wide scope)

Access specifiers

- *public* member (function/data)
  - Can be called/modified from outside.
- *protected*
  - Can be called/modified from derived classes
- *private*
  - Can be called/modified only from the current class
- *default (if no access modifier stated)*
  - Usually referred to as “Friendly”.
  - Can be called/modified/instantiated from the same package.

Inheritance

- Inheritance is a mechanism to reuse code.
- Java uses extends keyword for inheritance.
- A class can inherit only one class at a time.

Inheritance-Example

```
class Person

{
    String name;
    String address;
    }
class Student extends Person
{
    int rollno;
    String education;
}
```

Benefits of Inheritance

- Use already available functions
- Add more functions
- If necessary, modify the available function (method overriding)

Method overriding

- A method defined in the base class can be overridden in the derived class.
- This will change the behavior in the derived class.
- The signature should be same in both classes.

Method overriding

```
class Person
{
void method()
{
System.out.println("Person");
}
}
```

```
class Student extends Person
{
void method()
{
System.out.println("Student");
}
}
```

Rules for overriding

- **A subclass CANNOT override methods that are declared *final.***
- **A subclass MUST override methods that are declared *abstract.***

Use of final

final member data

Constant member

final member function

The method can't be overridden.

final class

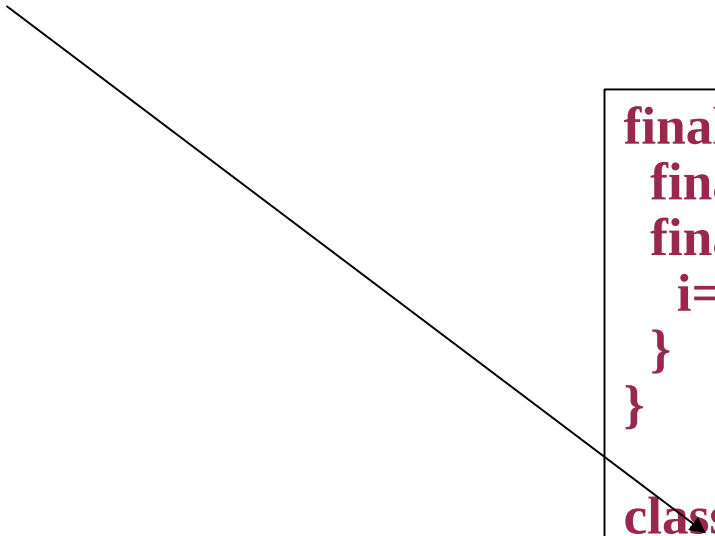
'Base' is final, thus it can't be extended

(String class is final)

```
final class Base {
  final int i=5;
  final void foo() {
     i=10;
//what will the compiler
say about this?
  }
}

class Derived extends Base
{ // Error
  // another foo ...
  void foo() {

  } }
```

final

```
Derived.java:6: Can't subclass final classes: class Base
class class Derived extends Base {
      ^
1 error
```



```
final class Base {
  final int i=5;
  final void foo() {
    i=10;
  }
}

class Derived extends Base { // Error
  // another foo ...
  void foo() {

  }
}
```

Use of *super* ***keyword***

- ***super*** **is a Java keyword that allows a method to access / refer the hidden variables and overridden methods of the super class.**

  Example

  super.regnumber;

  super.method();

The call *super()* is used to invoke the constructor of super class.

Use of *super()*

- The `super` keyword invokes the base class's constructor
  - Must be called from constructor of derived class
  - Must be first statement within constructor
  - Call must match the signature of a valid signature in the base class
  - Implicitly called in the constructor if omitted, so the base class must have a default constructor

Example

```
class Person
{
String gender;
public Person(String gender)
{
    this.gender=gender;
}
}
```

```
class Student extends Person
{
int rollno;
Student(int rollno,String
   gender)
{
    super(gender)
    this.rollno=rollno;
    }
}
```

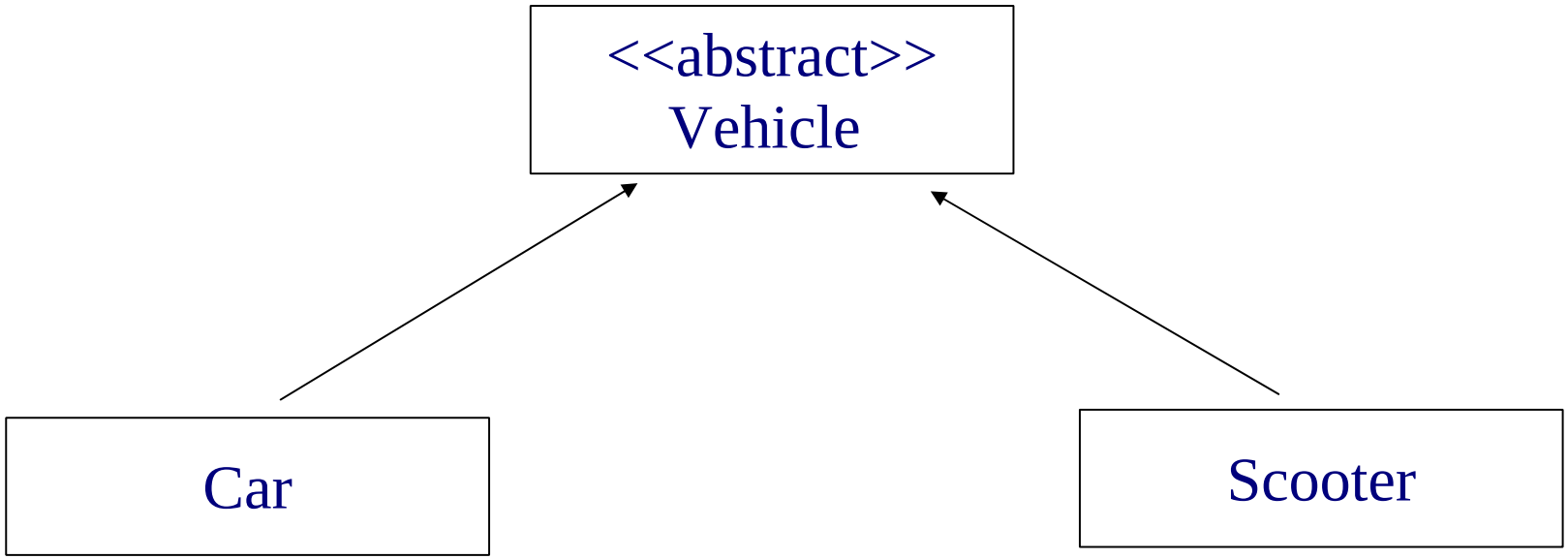
Abstract classes

- Classes without complete implementation.
- The keyword used is *abstract*.
- Abstract classes cannot be instantiated.
- Abstract classes help in bringing common functionalities in derived classes, but slight change in behaviour.

Abstract class

- **Abstract classes contain zero or more *abstract* methods, which are later implemented by concrete classes.**
- ***abstract* member function, means that the function does not have an implementation.**
  - **Abstract methods are not implemented.**
  - **Derived classes must provide method implementation.**

Abstract class



Abstract class-Example

```
abstract class Vehicle
{
int tot_distance;
float tot_fuel;
abstract void drive();
void calculateMileage()
{
    System.out.println(tot_distance/tot_fuel);
} }
```

Example-contd.

```
class Car extends Vehicle
{
void drive()
{
      //implementation to drive
         car
}
}
```

```
class Scooter extends Vehicle
{
void drive()
{
      //implementation to drive
         scooter
}
}
```

Interface

- **Pure abstract classes**
- **All methods are abstract**
- **Created using keyword *interface***

Interfaces

- An interface is a contract.
  - Every class that implements the interface must provide the interface's defined methods.
  - Each class implements the methods in its own way.
  - A class can implement multiple interfaces.
  - The `extends` keyword can be used where one intreface can extend another interface.

Interface

```
interface Polygon
{
double pi=3.1415;
void calc_area();
void draw();
}
```

- **Methods are by default public abstract final**
- **Variables are by default public static final**

Interface implementation

```
class Square implements
    Polygon
{   int side;
      public void calc_area()
      {
      System.out.println(side*side);
      }
      public void draw()
      {
            // code to draw square
      } }
```

```
class Triangle implements
    Polygon
{   int height,base;
      public void calc_area()
      {
      System.out.println(height*base/2
          );
      }
      public void draw()
      {   // code to draw triangle
      } }
```

Interfaces Vs. Multiple Inheritance

- Interfaces are not synonymous with multiple inheritance.
- However, a class can implement more than one interface.
- So in a way, we can tell interface is substitute for multiple inheritance.

Error handling

- Errors do occur in programming.
  - Problems opening a file, dividing by zero, accessing an out-of-bounds array element, hardware errors, and many more
- The question becomes: What do we do when an error occurs?
  - How is the error handled?
  - Who handles it?
  - Should the program terminate?
  - Can the program recover from the error? Should it?
- Java uses *exceptions* to provide the error-handling capabilities for its programs.

Exceptions

- An *exception* is an event that occurs during the execution of a program that disrupts the normal flow of instructions.
  - Represented as an object in Java
- Throwing an exception
  - An error occurs within a method. An exception object is created and handed off to the runtime system. The runtime system must find the code to handle the error.
- Catching an exception
  - The system searches for code to handle the thrown exception. It can be in the same method or in some method in the call stack.

Handling Exceptions

- **Three statements help define how exceptions are handled:**
  - ***try-*** identifies a block of statements within which an exception might be thrown
  - ***catch* -** must be associated with a try statement and identifies a block of statements that can handle a particular type of exception. The statements are executed if an exception of a particular type occurs within the try block. A try statement can have multiple catch statements associated with it.
  - ***finally* -** must be associated with a try statement and identifies a block of statements that are executed regardless of whether or not an error occurs within the try block. Even if the try and catch block have a return statement in them, finally will still run.

Handling Exceptions

- **General form:**

```
try {
    statement(s);
} catch (ExceptionType name) {
    statement(s);
} finally {
    statement(s);
}
```

Example of throw and catch

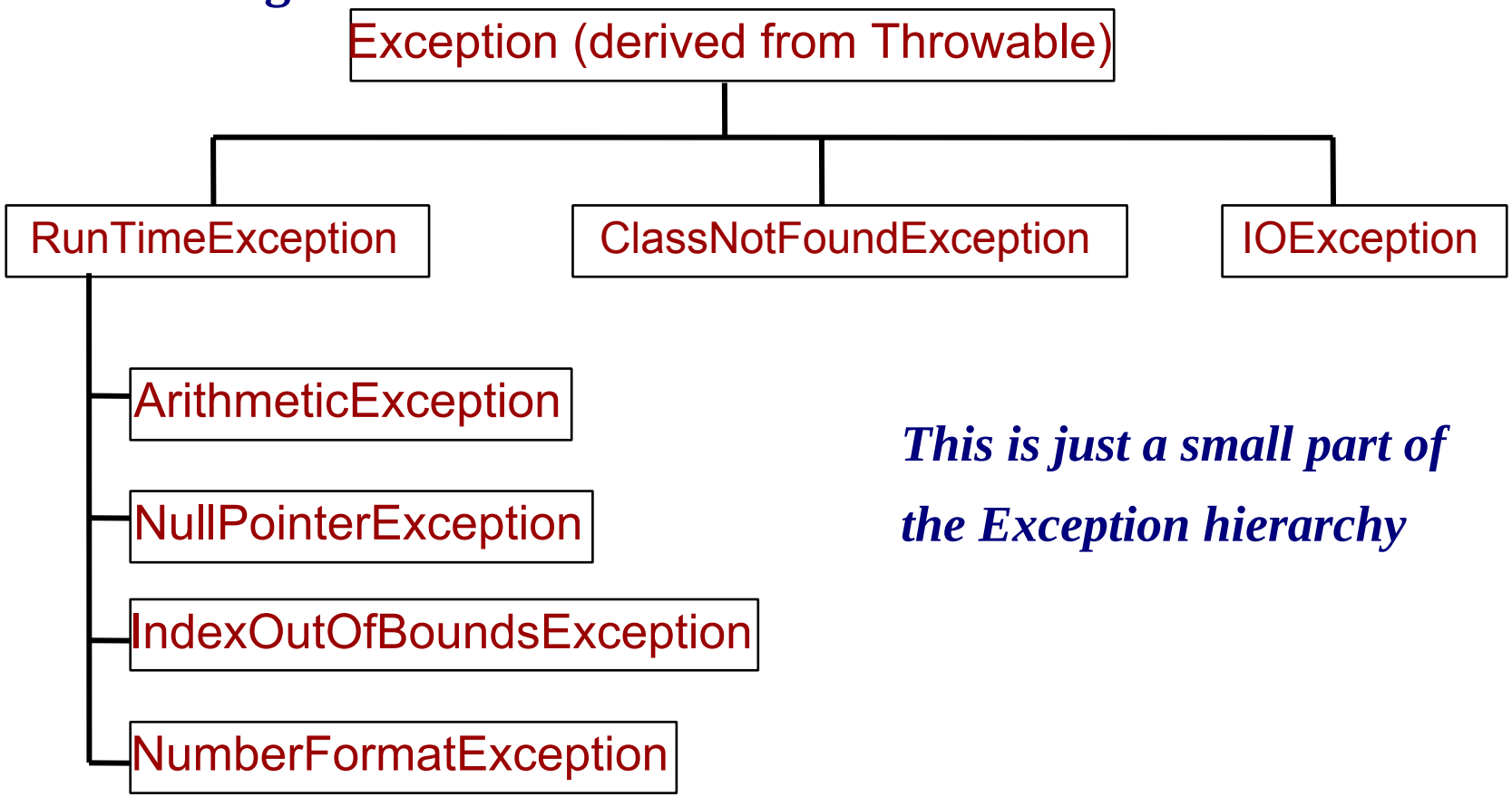
```
public void takeRisk() throws BadException {
    if (abandonAllHope) {
        throw new BadException();
    }
}

public void crossFingers() {
    try {
        anObject.takeRisk();
    } catch (BadException e) {
        System.out.println("Uh. Oh.");
        e.printStackTrace();
    }
}
```

The getMessage() method returns a string explaining why the exception was thrown.

The printStackTrace() method prints the call stack trace.

Exception Class Hierarchy

- **Java has a predefined set of exceptions and errors that can occur during execution.**



This is just a small part of the Exception hierarchy

Types of Exception

- **An exception is either** *checked* **or** *unchecked*
  - **checked- must be explicitly handled**
  - **unchecked- can be ignored**
    - ***The only unchecked exceptions in Java are objects of type*** `RuntimeException` ***or any of its descendants***
- **Why is** `RuntimeException` **ignored? (any guess…)**

Zero.java (ArithmeticException example)

```
public class Zero {

   public static void main(String[] args) {
      int numerator = 10;
      int denominator = 0;
      System.out.println(numerator/denominator);
      System.out.println("We never get to this statement.");
   }
}
```

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Zero.main(Zero.java:6)
```

Zero.java (ArithmeticException - Handled example)

```
public class Zero {
   public static void main(String[] args) {
      try{
      int numerator = 10;
      int denominator = 0;
      System.out.println(numerator/denominator);
      }
      catch {
      System.out.println("Hurrah! We Caught the  Exception.");
            }
   }
}
```

Zero.java (ArithmeticException - Handled Another example)

```
public class Zero {
   public static void main(String[] args) {
      int numerator = 10;
     int denominator = 0;
      int result=0;
      try{
            result= numerator/denominator;       }
      catch {
            System.out.println("Hurrah! We Caught the Exception.");
            }
      finally{            System.out.println("The Result: "+result);
            }   } }
```

Exception Occurrence

Raised implicitly by system

Raised explicitly by programmer

- *throw Statement*

Throwable class or its sub class

```
throw   ThrowableObject;
```

Exception Occurrence - Example

```
class ThrowStatement extends Exception  {
        public static void exp(int ptr) {
                if (ptr == 0)
                        throw new NullPointerException();
        }
        public static void main(String[] args) {
                int i = 0;
               ThrowStatement.exp(i);
        }
}
```

```
java.lang.NullPointerException
at ThrowStatement.exp(ThrowStatement.java:4)
at ThrowStatement.main(ThrowStatement.java:8)
```

Exception Definition

Treat exception as an object

All exceptions are instances of a class extended from Throwable class or its subclass

Generally, a programmer makes new exception class to extend the Exception class which is subclass of Throwable class.

User Exception (Example)

```
class UserErr extends Exception  { }


class UserClass {
  public void fun() {
        UserErr x = new UserErr();
        // ...
        if (val < 1) throw x;
  }
}
```

User Exception (Example)

```
class MainClass {
    public static void main(String args[]) {
     try{
     UserClass x = new UserClass();
            x.fun();
     }
     catch(UserErr e) {
      System.out.println("Hurrah!  Caught the User Err");
     }
        }
    }
```

User Exception (Example)

We can pass a message for the exception in string form

```
class UserErr extends Exception {

        UserErr(String s)  super(s);
         // constructor

    }

    class UserClass {

         // ...

         if (val < 1) throw new
UserErr("user exception throw
message");

     }
```

User Exception (Example)

```
class MainClass {
    public static void main(String args[]) {
     try{
     UserClass x = new UserClass();
            x.fun();
     }
     catch(UserErr e) {
      System.out.println("Hurrah!  Caught the User Err");
      Sytem.out.println(e.getMessage());
     }
        }
    }
```

Exception Propagation

```
public  class  Propagate {
        void  orange() {
              int  m = 25,  i = 0;
              i = m / i;
        }
        void  apple() {
              orange();
        }
        public static void main(String[] args) {
              Propagate p = new Propagate();
              p.apple();
        }
  }
```

ArithmeticException Occurred

Output by Default Exception Handler

```
java.lang.ArithmeticException:       / by zero
        at Propagate.orange(Propagate.java:4)
        at Propagate.apple(Propagate.java:8)
        at Propagate.main(Propagate.java:11)
```

Exception Propagation

Explicit Description for possibility of Exception Occurrence

System-Defined Exception

- *Do not need to announce the possibility of exception occurrence*

Programmer-Defined Exception

- *When it is not managed in correspond method, the exception type should be informed.*
- *Use the throws clause (Ducking of Exception)*

Exception Ducking

- **If you don't want to handle the exception, you can duck it by declaring it.**

```
public class Washer {
   Laundry laundry = new Laundry();

   public void foo() throws ClothingException {
      laundry.doLaundry();
   }

   public static void main(String[] args) throws ClothingException {
      Washer a = new Washer();
      a.foo();
   }
}
```

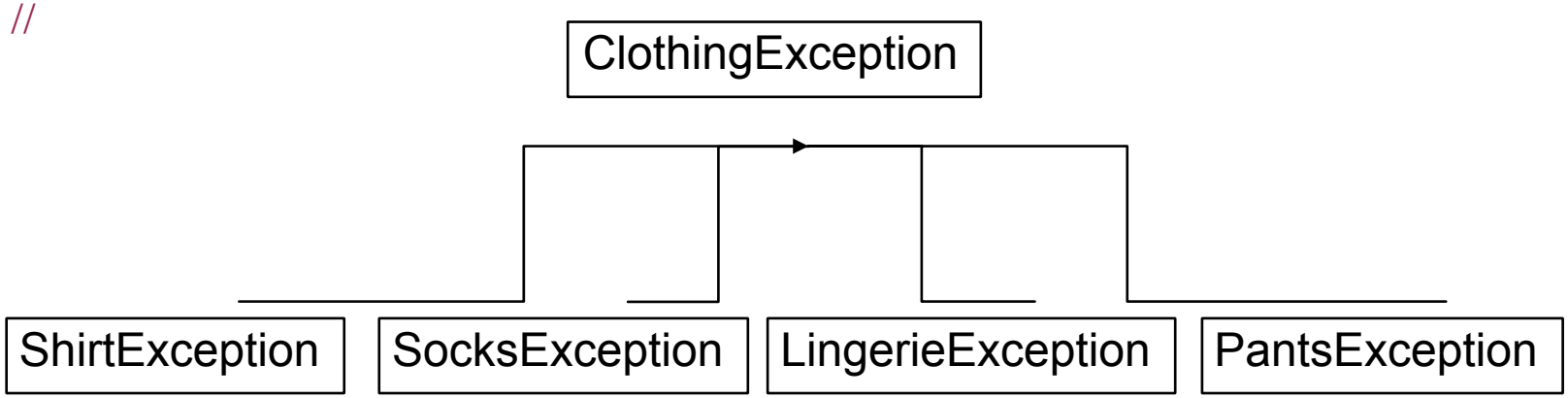
Printing the Call Stack

If you want to print all the methods and/or classes that, propagated the exception to you, then use *printStackTrace*.

```
public  class  Propagate {
        void  orange() {
              int  m = 25,  i = 0;
              i = m / i;   }
        void  apple() {
              orange();
        }
        public static void main(String[] args) {
      try{
              Propagate p = new Propagate();
              p.apple();
      }
      catch(ArithmeticException e) { e.printStackTrace(); }
    }    }
```

Polymorphic Exceptions

```
public void doLaundry() throws ClothingException { … }

try {
   laundry.doLaundry();
} catch(SocksException e) {
  //
} catch(ClothingException e) {
  //
}
```



Errors and Exception

Error Class

Critical error which is not acceptable in normal application program

Exception Class

Possible exception in normal application program execution

Possible to handle by programmer

System-Defined Exceptions

- Raised implicitly by system because of illegal execution of program
- When cannot continue program execution any more
- Created by Java System automatically
- Exception extended from Error class and RuntimeException class

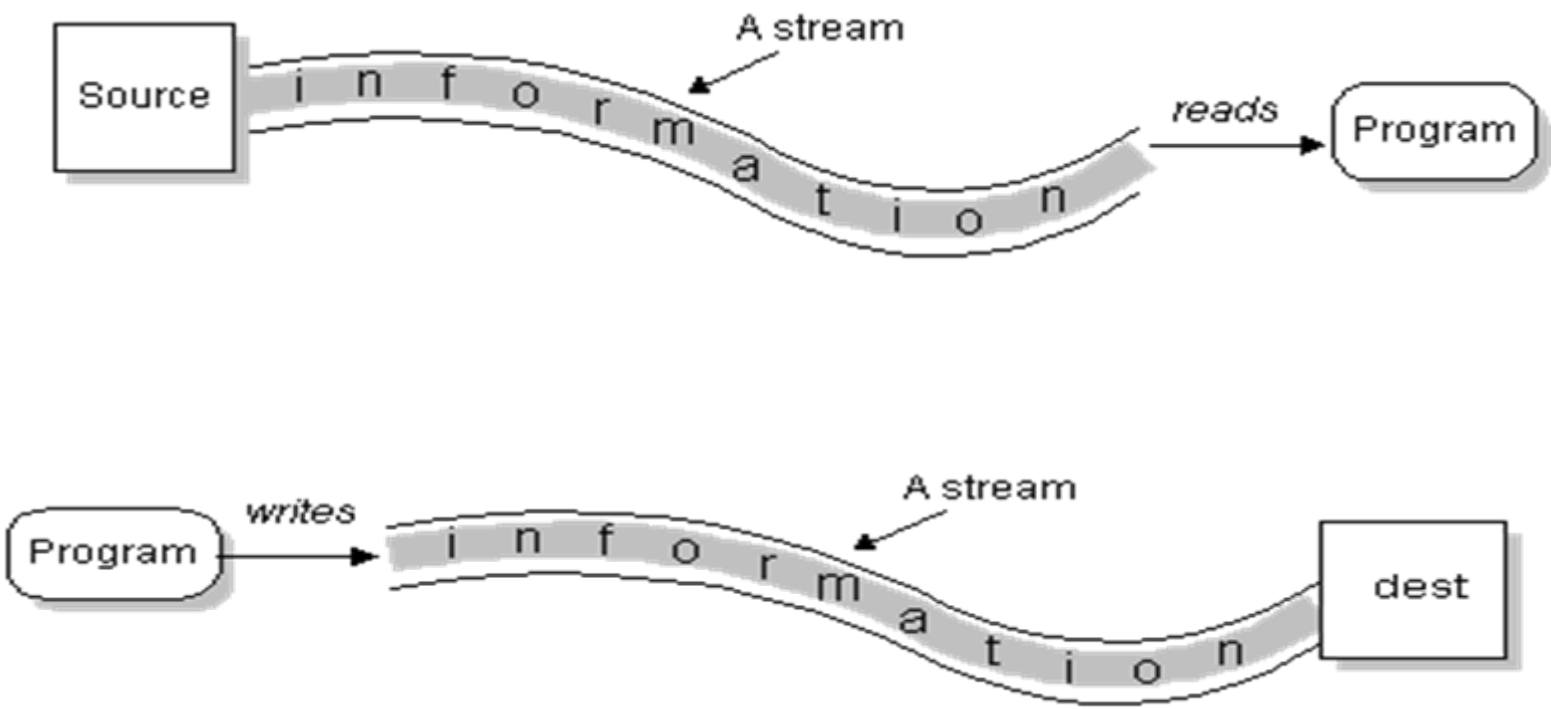
System-Defined Exceptions - Examples

- IndexOutOfBoundsException :
  - When beyond the bound of index in the object which use index, such as array, string, and vector
- ArrayStoreException :
  - When assign object of incorrect type to element of array
- NegativeArraySizeException :
  - When using a negative size of array
- NullPointerException :
  - When refer to object as a null pointer
- SecurityException :
  - When violate security. Caused by security manager
- IllegalMonitorStateException :
  - When the thread which is not owner of monitor involves wait or notify method

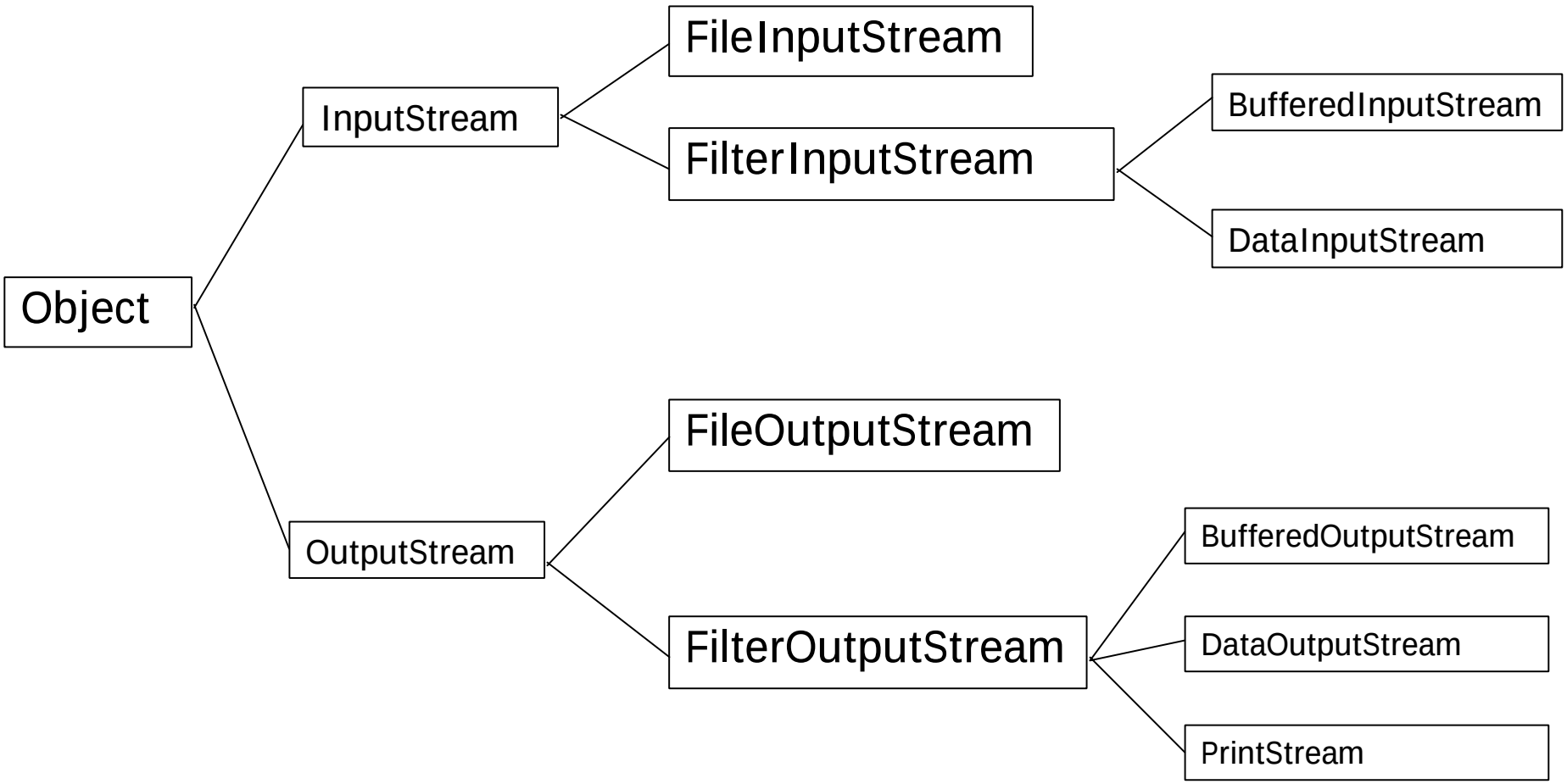
Exception Handling Review

- Exceptions must be handled or ducked
- Handle
  - Try-catch or try-catch-finally blocks
- Duck
  - Declare the exception and “pass the buck” to some other method to handle or duck again.

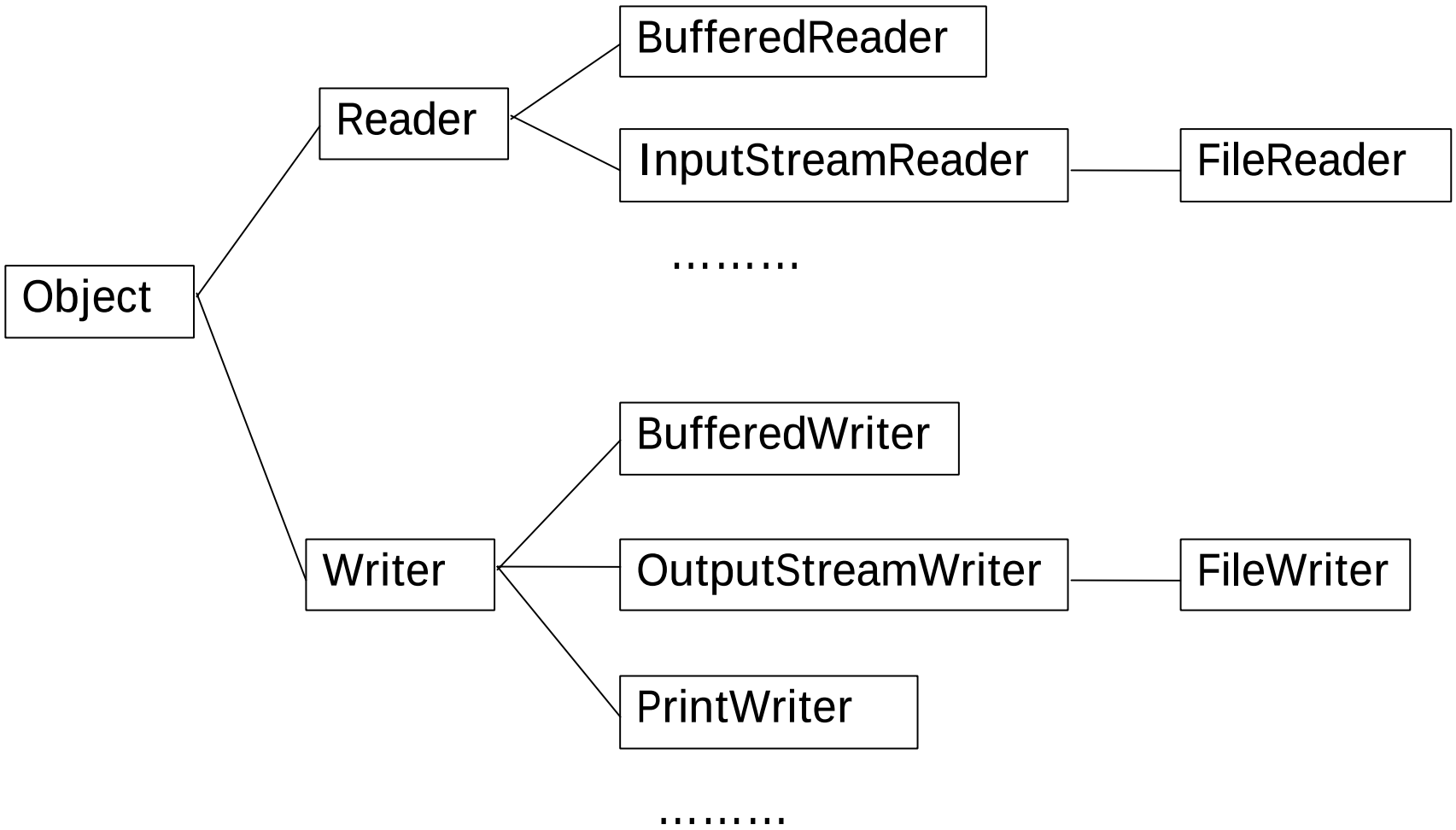
The I/O Package



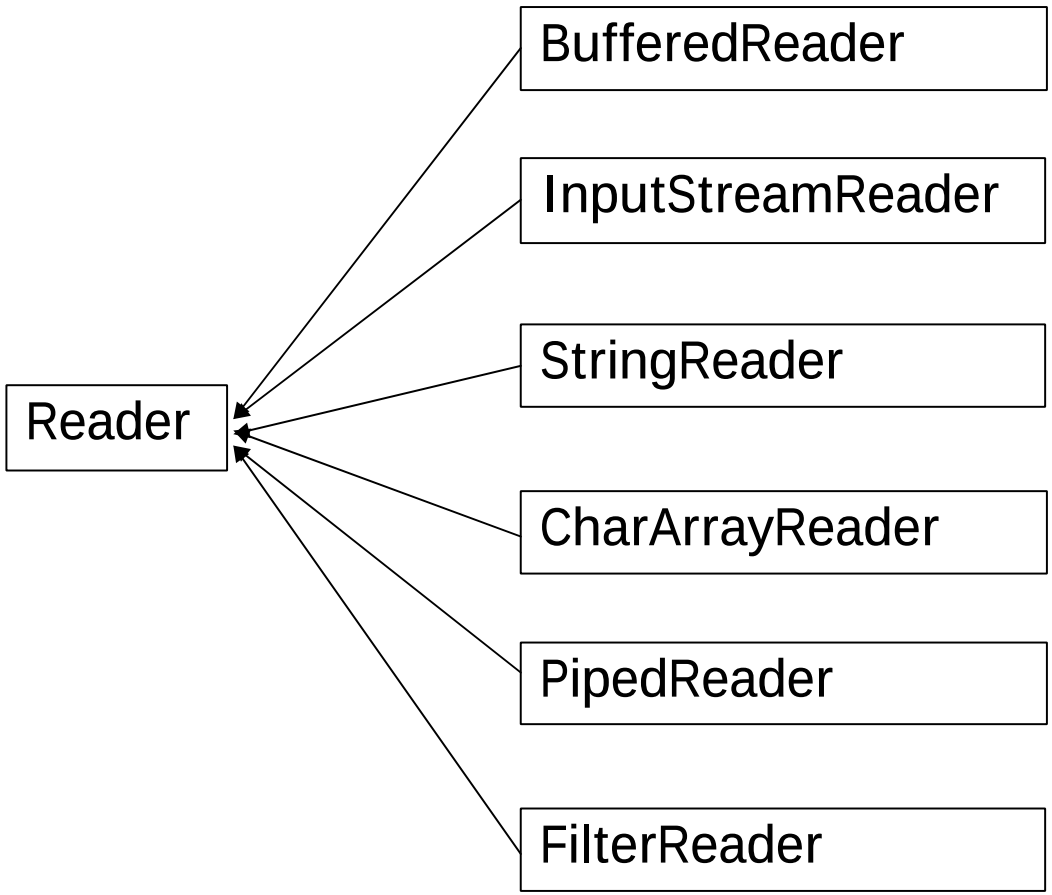
Byte Streams (Binary Streams)



Character Streams



Character Streams



Stream objects

All Java programs make use of standard stream objects

- **System.in**
  - **To input bytes from keyboard**
- **System.out**
  - **To allow output to the screen**
- **System.err**
  - **To allow error messages to be sent to screen**

Classes for file handling

Occur in package called `java.io`

`FileInputStream` **and** `FileOutputStream` **perform file input and output respectively**

- `FileReader` **and** `FileWriter`
  - **are used to read and write characters to a file**
- `DataInputStream` **and** `DataOutputStream`
  - **allow a program to read and write binary data using an** `InputStream` **and** `OutputStream` **respectively**
- `ObjectInputStream` **and** `ObjectOutputStream`
  - **deal with Objects implementing** `ObjectInput` **and** `ObjectOutput` **interfaces respectively**

Text file reading and writing

```
import java.io.*;
public class InAndOut {
public static void main(String[] args)throws IOException {
File inputFile = new File(“myInfile.txt");
File outputFile = new File(“myOutFile.txt");
FileReader inData = new FileReader(inputFile);
FileWriter outData = new FileWriter(outputFile);
int c;
while ((c = inData.read()) != -1)
        outData.write(c);
inData.close();
outData.close();
}
}
```

Creating a text file

```
import java.io.*;
import java.awt.*;
public class PriceListWriter{
 public static void main( String args[ ]  ) throws IOException {
   PrintWriter outfile = new PrintWriter(
                        new BufferedWriter(
                  new FileWriter(                        new File( "pricelist.txt" ) ) ) );
    outfile.println( "Sugar" );
    outfile.println( "0.84" );
    outfile.println( "Butter" );
    outfile.println( "1.02" );
    outfile.close( );
    System.exit( 0 );
  } }
```

Reading a text file

```
import java.io.*;
public class PriceListReader {
  public static void main( String args[ ] ) throws    IOException {
    String line;
    BufferedReader infile = new BufferedReader(
                          new FileReader (
                new File( "pricelist.txt" ) ) );
    line = infile.readLine( );
    while (line != null){
      System.out.println(line);
      line = infile.readLine( );
    }
    System.out.println("End of list");
    infile.close( );
    System.exit(0);} }
```

Reading characters from a file

```
// Program to read file in from disk, displaying contents of file on screen.
 import java.io.*;
 public class FileEcho {
  public static void main( String args[] ) {
 File  inFile = new File("mynumbers.txt");
      FileReader in = null;       int ch;
      try {
                in = new FileReader( inFile);
                while( ( ch = in.read()) != -1) { //check to see if there are characters left to read
                //use casting to ensure a character is printed
                  System.out.print( (char)  ch );    } //use casting to ensure a character is printed
      } //try
catch ( IOException  e) {System.err.println ( "FileEcho: " + e.getMessage() );}
  finally {                          //ensure file is closed regardless of error
                try { if ( in != null ) { in.close();} //if   } // try
                catch ( IOException e ) {
   System.out.println( "FileEcho: " +e.getMessage() );
    }//catch    } // finally    } //main} //FileEcho
```

Important point to note

Data must be read in in the same form that it is written out to a file

Writing

```
output = new ObjectOutputStream
( new FileOutputStream( filename ) );
output.writeObject( objectname );
output.close( );
```

Reading

```
input = new ObjectInputStream
( new FileInputStream( filename ) );
record = ( ObjectType ) input.readObject( );
input.close( );
```

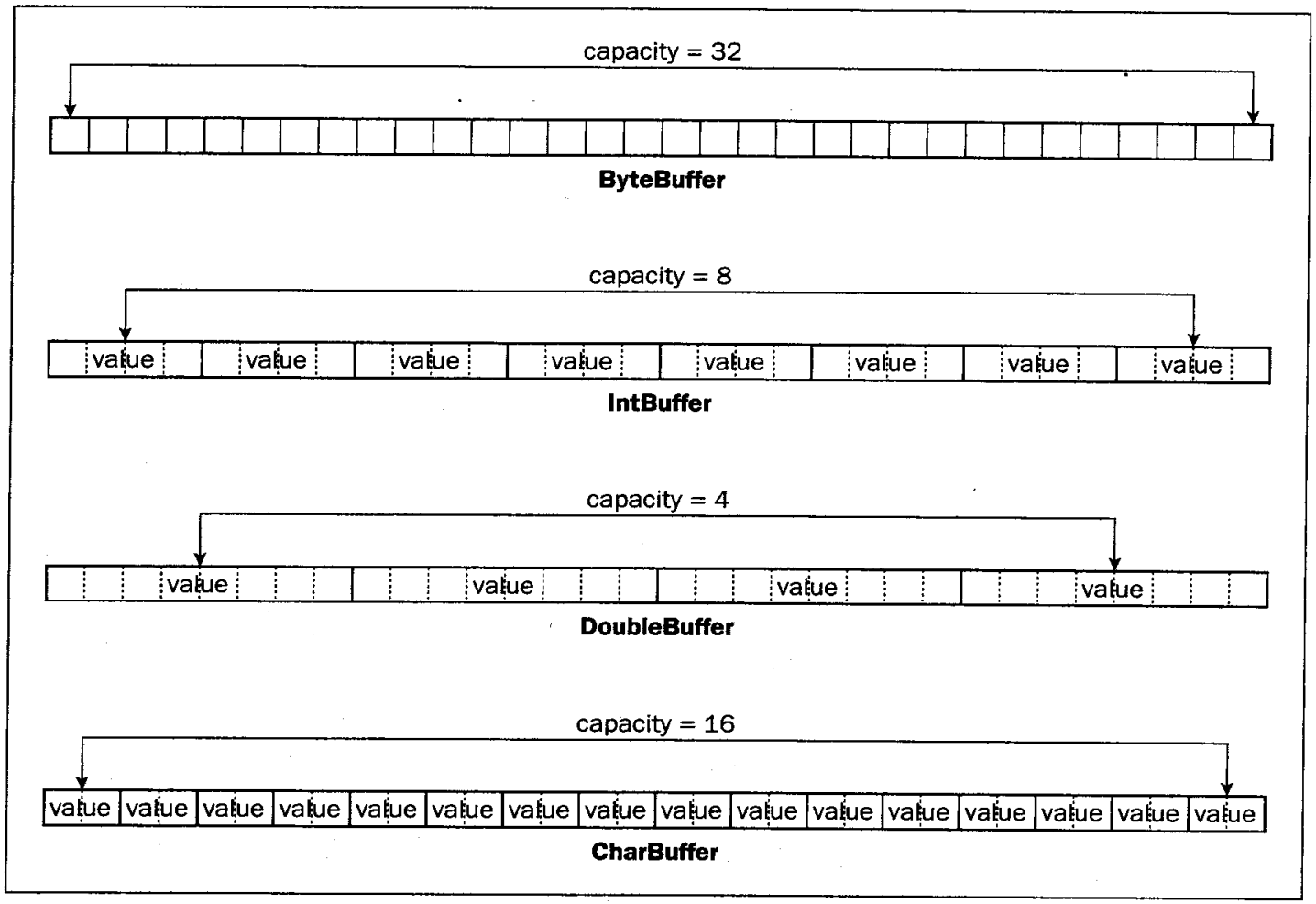
Accessing Files

- **Channels**
  - **Channels were introduced in the 1.4 release of Java to provide a faster capability for a faster capability for input and output operations with files, network sockets, and piped I/O operations between programs than the methods provided by the stream classes.**
  - **The channel mechanism can take advantage of buffering and other capabilities of the underlying operating system and therefore is considerably more efficient than using the operations provided directly within the file stream classes.**

- **A summary of the essential role of each of them in file operations**
  - **A *File* object encapsulates a path to a file or a directory, and such an object encapsulating a file path can be used to construct a file stream object.**
  - **A *FileInputStream* object encapsulates a file that can be read by a channel. A FileoutputStream object encapsulates a file that can be written by a channel.**
  - **A buffer just holds data in memory. The loaded data to be written to a file will be saved at buffer using the buffer's put() method, and retrieved using buffer's get() methods.**
  - **A *FileChannel* object can be obtained from a file stream object or a *RandomAccessFile* object.**

Accessing Files

The Capacities of Different Buffers



Collections

- Collection/container
  - object that groups multiple elements where eacd element is an object
  - used to store, retrieve, manipulate, communicate aggregate data
- Iterator - object used for traversing a collection and selectively remove elements

Java Collection Framework

- The Java collection framework is a set of utility classes and interfaces.
- Designed for working with collections of objects

Types of collections

Simple Collections

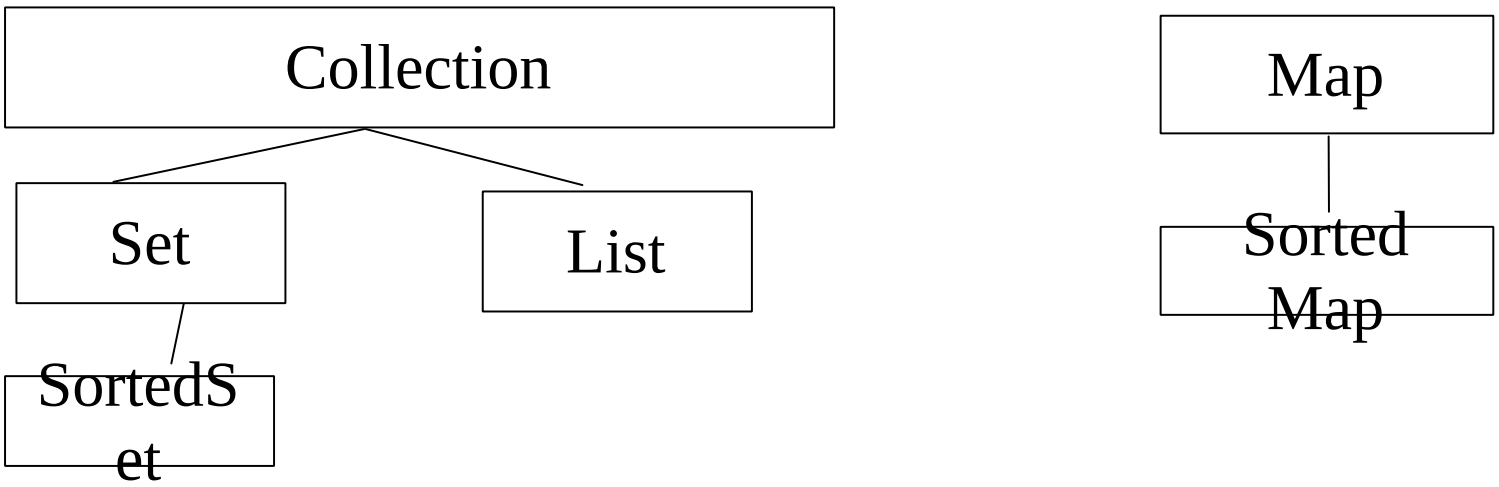
- **Sets have no duplicate elements.**
- **Lists are ordered collections that can have duplicate elements.**

Maps

- **Map uses key/value pairs to associate an object (the value) with a key.**

Both sets and maps can be sorted or unsorted.

Collection Interfaces



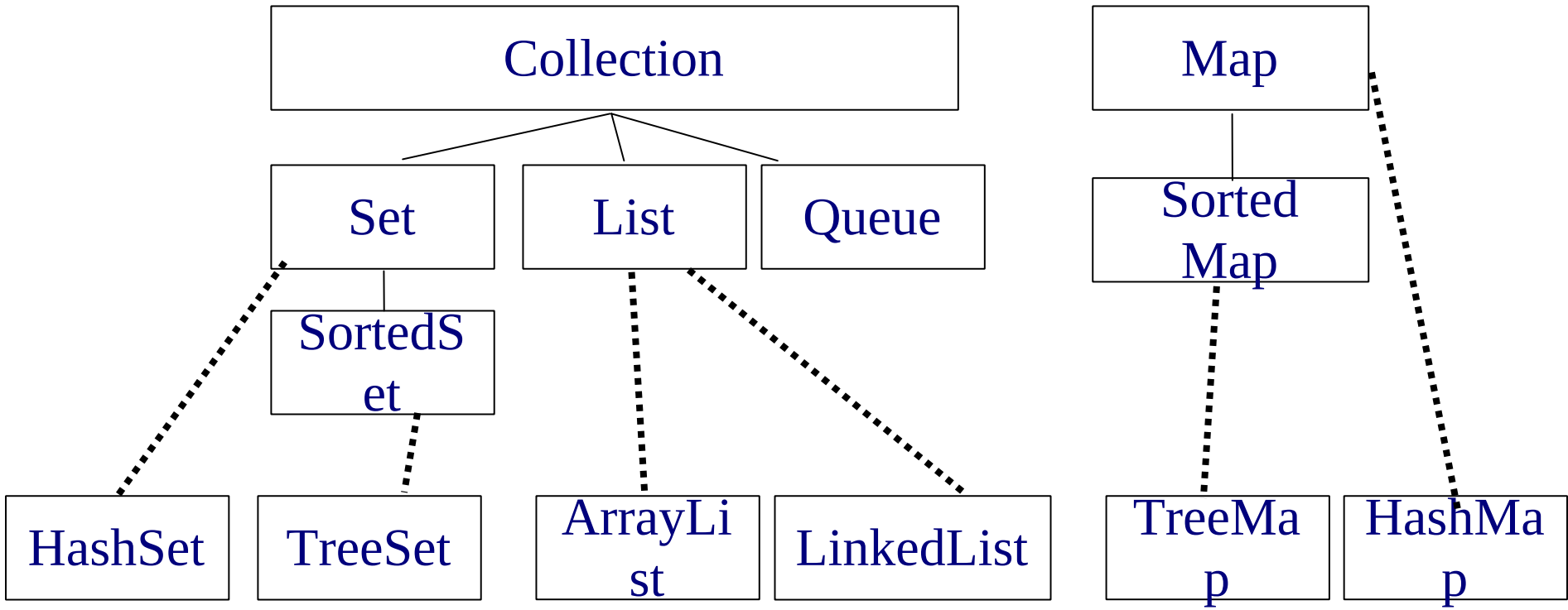
Collection Interface

- **Basic Operations**
  - **int size();**
  - **boolean isEmpty();**
  - **boolean contains(Object element);**
  - **boolean add(E element);**
  - **boolean remove(Object element);**
  - **Iterator iterator();**

Collection interface

- **Bulk Operations**
  - **boolean containsAll(Collection c);**
  - **boolean contains(Object o)**
  - **boolean addAll(Collection c);**
  - **boolean add(*Object o*)**
  - **boolean removeAll(Collection c);**
  - **boolean retainAll(Collection c);**
  - **void clear();**
  - **boolean isEmpty();**
- **Array Operations**
  - **Object[] toArray();**

General Purpose Implementations




```
List list2 = new LinkedListc);
List list1 = new ArrayList(c);
```

Methods of Interface `List`

- add(*i*,*element*)
- add(*element*)
- addAll(*collection*)
- addAll(*i*,*collection*)
- get(*i*)
- indexOf(*element*)
- lastIndexOf(*element*)
- containsAll(*collection*)
- listIterator()
- listIterator(*i*)
- remove(*i*)
- remove(*element*)
- removeAll(*collection*)
- set(*i*,*element*)
- subList(*i*,*j*)

Methods of Interface `Map`

- clear()
- containsKey(*k*)
- containsValue(*v*)
- entrySet()
- get(*k*)
- isEmpty()
- keySet()
- put(*k*,*v*)
- put(*map*)
- remove(*k*)
- removeAll(*collection*)
- size()
- values()

k = key

v = value

Concrete Collections

Sets

- HashSet
- LinkedHashSet
- TreeSet[1]

Lists

- Array List
- Linked List
- Vector

Maps

- HashMap
- IndentityHashMap
- LinkedHashMap
- TreeMap[1]
- Hashtable

[1]*sorted*

Set

- A set is a collection with no duplicate elements.
- A `HashSet` stores elements in a hash table.
- A `LinkedHashSet` is an ordered hash table.
- A `TreeSet` stores elements in a balanced binary tree.

HashSet

- Based on a hash table, a `HashSet` is a powerful data structure that can be used to retrieve objects quickly in a set.
- Unordered collection
- A hash code is used to organize objects in a `HashSet`

TreeSet

- `TreeSet` provides the properties of a set in a sorted collection.
  - Objects can be inserted in any order but are retrieved in sorted order
  - Uses `Comparable` interface to compare objects for sorting
  - Primitive types and `String` objects are comparable
  - Up to programmer to implement the `compareTo()` method for user-defined objects

Map

- An object that maps a key to a value
  - Key maps to one value
  - A map cannot have duplicate keys

Often used to associate a short key with a longer value

- Example: Dictionary
- Example: Employee database using employee ID number

Map implementation

- `TreeMap` orders the keys.
- `HashMap` stores the keys in a hash table.
- `HashMap` class is generally preferred for its efficiency (speed) unless sorted keys are needed.

Retriving Map elements

- Use the keys collection to retrieve the keys in a map.

```
Iterator keyIterator = myMap.keySet().iterator();
```

Keys collection holds “key” objects.

- Must cast key values to the appropriate type

```
Integer key = (Integer)keyIterator.next();
```

- Retrieves the next key from `keyIterator` (collection of keys) then casts it to an `Integer` and assigns it to the variable `key`

Ordering

- An order (technically a partial order) is a transitive binary relationship between two objects. Organized by some criteria, if *a* has a certain relationship with *b* and *b* has the same relationship with *c*, then *a* has that relationship with *c*.
- A sorted collection is one that orders its elements by a particular relationship.

Defining Orders on Objects

- A class can define a natural order among its instances by implementing the `comparable` interface.
- Arbitrary orders among different objects can be defined by comparators, or classes that by implement the `comparator` interface.
- `SortedSet` and `SortedMap` are sorted abstract collections that inherit the functions of sets and maps respectively.

Iterator

- ***iterator*: an object that provides a standard way to examine all elements of any collection**
- **uniform interface for traversing many different data structures without exposing their implementations**
- **supports concurrent iteration and element removal**
- **removes need to know about internal structure of collection or different methods to access data from different collections**

Iterator interfaces

```
public interface java.util.Iterator {
  public boolean hasNext();
  public Object next();
  public void remove();
}
public interface java.util.Collection {
  ...   // List, Set extend Collection
  public Iterator iterator();
}

public interface java.util.Map {
  ...
  public Set keySet();         // keys,values are Collections
  public Collection values();  // (can call iterator() on them)
}
```

Iterators in Java

- all Java collections have a method `iterator` that returns an iterator for the elements of the collection
- can be used to look through the elements of any kind of collection (an alternative to for loop)

```
List list = new ArrayList();
... add some elements ...

for (Iterator itr = list.iterator(); itr.hasNext()) {
  BankAccount ba = (BankAccount)itr.next();
  System.out.println(ba);
}
```

What is Reflection?

- A small set of Java classes (etc.) in the package `java.lang.reflect`, with help from the package `java.lang`
- Gives your program the ability (at runtime) to examine and manipulate itself and its execution environment, and to change what it does depending on what it finds

What can you do with Reflection?

- Determine the class of an object.
- Get information about a class's modifiers, fields, methods, constructors, and superclasses.
- Find out what constants and method declarations belong to an interface.
- Create an instance of a class whose name is not known until runtime.
- Get and set the value of an object's field, even if the field name is unknown to your program until runtime.
- Invoke a method on an object, even if the method is not known until runtime.
- Create a new array, whose size and component type are not known until runtime, and then modify the array's components.

Finding Classes

- The java runtime maintains, for each class or interface it knows about, a Class object (an immutable instance of `java.lang.Class`)
  - If you have an object, you can call it's `getClass()` method (inherited from `java.lang.Object`) to get it's Class object
  - If you just have the name of a class, you can call the static method: `Class.forName(String name)` to load and initialise the class (if necessary) and return it's Class object. Another form of this call allows you to specify which class loader should be used
  - (Don't be fooled by the existence of `java.lang.Package`, there is no way to find out all of the classes in a particular package)

General information about a `Class`

- **We've already seen** `ClassLoader getClassLoader()`
- `String getName()` **is common, especially in debugging print statements.**
- `Class [] getInterfaces()` **and** `Class getSuperclass()` **tell you what this class inherits from.**
- `int getModifiers()` **returns a (nastily encoded) view of whether the class is** `public`, `protected`, `private`, `final`, `static`, `abstract` **and/or an** `interface`

Finding out what a `Class` can do

- **(Supported by `Constructor`, `Method` and `Field`)**
- `getDeclaredConstructors()` **and** `getConstructors()` **return a** `Constructor []` **containing all the constructors for this class, or all the *public* constructors respectively.**
- `getConstructor(Class [] parameterTypes)` **returns the single (public) constructor whose signature matches the array of parameter types.** `getDeclaredConstructor` **does the same, but succeeds even if the constructor is not public.**
- **A call of** `setAccessible(true)` **on a constructor, method or field object suppresses the usual "visibility" checks.**

Finding out what a `Class` can do

- `getDeclaredMethods()` **and** `getMethods()` **do the same for the methods of the class (**`getMethods` **also includes any inherited public methods)**

- `getMethod(String name, Class [] params)` **gets the single matching method**

- `GetDeclaredFields()`,`getFields()`, `getDeclaredField(String name)` **and** `getField(String name)` **do the same for fields (attributes).**

Using Constructors, Fields and Methods

- **To invoke a `Constructor` (create a new object), call `newInstance(Object [] args)`, which returns an `Object` that you can downcast**
- **To find the type of a field, use `Class getType()`**
- **To get/set the value of a Field use `get(Object obj)` and `set(Object obj, Object value)`, where `obj` is the object with which the field is associated (ignored if the field is `static`). If the field is of primitive type use, e.g. `int getInt(Object obj)` and `void setInt(Object obj, int i)`**
- **To find the return type of a method call `Class getReturnType()`**
- **To invoke a method, call `Object invoke(Object obj, Object [] args)`**

Representing primitive types

- **Values of primitive types are reflected by values of their wrapper-types (`int` → `Integer` etc.). The primitive types themselves are represented by `Class` constants called (e.g.) `java.lang.Integer`.**

- **`TYPE` and `java.lang.Void.TYPE`. The method `isPrimitive()` returns true for these types. This allows you to differentiate between (e.g.) a method which returns an `int` and one that returns an `Integer`**

Representing arrays

- **Arrays in Java are (imperfect) objects, and are reflected as such by the `java.lang.reflect.Array` class**
- **They can be created through `Object newInstance(Class componentType, int length)` (or `int [] lengths`)**
- `void set(Object array, int index, Object value)` **and** `Object get(Object array, int index)` **set/get values, and there are set/get methods for the primitive types as in `Field`**

Compare:

```
Shape createShape(String shape) {
        if(shape.equals(“Square”))
                return new Square();
        if(shape.equals(“Triangle”))
                return new Triangle();
        // …
        return null;
}
```

With:

```
Shape createShape(String shape) {
    try {
     return Class.forName(shape).
     getConstructor(new Class [0]).
     newInstance(new Object [0]);
    } catch (Exception e) {
     return null;
    }
}
```

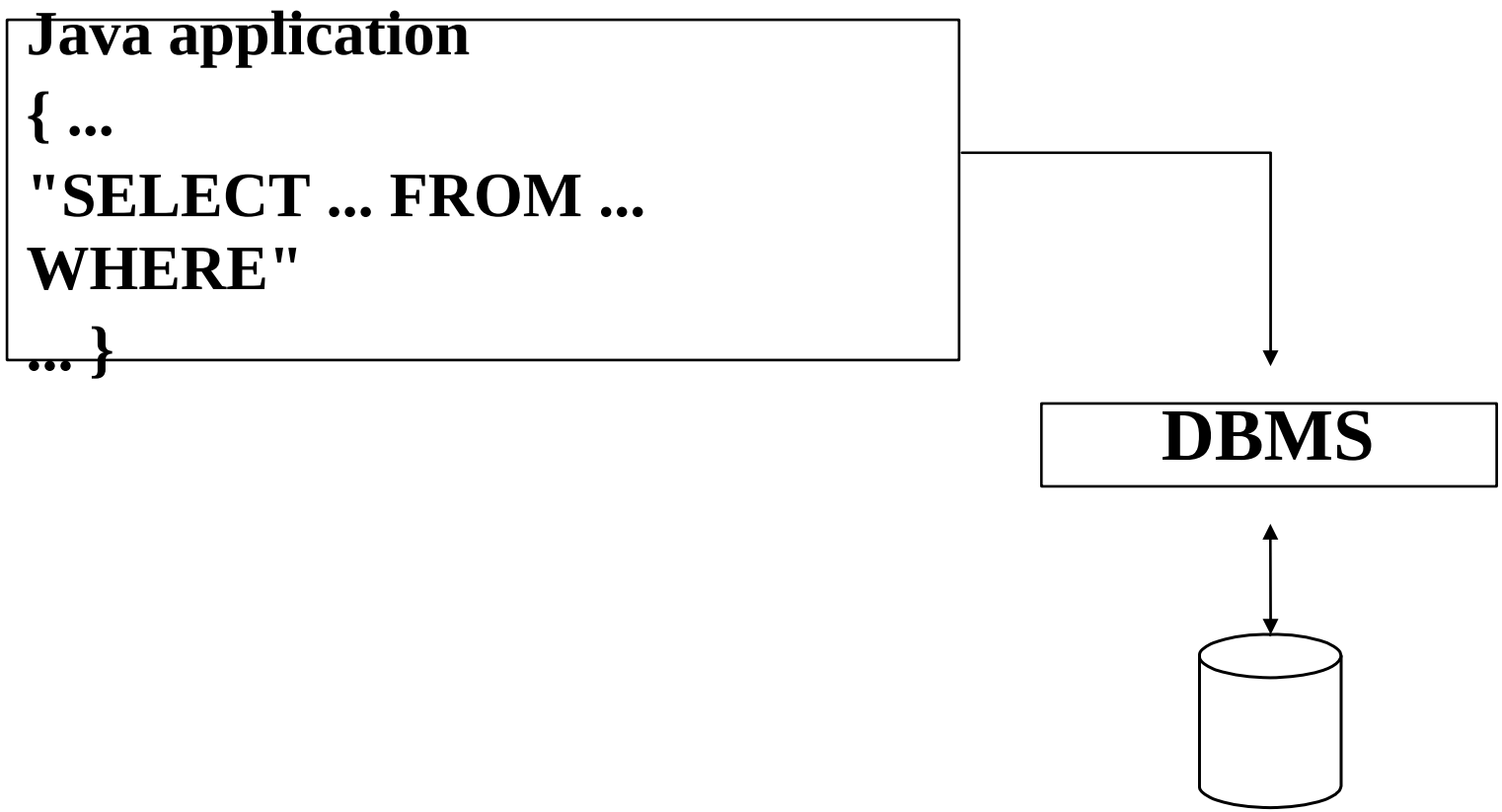
- Not as stupid as it looks, this is what `org.omg.CORBA.ORB.init()` does

Or maybe:

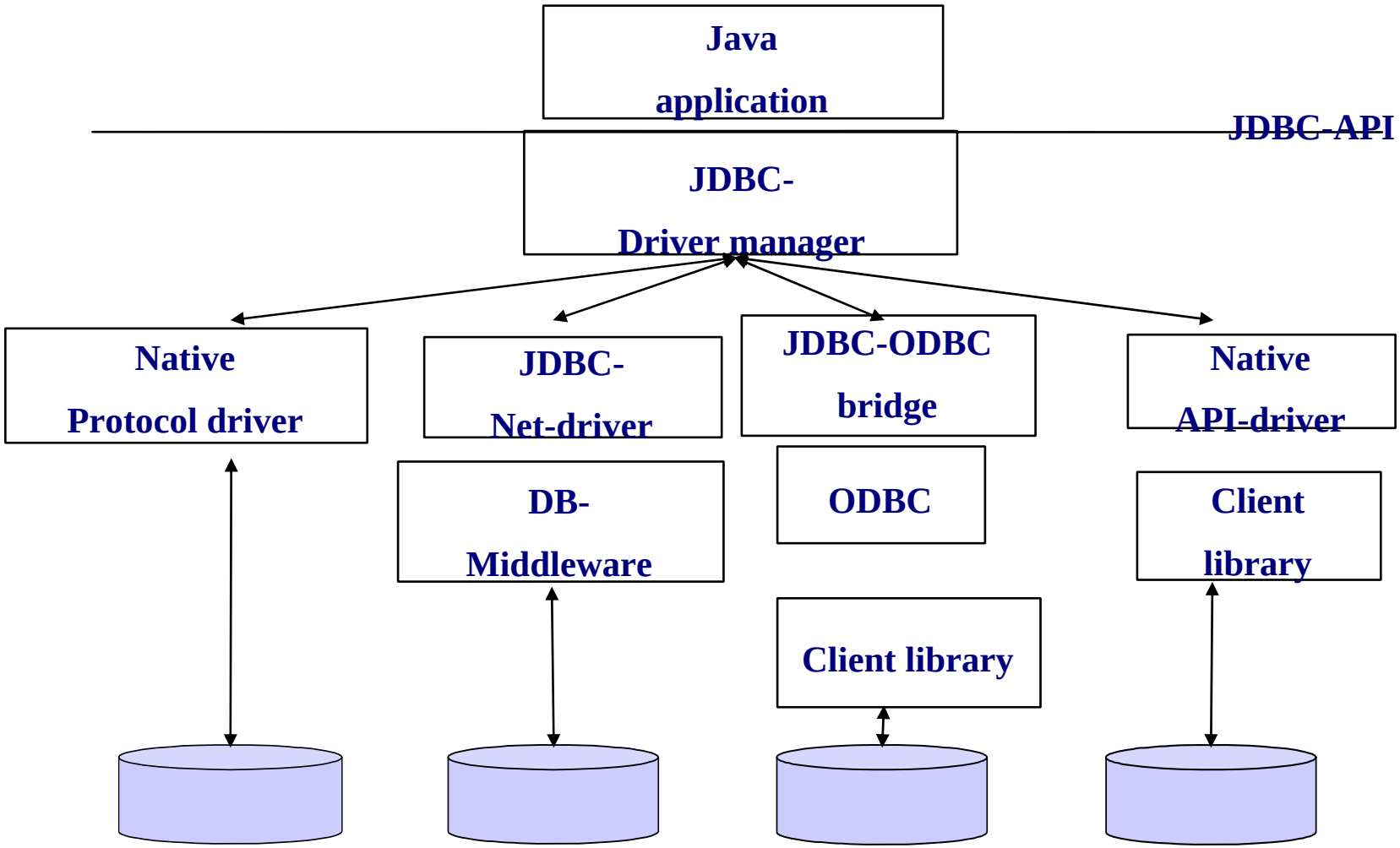
- You want to use a class which may not be present in the environment:

```
try {
    String macClassName = “com.apple.eio.FileManager”;
    Class macClass = Class.forName(macClassName);
    Method m = macClass.getMethod("openURL",
new Class [] { Class.forName("java.lang.String") });

    m.invoke(null, new String [] { url });
} catch (Exception e) {
    // Deal (via System.exec()?) with other platforms
}
```

JDBC (Java DB Connectivity)

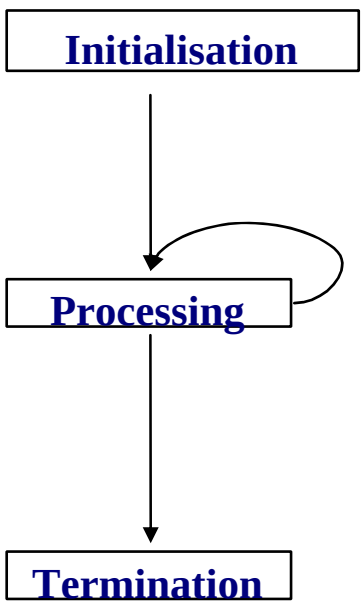


JDBC Drivers



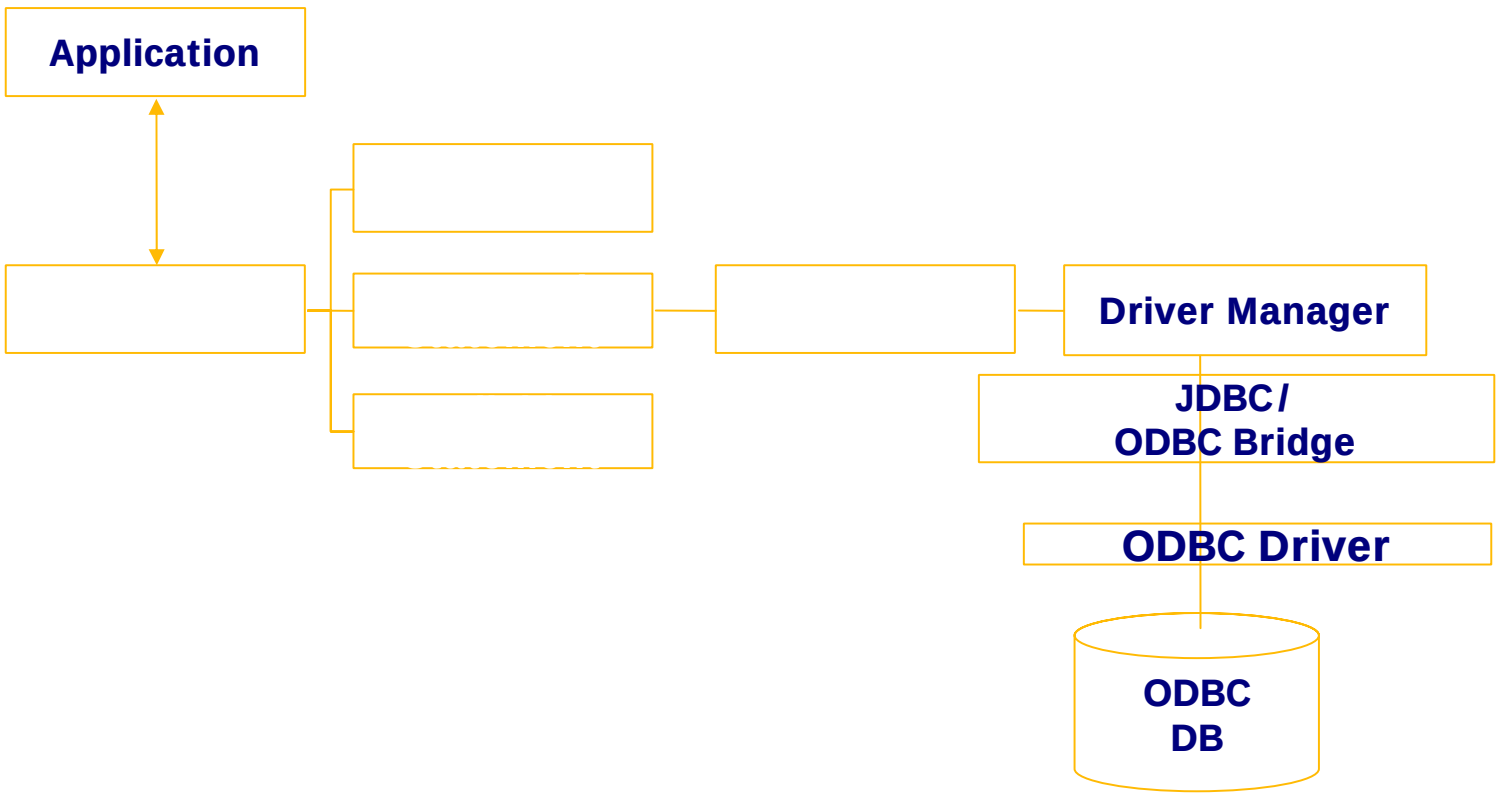
Running a JDBC Application

Phase	Task	Relevant java.sql classes
Initialisation	Load driver Create connection	DriverManager Connection
Processing	Generate SQL statements Process result data	Statement ResultSet etc.
Termination	Terminate connection Release data structures	Connection Statement etc.

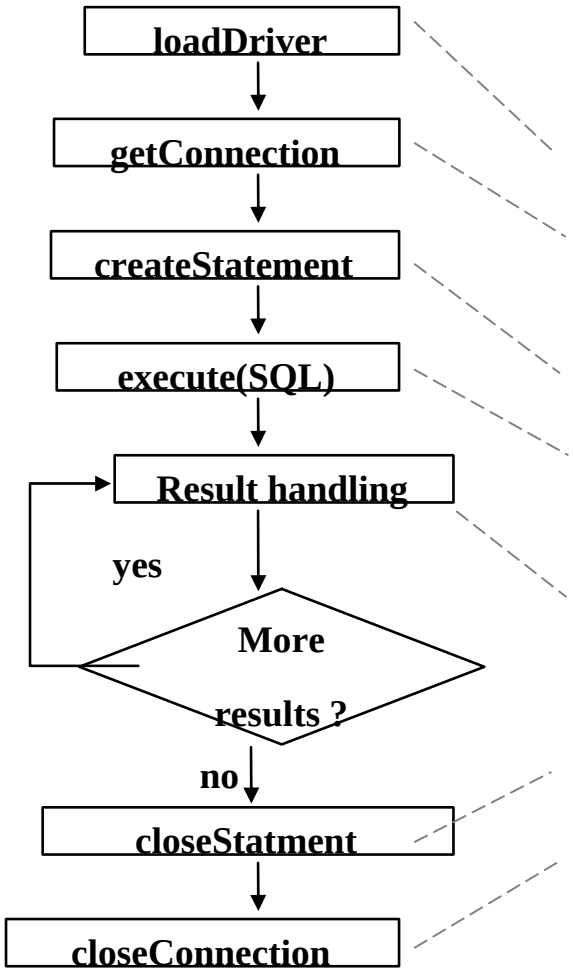


Database Access in Java

- Java provides JDBC to access relational data



A Simple JDBC application




```
import java.sql.*;
public class jdbctest {
 public static void main(String args[]){
    try{
     Class.forName("org.postgresql.Driver");
     Connection con = DriverManager.getConnection
        ("jdbc:postgresql://lsir-cis-pc8:5401/pcmdb", "user",
    "passwd");
     Statement stmt = con.createStatement();
     ResultSet rs = stmt.executeQuery
        ("select name, number from pcmtable where number < 2");
     while(rs.next())
            System.out.println(rs.getString(1) + " (" + rs.getInt(2) +
    ")");
     stmt.close()
     con.close();
    } catch(Exception e){
     System.err.println(e);     }}}
```

Loading of Driver

- Creates an instance of the driver
- Registers driver in the driver manager
- Explicit loading

```
String l_driver = "org.postgresql.Driver";
Class.forName(l_driver);
```

- Several drivers can be loaded and registered

Implicit Driver Loading

- Setting system property: jdbc.drivers
  - A colon-separated list of driver classnames.
- Can be set when starting the application
  - java -Djdbc.drivers=org.postgresql.Driver application
- Can also be set from within the Java application

```
Properties prp = System.getProperties();
prp.put("jdbc.drivers"
"com.mimer.jdbc.Driver:org.postgresql.Driver");
System.setProperties(prp);
```

- The DriverManager class attempts to load all the classes specified in jdbc.drivers when the DriverManager class is initialized.

Addressing Database

- A connection is a session with one database
- Databases are addressed using a URL of the form "jdbc:<subprotocol>:<subname>"
- Examples

  jdbc:postgresql:database

  jdbc:postgresql://host/database

  jdbc:postgresql://host:port/database

- Defaults: host=localhost, port=5432

Connecting to Database

- **Connection is established**

  ```
  Connection con =
  DriverManager.getConnection(URL,USERID,PWD);
  ```

- **Connection properties (class Properties)**
- **Close the connection**

  ```
  con.close();
  ```

Simple SQL Statements

- **Statement object for invocation**

```
stmt = conn.createStatement();
ResultSet rset= stmt.executeQuery(
        "SELECT address,script,type FROM worklist");
```

- **ResultSet object for result processing**

Impedance Mismatch

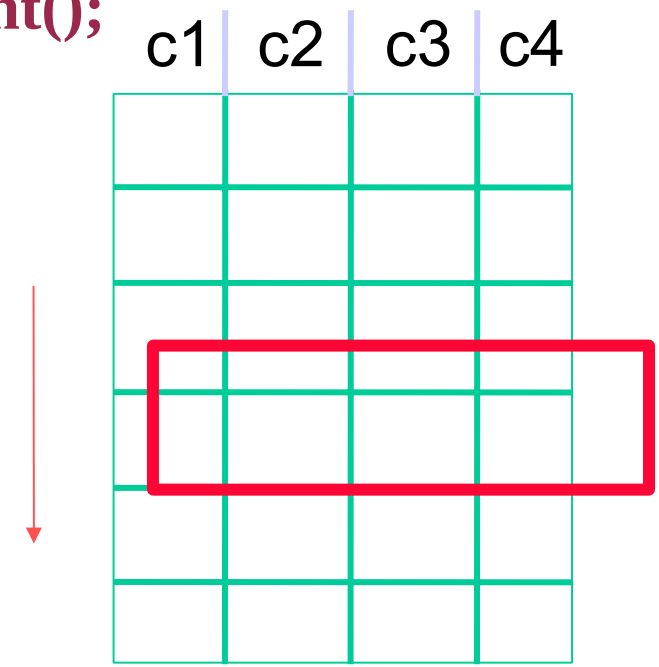
- **Example: SQL in Java:**
  - **Java uses int, char[..], objects, etc**
  - **SQL uses tables**
- **Impedance mismatch = incompatible types**
- **Why not use only one language?**
  - **SQL cannot do everything that the host language can do**
- **Solution: use cursors**

Using Cursors

- Access to tuples
  - ResultSet object manages a cursor for tuple access

Example

```
Statement stmt=con.createStatement();
ResultSet rset=stmt.executeQuery
                    (“SELECT …”);
  while (rset.next()) {
    …
  }
rset.close();
```

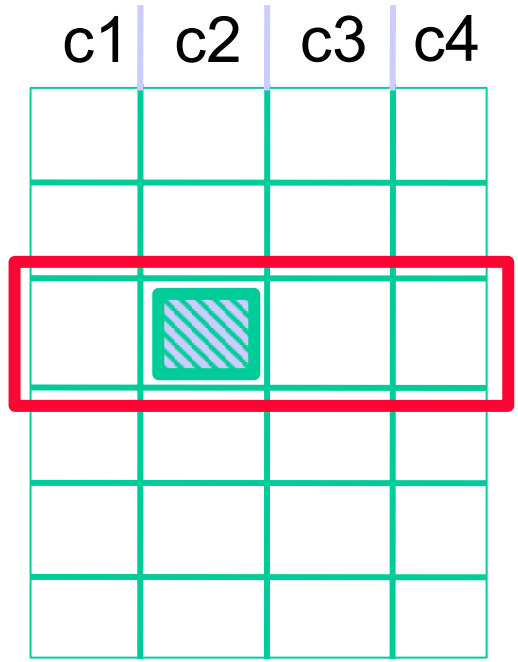


Accessing Attributes (Columns)

- **Access to columns of a tuple**
  - **Using column index or column name**

Example

```
while (rset.next())
{
//return the value of the first column as a String
String address = rset.getString(1);
//return the value of the column "type" as a String
String type = rset.getString("type")
...
}
```



More on Cursors

- Cursors can also modify a relation

  rset.updateString("script", "ebay");

  rset.updateRow(); // updates the row in the data source

- The cursor can be a scrolling one: can go forward, backward

  first(), last(), next(), previous(), absolute(5)

- We can determine the order in which the cursor will get tuples by the ORDER BY clause in the SQL query

Inserting a row with Cursors

```
rs.moveToInsertRow();  // moves cursor to the insert row
rs.updateString(1, "AINSWORTH");  // updates the
  // first column of the insert row to be AINSWORTH
rs.updateInt(2,35);  // updates the second column to be 35
rs.updateBoolean(3, true);  // updates the third column to
    true
rs.insertRow();
rs.moveToCurrentRow();
```

Dynamic JDBC Statements

- Variables within SQL statement
- Precompiled once, multiple executions
- PreparedStatement for invocation

```
PreparedStatement stmt = con.prepareStatement (
                    "SELECT * FROM data WHERE date = ?");
    stmt.setDate (1, j_date);
    ResultSet rset = stmt.executeQuery();
```

SQL Data Types

- For passing parameters to prepared statements specific SQL data types are needed
- Example

```
java.util.Date jd = new java.util.Date();
java.sql.Date j_date = new java.sql.Date(jd.getTime());
```

Update Statements

- **Updates have no result set**

```
int result = stmt.executeUpdate("delete from worklist");
```

- **Return value of executeUpdate**
  - **DDL-statement: always 0**
  - **DML-statement: number of tuples**

Error Handling

- Each SQL statement can generate errors
  - Thus each SQL method should be put into a try-block
- Exceptions are reported through exceptions of class SQLException

Example

```
Import java.sql.*;
    public class JdbcDemo {
    public static void main(String[] args) {
try {Class. forName(com.pointbase.jdbc.jdbcUniversalDriver);
   } catch (ClassNotFoundException exc) {System.out.println(exc.getMessage());}
try {Connection con = DriverManager.getConnection("jdbc:jdbc:demo","tux","penguin");
    Statement stmt = con.createStatement();
    ResultSet rs = stmt.executeQuery("SELECT * FROM data");
    while (rs.next()) {… process result tuples …}
    } catch (SQLException exc)
    {System.out.println("SQLException: " + exc.getMessage());} } }
```

Metadata

- Metadata allows to develop schema independent applications for databases
  - Generic output methods
  - Type dependent applications
- Two types of metadata are accessible
  - on result sets
  - on the database

ResultSet Metadata

- **java.sql.ResultSetMetaData**

  describes the structure of a result set object

- **Information about a ResultSet object**
  - **Names, types and access properties of columns**

Database Metadata

- java.sql.DatabaseMetaData

  provides information about the database (schema etc.)
- Information about the database
  - Name of database
  - Version of database
  - List of all tables
  - List of supported SQL types
  - Support of transactions

Example

```
ResultSet rset = stmt.executeQuery("SELECT * FROM data");

ResultSetMetaData rsmeta = rset.getMetaData();

int numCols = rsmeta.getColumnCount();

for (int i=1; i<=numCols; i++) {

    int ct     = rsmeta.getColumnType(i);

    String cn  = rsmeta.getColumnName(i);

    String ctn = rsmeta.getColumnTypeName(i);

    System.out.println("Column #" + i + ": " + cn +

      " of type " + ctn + " (JDBC type: " + ct + ")"); }
```