Advanced Java Programming

Course Outline

- Input and Output Stream
- Networking in Java
- Java Database Connectivity (JDBC)
- Java Collections
- Introduction to JavaFX
- Building UI Using JavaFX
- JavaFX Event Handling and Layouts
- Java SE 8 New Features

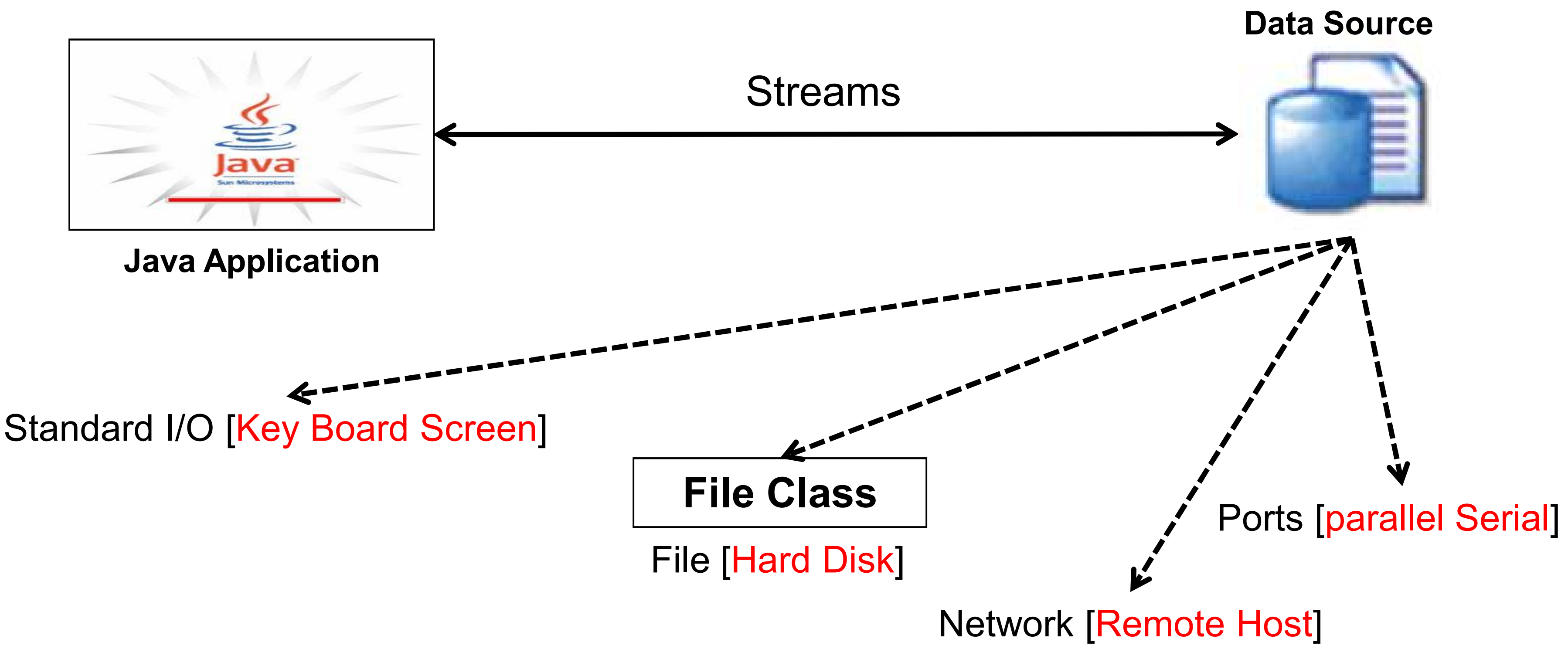
Lesson 1

Input and Output Stream

Lesson 1 Outline

- ❑ Input and Output Streams
- ❑ Low Level Streams and High Level Streams
- ❑ Working with File Streams

Streams



Data Source

Streams

Java Application

Standard I/O [Key Board Screen]

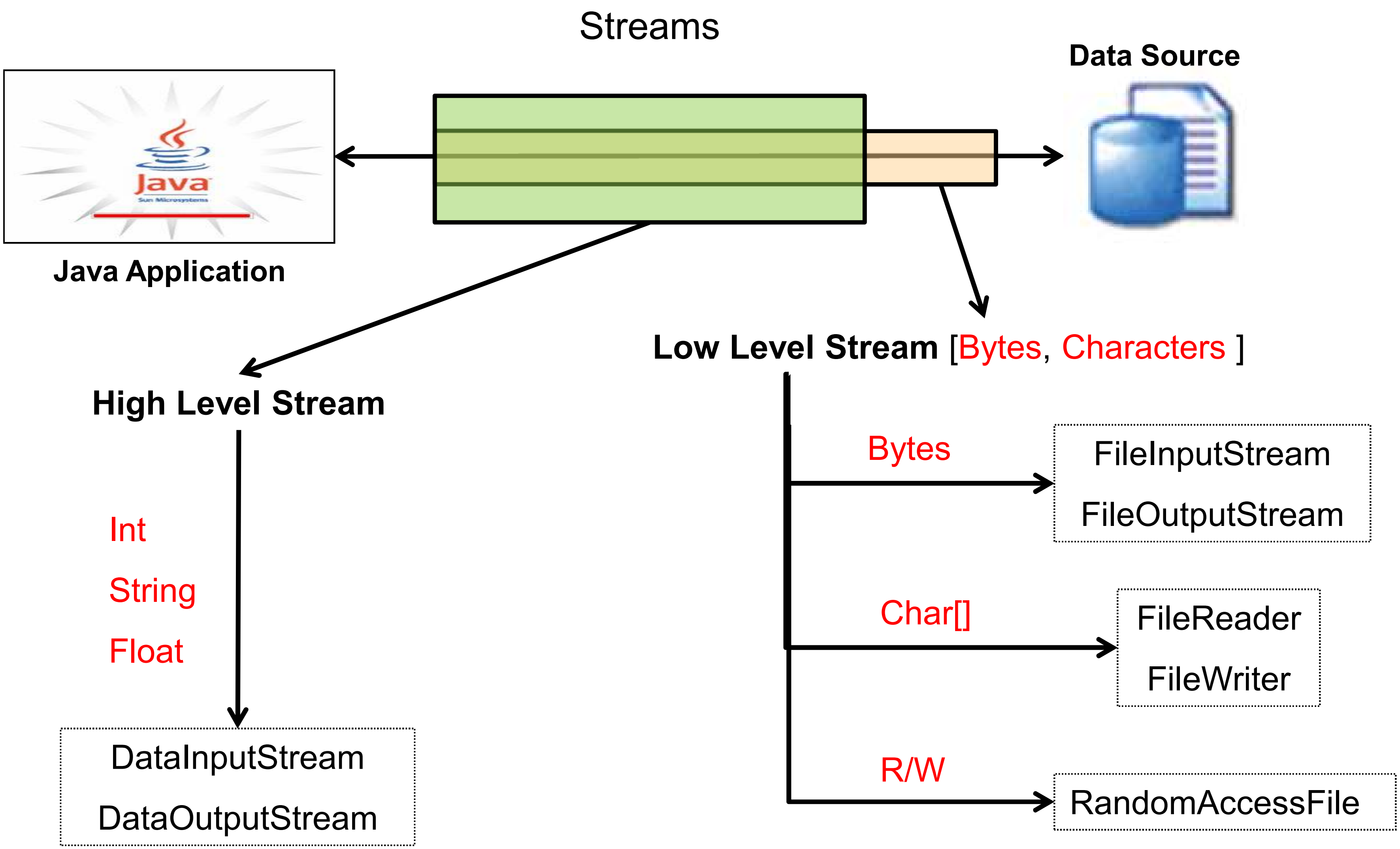
File Class

File [Hard Disk]

Ports [parallel Serial]

Network [Remote Host]

Streams



Streams

Java Application

Data Source

High Level Stream

Int

String

Float

DataInputStream

DataOutputStream

Low Level Stream [Bytes, Characters]

Bytes

FileInputStream

FileOutputStream

Char[]

FileReader

FileWriter

R/W

RandomAccessFile

Streams

- A stream is a flow of data between a Data Source and a Data Sink (Destination).
- Streams are used for data input and output.
- An Input Stream is a stream that reads input into the application. Reading is a blocking operation (i.e. it blocks its thread).
- An Output Stream is a stream that carries out data from the application.
- Streams can be classified into two categories: Low Level Streams and High Level Streams.

Low Level Streams

- A Low Level Stream is a stream that is attached directly to the source/destination.

- It can only deal with raw data in the form of bytes or characters.

High Level Streams

- A High Level Stream is a stream that is attached to a lower level stream (i.e. layered over it).

- It can deal with higher data types such as int, float, String, or even whole objects.

- A High Level Stream saves some conversion effort for the programmer. (e.g. Reading complete Strings instead of reading character by character then transferring them into a String).

File Class

- Commonly Used Constructor(s):
  - `File(String path)`
  - `File(String parent, String child)`
  - `File(File parent, String child)`

- Commonly Used Method(s):
  - `boolean exists()`
  - `boolean isFile()`
  - `boolean isDirectory()`
  - `String getName()`
  - `String getParent()`
  - `String getAbsolutePath()`

File Class Cont'd

- Commonly Used Method(s):
  - `String[] list()`
  - `boolean canRead()`
  - `boolean canWrite()`
  - `boolean delete()`
  - `long length()`
  - `boolean createNewFile()`
  - `boolean mkdir()`

FileInputStream Class

- Commonly Used Constructor(s):
  - `FileInputStream(String file)`
  - `FileInputStream(File file)`

- Commonly Used Method(s):
  - `int read()`
  - `int read(byte[] b)`
  - `int read(byte[] b, int offset, int length)`
  - `int available()`
  - `long skip(long)`
  - `void close()`

- The **`offset`** parameter determines the start index in the destination array **`b`**.

- The **`length`** parameter specifies the maximum number of bytes to read.

FileOutputStream Class

- Commonly Used Constructor(s):
  - `FileOutputStream(String file)`
  - `FileOutputStream(File file)`

- Commonly Used Method(s):
  - `void write(int b)`
  - `void write(byte[] b)`
  - `void write(byte[] b, int offset, int length)`
  - `void close()`
  - `void flush()`

- The **`offset`** parameter determines the start index at the source array **`b`**.

- The **`length`** parameter specifies the number of bytes to write.

FileReader Class

- Commonly Used Constructor(s):
  - `FileReader(String file)`
  - `FileReader(File file)`

- Commonly Used Method(s):
  - `int read()`
  - `int read(char[] c)`
  - `int read(char[] c, int offset, int length)`

FileWriter Class

- Commonly Used Constructor(s):
  - `FileWriter(String file)`
  - `FileWriter(File file)`

- Commonly Used Method(s):
  - `void write(c)`
  - `void write(char[] c)`
  - `void write(char[] c, int offset, int length)`
  - `void write(String str)`
  - `void write(String str, int offset, int length)`

RandomAccessFile Class

- Commonly Used Constructor(s):
  - **`RandomAccessFile(String file, String mode)`**
  - **`RandomAccessFile(File file, String mode)`**

- The **`mode`** parameter can assume the values “r” or “rw”.

- Commonly Used Method(s):
  - **`long getFilePointer()`**
  - **`void seek(long position)`**
  - **`long length()`**
  - **`int read()`**
  - **`int read(byte[] b)`**
  - **`int read(byte[], int offset, int length)`**
  - **`void write(int b)`**
  - **`void write(byte[] b)`**
  - **`void write(byte[] b, int offset, int length)`**

DataInputStream Class

- Commonly Used Constructor(s):
  - `DataInputStream(InputStream in)`

- Commonly Used Method(s):
  - `int readInt()`
  - `long readLong()`
  - `float readFloat()`
  - `double readDouble()`
  - `String readUTF()`

DataOutputStream Class

- Commonly Used Constructor(s):
  - `DataOutputStream(OutputStream out)`

- Commonly Used Method(s):
  - `void writeInt(int i)`
  - `void writeLong(long l)`
  - `void writeFloat(float f)`
  - `void writeDouble(double d)`
  - `void writeUTF(String str)`

Opening a Text File Example

- The following code sample is for printing a text file to the command prompt:

```
public static void main (String[] args)
 {
   FileInputStream fis = new FileInputStream(“sample.txt”);
   int size = fis.available();
   byte[] b = new byte[size];
   fis.read(b);
   System.out.println(new String(b));
   fis.close();
}
```

Saving a Text into File Example

- The following code sample is for printing data which are the arguments of the program, into a file:

```
public static void main (String[] args)
 {  FileWriter fileWriter = null;
    PrintWriter printWriter = null;
     try{
     //Opening a file in append mode using FileWriter
         fileWriter = new FileWriter("sample.txt", true);
     //Wrapping BufferedWriter object in PrintWriter
         printWriter = new PrintWriter(fileWriter);
     //Bringing cursor to next line
         printWriter.println();
      //Writing text to file
            for(String data : args){
            printWriter.println(data);
            }
       }
```

Saving a Text into File Example

```
    catch (IOException e){
         e.printStackTrace();
    }
    finally{ //Closing the resources
       try{
             printWriter.close();
             fileWriter.close();
        }catch (IOException e)
         {
             e.printStackTrace();
         }
  }
}
```

Lesson 2

Networking in Java

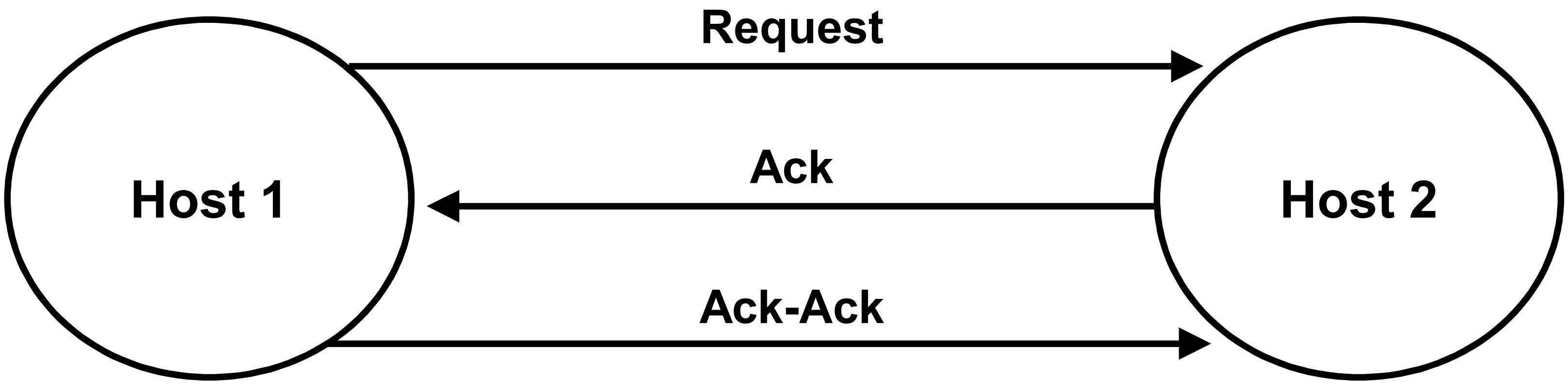
Lesson 2 Outlines

- ❑ Overview of TCP and UDP Protocols
- ❑ Basic Client/Server Communication
- ❑ Sockets and Server Sockets
- ❑ Simple Client/Server Console Application
- ❑ Chat Room GUI Application

Overview of TCP and UDP Protocols

TCP	UDP
Connection Oriented (Handshaking procedure)	Connection Less
Continuous Stream	Message Oriented
Reliable (Error Detection)	Unreliable

3-Way Handshaking Procedure



Establishing a Connection

- In order to connect to a remote host, two pieces of information are essentially required:
  - IP Address (of remote machine)
  - Port Number (to identify the service at the remote machine)

- **Socket = Address + Port**

- Range of port numbers: 0 → 65,535

- From 0 → 1,024 are reserved for well known services, such as:
  - HTTP: 80
  - FTP: 21
  - Telnet: 23
  - SMTP: 25

Basic Client/Server Communication

Server	Client
1. Create a server socket (bind the service to a certain port)	
2. Listen for connections	1. Create a socket (connect to the server)
3. Accept connection and transfer the client request to a virtual port.	
4. Obtain input and output streams	2. Obtain input and output streams
5. Send and receive data	3. Send and receive data.
6. Terminate connection (after communication has ended)	4. Terminate connection (after communication has ended)

ServerSocket Class

- Commonly Used Constructor(s):
  - `ServerSocket(int port)`
  - `ServerSocket(int port, int maxCon)`

- Commonly Used Method(s):
  - `Socket accept()`
  - `close()`

Socket Class

- Commonly Used Constructor(s):
  - `Socket(String address, int port)`
  - `Socket(InetAddress address, int port)`

- Commonly Used Method(s):
  - `InputStream getInputStream()`
  - `OutputStream getOutputStream()`

InetAddress Class

- InetAddress class has no public constructor.

- Commonly Used Method(s):
  - `static InetAddress getByName(String host)`
  - `static InetAddress[] getAllByName(String host)`
  - `static InetAddress getLocalHost()`
  - `String getHostName()`
  - `String getHostAddress()`
  - `Byte[] getAddress()`

Simple Client/Server Console Example
Server Application

- The following code sample is for creating a simple one-to-one client/server application, where each machine sends out a string and receives a string:

```
public class Server
{
   ServerSocket myServerSocket;
   Socket s;
   DataInputStream dis ;
   PrintStream ps;
   public static void main(String[] args)
   {
        new Server();
   }

   public Server()
   {
       try
       {
          myServerSocket = new ServerSocket(5005);
          s = myServerSocket.accept ();
          dis = new DataInputStream(s.getInputStream ());
          ps = new PrintStream(s.getOutputStream ());
```

Simple Client/Server Console Example
Server Application Cont'd

```
           String msg = dis.readLine();
           System.out.println(msg);
           ps.println("Data Received");
       }
       catch(IOException ex)
       {
           ex.printStackTrace();
       }
       try
       {
            ps.close();
            dis.close();
            s.close();
            myServerSocket.close();
       }
        catch(Exception ex)
        {
            ex.printStackTrace();
       }
  }
}
```

Simple Client/Server Console Example
Client Application

```
public class Client
{
   Socket mySocket;
   DataInputStream dis ;
   PrintStream ps;
   public static void main(String[] args)
   {
        new Client();
   }
   public Client()
   {
       try
       {
           mySocket = new Socket(“127.0.0.1”, 5005);
           dis = new DataInputStream(mySocket.getInputStream ());
           ps = new PrintStream(mySocket.getOutputStream ());
           ps.println(“Test Test");
           String replyMsg = dis.readLine();
           System.out.println(replyMsg);
```

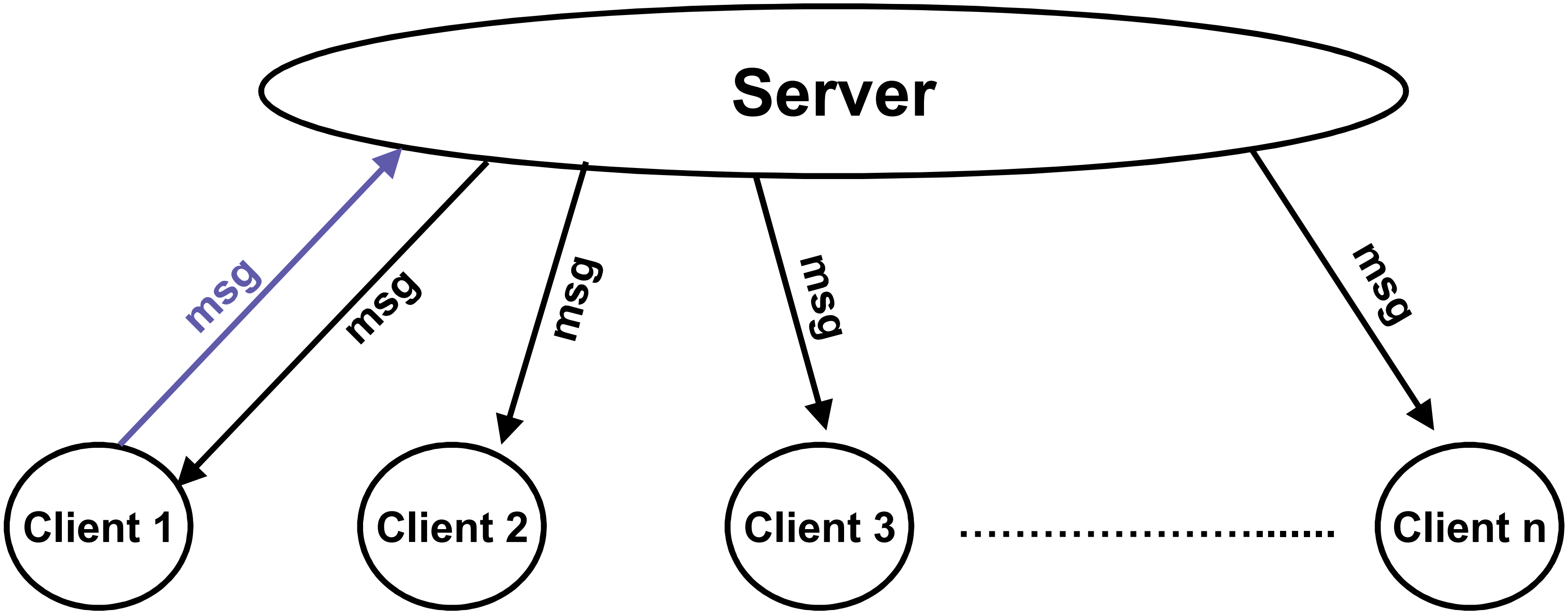
Simple Client/Server Console Example
Client Application

```
public class Client
{
   Socket mySocket;
   DataInputStream dis ;
   PrintStream ps;
   public static void main(String[] args)
   {
        new Client();
   }
   public Client()
   {
        try
        {
           mySocket = new Socket(InetAddress.getLocalHost(), 5005);
           dis = new DataInputStream(mySocket.getInputStream ());
           ps = new PrintStream(mySocket.getOutputStream ());
           ps.println(“Test Test");
           String replyMsg = dis.readLine();
           System.out.println(replyMsg);
```

Simple Client/Server Console Example
Client Application Cont'd

```
                    }
        catch(IOException ex)
        {
            ex.printStackTrace();
        }
        try
        {
            ps.close();
            dis.close();
            mySocket.close();
        }
        catch(Exception ex)
        {
            ex.printStackTrace();
        }

    }
}
```

Chat Room GUI Application Example



Chat Room GUI Application Cont'd

- The following code sample represents a server side application that holds a chat room that several clients can connect to and chat with each other. The server application is divided into two classes: ChatServer and ChatHandler:

```
public class ChatServer
{
  ServerSocket serverSocket;
  public ChatServer()
  {
     serverSocket = new ServerSocket(5005);
     while(true)
     {
       Socket s = serverSocket.accept();
       new ChatHandler(s);
     }
  }
  public static void main(String[] args)
  {
     new ChatServer();
  }
}
```

Chat Room GUI Application Cont'd

```
class ChatHandler extends Thread
{
  DataInputStream dis;
  PrintStream ps;
  static Vector<ChatHandler> clientsVector =
                                  new Vector<ChatHandler>();

  public ChatHandler(Socket cs)
  {
     dis = new DataInputStream(cs.getInputStream());
     ps = new PrintStream(cs.getOutputStream());
     clientsVector.add(this);
     start();
  }
```

Chat Room GUI Application Cont'd

```
  public void run()
  {
    while(true)
    {
      String str = dis.readLine();
      sendMessageToAll(str);
     }
  }

  void sendMessageToAll(String msg)
  {
    for(ChatHandler ch : clientsVector)
    {
      ch.ps.println(msg);
    }
  }
}
```

Chat Room GUI Application Cont'd

- General Guidelines for building the client side GUI application:
  - Construct GUI and Frame Layout.
  - Create Socket and Streams
  - Register the Send Button Listener
  - Create and start the Reader Thread

URL Class

- URL stands for Uniform Resource Locator.
- The general form of a URL is:

  ```
  protocol://host:port/fileRef#internalRef
  ```

- Commonly Used Constructor(s):
  - `URL(String url)`
  - `URL(String protocol, String host, int port, String file)`

URL Class Cont'd

- Commonly Used Method(s):
  - `String getProtocol()`
  - `String getHost()`
  - `int getPort()`
  - `String getFile()`
  - `InputStream openStream()`
  - `URLConnection openConnection()`

  Note: The returned URLConnection object provides some information about remote host (e.g. content encoding, expiration date, last modified date, …etc)

Lab Exercise

Another Event Handling Example

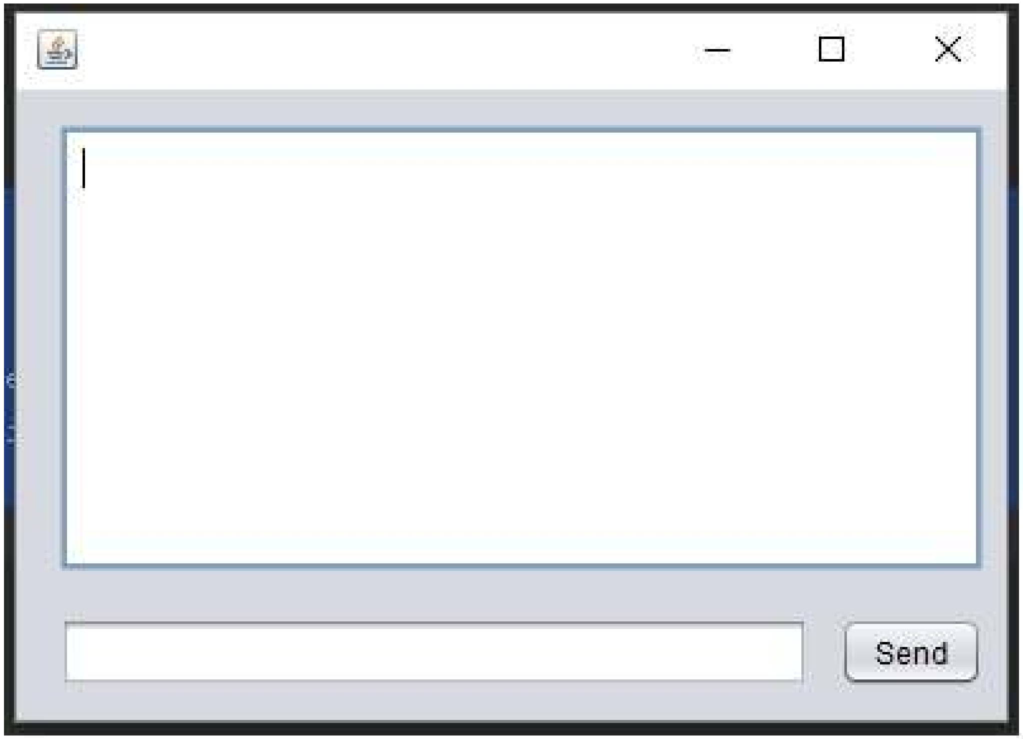
```
public class SampleUI extends JFrame {
    public SampleUI1(){
      this.setLayout(new FlowLayout());
      JTextArea ta=new JTextArea(20,50);
      JScrollPane scroll=new JScrollPane(ta);
      scroll.setViewportView(ta);
      JTextField tf=new JTextField(40);
      JButton okButton=new JButton("Send");
      okButton.addActionListener(new ActionListener(){
          public void actionPerformed(ActionEvent ae){
             // ta.append(tf.getText()+"\n");
                 ta.setText(“”);
           }
        });
      add(scroll);
      add(tf);
      add(okButton);
}
```

Another Event Handling Example

```
public static void main(String args[])
    {
        SampleUI ui=new SampleUI();
        ui.setSize(600, 400);
        ui.setResizable(false);
        ui.setVisible(true);
    }
}
```

Create A GUI Desktop Application

- Create a GUI Desktop Application which you can use as a chat Room client application interface .



Assignments

- Write the simple client/server application.
- Complete the GUI client side application of the Chat room Application.

Lesson 3

Java Database Connectivity (JDBC)

Lesson 3 Outline

- ❑ Introduction to JDBC API
- ❑ Types of JDBC Drivers
- ❑ Working with Database using Statement Object
- ❑ Working with Database using Prepared Statement Object
- ❑ Scrollable ResultSet

Introduction to JDBC API

- JDBC is a Java API for connecting to any DBMS and executing SQL statements.

  (Hides DB specific details from application)

- It consists of a set of classes and interfaces written in Java language.

- The JDBC interfaces are usually implemented by DBMS Vendors in order to provide their own vendor-specific JDBC Drivers.

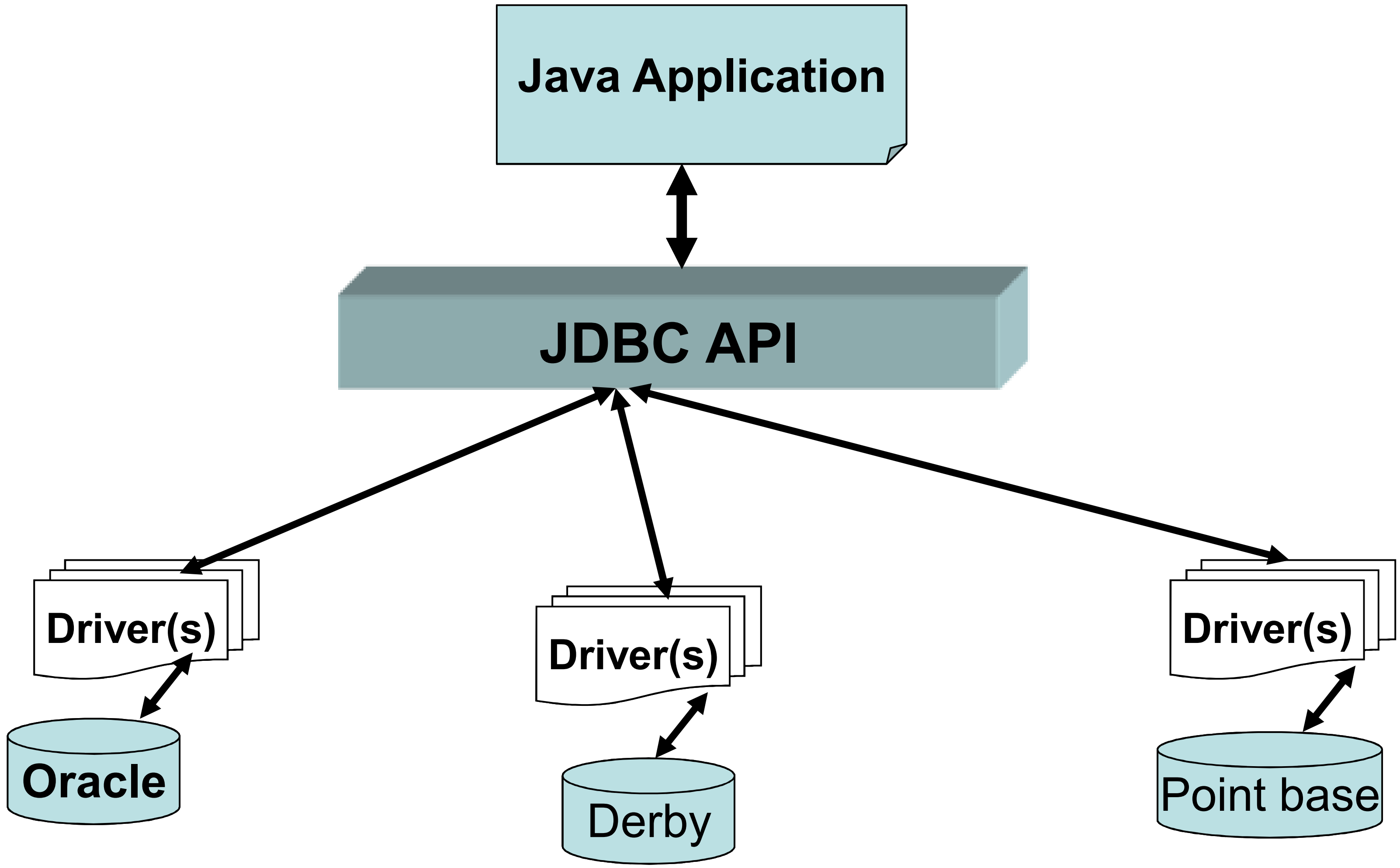
Introduction to JDBC API

JDBC Driver

- Database specific implementation of JDBC interfaces
  - Every database server has corresponding JDBC driver(s)

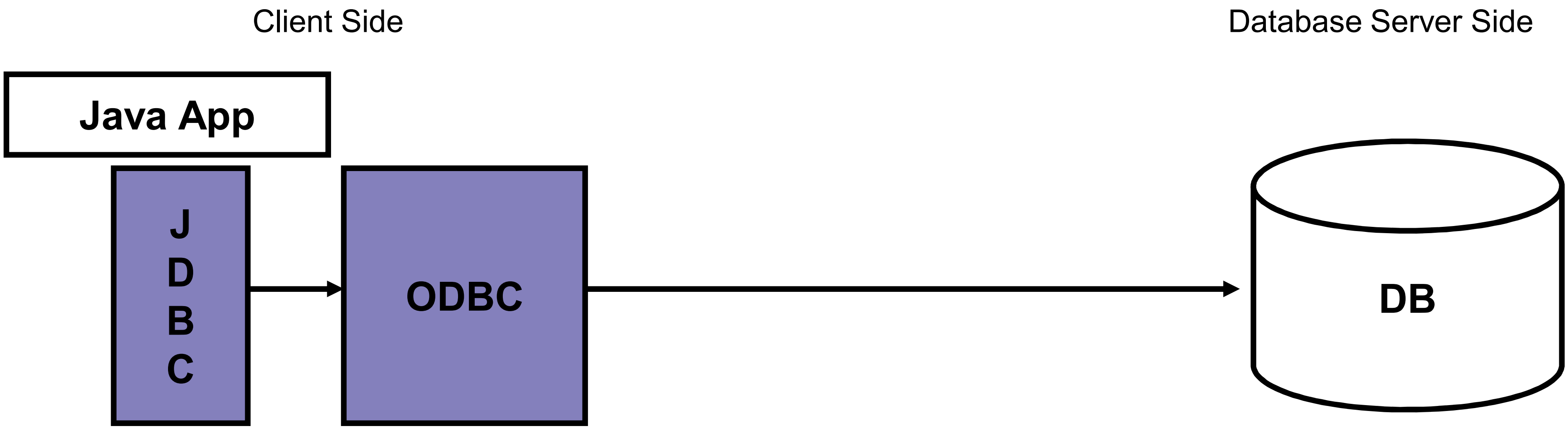
Introduction to JDBC API

- **JDBC Driver**



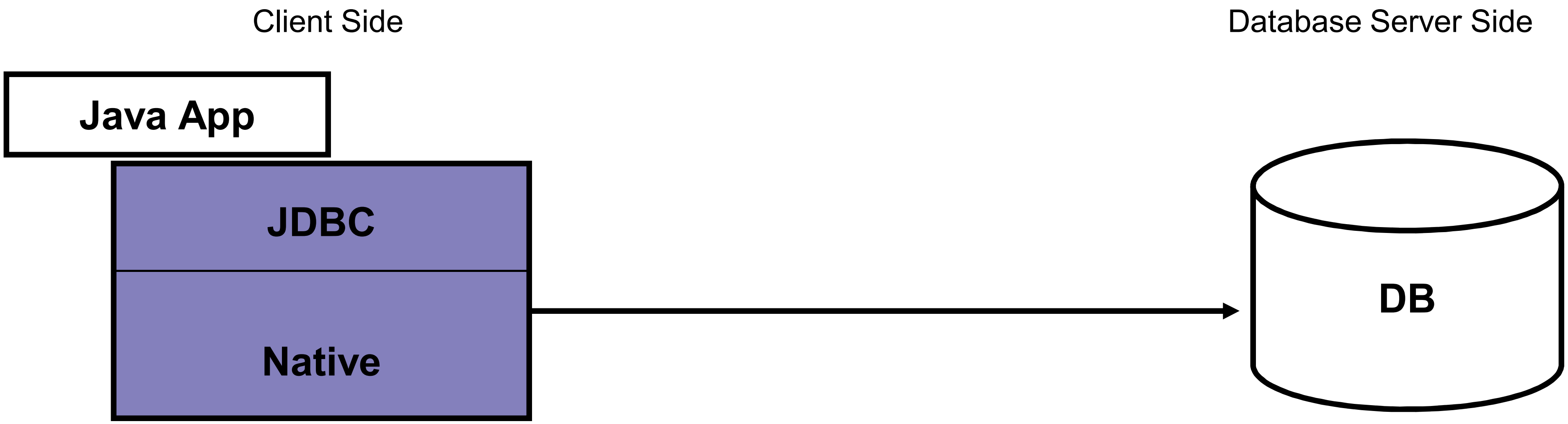
Types of JDBC Drivers

- **Type 1: JDBC-ODBC Bridge:**
  - Translates all JDBC calls into ODBC calls and sends them to the ODBC driver.
  - Classes are already available within JDK (before JDK 8).
  - Rather slow in performance due to the overhead of several layers.
  - Usually used for testing (at the early stages of building a system) or when there are no other alternative driver types.
  - Requires special ODBC configuration settings at the client side, and therefore, it is not appropriate for use with applets.



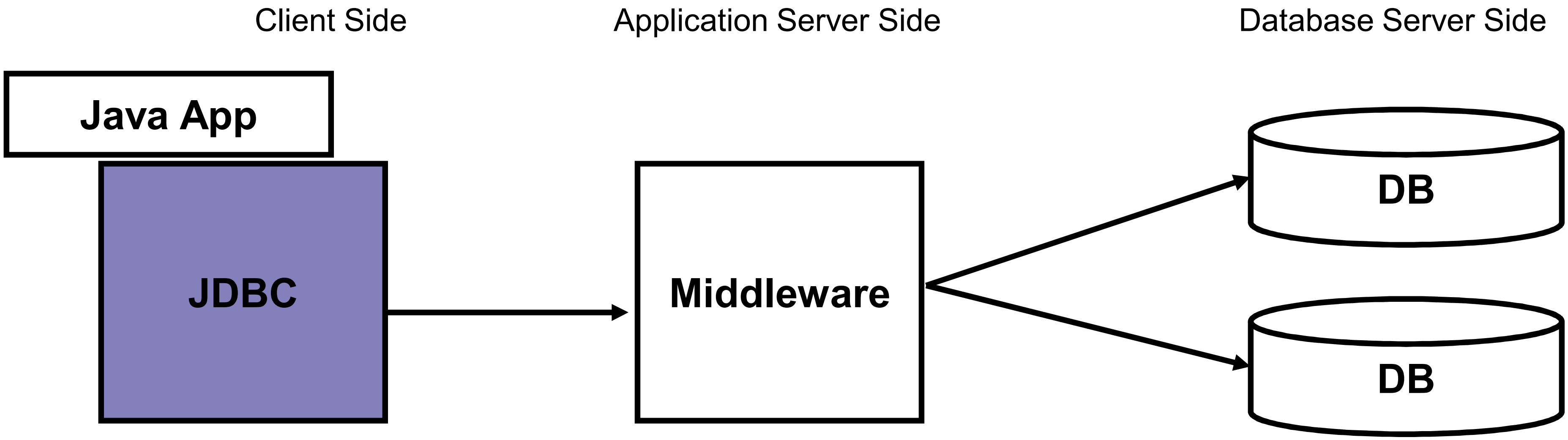
Types of JDBC Drivers cont'd

- **Type 2: Native API (partly Java) Driver:**
  - Converts JDBC calls into direct database-specific calls.
  - Classes are provided by DBMS Vendors (e.g. Oracle OCI Driver).
  - Fastest in performance (no translation overhead).
  - Native, therefore it is not Portable.
  - Requires setup at client side, and therefore, it is not appropriate for use with applets.



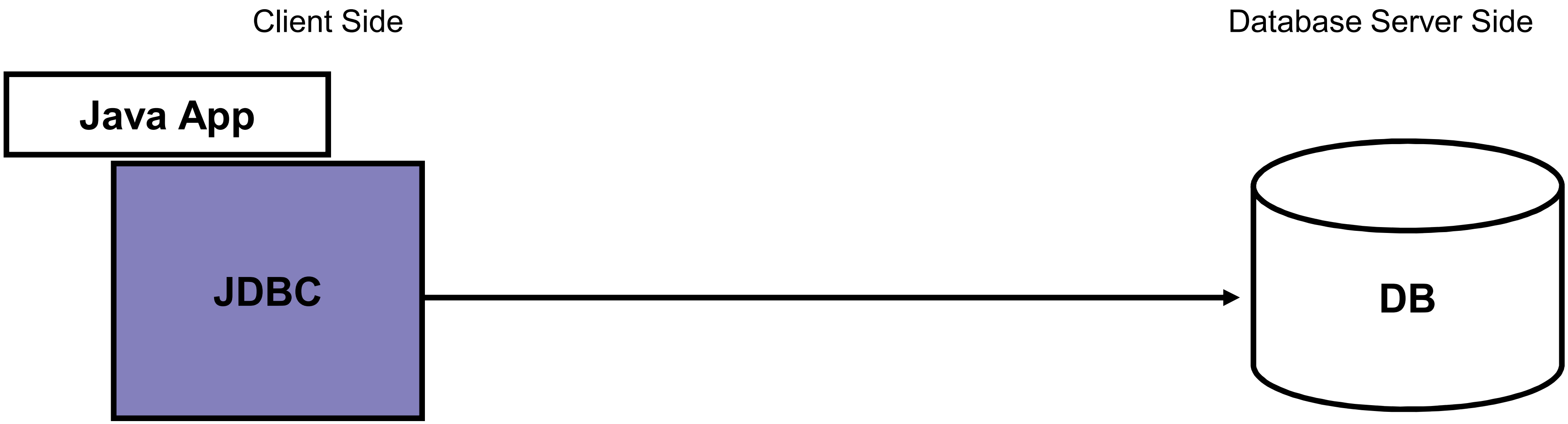
Types of JDBC Drivers cont'd

- **Type 3: Pure Java – Network Driver:**
  - Requests are passed through the network to the middleware server. The middleware then translates the request to the database. The middleware server can in turn use Type1, Type 2 or Type 4 drivers.
  - Classes are not vendor specific, and thus can be used to connect to any type of database, depending on the middleware's translation capabilities.
  - Classes are 100% written in Java Language, therefore, portable (may be includes in JAR files of the application).
  - Requires extra coding at the Application Server Side.
  - Rather slow performance because of translation layers.



Types of JDBC Drivers cont'd

- **Type 4: Pure Java – Native Protocol Driver:**
  - Communicates directly with the database.
  - Classes are provided by DBMS Vendors (e.g. Oracle thin Driver).
  - Rather fast in performance.
  - Classes are 100% written in Java Language, therefore, portable (may be includes in JAR files of the application).
  - This is the most commonly used type of drivers.



Introduction to JDBC API cont'd

- Steps for connecting and dealing with a Database:

1. Load (register) the driver.
2. Establish a connection.
3. Execute Queries.
4. Process Results (if retrieved).
5. Closing.

Working with Statement example 1

- The following code sample is for connecting to an Oracle Database Server using Oracle thin Driver (Type 4):

  Note: https://www.oracle.com/technetwork/database/application-development/jdbc/downloads/index.html

```
import java.sql.* ;
public class FirstDatabaseApp
{
  public FirstDatabaseApp()
  {
    try
    {
1    DriverManager.registerDriver(new
            oracle.jdbc.driver.OracleDriver());

2    Connection con = DriverManager.getConnection
            ("jdbc:oracle:thin:@127.0.0.1:1521:xe",
                                     "scott","tigger");
3     Statement stmt = con.createStatement()  ;
    String queryString = new String("select * from tab");
```

Working with Statement example 1

- The following code sample is for connecting to an MySQL Database Server using Type 4 Driver :

Note: Download from: https://dev.mysql.com/downloads/connector/j/8.0.html

```
import java.sql.* ;
public class FirstDatabaseApp
{
  public FirstDatabaseApp()
  {
    try
    {
1     DriverManager.registerDriver(new
                   com.mysql.cj.jdbc.Driver());

2     Connection con = DriverManager.getConnection
            ("jdbc:mysql://localhost:3306/sakila",
                                     “root",“passwd");
       Statement stmt = con.createStatement() ;
3    String queryString = new String("select * from tab");
```

Working with Database using Statement – Example cont‘d

```
3   Statement stmt = con.createStatement() ;
    String queryString = new String("select * from tab");
4   ResultSet rs = stmt.executeQuery(queryString) ;

   while(rs.next())
    {
      System.out.println(rs.getString(1));
    }

5   stmt.close();
    con.close();

   }
  catch(SQLException ex)
  {
     ex.printStackTrace();
  }
 }
}
```

Working with Database using Statement

```
    Statement stmt = con.createStatement() ;

   String queryString = new String("select * from tab
where name=‘Ali’ and age>20");

   String queryString = new String("select * from tab
where name=‘“+nameVar+”’ and age >”+ageVar);


    ResultSet rs = stmt.executeQuery(queryString) ;
```

Prepared Statement

- Using Prepared Statement instead of Statement has the following benefits:
  - Easy to use when we need to execute the same query many times but with different values for the where conditions.
  - We will not do tedious string concatenations for the query string.
  - Minimizes the query analyzing overhead.

Working with Database using Prepared Statement -Example

- The following code sample demonstrates the use of PreparedStatement:

```
PreparedStatement pst = con.prepareStatement("select *
   from Employee where gender = ? AND salary > ?");

pst.setString (1, "male");
pst.setFloat(2, 500);

ResultSet rs = pst.executeQuery() ;
// ResultSet executeQuery (String SQL);


// pst.executeUpdate();
// int executeUpdate (String SQL);
```

Statements Types

Interfaces	Recommended Use
Statement	Use the for general-purpose access to your database. Useful when you are using static SQL statements at runtime. The Statement interface cannot accept parameters.
PreparedStatement	Use the when you plan to use the SQL statements many times. The PreparedStatement interface accepts input parameters at runtime.
CallableStatement	Use the when you want to access the database stored procedures. The CallableStatement interface can also accept runtime input parameters.

ResultSet

- ResultSet object
  - The SQL statements that read data from a database query, return the data in a result set.
  - The java.sql.ResultSet interface represents the result set of a database query.
  - A ResultSet object maintains a cursor that points to the current row in the result set.
  - The term "result set" refers to the row and column data contained in a ResultSet object.
  - Characteristics of ResultSet [Scrollable, Updatable and Holdable]
  -

```
ResultSet rs = pst.executeQuery() ;
```

ResultSet

- JDBC provides the following connection methods to create statements with desired ResultSet:
  - createStatement (int RSType, int RSConcurrency, int resultSetHoldability); … Statement
  - prepareStatement (String SQL, int RSType, int RSConcurrency, int resultSetHoldability); … PreparedStatement
  - prepareCall (String sql, int RSType, int RSConcurrency, int resultSetHoldability);… CallableStatement

ResultSet

- Type of ResultSet :

Type	**Description**
ResultSet.TYPE_FORWARD_ONLY *	The cursor can only move forward in the result set.
ResultSet.TYPE_SCROLL_INSENSITIVE	The cursor can scroll forward and backward, and the result set is ***not sensitive to changes made by others to the database*** that occur after the result set was created.
ResultSet.TYPE_SCROLL_SENSITIVE	The cursor can scroll forward and backward, and the result set is ***sensitive to changes made by others to the database*** that occur after the result set was created.

ResultSet

- Concurrency of ResultSet:

Concurrency	Description
ResultSet.CONCUR_READ_ONLY *	Creates a read-only result set.
ResultSet.CONCUR_UPDATABLE	Creates an updateable result set

- Holdability of ResultSet:

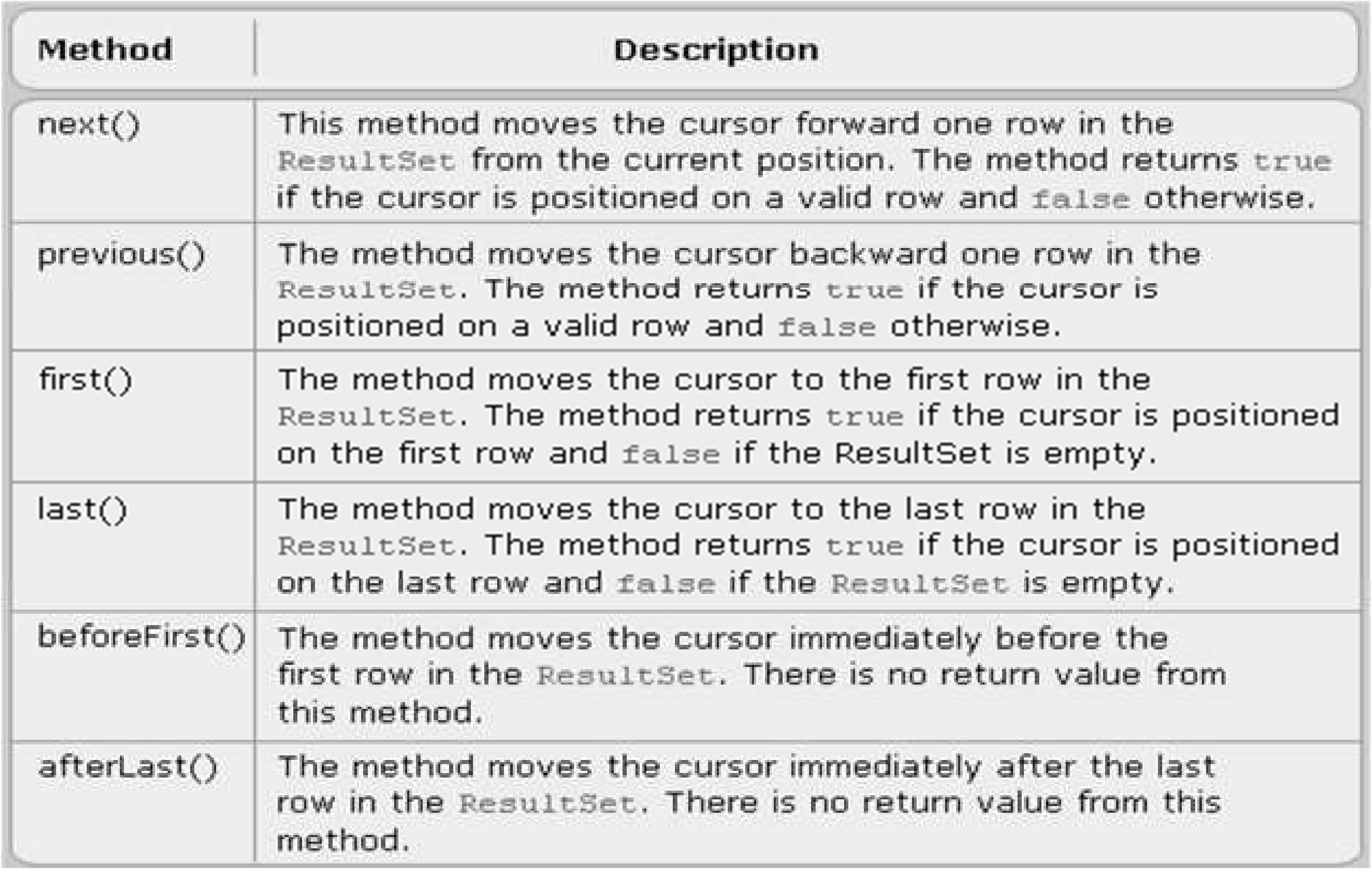
Holdability	Description
Result Set. HOLD_CURSORS_OVER_COMMIT *	ResultSet object will remain open when the current transaction is committed.
ResultSet. CLOSE_CURSORS_AT_COMMIT	ResultSet object will be closed when the current transaction is committed

ResultSet

- The methods of the ResultSet interface can be broken down into three categories :
  - **Navigational methods**: Used to move the cursor around.
  - **Get methods**: Used to view the data in the columns of the current row being pointed by the cursor.
  - **Update methods**: Used to update the data in the columns of the current row. The updates can then be updated in the underlying database as well.

ResultSet

Navigational methods:



Method	Description
next()	This method moves the cursor forward one row in the `ResultSet` from the current position. The method returns `true` if the cursor is positioned on a valid row and `false` otherwise.
previous()	The method moves the cursor backward one row in the `ResultSet`. The method returns `true` if the cursor is positioned on a valid row and `false` otherwise.
first()	The method moves the cursor to the first row in the `ResultSet`. The method returns `true` if the cursor is positioned on the first row and `false` if the ResultSet is empty.
last()	The method moves the cursor to the last row in the `ResultSet`. The method returns `true` if the cursor is positioned on the last row and `false` if the `ResultSet` is empty.
beforeFirst()	The method moves the cursor immediately before the first row in the `ResultSet`. There is no return value from this method.
afterLast()	The method moves the cursor immediately after the last row in the `ResultSet`. There is no return value from this method.

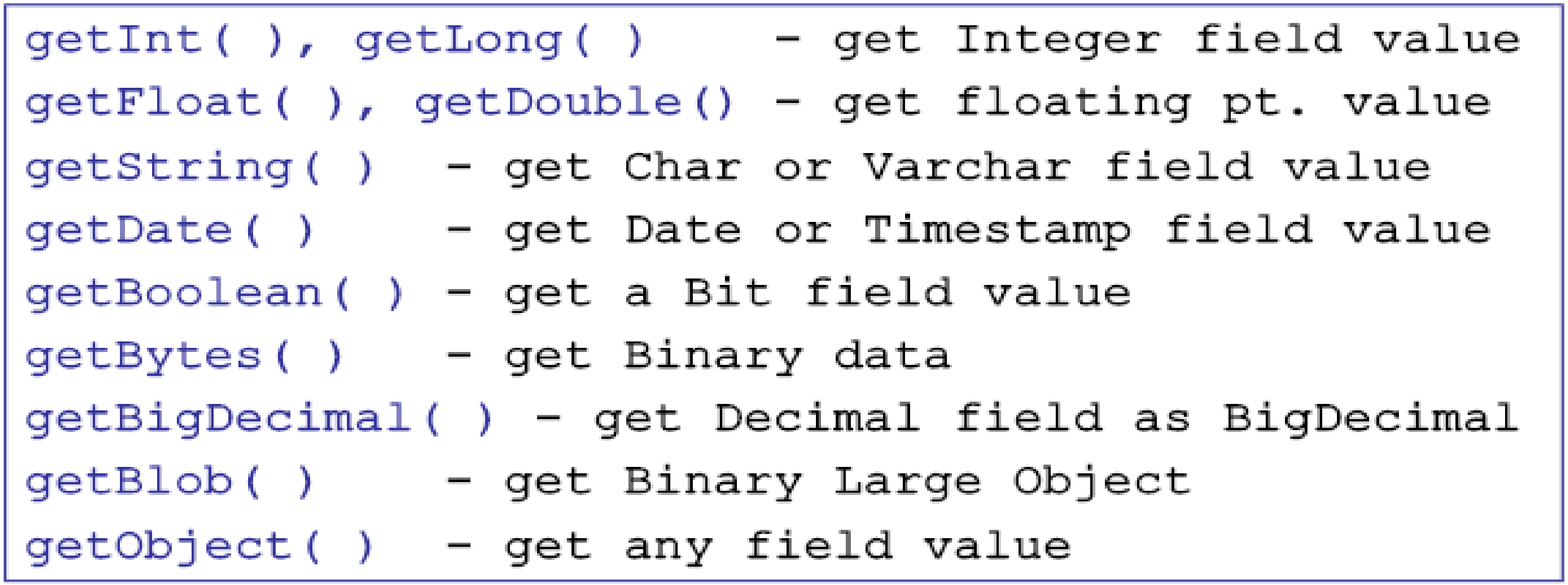
ResultSet

Get methods:

if the column you are interested in viewing contains an int, you need to use one of the :

getInt(2) : by col. Index.

getInt(“ EMPID“) : by col. Name



```
getInt( ), getLong( )     - get Integer field value
getFloat( ), getDouble() - get floating pt. value
getString( )   - get Char or Varchar field value
getDate( )     - get Date or Timestamp field value
getBoolean( ) - get a Bit field value
getBytes( )    - get Binary data
getBigDecimal( ) - get Decimal field as BigDecimal
getBlob( )     - get Binary Large Object
getObject( )   - get any field value
```

Batch updates

- A set of *multiple update statements* that is submitted to the database for processing as *a batch*

- Statement,
- PreparedStatement and
- CallableStatement can be used to submit batch updates

Implementing Batch Update using “Statement” interface

1. Disable auto-commit mode
2. Create a Statement instance
3. Add SQL commands to the batch
4. Execute the batch commands
5. Commit the changes to the database

Implementing Batch Update using “PreparedStatement” Interface

```
// 1. Turn off auto-commit
      con.setAutoCommit( false );

//2. Creating an instance of prepared Statement
 PreparedStatement  pst = con.preparedStatement ( “ INSERT INTO EMPLOYEE VALUES ( ? , ? )” );

// 3.Adding the calling statement batches
     pst.setInt(1,500);
     pst.setString(2 ,” Max “);

     pst.addBatch();

     pst.setInt(1,400);
     pst.setString(2 ,” Joe “);

     pst.addBatch();

// 4.Submit the batch for execution
   Int[] updateCounts= pst.executeBatch();

// 5. Commit
   con.commit();
```

Lesson 4

Java Collections

Collection Framework

- The Java collections framework is a set of generic types that you use to create **collection classes** that support various ways for you to store and manage objects of any kind in memory.

- **A collection class** is simply a class that organizes a set of objects of a given type in a particular way, such as in a linked list or a pushdown stack.

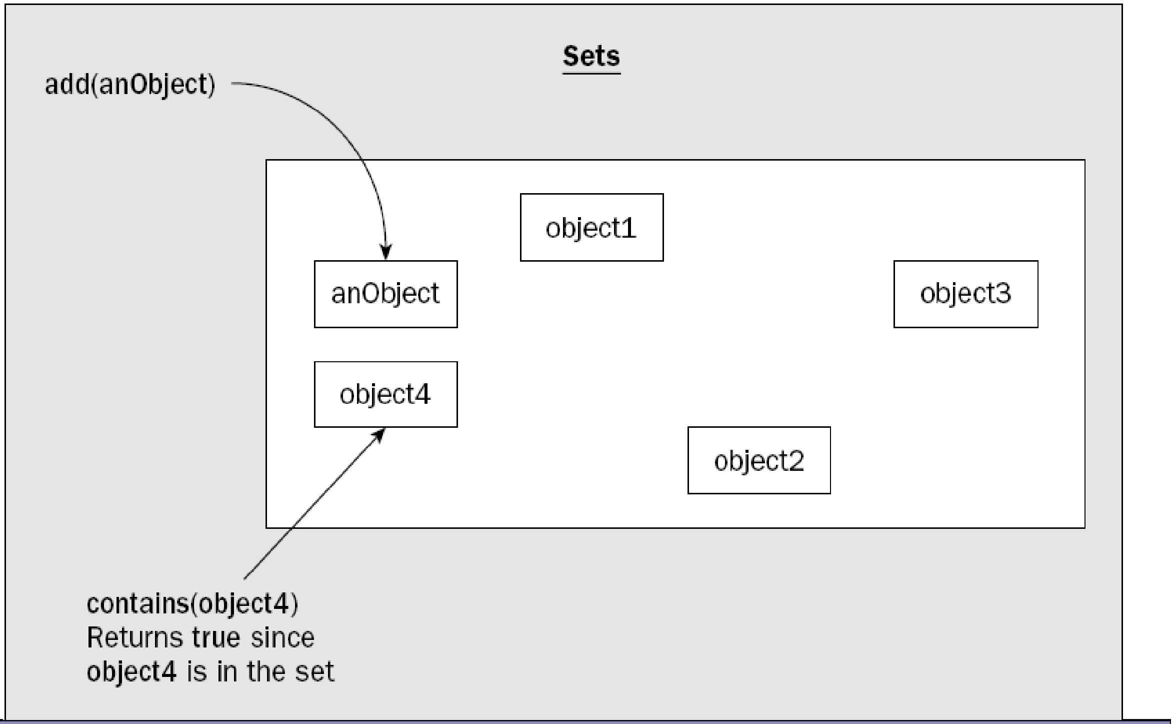
Collection Framework cont'd

- There are three main types of collections that organize objects in different ways:
  - **sets,**
  - **sequences,**
  - and **maps.**

- In Java, collections store references only—the objects themselves are external to the collection.

Collection Framework - Sets

- A **set is probably the simplest kind of collection you can have.**
- Here, the objects are not ordered in any particular way at all, and objects are simply added to the set without any control over where they go.



Collection Framework – Sets cont'd

- You can add objects to a set and iterate over all the objects in a set.

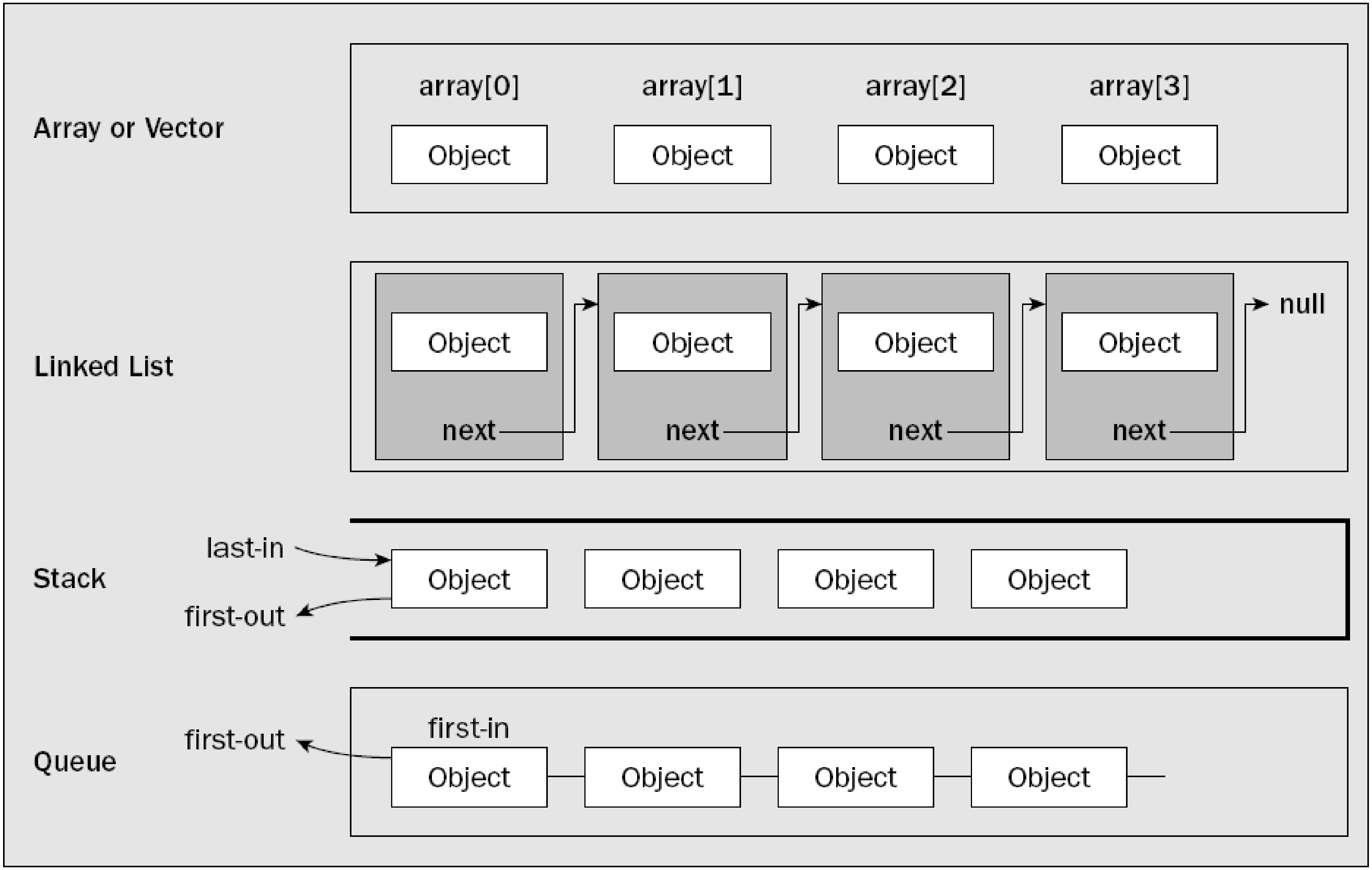
- You can also check whether a given object is a member of the set or not.

- For this reason you cannot have duplicate objects in a set—each object in the set must be unique.

Collection Framework - Sequences

- A primary characteristic of a sequence is that the objects are stored in a linear fashion, not necessarily in any particular order, but organized in an arbitrary fixed sequence with a beginning and an end.

Collection Framework – Sequences cont'd



Collection Framework – Sequences cont’d

- Because a sequence is linear, you will be able to add a new object only at the beginning or at the end, or insert a new object following a given object position in the sequence.

- In the Java collections framework, types that define sequences are subdivided into two subgroups, **lists** and **queues.**

- **Vectors, linked lists, and stacks are all lists.**

Collection Framework - Maps

- A **map is rather different from a set or a sequence collection because each entry involves a pair of objects.**

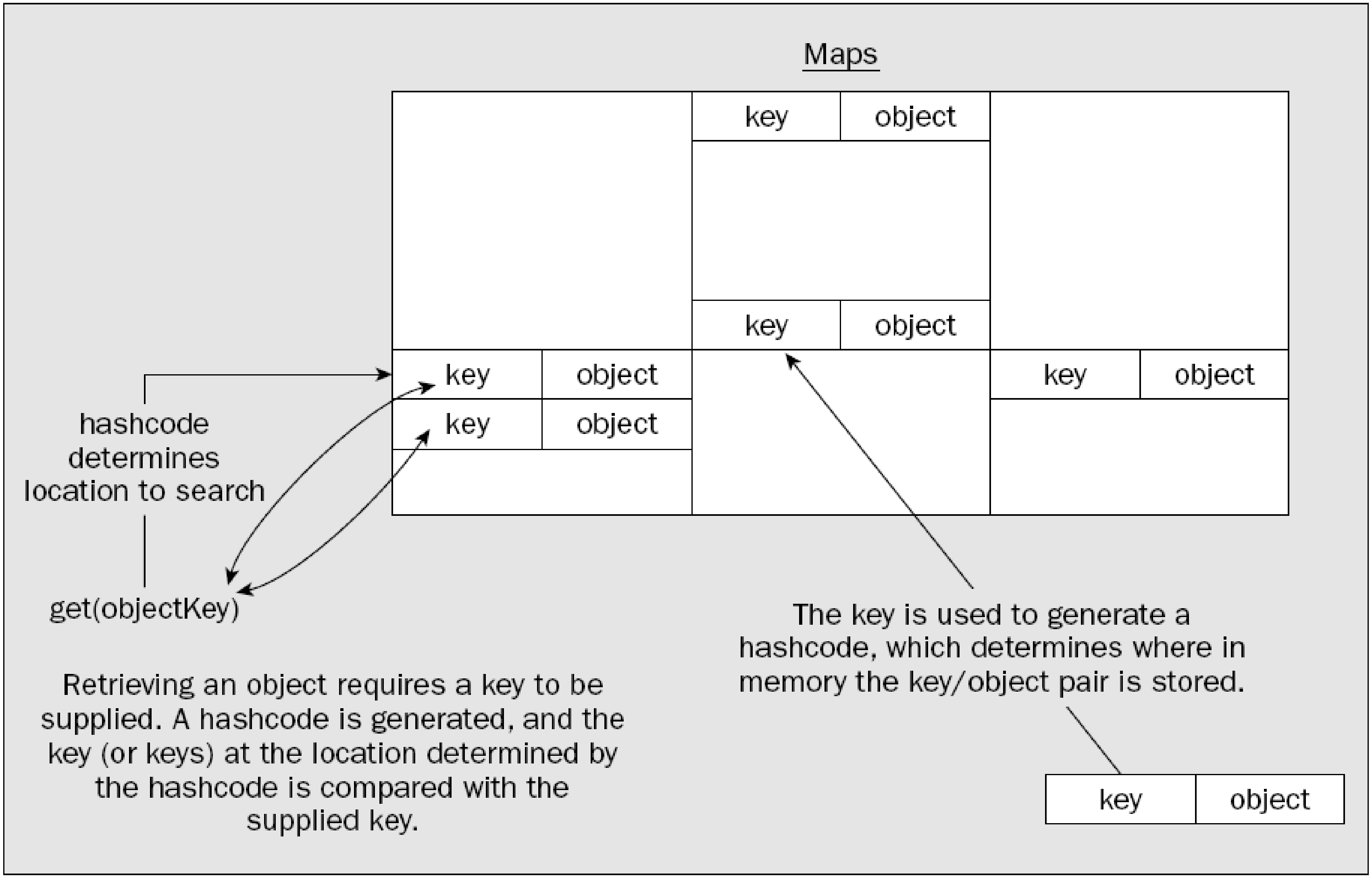
- A map is also referred to sometimes as a **dictionary because of the way it works.**

- **Each object that is stored** in a map has an associated **key object, and the object and its key are stored together as a pair.**

Collection Framework – Maps cont’d

- **The key** determines where the object is stored in the map, and when you want to retrieve an object, you must supply the appropriate key.

Collection Framework – Maps cont'd



Retrieving an object requires a key to be supplied. A hashcode is generated, and the key (or keys) at the location determined by the hashcode is compared with the supplied key.

The key is used to generate a hashcode, which determines where in memory the key/object pair is stored.

Iterator

- An Iterator is an object that you can use once to retrieve all the objects in a collection one by one.

- Using an **Iterator is a standard mechanism** for accessing each of the elements in a collection.

- Any collection object that represents a set or a sequence can create an object of type Iterator that behaves as an iterator.

- Types representing maps do not have methods for creating iterators.

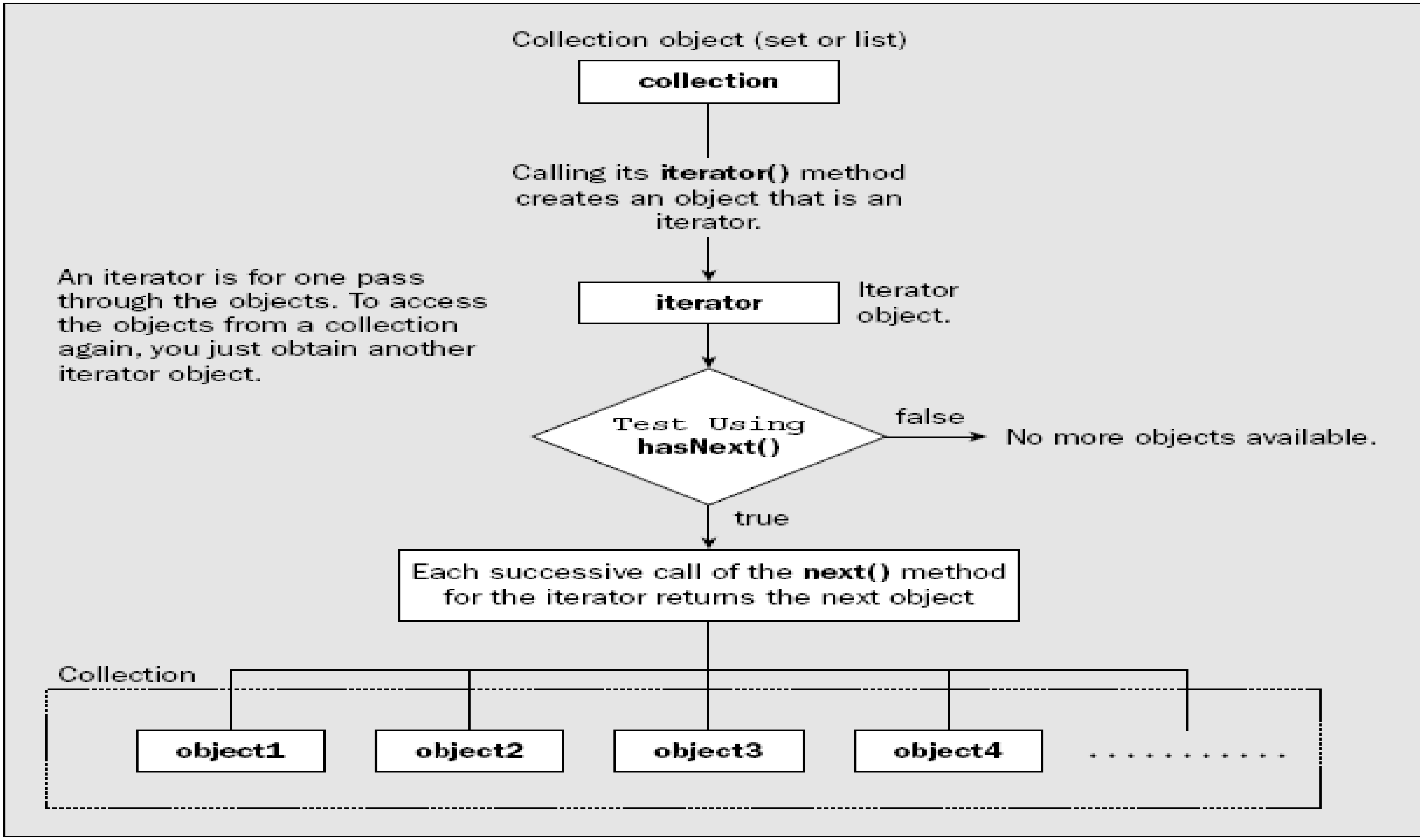
Iterator cont’d

- A map class provides methods to enable the keys or objects, or indeed the key/object pairs, to be viewed as a set, so you can then obtain an iterator to iterate over the objects in the set view of the map.

- An Iterator object encapsulates references to all the objects in the original collection in some sequence, and they can be accessed one by one using the Iterator interface methods

Iterator cont'd

- The basic mechanism for using an iterator is illustrated in the figure below:

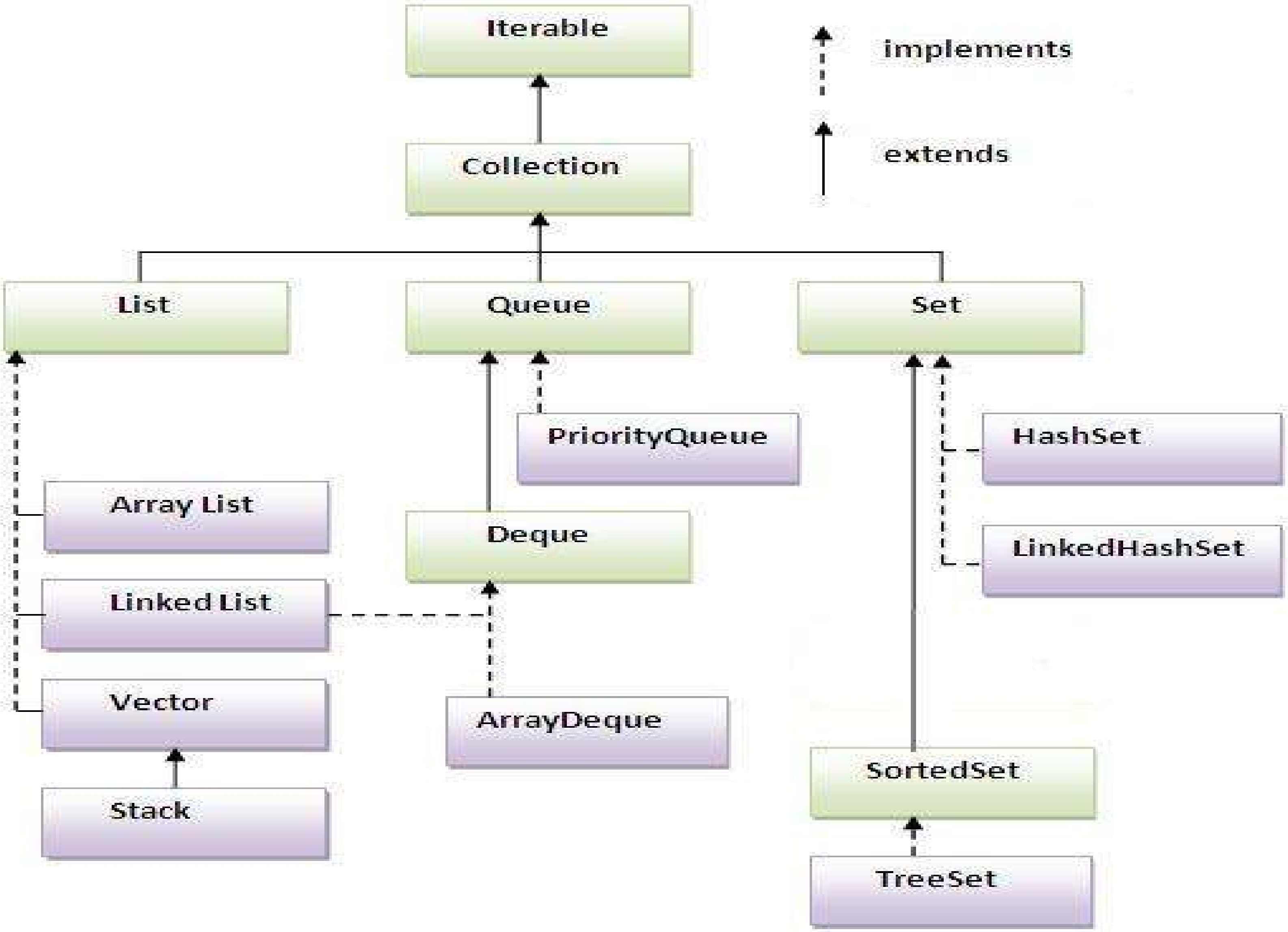

Collection object (set or list)
collection
Calling its iterator() method creates an object that is an iterator.
An iterator is for one pass through the objects. To access the objects from a collection again, you just obtain another iterator object.
iterator
Iterator object.
Test Using hasNext()
false
No more objects available.
true
Each successive call of the next() method for the iterator returns the next object
Collection
object1
object2
object3
object4
. . . .

Iterator cont’d

- Methods of Iterator interface :

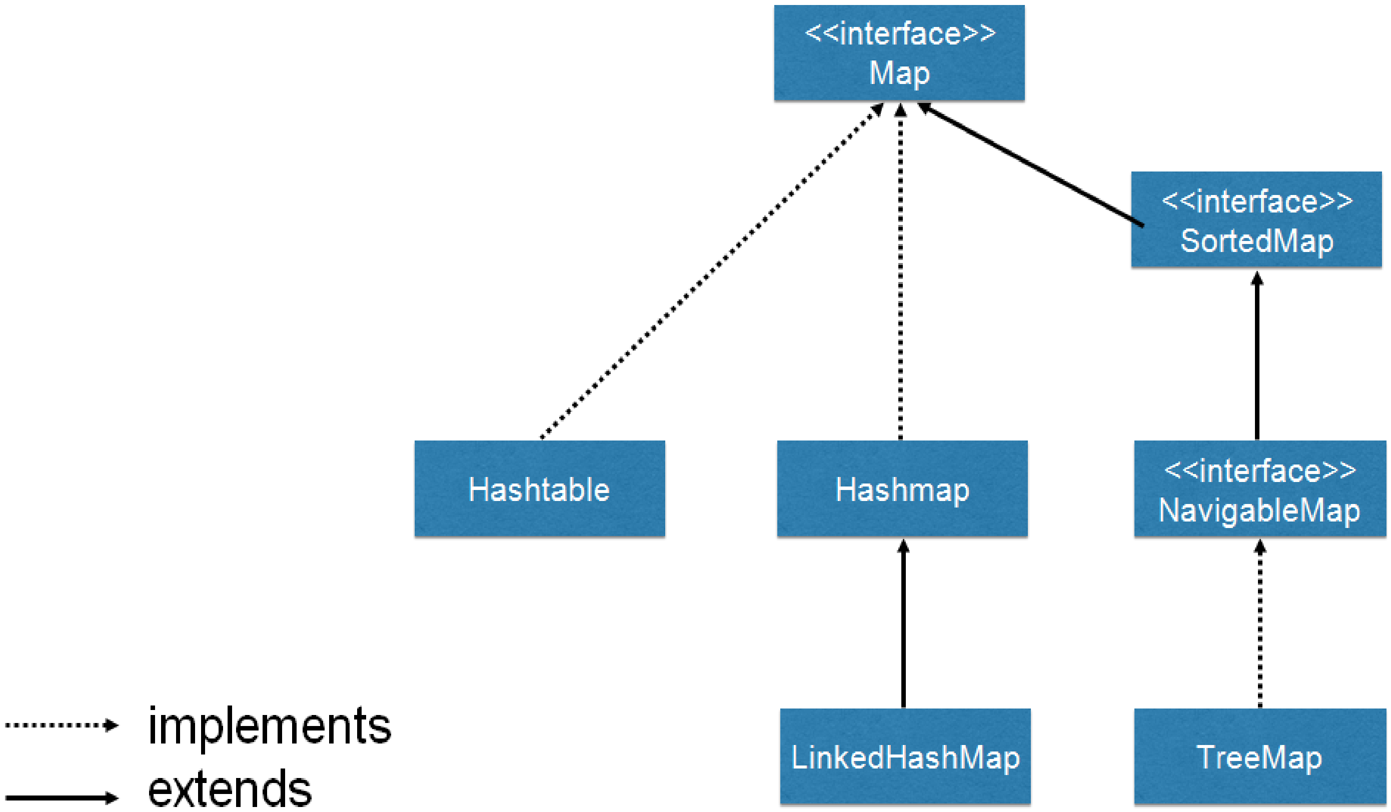
1. **public boolean hasNext()** it returns true if iterator has more elements.
2. **public object next()** it returns the element and moves the cursor pointer to the next element.
3. **public void remove()** it removes the last elements returned by the iterator.

Collection Classes [java.util]



Collection Classes cont'd

Map Interface



Generics

- Generics allow you to abstract over types. The most common examples are container types, such as those in the Collections hierarchy.

- Here is a typical usage of that sort:

```
List myIntList = new LinkedList(); // 1
myIntList.add(new Integer(0)); // 2
Integer x = (Integer) myIntList.iterator().next();// 3
```

- The Cast in line three is annoying, It also introduces the possibility of a run time error, since the programmer may be mistaken.

Generics cont’d

- What if programmers could actually express their intent, and mark a list as being restricted to contain a particular data type? This is the core idea behind generics. Here is a version of the program fragment given above using generics:

```
List<Integer> myIntList = new LinkedList<Integer>(); //1`
myIntList.add(new Integer(0)); //2'
Integer x = myIntList.iterator().next(); //3'
```

Generics cont’d

- Notice the type declaration for the variable myIntList. It specifies that this is not just an arbitrary List, but a List of Integer, written List<Integer>.
- The compiler can now check the type correctness of the program at compile-time.
- In contrast, the cast tells us something the programmer thinks is true at a single point in the code.
- The net effect, especially in large programs, is improved readability and robustness

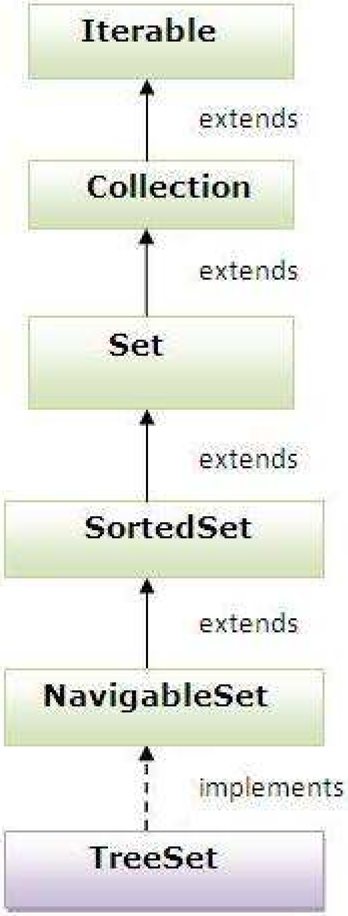
The Diamond "Java 7"

- In Java SE 7,
  - You can replace the type arguments required to invoke the constructor of a generic class with an empty set of type arguments (<>).
  - This pair of angle brackets, <>, is informally called *the diamond*.
  - For example:

```
Box<Integer> integerBox = new Box<>();
```

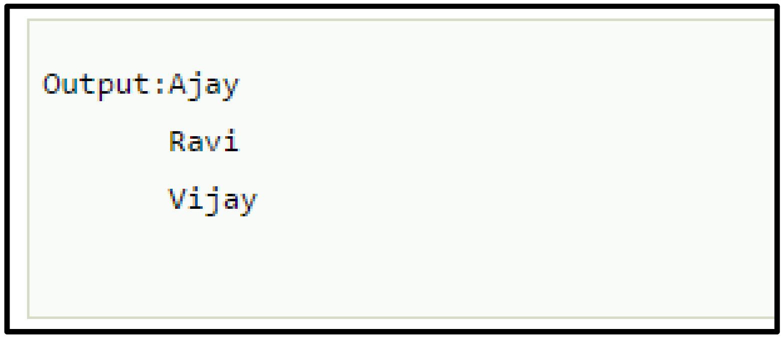
Example

- Java TreeSet class
  - contains unique elements only
  - maintains ascending order.



Example

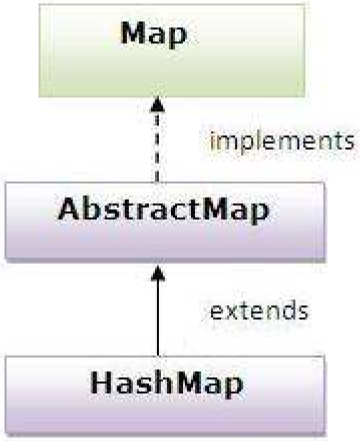
```
import java.util.*;
class TestCollection {
          public static void main(String args[]){

            TreeSet<String> al = new TreeSet<String>();
            al.add(“Ravi”);
            al.add(“Vijai”);
            al.add(“Ravi”);
            al.add(“Ajay”);

            Iterator <String>   itr = al.iterator();

            while(itr.hasNext()){
                    System.out.println(itr.next());
             }
          }
}
```



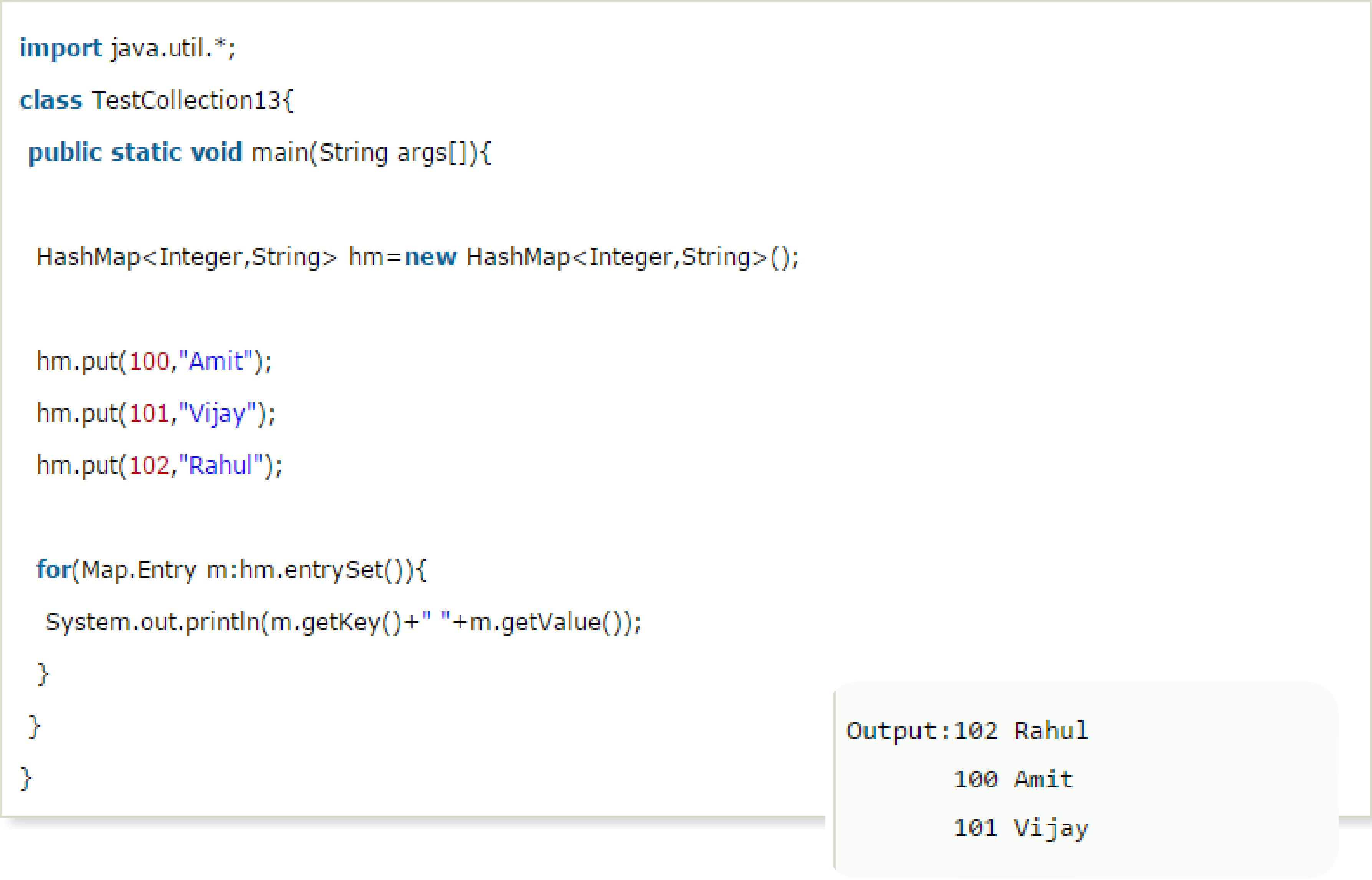
```
Output:Ajay
       Ravi
       Vijay
```

Example

- Java HashMap class
  - A HashMap contains values based on the key.
  - It implements the Map interface and extends AbstractMap class.
  - It may have one null key and multiple null values.
  - It contains only unique elements.
  - It maintains no order.



Example



```
import java.util.*;
class TestCollection13{
 public static void main(String args[]){

  HashMap<Integer,String> hm=new HashMap<Integer,String>();

  hm.put(100,"Amit");
  hm.put(101,"Vijay");
  hm.put(102,"Rahul");

  for(Map.Entry m:hm.entrySet()){
   System.out.println(m.getKey()+" "+m.getValue());
  }
 }
}
```

```
Output:102 Rahul
       100 Amit
       101 Vijay
```

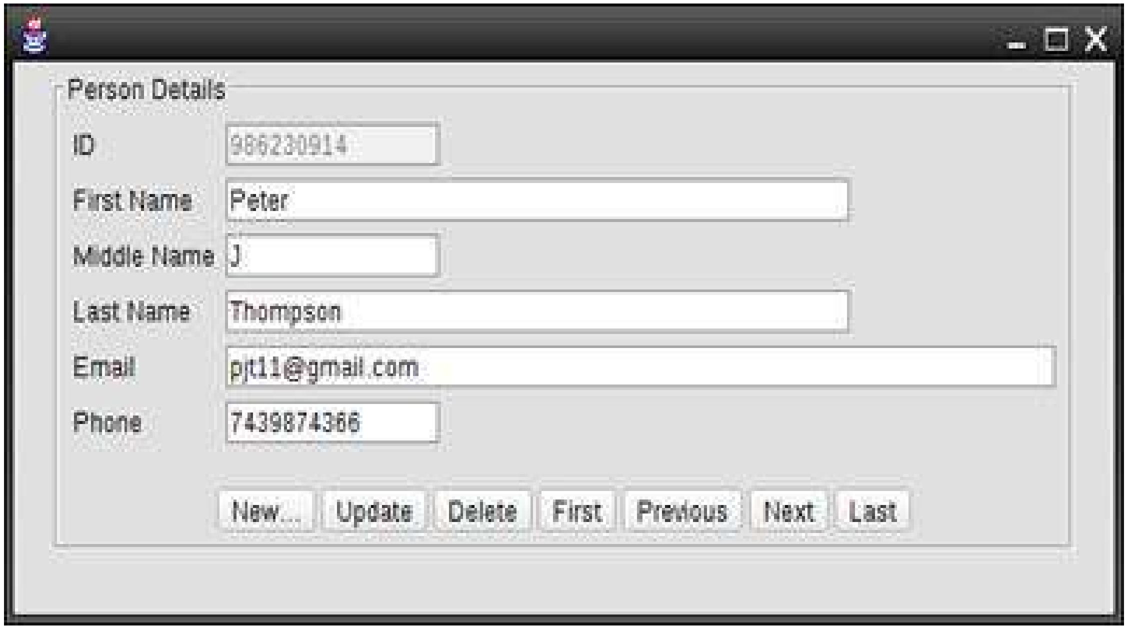
Lab Exercise

1. Connecting to a Database Server

- Write an Application that connects to a database server (e.g. Oracle, DB2, MySQL, SQL Server…etc) using a vendor-specific JDBC Driver (Type 4).
- Execute different types of queries that manipulate the data in the tables (retrieve, insert, edit, delete).

1. Connecting to a Database Server

- Write an Application that connects to a database server (e.g. Oracle, DB2, MySQL, SQL Server…etc) using a vendor-specific JDBC Driver (Type 4).
- Execute different types of queries that manipulate the data in the tables (retrieve, insert, edit, delete).
- Try the Commit and Rollback features



Person Details

ID 986230914

First Name Peter

Middle Name J

Last Name Thompson

Email pjt11@gmail.com

Phone 7439874366

Lesson 5

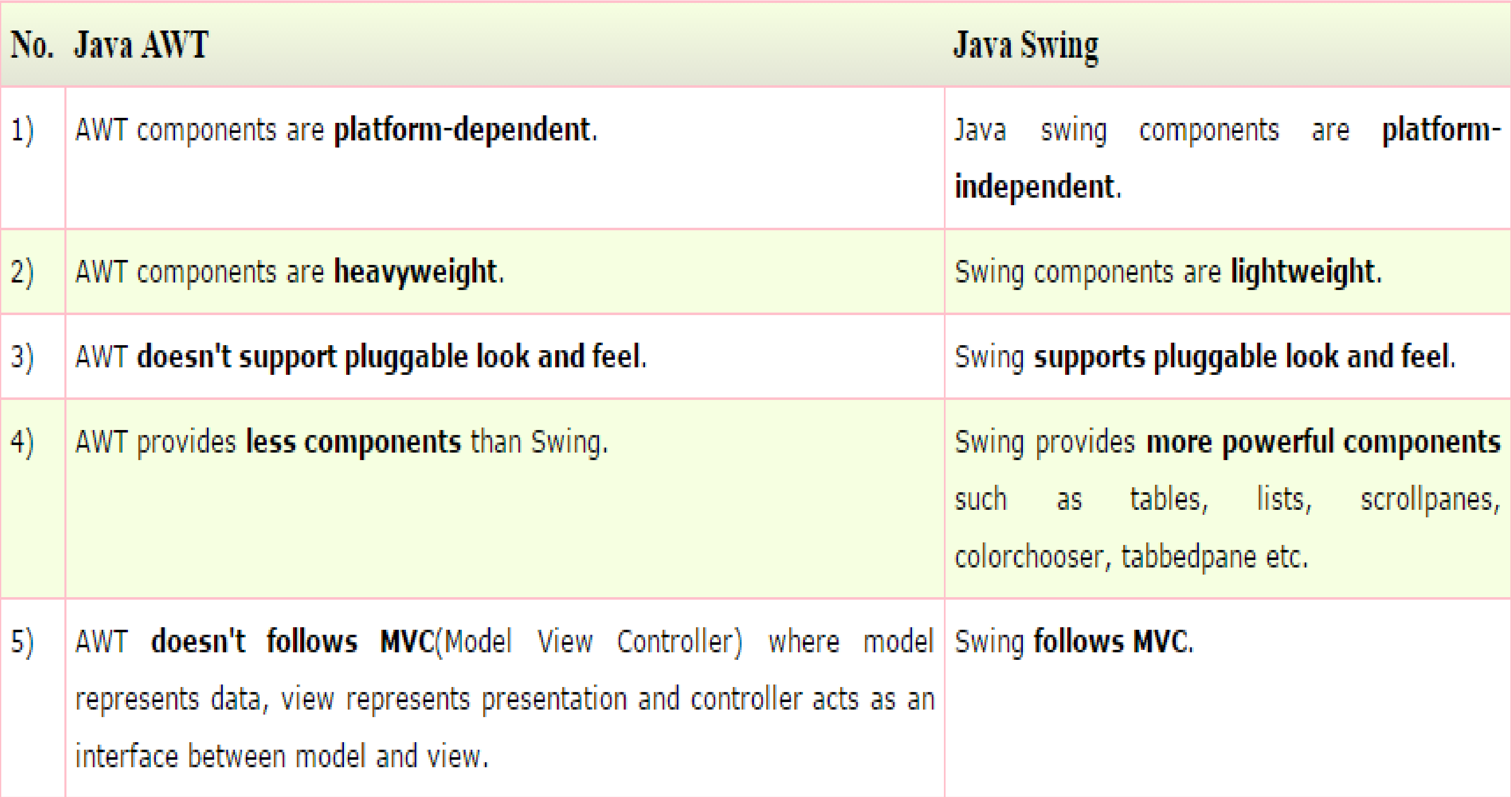
Introduction to JavaFX 8

Java UI Components History

- **AWT** Abstract Window Toolkit :
  - is an API to develop GUI or window-based application in java.
  - AWT components are platform-dependent [components are displayed according to the view of operating system].
  - AWT is heavyweight [its components uses the resources of system].
- **Swing**:
  - It is built on the top of AWT.
  - Java Swing provides platform-independent and lightweight components.

Java UI Components History

- **AWT** VS. **Swing [** Desktop Application**]**:



No.	Java AWT	Java Swing
1)	AWT components are **platform-dependent**.	Java swing components are **platform-independent**.
2)	AWT components are **heavyweight**.	Swing components are **lightweight**.
3)	AWT **doesn't support pluggable look and feel**.	Swing **supports pluggable look and feel**.
4)	AWT provides **less components** than Swing.	Swing provides **more powerful components** such as tables, lists, scrollpanes, colorchooser, tabbedpane etc.
5)	AWT **doesn't follows MVC**(Model View Controller) where model represents data, view represents presentation and controller acts as an interface between model and view.	Swing **follows MVC**.

JavaFX History

- F3 (Form Follows Function) by Chris Oliver.
- F3 is a declarative scripting language with good support of IDE, and compile time error reporting unlike javascript.
- Chris Oliver became a Sun employee through Sun acquisition of See Beyond Technology Corporation in September 2005.
- At JavaOne 2007 , Its name was changed to JavaFX [Open Source].
- The first version of JavaFX Script was an interpreteduage, and was considered a prototype of the compiled JavaFX Script language.

JavaFX History

- At JavaOne 2009, the JavaFX SDK 1.2 was released.
- At JavaOne 2010, the JavaFX SDK 2.0 was announced.
- **JavaFX 2.0** road-map :
  - Deprecating JavaFX script.
  - Porting all JavaFX scripting features into JavaFX 2.0 APIs.
  - Providing web component for embedding HTML and JavaScript into JavaFX code.
  - Enable JavaFX interoperability with swing.
- At JavaOne 2011, the JavaFX SDK 2.0 was released

What is JavaFX ?

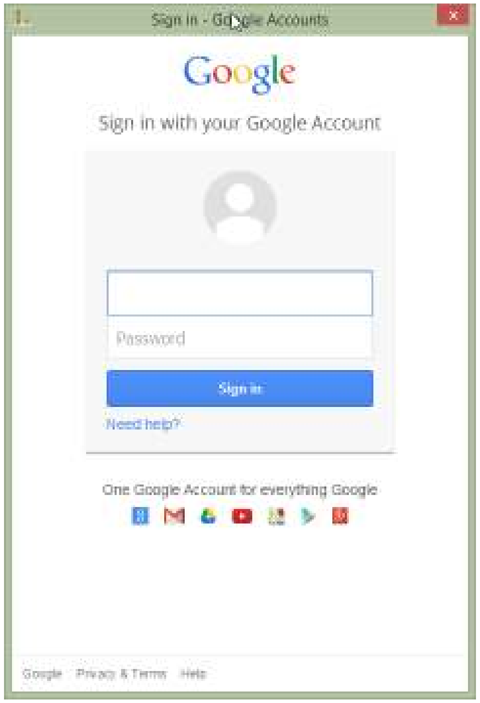
- ❑ **JavaFX** is a next generation graphical user interface toolkit.
- ❑ It is intended to replace java Swing as the standard GUI library for JavaSE.
- ❑ It is a set of graphics and media packages that enables developers to design, create, test, debug, and deploy rich client applications that operate consistently across diverse platforms.
- ❑ **JavaFX** has included a feature of customized style using Cascading Style Sheets (CSS) style.

JavaFX Features

- ❑ It is easy to learn because it is very similar to Java **swing**.
- ❑ Swing application can be updated with JavaFX features
- ❑ A new language is added to JavaFX called **FXML**, which is XML that is used only to define the interface of an application, So that it completely separated from the logic of the code.
- ❑ The library of JavaFX is created using **Java native code** or Java API.
- ❑ JavaFX is also **platform independent** so that it can run any platform using the JVM.

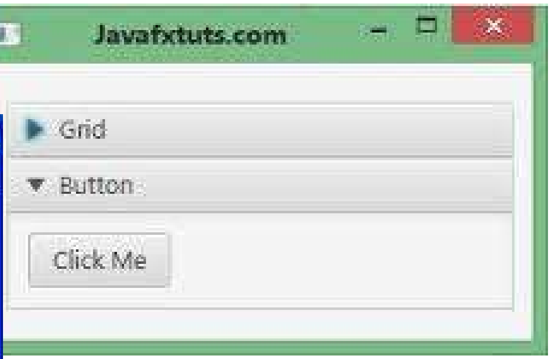
JavaFX Features

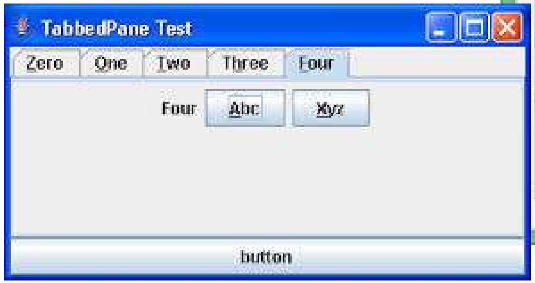
- ❑ **JavaFX** supports:
  - ❑ Webview: A web component can be embedded inside the JavaFX application to view web pages.
  - ❑ Many extra features like date-picker, accordion pane, tabbed pane and pie-chart.
  - ❑ Animations, 2D and 3D graphics .
  - ❑ Powerful way of designing using CSS.
  - ❑ Multi-touch operations.



Sign in with your Google Account

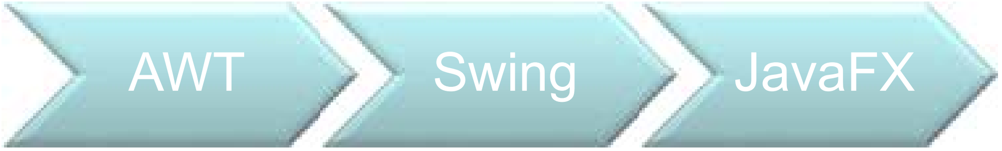
One Google Account for everything Google





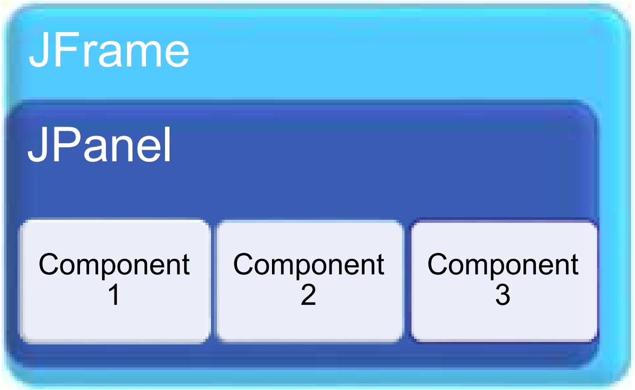
Java UI Components

- **AWT**
  - Platform Specific.



- **Swing**
  - Only for Desktop Applications
- **JavaFX**
  - Desktop, websites, Handheld Devices friendly.

Java Swing Vs JavaFX

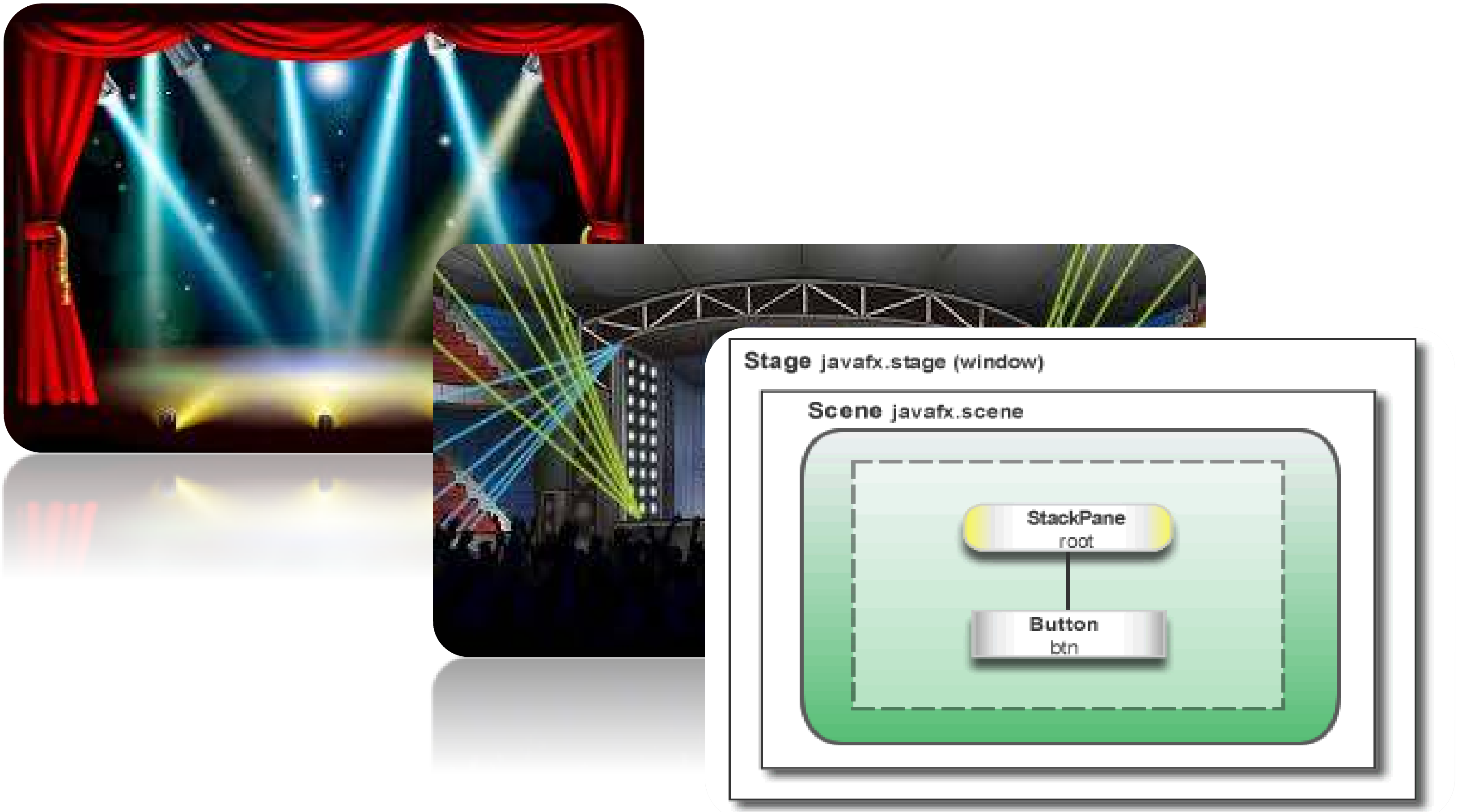


In Swing,

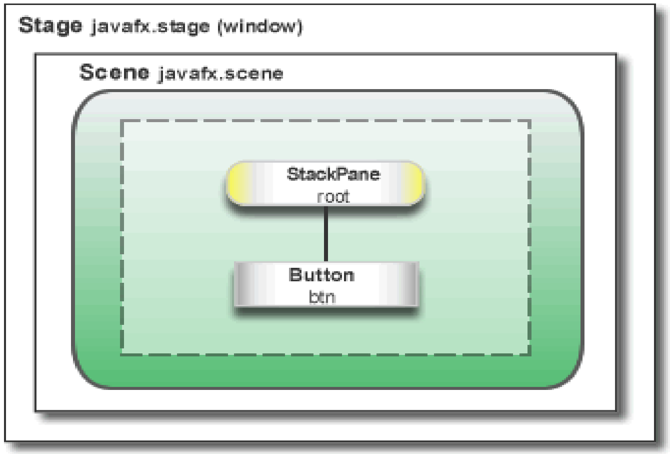
- The class that holds your user interface components is called a JFrame class.
- A JFrame is an empty window to which you can add a JPanel, which serves as a container for your user-interface elements.
- A Swing application is actually a class that extends the JFrame class. To display user-interface components, you add components to a JPanel and then add the panel to the frame.

Java Swing Vs JavaFX

In JavaFX, All the world is Stage



Java Swing Vs JavaFX



In JavaFX, All the world is Stage

- A stage is the highest level container .
- The individual controls and other components that make up the user interface are contained in a scene .
- An application can have more than one scene, but only one of the scenes can be displayed on the stage at any given time.
- A scene contains a scene graph, is a collection of all the elements [nodes] that make up a user interface .

Java Swing Vs JavaFX

- JavaFX formatting can be controlled with CSS.
- JavaFX has several interesting controls that Swing doesn't have, such as the collapsible TitledPane control and the Accordion control.
- The javafx.scene.effect package contains a number of classes that can easily apply special effects to any node in the scene graph.
- Swing does not provide any direct support for animation.
- JavaFX has built-in support for sophisticated animations that can be applied to any node in the scene graph.
- JavaFX supports multi-touch operations.
- .

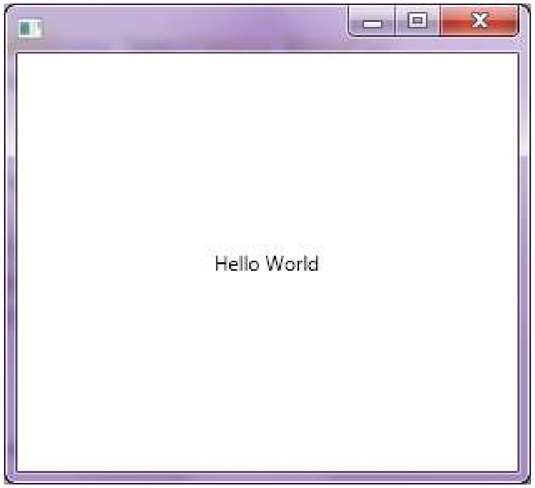
Hello world program in JavaFX

- There is three way to make a hello world program

1. Simple code using classes
2. Using FXML (XML)
3. Using Scene builder

Hello world program in JavaFX

```
public class HelloWorld extends Application{

    @Override
    public void start(Stage primaryStage) throws Exception {

        Text helloWorld = new Text("Hello World FX!!!");

        StackPane rootPane = new StackPane(helloWorld);

        Scene scene = new Scene(rootPane, 400, 300);

        primaryStage.setScene(scene);
        primaryStage.show();

    }


    public static void main(String[] args) {
        Application.launch(args);
    }

}
```

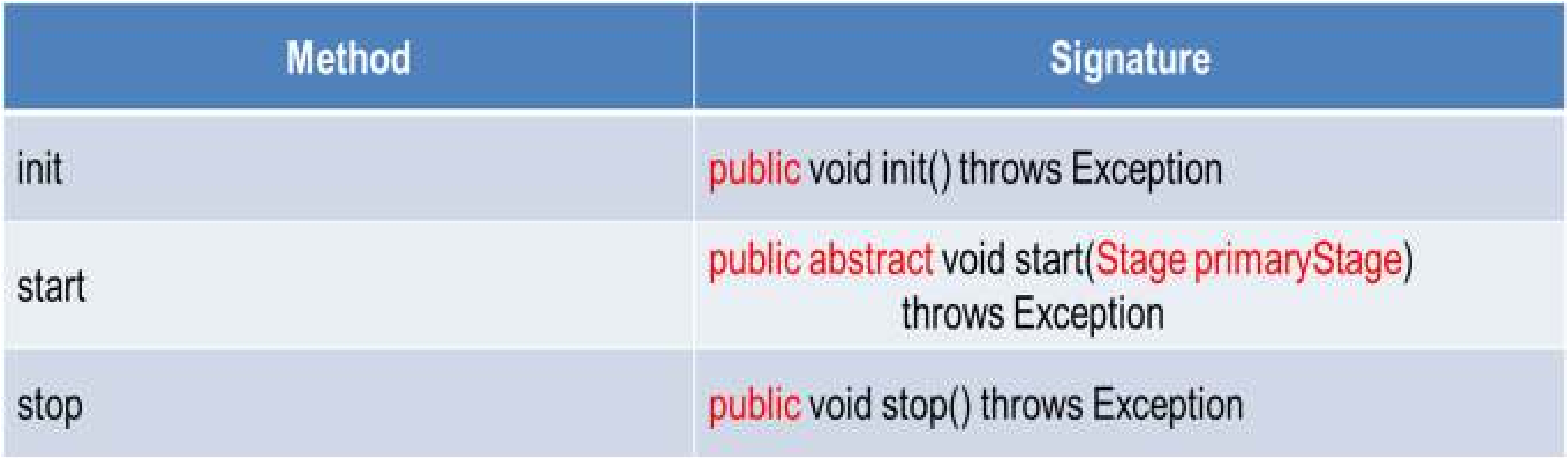


Hello World

Hello world program in JavaFX

Application Class

- The entry point for JavaFX applications.
- The Application class is an abstract class with a single abstract method start.
- The application class has three important method :



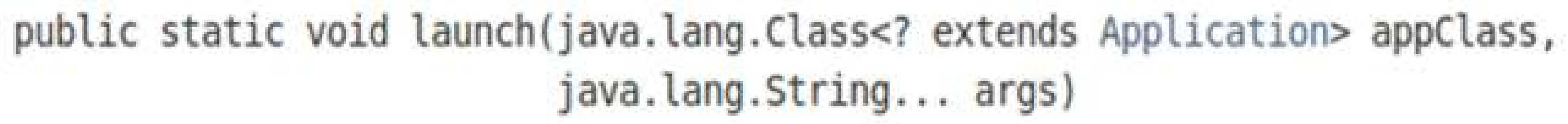
Method	Signature
init	public void init() throws Exception
start	public abstract void start(Stage primaryStage) throws Exception
stop	public void stop() throws Exception

Hello world program in JavaFX

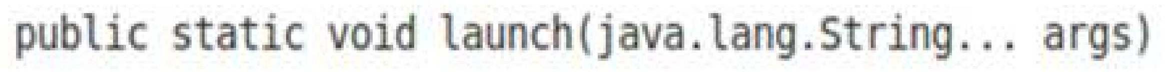
Application Class

launch() method:

- This method is typically called from the main method().
- It must not be called more than once or an exception will be thrown.
- The launch method does not return until the application has exited , either via a call to Platform.exit or all of the application windows have been closed and the Platform attribute implicitExit is set to true.

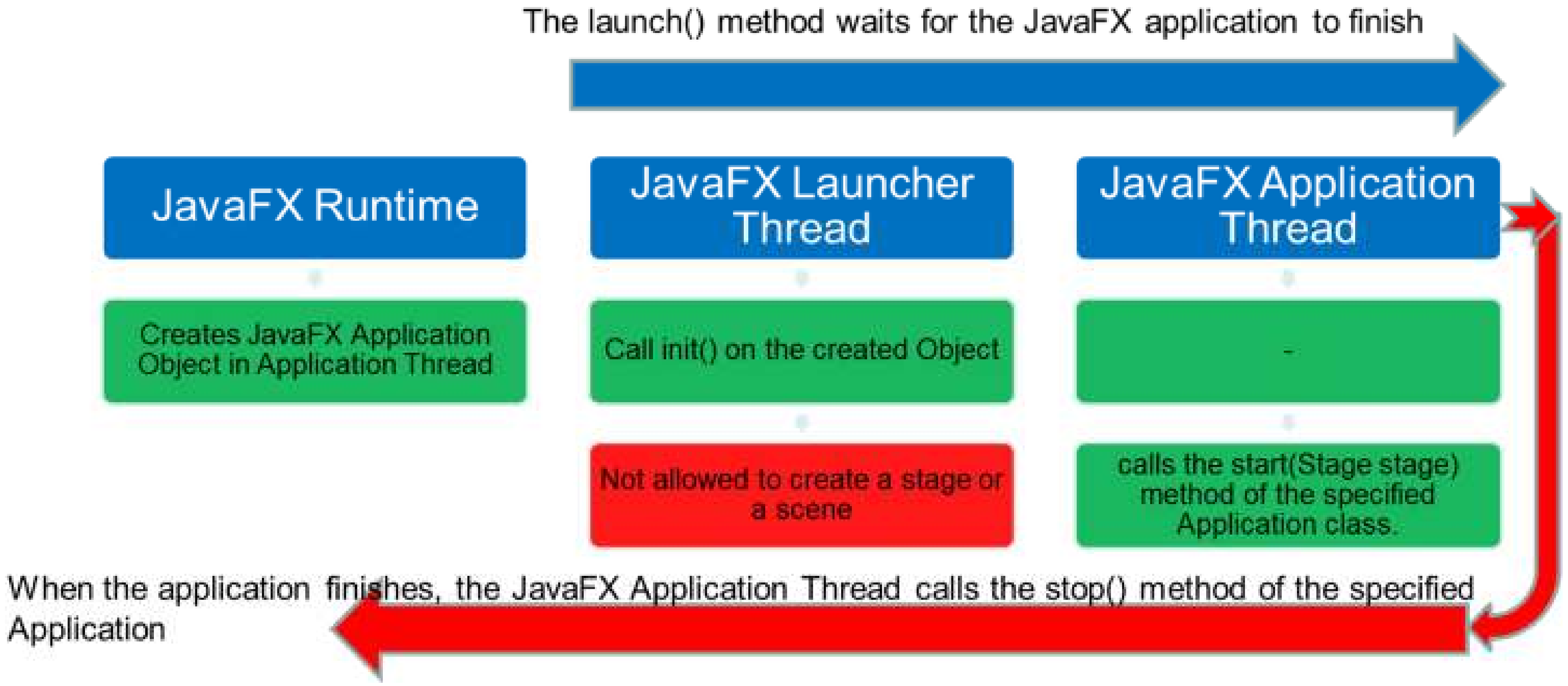


```
public static void launch(java.lang.Class<? extends Application> appClass,
                          java.lang.String... args)
```



```
public static void launch(java.lang.String... args)
```

The Life Cycle of a JavaFX Application



The launch() method waits for the JavaFX application to finish

JavaFX Runtime

Creates JavaFX Application Object in Application Thread

JavaFX Launcher Thread

Call init() on the created Object

Not allowed to create a stage or a scene

JavaFX Application Thread

-

calls the start(Stage stage) method of the specified Application class.

When the application finishes, the JavaFX Application Thread calls the stop() method of the specified Application

The Life Cycle of a JavaFX Application

- JavaFX runtime is responsible for creating several threads
- At different phases in the application, threads are used to perform different tasks.
- The JavaFX runtime creates, among other threads, two threads:
  - JavaFX-Launcher Thread
  - JavaFX Application Thread

- The launch() method of the Application class create these threads.

The Life Cycle of a JavaFX Application

- During the lifetime of a JavaFX application, the JavaFX runtime calls the following methods of the specified JavaFX Application class in order:
  - The no-args constructor [in Application Thread]
  - The init() method [in Launcher Thread]
  - The start() method [in Application Thread]
  - The stop() method [in Application Thread]

The Life Cycle of a JavaFX Application

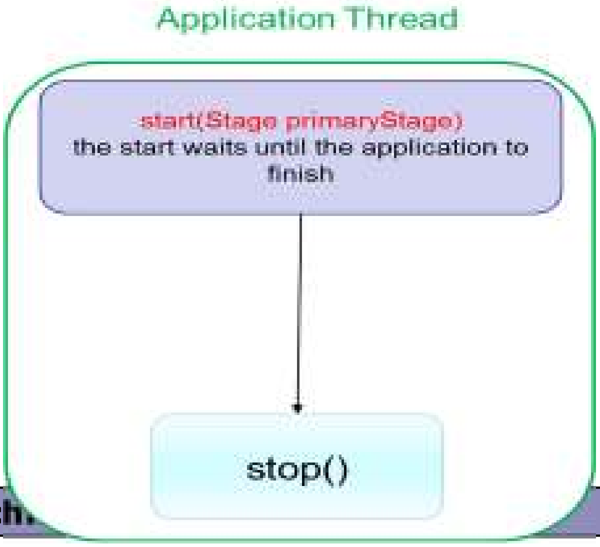
Launcher Thread

- ❑ Is the thread that is used to launch the application.
- ❑ Constructing and modifying the Stage on the launcher thread is not allowed as it will throw an exception.
- ❑ Also modifying objects that are attached to the scene graph.

The Life Cycle of a JavaFX Application

Application Thread

- Is the thread that is used to invoke the start() and stop() methods.
- This thread is used to construct and modify the JavaFX Stage and Scene, Process input events, running animation timeline, and apply modifications to live objects (objects that are attached to the scene).



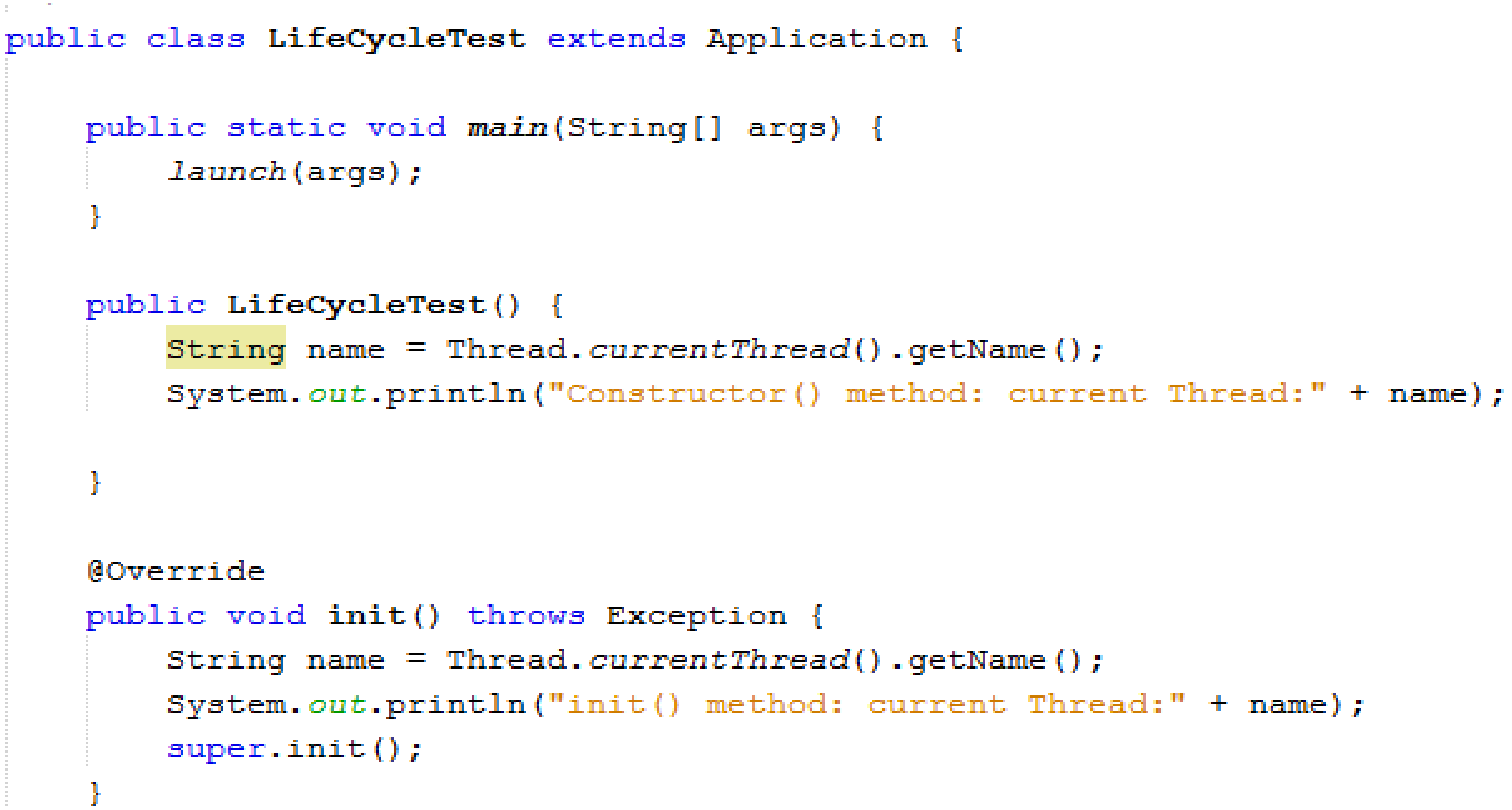
Application Thread

start(Stage primaryStage)
the start waits until the application to finish

stop()

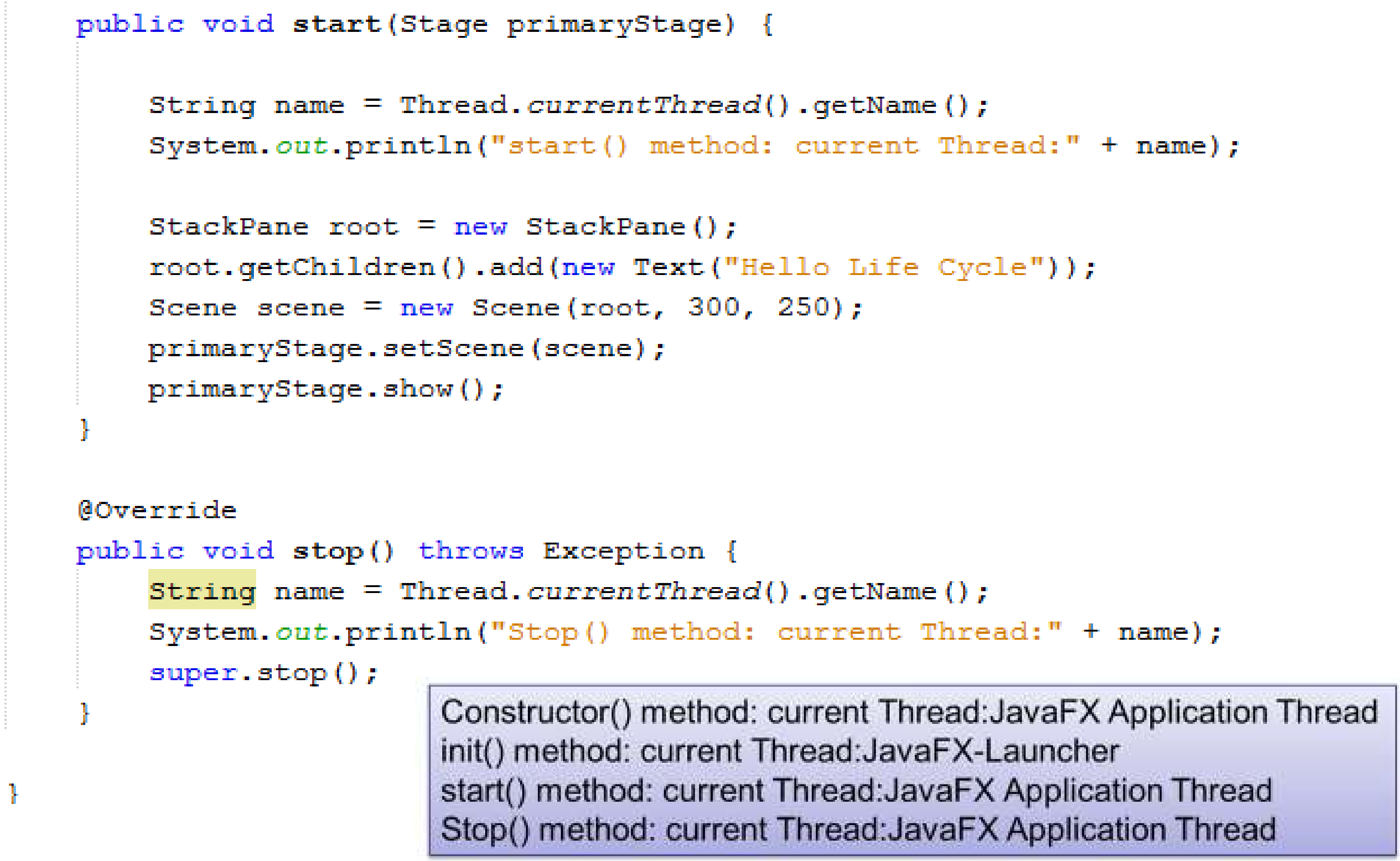
The Life Cycle of a JavaFX Application

- The following example code illustrates the life cycle of a JavaFX application.



```
public class LifeCycleTest extends Application {

    public static void main(String[] args) {
        launch(args);
    }

    public LifeCycleTest() {
        String name = Thread.currentThread().getName();
        System.out.println("Constructor() method: current Thread:" + name);

    }

    @Override
    public void init() throws Exception {
        String name = Thread.currentThread().getName();
        System.out.println("init() method: current Thread:" + name);
        super.init();
    }
```

The Life Cycle of a JavaFX Application



```java
    public void start(Stage primaryStage) {

        String name = Thread.currentThread().getName();
        System.out.println("start() method: current Thread:" + name);

        StackPane root = new StackPane();
        root.getChildren().add(new Text("Hello Life Cycle"));
        Scene scene = new Scene(root, 300, 250);
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    @Override
    public void stop() throws Exception {
        String name = Thread.currentThread().getName();
        System.out.println("Stop() method: current Thread:" + name);
        super.stop();
    }

}
```

```
Constructor() method: current Thread:JavaFX Application Thread
init() method: current Thread:JavaFX-Launcher
start() method: current Thread:JavaFX Application Thread
Stop() method: current Thread:JavaFX Application Thread
```

Platform class

- ❑ Is the JavaFX application Platform support class.
- ❑ It can check the current running thread (application thread or not),
- ❑ enqueue a task into the application thread,
- ❑ and control the default exit behavior.

- public static boolean isFxApplicationThread () //Returns true if the calling thread is the JavaFX Application Thread
- public static void runLater (Runnable runnable) //Run the specified Runnable on the JavaFX Application Thread in the future
- public static void exit () //Causes the JavaFX application to terminate

Program Structure

- A JavaFX application consists of three main components.
  - Nodes.
  - Scene.
  - Stage.

- The Node is the main actor of the application, and the visible component in our application.

- Scene is the component that the nodes are displayed on it.

- Stage is the base for the scene, and nodes.

Node

- A scene graph is a set of tree data structures where every item is a Node.

- Each item is either a "leaf" with zero sub-items or a "branch" with zero or more sub-items.

- A node may occur at most once anywhere in the scene graph.

- Node objects may be constructed and modified on any thread as long they are not yet attached to a Scene.

- Modifying nodes that are already attached to a Scene (live objects), on the JavaFX Application Thread only.

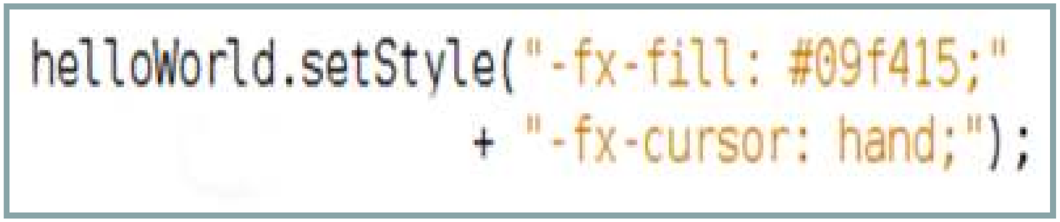
Node

- One of the greatest advantages of JavaFX is the ability to use CSS to style your nodes in the scene graph.

- JavaFX CSS are based on the W3C CSS version 2.1 specification.

- The default style sheet for JavaFX applications is *caspian.css*, which is found in the JavaFX runtime JAR file, jfxrt.jar. This style sheet defines styles for the root node and the UI controls.

- To change the default style of a node you can use the setStyle method.

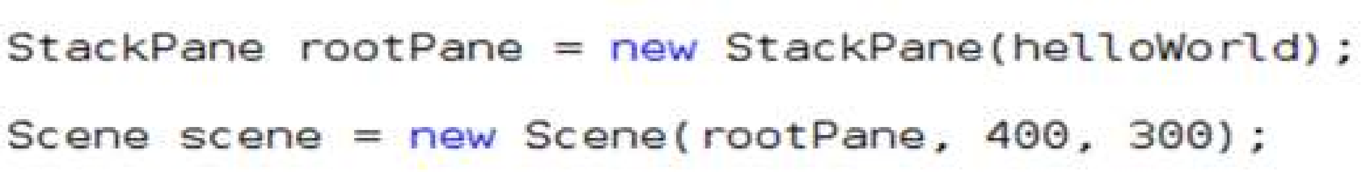


```
helloWorld.setStyle("-fx-fill: #09f415;"
                  + "-fx-cursor: hand;");
```

Scene

The JavaFX Scene class is the container for all content in a scene graph.

The application must specify the root Node for the scene graph by setting the root property.



```
StackPane rootPane = new StackPane(helloWorld);

Scene scene = new Scene(rootPane, 400, 300);
```

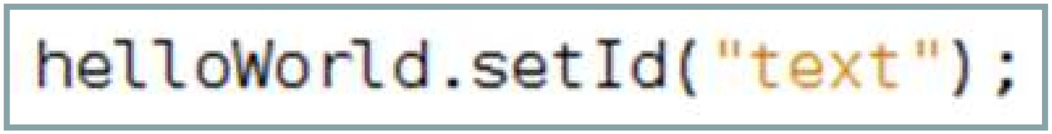
The scene's size may be initialized by the application during construction. If no size is specified, the scene will automatically compute its initial size based on the preferred size of its content.

Scene objects must be constructed and modified on the JavaFX Application Thread.

Scene and Style

Any node that will be displayed on the screen must be attached to the scene some how.

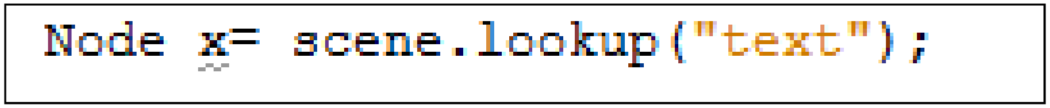
Each node can have an Id property to identify it.



```
helloWorld.setId("text");
```

The node can be later re-located using the lookup method.

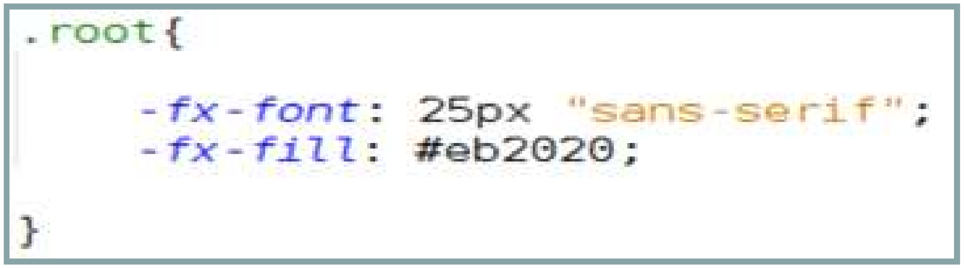
This is very helpful when using CSS styles, as we will not write a style for node by node we use style classes.



```
Node x= scene.lookup("text");
```

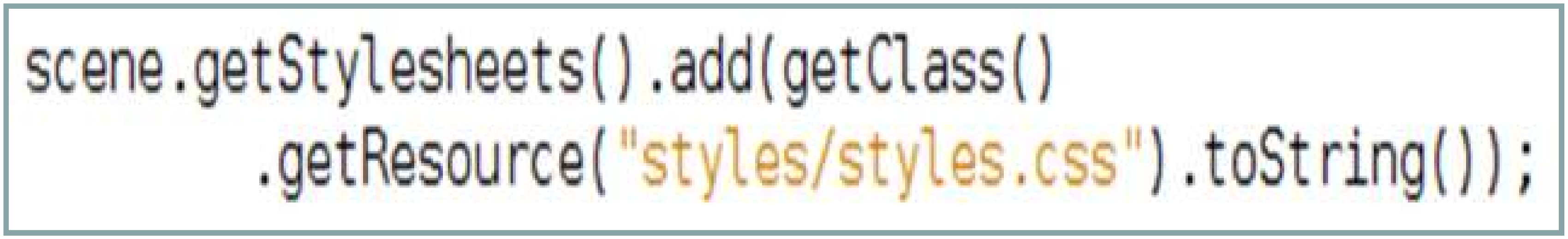
Scene and Style

To create a style class we first create a .css file to indicate a CSS style sheet.



```
.root{

      -fx-font: 25px "sans-serif";
      -fx-fill: #eb2020;

}
```

To apply this style to our scene we must add this sheet to the scene styles.

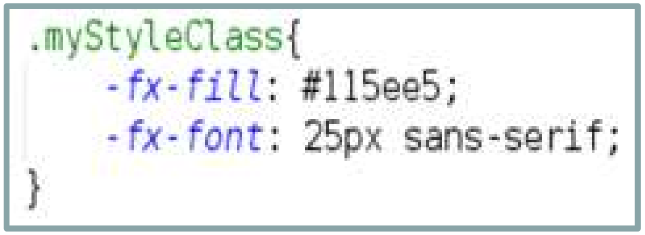


```
scene.getStylesheets().add(getClass()
        .getResource("styles/styles.css").toString());
```

Scene and Style

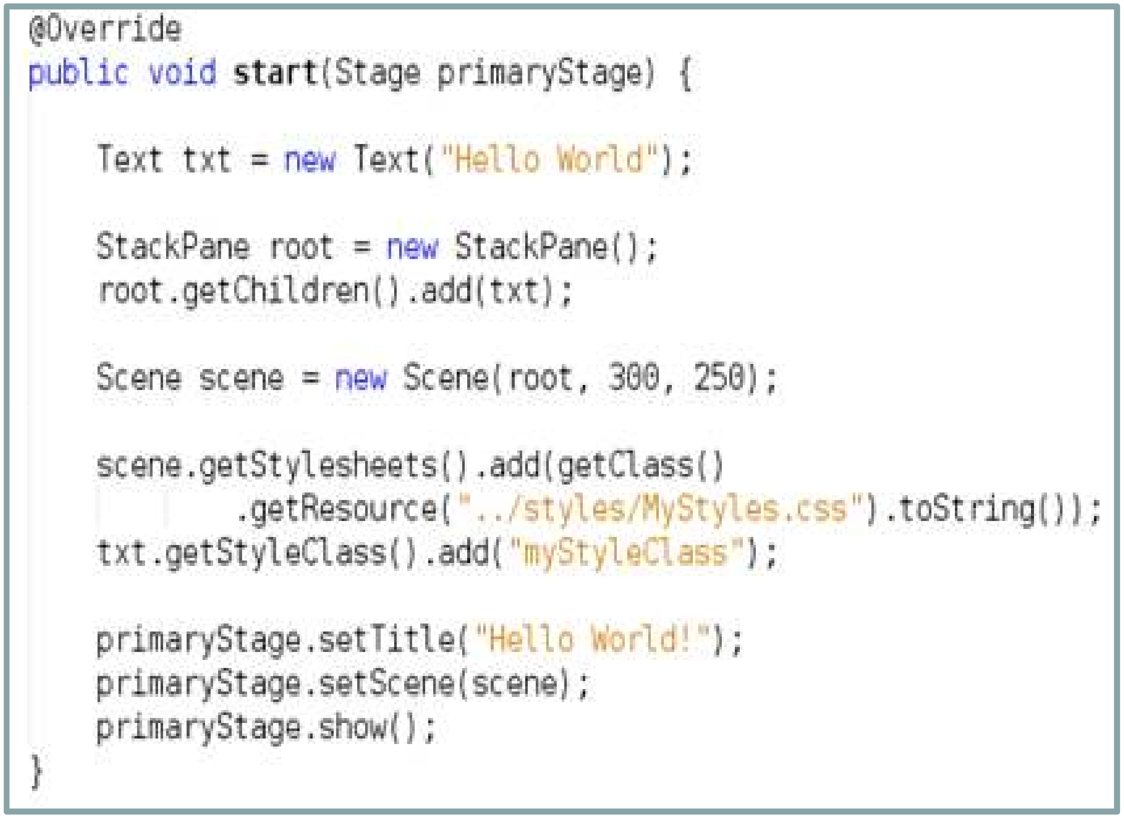
To add a certain class to a node you can use the getStyleClasses().add() method.

MyStyles.css



```
.myStyleClass{
    -fx-fill: #115ee5;
    -fx-font: 25px sans-serif;
}
```

Application start() mehod

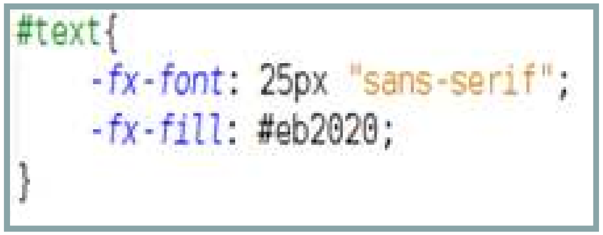


```
@Override
public void start(Stage primaryStage) {

    Text txt = new Text("Hello World");

    StackPane root = new StackPane();
    root.getChildren().add(txt);

    Scene scene = new Scene(root, 300, 250);

    scene.getStylesheets().add(getClass()
            .getResource("../styles/MyStyles.css").toString());
    txt.getStyleClass().add("myStyleClass");

    primaryStage.setTitle("Hello World!");
    primaryStage.setScene(scene);
    primaryStage.show();
}
```

Scene and Style

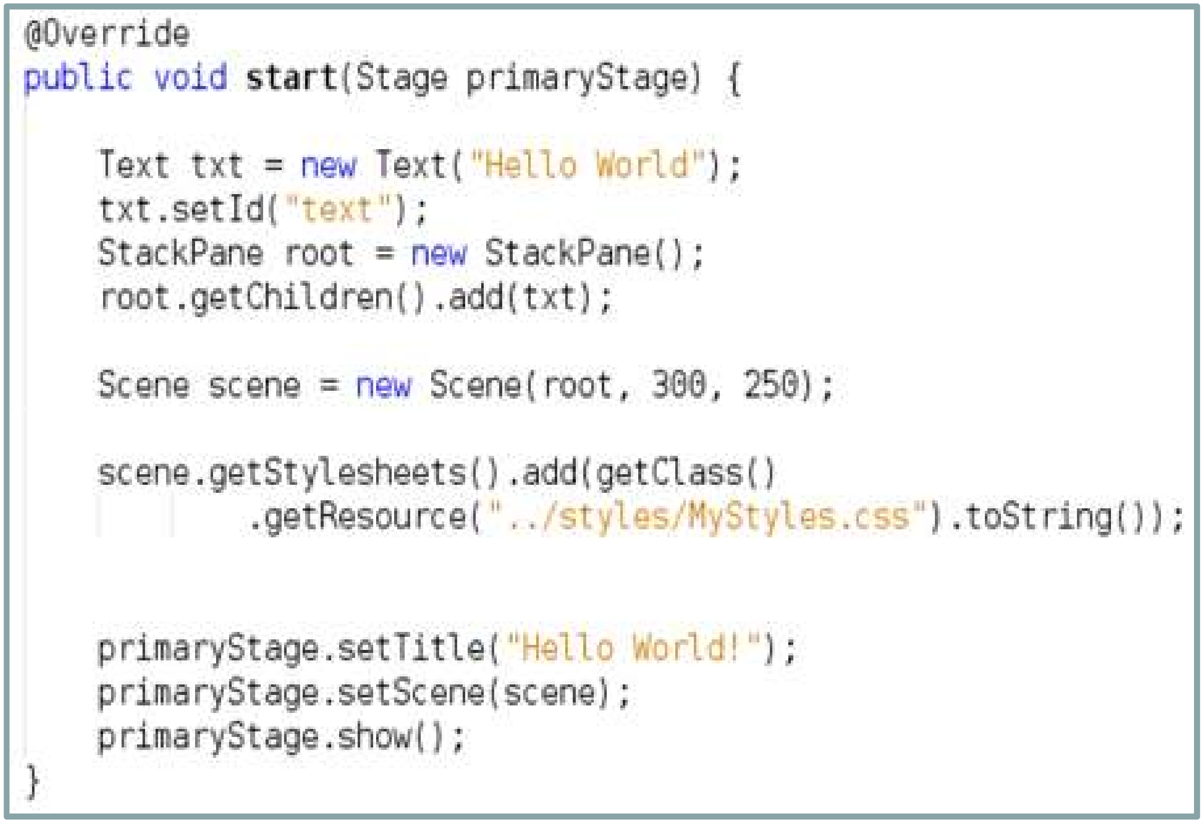
You can create a style class for a certain node in the scene using the hash symbol (#) and the node Id.

MyStyle.css



```
#text{
    -fx-font: 25px "sans-serif";
    -fx-fill: #eb2020;
}
```

Application start() method



```
@Override
public void start(Stage primaryStage) {

    Text txt = new Text("Hello World");
    txt.setId("text");
    StackPane root = new StackPane();
    root.getChildren().add(txt);

    Scene scene = new Scene(root, 300, 250);

    scene.getStylesheets().add(getClass()
            .getResource("../styles/MyStyles.css").toString());


    primaryStage.setTitle("Hello World!");
    primaryStage.setScene(scene);
    primaryStage.show();
}
```

Stage

The JavaFX Stage class is the top level JavaFX container.

The primary Stage is constructed by the platform.

Stage object must be constructed and modified on the JavaFX Application Thread.

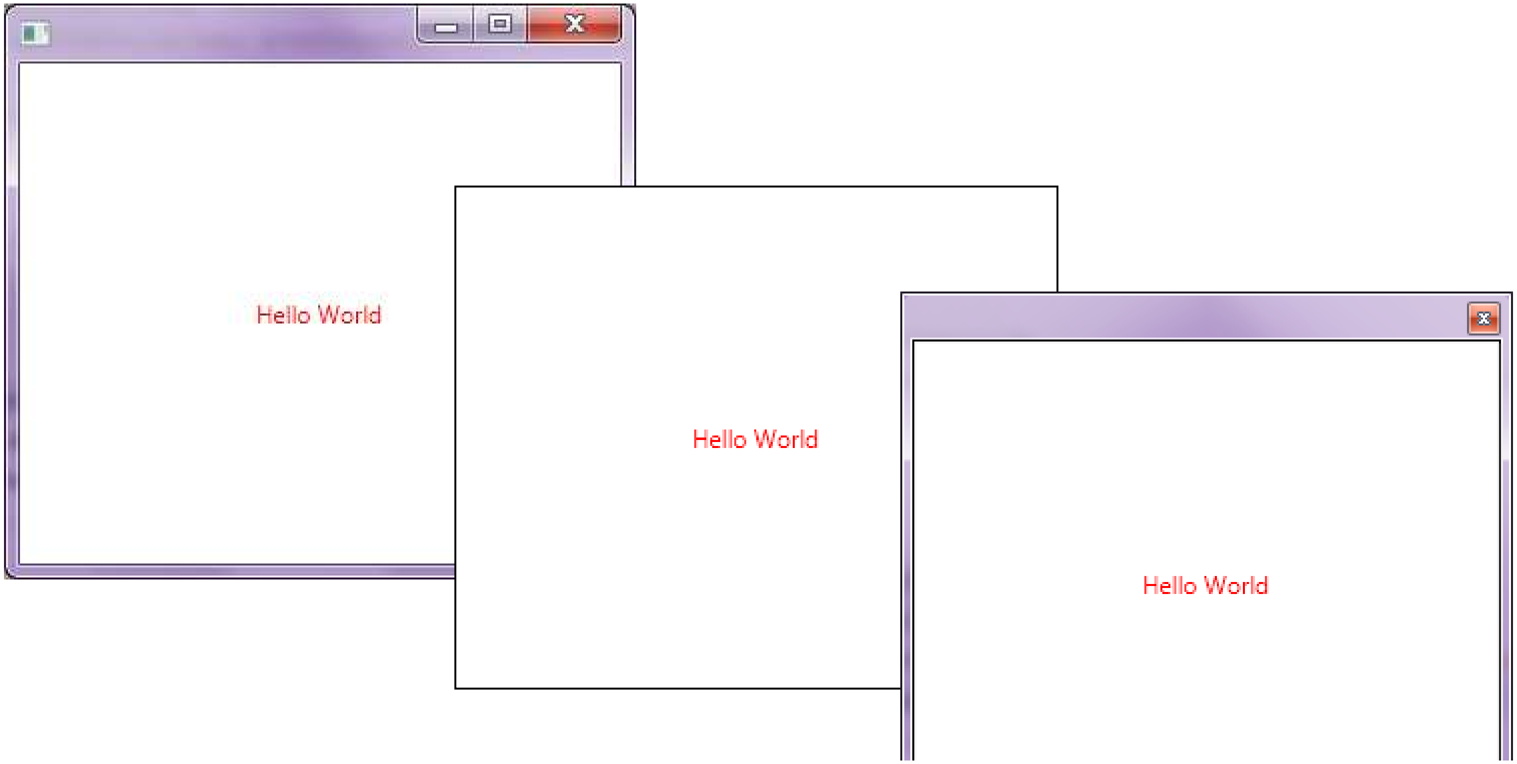
A stage has one of the following styles:

- StageStyle.DECORATED : a stage with a solid white background and platform decorations.
- StageStyle.UNDECORATED : a stage with a solid white background and no decorations.
- StageStyle.TRANSPARENT : a stage with a transparent background and no decorations.
- StageStyle.UTILITY : a stage with a solid white background and minimal platform decorations.

- The style must be initialized before the stage is made visible.

Stage

A stage has one of the following styles:



```
primaryStage.initStyle(StageStyle.UTILITY);
```

Stage

A stage can optionally have an owner Window.

```
public final void initOwner(Window owner)
```

When a parent window is closed, all its descendant windows are closed.

A stage has one of the following modalities:

- **None**:stage that does not block any other window.
- **Window Modal**:a stage that blocks input events from being delivered to all windows from its owner (parent) to its root.
- **Application Modal**:a stage that blocks input events from being delivered to all windows from the same application, except for those from its child hierarchy.

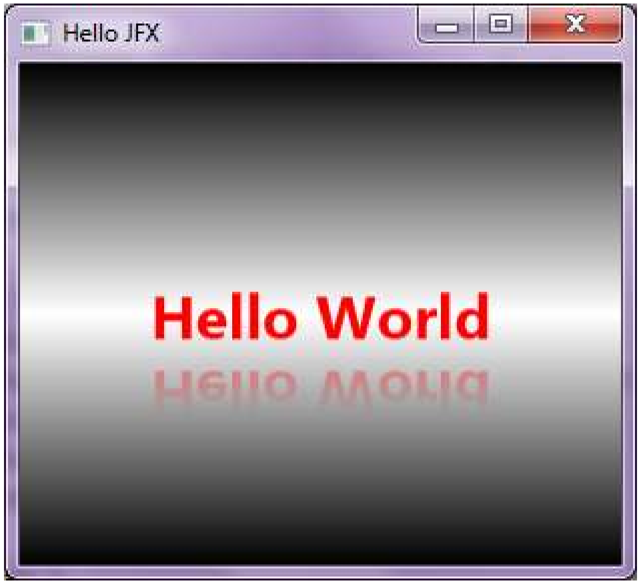
Lab Exercise

Hello World

- JavaFX Lifecycle application
- Create the Hello World application to match the following style.

1. Use JavaFX code [Reflection , LinearGradient]

2. Use CSS style sheet.



Lesson 6

Building UI Using JavaFX

Basic Controls, Event Handling, and Layout

Building UI Using JavaFX

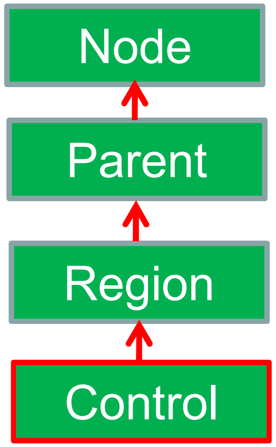
Basic Controls

Control

- Class Control is the base class for all javaFX Controls.

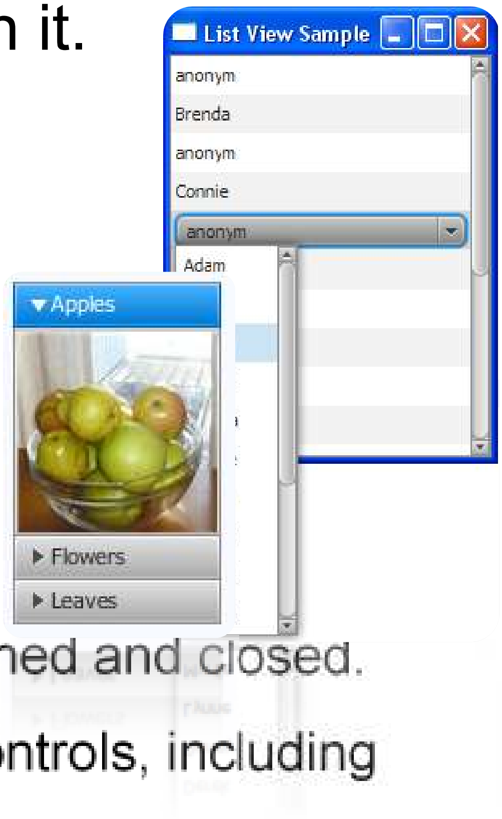
- Class Control is a sub-class of class Node, so it can be treated as node in the scene plus its variables and behavoiurs as control to support user interactions.

- controls support explicit skinning to make it easy to leverage the functionality of a control while customizing its appearance (Context menu, skin, Tooltip).



Labeled controls

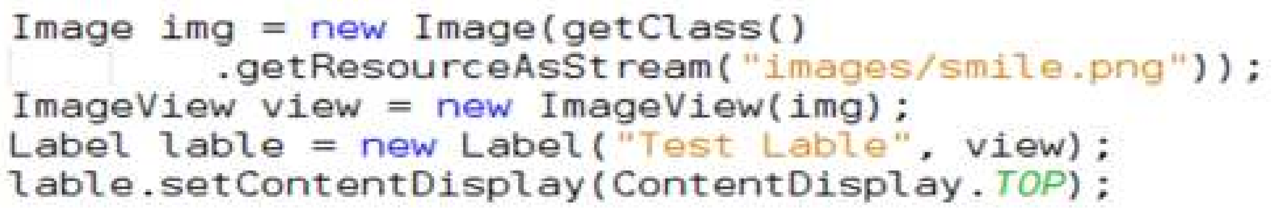
- A Labeled Control is one which has as part of its user interface a textual content associated with it.



- It has four sub-classes:
  - **Cell**: used for virtualized controls such as:
    - ListView, TreeView, and TableView.
  - **Label:** is a non-editable text control.
  - **TitledPane**: panel with a title that can be opened and closed.
  - **ButtonBase:** Base class for button-like UI Controls, including Hyperlinks, Buttons, ToggleButtons, CheckBoxes, and RadioButtons.

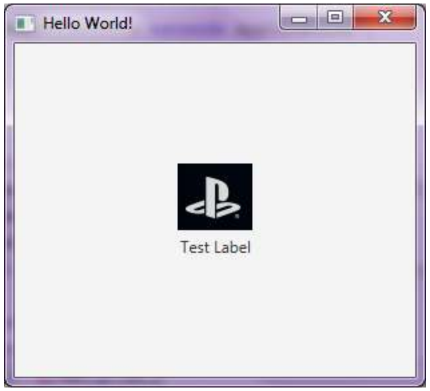
Labeled controls

- We can customize a Labeled control to hold also images and text.



```
Image img = new Image(getClass()
        .getResourceAsStream("images/smile.png"));
ImageView view = new ImageView(img);
Label lable = new Label("Test Lable", view);
lable.setContentDisplay(ContentDisplay.TOP);
```

-



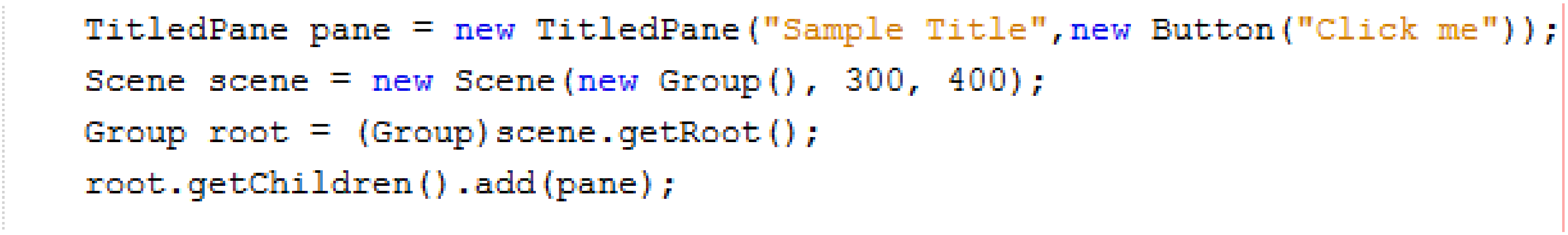
Labeled controls

- The panel in a TitledPane can be any Node such as UI controls or groups of nodes added to a layout container.

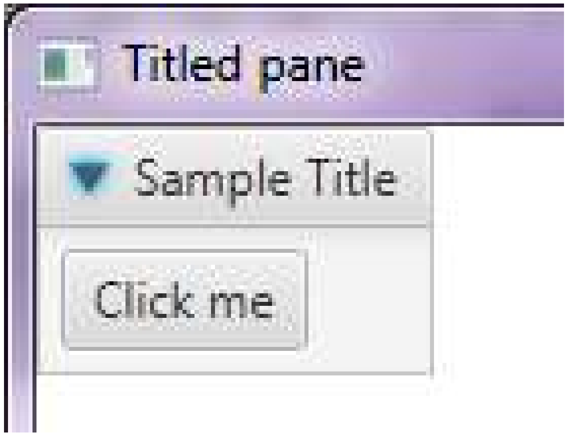
- Note: the inherited properties from class Labeled are used to manipulate the header area not the content area.

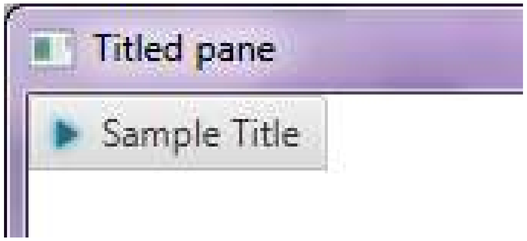
- It is not recommended to set the MinHeight, PrefHeight, or MaxHeight for this control. Unexpected behavior will occur because the TitledPane's height changes when it is opened or closed.

Labeled controls



```
TitledPane pane = new TitledPane("Sample Title",new Button("Click me"));
Scene scene = new Scene(new Group(), 300, 400);
Group root = (Group)scene.getRoot();
root.getChildren().add(pane);
```

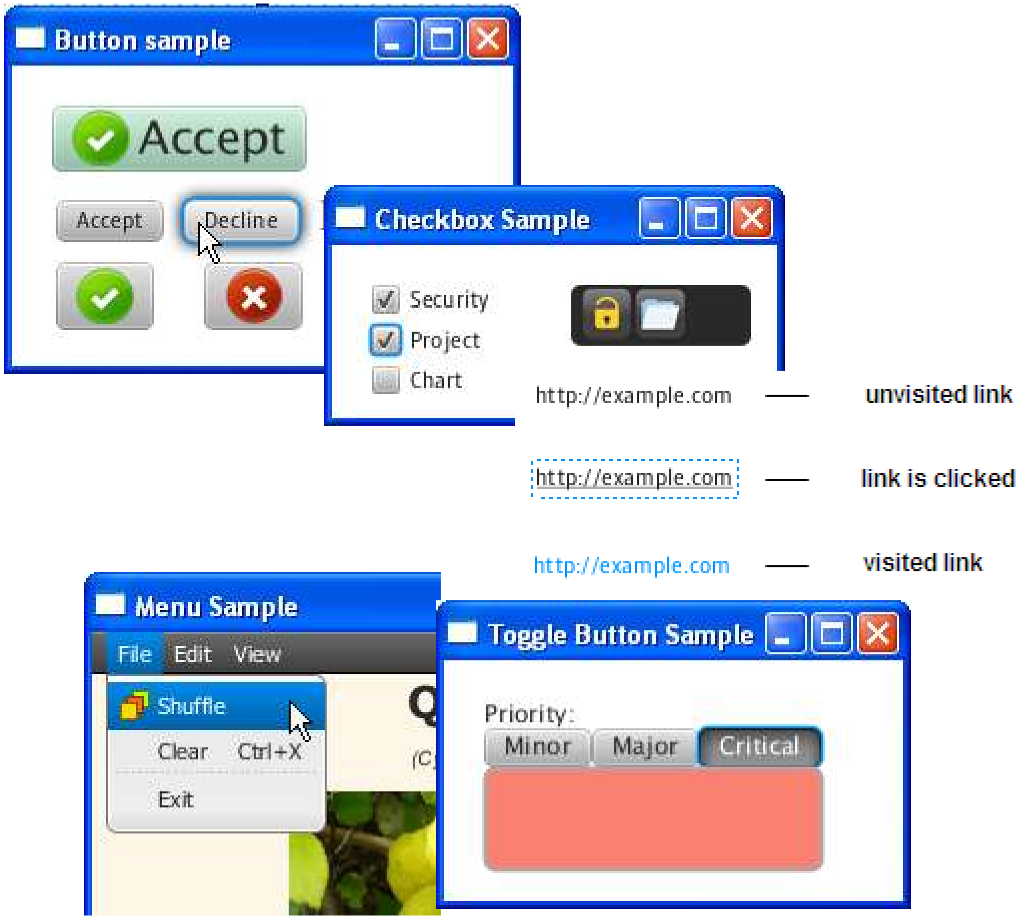




ButtonBase class

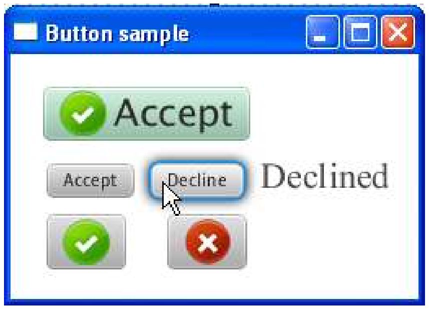
- The ButtonBase class is an extension of the Labeled class. It can display text, an image, or both.

- Sub-classes are:
  - Button.
  - CheckBox.
  - HyberLink.
  - MenuButton.
  - ToggleButton.



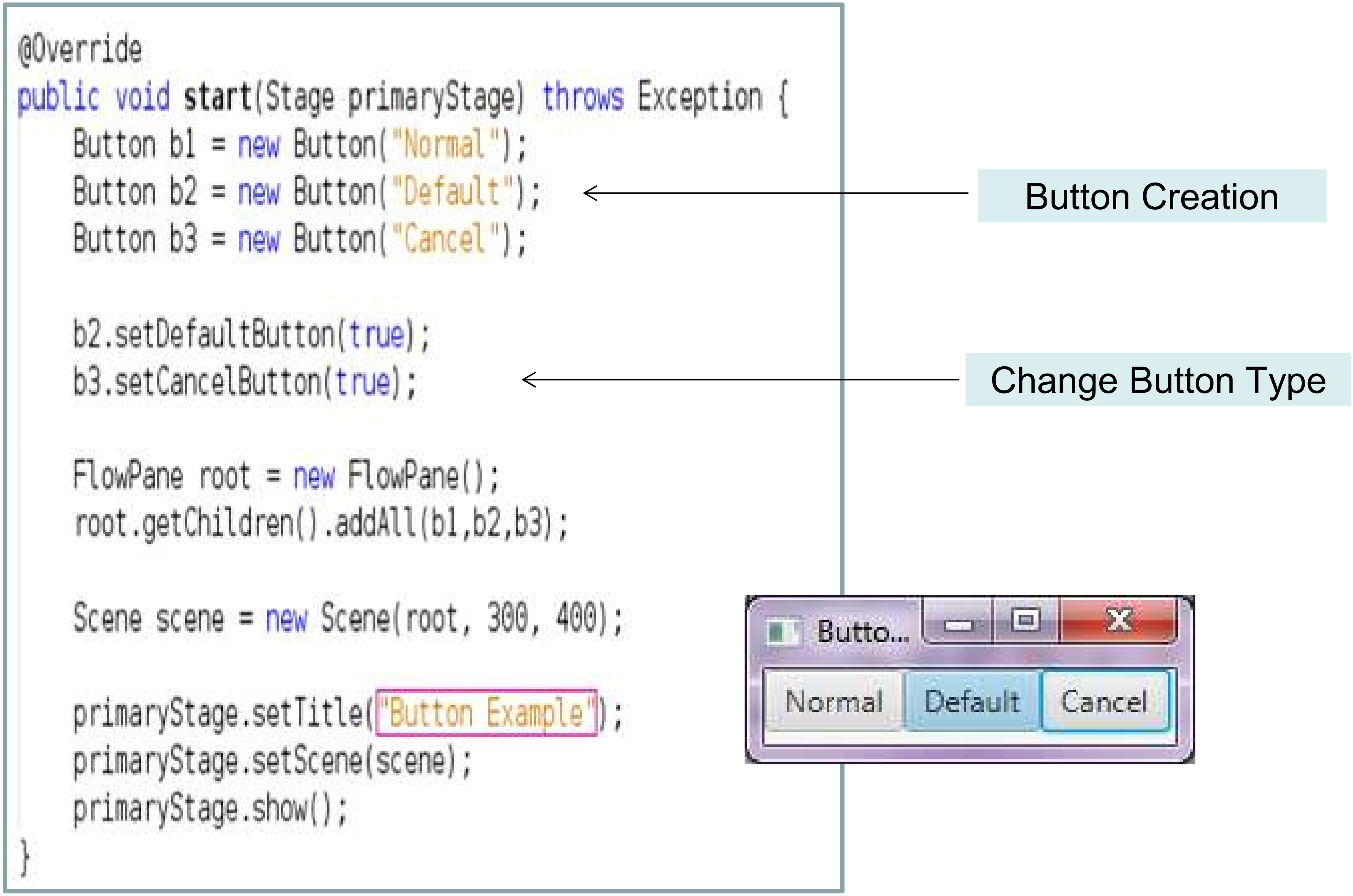
Button

- A simple button control.
- The button control can contain text and/or a graphic.



- A button control has three different modes:
  - **Normal:** A normal push button.
  - **Default:** A default Button is the button that receives a keyboard VK_ENTER press, if no other node in the scene consumes it.
  - **Cancel:** A Cancel Button is the button that receives a keyboard VK_ESC press, if no other node in the scene consumes it.

Button



```java
@Override
public void start(Stage primaryStage) throws Exception {
    Button b1 = new Button("Normal");
    Button b2 = new Button("Default");
    Button b3 = new Button("Cancel");

    b2.setDefaultButton(true);
    b3.setCancelButton(true);

    FlowPane root = new FlowPane();
    root.getChildren().addAll(b1,b2,b3);

    Scene scene = new Scene(root, 300, 400);

    primaryStage.setTitle("Button Example");
    primaryStage.setScene(scene);
    primaryStage.show();
}
```

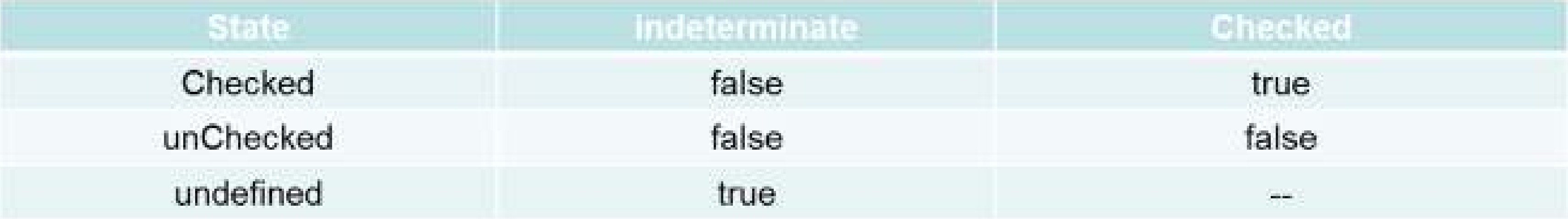
Button Creation

Change Button Type

CheckBox

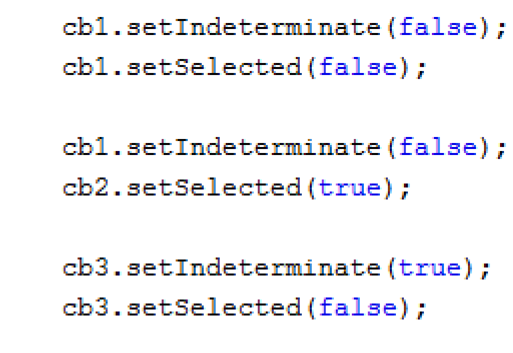
- A tri-state selection Control typically skinned as a box with a checkmark or tick mark when checked.

- A CheckBox control can be in one of three states:

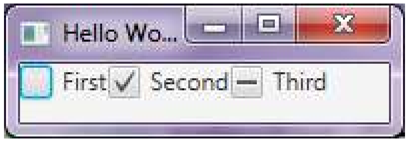


State	Indeterminate	Checked
Checked	false	true
unChecked	false	false
undefined	true	--

- When the checkbox is undefined, it cannot be selected or deselected.

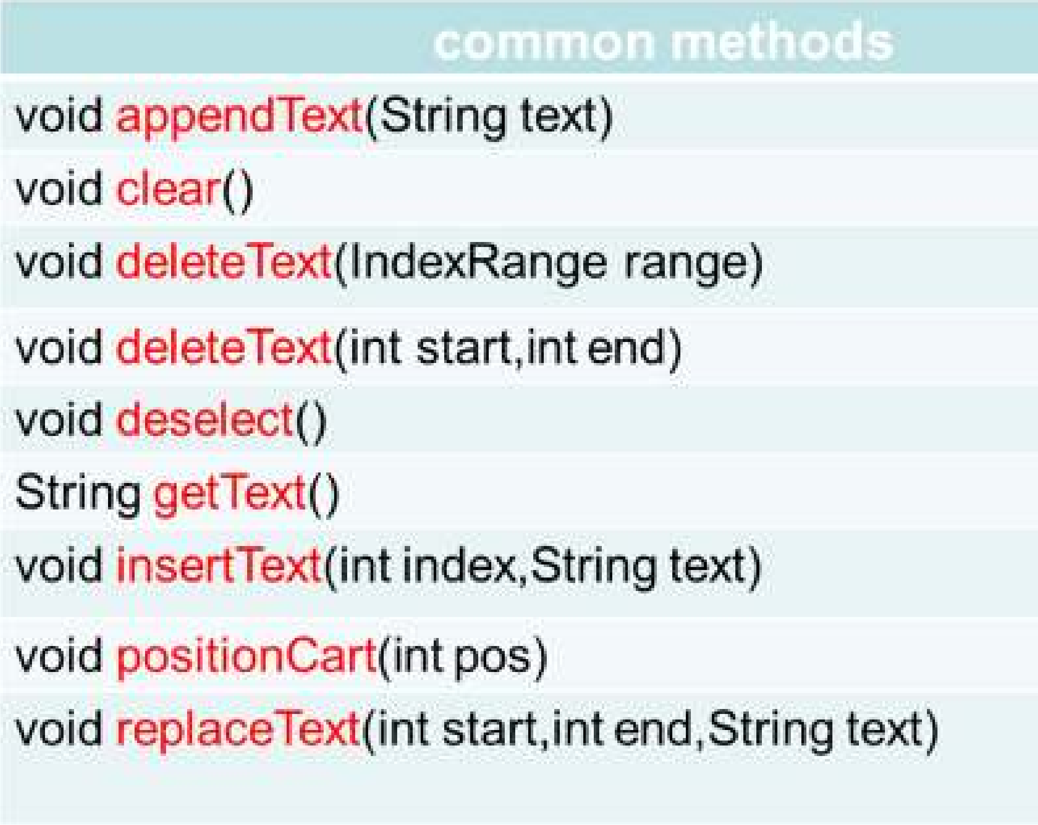


```
cb1.setIndeterminate(false);
cb1.setSelected(false);

cb1.setIndeterminate(false);
cb2.setSelected(true);

cb3.setIndeterminate(true);
cb3.setSelected(false);
```

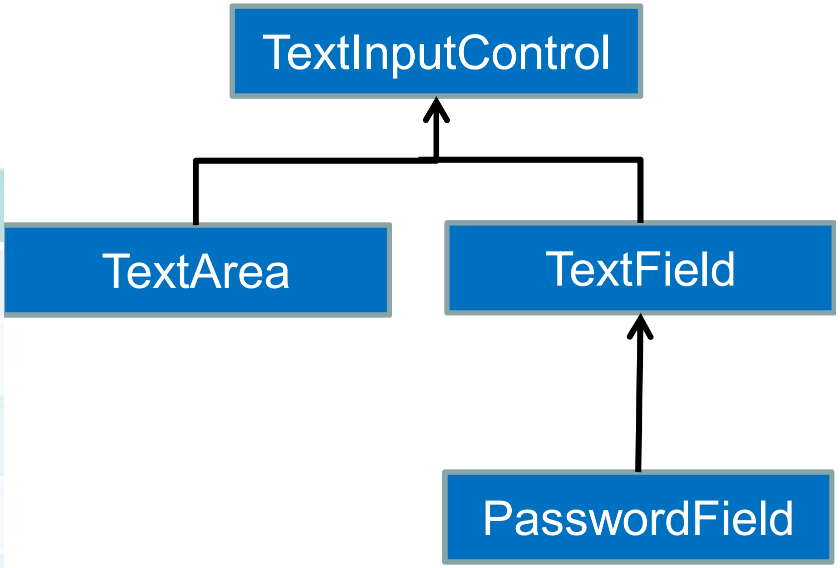


TextInputControl

- Abstract base class for text input controls.



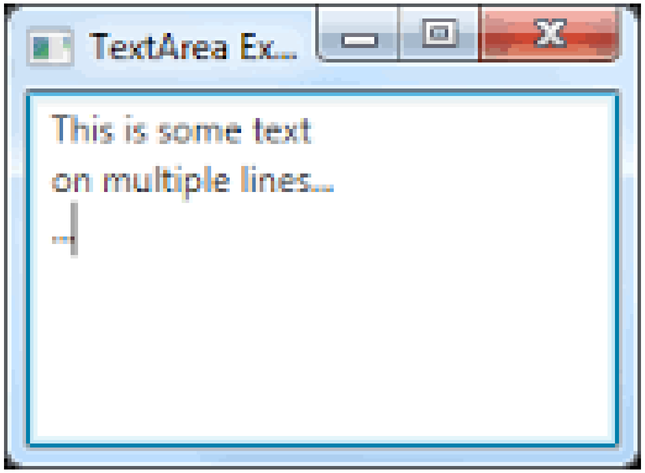
common methods
void appendText(String text)
void clear()
void deleteText(IndexRange range)
void deleteText(int start,int end)
void deselect()
String getText()
void insertText(int index,String text)
void positionCart(int pos)
void replaceText(int start,int end,String text)



TextArea

- Text input component that allows a user to enter multiple lines of plain text.

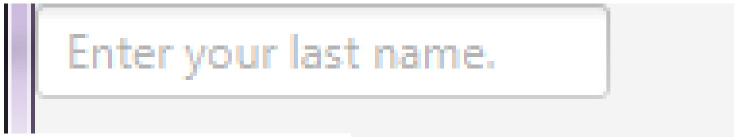
- You can use the setPrefRawCount(), and setPrefColCount() to adjust the prefared size of the TextArea.

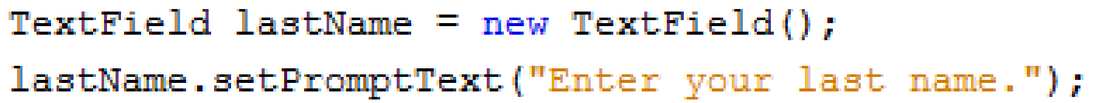


This is some text
on multiple lines...
...

TextField

- Text input component that allows a user to enter a single line of unformatted text.
- As it is one single line setPrefColCount() to control the number of colums.
- TextField fires ActionEvent upon typing the Enter key.

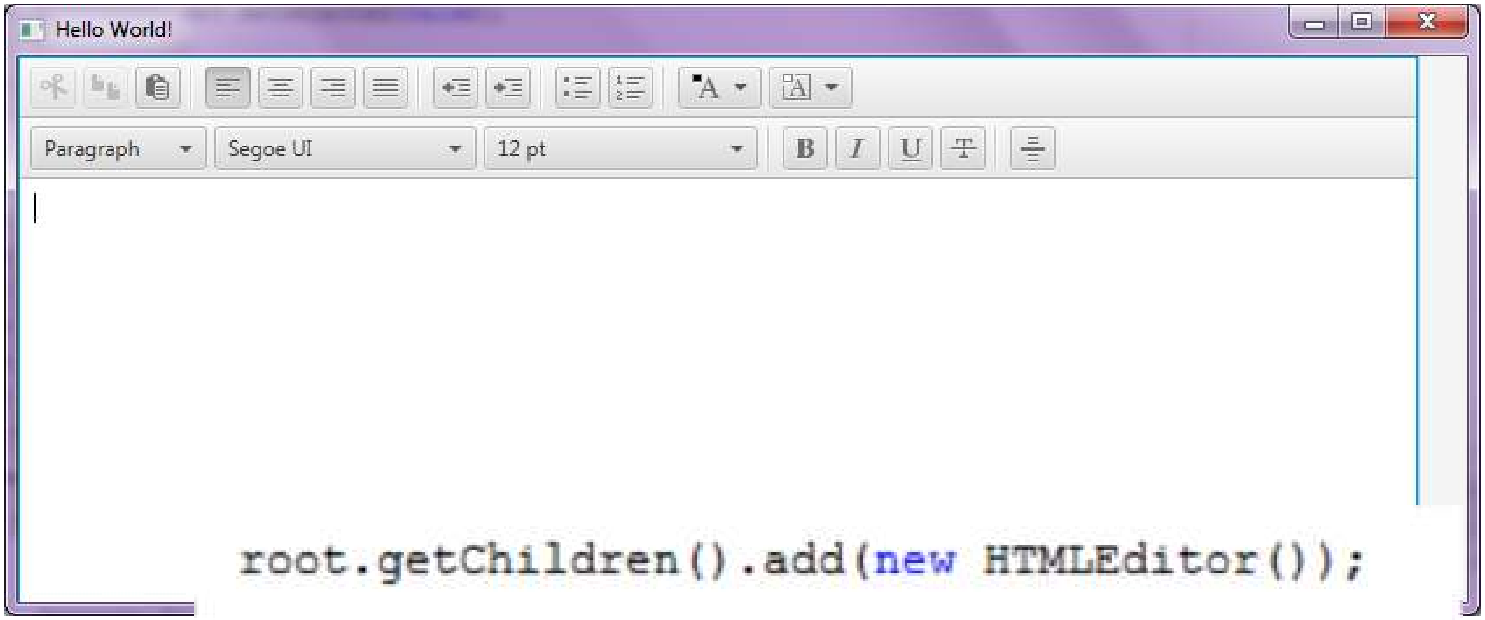




```
TextField lastName = new TextField();
lastName.setPromptText("Enter your last name.");
```

HTML Editor

- It allows you to edit text in your JavaFX applications by using the embedded HTML editor.



```
root.getChildren().add(new HTMLEditor());

Scene scene = new Scene(root, 300, 250);
```

Creating Menus

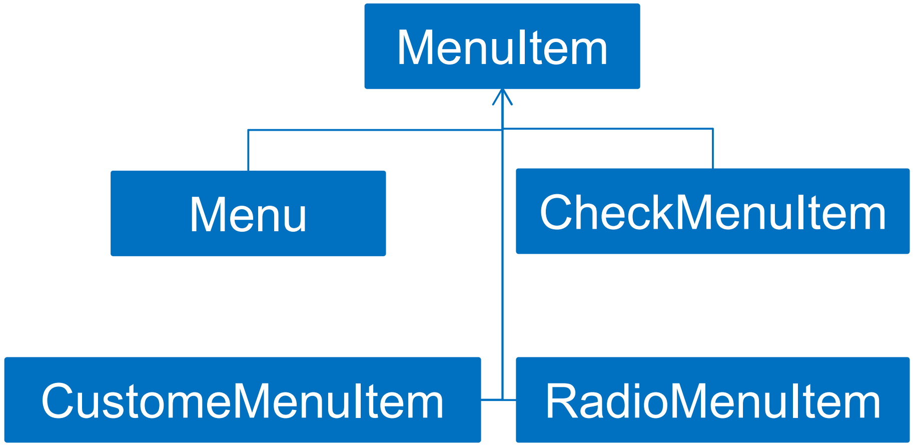
- Constructing a Menu in JavaFX is no deferent than Swing, you create MenuBar, Menu, and MenuItems, then we add them to each other.
- The deference between JavaFX and Swing that JavaFX does not have a pre-made Anchor for the menubar, so there is no setMenuBar() method like Swing.
- MenuBar it self is considered a node that can be added to any part of the located Pane.

MenuBar

- A MenuBar control traditionally is placed at the very top of the user interface, and embedded within it are Menus.
- To add a Menu to a MenuBar , you add it to the menus Observable List.

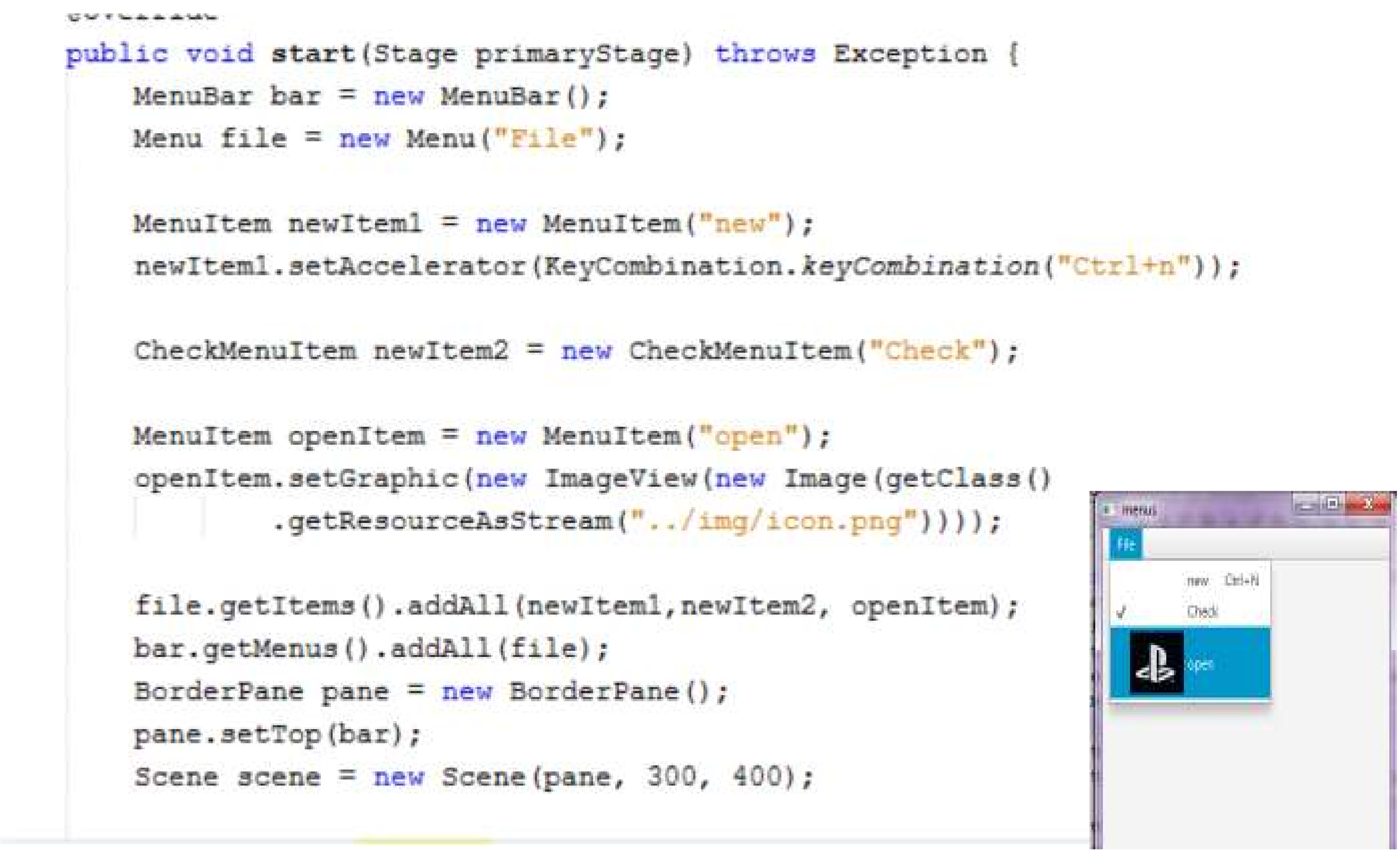
Constructors
MenuBar()
MenuBar(Menu...)
Methods
ObservableList<Menu> getMenus()

Menus and MenuItems



- MenuItem :
  - to create one actionable option
  - The accelerator property enables accessing the associated action in one keystroke.
- Menu : to create a Menu / submenu
- RadioButtonItem : to create a mutually exclusive selection
- CheckMenuItem : to create an option that can be toggled between selected and unselected states

Menus and MenuItems



```
@Override
public void start(Stage primaryStage) throws Exception {
    MenuBar bar = new MenuBar();
    Menu file = new Menu("File");

    MenuItem newItem1 = new MenuItem("new");
    newItem1.setAccelerator(KeyCombination.keyCombination("Ctrl+n"));

    CheckMenuItem newItem2 = new CheckMenuItem("Check");

    MenuItem openItem = new MenuItem("open");
    openItem.setGraphic(new ImageView(new Image(getClass()
            .getResourceAsStream("../img/icon.png"))));

    file.getItems().addAll(newItem1,newItem2, openItem);
    bar.getMenus().addAll(file);
    BorderPane pane = new BorderPane();
    pane.setTop(bar);
    Scene scene = new Scene(pane, 300, 400);
```

Building UI Using JavaFX

Event Handling

Event Handling

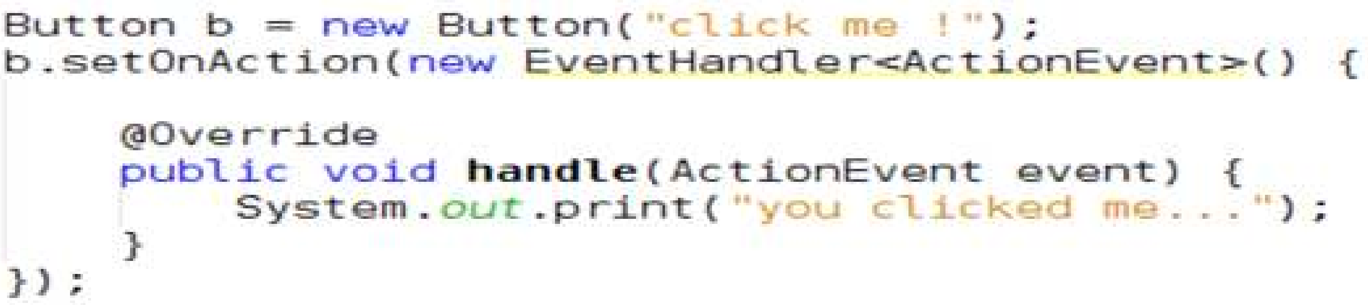
- Event model in JavaFX is no deferent than Swing, there is an event source and a listener to the event.

- Unlike swing JavaFX consider all event triggers as a reference property inside class Node. We only need to link the correct event listener to the property we want to respond to.

- JavaFX uses only one generic interface to respond to all events EventHandler<T extends Event>. The only method in this interface is handle(T Event).

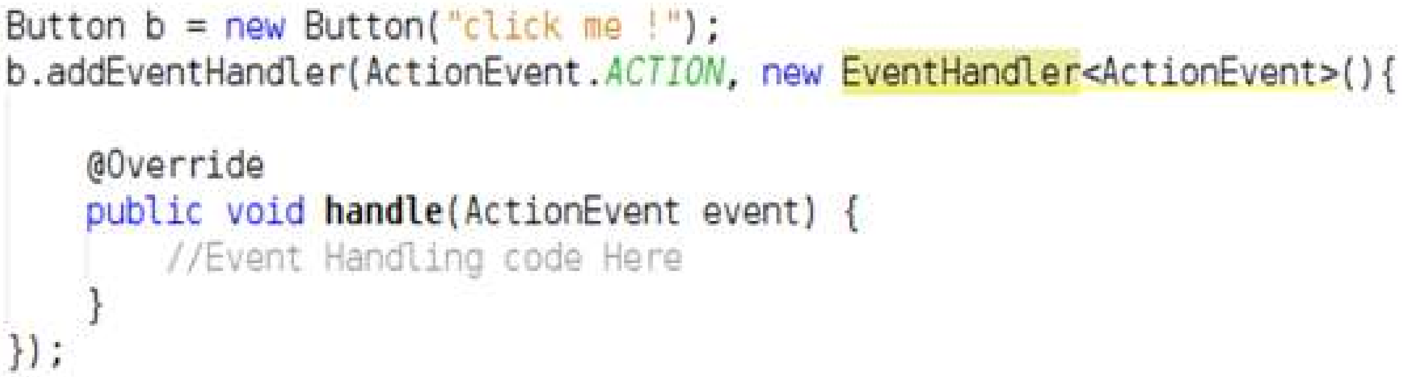
Event Handling

- To handle the event using the event property reference.



```
Button b = new Button("click me !");
b.setOnAction(new EventHandler<ActionEvent>() {

    @Override
    public void handle(ActionEvent event) {
        System.out.print("you clicked me...");
    }
});
```

- Using the addEventHandler() method.



```
Button b = new Button("click me !");
b.addEventHandler(ActionEvent.ACTION, new EventHandler<ActionEvent>(){

    @Override
    public void handle(ActionEvent event) {
        //Event Handling code Here
    }
});
```

Building UI Using JavaFX

Layouts

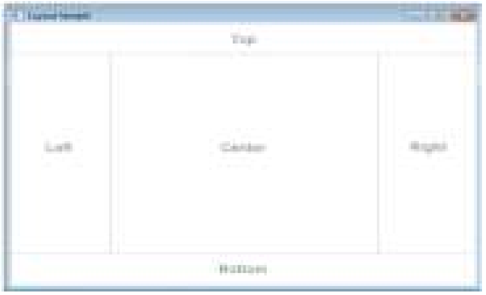
Layouts

- A JavaFX application can manually lay out the UI by setting the position and size properties for each UI element.
- JavaFX containers (Panes) are set of classes used to manage UI components positioning and size over the scene graph.
- Layout pane automatically repositions and resizes the nodes that it contains according to the properties for the nodes and the pane.

- All panes are sub-class of Node and they can be added to each other to form more complex layout.

BorderPane

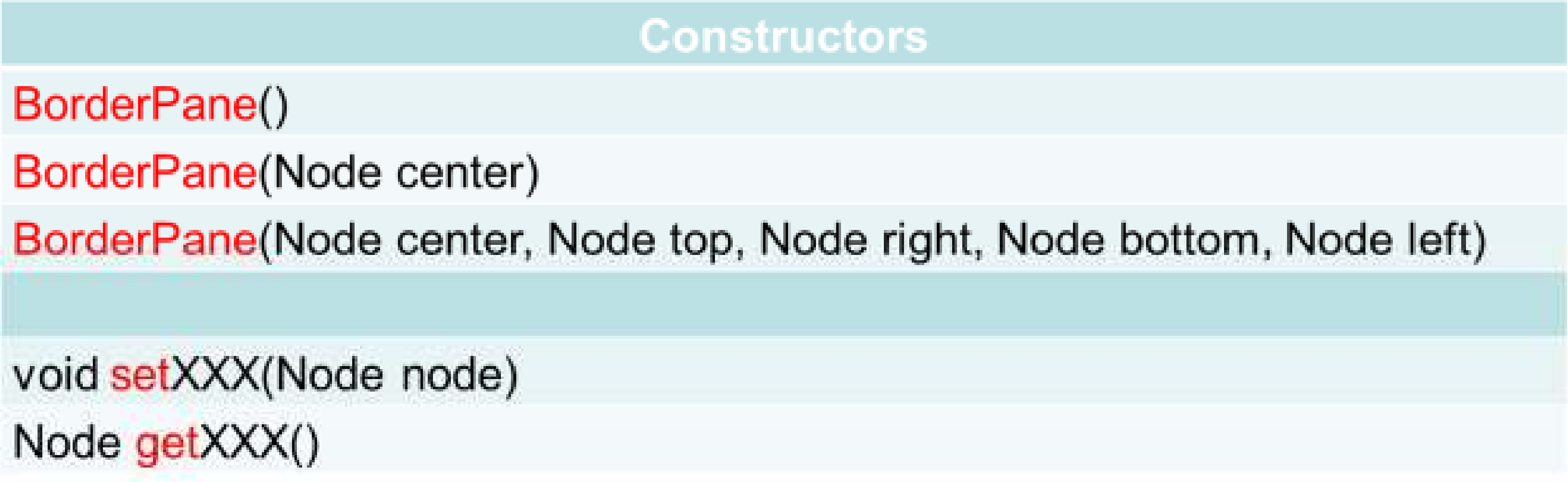
- BorderPane lays out children in top, left, right, bottom, and center positions.



- Only one node can be hosted at each position.
- The top and bottom children will be resized to their preferred heights and extend the width of the border pane.
- The left and right children will be resized to their preferred widths and extend the length between the top and bottom nodes.
- And the center node will be resized to fill the available space in the middle.
- BorderPane is commonly used as the root of a Scene.

BorderPane

- listed below are the commonly used constructors and methods of this pane:

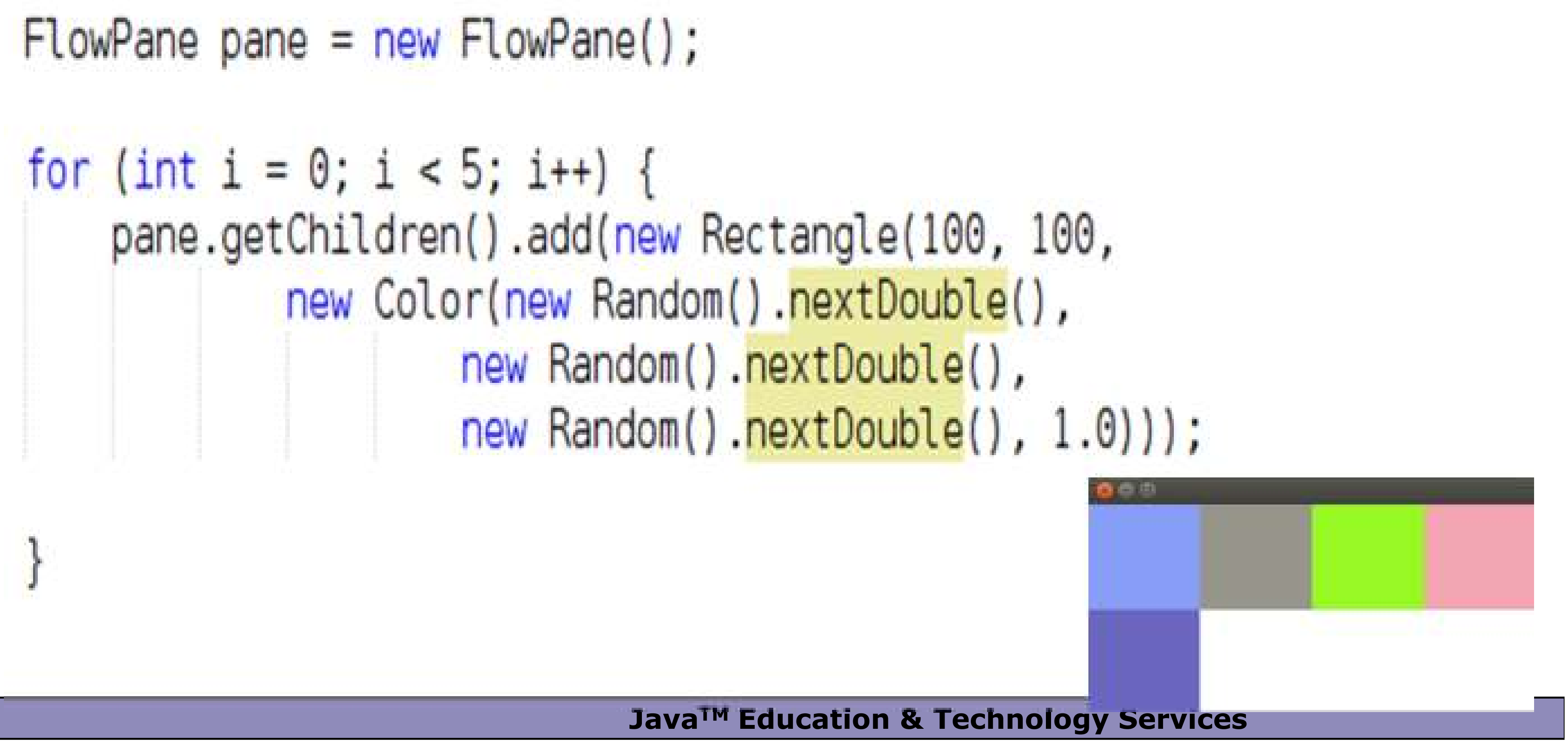


Constructors
BorderPane()
BorderPane(Node center)
BorderPane(Node center, Node top, Node right, Node bottom, Node left)
void setXXX(Node node)
Node getXXX()

- Note: XXX will be replaced with one of the pane positions (center,left,right,top,bottom).

FlowPane

- FlowPane lays out its children in a flow that wraps at the flowpane's boundary



```
FlowPane pane = new FlowPane();

for (int i = 0; i < 5; i++) {
    pane.getChildren().add(new Rectangle(100, 100,
            new Color(new Random().nextDouble(),
                    new Random().nextDouble(),
                    new Random().nextDouble(), 1.0)));

}
```

FlowPane

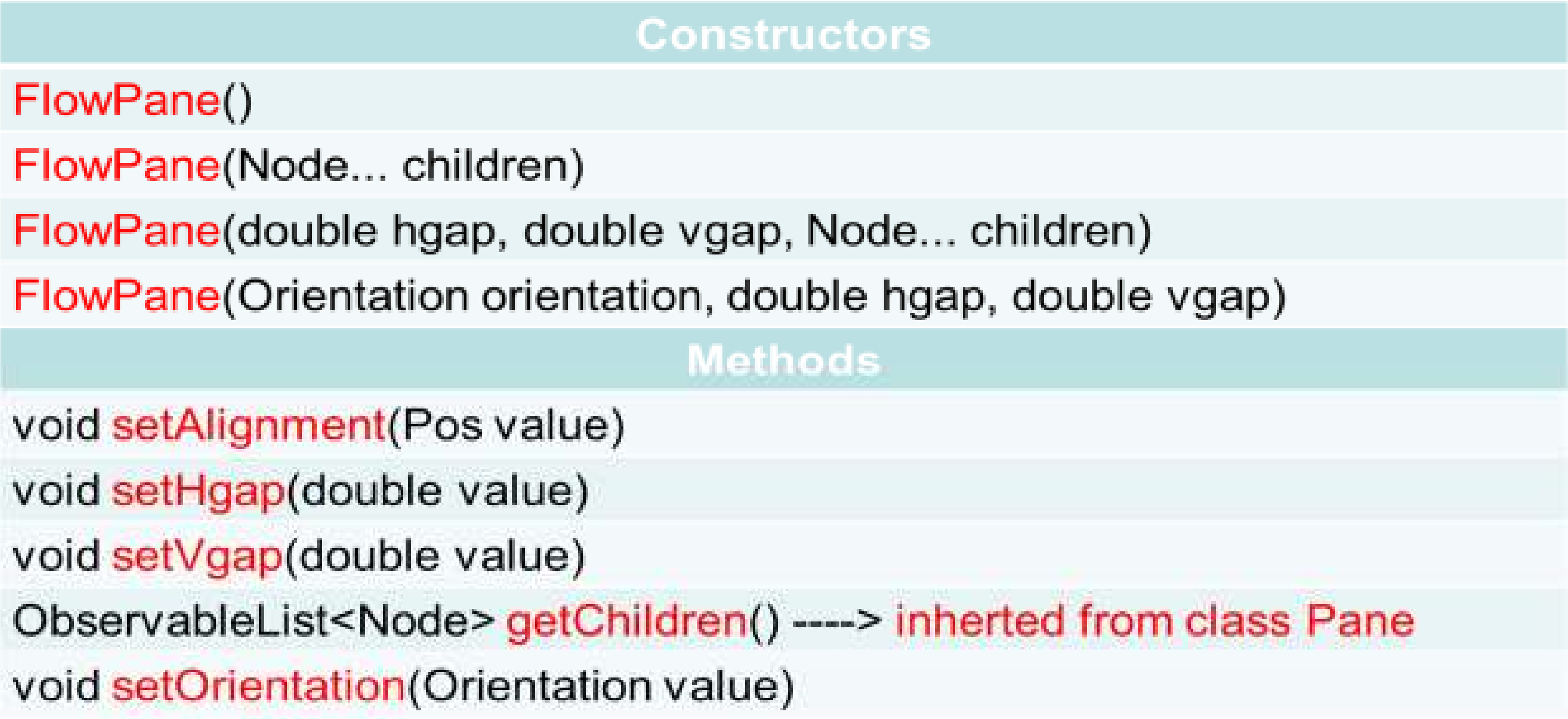
- Nodes within the FlowPane can be aligned Horizontally or Virtically depending on the alignment property value.

- Spacing between nodes can be managed using the vgap, and hgap properties.

- To add nodes to a FlowPane we use the getChildren() method to get the node list of this container, then we use add(), or addAll() method to add nodes.

FlowPane

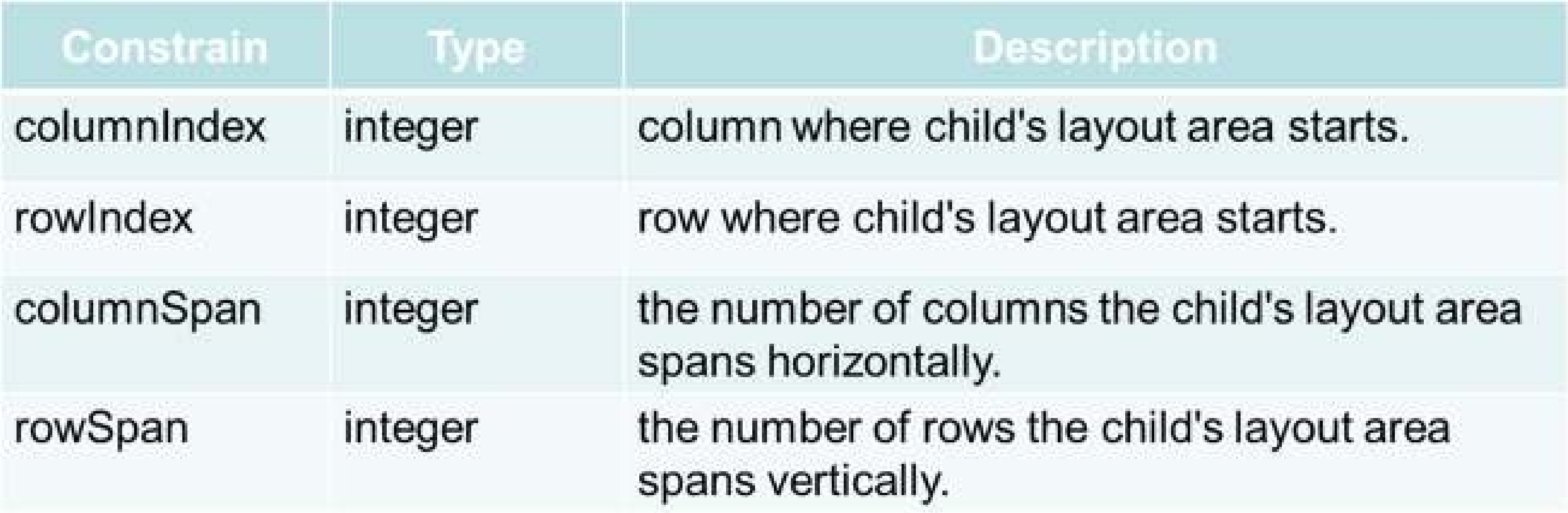
- Listed below are the commonly used constructors and methods of this pane:



Constructors
FlowPane()
FlowPane(Node... children)
FlowPane(double hgap, double vgap, Node... children)
FlowPane(Orientation orientation, double hgap, double vgap)
Methods
void setAlignment(Pos value)
void setHgap(double value)
void setVgap(double value)
ObservableList<Node> getChildren() ----> inherted from class Pane
void setOrientation(Orientation value)

GridPane

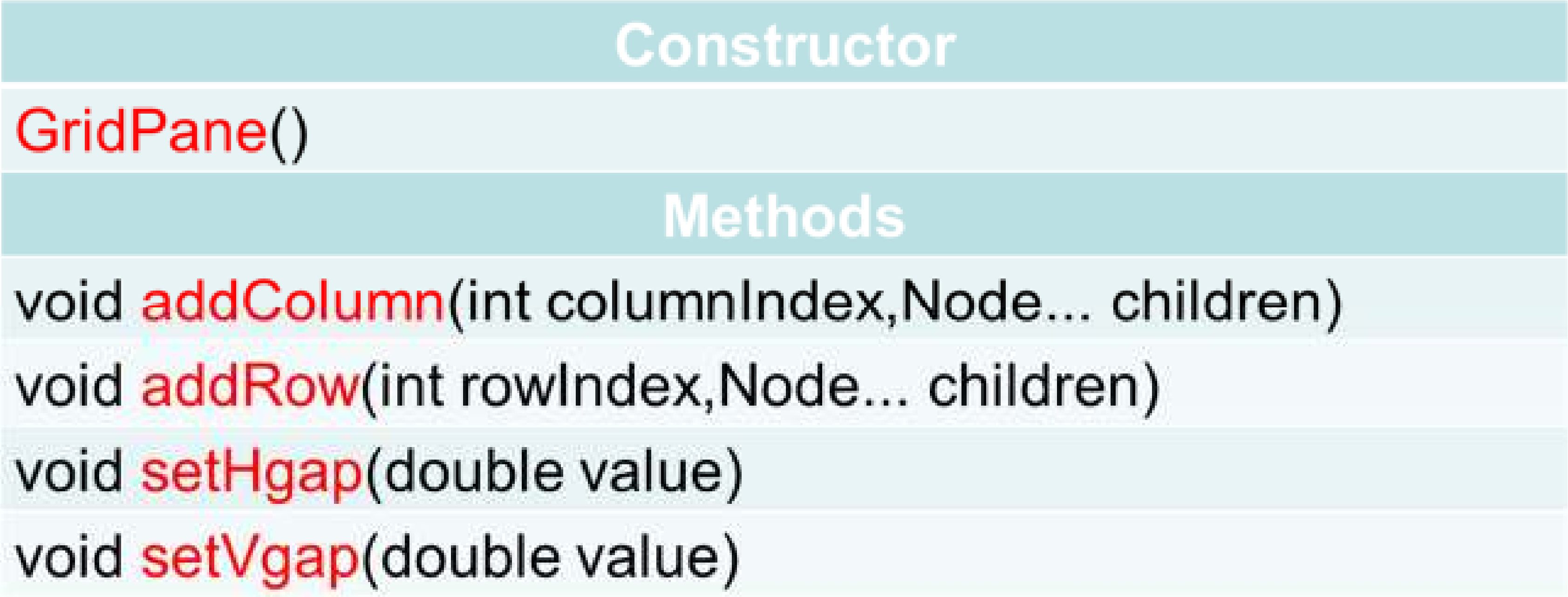
- GridPane lays out its children within a flexible grid of rows and columns.
- A child may be placed anywhere within the grid and may span multiple rows/columns.
- A child's placement within the grid is defined by it's layout constraints:



Constrain	Type	Description
columnIndex	integer	column where child's layout area starts.
rowIndex	integer	row where child's layout area starts.
columnSpan	integer	the number of columns the child's layout area spans horizontally.
rowSpan	integer	the number of rows the child's layout area spans vertically.

GridPane

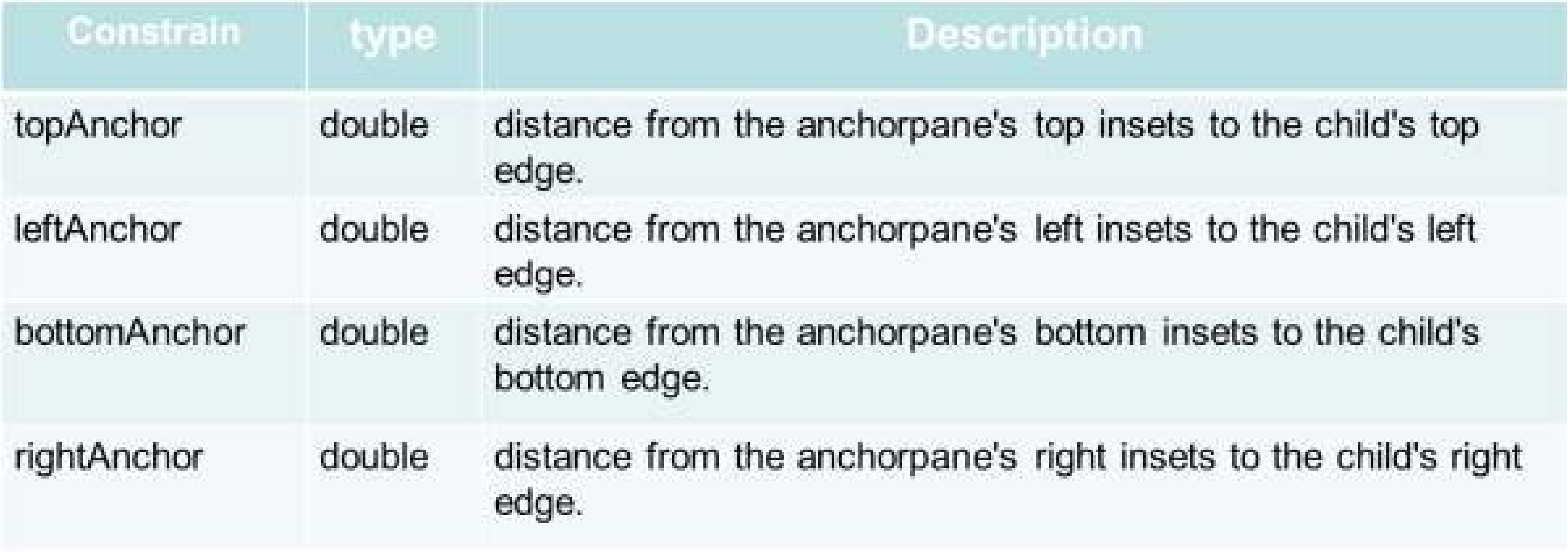
- If the row/column indices are not explicitly set, then the child will be placed in the first row/column.
- To add nodes to the GridPane we use the add(node,colIndex,rowIndex) method.



Constructor
GridPane()
Methods
void addColumn(int columnIndex,Node... children)
void addRow(int rowIndex,Node... children)
void setHgap(double value)
void setVgap(double value)

AnchorPane

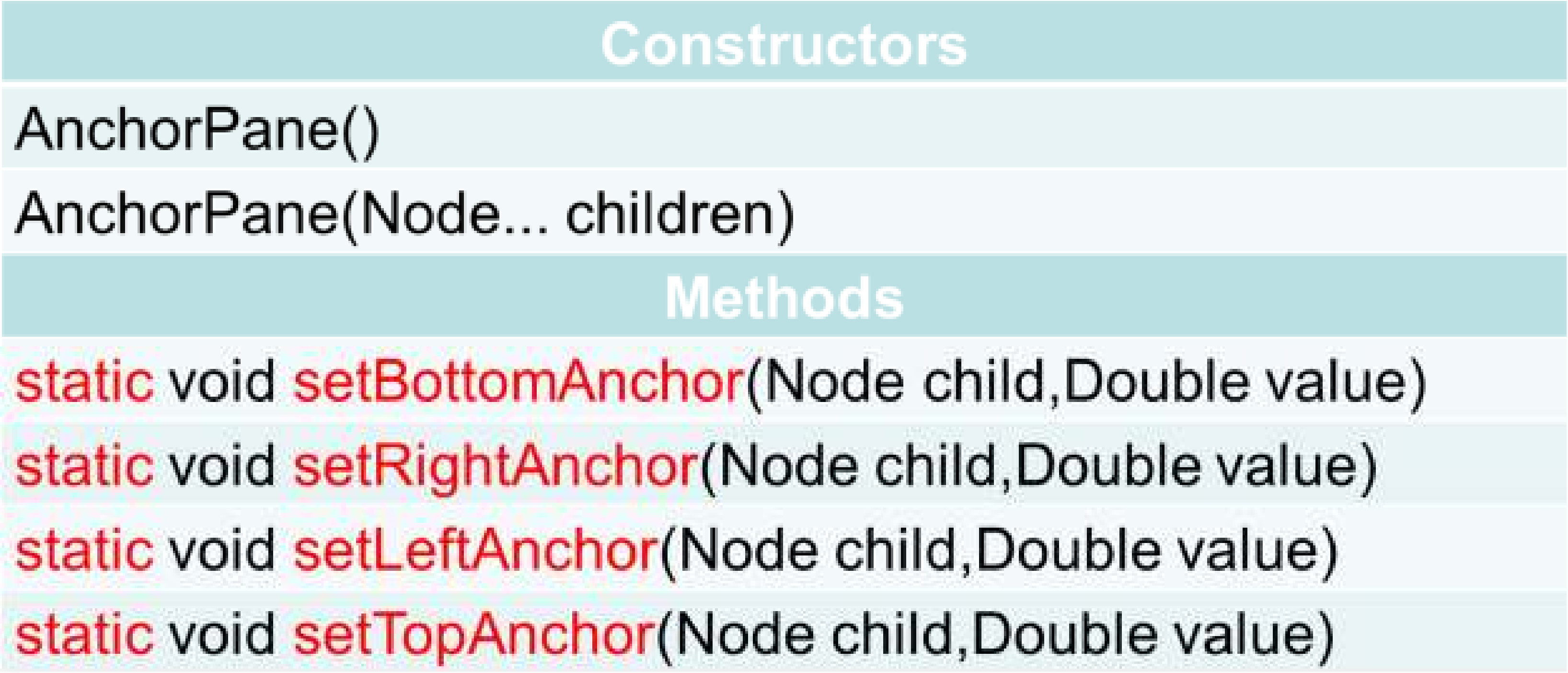
- AnchorPane allows the edges of child nodes to be anchored to an offset from the anchorpane's edges.
- If the anchorpane has a border and/or padding set, the offsets will be measured from the inside edge of those insets.
- AnchorPane has four constrains can be set for each child.



Constrain	type	Description
topAnchor	double	distance from the anchorpane's top insets to the child's top edge.
leftAnchor	double	distance from the anchorpane's left insets to the child's left edge.
bottomAnchor	double	distance from the anchorpane's bottom insets to the child's bottom edge.
rightAnchor	double	distance from the anchorpane's right insets to the child's right edge.

AnchorPane

- The following are the commonly used methods of the AnchorPane:

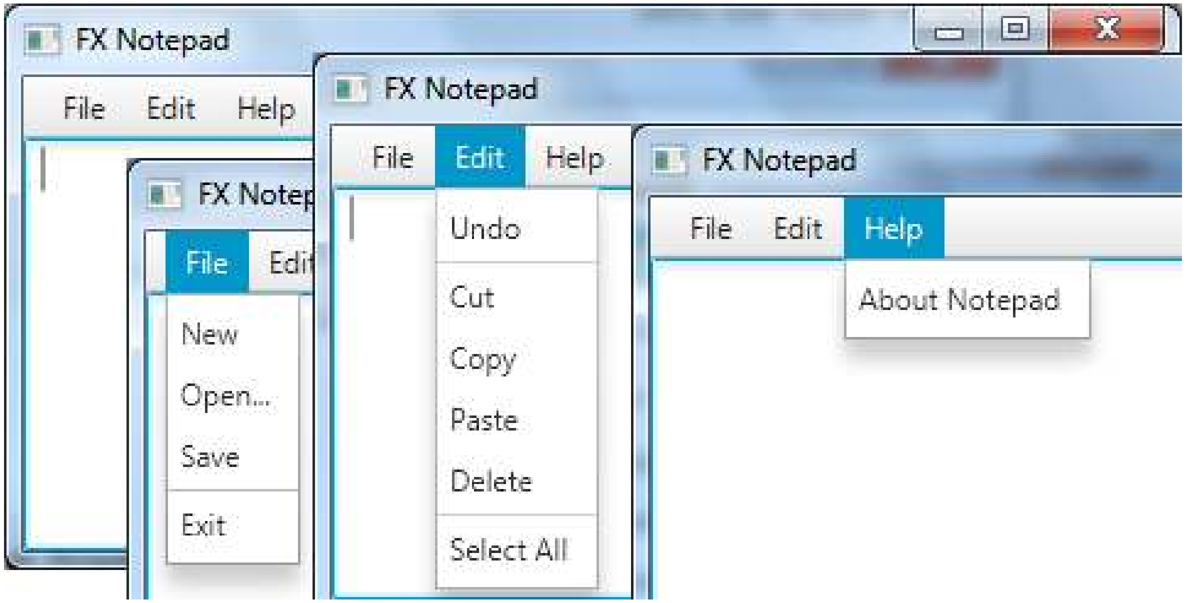


Constructors
AnchorPane()
AnchorPane(Node... children)
Methods
static void setBottomAnchor(Node child,Double value)
static void setRightAnchor(Node child,Double value)
static void setLeftAnchor(Node child,Double value)
static void setTopAnchor(Node child,Double value)

Lab Exercise

Create A GUI Desktop Application

- Create a JavaFX NotePad Desktop Application .
  - •File menu [new , open, save, Exit]
  - •Edit menu [Cut, copy, Paste, Delete, Select All]
  - •Help menu [About]



Lesson 7

Java SE 8 New Features

Lambda Expressions, Functional Interfaces, Stream API, and More

Outline

- Functional Programming Overview
- Functional Interfaces
- Lambda Expressions
- Method References
- Stream API
- Parallel data processing and performance
- Date & Time API

Java SE 8 New Features

Functional Interface

Functional Programming

- Functional programming is a style of programming (or a programming paradigm) where programs are executed by evaluating expressions.
- The output of a function is dependent on the values of its inputs [*This means that if we call a function x amount of times with the same parameters we'll get exactly the same result every time*].
- Allows us to focus on the problem rather than the code [*you specify what you want to accomplish in a task, but not how to accomplish it*]
- Functions can take functions as arguments and return functions as results.

Functional interfaces [SAM]

- Functional interfaces are new additions in java 8
- These interfaces are also called Single Abstract Method interfaces (SAM Interfaces).
- A functional interface is a Java interface with exactly one non-default method.
- Java 8 introduces an annotation i.e. @FunctionalInterface too, which can be used for compiler level errors .
- The package java.util.function defines many new useful functional interfaces.
- These can be represented using Lambda expressions, Method reference and constructor references as well.

Functional interfaces [SAM]

```
package functionalInterfaceExample;
@FunctionalInterface
public interface MyFirstFunctionalInterface {
            public void firstWork();
}
```

Functional interfaces [SAM]

```
package functionalInterfaceExample;
@FunctionalInterface
public interface MyFirstFunctionalInterface {
            public void firstWork();
            public void doSomeMoreWork();

}
```

Compilation Error

Unexpected @FunctionalInterface annotation
@FunctionalInterface MyFirstFunctionalInterface is not a functional interface
multiple non-overriding abstract methods found in interface
MyFirstFunctionalInterface

Functional interfaces [SAM]

```
package functionalInterfaceExample;
@FunctionalInterface
public interface MyFirstFunctionalInterface {
            public void firstWork();
  default public void doSomeMoreWork(){
                    ..........................................................
          }
}
```

Compilation Error

Unexpected @FunctionalInterface annotation
@FunctionalInterface MyFirstFunctionalInterface is not a functional interface
multiple non-overriding abstract methods found in interface
MyFirstFunctionalInterface

Generic Functional Interface

- The functional interface associated can be generic.

```
// Use a generic functional interface.
// A generic functional interface with two parameters
// that returns a boolean result.
@FunctionalInterface
public interface SomeTest<T>
{
   boolean test(T n, T m);
}
```

Generic Functional Interface

- The functional interface associated can be generic.

```
class GenericFunctionalInterfaceDemo
{
public static void main(String args[]) {
    SomeTest<Integer>  isFactor =new SomeTest<Integer>(){
         public boolean test(Integer n, Integer m){
              return n%m ==0;
         }
    };
   if(isFactor.test(10, 2))
          System.out.println("2 is a factor of 10");
    System.out.println();
}
```

Functional Interfaces Examples

- ❑ Package java.util.function contains several functional interfaces.
- ❑ Throughout the table, T and R are generic type names that represent the type of the object on which the functional interface operates and the return type of a method, respectively.
- ❑ The Following Tables show the six basic generic functional interfaces.

Functional Interfaces Examples

Interface	Description
BinaryOperator<T>	Contains method **apply** that takes two **T** arguments, performs an operation on them (such as a calculation) and returns a value of type **T**.
Consumer<T>	Contains method **accept** that takes a **T** argument and returns **void**. Performs a task with it's T argument, such as outputting the object, invoking a method of the object, etc.
Function<T,R>	Contains method **apply** that takes a **T** argument and returns a value of type **R**. Calls a method on the T argument and returns that method's result.

Functional Interfaces Examples

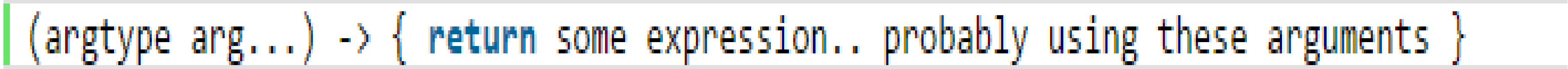
Interface	Description
Predicate<T>	Contains method `test` that takes a **T** argument and returns a **boolean**. Tests whether the T argument satisfies a condition.
Supplier<T>	Contains method `get` that takes **no arguments** and produces a value of type **T**. Often used to create a collection object in which a stream operation’s results are placed.
UnaryOperator<T>	Contains method `apply` that takes a **T** argument and returns a value of type **T**.

Java SE 8 New Features

Lambda Expressions

Lambda Expressions

- As of JDK 8, a new feature has been added to Java that profoundly enhances the expressive power of the language.
- A lambda expression is like a method: it provides a list of formal parameters and a body—an expression or block —expressed in terms of those parameters.



```
(argtype arg...) -> { return some expression.. probably using these arguments }
```

- A lambda expression is, essentially, an anonymous (that is, unnamed) method.
- However, this method is not executed on its own. Instead, it is used to implement a method defined by a functional interface.
- Lambda expressions are also commonly referred to as *closures*.

Lambda Expressions

- ❑ The lambda expression introduces a new syntax element and operator into the Java language.
- ❑ The new operator, sometimes referred to as the lambda operator or the arrow operator, is –>.
- ❑ It divides a lambda expression into two parts
  - ❑ The left side specifies any parameters required by the lambda expression.
  - ❑ On the right side is the lambda body, which specifies the actions of the lambda expression. Java defines two types of lambda bodies:
    - ❑ single expression lambda bodies
    - ❑ block of code lambda bodies

Lambda Expressions

- Examples:
- **() -> 98.6**
  - It evaluates and returns to a constant value (98.6).
  - This lambda expression takes no parameters.
  - it is similar to the following method :

```
double myMath() { return 98.6; }
```

- **() -> Math.random() * 100**
  - obtains a pseudo-random value from Math.random( ), multiplies it by 100, and returns the result.
  - It, too, does not require a parameter.
  - it is similar to the following method:

```
double myMath() { return Math.random *100; }
```

Lambda Expressions

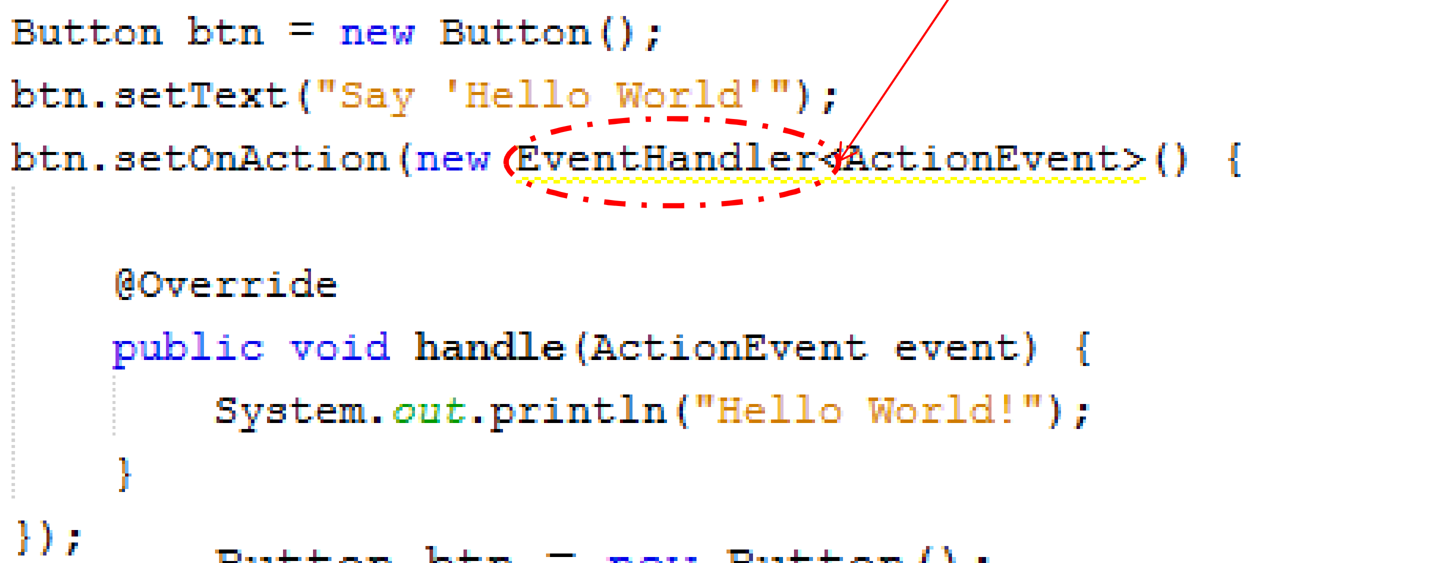
- Examples:
- **(n) -> 1.0 / n**
  - returns the reciprocal of the value of n
  - If n is 4.0 then the reciprocal is 0.25
  - The type of n not needed to explicitly specify it.
  - Like a named method, a lambda expression can specify as many parameters as needed
- **(n) -> (n % 2)==0**
  - Any valid type can be used as the return type of a lambda expression.
  - When a lambda expression has only one parameter, it is not necessary to surround the parameter name with parentheses

Lambda Expressions

- ❑ Typical Use Cases:
  - ❑ Anonymous classes (GUI listeners)
  - ❑ Implement Functional interfaces
  - ❑ Apply operation to a collection via *foreach* method

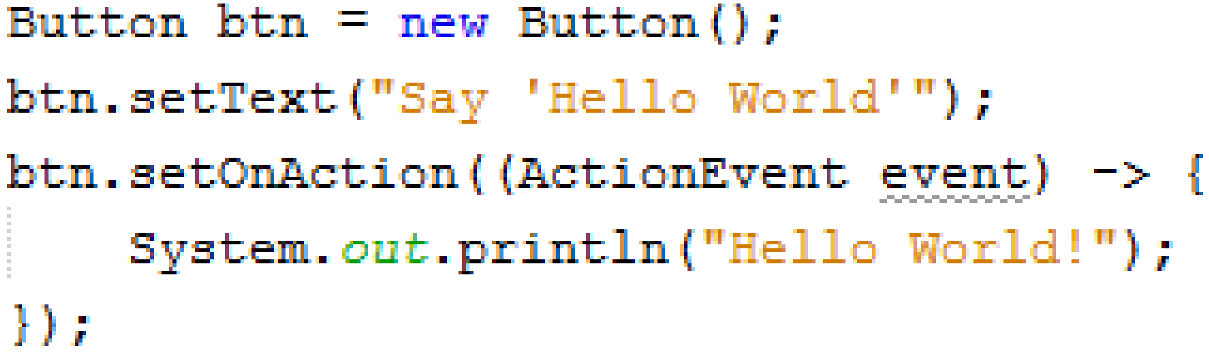
Lambda Expressions

- Typical Use Cases:
  - Anonymous classes (GUI listeners)



Functional Interface

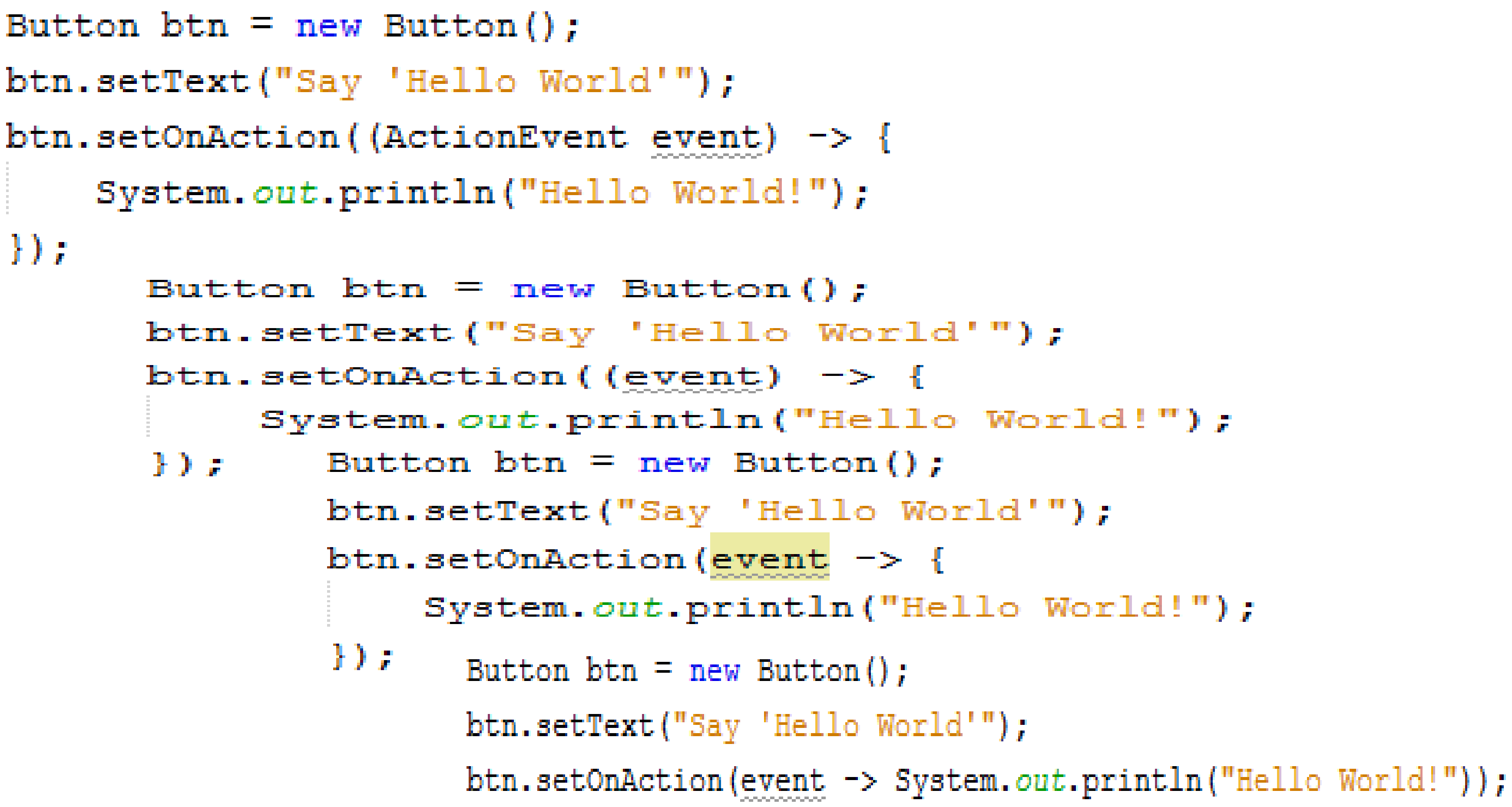
```
Button btn = new Button();
btn.setText("Say 'Hello World'");
btn.setOnAction(new EventHandler<ActionEvent>() {

    @Override
    public void handle(ActionEvent event) {
        System.out.println("Hello World!");
    }
});
```



```
Button btn = new Button();
btn.setText("Say 'Hello World'");
btn.setOnAction((ActionEvent event) -> {
    System.out.println("Hello World!");
});
```

Lambda Expressions

- ❑ Typical Use Cases:
  - ❑ Anonymous classes (GUI listeners)



```
Button btn = new Button();
btn.setText("Say 'Hello World'");
btn.setOnAction((ActionEvent event) -> {
    System.out.println("Hello World!");
});
```

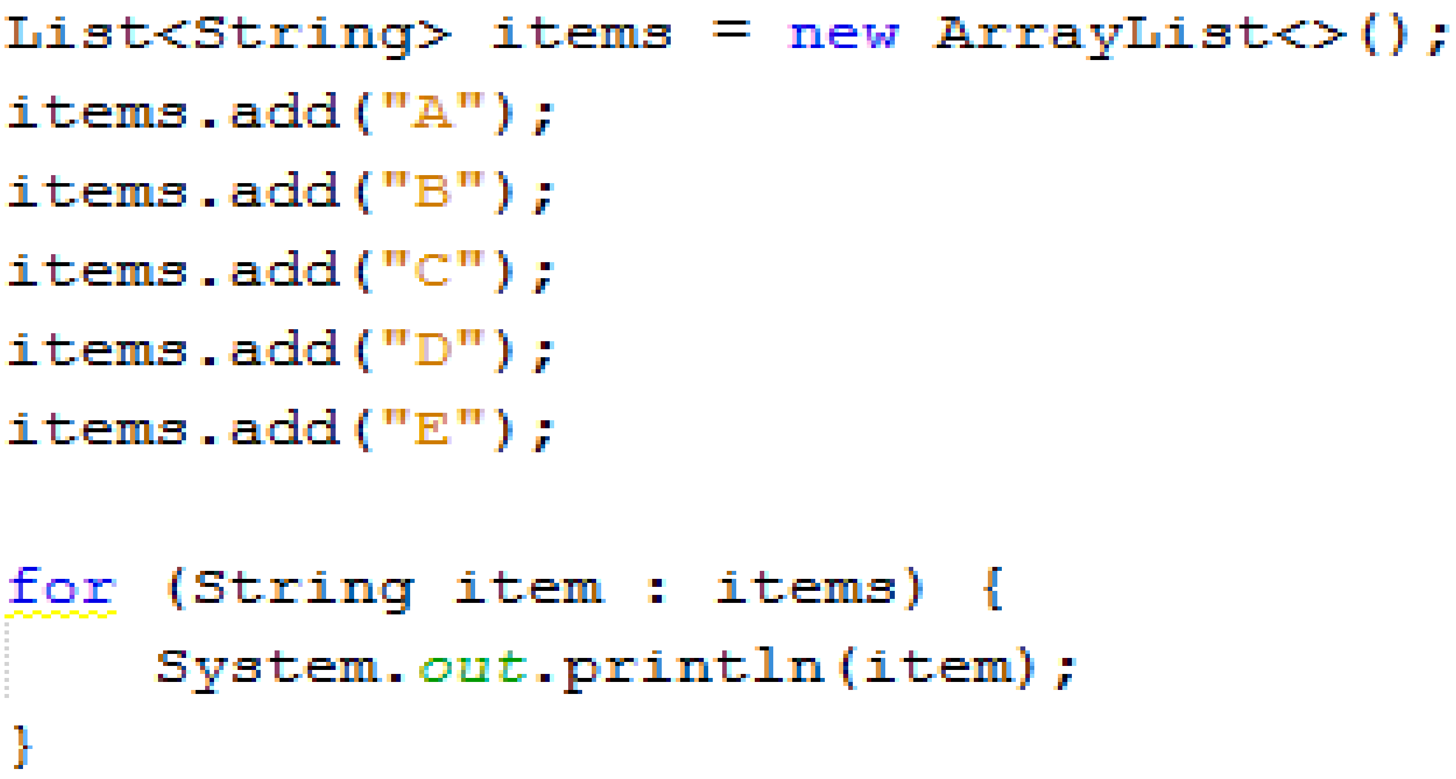
```
Button btn = new Button();
btn.setText("Say 'Hello World'");
btn.setOnAction((event) -> {
    System.out.println("Hello World!");
});
```

```
Button btn = new Button();
btn.setText("Say 'Hello World'");
btn.setOnAction(event -> {
    System.out.println("Hello World!");
});
```

```
Button btn = new Button();
btn.setText("Say 'Hello World'");
btn.setOnAction(event -> System.out.println("Hello World!"));
```

Lambda Expressions

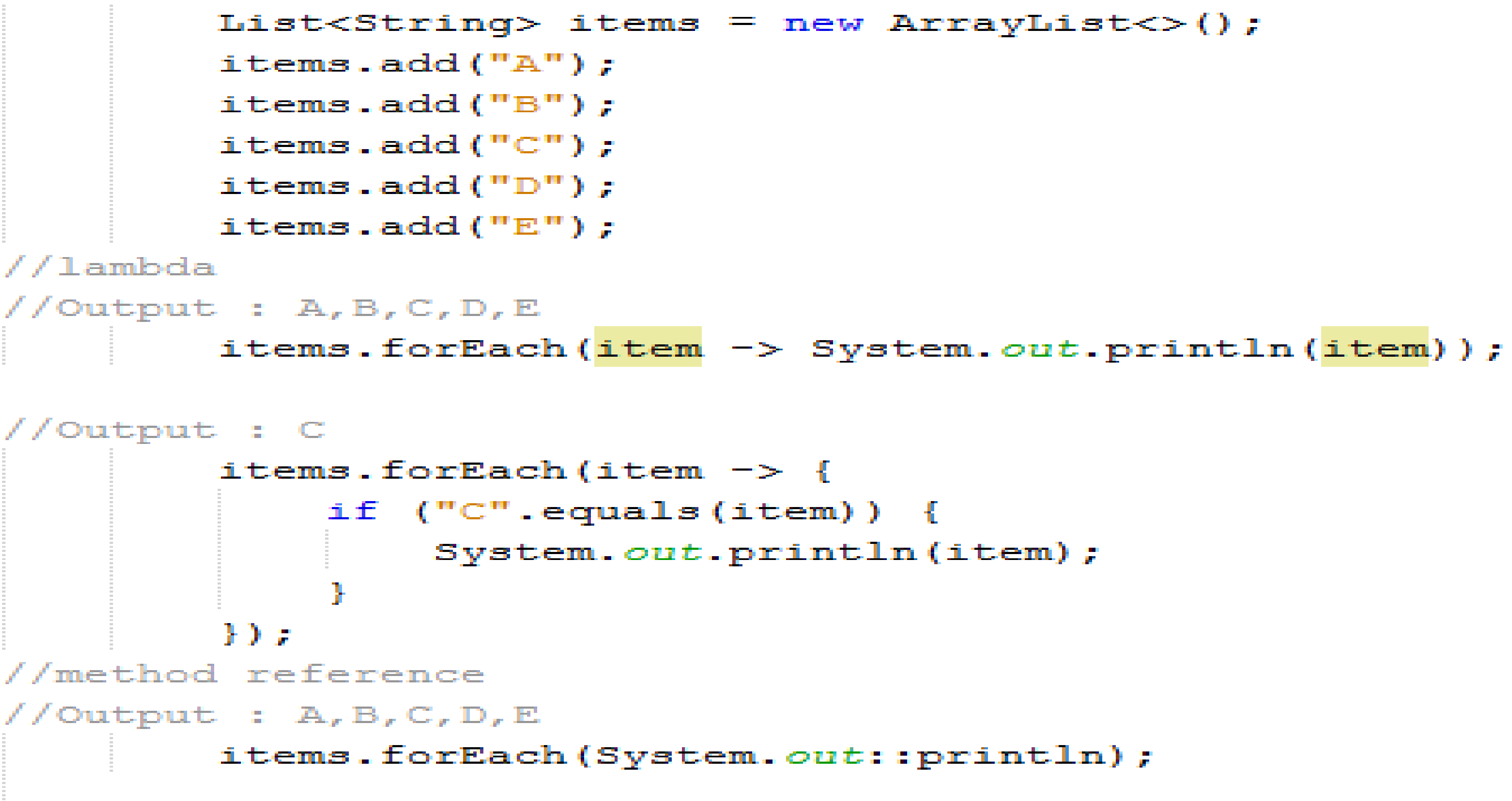
- ❑ Typical Use Cases:
  - ❑ forEach and List [Normal for-loop to loop a List]



```
List<String> items = new ArrayList<>();
items.add("A");
items.add("B");
items.add("C");
items.add("D");
items.add("E");

for (String item : items) {
    System.out.println(item);
}
```

Lambda Expressions

- ❑ Typical Use Cases:
  - ❑ forEach and List [In Java 8, you can loop a List with forEach + lambda expression or method reference]



```
        List<String> items = new ArrayList<>();
        items.add("A");
        items.add("B");
        items.add("C");
        items.add("D");
        items.add("E");
//lambda
//Output : A,B,C,D,E
        items.forEach(item -> System.out.println(item));

//Output : C
        items.forEach(item -> {
            if ("C".equals(item)) {
                System.out.println(item);
            }
        });
//method reference
//Output : A,B,C,D,E
        items.forEach(System.out::println);
```

Lambda Expressions

- ❑ Examples:

```
 // A functional interface.
@FunctionalInterface
 interface MyValue
 {
    double getValue();
 }
@FunctionalInterface
 interface MyParamValue
 {
    double getValue(double v);
 }
```

Lambda Expressions

- Examples:

```
class LambdaDemo
{
    public static void main(String args[]) {
        MyValue myVal;
        myVal = new MyValue(){
    public double getValue(){return 98.6;}
    };
    System.out.println("A constant value: " +
        myVal.getValue());
    }
}
```

```
A constant value: 98.6
```

Lambda Expressions

❑ Examples:

```
class LambdaDemo
{
    public static void main(String args[]) {
        MyValue myVal;
        myVal = () -> 98.6;



    System.out.println("A constant value:  " +
        myVal.getValue());
    }
}
```

```
A constant value: 98.6
```

Lambda Expressions

- ❑ Examples:

```java
class LambdaDemo
{
   public static void main(String args[]) {
    MyParamValue myPval = (n) -> 1.0 / n;

    System.out.println("Reciprocal of 4 is " +
                         myPval.getValue(4.0));

    System.out.println("Reciprocal of 8 is " +
                        myPval.getValue(8.0));
   }
}
```

```
Reciprocal of 4 is 0.25
Reciprocal of 8 is 0.125
```

Lambda Expressions

❑ Examples:

```
// A functional interface.
@FunctionalInterface
interface NumericTest
{
   boolean test(int n, int m);

}
```

Lambda Expressions

- ❑ Examples:

```
class LambdaDemo
{
  public static void main(String args[]) {
    NumericTest isFactor = (n, d) -> (n % d) == 0;
if(isFactor.test(10, 2)) System.out.println("2 is a factor of
10");
if(!isFactor.test(10, 3)) System.out.println("3 is not a factor
of 10");
NumericTest lessThan = (n, m) -> (n < m);
if(lessThan.test(2, 10)) System.out.println("2 is less than 10");

      }
}
```

```
2 is a factor of 10
3 is not a factor of 10
2 is less than 10
```

Lambda Expressions

- ❑ Examples:

```
 class LambdaDemo
 {
    public static void main(String args[]) {
 NumericTest absEqual = (n,m) -> (n<0 ? -n:n) == (m<0 ? -m : m);
      if(absEqual.test(4, -4))
    System.out.println("Absolute values of 4 and -4 are equal.");
      if(!lessThan.test(4, -5))
System.out.println("Absolute values of 4 and -5 are not equal.");
      }
 }
```

```
Absolute values of 4 and -4 are equal.
Absolute values of 4 and -5 are not equal.
```

Lambda Expressions

❑ Lambdas that have block bodies are sometimes referred to as *block lambdas*.

❑ A block lambda expands the types of operations that can be handled within a lambda expression because *it allows the body of the lambda to contain multiple statements*.

❑ Aside from allowing multiple statements, block lambdas are used much like the expression lambdas just discussed.

❑ One key difference, however, is that you must explicitly use a ***return*** statement to return a value.

Lambda Expressions

- ❑ Here is an example that uses a block lambda to find the smallest positive factor of an int value.
- ❑ It uses an interface called NumericFunc that has a method called func( ), which takes one int argument and returns an int result. Thus, NumericFunc supports a numeric function on values of type int.

```
// A block lambda that finds the smallest
// positive factor of an int value.
interface NumericFunc {
     int func(int n);
}
```

Lambda Expressions

```
class BlockLambdaDemo {
       public static void main(String args[]) {
// This block lambda returns the smallest positive factor of a value.
    NumericFunc    smallestF = (n) -> {
                   int result = 1;
                   // Get absolute value of n.
                   n = n < 0 ? -n : n;
                   for(int i=2; i <= n/i; i++)
                     if((n % i) == 0) {
                          result = i;
                          break;
                     }
                   return result;
           };
       System.out.println("Smallest factor of 12 is " +
smallestF.func(12));
       System.out.println("Smallest factor of 11 is " +
smallestF.func(11)); } }
```

Block Lambda Expression

Java SE 8 New Features

Method References

Method References

- ❑ What are method references?
  - ❑ A new feature in Java SE 8
  - ❑ Allows to use a method as a value
- ❑ Let see this with an example. Let’s say you want to list all directories under the current path. You will use.
  - ❑ listFiles (FileFilter) function in file class
  - ❑ FileFilter interface with accept method.
  - ❑ isDirectory function in file class

Method References

```
File fObj = new File(“.”);
File[] directories = fObj.listFiles(new FileFilter(){
      public boolean accept(File file) {
            return file.isDirectory(); }
      });
```

Filter subdirectories

Method References

```
File fObj = new File(“.”);
File[] directories = fObj.listFiles((file)->
                                   file.isDirectory();

       );
```

Filter subdirectories

Now, in Java 8 you can rewrite that code as follows:

File[] directories= new File(".").listFiles(File::isDirectory);

Method References

❑ Types of Method Reference

Type	Syntax
Reference to a static method	ClassName::staticMethodName
Reference to a constructor	ClassName::new
Reference to an instance method of an arbitrary object of a particular type	ClassName::instanceMethodName
Reference to an instance method of a particular object	objectReference::instanceMethodName

Method References

```
@FunctionalInterface
 interface StringFunction {
             String applyFunction(String s);
      }
@FunctionalInterface
interface StringConsumer {
            void consumeFunction(String s);
}
```

Method References

```java
class Utils {
 public static String transform(String st,StringFunction f)
   {
      return(f.applyFunction(st));
   }
   public static void byebye(String st, StringConsumer f)
   {
      f.consumeFunction(st);
   }
  public static String makeExciting(String s)
   {
      return(s + " ** !!");
   }
      private Utils() {}
}
```

Method References

```
class RefMethodEx2{
  public static void main(String[] args) {
      String s = "TestITI";
      // SomeClass::staticMethod
String result1 = Utils.transform(s, Utils::makeExciting);
      System.out.println("\n1 Static "+result1);
      // someObject::instanceMethod
      String prefix = "Blah @@ ";
String result2 = Utils.transform(s, prefix::concat);
System.out.println("\n2- object::instance method: “
                                             +result2);
```

Method References

```
      // SomeClass::instanceMethod
 String result3 = Utils.transform(s, String::toUpperCase);
 System.out.println("\n3- Class::instance Method:"+result3);

      // SomeClass::Constructor
 String result4 = Utils.transform("hi - > “+s, String::new);
System.out.println("\n\n\n4- Class::constructor: "+result4);
      }
} // End of
```

```
1- Static TestITI ** !!

2- object::instance method: Blah @@ TestITI

3- Class::instance Method: TESTITI

4- Class::constructor: hi - > TestITI
```

Java SE 8 New Features

Streams API

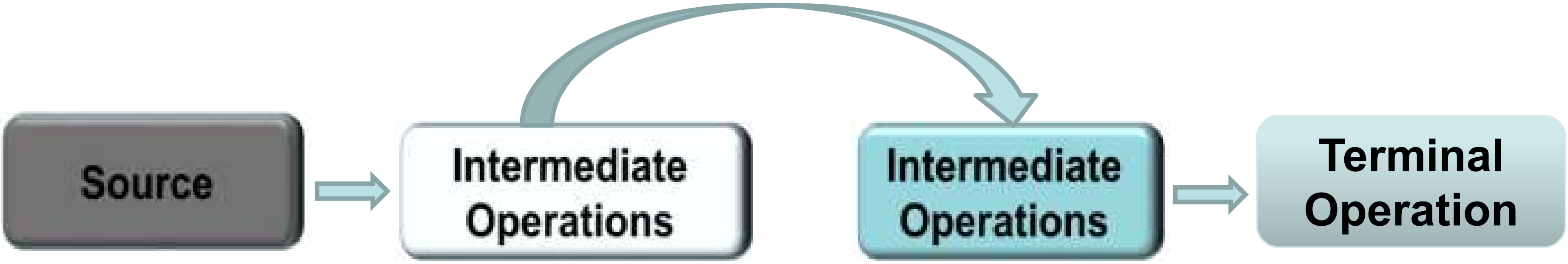
Stream API

- Java SE 8 introduces the concept of streams, which are similar to the iterators .
- Streams are objects of classes that implement interface Stream (***java.util.stream***) or one of the specialized stream interfaces for processing collections of int, long or double values.
- Together with lambdas, *streams* enable you to perform tasks on collections of elements.

Stream API: Stream Pipelines

- Streams move elements through a sequence of processing steps—known as a *stream pipeline*—that begins with a data source (such as an array or collection), performs various intermediate operations on the data source's elements and ends with a terminal operation.
- A stream pipeline *is* formed by chaining method calls. Unlike collections, *streams do not have their own storage*—once a stream is processed, it cannot be reused, because it does not maintain a copy of the original data source.

Stream API: Stream Pipelines



Source

Intermediate Operations

Intermediate Operations

Terminal Operation

```
.stream()
   .filter(b -> b.getColor()== red)
   .mapToInt( b -> b.getWeight())
   .sum();
```

Stream API: Intermediate Operations

- An intermediate operation specifies tasks to perform on the stream’s elements and always results in a new stream.
- Intermediate operations are *lazy*—they aren’t performed until a terminal operation is invoked.
- This allows library developers to optimize stream processing performance.

Stream API: Intermediate Operations

- For example, if you have a collection of 1,000,000 Person objects and you’re looking for the first one with the last name “Abbas", stream processing can terminate as soon as the first such Person object is found.

Stream API

Common Intermediate Stream Operations

Operation	Description
`filter`	Results in a stream containing only the elements that satisfy a condition.
`distinct`	Results in a stream containing only the unique elements.
`limit`	Results in a stream with the specified number of elements from the beginning of the original stream.
`map`	Results in a stream in which each element of the original stream is mapped to a new value (possibly of a different type). The new stream has the same number of elements as the original stream.
`sorted`	Results in a stream in which the elements are in sorted order. The new stream has the same number of elements as the original stream.

Stream API: Terminal Operations

- A terminal operation initiates processing of a stream pipeline’s intermediate operations and produces a result.
- Terminal operations are *eager*—they perform the requested operation when they are called.

Stream API: Terminal Operations

- Loop
  - `forEach`- Performs processing on every element in a stream (e.g., display each element).
- Reduction operations
  - Take all values in the stream and return a single value
- Mutable reduction operations
  - Create a container (such as a collection)
- Search operations
  - Performs different search or match operations on a stream

Stream API

Common Terminal Stream Operations

Operation	Description
`average`	**Calculates the average of the elements in a numeric stream.**
`count`	**Returns the number of elements in the stream.**
`max`	**Locates the largest value in a numeric stream.**
`min`	**Locates the smallest value in a numeric stream.**
`reduce`	**Reduces the elements of a collection to a single value using an associative accumulation function (e.g., a lambda that adds two elements).**

Stream API

Common Terminal Stream Operations

Operation	Description
`collect`	Creates a new collection of elements containing the results of the stream’s prior operations.
`toArray`	Creates an array containing the results of the stream’s prior operations..
`findFirst`	Finds the first stream element based on the prior intermediate operations; immediately terminates processing of the stream pipeline once such an element is found.

Stream API

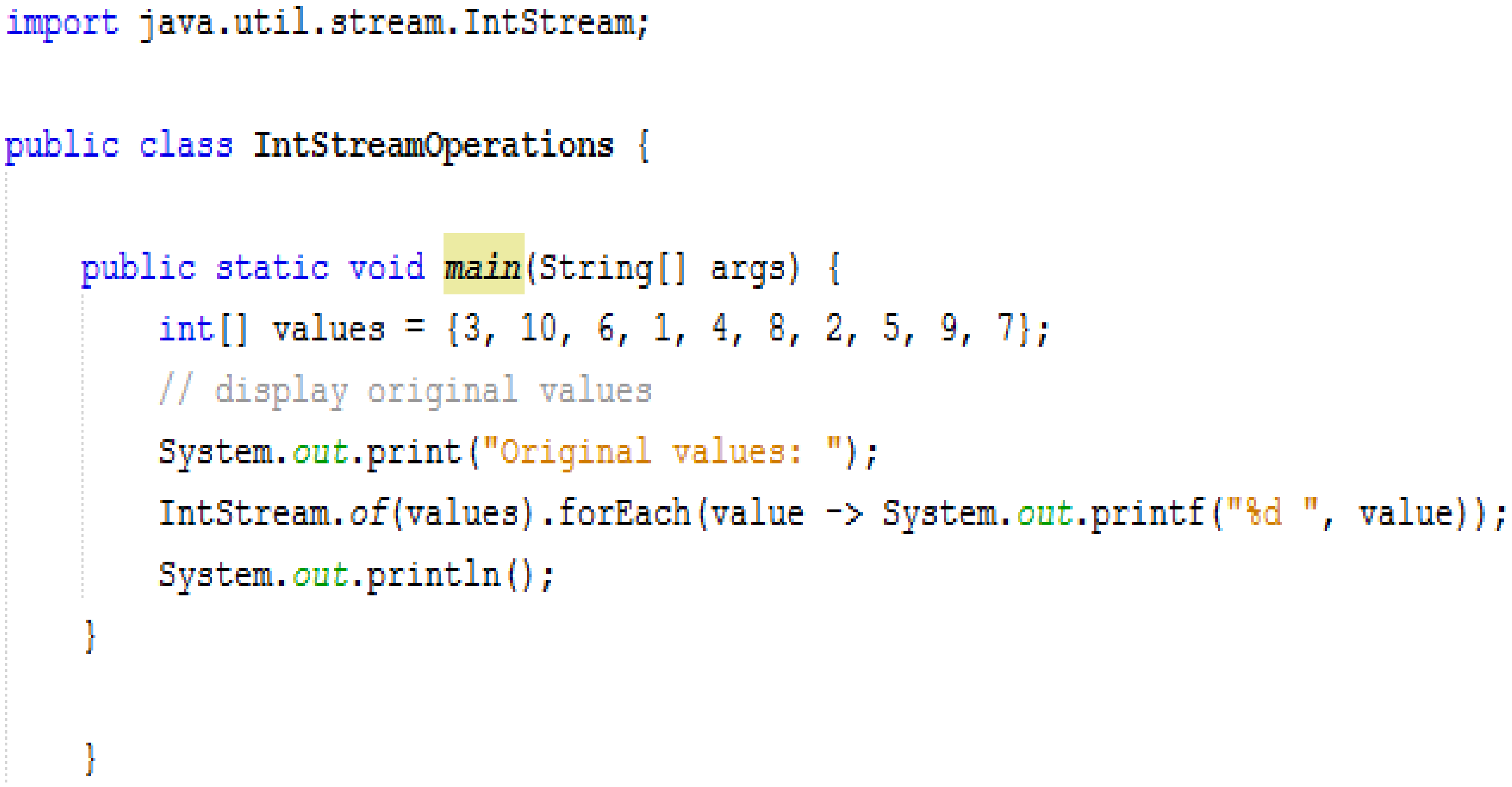
Common Terminal Stream Operations

Operation	Description
`findAny`	Finds any stream element based on the prior intermediate operations; immediately terminates processing of the stream pipeline once such an element is found.
`anyMatch`	Determines whether any stream elements match a specified condition; immediately terminates processing of the stream pipeline if an element matches.
`allMatch`	Determines whether all of the elements in the stream match a specified condition.

Stream API Example

- In the following example we will show how to use Stream API and lambda expressions to simplify programming tasks .
- The Example demonstrates operations on an IntStream (package java.util.stream)—a specialized stream for manipulating ***int*** values.
- The techniques shown in this example also apply to LongStreams and DoubleStreams for long and double values, respectively.

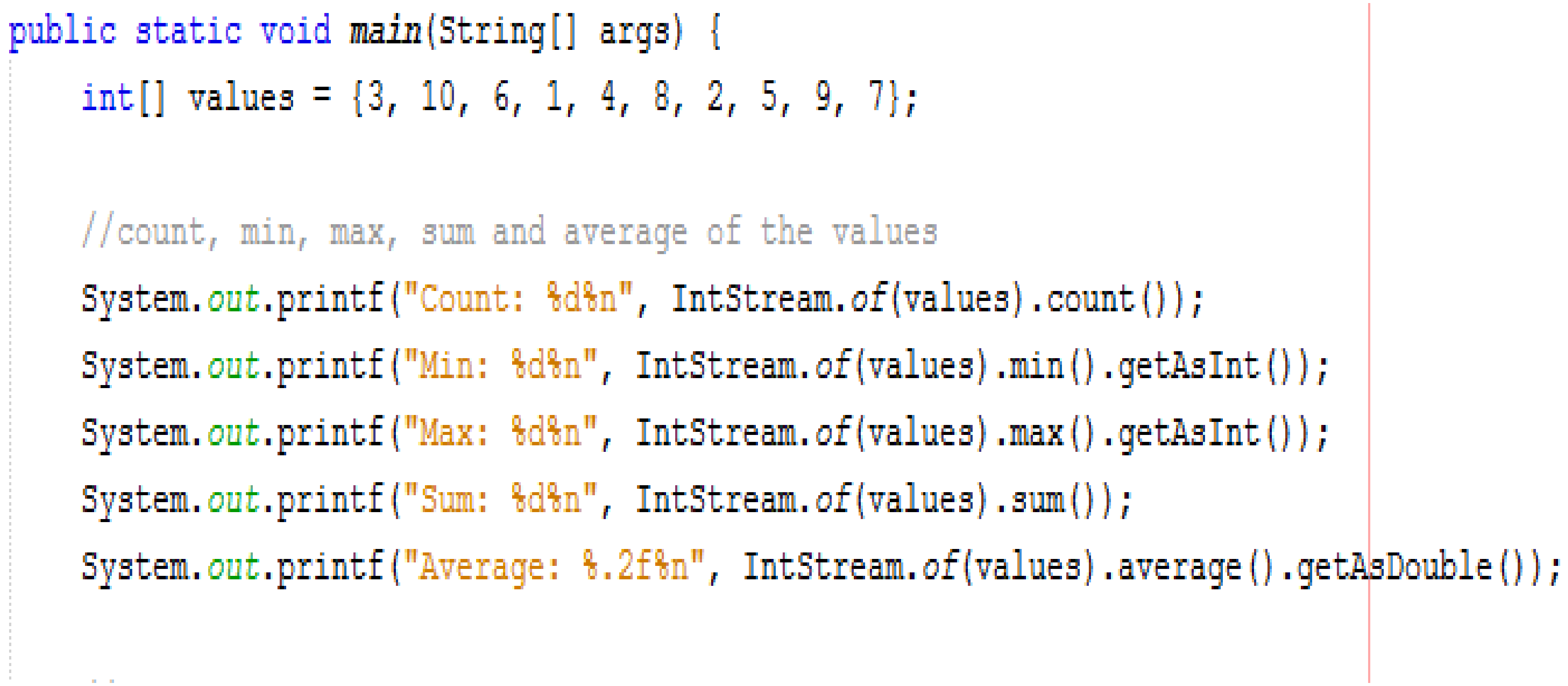
Stream API Example



```
import java.util.stream.IntStream;

public class IntStreamOperations {

    public static void main(String[] args) {
        int[] values = {3, 10, 6, 1, 4, 8, 2, 5, 9, 7};
        // display original values
        System.out.print("Original values: ");
        IntStream.of(values).forEach(value -> System.out.printf("%d ", value));
        System.out.println();
    }

    }
```

Original values: 3 10 6 1 4 8 2 5 9 7

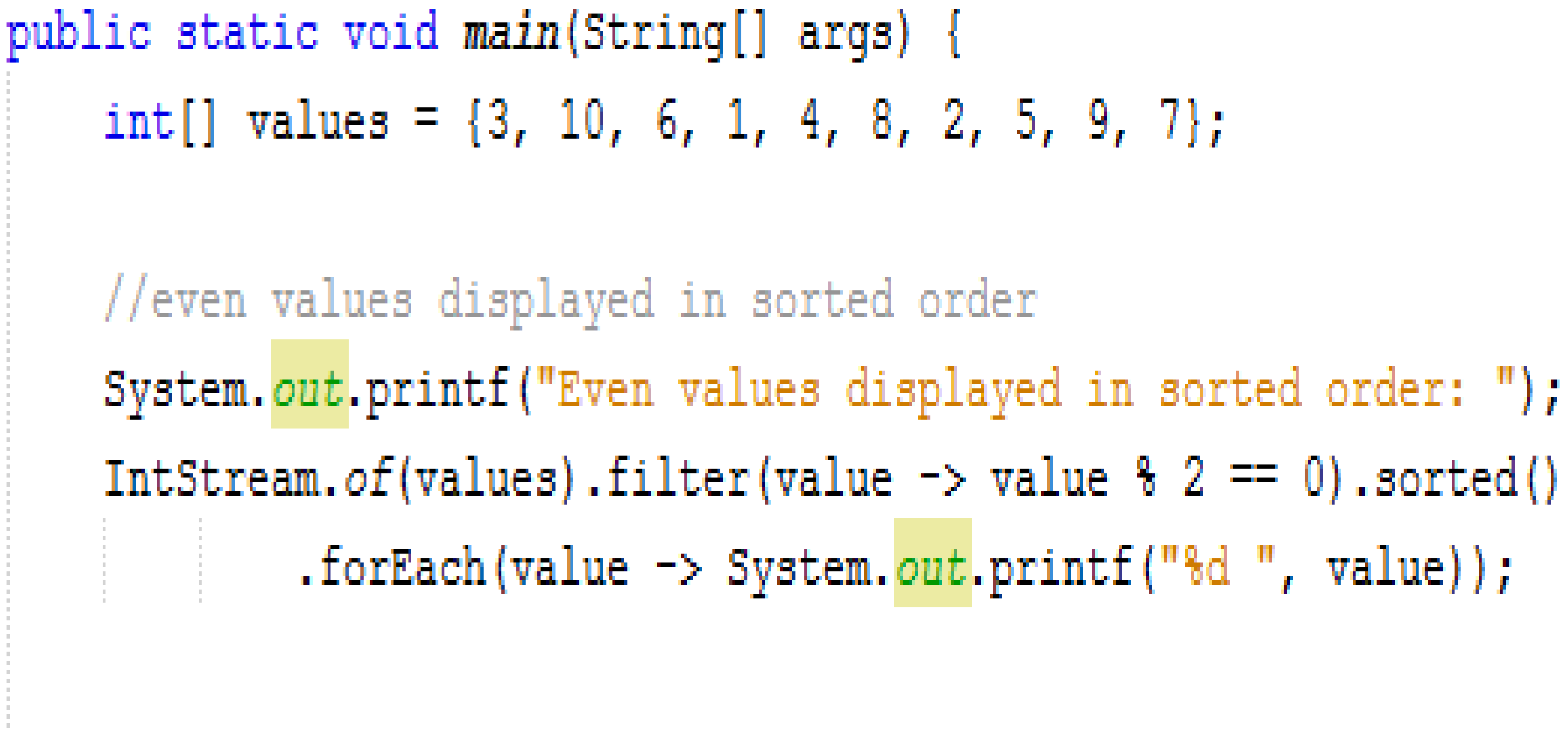
Stream API Example



```java
public static void main(String[] args) {
    int[] values = {3, 10, 6, 1, 4, 8, 2, 5, 9, 7};

    //count, min, max, sum and average of the values
    System.out.printf("Count: %d%n", IntStream.of(values).count());
    System.out.printf("Min: %d%n", IntStream.of(values).min().getAsInt());
    System.out.printf("Max: %d%n", IntStream.of(values).max().getAsInt());
    System.out.printf("Sum: %d%n", IntStream.of(values).sum());
    System.out.printf("Average: %.2f%n", IntStream.of(values).average().getAsDouble());

    ..
```

```
Count: 10
Min: 1
Max: 10
Sum: 55
Average: 5.50
```

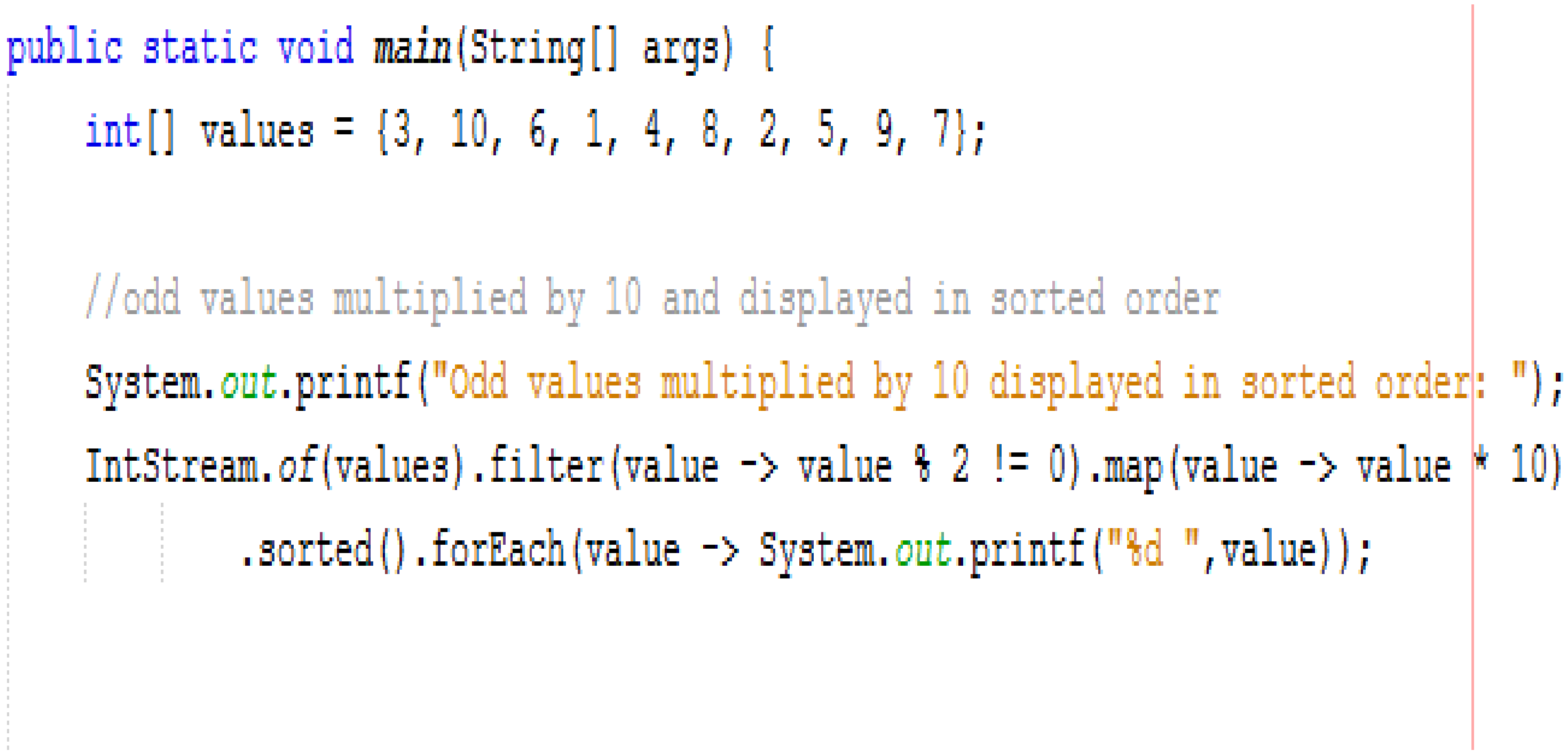
Stream API Example



```java
public static void main(String[] args) {
    int[] values = {3, 10, 6, 1, 4, 8, 2, 5, 9, 7};

    //even values displayed in sorted order
    System.out.printf("Even values displayed in sorted order: ");
    IntStream.of(values).filter(value -> value % 2 == 0).sorted()
            .forEach(value -> System.out.printf("%d ", value));
```

```
Even values displayed in sorted order: 2 4 6 8 10
```

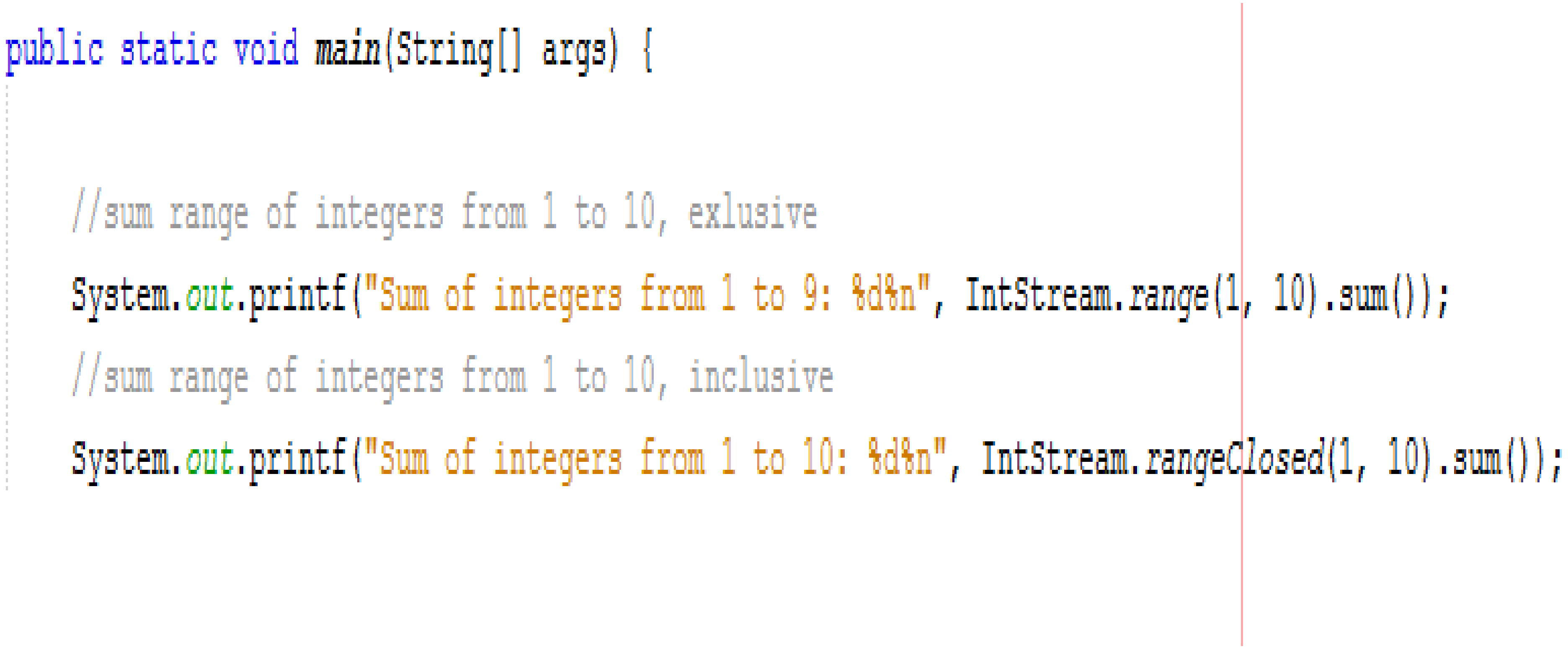
Stream API Example



```
public static void main(String[] args) {
    int[] values = {3, 10, 6, 1, 4, 8, 2, 5, 9, 7};

    //odd values multiplied by 10 and displayed in sorted order
    System.out.printf("Odd values multiplied by 10 displayed in sorted order: ");
    IntStream.of(values).filter(value -> value % 2 != 0).map(value -> value * 10)
            .sorted().forEach(value -> System.out.printf("%d ",value));
```

Odd values multiplied by 10 displayed in sorted order: 10 30 50 70 90

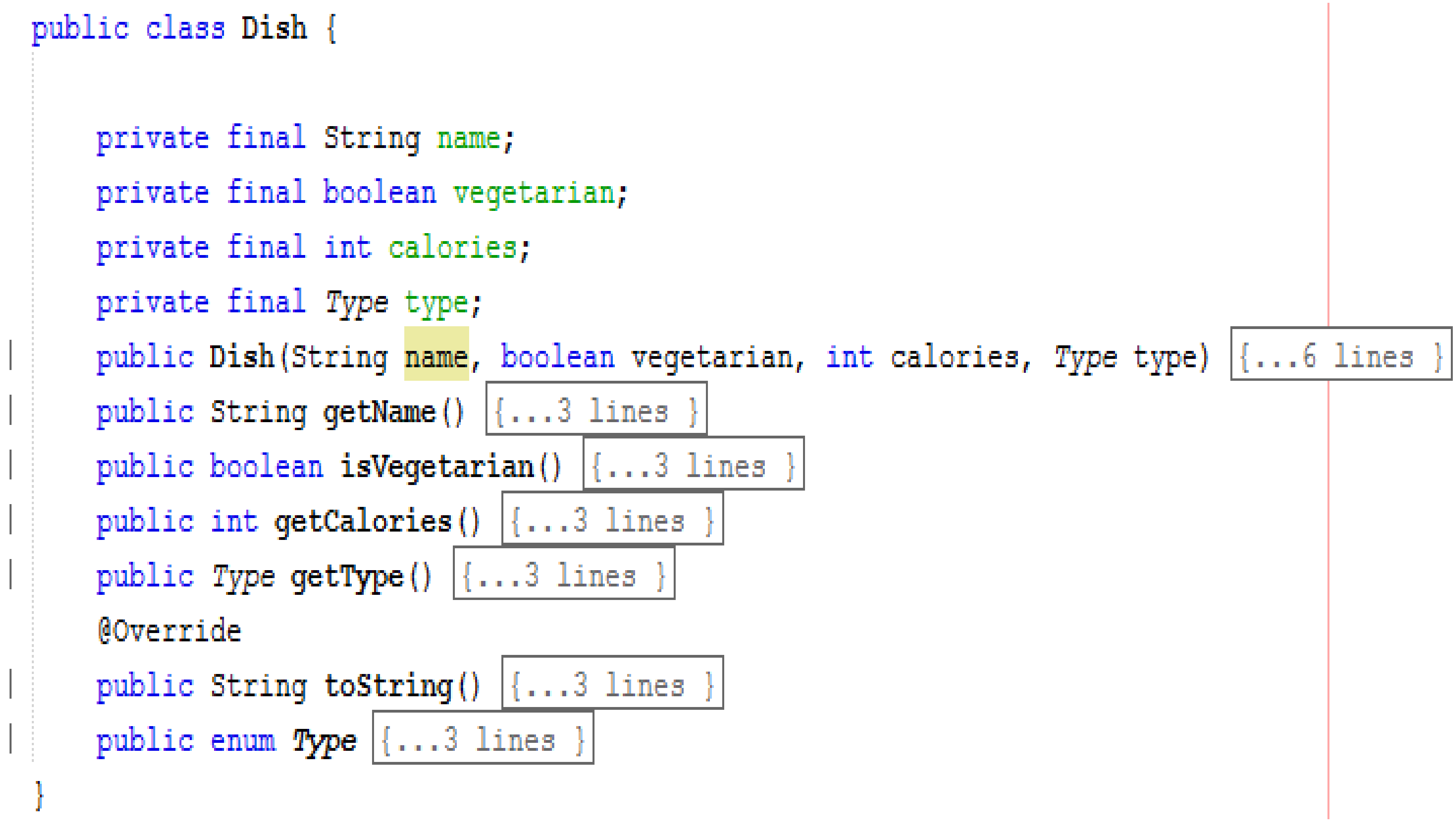
Stream API Example



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


    //sum range of integers from 1 to 10, exlusive
    System.out.printf("Sum of integers from 1 to 9: %d%n", IntStream.range(1, 10).sum());
    //sum range of integers from 1 to 10, inclusive
    System.out.printf("Sum of integers from 1 to 10: %d%n", IntStream.rangeClosed(1, 10).sum());
```

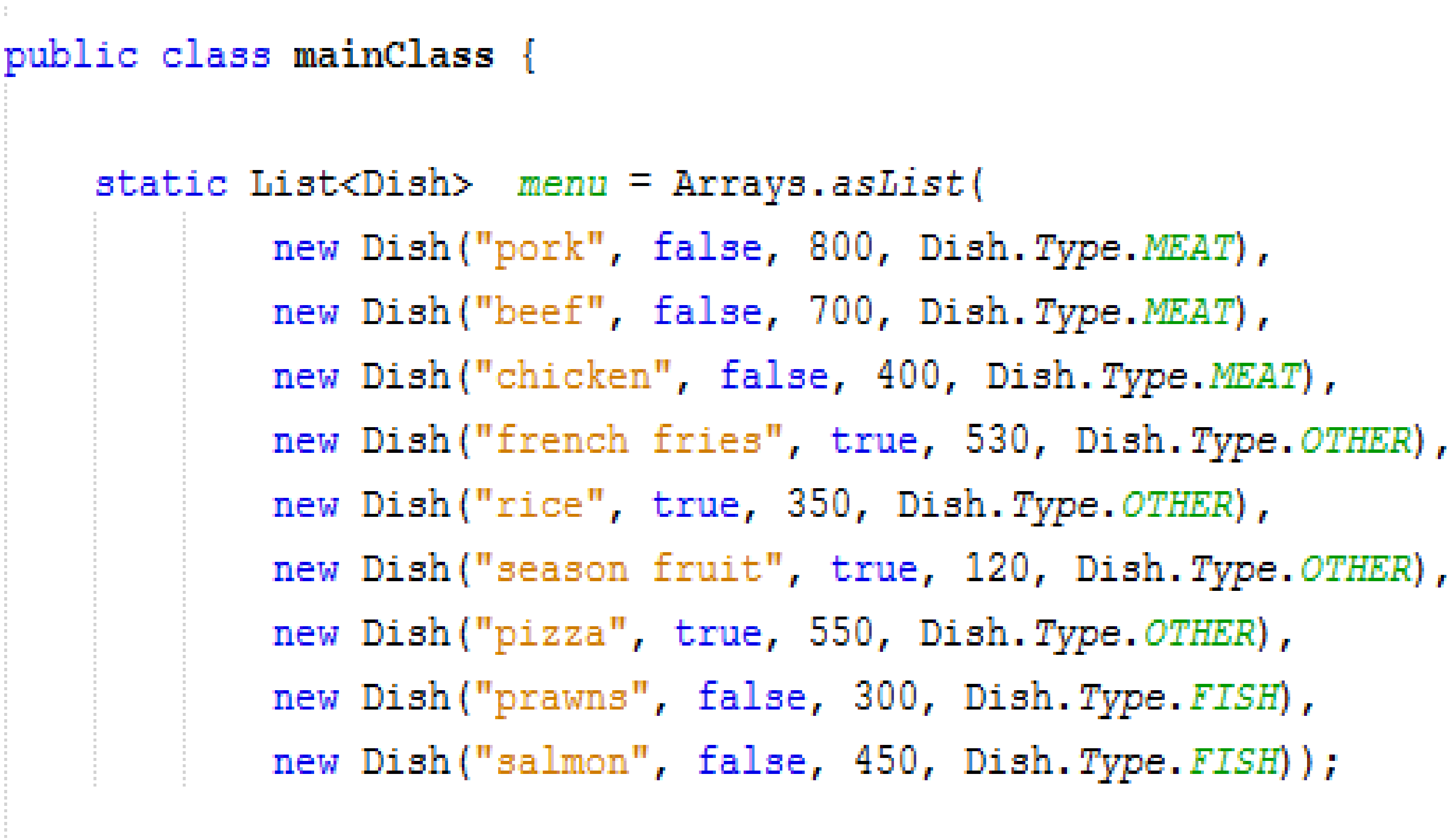
```
Sum of integers from 1 to 9: 45
Sum of integers from 1 to 10: 55
```

Stream API Example



```
public class Dish {

    private final String name;
    private final boolean vegetarian;
    private final int calories;
    private final Type type;
    public Dish(String name, boolean vegetarian, int calories, Type type) {...6 lines }
    public String getName() {...3 lines }
    public boolean isVegetarian() {...3 lines }
    public int getCalories() {...3 lines }
    public Type getType() {...3 lines }
    @Override
    public String toString() {...3 lines }
    public enum Type {...3 lines }
}
```

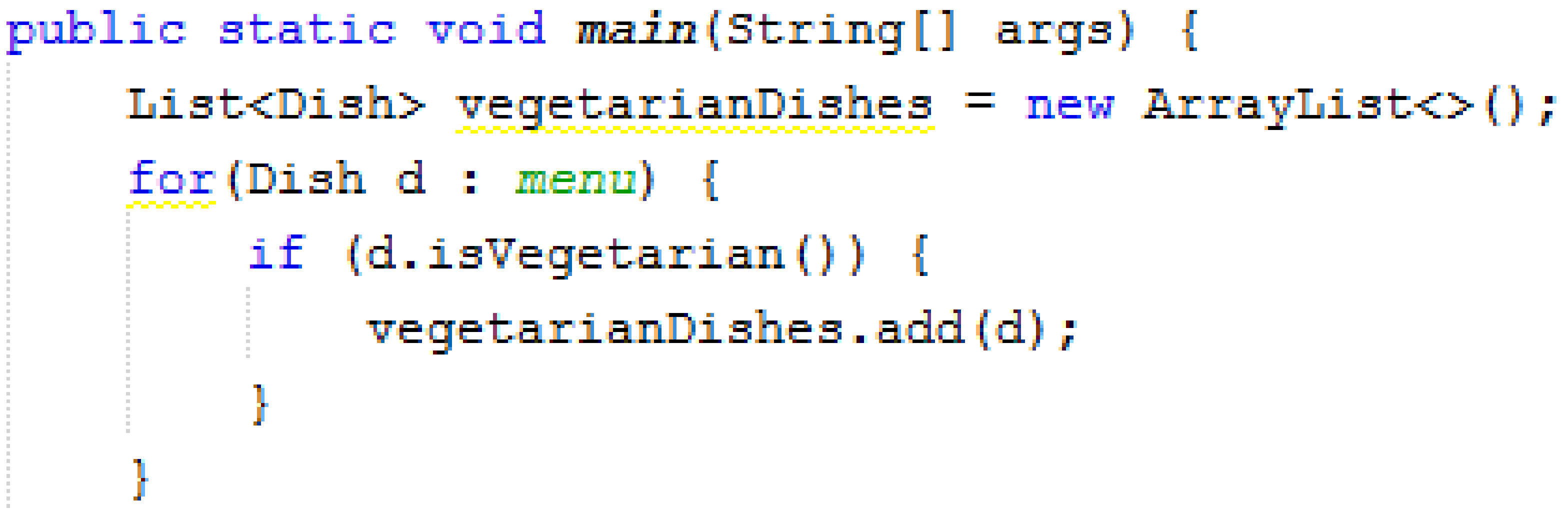
Stream API Example



```
public class mainClass {

    static List<Dish>  menu = Arrays.asList(
            new Dish("pork", false, 800, Dish.Type.MEAT),
            new Dish("beef", false, 700, Dish.Type.MEAT),
            new Dish("chicken", false, 400, Dish.Type.MEAT),
            new Dish("french fries", true, 530, Dish.Type.OTHER),
            new Dish("rice", true, 350, Dish.Type.OTHER),
            new Dish("season fruit", true, 120, Dish.Type.OTHER),
            new Dish("pizza", true, 550, Dish.Type.OTHER),
            new Dish("prawns", false, 300, Dish.Type.FISH),
            new Dish("salmon", false, 450, Dish.Type.FISH));
```

Stream API Example

- To make a list of vegetarian Dishes without Stream API



```java
public static void main(String[] args) {
    List<Dish> vegetarianDishes = new ArrayList<>();
    for(Dish d : menu) {
        if (d.isVegetarian()) {
            vegetarianDishes.add(d);
        }
    }
```

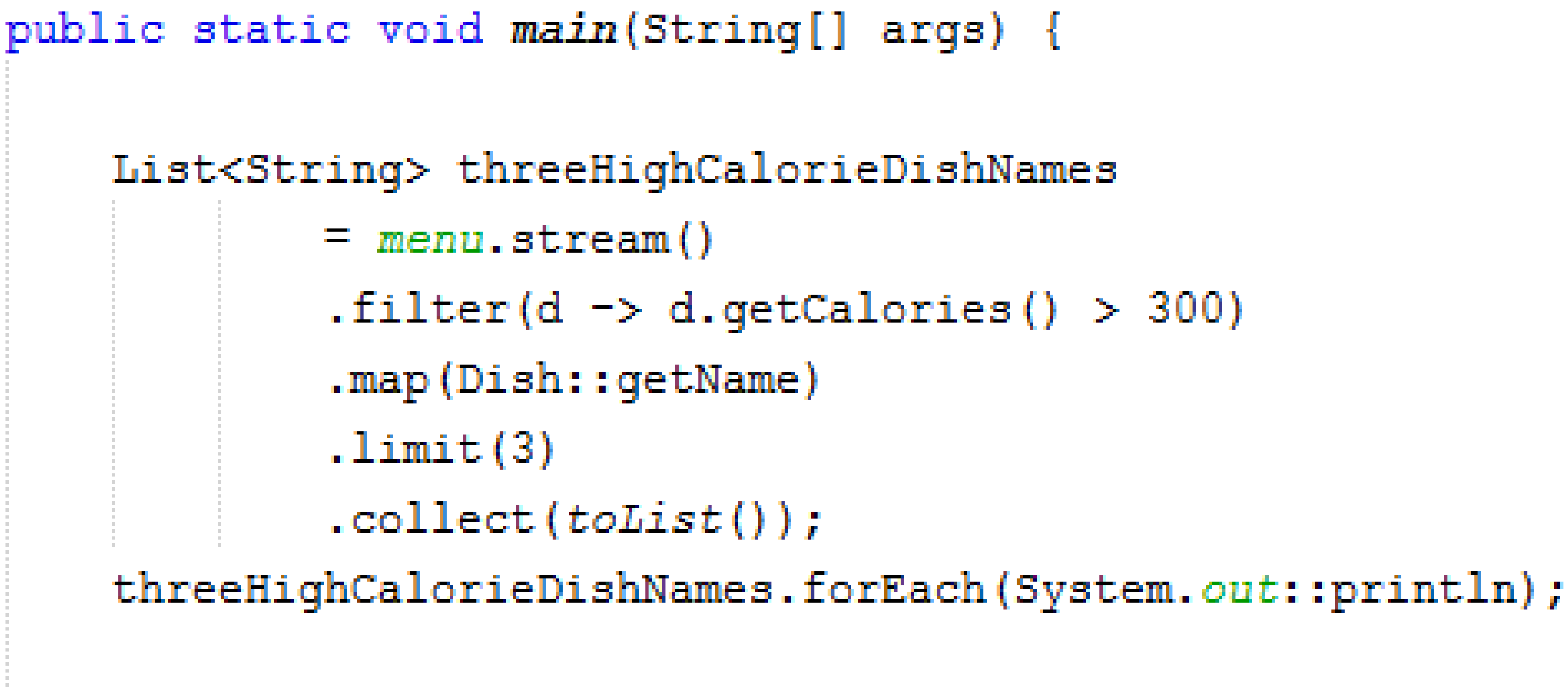
Stream API Example

- To make a list of vegetarian Dishes with Stream API

```
import static java.util.stream.Collectors.toList;


List<Dish> vegetarianDishes =
     menu.stream()
          .filter(Dish::isVegetarian)
          .collect(toList());
```

Stream API Example

- To make a list of the three High Calorie Dish Names with Stream API

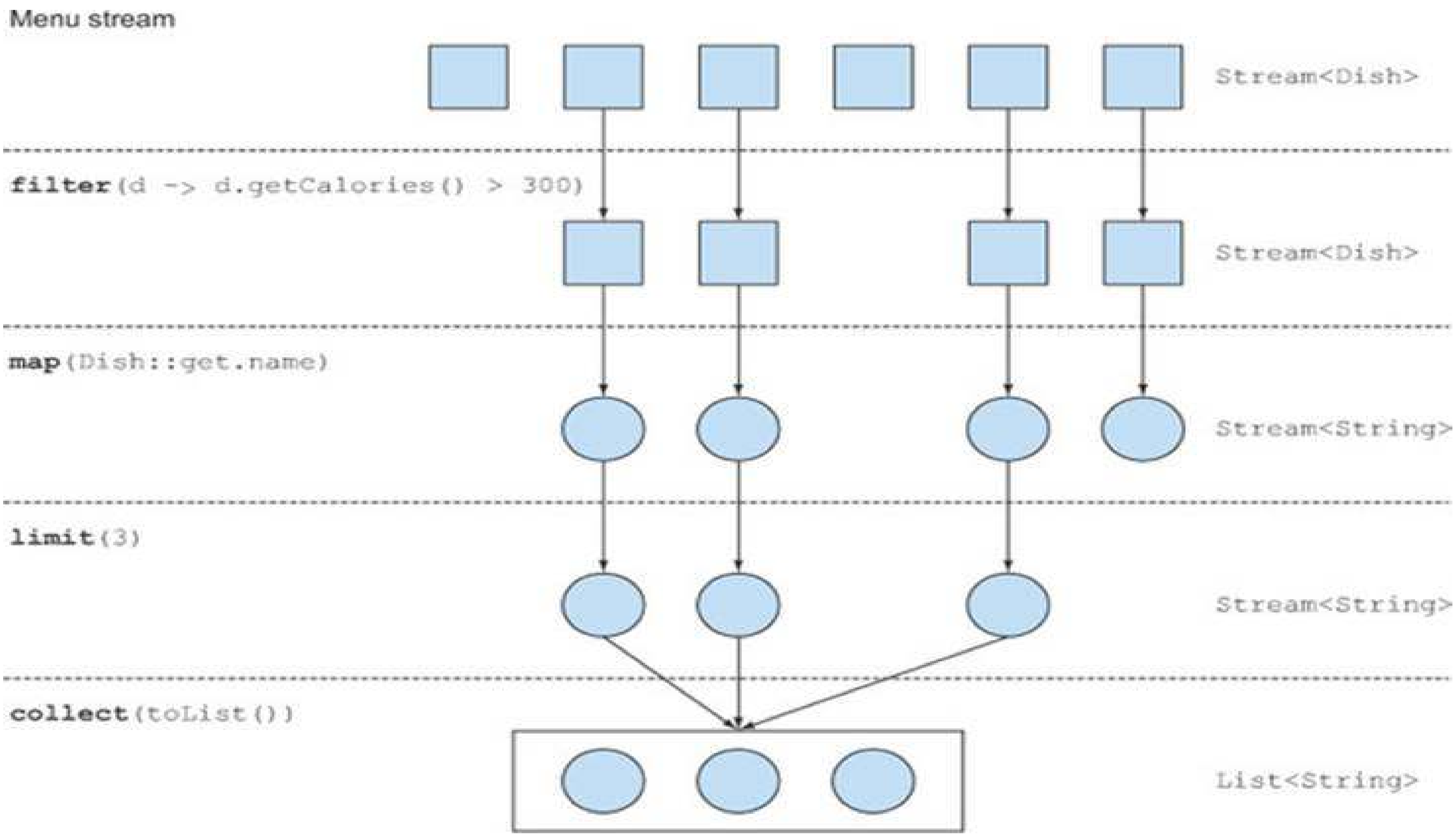


```java
public static void main(String[] args) {

    List<String> threeHighCalorieDishNames
            = menu.stream()
            .filter(d -> d.getCalories() > 300)
            .map(Dish::getName)
            .limit(3)
            .collect(toList());
    threeHighCalorieDishNames.forEach(System.out::println);
```

```
pork
beef
chicken
```

Stream API Example

- To make a list of the three High Calorie Dish Names with Stream API



Java SE 8 New Features

Parallel data processing and performance

Parallel data processing

- One of the most important benefit of Stream API is the possibility of executing a pipeline of operations on collections that automatically makes use of the multiple cores on your computer.
- Processing a collection of data in parallel was extremely cumbersome.
  - First, you needed to explicitly split the data structure containing your data into subparts.
  - Second, you needed to assign each of these subparts to a different thread.
  - Third, you needed to synchronize them opportunely to avoid unwanted race conditions, wait for the completion of all threads, and finally combine the partial results.
  - Java 7 introduced a framework called fork/join to perform these operations more consistently and in a less error-prone way.

Parallel data processing

- The Stream interface gives you the opportunity to execute operations in parallel on a collection of data without much effort.
- It lets you declaratively turn a sequential stream into a parallel one.
- Moreover, you'll see how Java can make this magic happen or, more practically, how parallel streams work under the hood by employing the fork/join framework introduced in Java 7.
- you can take control of this splitting process by implementing and using your own *Spliterator*.

Parallel Streams

- Let’s suppose you need to write a method accepting a number **n** as argument and returning the sum of all the numbers from 1 to the given argument.
- A straightforward approach is to generate an infinite stream of numbers, limiting it to the passed number, and then reduce the resulting stream with a BinaryOperator that just sums two numbers

```
public static long sequentialSum(long n) {
 return Stream.iterate(1L, i-> i+1)
             .limit(n)
             .reduce(0L, Long::sum);
}
```

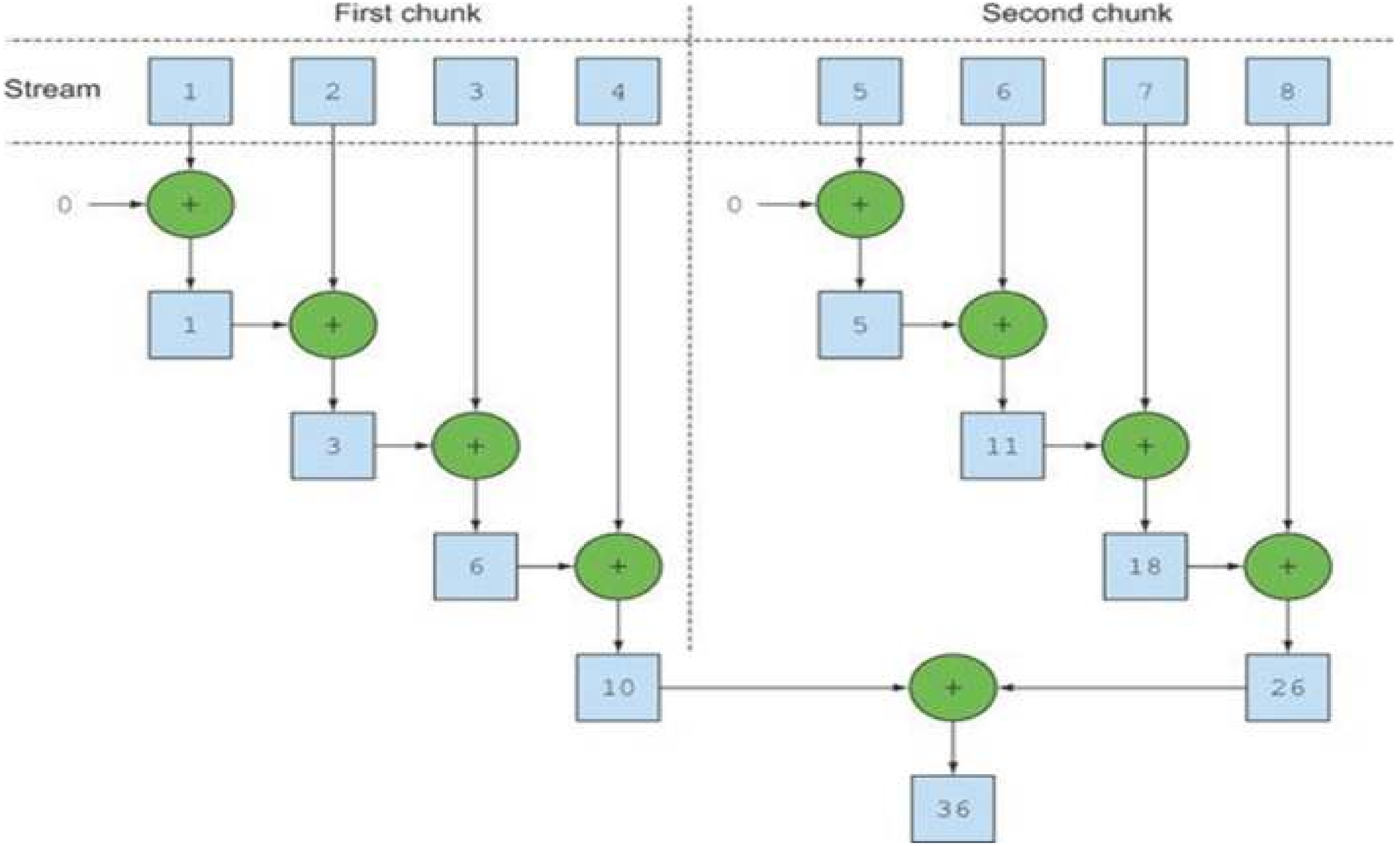
Parallel Streams

- This operation seems to be a good candidate to leverage parallelization, especially for large values of **n**.
- But where do you start?
- Do you synchronize on the result variable?
- How many threads do you use?
- Who does the generation of numbers?
- Who adds them up?
- Don’t worry about all of this. It’s a much simpler problem to solve if you adopt parallel streams!

Parallel Streams

```
public static long sequentialSum(long n) {
   return Stream.iterate(1L, i-> i+1)
                .limit(n)
                .parallel()
                .reduce(0L, Long::sum);
}
```

Parallel Streams



Java SE 8 New Features

Date and Time API

Date and Time API

- ❑ Almost all Java developers will agree that date and time support prior to Java 8 is far from ideal and most of the time we had to use third party libraries like *Joda-Time* in our applications.
- ❑ The new Date Time API is heavily influenced by *Joda-Time* API and if you have used it then you will feel home.
- ❑ Before we learn about new Date Time API let’s understand why existing Date API sucks. Look at the code shown below and try to answer what it will print.

Date and Time API

- Examples:

```
import java.util.Date;
class DateSucks
{
    public static void main(String args[]) {
      Date date = new Date(12, 12, 12);
      System.out.println(date);
    }
}
```

```
Sun Jan 12 00:00:00 GMT+02:00 1913
```

Date and Time API

- ❑ What each 12 means?
  - ❑ Is it month, year, date
  - ❑ or date, month, year or any other combination.
- ❑ Date API month index starts at 0. So, December is actually 11.
- ❑ Date API rolls over i.e. 12 will become January.
- ❑ Year starts with 1900. And because month also roll over so year becomes 1900 + 12 + 1 == 1913.
- ❑ Who asked for time? I just asked for date but program prints time as well.
- ❑ Why is there time zone? Who asked for it? The time zone is JVM's default time zone.

Date and Time API

❑ Java 8 Date Time API reside insides java.time package.

❑ The API applies domain-driven design principles with domain classes like LocalDate, LocalTime .

❑ This makes API intent clear and easy to understand.

❑ All the core classes in the java.time are immutable.

❑ The three classes that you will encounter most in the new API are :

  ❑ **LocalDate**: It represents a date with no time or timezone.

  ❑ **LocalTime**: It represents time with no date or timezone

  ❑ **LocalDateTime**: It is the combination of LocalDate and LocalTime i.e. date with time without time zone.

Date and Time API

- ❑ Examples: LocalDate has a static factory method of that takes year, month, and date and gives you a LocalDate.

```
 import java.time.LocalDate;
 import java.time.Month;
 class DateSucks
 {
     public static void main(String args[]) {
LocalDate ld = LocalDate.of(1931, Month.OCTOBER, 15);
       System.out.println(ld);
     }
  }
```

```
1931-10-15
```

Date and Time API

- ❑ Examples: LocalDate has a static factory method of that takes year, and date and gives you a LocalDate [**ofYearDay**].

```
import java.time.LocalDate;

class DateSucks

{
   public static void main(String args[]) {
 LocalDate ld1 = LocalDate.ofYearDay(2017, 21);
     System.out.println(ld1);
 LocalDate ld2 = LocalDate.ofYearDay(2017, 90);
     System.out.println(ld2);
   }
 }
```

```
2017-01-21
2017-03-31
```

Date and Time API

- ❑ Examples: The [**ofEpochDay]** creates LocalDate instance using the epoch day count. The starting value of is 1970-01-01.

```
import java.time.LocalDate;
class DateSucks
{
    public static void main(String args[]) {
      LocalDate ld3 = LocalDate.ofEpochDay(1);
       System.out.println(ld3);
     LocalDate ld4 = LocalDate.ofEpochDay(90);
       System.out.println(ld4);
     }
 }
```

```
1970-01-02
1970-04-01
```

Date and Time API

- Examples:
  - Date: Instant
  - Date: Duration [is the amount of time between instants]

```
import java.time.Instant;

import java.time.Duration;

class DateSucks

{
   public static void main(String args[]) {
      Instant start= Instant.now();
      Instant end= Instant.now();
      Duration d=Duration.between(start, end);
      System.out.println("Time between "+ d.toMillis());
    }
 }
```

Date and Time API

- Examples: A [**period**] is the amount of time between local dates

```java
import java.time.LocalDate;

import java.time.Period;

import  java.time.temporal.ChronoUnit;

class DateSucks

{
   public static void main(String args[]) {
    LocalDate now = LocalDate.now();
   LocalDate past = LocalDate.of(1564, Month.APRIL, 23);
     Period p = past.until(now);
     System.out.println("years =" + p.getYears());
     long days = past.until(now, ChronoUnit.DAYS);
     System.out.println("days =" + days);
   }
 }
```

```
years =452
days =165349
```

Date and Time API

- ❑ Examples:
  - ❑ LocalTime: is an everyday time: ex. 10:20

```
import java.time.LocalTime;

class DateSucks

{
    public static void main(String args[]) {
      LocalTime now=LocalTime.now();
      LocalTime time=LocalTime.of(10,20); //10:20
      LocalTime lunchTime= LocalTime.of(12,30);
    LocalTime coffeeTime =lunchTime.plusHours(2); //14:20
    }
 }
```

Date and Time API

- ❑ Examples: A [**period**] is the amount of time between local dates

```
import  java.time.ZonedDateTime;
import  java.time.ZoneId;
class DateSucks
{
  public static void main(String args[]) {
   System.out.println(ZonedDateTime.of(
    1564, Month.APRIL.getValue(), 23, //year/month/day
    10, 0, 0,0,                       //h/mn/s/nanos
    ZoneId.of("Europe/London")));
          }
 }
```

```
1564-04-23T10:00-00:01:15[Europe/London]
```

Date and Time API

- Examples: A [**period**] is the amount of time between local dates

```
import  java.time.ZonedDateTime;
import  java.time.ZoneId;
class DateSucks
{
  public static void main(String args[]) {
     Set<String> allZonesIds = ZoneId.getAvailableZoneIds();
     allZonesIds.forEach(System.out::println);
           }
 }
```

```
Asia/Aden
America/Cuiaba
Etc/GMT+9
:
:
```