Core Java 8 with JAXB and Development Tools

Java SE 8

IGATE

Speed.Agility.Imagination

Copyright © 2011 IGATE Corporation. All rights reserved. No part of this publication shall be reproduced in any way, including but not limited to photocopy, photographic, magnetic, or other record, without the prior written permission of IGATE Corporation.

IGATE Corporation considers information included in this document to be Confidential and Proprietary.

Document History

Date	Course Version No.	Software Version No.	Developer / SME	Change Record Remarks
12-Oct–2009	2.0	1.5	Anitha, Habib & Mahima	Revamped from J2SE 1.4 to J2SE 1.5
27-Oct-2009	3.0	1.5	CLS Team	Review
4 Jul 2011	4.0	1.5	Shrilata T	Changes in material made based on integration process
1 Mar 2015	5.0	1.8	Vinod Satpute	Changes made to include new features of Java version 6,7 and 8

Course Goals and Non Goals

- **Course Goals**
  - Implementing OOPs features in Java
  - Developing Java Desktop Applications
  - Use of Core JDK 1.8 API including JDBC 4.0
  - Testing using Junit 4
  - Logging Application using Log4J
  - Processing XML document using JAXB
- **Course Non Goals**
  - Developing GUI applications
  - Developing Multithreaded applications

IGATE
Speed.Agility.Imagination

Pre-requisites

- **Basic Programming Concepts**
- **OOPs**
- **DBMS/SQL**
- **XML**

IGATE
Speed.Agility.Imagination

Intended Audience

- **Developers new to Java technology**

IGATE
Speed.Agility.Imagination

Day Wise Schedule

- **Day 1**
  - Lesson 1:Introduction to Java
  - Lesson 2: Eclipse 4.4 (Luna) as an IDE
  - Lesson 3: Language Fundamentals
  - Lesson 4: Classes and Objects
- **Day 2**
  - Lesson 5: Exploring Basic Java Class Libraries
  - Lesson 6: Inheritance and Polymorphism
- **Day 3**
  - Lesson 7: Abstract Classes and Interfaces
  - Lesson 8: Regular Expressions
  - Lesson 9: Exception Handling
- **Day 4**
  - Lesson 10: Array
  - Lesson 11: Collection
  - Lesson 12: Generics

IGATE
Speed.Agility.Imagination

Day Wise Schedule

- **Day 5**
  - Lesson 13: File IO
  - Lesson 14: Introduction to Junit 4
- **Day 6**
  - Lesson 15: Lambda Expressions
  - Lesson 16: Stream API
- **Day 7**
  - Lesson 17: Property Files
  - Lesson 18: Java Database Connectivity (JDBC 4.0)
- **Day 8**
  - Lesson 19: Introduction to Layered Architecture
  - Lesson 20: Advanced Testing
- **Day 9**
  - Lesson 21: Logging with Log4J
  - Lesson 22: XML Processing using JAXB
  - Lesson 23: Security

Table of Contents – Day 1

- **Lesson 1: Introduction to Java**
  - 1.1: Introduction to Java
  - 1.2: Features of Java
  - 1.3: Evolution in Java
  - 1.4: Developing software in Java
- **Lesson 2: Eclipse 4.4 (Luna) as an IDE**
  - 2.1: Installation and Setting up Eclipse
  - 2.2: Introduction to Eclipse IDE
  - 2.3: Creating and Managing Java Projects
  - 2.4: Miscellaneous Options

IGATE
Speed.Agility.Imagination

Table of Contents – Day 1

- **Lesson 3: Language Fundamentals**
  - 3.1: Keywords
  - 3.2: Primitive Data Types
  - 3.3: Operators and Assignments
  - 3.4: Variables and Literals
  - 3.5: Flow Control: Java's Control Statements
  - 3.6: Best Practices
- **Lesson 4: Classes and Objects**
  - 4.1: Classes and Objects
  - 4.2: Packages
  - 4.3: Access Specifiers
  - 4.4: Constructors - Default and Parameterized
  - 4.5: this reference
  - 4.6: using static keyword
  - 4.7: Best Practices

IGATE
Speed.Agility.Imagination

Table of Contents – Day 2

- **Lesson 5: Exploring Basic Java Class Libraries**
  - 5.1: The Object Class
  - 5.2: Wrapper Classes
  - 5.3: Type casting
  - 5.4: Using Scanner Class
  - 5.5: String Handling
  - 5.6: Date and Time API
  - 5.7: Best Practices
- **Lesson 6: Inheritance and Polymorphism**
  - 6.1: Inheritance
  - 6.2: Using super keyword
  - 6.3: InstanceOf Operator
  - 6.4: Method & Constructor overloading
  - 6.5: Method overriding
  - 6.6: @override annotation
  - 6.6: Using final keyword
  - 6.7: Best Practices

IGATE
Speed.Agility.Imagination

Table of Contents – Day 3

- **Lesson 7: Abstract Classes and Interfaces**
  - 7.1: Abstract class
  - 7.2: Interfaces
  - 7.3: default methods
  - 7.4: static methods on Interface
  - 7.5: Runtime Polymorphism
  - 7.6: Best Practices
- **Lesson 8: Regular Expressions**
  - 8.1: Regular Expressions
  - 8.2: Validating data
  - 8.3: Best Practices

IGATE
Speed.Agility.Imagination

Table of Contents – Day 3

- **Lesson 9: Exception Handling**
  - 9.1: Introduction
  - 9.2: Exception Types
  - 9.3: Exception Hierarchy
  - 9.4: Try-catch-finally
  - 9.5: Try-with-resources
  - 9.6: Multi catch blocks
  - 9.7: Throwing exceptions using throw
  - 9.8: Declaring exceptions using throws
  - 9.9: User defined Exceptions
  - 9.10: Best Practices

IGATE
Speed.Agility.Imagination

Table of Contents – Day 4

- **Lesson 10: Array**
  - 10.1: One dimensional array
  - 10.2: Multidimensional array
  - 10.3: Using varargs
  - 10.4: Using Arrays class
  - 10.5: Best Practices
- **Lesson 11: Collection**
  - 11.1: Collections Framework
  - 11.2: Collection Interfaces
  - 11.3: Implementing Classes
  - 11.4: Iterating Collections
  - 11.5: Comparable and Comparator
  - 11.6: Best Practices
- **Lesson 12: Generics**
  - 12.1: Generics
  - 12.2: Writing Generic Classes
  - 12.3: Using Generics with Collections
  - 12.4: Best Practices

IGATE
Speed.Agility.Imagination

Table of Contents – Day 5

- **Lesson 13: File IO**
  - 13.1: Overview of I/O Streams
  - 13.2: Types of Streams
  - 13.3: The Byte-stream I/O hierarchy
  - 13.4: Character Stream Hierarchy
  - 13.5: Buffered Stream
  - 13.6: The File class
  - 13.7: The Path class
  - 13.8: Object Stream
  - 13.9: Best Practices
- **Lesson 14 : Introduction to Junit 4**
  - 14.1: Introduction
  - 14.2: JUnit
  - 14.3: Installing and Running JUnit
  - 14.4: Testing with JUnit
  - 14.5: Testing Exceptions
  - 14.6: Test Fixtures
  - 14.7: Best Practices

IGATE
Speed.Agility.Imagination

Table of Contents – Day 6

- **Lesson 15: Lambda Expressions**
  - 15.1: Introduction
  - 15.2: Writing Lambda Expressions
  - 15.3: Functional Interfaces
  - 15.4: Types of Functional Interfaces
  - 15.5: Method reference
- **Lesson 16: Stream API**
  - 16.1: Introduction
  - 16.2: Stream API with Collections
  - 16.3: Stream Operations

IGATE
Speed.Agility.Imagination

Table of Contents – Day 7

- **Lesson 17: Property Files**
  - 17.1: What are Property Files?
  - 17.2: Types of Property files
  - 17.3: User defined Properties
- **Lesson 18: Java Database Connectivity (JDBC 4.0)**
  - 18.1: Java Database Connectivity - Introduction
  - 18.2: Database Connectivity Architecture
  - 18.3: JDBC APIs
  - 18.4: Database Access Steps
  - 18.5: Calling database procedures
  - 18.6: Using Transaction
  - 18.7: Connection Pooling
  - 18.8: DAO Design Pattern
  - 18.9: Best Practices

IGATE
Speed.Agility.Imagination

Table of Contents – Day 8

- **Lesson 19: Introduction to Layered Architecture**
  - 19.1: Introduction
  - 19.2: Testing DAO Classes
  - 19.3: Testing Exceptions
- **Lesson 20: Advanced Testing Concepts**
  - 20.1: Advanced Testing concepts
  - 20.2: Test Suites
  - 20.3: Parameterized Tests
  - 20.4: Mocking Concepts

IGATE
Speed.Agility.Imagination

Table of Contents – Day 9

- **Lesson 21: Logging with Log4J**
  - 21.1 Log4J Concepts
  - 21.2 Installation of Log4J
  - 21.3 Configuring Log4J
  - 21.4: Best Practices
- **Lesson 22: XML Processing using JAXB**
  - 22.1: Introduction
  - 22.2: JAXB Architecture
  - 22.2: JAXB Bindings
  - 22.3: Java to Schema binding
- **Lesson 23: JVM Overview**
  - 23.1: Java Security Architecture

IGATE
Speed.Agility.Imagination

References

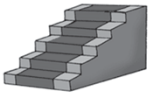
- **Books:**
  Java, The Complete Reference; by Herbert Schildt
  Thinking in Java; by Bruce Eckel
  Beginning Java 8 Fundamentals by Kishori Sharan

- **Websites:**
  Java home page: http://java.sun.com/
  JDK 1.8 documentation: http://docs.oracle.com/javase/8/docs/

IGATE
Speed.Agility.Imagination

Next Step Courses

- **Servlets**
- **JSP**



IGATE
Speed.Agility.Imagination

Other Parallel Technology Areas

- **C ++**
- **C#.Net**
- **Visual Basic.Net**

IGATE
Speed.Agility.Imagination

Core Java 8 with JAXB and Development Tools

Introduction to Java

IGATE
Speed.Agility.Imagination

Lesson Objectives

- **After completing this lesson, participants will be able to -**
  - Explain the main features of Java
  - Differentiate between JRE, JVM and JDK
  - Write, compile, and execute a simple Java program

Java's Lineage

- C language was result of the need for structured, efficient, high-level language replacing assembly language.
- C++, which followed C, became the common (but not the first) language to offer OOP features, winning over procedural languages such as C.
- Java, another object oriented language offering OOP features, followed the syntax of C++ at most places. However, it offered many more features.

June 11, 2015 | Proprietary and Confidential | - 3 -

IGATE Speed.Agility.Imagination

Introduction to Java:

- To understand Java, a new age Internet programming language, first it is essential to know the forces that drove to the invention of this kind of language. The history starts from a language called as **B,** which led to a famous one **C** and then to **C++**. However, the jump from C to C++ was a major development, as the whole approach of looking at an application changed from **procedural way** to **object oriented way**. The languages like, Smalltalk, were already using the Object Oriented features.
- By the time C++ got the real acknowledgement from the industry, there was a strong need for a language which creates architecture neutral, platform independent, and portable software. The reason behind this particular need was the Embedded software market, which runs on variety of consumer electronic devices. Hence a project called as "OAK" was started at Sun Microsystems.
- The second force behind this is WWW (World Wide Web). Now on WWW, most of computers over the Internet are divided into three major architectures namely Intel, Macintosh, and Unix. Thus, a program written for the Internet has to be a portable program, so that it runs on all the available platforms.
- All this led to the development of a new language in 1995, when they renamed the "OAK" language to "JAVA".

1.1: Introduction to Java

What is Java?

- **Java is an Object-Oriented programming language – most of it is free and open source!**
  - It is developed in the early 1990s, by James Gosling of Sun Microsystems
  - It allows development of software applications.
  - It is amongst the preferred choice for developing internet-based applications

June 11, 2015 | Proprietary and Confidential | - 4 -

IGATE Speed.Agility.Imagination

Introduction to Java Features – What is Java?

- Java programming language is a high level programming language offering Object-Oriented features. James Gosling developed it at Sun Microsystems in the early 1990s. It is on the lines of C/C++ syntax, however, it is based on an object model. Sun offers much of Java as free and open source. Today we have the Java version 6 in the market.
- The Java programming language forms a core component of Sun's Java Platform. By platform, we mean the mix of hardware and software environment in which a program executes – such as MS Windows and Linux. Sun's Java is a "software-only" platform which runs on top of other hardware platforms. We will learn more about this on subsequent slides.
- Today, Java has become an extremely popular language. The platform is amongst the preferred choices for developing internet based applications. Why is it so? Let us explore!

1.2 : Features of Java

Java Language Features

- **Java has advantages due to the following features:**
  - Completely Object-Oriented
  - Simple
  - Robust: Strongly typed language
  - Security
    - Byte code Verifier
    - Class Loader
    - Security Manager
  - Architecture Neutral: Platform independent
  - Interpreted and Compiled
  - Multithreaded: Concurrent running tasks
  - Dynamic
  - Memory Management and Garbage Collection

June 11, 2015 | Proprietary and Confidential | - 5 -

IGATE Speed.Agility.Imagination

Features of Java:

Java is completely object oriented. C++, being more compatible to C, allows code to exist outside classes too. However in Java, every line of code has to belong to some or other class. Thus it is closer to true object oriented language.
Java is simpler than C++, since concepts of pointers or multiple inheritance do not exist.
Let us discuss these features in detail.

Object-Oriented:
To stay abreast of modern software development practices, Java is Object-Oriented from the ground up. Many of Java's Object-Oriented concepts are inherited from C++, the language on which it is based, plus concepts from other Object-Oriented languages as well.

Simple:
Java omits many confusing, rarely used features of C++. There is no pointer level programming or pointer arithmetic. Memory management is automatic. There are no header files, structures, unions, operator overloading, virtual base classes, and multiple inheritance.

Robust:
Java programs are reliable. Java puts a lot of emphasis on early checking for potential problems, dynamic checking and eliminating situations that are error-prone. Java has a pointer model that eliminates possibility of overwriting memory and corrupting data.

Features of Java (contd.):

Security:
Java is intended to be used in networked / distributed environments. Thus a lot of emphasis is placed on security. Normally two things affect security:

- confidential information may be compromised, and
- computer systems are vulnerable to corruption or destruction by hackers

Java's security model has three primary components:

- **Byte code Verifier**: The byte code verifier ensures the following:
  - the Java programs have been compiled correctly,
  - they will obey the virtual machine's access restrictions, and
  - the byte codes will not access private data when they should not
- **Class Loader:** When the loader retrieves classes from the network, it keeps classes from different servers separate from each other and from local classes. Through this separation, the class loader prevents a class that is loaded off the network from pretending to be one of the standard built-in classes, or from interfering with the operation of classes loaded from other servers.
- **Security Manager:** It implements a security policy for the VM. The security policy determines which activities of the VM is allowed to perform and under what circumstances, operation should pass.

Architecture-Neutral:
A central issue for the Java designers was of code longevity and portability. One of the main problems facing programmers is that there is no guarantee that when you write a program today, it will run tomorrow — even on the same machine. Operating system upgrades, processor upgrades, and changes in core system resources can all combine to make a program malfunction. The Java designers made several hard decisions in the Java language and the Java Virtual Machine (JVM) in an attempt to alter this situation. Their goal was "write once; run anywhere, any time, forever".
To a great extent, this goal was accomplished.

Platform-Independent:
This refers to a program's capability of moving easily from one computer system to another. Java is platform-independent at both the source and the binary level.

- At the source level, Java's primitive data types have consistent sizes across all development platforms.
- Java binary files are also platform-independent and can run on multiple platforms without the need to recompile the source because they are in the form of byte codes.

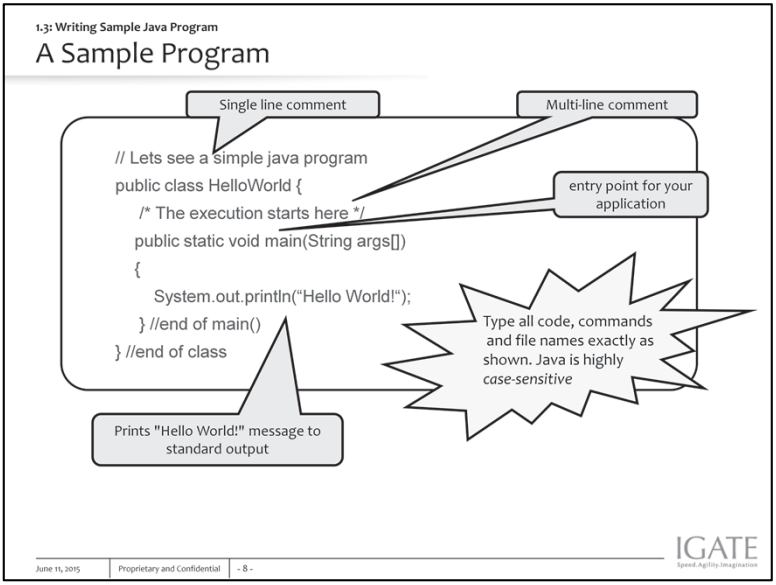
Features of Java (contd.):

Interpreted and Compiled:
Java programs are compiled into an intermediate byte code format, which in turn will be interpreted by the VM at run time. Hence, any Java program is checked twice before it actually runs.

Multithreaded:
A multi-threaded application can have several threads of execution running independently and simultaneously. These threads may communicate and co-operate. To the user it will appear to be a single program. Java implements multithreading through a part of its class library. However, Java also has language constructs to make programs thread-safe.

Dynamic:
Java programs carry with them substantial amounts of run-time type information that is used to verify and resolve accesses to objects at run time. This makes it possible to dynamically link code in a safe and expedient manner. This is crucial to the robustness of the applet environment, in which small fragments of bytecode may be dynamically updated on a running system.

Memory Management and Garbage Collection:
Java manages memory de-allocation by using garbage collection. Temporary memory is automatically reclaimed after it is no longer referenced by any active part of the program. To improve performance, Java's garbage collector runs in its own low-priority thread, providing a good balance of efficiency and real-time responsiveness.

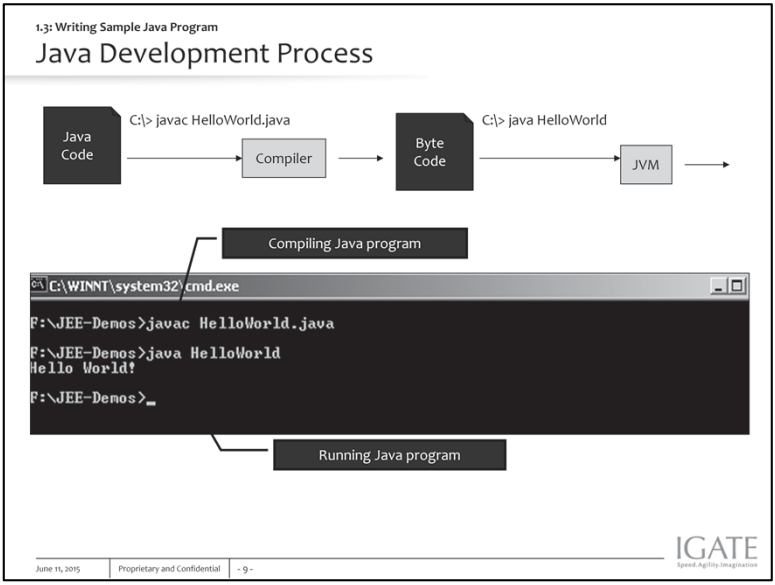


Introduction to Java Features – A Sample Program:

- Here is our first Java program..

- Let us have a closer Look at the "Hello World!" program:
  - **class HelloWorld** begins the class definition block for the "Hello World!" program. Every Java program must be contained within a "class" block.
  - **public static void main(String[] args) { …}** is the entry point for your application. Subsequently, it invokes all the other methods required by the program.

- This program when executed will display "Hello World" on the screen. We shall see this in the next slide.



Introduction to Java Features – Java Development Process:

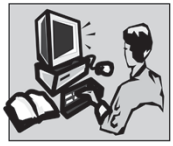
The Java Development Process involves the following steps:

1. Write the Java code in a text file with a .java extension, namely "HelloWorld.java". By convention, source file names are named by the public class name they contain.
2. At the command line, compile the code using the Java compiler (javac). The javac compiler converts the source into Byte Codes and stores it in a file having .class extension. What is special about byte codes? Unlike traditional compilers, javac does not produce processor specific native code. Instead it generates code which is in the language of JVM (Java Virtual Machine).
   Note: To have access to the **javac** and the **java** commands, you must set your path first. To do so, you may type the following at the command prompt:
   Set path=<your java-home directory>\bin
   For example: *Set path= C:\Program Files\Java\jdk1.8.0_25 \bin*
   We can also set the environment variables (path and classpath) through the Control Panel.
3. To run this program, use the Java command with class file name as the command parameter as shown on the above slide. The output of the program would, of course, be "Hello World!"

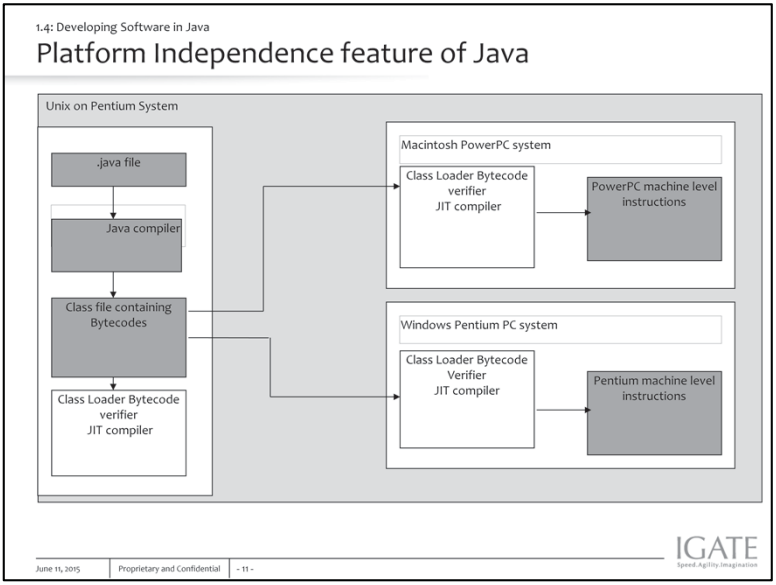
1.3: Writing Sample Java Program

Demo

- **Creating and executing the First Java application**



IGATE
Speed.Agility.Imagination



Platform Independence:

- The figure illustrates how platform independence is achieved using Java. Once you write Java code on a platform and run it through Java Compiler, the class file containing byte codes is obtained.
- Different JVMs are available for different platforms. So the JVM for Unix on Pentium will be different from the JVM for Mac or for Windows. Each of these JVMs take the same input, namely the Class File, and produce the machine level instructions for the respective platforms.
- One common grouse among developers is that Java programs take longer to execute because the compiled bytecodes are *interpreted* by the JVM. The Java just-in-time (JIT) compiler, compiles the bytecode into platform-specific executable code (**native code**) that is immediately executed, thus speeding up execution! Traditional native code compilers run on the developer's machine and are used by programmers, and produce non-portable executables. JIT compilers run on the user's machine and are transparent to the user. The resulting native code instructions do not need to be ported because they are already at their destination.

Platform Independence (contd.):

JVM:

When you compile a Java program (which usually is a simple text file with .java extension), it is compiled to be executed under VM. This is in contrast to C/C++ programs, which are compiled to be run on a real hardware platform, such as a Pentium processor running on, say Win 95. The VM itself has characteristics very much like a physical microprocessor. However, it is entirely a software construct. You can think of the VM as an intermediary between **Java programs** and the underlying **hardware platform** on which all programs must eventually execute.

- Even with the VM, at some point, all Java programs must be resolved to a particular underlying hardware platform. In Java, this resolution occurs within each particular VM implementation. The way this works is that Java programs make calls to the VM, which in turn routes them to appropriate native calls on the underlying platform. It is obvious that the **VM itself** is very much **platform dependent**.

How does the JIT compiler work?

- The VM instead of calling the underlying native operating system, it calls the JIT compiler. The JIT compiler in turn generates native code that can be passed on to the native operating system for execution. The primary benefit of this arrangement is that the JIT compiler is completely transparent to everything except VM. The neat thing is that a JIT compiler can be integrated into a system without any other part of the Java runtime system being affected.
- The integration of JIT compilers at the VM level makes JIT compilers a legitimate example of component software. You can simply plug in a JIT compiler and reap the benefits with no other work or side effects.
- A Java enabled browser contains its own VM. Web documents that have embedded Java applets must specify the location of the main applet class file. The Web browser then starts up the VM and passes the location of the applet class file to the class loader. Each class file knows the names of any additional class files that it requires. These additional class files may come from the network or from client machine. Supplement classes are fetched only if they are actually going to be used or if they are necessary for the verification process of the applets.

1.4: Developing Software in Java

JRE versus JDK

- **JRE is the "Java Runtime Environment". It is responsible for creating a Java Virtual Machine to execute Java class files (that is, run Java programs).**
- **JDK is the "Java Development Kit". It contains tools for Development of Java code (for example: Java Compiler) and execution of Java code (for example: JRE)**
- **JDK is a superset of JRE. It allows you to do both – write and run programs.**

June 11, 2015 | Proprietary and Confidential | - 13 -

IGATE Speed.Agility.Imagination

Difference between JRE and JDK:

- The **Java Development Kit (JDK)** is a superset which includes Java Compilers, Java Runtime Environments (JRE), Development Libraries, Debuggers, Deployment tools, and so on. One needs JDK to develop Java applications. We have different versions that include JDK 1.2, JDK 1.4, and so on.
- The **Java Runtime Environment (JRE)** is an implementation of JVM that actually executes the Java program. It is a subset of JDK. One needs JRE to execute Java applications.

Summary

- **In this lesson, you have learnt:**
  - Features of Java and its different versions
  - How Java is platform Independent
  - Difference between JRE and JDK
  - Writing, Compiling, and Executing a simple program

Review Question

- **Question 1: A program written in the Java programming language can run on any platform because...**
  - **Option 1:** The JIT Compiler converts the Java program into machine equivalent
  - **Option 2:** The Java Virtual Machine1(JVM) interprets the program for the native operating system
  - **Option 3:** The compiler is identical to a C++ compiler
  - **Option 4:** The APIs do all the work
- **Question 2: Java Compiler compiles the source code into ___ code, which is interpreted by ___ to produce Native Executable code.**

Review Question

- **Question 3: Which of the following are true about JVM?**
  - **Option 1:** JVM is an interpreter for byte code
  - **Option 2:** JVM is platform dependent
  - **Option 3:** Java programs are executed by the JVM
  - **Option 4:** All the above is true
- **Question 4 : ____ allows a Java program to perform multiple activities in parallel.**
  - **Option 1:** Java Beans
  - **Option 2:** Swing
  - **Option 3:** Multithreading
  - **Option 4:** None of the above

Core Java 8 with JAXB and Development Tools

Eclipse 4.4 as an IDE

IGATE
Speed.Agility.Imagination

Lesson Objectives

- **After completing this lesson, participants will be able to:**
  - Understand fundamentals of working with Eclipse
  - Creating and Managing Java Projects through Eclipse IDE
  - Use different features of Eclipse to develop rapid applications

June 11, 2015 | Proprietary and Confidential | - 2 -

IGATE Speed.Agility.Imagination

This lesson demonstrate the use of IDE to create Java applications with ease.

Lesson Outline:

Lesson 2: Eclipse4.4 as an IDE

- 2.1: Installation and setting up Eclipse
- 2.2: Introduction to Eclipse IDE
- 2.3: To create and manage Java projects
  - 2.3.1: Debugging your Java Program
- 2.4: Miscellaneous options
  - 2.4.1: Creating Jar files
  - 2.4.2: Verifying JRE installation
  - 2.4.3: Creating a Jar file
  - 2.4.4: Setting Classpath
  - 2.4.5: Passing Command line arguments
  - 2.4.6: Import and Export Options
  - 2.4.7: Automatic Build/Manual Build options
  - 2.4.8: Using Javadocs
  - 2.4.9: Tips and Tricks

2.1: Installation and Setting up Eclipse

Installing Eclipse 4.4 (Luna)

- **You need to follow the given steps to install Eclipse 4.4:**
  - Download Eclipse-SDK zip file from https://eclipse.org/downloads/
  - Unpack the Eclipse SDK into the target directory
    - For example: c:\eclipse4.4
  - To start Eclipse, go to the eclipse subdirectory of the folder in which you extracted the zip file
    (for example: c:\eclipse4.4\eclipse) and run eclipse.exe

2.2 : Introduction to Eclipse IDE

Integrated Development Environment

- **IDE is an application or set of tools that allows a programmer to write, compile, edit, and in some cases test and debug within an integrated, interactive environment**
- **IDE combines:**
  - Editor
  - Compiler
  - Runtime environment
  - debugger

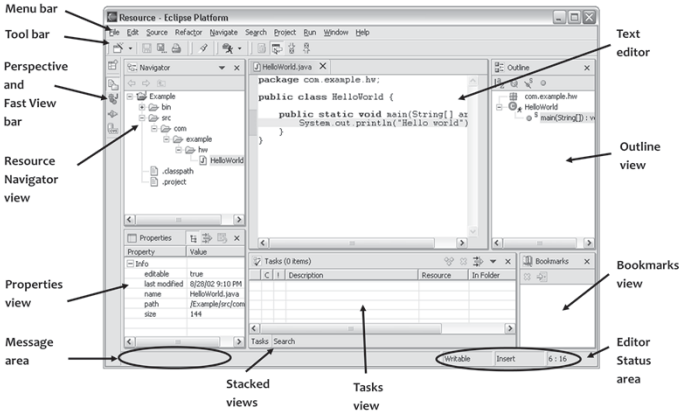
June 11, 2015 | Proprietary and Confidential | - 4 -

IGATE Speed.Agility.Imagination

What is an IDE?

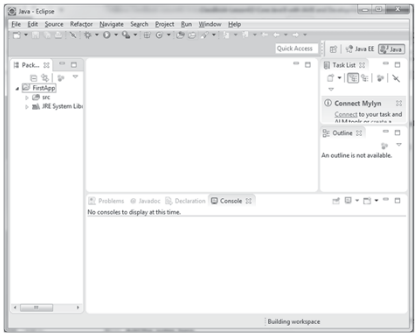
- The **Eclipse Project** is an open source software development project dedicated to providing a robust, full-featured, commercial-quality, industry platform for the development of highly integrated tools and rich client applications. Eclipse runs on Windows, Linux, Mac OSX, Solaris, AIX and HP-UX. Eclipse is actually a generic application platform with a sophisticated plug in architecture - the Java IDE is just one set of plugins. There is an active community of third party Eclipse plugin developers, both open source and commercial. Our objective is to code Java programs faster with Eclipse 4.4 as an IDE.
- Eclipse4.4 features include the following:
  - Creation and maintenance of the Java project
  - Developing Packages
  - Debugging a java program with variety of tools available
  - Running a Java program
- Developing the Java program will be easier as Eclipse editor provides the following:
  - Syntax highlighting
  - Content/code assist
  - Code formatting
  - Import assistance
  - Quick fix

Workbench Terminology



The Workbench

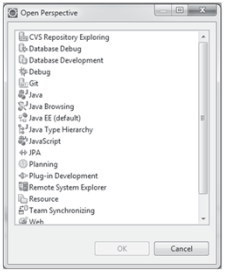
- **The term "**Workbench**" refers to the desktop development environment**
- **It allows you to select the Workspace**
- **A Workbench consists of the following:**
  - perspectives
  - views
  - editors



IGATE
Speed.Agility.Imagination

The Workbench

- **Perspective:**
  - A perspective defines the initial set and layout of views in the Workbench window
    - Workbench offers one or more Perspectives
    - A perspective contains editors and views, such as the Navigator
    - By default the **Java perspective** is selected
    - The title bar indicates which perspective is open

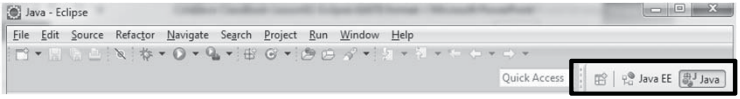


June 11, 2015 | Proprietary and Confidential | - 7 -

IGATE Speed.Agility.Imagination

<u>The Workbench</u>:
The term **Workbench** refers to the desktop development environment.

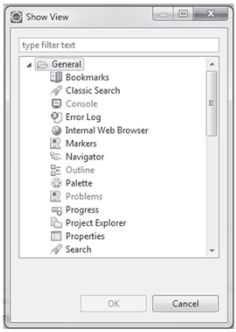
Perspectives:
Each Workbench window contains one or more perspectives. Perspectives contain **views** and **editors** and control what appears in certain **menus** and **tool bars**. They define visible **action sets**, which you can change to customize a perspective. You can save a perspective that you build in this manner, making your own custom perspective that you can open again later. By default the Java perspective is selected.



Each perspective provides a set of functionality aimed at accomplishing a specific type of task or works with specific types of resources.
For example: The **Java perspective** combines views that you would commonly use while editing Java source files, while the **Debug perspective** contains the views that you would use while debugging Java programs. As you work in the Workbench, you will probably switch perspectives frequently.

The Workbench

- **View:**
  - It is the visual component within the Workbench
  - It is used to navigate a hierarchy of information or display properties for the active editor



IGATE Speed.Agility.Imagination

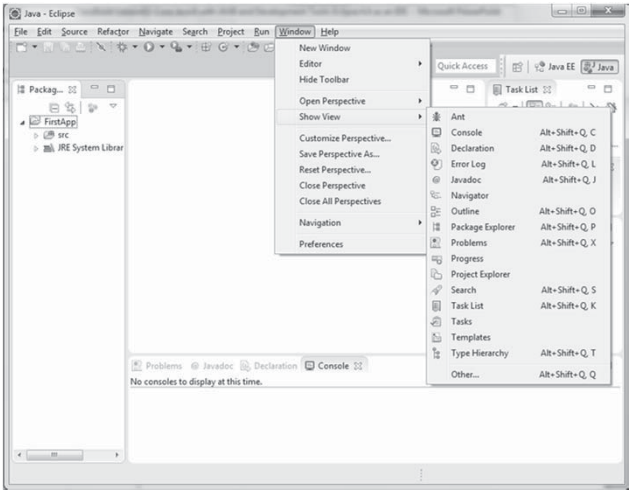
<u>The Workbench</u>:

View:

Views support editors and provide alternative presentations as well as ways to navigate the information in your Workbench.

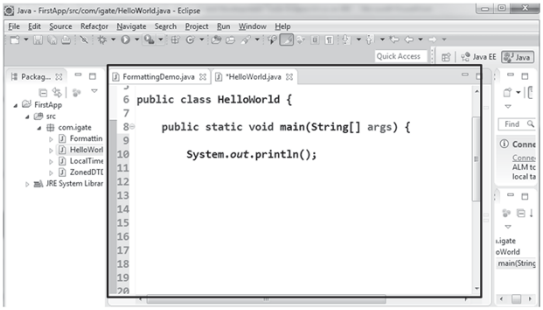
For example: The Project Explorer and other navigation views display projects and other resources that you are working with.

Perspectives offer pre-defined combinations of views and editors. To open a view that is not included in the current perspective, select **Window → Show View** from the main menu bar.



The Workbench

- **Editor:**
  - It is the visual component within the Workbench
  - It is used to edit or browse a resource



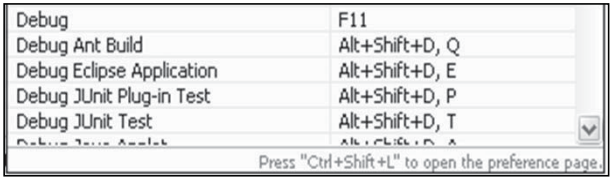
IGATE
Speed.Agility.Imagination

<u>The Workbench</u>:
Editor: Most perspectives in the Workbench comprise an **editor area** and one or more **views**. You can associate different editors with different types of files.
For example: When you open a file for editing by double-clicking it in one of the navigation views, the associated editor opens in the Workbench.

If there is no associated editor for a resource, then the Workbench attempts to launch an external editor outside the Workbench.
For example: Suppose you have a .doc file in the Workbench, and Microsoft Word is registered as the editor for .doc files in your operating system. Then opening the file will launch Word as an OLE document within the Workbench editor area.

The Workbench:

- The Workbench menu bar and toolbar will be updated with options for Microsoft Word).
- Any number of editors can be open at once. However, only one can be active at a time. The main menu bar and toolbar for the Workbench window contain operations that are applicable to the active editor.
- Useful Tips and Tricks related with Work Bench:
  - **View all Keyboard shortcuts:** While working with your favorite editors and views in Eclipse, just press **Ctrl+Shift+L** to see a full list of the currently available key bindings.



Debug	F11
Debug Ant Build	Alt+Shift+D, Q
Debug Eclipse Application	Alt+Shift+D, E
Debug JUnit Plug-in Test	Alt+Shift+D, P
Debug JUnit Test	Alt+Shift+D, T

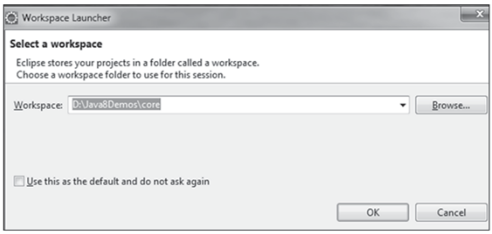
Press "Ctrl+Shift+L" to open the preference page.

  - **Importing Files:** You can quickly import files and folders into your workspace by dragging them from the file system (for example: from a Windows Explorer window) and dropping them into the Project Explorer view. The files and folder are always copied into the project. The originals are not affected. Copy and paste also work.
  - **Exporting Files:** You can drag files and folder from the Project Explorer view to the file system (e.g., to a Windows Explorer window) to export the files and folders. The files and folder are always copied. However, the workspace resources are not affected. Copy and paste also work.
  - **Switch Workspace:** Instead of shutting down eclipse and restarting with a different workspace, you can instead use **File → Switch Workspace**. From here you can either open previous workspaces directly from the menu or you can open the **workspace chooser** dialog to choose a new one.

2.3: Creating and Managing Java Projects

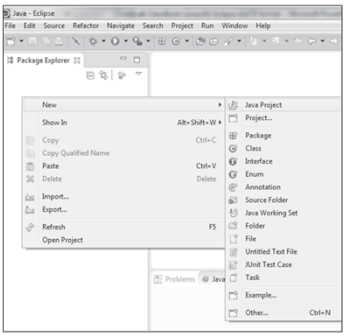
Create Workspace

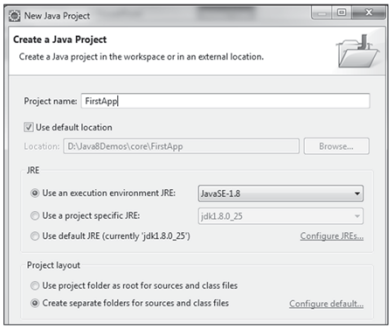
- **You need to follow the given steps to create a workspace:**
  - Start up Eclipse
  - Supply a path to a new folder which will serve as your workspace
  - The workspace is a folder which Eclipse uses to store your source code



Create a Java Project

- **Right-click the Package Explorer panel, and select New-JavaProject.**
- **Select Java project and provide a Project Name.**





IGATE Speed.Agility.Imagination

<u>Creating and Managing Java Projects</u>:

Create a Java Project:

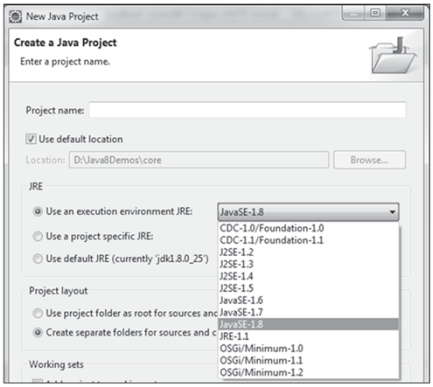
- When Eclipse starts, you will see the Welcome page. Close the Welcome page.
- Right-click in the Package Explorer panel, and select **New → JavaProject** and provide a Project name.

Note:

- The name of the project is FirstApp. It is created in the workspace which you have selected in the beginning.
- The Project uses jre1.8.0_25. The same folder will be used for storing both the source files and the class files.
- The Java project can now include the following:
    - Class
    - Package
    - Interface
    - Source folder
    - Folder
    - File
    - Junit Test Case
    - Other - which may include other resources as text files

Select the JRE

- **In order to develop code compliant with Java SE 8, you will need a JavaSE-1.8 Java Runtime Environment (JRE)**

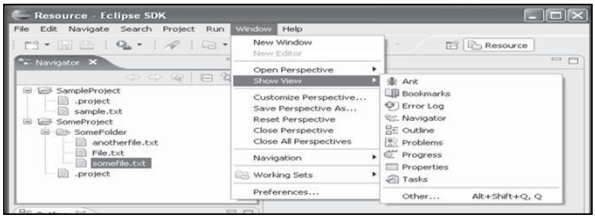


IGATE Speed.Agility.Imagination

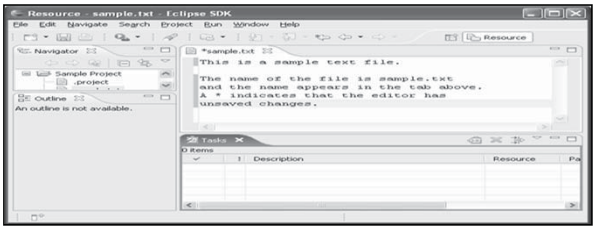
<u>Creating and Managing Java Projects</u>:

Select the JRE:

- To use the new **Java SE 1.8** features, you must be working on a project that has a **1.8 compliance level** enabled and has a **1.8 JRE**. New projects will automatically get 1.8-compliance while choosing a 1.8 JRE on the first page of the New Java Project wizard.

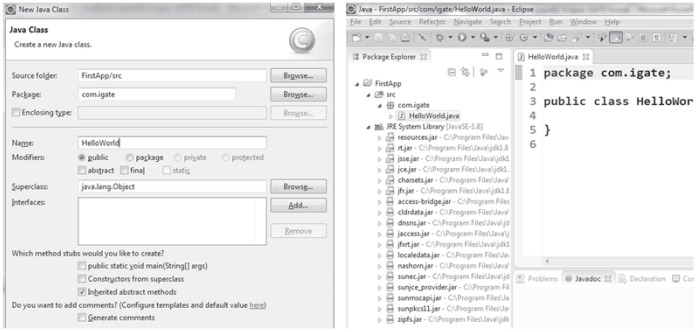


- Depending on the type of file that is being edited, the appropriate editor is displayed in the editor area.
  For example: If a .TXT file is being edited, a text editor is displayed in the editor area.



My first Java Program – Hello World

- **Right-click on the project and select "New->Class" Type in your Program code**



IGATE Speed.Agility.Imagination

<u>Creating and Managing Java Projects</u>:

My first Java Program – Hello World:

- If you want to create some new Java code, right-click the project and select "**New → Class**".
- In the dialog box created, give a name for the Java program in the **Name** textbox.
- Also notice that a package name is also given, as the usage of default package is discouraged. A package in Java is a group of classes which are often closely or logically related in some way. (The **Package chapter** is discussed later in the course).
- Eclipse will generate skeleton of the class including **default** constructor.

Note:

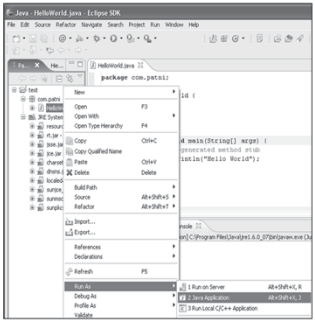
- The package name is com.igate.
- The Java program's name is HelloWorld.
- The HelloWorld class will have the public access modifier.
- The HelloWorld class inherits from java.lang.Object.

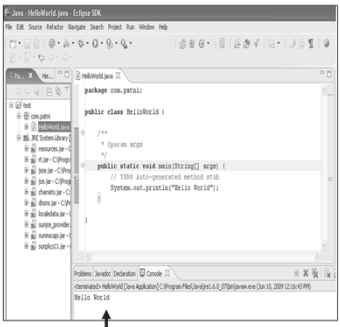
Developing the Java program will be easier as Eclipse editor will provide:

- Syntax highlighting
- Content/code assist
- Code formatting
- Import assistance
- Quick fix

Executing Hello World Program

- **Right-click the program and select Run As-Java Application.**





Output

June 11, 2015 | Proprietary and Confidential | - 15 -

IGATE Speed.Agility.Imagination

<u>Creating and Managing Java Projects</u>:

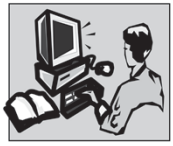
Executing Hello World Program:

- Right-click the **HelloWorld.java** in the Package Explorer, and select **Run As → Java Application**.
- The program after execution produces the output in the **Console view**.
- Running class from the Package Explorer as a Java Application uses the default settings for launching the selected class, and does not allow you to specify any arguments.

2.3: Creating and Managing Java Projects

Demo

- **HelloWorld Program using Eclipse IDE**



IGATE
Speed.Agility.Imagination

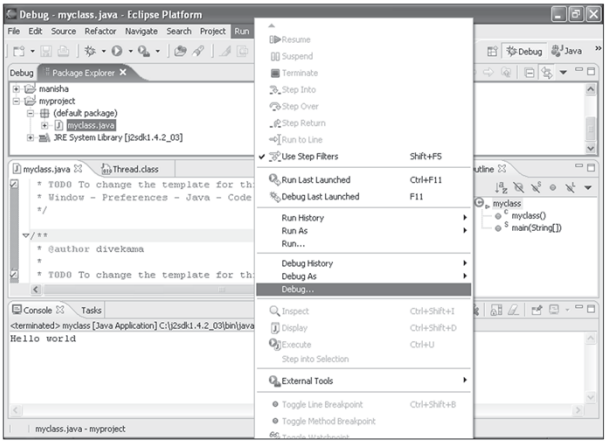
2.3: Creating and Managing Java Projects

Debugging your Java Program using Eclipse

- **The Java Development Toolkit (JDT) includes a debugger that enables you to detect and diagnose errors in your programs running either locally or remotely**
- **The debugger allows you to control the execution of your program by employing the following:**
  - setting breakpoints, suspending launched programs, stepping through your code, and examining the contents of variables

IGATE
Speed.Agility.Imagination

Debugging your Java Program using Eclipse



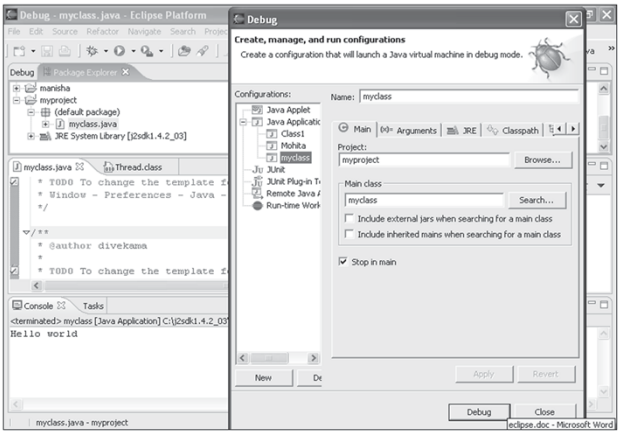
IGATE
Speed.Agility.Imagination

<u>Creating and Managing Java Projects</u>:

Debugging a Java Program:

- Eclipse gives you **auto-build** facility, where recompilation of the necessary Java classes is done automatically.
- To debug your Java program, select the Java source file which needs to be debugged, select **Run → Debug**. This will ask you to select an option to halt in public static void main() method, from where you may select to step into each and every function you come across or step over every function and only capture output of each function.

Specifying Debugging options

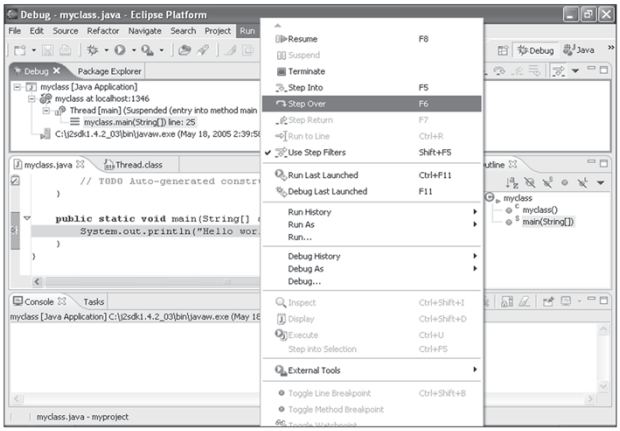


June 11, 2015 | Proprietary and Confidential | - 19 -

IGATE Speed.Agility.Imagination

Note: Click **Debug** to start debugging process.

Debugging a Java Program



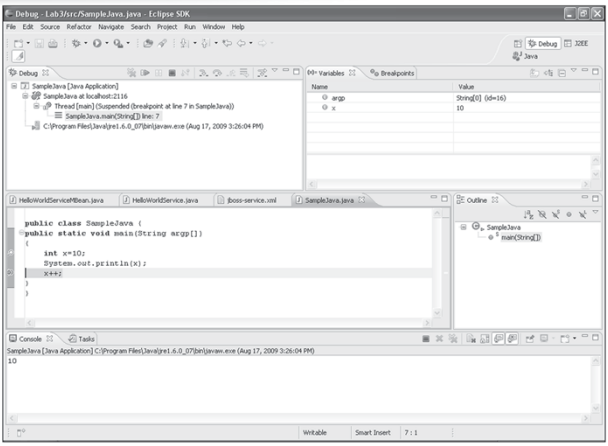
IGATE
Speed.Agility.Imagination

<u>Creating and Managing Java Projects</u>:

Debugging a Java Program:

- Debugging can be attained by stepping-into or stepping-over the statements.
  - **Step-into** will traverse through each and every statement in a function.
  - **Step-over** will generate output after the function call is over.
- Tracing and watching the variable values is available as different debug views.

Debugging a Java Program



IGATE
Speed.Agility.Imagination

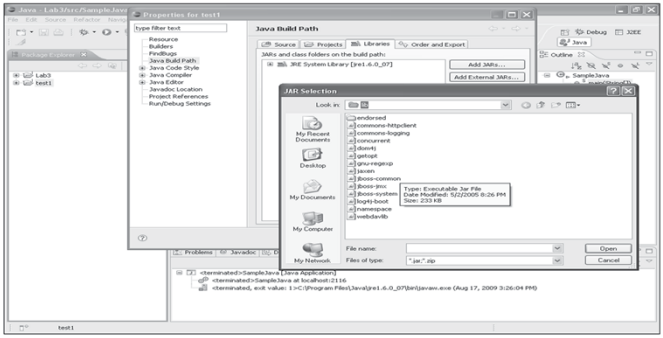
<u>Creating and Managing Java Projects</u>:

Debugging a Java Program:

- You may launch your Java programs from the workbench. The programs may be launched in either **run** or **debug** mode.
  - In **run** mode, the program executes. However, the execution may not be suspended or examined.
  - In **debug** mode, execution may be suspended and resumed, variables may be inspected, and expressions may be evaluated.
- Variables view gives contents of the program variables at different statements in execution.
- Breakpoints can be set for debugging, by opening the **marker-bar** pop-menu and selecting **Toggle Breakpoint**. While the Breakpoint is enabled, the thread execution suspends before the execution of the line happens. **Breakpointing** is a technique to set-up the starting point for program debugging.

2.4: Miscellaneous Options

Adding external jar file



IGATE
Speed.Agility.Imagination

<u>Miscellaneous Options</u>:

Adding an external jar file:

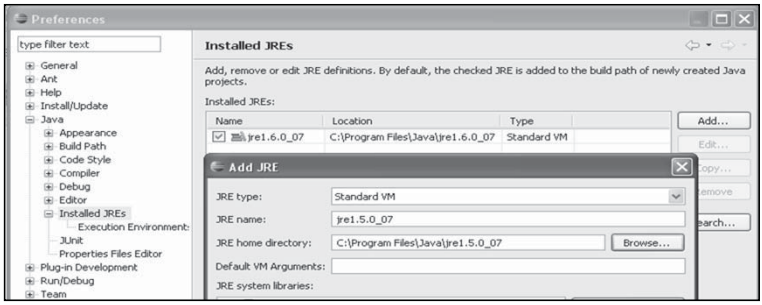
- When you are developing advanced Java programs, you might have to include some external jar files.
  - Click **Project Properties**.
  - Select **Java build Path**.
  - Click the **Libraries** tab, and click the **Add External Jar Files** button.
  - Locate the folder which contains the jar files, and click **Open**.

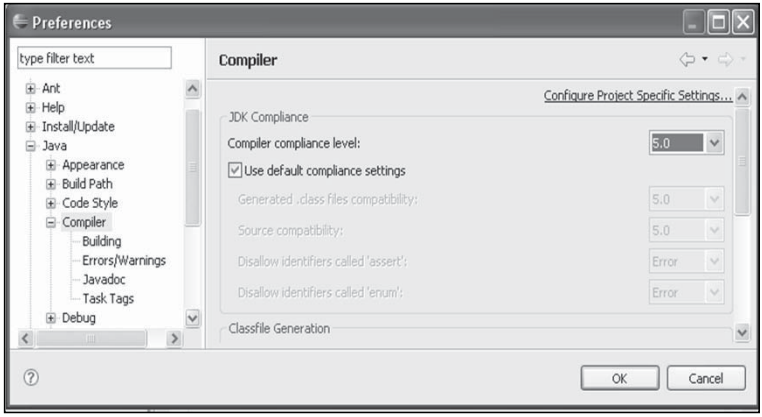
2.4: Miscellaneous Options

Verifying / Changing JRE Installation

- **Changing the JRE is a common need while working with Eclipse which can be achieved as follows:**
  - Select the menu item **Window → Preferences** to open the workbench preferences
  - Select **Java → Installed JREs** in tree pane on the left, to display the **Installed Java Runtime Environments preference** page
    - To add a new JRE, click the **Add** button, and select the new **JRE home directory**
  - Change the appropriate compiler.
    - Select **Java → Compiler** and select the appropriate compiler

IGATE Speed.Agility.Imagination





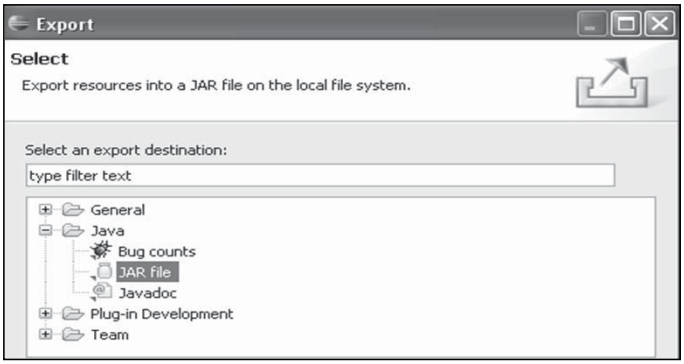
2.4: Miscellaneous Options

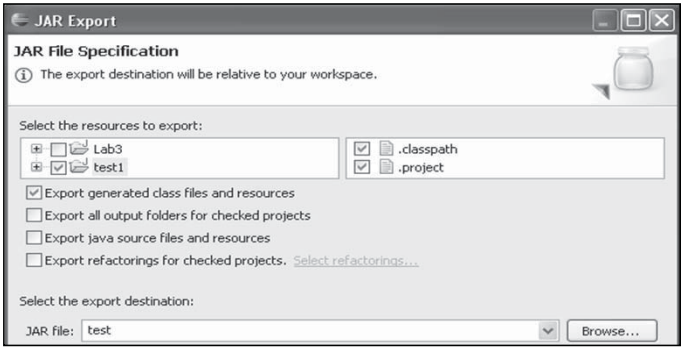
Jar File Creation

- **In the Package Explorer, you can optionally pre-select one or more Java elements to export**
  - Select **Export** from either the **Context** menu or from the **File** menu
  - Expand the Java node, and select **JAR** file, and click **Next**
  - On the **JAR File Specification** page, select the resources that you want to export
  - Specify a name to the JAR file
  - Click **Finish** to create the JAR file

June 11, 2015 | Proprietary and Confidential | - 24 -

IGATE Speed.Agility.Imagination





2.4: Miscellaneous Options

Class path Setting

- **Classpath variables allow you to avoid references to the location of a JAR file on your local file system**
- **Classpath variables can be used in a** Java Build Path **to avoid a reference to the local file system**
- **The value of such variables is configured at the following path:**
  - **Window → Preferences → Java → Build Path → Classpath Variables**

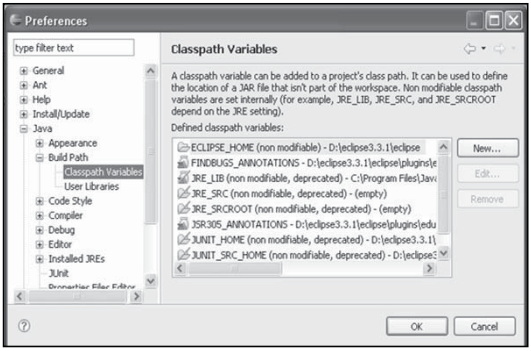
June 11, 2015 | Proprietary and Confidential | - 25 -

IGATE Speed.Agility.Imagination

<u>Miscellaneous Options</u>:

Setting Classpath:

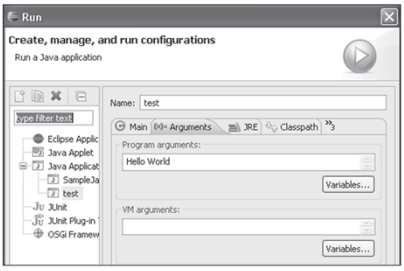
Command	Description
1. **New...**	It adds a new variable entry. In the resulting dialog, specify a name and path for the new variable. You can click the File or Folder buttons to browse for a path.
2. **Edit...**	It allows you to edit the selected variable entry. In the resulting dialog, edit the name and/or path for the variable. You can click the File or Folder buttons to browse for a path.
3. **Remove...**	It removes the selected variable entry.



2.4: Miscellaneous Options

Passing Command Line Arguments

- **Command line arguments can be passed to the program in the following ways:**
  - Select **Run** → **Open Run** dialog → **Arguments** tab



IGATE Speed.Agility.Imagination

Note: In the snapshot shown in the above slide, there are two arguments **"Hello"** and **"World"**.

2.4: Miscellaneous Options

Import a Project

- **To import an existing project to the workspace:**
  - Go to **File → Import**
  - Select **Existing Projects into Workspace** option
  - Select the radio button next to **Select archive file**, and click the **Browse** button
  - Find the archive file on your hard disk and click **Open** to select
  - If you have selected an archive file containing an entire Eclipse project, then the project name will appear in the box below, that is already checked
  - Click **Finish** to perform the import

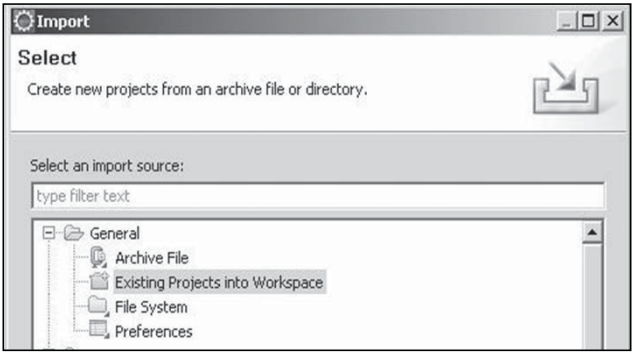
June 11, 2015 | Proprietary and Confidential | - 27 -

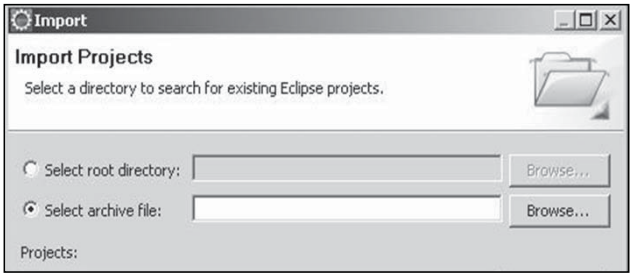
IGATE Speed.Agility.Imagination

<u>Import and Export Options</u>:

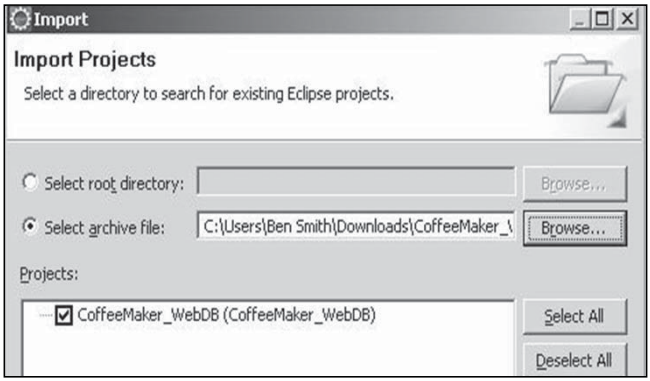
Import a Project:

The snapshots of the above steps are given below:





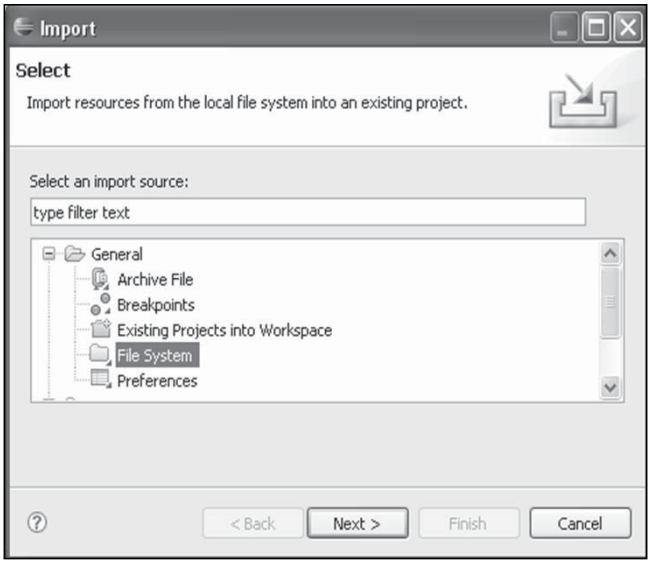
Import and Export Options:
Import a Project (contd.):



Importing Files into Project
Files can be imported into the Workbench either by:

- Dragging and dropping from the file system, or
- Copying and pasting from the file system, or
- Using the Import wizard

Using **drag and drop** or **copy and paste** to import files relies on operating system support and that is not necessarily available on all platforms. If the platform you are using does not have this support, then you can always use the **Import wizard**.



2.4: Miscellaneous Options

Build options

- **By default, builds are performed automatically when you save resources**
- **Two types of Build are available, namely:**
  - **Auto Build:** By selecting **Project → Build automatically**
  - **Manual build:** By deselecting **Project → Build automatically**
    - It is desirable in cases where you know building should wait until you finish a large set of changes
- **To build all the resources from the scratch you have to select** Project → Clean

June 11, 2015 | Proprietary and Confidential | - 29 -

IGATE Speed.Agility.Imagination

Builds:

- Builders create or modify workspace resources, usually based on the existence and state of other resources. They are a powerful mechanism for enforcing the constraints of some domain.
  For example: A Java builder converts Java source files (.java files) into executable class files (.class files), a web link builder updates links to files whose name/location have changed, and so on.
- As resources are created and modified, builders are run and the constraints are maintained. This transform need not be one to one.
  For example: A single .java file can produce several .class files.

Auto-build versus Manual Build:

- There are two distinct user work modes with respect to building:
  - Auto-build
  - User initiated manual build
- If you do not need fine-grained control over when builds occur, you can turn on **auto-building**. By keeping auto-building on, builds occur after every set of resource changes (for example, saving a file, importing a ZIP, ...). Auto-building is efficient because the amount of work done is proportional to the amount of change done. The benefit of auto-building is that your derived resources (for example, Java .class files) are always up to date. Auto-building is turned on/off via the **Build automatically** option accessed as **Project → Build automatically** option.
- If you need more control over when builds occur, you can turn off auto-building and invoke builds **manually**. This is sometimes desirable in cases where, for example, you know building is of no value until you finish a large set of changes. In this case, there is no benefit in paying the cost of auto-building.
- Auto-building always uses incremental building for efficiency.
- A clean build (**Project → Clean**) discards any existing built state. The next build after a clean will transform all resources according the domain rules of the configured builders.

2.4: Miscellaneous Options

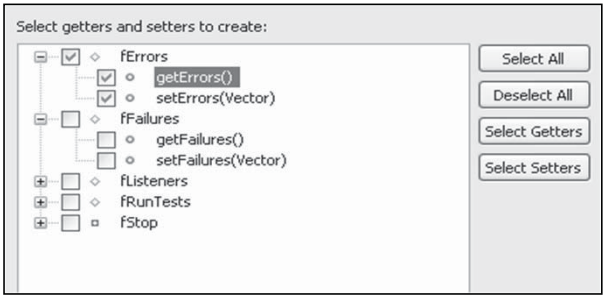
General Tips and Tricks

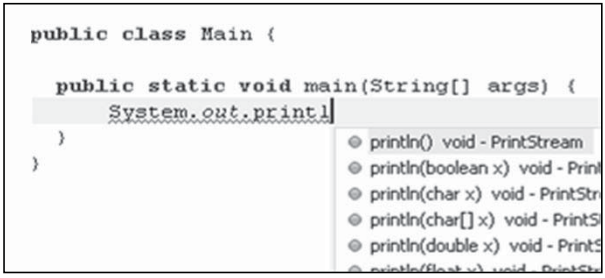
- **Creating Getters and Setters:**
  - To create getter and setter methods for a field:
    - Select the field's declaration
    - Invoke **Source → Generate Getter and Setter**
- **Content assist:**
  - Content assist provides you with a list of suggested completions for partially entered strings
  - In the Java editor, press **CTRL+SPACE** or invoke **Edit → Content Assist**

June 11, 2015 | Proprietary and Confidential | - 31 -

IGATE Speed.Agility.Imagination

<u>Miscellaneous Options</u>:
Tips and Tricks:
Creating getters and Setters :
Select the field's declaration, and invoke **Source → Generate Getter and Setter**.





2.4: Miscellaneous Options

General Tips and Tricks

- **Source menu contains a lot of options which can be used during code generation:**
  - **Code Comments:** You can quickly add and remove comments in a Java expression
  - **Import Statements:** You can use it to clean up unresolved references, add import statements, and remove unneeded ones
  - **Method Stubs:** You can create a stub for an existing method by dragging it from one class to another

2.4: Miscellaneous Options

General Tips and Tricks

- **Try / Catch statements:** You can create Try / Catch block for expression by Source → Surround with try/catch
- **Javadoc Comments:** You can generate Javadoc comments for classes and methods with Source → Add Javadoc Comment
- **Superclass constructor:** Add the superclass constructors with Source → Add Constructor from Superclass

2.4: Miscellaneous Options

Using Java documentation

- **For new developers, to quickly get familiar with the Java API, Java provides API documentation.**
- **The documentation also provides description and examples for all methods of each class.**
- **It can be downloaded from** http://docs.oracle.com/javase/8/docs/api/ **for offline access.**
- **To see Java documentation for any class or method, eclipse provides "javadoc" view.**
- **To enable this view, select Windows → Show View → Javadoc.**
- **You can also view the javadoc contents in HTML format by using shortcut key "Shift + F2".**

June 11, 2015 | Proprietary and Confidential | - 34 -

IGATE Speed.Agility.Imagination

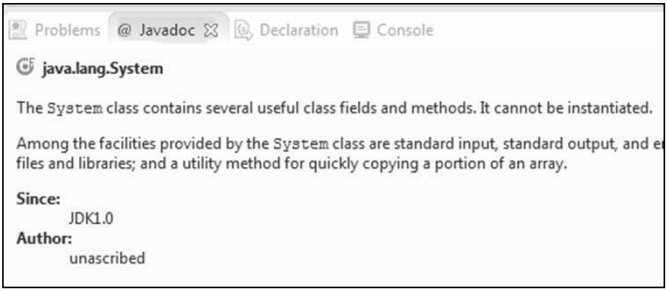
Using Java documentation:

Java API provides documentation which is good resource to get familiar with it. This documentation is valuable resource for programmers wishing to construct the applications in Java.

The documentation provides all information about the Java API. It includes list of classes or objects available to programmer.

Javadoc View:

Eclipse provides the javadoc view which shows the available documentation for selected class or method. To enable this view, select Windows → Show View → Javadoc.



Lab

- **Lab 1: Working with Java & Eclipse**

Summary

- **In this lesson, you have learnt:**
  - The method to install Eclipse
  - Process to create a Java Project with Eclipse
  - Various useful features of Eclipse

Review Question

- **Question 1: Which of the following are true with Eclipse 4.4?**
  - **Option 1:** A Java Project in Eclipse has got a Java builder that can incrementally compile Java source files as they are changed
  - **Option 2:** A workspace can have one project only
  - **Option 3:** The source and class files can be kept in different folders
- **Question 2: To build all resources, even those that have not changed since the last build, you have to select the following option:**
  - **Option1:** Project → Build Project
  - **Option2:** Project → Build All
  - **Option3:** Project → Clean

Add the notes here.

Core Java 8 with JAXB and Development Tools

Language Fundamentals

IGATE
Speed.Agility.Imagination

Lesson Objectives

- **After completing this lesson, participants will be able to:**
  - Understand Basic Java Language constructs like:
    - Keywords
    - Primitive Data Types
    - Operators
    - Variables
    - Literals
  - Write Java programs using control structures

Lesson Outline:

3.1: Keywords
3.2: Primitive Data Types
3.3: Operators and Assignments
3.4: Variables and Literals
3.5: Flow Control: Java's Control Statements
3.6: Best Practices

3.1 : Keywords

Keywords in Java

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

IGATE Speed.Agility.Imagination

Keywords are reserved identifiers that are predefined in the language and cannot be used to denote other entities. E.g. class, boolean, abstract, do, try etc. Incorrect usage results in compilation errors.

In addition, three identifiers are reserved as predefined literals in the language: null, true and false.

The table above shows the keywords available in Java 5.

3.2: Primitive Data types

Java Data types

Type	Size/Format	Description
byte	8-bit	Byte-length integer
short	16-bit	Short Integer
int	32-bit	Integer
long	64-bit	Long Integer
float	32-bit IEEE 754	Single precision floating point
double	64-bit IEE 754	Double precision floating point
char	16-bit	A single character
boolean	1-bit	True or False

June 11, 2015 | Proprietary and Confidential | - 4 -

IGATE Speed.Agility.Imagination

There are two data types available in Java:
Primitive Data Types.
Reference/Object Data Types :- is discussed later.

There are eight primitive data types supported by Java (see slide above). Primitive data types are predefined by the language and named by a key word.

The default character set used by Java language is **Unicode character** set and hence a **character data type** will consume **two bytes** of memory instead of a byte (a standard for ASCII character set). Unicode is a character coding system designed to support text written in diverse human languages.

This allows you to use characters in your Java programs from various alphabets such as Japanese, Greek, Russian, Hebrew, and so on. This feature **supports** a readymade support for **internalization** of java.

The default values for the various data types are as follows:

Integer : 0
Character : '\u0000'
Decimal : 0.0
Boolean : false
Object Reference: null

[Note: C or C++ data types pointer, struct, and union are not supported. Java does not have a typedef statement (as in C and C++).]

3.3 : Operators and Assignments

Operators in Java

- **Operators can be divided into following groups:**
  - Arithmetic
  - Bitwise
  - Relational
  - Logical
  - *instanceof* Operator

IGATE
Speed.Agility.Imagination

Java provides a rich set of operators to manipulate variables. These are classified into several groups as shown above.

Arithmetic Operators

Operator	Result
+	Addition
-	Subtraction (or unary) operator
*	Multiplication
/	Division
%	Modulus
++	Increment
+=	Addition assignment
-=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment
--	Decrement

IGATE Speed.Agility.Imagination

Arithmetic operators are summarized in the table above:
Integer division yields an integer quotient for example, the expression 7 / 4 evaluates to 1, and the expression 17 / 5 evaluates to 3. Any fractional part in integer division is simply discarded (i.e., truncated) no rounding occurs.
Java provides the remainder operator, %, which yields the remainder after division. The expression x % y yields the remainder after x is divided by y. Thus, 7 % 4 yields 3, and 17 % 5 yields 2. This operator is most commonly used with integer operands, but can also be used with other arithmetic types.
Parentheses are used to group terms in Java expressions in the same manner as in algebraic expressions. For example, to multiply a times the quantity b + c, we write **a*(b+c).**
If an expression contains nested parentheses, such as **((a+b)*c)** the expression in the innermost set of parentheses (a + b in this case) is evaluated first.
Order of Precedence:
Multiplication, division and remainder operations are applied first. If an expression contains several such operations, the operators are applied from left to right. **Multiplication, division and remainder operators have the same level of precedence**.
Addition and subtraction operations are applied next. If an expression contains several such operations, the operators are applied from **left to right**. **Addition and subtraction operators have the same level of precedence.**

Bitwise Operators

- **Apply upon *int*, *long*, *short*, *char* and *byte* data types:**

Operator	Result
~	Bitwise unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Shift right zero fill
<<	Shift left
&=	Bitwise AND assignment
|=	Bitwise OR assignment

The Java programming language also provides operators that perform bitwise and bit shift operations on integral types. The operators discussed in this section are less commonly used.
The unary bitwise complement operator "~" inverts a bit pattern; it can be applied to any of the integral types, making every "0" a "1" and every "1" a "0". For example, a byte contains 8 bits; applying this operator to a value whose bit pattern is "00000000" would change its pattern to "11111111".
The signed left shift operator "<<" shifts a bit pattern to the left, and the signed right shift operator ">>" shifts a bit pattern to the right. The bit pattern is given by the left-hand operand, and the number of positions to shift by the right-hand operand. The unsigned right shift operator ">>>" shifts a zero into the leftmost position, while the leftmost position after ">>" depends on sign extension.
Bitwise & operator performs a bitwise AND operation.
Bitwise ^ operator performs a bitwise exclusive OR operation.
Bitwise | operator performs a bitwise inclusive OR operation.

Relational Operators

- **Determine the relationship that one operand has to another.**
  - Ordering and equality.

Operator	Result
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

IGATE Speed.Agility.Imagination

A condition is an expression that can be either true or false. For example, the condition "grade is greater than or equal to 60" determines whether a student passed a test. If the condition in an if statement is true, the body of the if statement executes. If the condition is false, the body does not execute.

Conditions in if statements can be formed by using the equality operators (== and !=) and relational operators (>, <, >= and <=). Both equality operators have the same level of precedence, which is lower than that of the relational operators. The equality operators associate from left to right. The relational operators all have the same level of precedence and also associate from left to right.

Logical Operators

Operator	Result
&&	Logical AND
||	Logical OR
^	Logical XOR
!	Logical NOT
==	Equal to
?:	Ternary if-then-else

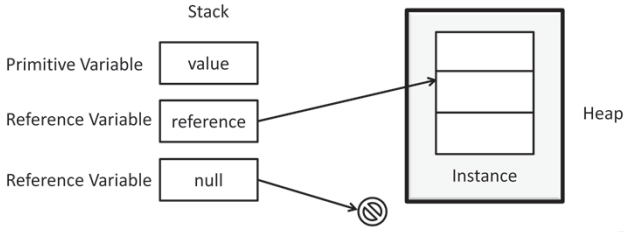
IGATE
Speed.Agility.Imagination

Java provides logical operators to enable programmers to form more complex conditions by combining simple conditions. The logical operators are && (conditional AND), || (conditional OR), & (boolean logical AND), | (boolean logical inclusive OR), ^ (boolean logical exclusive OR) and ! (logical NOT).

3.4: Variables and Literals

Variables

- **Variables are data placeholders.**
- **Java is a strongly typed language, therefore every variable must have a declared type.**
- **The variables can be of two types:**
  - reference types: A variable of reference type provides a reference to an object.
  - primitive types: A variable of primitive type holds a primitive.
- **In addition to the data type, a Java variable also has a name or an identifier.**



IGATE
Speed.Agility.Imagination

Types of Variables

- **Variable is basic storage in a Java program**
- **Three types of variables:**
  - Instance variables
    - Instantiated for every object of the class
  - Static variables
    - Class Variables
    - Not instantiated for every object of the class
  - Local variables
    - Declared in methods and blocks

June 11, 2015 | Proprietary and Confidential | - 11 -

IGATE Speed.Agility.Imagination

Instance variables: These are members of a class and are instantiated for every object of the class. The values of these variables at any instant constitute the *state* of the object.
Static variables: These are also members of a class, but these are not instantiated for any object of the class and therefore belong only to the class. We shall be covering the static modifier in later section.
Local variables: These are declared in methods and in blocks. They are instantiated for every invocation of the method or block. In Java, local variables must be declared before they are used.
Life-cycle of the variable is controlled by the scope in which those are defined.

Refer the example on the subsequent slide.

Types of Variables

```
public class Box {
private double dblWidth;
private double dblHeight;
private double dblDepth;
private static int boxid;
public double calcVolume() {
  double dblTemp;
  dblTemp = dblWidth * dblHeight * dblDepth;
  return dblTemp;
 }
}
```

Instance Variable

Static Variable

Local Variable

IGATE Speed.Agility.Imagination

Add the notes here.

Literals

- **Literals represents value to be assigned for variable.**
- **Java has three types of literals:**
  - Primitive type literals
  - String literals
  - null literal
- **Primitive literals are further divided into four subtypes:**
  - Integer
  - Floating point
  - Character
  - Boolean
- **For better readability of large sized values, Java 7 allows to include '_' in integer literals.**

June 11, 2015 | Proprietary and Confidential | - 13 -

IGATE Speed.Agility.Imagination

Literal Type	Example
Integer	int x = 10
Octal	int x = 0567
Hexadecimal	int x = 0x9E (to represent number 9E)
Long	long x = 9978543210L;
Binary	byte twelve = 0B1100; (to represent decimal 12)

Using Underscores	int million = 1_000_000;
	int twelve = 0B_1100;
	long multiplier = 12_34_56_78_90_00L;

Float	float x = 0.4f; float y = 1.23F, float z = 0.5e10;
Double	double x = 0.0D; double pi=3.14; double z=9e-9d;
Boolean	boolean member=true; boolean applied=false;
Character	char gender = 'm';
String	String str = "Hello World";
Null	Employee emp = null;

3.5: Flow Control: Java's Control Statements

Control Statements

- **Use control flow statements to:**
  - Conditionally execute statements
  - Repeatedly execute a block of statements
  - Change the normal, sequential flow of control
- **Categorized into two types:**
  - Selection Statements
  - Iteration Statements

IGATE Speed.Agility.Imagination

Java being a programming language, offers a number of programming constructs for decision making and looping.

Selection Statements

- **Allows programs to choose between alternate actions on execution.**
- **"if" used for conditional branch:**

```
if (condition) statement1;
else statement2;
```

- **"switch" used as an alternative to multiple "if's":**

```
switch(expression){
   case value1:   //statement sequence
                    break;
   case value2:   //statement sequence
                    break; ...
   default:         //default statement sequence
}
```

Expression can be of String type!

June 11, 2015 | Proprietary and Confidential | - 15 -

IGATE Speed.Agility.Imagination

If Statement:
The if statement is Java's conditional branch statement. It can be used to route program execution through two different paths.
Each statement may be a single statement or a compound statement enclosed in curly braces. The condition is any expression that returns a boolean value. The else clause is optional.
The if works like this: If the condition is true, then statement1 is executed. Otherwise, statement2 (if it exists) is executed. In no case will both statements be executed. For example, consider the following:

```
int a, b;
if(a < b) a = 0;
else b = 0;
```

switch case : an example

```
class SampleSwitch {
  public static void main(String args[]) {
    for(int i=0; i<=4; i++)
      switch(i) {
      case 0:
          System.out.println("i is zero."); break;
      case 1:
          System.out.println("i is one."); break;
      case 2:
          System.out.println("i is two."); break;
      case 3:
          System.out.println("i is three."); break;
      default:
          System.out.println("i is greater than 3.");
      }
  }}
```

Output:
i is zero.
i is one.
i is two.
i is three.
i is greater than 3.

IGATE Speed.Agility.Imagination

Switch – Case:
The *switch* statement is Java's multi-way branch statement. It provides an easy way to dispatch execution to different parts of your code based on the value of an expression.
The switch-expression must evaluate to any of the following type:

- byte
- short
- char
- int
- enum
- **String**.

As such, it often provides a better alternative than a large series of *if-else-if* statements.

Iteration Statements

- **Allow a block of statements to execute repeatedly**
  - While Loop: Enters the loop if the condition is true

```
while (condition)
{ //body of loop
 }
```

  - Do – While Loop: Loop executes at least once even if the condition is false

```
do
{ //body of the loop
} while (condition)
```


IGATE Speed.Agility.Imagination

while statement:
The body of the loop is executed as long as the conditional expression is true. When *condition* becomes false, control passes to the next line of code immediately following the loop. Example:

```
class Samplewhile {
        public static void main(String args[]) {
            int n = 5;
            while(n > 0) {
              System.out.print(n+"\t");
               n--;
      } } }
```

Output: 5 4 3 2 1
The while loop evaluates its conditional expression at the top of the loop. Hence, if the condition is false to begin with, the body of the loop will not execute even once.
do – while loop:
This construct executes the body of a **while** loop at least once, even if the conditional expression is false to begin with. The termination expression is tested at the **end** of the loop rather than at the beginning.

Iteration Statements

- For Loop:

```
for( initialization ; condition ; iteration)
{ //body of the loop }
```

- Example

```
// Demonstrate the for loop.
class SampleFor {
   public static void main(String args[]) {
      int number;
      for(number =5; number >0; n--)
         System.out.print(number +"\t");
   }
}
```

Output: 5 4 3 2 1

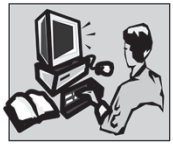
IGATE Speed.Agility.Imagination

<u>for loop:</u>
When the **for** loop first starts, the *initialization* portion of the loop is executed. Generally, this is an expression that sets the value of the *loop control variable,* which acts as a counter that controls the loop. The initialization expression is only executed once. Next, *condition* is evaluated. It usually tests the loop control variable against a target value. If this expression is true, then the body of the loop is executed, else the loop terminates. Next, the *incremental* portion of the loop is executed. This is usually an expression that increments or decrements the loop control variable. The loop then iterates, first evaluating the conditional expression, then executing the body of the loop, and then executing the iteration expression with each pass. This process repeats until the controlling expression is false.

3.5: Control Structures

Demo

- **Data types in Java**
- **Switch Statement using String as expression**



IGATE
Speed.Agility.Imagination

Add the notes here.

3.6: Best Practices

Best practices: Iteration Statements

- **Always use an int data type as the loop index variable whenever possible**
- **Use for-each liberally**
- **Switch case statement**
- **Terminating conditions should be against 0**
- **Loop invariant code motion**

E.g If you call length() in a tight loop, there can be a performance hit.

June 11, 2015 | Proprietary and Confidential | - 20 -

IGATE Speed.Agility.Imagination

Always use an int data type as the loop index variable whenever possible.
It is efficient when compared to using byte or short data types. This is because when we use byte or short data type as the loop index variable they involve implicit type cast to int data type.

Use for-each liberally.
The for-each loop is used with both collections and arrays. It is intended to simplify the most common form of iteration, where the iterator or index is used solely for iteration, and not for any other kind of operation, such as removing or editing an item in the collection or array.

When there is a choice, the for-each loop should be preferred over the for loop, since it increases legibility.

Switch Case statement.
One peculiar fact about switch statements is that code consisting of case with consecutive constants like 0,1,2 are faster than with case with constants like 2, 6, 7, 14, etc. This is because in the former switching between options requires less offset and it takes only 16 bytes. But in the later the offset is higher and it might take 32 or more bytes.

Terminating conditions should be against 0.
For example:

```
for(int i=0;i<length;i++){
    …… }

for(int i=length-1;i>=0;i--){
    …… }
```

The later one is 30% faster than the former. Just you are comparing with constant instead of accessing a.length everytime.

Loop invariant code motion.
If in a loop a expression is evaluated in every iteration the performance may be slow. This is called "Loop invariant code motion". Instead evaluate the expression outside the loop.
length() is a method in String used to obtain the number of characters currently in the string. If you call length()in a tight loop, there can be a performance hit.

Example:

```
for(int i =0;i <s.length(); i++) {   //AVOID
…
}

// precomputed length
            int len = s.length();
for(int i =0; i < len; i++) {
….
}
```

Summary

- **In this lesson you have learnt:**
  - Keywords
  - Primitive Data Types
  - Operators and Assignments
  - Variables and Literals
  - Flow Control: Java's Control Statements
  - Best Practices

Review Question

- **Question 1: Java considers variable number and NuMbEr to be identical.**
  - True/False
- **Question 2: The *do...while* statement tests the loop-continuation condition _____ it executes executing the loop's body; hence, the body executes at least once.**
  - **Option1:** before
  - **Option2:** after

Core Java 8 with JAXB and Development Tools

Classes and Objects

IGATE
Speed.Agility.Imagination

Lesson Objectives

- **After completing this lesson, participants will be able to:**
  - Define classes and objects
  - Create Packages
  - Work with Access Specifiers
  - Define Constructors
  - understand **this** reference
  - Understand memory management in java
  - use **static** keyword
  - Declaring and using Enum

Lesson Objectives:
This lesson introduces to the fundamentals of the Java Programming Language.

Lesson 4: Classes and Objects

4.1: Classes and Objects
4.2: Packages
4.3: Access Specifiers
4.4: Constructors
4.5: this reference
4.6: Memory Management
4.7: using static keyword
4.8: Enums
4.9: Best Practices

4.1 : Classes and Objects

Classes and Objects

- **Class:**
  - A template for multiple objects with similar features
  - A blueprint or the definition of objects
- **Object:**
  - Instance of a class
  - Concrete representation of class

```
class < class_name>
{
   type var1; ...
   Type method_name(arguments )
   {
      body
    } ...
} //class ends
```

June 11, 2015 | Proprietary and Confidential | - 3 -

IGATE Speed.Agility.Imagination

Classes describe objects that share characteristics, methods, relationships, and semantics. Each class has a name, attributes (its values determine state of an object), and operations (which provides the behavior for the object).

What is the relationship between objects and classes? What exists in real world is objects. When we classify these objects on the basis of commonality of structure and behavior, the result that we get are the classes.

Classes are “logical”. They don’t really exist in real world. When writing software programs, it is the classes that get defined first. These classes serve as a blueprint from which objects are created.

Reference: Refer to OOP material for a detailed discussion on Object-Oriented Programming Concept.
Note: Java does not have functions defined outside classes (as C++ does).

4.1 : Classes and Objects

Introduction to Classes

- **A class may consist the following elements:**
  - Fields
  - Methods
  - Constructors
  - Initializers

June 11, 2015 | Proprietary and Confidential | - 4 -

IGATE Speed.Agility.Imagination

A Java class may consist of the components as listed in the above slide. Fields and methods are primary components of any class and are termed as members of class. A class can have zero or more class members.

A class methods are categorized as business method or getters/setters. A business method of a class usually contains the business logic for the given problem/requirement.

A getter/setter usually written to access the private properties/attributes of a class. For example, consider the property given below:

```
private String name;
//getter
public String getName() { return name;}
//setter
public void setName(String name)
{this.name=name;}
```

For the boolean properties, the getter format isXXX()

Constructors are special methods (having same name as the class name and with no value return) which are used to create object of class. A class must have at least one constructor.

Initializers are special blocks in class, used to initialize members of class. A class can have zero or more initializers.

4.1 : Classes and Objects

Introduction to Classes

```
class Box{
        double dblWidth;
        double dblHeight;
        double dblDepth;
        double calcVolume(){
                return dblWidth * dblHeight * dblDepth;
        } //method calcVolume ends.
}//class Box ends.
```

```
class BoxDemo{
   public static void main(String a[])
   {
      Box box;                    //declare a reference to object
      box = new Box();            //allocate a memory for box object.
      box.calcVolume();           // call a method on that object.
} }
```

June 11, 2015 | Proprietary and Confidential | - 5 -

IGATE Speed.Agility.Imagination

The above slide shows an example of Box class with 3 fields and 1 method.

The new operator followed by the call to constructor of the class is used to create object. The above slide shows how to create an object of Box class.

new <<call to constructor>>;

Even though the Box class doesn't add constructor, Java compiler internally does. This constructor is called as default constructor and added by Java compiler only when class don't have any constructor declared.

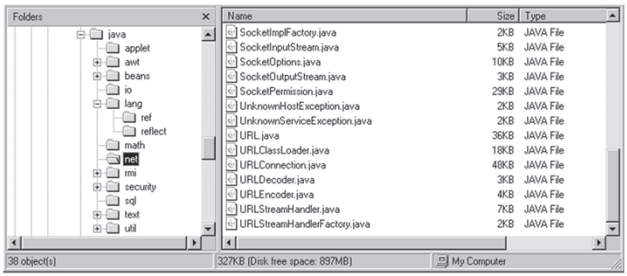
Box() { }

Once object created, we can access class members using dot operator. The contents of main() method shows how objects are instantiated and how methods are invoked.

4.2: Packages

Packages

- In Java, by the use of packages, you can group a number of related classes and/or interfaces together into a single unit.



June 11, 2015 | Proprietary and Confidential | - 6 -

IGATE Speed.Agility.Imagination

Packages:

When you work on some small projects you intend to put all java files into one single directory. It is quick, easy and harmless. However, if your project gets bigger, and the number of files increase, putting all these files into the same directory would be tedious for you. In java, you can avoid this sort of problem by using packages.

Basically, files in one directory (or package) have different functionality from those of another directory. For example, files in java.io package carry out functions related to I/O, but files in java.net package provide you the way to deal with the Network.

4.2: Packages

Benefits of Packages

- **These are the benefits of organising classes into packages:**
  - It prevents name-space collision.
  - It indicates that the classes and interfaces in the package are related.
  - You know where to find the classes you want if they're in a specific package.
  - It is convenient for organizing your work and separating your work from code libraries provided by others.

June 11, 2015 | Proprietary and Confidential | - 7 -

IGATE Speed.Agility.Imagination

Benefits of Packages:
Prevents name-space collision: One of the many concerns that programmers face today is, how to ensure that their source code does not conflict with the source code of other programmers. A typical example is the case in which two programmers define two distinct classes that have the same name. Suppose, you decide to write a List class that keeps a sorted list of objects. This would inherently conflict with the List class in the Java API that is used for displaying a list of items in a Graphical User Interface. To this problem, Java has a simple solution, which is known as namespace management. In namespace management, each programmer defines their own namespace and places their code within that namespace, thus two classes that have the exactly same name are now distinguishable since they occur in different namespaces. Namespaces are called packages in Java.
Provides greater control over source codes: Another important reason for using packages is that it provides programmers with greater control over their source code. It is typical to have a few thousand source files in medium to large scale applications. Trying to maintain them would be difficult, if not impossible. However, separating these source files into packages makes it much easier to manage the source code. Usually, related classes are grouped into a single package, for example, all the user interface classes of an application are grouped into a package.
Makes it easy to find a class: If you are looking for a specific class and you know about the functionality it provides, you will naturally be able to find it in the right package. For example, if you are looking for an InputStreamReader, you will find it in the input and output package, that is, java.io.

4.2: Packages

Creating Your Own Package

```
package com.igate.trg.demo;
public class Balance {
   String name;
   public Balance(String n) {
      name = n;
   }
 public void show() {
  ……
   if( bal < 0)
         System.out.println(name + ": $" + bal);
}
```

Package should be the first statement

June 11, 2015 | Proprietary and Confidential | - 8 -

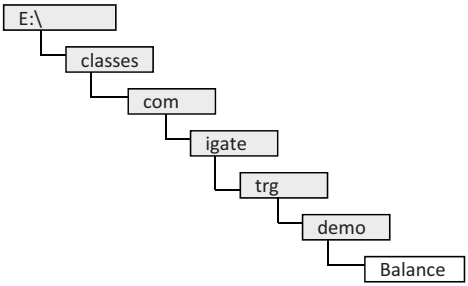
IGATE Speed.Agility.Imagination

Creating Your Own Package:
A package is a directory structure mapped on the operating system, hence compilation of Balance.java file results in the creation of directory structure com\igate\trg\demo and under that directory, Balance.class file is created. A package can contain more than one class and interface.

Compiling into a Package:
While compiling the file, we need to specify the URL where directory, com is to be created. For example, if you give the following command at command prompt.

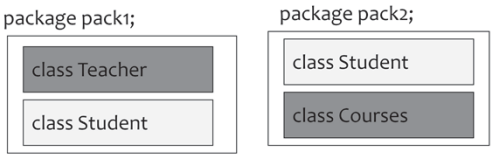
E:\yourdirectory> javac –d E:\classes Balance.java

It is expected that you have a directory called classes under E:\. the result of this command is this directory structure. Here, all the grayed boxes are directories.



4.2: Packages

Packages and Name Space Collision



- **Namespace collision can be avoided by accessing classes with the same name in multiple packages by their fully qualified name.**

```
import pack1.*;
import pack2.*;
pack1.Student stud1;
pack2.Student stud2;
Teacher teacher1;
Courses course1;
```


IGATE Speed.Agility.Imagination

4.2: Packages

Using Packages

- **Use fully qualified name.**

```
java.util.Date = new java.util.Date();
```

- **You can use import to instruct Java where to look for things defined outside your program.**

```
import java.util.Scanner;
Scanner sc = new Scanner (System.in);
```

You can use multiple import statements

- **You can use * to import all classes in package:**

```
import java.util.*;
Scanner sc = new Scanner (System.in);
```

Use * carefully; you may overwrite definitions

June 11, 2015 | Proprietary and Confidential | - 10 -

IGATE Speed.Agility.Imagination

java.lang package is automatically imported by every Java program.

4.2: Packages

Static Import

- **Static import enables programmers to import static members.**
- **Class name and a dot (.) are not required to use an imported static member.**

```
import static java.lang.Math.*;
public class StaticImportTest
{
 public static void main( String args[] )
 {
 System.out.printf( "sqrt( 900.0 ) = %.1f\n", sqrt( 900.0 ) );
 } // end main
}
```

Note: It's not Math.sqrt

June 11, 2015 | Proprietary and Confidential | - 11 -

IGATE Speed.Agility.Imagination

Static Import:

A static import declaration has two forms:
1. The form that imports a particular static member (which is known as single static import):
The following syntax imports only the sqrt method of the Math class:

```
import static java.lang.Math.sqrt;
```

2. The form that imports all static members of a class:
The following syntax imports all static members of the Math class:

```
import static java.lang.Math.*
```

4.2: Packages

Some Java Packages

Package Name	Description
java.lang	Classes that apply to the language itself, which includes the Object class, the String class, and the System class. It also contains the Wrapper classes. "Classes belonging to java.lang package need not be explicitly imported".
java.util	Utility classes, such as Date, as well as collection classes, such as Vector and Hashtable
java.io	Input & output classes for writing to & reading from streams (such as standard input and output) & for handling files
java.net	Classes for networking support, including Socket and URL (a class to represent references to documents on the WWW)
java.applet	Classes to implement Java applets, including the Applet class itself, as well as the AudioClip interface

IGATE
Speed.Agility.Imagination

Refer to the java documentation for more details on other important packages.

4.2: Packages

Demo : Package

- **Execute the following programs:**
  - Balance.java
  - AccountBalance.java
  - StaticImportDemo.java
  - StaticImportNotUsed.java

IGATE Speed.Agility.Imagination

```
package com.igate.lesson4.demo;
public class Balance {
   String name;
   double bal;
   public Balance(String n, double b) {
       name = n;
       bal = b;
   }
   public void show() {
     if(bal<0)
        System.out.println(name + ": $" + bal);
   }}
```

```
package com.igate.lesson4;
import com.igate.lesson4.demo.Balance;
class AccountBalance {
   public static void main(String args[]) {
      Balance object= new Balance("K. J. Fielding", 123.23);
     object.show();
   }
}
```

4.3: Access Modifiers

Types of Access Modifiers

- Default
- Private
- Public
- Protected

Location/Access Modifier	Private	Default	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

IGATE Speed.Agility.Imagination

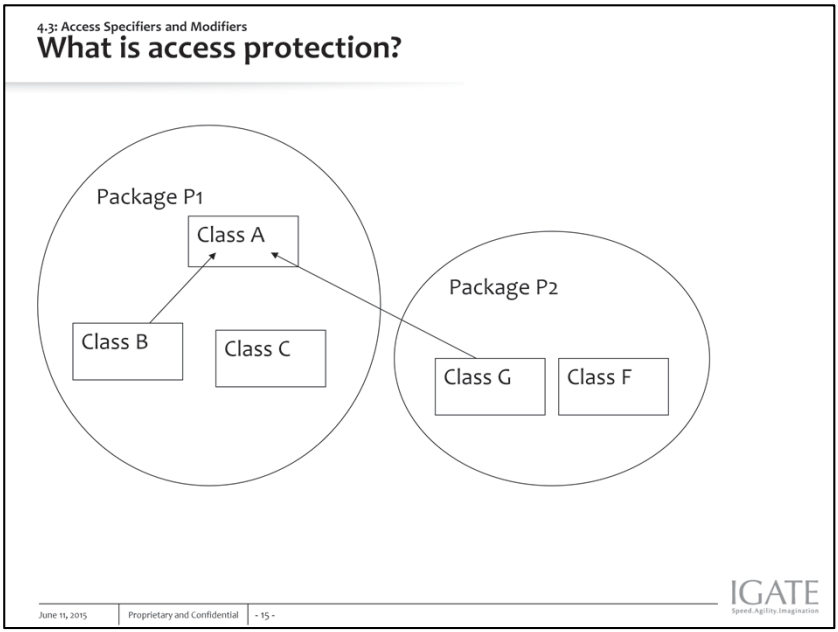
public access modifier
Fields, methods and constructors declared public (least restrictive) within a public class are visible to any class in the Java program, whether these classes are in the same package or in another package.

private access modifier
Fields, methods or constructors declared private (most restrictive) cannot be accessed outside an enclosing class. This modifier cannot be used for classes. It also cannot be used for fields and methods within an interface. A standard design strategy is to make all fields private and provide public getter methods for them.

protected access modifier
Fields, methods and constructors declared protected in a superclass can be accessed only by subclasses in other packages. Classes in the same package can also access protected fields, methods and constructors, even if they are not a subclass of the protected member's class. This modifier cannot be used for classes. It also cannot be used for fields and methods within an interface.

default access modifier
Default specifier is used when "no access modifier is present". Any class, field, method or constructor that has no declared access modifier is accessible only by classes in the same package. The default modifier is not used for fields and methods within an interface.

The table shown above is applicable only to members of classes. A class has only two possible access levels: default and public.
When a class is declared as public, it is accessible by any other code.
If a class has default access, then it can only be accessed by other code within its same package



Access Protection:

Classes and packages are both means of encapsulating and containing the name space and scope of variables and methods. Packages act as containers for classes. Classes act as containers for data and code. Accessibility in java is specified with respect to packages. Because of the interplay between classes and packages, Java addresses five categories of visibility for class members.

Same Class
Subclass in the same package
Non-subclasses in the same package
Subclasses in different package
Classes that are neither in the same package nor in subclasses.

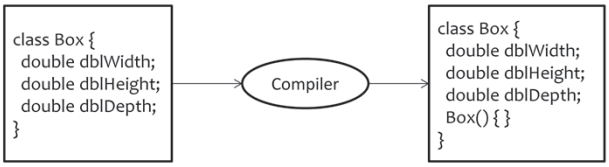
See the diagram given in the slide, to understand the five categories.
There are two packages P1 and P2. Package P1 contains Class A, B and C. B inherits from A. Package P2 contains Class G and F. G inherits from A. Which class comes under which category with respect to class A, is described below:

Same Class (Class A)
Subclass in the same package (Class B)
Non-subclasses in the same package (Class C)
Subclasses in different package (Class G)
Classes that are in neither the same package nor subclasses. (Class F)

4.4: Constructors

Default Constructors

- **All Java classes have *constructors***
  - Constructors initialize a new object of that type
- **Default no-argument constructor is provided if program has no constructors**
- **Constructors:**
  - Same name as the class
  - No return type, not even void



June 11, 2015 | Proprietary and Confidential | - 16 -

IGATE Speed.Agility.Imagination

Class basically holds blueprint of Objects. In order to create and instantiate objects of class and also to provide initial values for the object, constructors are used. The structure of a constructor looks similar to a method. Constructors are used to create objects of a class. A Java class must have at least one constructor.

A class can have many constructors in order to facilitate the object creation in different ways.

Default Constructor: If class doesn't declare any constructor, compiler adds a constructor to the class. This constructor is called as default constructor. The default constructor accepts no arguments. Above slide shows an example of default constructor.

4.4: Constructors

Demo

- **Execute the BoxDemo.java program.**
  - This uses the Box.java

IGATE
Speed.Agility.Imagination

This demo example contains default (no-arg) constructor as well as a constructor that takes 3 parameters. Three box objects are created with initialization done using constructors and setter methods.

4.5: this reference

this reference

- **The this keyword is used to refer to the current object from any method or constructor.**
- **There are mainly two uses of this keyword:**
  - Refer the class level fields
  - Chaining constructors

```
// Field reference using this
class Point {
      int xCord;            // instance variable
      int yCord;

      Point(int xCord, int yCord) {
         this.xCord = xCord;
         this.yCord = yCord;
      }
}
```

June 11, 2015 | Proprietary and Confidential | - 18 -

IGATE Speed.Agility.Imagination

this keyword:

this keyword is used to refer the current object. As shown in the above example, the constructor parameters are shadowing the instance variables. Therefore we can use this keyword to make difference between the local variables/parameters and instance variables.

The other use of this keyword is to invoke constructor of same class.

```
class Point {
        int xCord;
        int yCord;
        Point() {
            this(0, 0);      //chaining constructors using this
        }
        Point(int xCord, int yCord) {
            this.xCord = xCord;
            this.yCord = yCord;
        }
}
```

Note: this keyword cannot be used to refer static variables.

4.6: Memory Management

Memory Management

- Dynamic and Automatic
- No *Delete* operator
- Java Virtual Machine (JVM) de-allocates memory allocated to unreferenced objects during the garbage collection process

IGATE
Speed.Agility.Imagination

4.6: Memory Management

Enhancement in Garbage Collector

- **Garbage Collector:**
  - Lowest Priority Daemon Thread
  - Runs in the background when JVM starts
  - Collects all the unreferenced objects
  - Frees the space occupied by these objects
  - Call *System.gc()* method to "hint" the JVM to invoke the garbage collector
    - There is no guarantee that it would be invoked. It is implementation dependent

June 11, 2015 | Proprietary and Confidential | - 20 -

IGATE Speed.Agility.Imagination

There is a common misconception that system.gc() invokes the garbage collector, however that is not true. It just gives a request or hint to JVM to start garbage collector, but JVM may not start it immediately or even till end of the program execution. It is JVM implementation dependent issue, as to when it would start. It can even do some optimization by starting garbage collection, only when certain amount of memory is consumed etc. An object is eligible for garbage collection when there are no more references to that object. References that are held in a variable are naturally dropped when the variable goes out of scope. So, you can explicitly drop an object reference by setting the value of a variable whose data type is a reference type to null.

```
StringBuffer sb = new StringBuffer("hello");
System.out.println(sb);
// The StringBuffer object is not eligible for collection
sb = null;
// Now the StringBuffer object is eligible for collection
```

4.6: Memory Management

Finalize() Method

- **Memory is automatically de-allocated in Java**
- **Invoke *finalize()* to perform some housekeeping tasks before an object is garbage collected**
- **Invoked just before the garbage collector runs:**
  - protected void finalize()

IGATE Speed.Agility.Imagination

Note :
'protected' prevents access to finalize() by code defined outside its class. This method only approximates the working of C++'s destructor. There is no way to determine when the finalize() method will run. There is no concept of destructors in Java as is there in C++.
To add a finalizer to a class, you simply have to override the finalize() method from the object class and can write the code for finalization inside the finalize() method
Syntax

```
class A {
  protected void finalize() {
   super.finalize();
    //Write the code for finalization over here.
    . ...
  }
}
```

4.7: using static keyword

Static modifier

- **Static modifier can be used in conjunction with:**
  - A variable
  - A method
- **Static members can be accessed before an object of a class is created, by using the class name**
- **Static variable :**
  - Is shared by all the class members
  - Used independently of objects of that class
  - Example: static int intMinBalance = 500;

June 11, 2015 | Proprietary and Confidential | - 22 -

IGATE Speed.Agility.Imagination

Variables and methods marked with static modifier belong to the class rather than any particular instance. Ie, you do not have to instantiate the class to invoke a static method or access a static variable.

4.7: using static keyword

Static modifier

- **Static methods:**
  - Can only call other static methods
  - Must only access other static data
  - Cannot refer to this or super in any way
  - Cannot access non-static variables and methods
- **Static constructor:**
  - used to initialize static variables

Method main() is a static method. It is called by JVM.

June 11, 2015 | Proprietary and Confidential | - 23 -

IGATE Speed.Agility.Imagination

main() is called by JVM. Making this method static means the JVM does not have to create an instance of your class to start running code.

Static constructor is also known as static initialization block. This is a normal block of code enclosed in braces { } and preceded by the static keyword. This can appear anywhere in the class body. It is normally used to initialize static variables.

4.7: using static keyword

Static modifier

```
// Demonstrate static variables, methods, and blocks.
public class UseStatic {
   static int intNum1 = 3;                    // static variable
   static int intNum2;
static {                                      //static constructor
        System.out.println("Static block initialized.");
        intNum2 = intNum1 * 4;
    }
   static void myMethod(int intNum3) {       // static method
         System.out.println("Number3 = " + intNum3);
         System.out.println("Number1 = " + intNum1);
         System.out.println("Number2 = " + intNum2);
   }
public static void main(String args[]) {
   myMethod(42);
  }}
```

Output:
Static block initialized.
Number3 = 42
Number1 = 3
Number2 = 12

IGATE Speed.Agility.Imagination

4.8: Enums

Enums

- **ENUM representation**

  pre-J2SE 5.0

```
public static final int SEASON_WINTER = 0;
public static final int SEASON_SUMMER = 1;
public static final int SEASON_SUMMER = 2;
```

- **Problem?**
  - Not type safe (any integer will pass)
  - No namespace (SEASON_*)
  - Brittleness (how do add value in-between?)
  - Printed values uninformative (prints just int values)
- **Solution: New type of class declaration**
  - enum type has public, self-typed members for each enum constant

June 11, 2015 | Proprietary and Confidential | - 25 -

IGATE Speed.Agility.Imagination

In pre Java 5, the standard way to represent an enumerated type was the int Enum pattern. An example shown in the box above.

But this pattern suffers from problems such as:
Not typesafe – A season is just an int you can pass in any other int value where a season is required, or add two seasons together (makes no sense)!
No namespace - Constants of an int enum must be prefixed with a string (eg SEASON_) to avoid collisions with other int enum types.
Brittleness - Since int enums are compile-time constants, they are compiled into clients that use them. If a new constant is added between two existing constants or the order is changed, clients must be recompiled.
Printed values are uninformative - Since they are just ints, printing one out will get you a number, which tells you nothing about what it represents, or what type it is.

In J2SE5 the enumerations have become separate classes. Thus, they are type-safe, flexible to use. For example the above enum is now represented as :
enum Season { WINTER, SPRING, SUMMER, FALL }

4.8: Enums

Declaring Type Safe Enums

- **Permits variable to have only a few pre-defined values from a given list**
- **Helps reduce bugs in the code**
  - Example:

```
enum CoffeeSize { BIG, HUGE, OVERWHELMING };
CoffeeSize cs = CoffeeSize.BIG;
```

- ***cs* can have values *BIG, HUGE* and *OVERWHELMING* only**

June 11, 2015 | Proprietary and Confidential | - 26 -

IGATE Speed.Agility.Imagination

Enum lets you restrict a variable to having one of only a few pre-defined values—in other words, one value from an enumerated list. This can help reduce the bugs in your code. For instance, in your coffee shop application you might want to restrict your size selections to BIG, HUGE, and OVERWHELMING. If you let an order for a LARGE or a GRANDE slip in, it might cause an error. See the declaration above. With this, you can guarantee that the compiler will stop you from assigning anything to a CoffeeSize except BIG, HUGE, or OVERWHELMING.
Then, the only way to get a CoffeeSize is with a statement like the following:

```
CoffeeSize cs = CoffeeSize.BIG;
```

It is not required that enum constants be in all upper case, but borrowing from the Sun code convention that constants are named in upper case, it is a good idea.

Important point to ponder about enums:

enum can be declared with only a public or default modifier.
enums are not strings or integer type.
Enums can be declared as their own class, or enclosed in another class, and that the syntax for accessing an enum's members depends on where the enum was declared.
enum cannot be declared in functions.
A semicolon after an enum is optional

4.8: Enums

Enums with Constructors, Methods and Variables

- **Add constructors, instance variables, methods, and a *constant specific class body***
  - Example:

```
enum CoffeeSize {
    BIG(8), HUGE(10), OVERWHELMING(16);
    // the arguments after the enum value are "passed"
    // as values to the constructor
   CoffeeSize(int ounces) {
        this.ounces = ounces;
        // assign the value to an instance variable
}
```

June 11, 2015 | Proprietary and Confidential | - 27 -

IGATE Speed.Agility.Imagination

Because an enum really is a special kind of class, you can do more than just list the enumerated constant values. You can add constructors, instance variables, methods, and something really strange known as a constant specific class body. To understand why you might need more in your enum, think about this scenario: imagine you want to know the actual size, in ounces, that map to each of the three CoffeeSize constants.

For example, you want to know that BIG is 8 ounces, HUGE is 10 ounces, and OVERWHELMING is a whopping 16 ounces. You could make some kind of a lookup table, using some other data structure, but that would be a poor design and hard to maintain. The simplest way is to treat your enum values (BIG, HUGE, and OVERWHELMING), as objects that can each have their own instance variables. Then you can assign those values at the time the enums are initialized, by passing a value to the enum constructor.

4.8: Enums

Demo

- Demo: EnumMonths.java

June 11, 2015 | Proprietary and Confidential | - 28 -

IGATE Speed.Agility.Imagination

You can do much more than the plain representation you saw in the previous example. You can define a full-fledged class (dubbed an enum type). It not only solves all the problems mentioned previously, it also allows you to add arbitrary methods and fields to an enum type, to implement arbitrary interfaces, and more. Enum types provide high-quality implementations of all the Object methods. For more information on Enum please refer the appendix.

4.9: Best Practices

Constructor

- **Initializing fields to default values is redundant**
- **Constructors should not call *overridables***
- **Beware of mistaken field *redeclares***

```
public final class Quark {
//private String fName;
//private double fMass;
 public Quark(String aName, double aMass){
    fName = aName;
    fMass = aMass;
   }
//WITH redundant initialization to default values
private String fName = null;
private double fMass = 0;
}


>javap -c -classpath . Quark
```

June 11, 2015 | Proprietary and Confidential | - 29 -

IGATE Speed.Agility.Imagination

Initializing fields to default values is redundant.
In the declaration of a field, setting it explicitly to its default initial value is always redundant, and may even cause the same operation to be performed twice (depending on your compiler). Declaring and initializing object fields as null, for instance, is simply not necessary in Java.

(It is worth recalling here that Java defines default initial values only for fields, and not for local variables).

Constructors should not call overridables.
That is, they should only call methods that are private, static, or final.

Beware of mistaken field re-declares.
Beware of this simple mistake, which can be hard to track down : if a field is mistakenly redeclared within the body of a method, then the field is not being referenced, but rather a temporary local variable with the same name. A common symptom of this problem is that a stack trace indicates that a field is null, but a cursory examination of the code makes it seem as if the field has been correctly initialized.

4.9: Best Practices

Static and Constants

- Declare constants as static and final
- Static, final and private methods are faster
- If possible, use constants in *if* conditions

IGATE
Speed.Agility.Imagination

Declare constants as static and final.
Use static, if variable is not going to change. This reduces the memory space for the object. A single copy is shared among all the objects.
Static, final and private methods are faster.
Static, final and private functions are treated as inline functions. These types of methods are resolved faster than other types because resolutions are done only within the class. So, if the method is not to be overridden by the sub class then define it as final, static and private.
If possible, use constants in if conditions.
Conditional Compilation may be useful. Conditional compilation gets rid of the run time resolution of variables and their values.

```
class A {
    public static final constA = 1;
}
class B {
  public static void main(String arg[])  {
       if (A.constA == 1)  System.out.println("inside if ");
  }
}
```

In the above example, during the compilation of Class B the if condition is checked. Since the condition involves the final (constant), the time required for resolving the if condition and variables are not done at runtime, which can save some time. But if the value of const "constA" is changed ensure that both classes are compiled.

Lab

- **Lab 2: Language Fundamentals , Classes and Objects**

Summary

- **In this lesson you have learnt:**
  - Classes and Objects
  - Packages
  - Access Specifiers
  - Constructors - Default and Parameterized
  - this reference
  - Using static keyword
  - Enums
  - Best Practices

Add the notes here.

Review Questions

- **Question 1: Which of the following are the benefits of using Package?**
  - **Option1:** prevents name-space collision.
  - **Option2:** To implement security of contained classes.
  - **Option3:** Better code library management.
  - **Option4:** To increase performance of your class.
- **Question 2: Which of the following is true regarding enum?**
  - **Option1:** enum cannot be used inside methods.
  - **Option2:** enum need not have a semicolon at the end.
  - **Option3:** enum can be only declared with public or default access specifier.
  - **Option4:** All the above are true.

Core Java 8 with JAXB and Development Tools

Exploring Basic Java Class Libraries

IGATE
Speed.Agility.Imagination

Lesson Objectives

- **After completing this lesson, participants will be able to:**
  - Understand The Object Class and different Wrapper Classes
  - Use Type casting
  - Work with Scanner, System Class and String Handling
  - Understand new Date and Time API

Lesson Objectives:
This lesson introduces to the fundamental Java API that is used in almost every type of Java applications.

Lesson 5: Exploring Java Basics
- 5.1: The Object Class
- 5.2: Wrapper Classes
- 5.3: Type casting
- 5.4: Using Scanner Class
- 5.5: The System Class
- 5.6: String Handling
- 5.7: Date and Time API
- 5.7: Best Practices

5.1: The Object Class

The Object Class

- **Cosmic super class**
- **Ultimate ancestor**
  - Every class in Java implicitly extends Object
- **Object type variables can refer to objects of any type:**

  Example:

```
Object obj = new Emp();
```

June 11, 2015 | Proprietary and Confidential | - 3 -

IGATE Speed.Agility.Imagination

The Object super class is a cosmic super class. Every class in Java extends Object class. It means a reference of object type can refer to an object of any other class. In addition, an array can be referenced by an object reference.

Example:

```
Object Obj1 = new Box(…….);
```

5.1: The Object Class

Object Class Methods

Method	Description
`boolean equals(Object)`	Determines whether one object is equal to another
`void finalize()`	Called before an unused object is recycled.
`class getClass()`	Obtains the class of an object at run time.
`int hashCode()`	Return the hashcode associated with the invoking object.
`String toString()`	Returns a string that describes the object

June 11, 2015 | Proprietary and Confidential | - 4 -

IGATE Speed.Agility.Imagination

The table above shows some of the methods of the Object superclass. Please refer to Java docs for the entire list.

5.2: Wrapper Classes

Wrapper Classes

- **Correspond to primitive data types in Java**
- **Represent primitive values as objects**
- **Wrapper objects are immutable**

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

June 11, 2015 | Proprietary and Confidential | - 5 -

IGATE Speed.Agility.Imagination

Wrapper classes correspond to the primitive data types in the Java language. These classes represent the primitive values as objects. Wrapper objects are immutable. This means that once a wrapper object has a value assigned to it, that value cannot be changed.
Java uses simple or primitive data types, such as int, char and Boolean etc. These data types are not part of the object hierarchy. They are passed by value to methods and cannot be directly passed by reference. However, at times there is a need to create an object representation of these simple data types. Java provides classes that correspond to each of these simple types. These classes encapsulate, or wrap, the simple data type within a class. Thus, they are commonly referred to as wrapper classes. The abstract class Number defines a superclass that is implemented by all numeric wrapper classes.

5.2: Wrapper Classes

Integer Wrapper Class

- **Integer class wraps a value of primitive type "int" into an object**
- **This class also provides several methods to convert int to String and vice versa**
- **Important methods of Integer class:**
  - intValue() : retrieves primitive int value of the Integer object
  - compareTo(): compares two Integer Objects
  - parseInt(): static method used to convert String value to int
  - toString(): retrives as String value from Integer object
  - isNaN(): check whether the given values is number or not
- **Important Constants of Integer class:**
  - MAX_VALUE: represents largest value of Integer class range
  - MIN_VALUE: represent lowest value of Integer class range

```
String strValue = "1234";
int num = Integer.parseInt(strValue);
```


IGATE Speed.Agility.Imagination

Above slide represents methods and constants provided by Integer wrapper class. Other wrapper classes too have similar kind of method and constant structure. The method names varies according to the wrapper class name. For example, intValue() method is anonymous to longValue() of Long wrapper class.

5.3: Type casting

Casting for Conversion of Data type

- **Casting operator converts one variable value to another where two variables correspond to two different data types**

```
variable1 = (variable1) variable2
```

- **Here, *variable2* is typecast to *variable1***
- **Data type can either be a *reference* type or a *primitive* one**

IGATE
Speed.Agility.Imagination

5.3: Type casting

Casting Between Primitive Types

- **When one type of data is assigned to another type of variable, *automatic type conversion* takes place if:**
  - Both types are compatible
  - Destination type is larger than the source type
  - No explicit casting is needed (widening conversion)

```
int a=5; float b; b=a;
```

- **If there is a possibility of data loss, explicit cast is needed:**

```
int i = (int) (5.6/2/7);
```

June 11, 2015 | Proprietary and Confidential | - 8 -

IGATE Speed.Agility.Imagination

When one type of data is assigned to another type of variable, an automatic type conversion will take place if the following two conditions are met:

The two types are compatible.

The destination type is larger than the source type.

In this case, a widening conversion takes place. For example, the int type is always large enough to hold all valid byte values, so no explicit cast statement is required.

To widen conversions, numeric types, including integer and floating-point types, are compatible with each other. However, numeric types are not compatible with char or boolean. Also, char and boolean are not compatible with each other.

As mentioned earlier, Java also performs an automatic type conversion when storing a literal integer constant into variables of type byte, short, or long.

Casting Incompatible Types

Automatic type conversions may not fulfill all needs though. For example, assigning an int value to a byte variable. This conversion will not be performed automatically, because a byte is smaller than an int. This is called a narrowing conversion, since you are explicitly making the value narrower so that it will fit into the target type. This is done using a cast - an explicit type conversion. It has this general form: (target-type) value

Here, target-type specifies the desired type to convert the specified value to. For example, the following fragment casts an int to a byte:

```
int i = 125;  byte b;
b = (byte) i;
```

5.3: Type casting

Casting Between Reference Types

- One class types involved must be the same class or a subclass of the other class type
- Assignment to different class types is allowed only if a value of the class type is assigned to a variable of its superclass type
- Assignment to a variable of the subclass type needs explicit casting:

```
String StrObj = Obj;
```

- Explicit casting is not needed for the following:

```
String StrObj = new String("Hello");
Object  Obj = StrObj;
```

June 11, 2015 | Proprietary and Confidential | - 9 -

IGATE Speed.Agility.Imagination

The first rule of casting between reference types is that one of the class types involved must be the same class as, or a subclass of, the other class type. Assignment to different class type is allowed only if a value of the class type is assigned to a variable of its superclass type. Assignment to a variable of the subclass type needs explicit casting.

```
Object Obj = new Object ( );
String StrObj = Obj;
/*    The second statement of the above code is not a legitimate one,
because class String is a subclass of class object.  So, an explicit
casting is needed. This is called as downcasting
*/
String StrObj = (String) Obj;
//The reverse statement will not require casting. It is upcasting
String StrObj = new String("Hello");
Object  Obj = StrObj;
```

Rule number one in casting is that you cannot cast a primitive type to a reference type, nor can you cast the other way around. If a casting violation is detected at runtime, the exception ClassCastException is thrown.

5.3: Type casting

Casting Between Reference Types (contd..)

- **Two types of reference variable castings:**
  - Downcasting:

```
Object Obj = new Object ( );
String StrObj = (String) Obj;
```

  - Upcasting:

```
String StrObj = new String("Hello");
Object  Obj = StrObj;
```


IGATE
Speed.Agility.Imagination

5.4: Using Scanner Class

Scanner Class

- Prior to Java 1.5 getting input from the console involved multiple steps.
- Java 1.5 introduced the *Scanner* class to simplify console input.
- Also reads from files and Strings (among other sources).
- Used for powerful pattern matching.
- Scanner is in the Java.util package; therefore needs to be imported

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


IGATE
Speed.Agility.Imagination

5.4: Using Scanner Class

Creating Scanner Objects

- Scanner(File source): Constructs a new Scanner that produces values scanned from the specified file.
- Scanner(InputStream source): Constructs a new Scanner that produces values scanned from the specified input stream.
- Scanner(Readable source): Constructs a new Scanner that produces values scanned from the specified source.
- Scanner(String source): Constructs a new Scanner that produces values scanned from the specified string.

5.4: Using Scanner Class

How to use Scanner class?

- **Scanner class basically parses input from the source into tokens by using delimiters to identify the token boundaries.**
- **The default delimiter is whitespace.**
- **Example:**

```
Scanner sc = new Scanner (System.in);
int i = sc.nextInt();
System.out.println("You entered" + i);
```


IGATE
Speed.Agility.Imagination

Example 1:

```
import java.util.Scanner;
public class ParseString
{
  public static void main(String[] args)
  {
    private static Scanner scanner = new
                                Scanner("1 2 3 4 5 6 7 8");

    while (scanner.hasNextInt()) {
      int num = scanner.nextInt();

      if (num % 2 == 0)
        System.out.print(num);
    }
  }
}
```

Output:
2
4
6
8

5.4: Using Scanner Class

Scanner class : nextXXX() Methods

- String next()
- boolean nextBoolean()
- byte nextByte()
- double nextDouble()
- float nextFloat()
- int nextInt()
- String nextLine()
- long nextLong()
- short nextShort()

June 11, 2015 | Proprietary and Confidential | - 14 -

IGATE Speed.Agility.Imagination

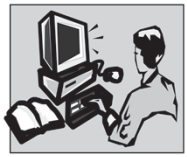
String next(): Finds and returns the next complete token from this scanner.
boolean nextBoolean(): Scans the next token of the input into a boolean value and returns that value.
byte nextByte(): Scans the next token of the input as a byte.
double nextDouble(): Scans the next token of the input as a double.
float nextFloat(): Scans the next token of the input as a float.
int nextInt(): Scans the next token of the input as an int.
String nextLine(): Advances this scanner past the current line and returns the input that was skipped.
long nextLong(): Scans the next token of the input as a long.
short nextShort(): Scans the next token of the input as a short.

InputMismatchException: This exception can be thrown if you try to get the next token using a next method that does not match the type of the token

5.4: Using Scanner Class

Demo : How to use Scanner class?

- **Execute**
  - ScannerDemo.java program
  - ParseString.java program



June 11, 2015 | Proprietary and Confidential | - 15 -

IGATE Speed.Agility.Imagination

When you create an instance of the Scanner class, the default delimiter is whitespace. The Scanner class provides the useDelimiter method for specifying the Delimiter.

```
import java.util.Scanner;
public class ParseString
{
   public static void main(String[] args)
   {
     Scanner scanner =  new Scanner("1, 2, 3, 4, 5, 6,
7,8").useDelimiter(", ");

      while (scanner.hasNextInt()) {
         int num = scanner.nextInt();

         if (num % 2 == 0)
            System.out.println(num);
      }
   }
}
```

5.5: The System Class

The System Class

- **Used to interact with any of the system resources**
- **Cannot be instantiated**
- **Contains a methods and variables to handle system I/O**
- **System class provides facilities like Standard input, Standard output and Error output streams**

Method	Description
`void currentTimeMillis()`	Returns the current time in terms of milliseconds since midnight, January 1, 1970
`void gc()`	Initiates the garbage collector.
`void exit(int code)`	Halts the execution and returns the value of integer to parent process usually to an operating system.

June 11, 2015 | Proprietary and Confidential | - 16 -

IGATE Speed.Agility.Imagination

Note : System.gc() is a suggestion and not a command. It is not guaranteed to cause the garbage collector to collect everything.

5.5: The System Class

Demo

- **Execute the Elapsed.java program**

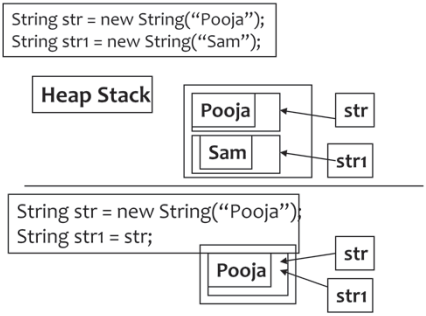
IGATE
Speed.Agility.Imagination

```
class Elapsed {
  public static void main(String args[]) throws IOException {
    long start, end;
    int i = 0, sum = 0;
    String str = null;
     System.out.println("Timing a for loop from 0 to 1,000,000");
     // time a for loop from 0 to 1,000,000
    start = System.currentTimeMillis(); // get starting time
    for(int j=0; j < 1000000; j++) ;
   end = System.currentTimeMillis(); // get ending time
   System.out.println("Elapsed time: " + (end-start));
   // Demo to read from the system input and write to standard output.
   BufferedReader br = new
                    BufferedReader(new InputStreamReader(System.in));
    do {
        System.out.println("Enter 0 to quit");
        str = br.readLine();
         i = Integer.parseInt(str);
         if ( i == 0 )   System.exit(0); // normal exit
         sum += i;
         System.out.println("Current sum is: " + sum);
    } while(i != 0);
}}
```

5.6: String Handling

String Handling

- **String is handled as an object of class String and not as an array of characters**
  - **String class is a better & convenient way to handle any operation**
  - **String objects are immutable**



IGATE Speed.Agility.Imagination

String is not an array of characters but it is actually a class and is part of core API. We can use the class String as a usual data-type. It can store up to 2 billion characters.
Note: A String in java is not equivalent to character array.
Strings are built-in objects & thus have a full complement of features that make string handling convenient. For example, Java has methods to compare two strings, search for a sub string, concatenate two strings etc. In addition, String objects can be constructed in number of ways.

String objects are immutable objects. That is, once you create a String object you cannot change the characters, which are part of String. This seems to be a major restriction, but that is not the case. Every time you perform some modification operation on the object, a new String object is created that contains the modifications. The original string is left unchanged. Hence, the number of operations performed on one particular string creates those many string objects.

For those cases in which modifiable string is desired, there is a companion class to String called StringBuffer, whose objects contain strings that can be modified after they are created.

5.6: String Handling

Important Methods

- **length(): length of string**
- **indexOf(): searches an occurrence of a char, or string within other string**
- **substring(): Retrieves substring from the object**
- **trim(): Removes spaces**
- **valueOf(): Converts data to string**
- **isEmpty(): Added in Java 6 to check whether string is empty or not**
- **concat(String s) : Used to concatenate a string to an existing string. Eg**

```
String string = "Core ";
System.out.println( string=string.concat(" Java") );
Output -> "Core Java"
```

June 11, 2015 | Proprietary and Confidential | - 19 -

IGATE Speed.Agility.Imagination

String.isEmpty() method added in Java 6 to check whether the given string is empty or not. Code prior to JDK 6 is as shown below:

```
//code prior to JDK 6

public boolean checkStringForEmpty(String str) {
   If(str.equals("")) {              //str.length==0
      return true;
   else
      return false;
}


//now with JDK 6 enhancement

public boolean checkStringForEmpty(String str) {
   If(str.isEmpty()) {               //much faster than the previous code
      return true;
   else
      return false;
}
```

5.6: String Handling

String Concatenation

- **Use a "+" sign to concatenate two strings Examples:**

```
Example: String string = "Core " + "Java";   -> Core Java
```

  - *String concatenation* operator if one operand is a string:

```
String a = "String"; int b = 3; int c=7
System.out.println(a + b + c);  -> String37
```

  - *Addition* operator if both operands are numbers:

```
System.out.println(a + (b + c)); -> String10
```

June 11, 2015 | Proprietary and Confidential | - 20 -

IGATE Speed.Agility.Imagination

The concat() method seen in previous page allows one string to be concatenated to another. But Java also supports string concatenation with the "+" operator.
In general, Java does not support operator overloading. The exception to this rule is the + operator, which concatenates two strings, and produces a new string object as a result.

```
class SimpleString {
   public static void main(String args[]) {
      // Simple String Operations
      char c[] = {'J', 'a', 'v', 'a'};
      String s1 = new String(c); // String constructor using
      String s2 = new String(s1);
      // String constructor using string as arg.
      System.out.println(s1);
      System.out.println(s2);
      System.out.println("Length of String s2 : " + s2.length());
      System.out.println("Index of v : " + s2.indexOf('v'));
      System.out.println("s2 in uppercase : " + s2.toUpperCase());
      System.out.println("Character at position 2 is  : " +  s2.charAt(1));
      // Using concatenation to prevent long lines.
       String longStr = "This could have been " +
                  "a very long line that would have " +
                  "wrapped around.  But string concatenation " +
                  "prevents this.";
       System.out.println(longStr);
   } }
```

5.6: String Handling

String Comparison

```
class EqualsNotEqualTo {
  public static void main(String args[]) {
      String str1 = "Hello";
      String str2 = new String(str1);
      System.out.println(str1 + " equals " + str2 + " -> " +
              str1.equals(str2));
      System.out.println(str1 + " == " + str2 + " -> " + (str1 ==str2));
  }
}
```

Output : Hello equals Hello -> true
Hello == Hello -> false

IGATE Speed.Agility.Imagination

The String class includes various methods that compare strings or substrings within each string. The most popularly used two ways to compare the strings is either using = = operator or by using the equals method.
The equals() method compares the characters inside a String object. The = = operator compare two object references to see whether they refer to the same instance. The program above shows the difference between the two.

5.6: String Handling

StringBuffer Class

- **Following classes allow modifications to strings:**
  - java.lang.StringBuffer
  - java.lang.StringBuilder
- **Many string object manipulations end up with a many abandoned string objects in the String pool, since String objects are immutable**

```
StringBuffer sb = new StringBuffer("abc");
sb.append("def");
System.out.println("sb = " + sb); // output is "sb = abcdef"
```

June 11, 2015 | Proprietary and Confidential | - 22 -

IGATE Speed.Agility.Imagination

Let us understand StringBuilder with an example.

```
String x = "abc";
x.concat("def");
System.out.println("x = " + x); // output is "x = abc"
```

Because no new assignment was made, the new String object created with the concat() method was abandoned instantly. We also saw examples like this:

```
String x = "abc";
x = x.concat("def");
System.out.println("x = " + x); // output is "x = abcdef"
```

We got a nice new String out of the deal, but the downside is that the old String "abc" has been lost in the String pool, thus wasting memory. If we were using a StringBuffer instead of a String, the code would look like this:

```
StringBuffer sb = new StringBuffer("abc");
        sb.append("def");
      System.out.println("sb = " + sb); // output is "sb = abcdef“
```

Note: Refer Javadocs to know more about other methods of StringBuilder.

5.6: String Handling

StringBuilder Class

- **Added in Java 5**
- **Exactly the same API as the *StringBuffer* class, except:**
  - It is not thread safe
  - It runs faster than StringBuffer

```
StringBuilder sb = new StringBuilder("abc");
sb.append("def").reverse().insert(3, "---");
System.out.println( sb ); // output is "fed---cba"
```

June 11, 2015 | Proprietary and Confidential | - 23 -

IGATE Speed.Agility.Imagination

The StringBuilder class was added in Java 5. It has exactly the same API as the StringBuffer class, except StringBuilder is not thread safe. In other words, its methods are not synchronized. Sun recommends that you use StringBuilder instead of StringBuffer whenever possible because StringBuilder will run faster (and perhaps jump higher). So, apart from Synchronization, anything we say about StringBuilder's methods holds true for StringBuffer's methods, and vice versa.

Note: Refer Javadocs to know more about methods of StringBuilder.

5.6: String Handling

Demo

- **Execute the following programs:**
  - SimpleString.java
  - ToStringDemo.java
  - StringBufferDemo.java
  - CharDemo.java

IGATE
Speed.Agility.Imagination

5.7: Date and Time API

Date and Time API

- **Added in Java SE 8 under java.time package.**
- **Enhanced API to make extremely easy to work with Date and Time.**
- **Immutable API to store date and time separately.**
  - Instant
  - LocalDate
  - LocalTime
  - LocalDateTime
  - ZonedDateTime
- **It has also added classes to measure date and time amount.**
  - Duration
  - Period
- **Improved way to represent units like day and months.**
- **Generalised parsing and formatting across all classes.**

June 11, 2015 | Proprietary and Confidential | - 25 -

IGATE Speed.Agility.Imagination

A long-standing bugbear of Java developers has been the inadequate support for the date and time. In order to address problems in legacy API (Date and Calender) and provide better support in the JDK core, a new date and time API (JSR 310) has been designed for Java SE 8 under java.time package.

LocalDate and LocalTime represents date and time respectively. The combination of date and time is represented by LocalDateTime. If a time zone is important, ZonedDateTime class is handy.

Many times developers need to measure amount of time between to date instants. Duration class used to measure amount of time including nanosecond precision. Period class is used to measure in terms of days, months or years.

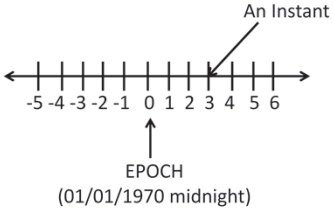
This API has included two Enums DayOfWeek and Month to represent day and month constant respectively.

DateTimeFormatter class is used to format the date and times. Also almost all classes now include parse and format method to support parsing and formatting of date and time.

5.7: Date and Time API

The Instant Class

- **An object of instant represent point on the time line.**
- **The reference point is the standard java epoch.**
- **This class is useful to represent machine timestamp.**



- **The static method "now" of Instant class is used to represent current time.**

```
Instant currentTime = Instant.now();
```


Instant class is useful for generating a time stamp to represent machine time. A value returned from the Instant class counts time beginning from the first second of January 1, 1970. This value is known as EPOCH.

An epoch is an instant on a timeline that is used as a reference point (or the origin) to measure other instants. As shown in the above figure, an Instant at epoch is represent by zero. Instants after epoch are positive numbers where as instants before epoch are negative.

5.7: Date and Time API

The LocalDate Class

- **It represent date without time and zone.**
- **Useful to represent date events like birthdate .**
- **Following table shows important methods of LocalDate:**

Method	Uses
now	A static method to return today's date.
of	Creates local date from year, month and date.
getXXX ()	Used to return various part of date.
plusXXX()	Add the specified factor and return a LocalDate.
minusXXX()	Subtracts the specified factor and return a LocalDate.
isXXX()	Performs checks on LocalDate and returns Boolean value.
withXXX()	Returns a copy of LocalDate with the factor set to the given value.

IGATE Speed.Agility.Imagination

A LocalDate represents a year-month-day in the ISO calendar and is useful for representing a date without a time.

```
LocalDate now = LocalDate.now();
LocalDate independence = LocalDate.of(1947, Month.AUGUST, 15);
System.out.println("Independence:"+ independence);
System.out.println("Today:"+now);
System.out.println("Tomorrow:"+ now.plusDays(1));
System.out.println("Last Month:"+ now.minusMonths(1));
System.out.println("Is leap?:"+ now.isLeapYear());
System.out.println("Move to 30th day of month:"+ now.withDayOfMonth(30));
```

The other two classes LocalTime and LocalDateTime as name reflects are used to store time and date with time respectively.

Most of the methods of LocalDateTime are analogous to LocalDate with few additional methods like plusHours(), plusMinutes() etc.

5.7: Date and Time API

The ZonedDateTime Class

- **It stores all date and time fields, to a precision of nanoseconds, as well as a time-zone and zone offset.**
- **Useful to represent arrival and departure time in airline applications.**
- **Following table shows important methods of ZonedDateTime:**

Method	Uses
now	A static method to return today's date.
of	Overloaded static method to create zoned date time object.
getXXX ()	Used to return various part of ZonedDateTime.
plusXXX()	Add the specified factor and return a ZonedDateTime.
minusXXX()	Subtracts the specified factor and return a ZonedDateTime.
isXXX()	Performs checks on ZonedDateTime and returns Boolean value.
withXXX()	Returns ZonedDateTime with the factor set to the given value.

IGATE Speed.Agility.Imagination

A ZonedDateTime represents a point in time in a given time zone that can be converted to an instant on the timeline; it is aware of Daylight Saving Time.

```
ZonedDateTime currentTime = ZonedDateTime.now();
ZonedDateTime currentTimeInParis = ZonedDateTime.now(ZoneId.of("Europe/Paris"));
System.out.println("India:" + currentTime);
System.out.println("Paris:" + currentTimeInParis);
```

5.7: Date and Time API

Period and Duration

- **The Period class models a date-based amount of time, such as five days, a week or three years.**
- **Duration class models a quantity or amount of time in terms of seconds and nanoseconds. It is used represent amount of time between two instants.**
- **Following table shows important and common methods of both:**

Method	Uses
between	Use to create either Period or Duration between LocalDates.
of	Creates Period/Duration based on given year, months & days.
ofXXX()	Creates Period/Duration based on specified factors.
getXXX ()	Used to return various part of Period/Duration.
plusXXX()	Add the specified factor and return a LocalDate.
minusXXX()	Subtracts the specified factor and return a LocalDate.
isXXX()	Performs checks on LocalDate and returns Boolean value.
withXXX()	Returns a copy of LocalDate with the factor set to the given value.

June 11, 2015 | Proprietary and Confidential | - 29 -

IGATE Speed.Agility.Imagination

The following example shows how to find period between two dates.

```
LocalDate start = LocalDate.of(1947, Month.AUGUST, 15);
LocalDate end = LocalDate.now(); //18/02/2015
Period period = start.until(end);

System.out.println("Days:"+ period.get(ChronoUnit.DAYS));
System.out.println("Months:"+period.get(ChronoUnit.MONTHS));
System.out.println("Years:"+ period.get(ChronoUnit.YEARS));
```

5.7: Date and Time API

Formatting and Parsing Date and Time

- **Java SE 8 adds DateTimeFormatter class which can be used to format and parse the date and time.**
- **To either format or parse, the first step is to create instance of DateTimeFormatter.**
- **Following are few important methods available on this to create DateTimeFormatter.**

Method	Uses
ofLocalizedDate(dateStyle)	Date style formatter from locale
ofLocalizedTime(timeStyle)	Time style formatter from locale
ofLocalizedDateTime(dateTimeStyle)	Date and time style formatter from locale
ofPattern(StringPattern)	Custom style formatter from string

- **Once formatter object created, parsing/formatting is done by using parse() and format() methods respectively. These methods are available on all major date and time classes .**

June 11, 2015 | Proprietary and Confidential | - 30 - IGATE Speed.Agility.Imagination

To format or parse a date/time, first we need to create instance of DateTimeFormatter. Following example shows, how to format a date using this class. The below example shows how to format the LocalDate in medium style.

```
DateTimeFormatter formatter = DateTimeFormatter.ofLocalizedDate(FormatStyle.MEDIUM);
LocalDate currentDate = LocalDate.now();
System.out.println(currentDate.format(formatter));
```

The following example shows how to parse a text string into date.

```
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd/MM/yyyy");
String text = "12/02/2015";
LocalDate date = LocalDate.parse(text, formatter);
System.out.println(date);
```

5.7: Date and Time API

Demo

- **Execute the following programs:**
  - LocalDateDemo.java
  - ZonedDateTimeDemo.java
  - CalculatingPeriod.java
  - FormattingDate.java
  - ParsingDate.java

IGATE
Speed.Agility.Imagination

5.8: Best Practices

Best Practices - String Handling

- Use *StringBuffer* for appending
- *String.charAt()* is slow
- Use *String.intern* method to improve performance
- Use isEmpty() method to check empty string in faster way

June 11, 2015 | Proprietary and Confidential | - 32 -

IGATE Speed.Agility.Imagination

Use StringBuffer for appending
Always use StringBuffer for appending. Appending by StringBuffer is almost 200 times faster than by using String.concat and "+" operator. But if String constants are to be appended "+" is faster than StringBuffer.append because it is resolved in compile time. Any code with constants is faster.

String.charAt() is slow
The method charAt(int index) returns an individual char from inside a string (see also the substring() method below). The valid index numbers are in the range 0..length-1. Using an index number outside of that range will raise a runtime exception and stop the program at that point.

```
String string = "hello"; char a = string.charAt(0); // a is 'h'
char b = string.charAt(4); // b is 'o‘
char c = string.charAt(string.length() - 1); // same as above line
char d = string.charAt(99); // ERROR, index out of bounds
```

charAt() is slow because it does check for bounds before finding the character. Secondly, it is not declared as final which actually make a bit slower. One can use indexOf and a loop. Which is at least 3 times faster than charAt().

String indexOf()
The indexOf() method searches inside the receiver string for a "target" string. The indexOf() method returns the index number where the target string is first found (searching left to right), or -1 if the target is not found.

int indexOf(String target) -- searches for the target string in the receiver. Returns the index where the target is found, or -1 if not found.

The indexOf() search is case-sensitive -- upper and lowercase letters must match exactly.

```
String string = "Here there everywhere";
int a = string.indexOf("there"); // a is 5
int b = string.indexOf("er"); // b is 1
int c = string.indexOf("eR"); // c is -1, "eR" is not found
```

String.intern method can be used to improve performance
Consider the following example:

```
for(int i=0;i<variables.length;i++){
variables[i] = new String("hello");
}

here since strings are immutable. For every assignment a separate
object is created. But if used in the following way
int len = variables.length()
for(int i=0;i<len;i++){
variables[i] = new String("hello");
variables[i] = variables[i].intern();
}
```

The problem of immutability could be solved to some extent. The calling of intern method ensures that when the value of that particular string is changed, then instead of creating new string, the reference of old string is obtained and value of that is changed. It gives considerable performance improvement. It is nothing but creating a pool of unique String objects.

5.8: Best Practices

Common Best Practices (contd..)

- **Assert is for private arguments only**
- **Validate method arguments**
- **Fields should usually be private**
- **Instance variable should not be used directly in a method**
- **Do not use *valueOf* to convert to primitive type**
- **Downward cast is costly**

June 11, 2015 | Proprietary and Confidential | - 34 -

IGATE Speed.Agility.Imagination

Assert is for private arguments only

Most validity checks in a program are checks on parameters passed to non-private methods. The assert keyword is not meant for these types of validations.

Assertions can be disabled. Since checks on parameters to non-private methods implement the requirements demanded of the caller, turning off such checks at runtime would mean that part of the contract is no longer being enforced.

Conversely, checks on arguments to private methods can indeed use assert. These checks are made to verify assumptions about internal implementation details, and not to check that the caller has followed the requirements of a non-private method's contract.

Validate method arguments

The first lines of a method are usually devoted to checking the validity of method arguments. The idea is to fail as quickly as possible in the event of an error. This is particularly important for constructors. It is a reasonable policy for a class to skip validating arguments of private methods. The reason is that private methods can only be called from the class itself. Thus, a class author should be able to confirm that all calls of a private method are valid. If desired, the assert keyword can be used to check private method arguments, to check the internal consistency of the class.

Fields should usually be private:
Fields should be declared private unless there is a good reason for not doing so.
One of the guiding principles of lasting value in programming is "Minimize ripple effects by keeping secrets." When a field is private, the caller cannot usually get inappropriate direct access to the field.

Accessing Instance variables in a method:

```
private char buf[] = new char[4096];
public int find(char c)
{ for (int i = 0; i < buf.length; i++) {
if (buf[i] == c)
return i; }
return -1;}
```

The above code is optimized drastically if the instance variable buf is copied to a local variable in the method and used in the loop. This is because the JVM invokes getField method internally to access the instance variable which is a overhead. Another optimization is never use ".length" of array in the for loop. Instead get the length in a local variable and use that in for loop condition, because JVM calls arrayLength method internally for every iteration

ValueOf (double) to convert to primitive type
Avoid using this method. It creates a unnecessary Double object inside this method. Instead use parseDouble. Follow the same for other wrapper classes also
Downward cast is costly
Downward cast from parent class to its subclass/es are always costly, since it has to check the validity of both the classes and if not valid it has throw an exception "ClassCastException" which is time consuming. Instead first check the validity of subclass using **instanceOf** and then cast it. It removes the chance of throwing exception

5.9: Exploring Java Basics

Lab

- **Lab 3: Exploring Basic Java Class Libraries**

IGATE
Speed.Agility.Imagination

Summary

- **In this lesson you have learnt:**
  - The Object Class
  - Wrapper Classes
  - Type casting
  - Using Scanner Class
  - The System Class
  - String Handling
  - Date and Time API
  - Best Practices

Add the notes here.

Review Questions

- **Question 1: String objects are mutable and thus suitable to use if you need to append or insert characters into them.**
  - True/False
- **Question 2: Which of the following static fields on wrapper class indicates range of values for its class:**
  - Option 1:MIN_VALUE
  - Option 2: MAX_VALUE
  - Option 3: SMALL_VALUE
  - Option 4: LARGE_VALUE

Core Java 8 with JAXB and Development Tools

Inheritance and Polymorphism

IGATE
Speed.Agility.Imagination

Lesson Objectives

- **After completing this lesson, participants will be able to -**
  - Understand concept of Inheritance and Polymorphism
  - Implement inheritance in java programs
  - Implement different types of polymorphism

Outline:

Lesson 6: Inheritance and Polymorphism

- 6.1: Inheritance
- 6.2: Using super keyword
- 6.3: InstanceOf Operator
- 6.4: Method & Constructor overloading
- 6.5: Method overriding
- 6.6: @override annotation
- 6.7: Using final keyword
- 6.8: Best Practices

6.1: Inheritance

What is Inheritance?

Basic TV

Smart TV

Smart TV's are inherited from basic television which apart from multimedia functionality of TV allows us to do more like streaming video contents from Internet.

June 11, 2015 | Proprietary and Confidential | - 3 -

IGATE Speed.Agility.Imagination

<u>What is Inheritance</u>?

If you look at the slide example of how televisions are evolved in last decade, you will find two major differences between these two models. Basic TV's are used to watch programs typically they are streamed from set top box. On the other hand, the most advanced "smart TVs" are similar in function to "smart phones" and some tablets. These TVs go beyond providing access to web-based media, or streaming from content stored on your home computer. Smart TVs have the built in computing power that allows you to do many of the same things you can do with a smart phone or tablet such as web browsing, use of web-based services like Skype, and interactive access to social media sites.

Apart from the functional **enhancement**, there is slight **alteration** from hardware point of view. The position of sound boxes is altered and in smart TV's like to hide them from front view.

Smart TV's are **"inherited"** from Basic TV for two reasons:

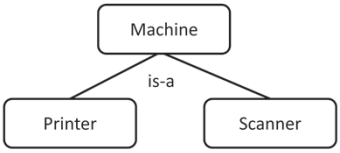
1. To make enhancement and/or
2. To do alteration

This is basis of inheritance in OOPs where one class we can extend to either enhance its functionality or alter its behavior.

6.1: Inheritance

What is Inheritance?

- **Inheritance allows programmers to reuse of existing classes and make them extendible either for enhancement or alteration**
- **Allows creation of hierarchical classification**
- **Advantage is reusability of the code:**
  - A class, once defined and debugged, can be used to create further derived classes
- **Extend existing code to adapt to different situations**
- **Inheritance is ideal for those classes which has "is-a" relationship**
- **"Object" class is the ultimate superclass in Java**

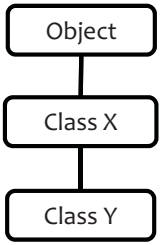


June 11, 2015 | Proprietary and Confidential | - 4 -

IGATE Speed.Agility.Imagination

It is one of the fundamental mechanisms for code reuse in object-oriented programming. Inheritance allows new classes to be derived from existing classes. In Java, inheritance specifies how different is the sub class from its parent class. Thus, we can add new variables, methods and also modify the inherited methods. To inherit a class, you simply **extend** the super class into the subclass. Only single level inheritance is possible in Java.

Superclass and Subclass:
Any class **preceding** a specific class in the class hierarchy is said to be super class. On the other hand Any class **following** specific class in the hierarchy is called as subclass. All classes in Java are by default extensible. All All Java classes that do not explicitly extend a parent class automatically extend the **java.lang.Object** class.



In the above example, class Y is subclass. Class X is superclass of Y but subclass of Object class. **Object** class is the ultimate superclass in Java.

6.2: Using super keyword

Using super Keyword

- **The super keyword is used to refer instance of its direct superclass**
- **There are two uses of super keyword:**
  - Calling parent class constructor
  - Call member of parent class

June 11, 2015 | Proprietary and Confidential | - 5 -

IGATE Speed.Agility.Imagination

When you create instance of child class by calling its one of the constructor, it invokes immediate parent class default constructor which in turn calls its parent-parent class constructor. This process continues until constructor of Object class is called.
This constructor call chain will break if there is no default constructor available on parent class. To avoid this, you can call non-default constructor of parent class by using super keyword.

Kindly note when you call parent class constructor using super keyword, it must be written as first line of child constructor.

The other use of super keyword is to call to access the members of parent class. The member can be accessed by using syntax super.memberName or super.memeberMethod().

6.2: Using super keyword

Inheritance : Example

```
class Base {
  public void baseMethod() {
      System.out.println("Base");
  }
}
class Derived extends Base {
    public void derivedMethod() {
      super. baseMethod ();
      System.out.println("Derived");
    }
}
class Test {
    public static void main(String args[]){
            Derived derived=new Derived();
            derived. derivedMethod();
    }
}
```

Output:
Base
Derived

IGATE
Speed.Agility.Imagination

6.3: InstanceOf Operator

instanceOf Operator

- **The instanceOf operator compares an object to a specified type**
- **Checks whether an object is:**
  - An instance of a class.
  - An instance of a subclass.
  - An instance of a class that implements a particular interface.
  - Example : The following returns true:

```
new String("Hello") instanceof String;
```

June 11, 2015 | Proprietary and Confidential | - 7 -

IGATE Speed.Agility.Imagination

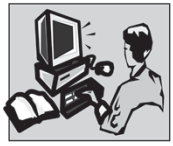
The instanceof operator is used to make a test whether the given object belongs to specified type. Consider the below example. The if statement returns true here as the child object is type of its superclass.

```
class Ticket{
}
class ConfirmedTicket extends Ticket {
}
…
…
ConfirmedTicket tkt= new ConfirmedTicket();
If(tkt instanceof Ticket) {
   //some processing
}
```

6.3: InstanceOf Operator

Demo

- **Inheritance**
  - Basic Inheritance
  - Using super Keyword
  - Use of instanceof keyword

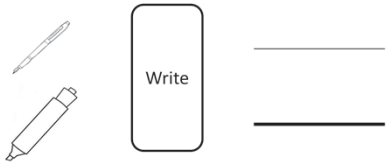


IGATE
Speed.Agility.Imagination

6.3: Polymorphism

What is Polymorphism?

- **Poly meaning "many" and morph means "forms"**
- **It's capability of method to do different things based on the object used for invoking method**



- **Polymorphism also enables an object to determine which method implementation to invoke upon receiving a method call**
- **Java implements polymorphism in two ways**
  - Method Overloading
  - Method Overriding

June 11, 2015 | Proprietary and Confidential | - 9 -

IGATE Speed.Agility.Imagination

There are two ways in which polymorphism is implemented in Java.

1. Method Overloading
2. Method Overriding

Method Overloading is compile time polymorphism where the same method name has different meanings. Method overriding on other hand, is kind of runtime polymorphism where a subclass defines method with the same signature as defined by its superclass.

6.3: Polymorphism

Method Overloading

- **Two or more methods within the same class share the *same* name. Parameter declarations are different**
- **You can overload Constructors** and **Normal Methods**

```
class Box {
   Box(){
      //1. default no-argument constructor
   }
   Box(dbl dblValue){
      // 2. constructor with 1 arg
   }
  public static void main(String[] args){
Box  boxObj1 = new Box(); // calls constructor 1
Box  boxObj2 = new Box(30);  // calls constructor 2
      } }
```

June 11, 2015 | Proprietary and Confidential | - 10 -

IGATE Speed.Agility.Imagination

In Java it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different. Here, the methods are said to be overloaded. When an overloaded method is invoked, Java determines which method to actually call by using the type and/or number of operators as its guide.

Refer the example above. Here, two constructors have been declared; one with no arguments, the second with a single argument
Note: In addition to overloading constructors, you can also overload normal methods.

6.3: Polymorphism

Constructor Overloading

- **If class has more than one constructors, then they are called as overloaded constructors.**
- **When constructors are overloaded, they must differ in**
  - Number of parameters
  - Type of parameters
  - Order of parameters

A same model car can be constructed in different ways as per the requirement. Few of them are basic, while the other differ in fuel type, color, engine power, CNG, A.C. and or other accessories.

June 11, 2015 | Proprietary and Confidential | - 11 -

IGATE Speed.Agility.Imagination

Example:

```
class Car {
 int noOfCylinders;
 int noOfValves;
 int enginePower;
 boolean isPowerSteering;
 Car(){
  //1. default no-argument constructor
     noOfCylinders = 3;
     noOfValves = 4;
     enginePower = 48;                //48 ps
     isPowerSteering = false;
 }
 Car(boolean isPowerSteering){
  // 2. constructor with 1 arg
     this();
     this.isPowerSteering = isPowerSteering;
 }
 Car(int enginePower,int noOfCylinders, int noOfValves){
  // 3. constructor with // 3 args
     this.noOfCylinders = noOfCylinders;
     this.noOfValves = noOfValves ;
     this.enginePower = enginePower;
     this.isPowerSteering = true;
 }
}
```

6.3: Polymorphism

Method Overriding

- **In a class hierarchy, when a method in a subclass has the same *name* and *type signature* as a method in its super class, then the subclass method overrides the super class method**
- **Overridden methods allow Java to support run-time polymorphism**

Normal Swap Machine

Chip card Machine which **overrides** the card reading for better security

June 11, 2015 | Proprietary and Confidential | - 12 -

IGATE Speed.Agility.Imagination

Example:

```
class SwipeMachine{
   void readCard (){
         // functionality to read normal cards
    }
}
class ChipCardMachine extends SwipeMachine{
   void readCard(){
         //functionality to read chip card
    }
}
public static void main(String args[]){
     SwipeMachine normal = new SwipeMachine();
     normal.readCard();          //reading normal swipe card
     normal = new ChipCardMachine();
     normal.readCard();          //reading chip based swipe card
} }
```

6.4: Override Annotation

@Override Annotation

- **The @Override annotation informs the compiler that the element is meant to override an element declared in a superclass**
- **It applies to only methods**

```
public class Employee {
    @override
    public String tostring()  {
      //statements
      return "EmpName is:"+empname;
    }
}
```

- **Above code will throw a compilation error as it is not overriding the toString method of Object Class**

June 11, 2015 | Proprietary and Confidential | - 13 -

IGATE Speed.Agility.Imagination

<u>@Override:</u>
The Compiler checks that a method with this annotation really overrides a method from the Super class or not

While it's not required to use this annotation while overriding a method, it helps to prevent errors. If a method marked with **@Override** fails to correctly override a method in one of its superclasses, the compiler generates an error.

Most commonly, it is useful when a method in the base class is changed to have a different parameter list. A method in a subclass that used to override the superclass method no longer does so due to the changed method signature. This can sometimes cause strange and unexpected behavior, especially while dealing with complex inheritance structures. The **@Override** annotation safeguards against this.
@Override is useful in detecting changes in parent classes which has not been reported down the hierarchy. Without it, a method signature can be changed and altering its overrides can be forgotten. With **@Override**, the compiler catches it for you.

Other annotations supported in Java SE:

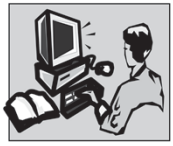
The **@SuppressWarnings** annotation instructs the compiler to suppress the warning messages it normally shows during compilation time.

@Deprecated marks an old method as deprecated. Which says this method must not be used anymore because in the future versions, this old method may not be supported.

6.5: Polymorphism

Demo

- **Polymorphism**
  - Method Overloading
  - Method Overriding



IGATE
Speed.Agility.Imagination

6.6: Using Final Modifier

Final Modifier

- **Final Modifier : Can be applied to variables, methods and classes**
- **Final variable:**
  - Behaves like a constant; i.e. once initialized, it's value cannot be changed
  - Example: final int i = 10;
- **Final Method:**
  - Method declared as final cannot be overridden in subclasses
  - Their values cannot change their value once initialized
  - Example:

```
class A {
  public final int add (int a, int b) { return a+b; }
}
```

A class or method cannot be abstract & final at the same time.

- **Final class:**
  - Cannot be sub-classed at all
  - Examples: *String* and *StringBuffer* class

IGATE Speed.Agility.Imagination

6.7: Inheritance and Polymorphism

Lab

- **Lab 4: Inheritance and Polymorphism**

IGATE
Speed.Agility.Imagination

Summary

- **In this lesson, you have learnt about:**
  - Inheritance
  - Using super keyword
  - InstanceOf Operator
  - Method & Constructor overloading
  - Method overriding
  - @override annotation
  - Using final keyword
  - Best Practices

Review Question

- **Question 1: Which of the following options enable parent class to avoid overriding of its methods.**
  - extends
  - Override
  - Final
- **Question 2: When you want to invoke parent class method from child, it should be written as first statement in child class method**
  - True/False
- **Question 3: Which of the following access specifier enables child class residing in different package to access parent class methods?**
  - private
  - public
  - Final
  - Protected

Core Java 8 with JAXB and Development Tools

Abstract Classes and Interfaces

IGATE
Speed.Agility.Imagination

Lesson Objectives

- **After completing this lesson, participants will be able to:**
  - Understand concept of Abstract classes and Interfaces
  - Differentiate between abstract classes and interfaces
  - Implement Runtime polymorphism

Lesson Outline:

7.1: Abstract class
7.2: Interfaces
7.3: default methods
7.4: static methods on Interface
7.5: Runtime Polymorphism
7.6: Best Practices

7.1: Abstract Classes

Abstract Class

- **Provides common behavior across a set of subclasses**
- **Not designed to have instances that work**
- **One or more methods are declared but may not be defined, these methods are abstract methods.**
- **Abstract method do not have implementation**
- **Advantages:**
  - Code reusability
  - Help at places where implementation is not available

June 11, 2015 | Proprietary and Confidential | - 3 -

IGATE Speed.Agility.Imagination

Example of Abstract class – Shape Hierarchy –
You have two hierarchies : Point-Circle-Cylinder and Line-Square-Cube
Here common method is area() – if I create an array that contains the objects of all these classes. How can I do it generically ?
Can Point p = new Line() be valid ? of course not, no inheritance !!!!
So, I force a super class Shape which would contain the method area(). Now what implementation I am going to write in that method. I do not know, at runtime whose area() is going to be called. So, the information that I do not know I put it as "abstract".

Important points about abstract class :

- You cannot create object of abstract class : why ?
  For example, if a surgeon know how to perform an operation but he does not know how to stitch back the cut stomach will I allow him to touch me?
- An abstract class can contain concrete methods.
- An abstract class may not contain any abstract methods.
- In Java a pure virtual method is called as abstract method.

```
abstract class Shape{
    public abstract float area();
}
```

Abstract Class (cont..)

- **Declare any class with even one method as abstract as *abstract***
- **Cannot be instantiated**
- **Cannot use *Abstract* modifier for:**
  - Constructors
  - Static methods
- **Abstract class' subclasses should implement all methods or declare themselves as *abstract***
- **Can have concrete methods also**

IGATE Speed.Agility.Imagination

Example of Abstract modifier:

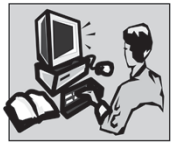
```
abstract class Shape{
    abstract void draw();       // observe : no implementation
}

class Rect extends Shape{
    void draw(){     // draw() implemented in subclass Rect
        System.out.println("Drawing a rectangle");
    }
    public static void main(String args[]){
        Shape r1 = new Rect();
        r1.draw();
    }
}
```

```
Output :
Drawing a rectangle
```

7.1: Abstract Classes

Demo

- **Execute the Executor.java program**



IGATE
Speed.Agility.Imagination

The example shows how to declare and use an interface

7.2: Interfaces

Interface

- Special kind of class which consist of only the constants and the method signatures.
- Approach also known as "programming by contract".
- It's essentially a collection of constants and abstract methods.
- It is used via the keyword "implements". Thus, a class can be declared as follows:

```
class MyClass implements MyInterface{
  ...
}
```

June 11, 2015 | Proprietary and Confidential | - 6 -

IGATE Speed.Agility.Imagination

Interface:

You may come across a situation, in which you want to have different implementations of a method in different classes and delay the decision on which implementation of a method to execute until runtime. In java, the class where the method is defined must be present at compile time so that the compiler can check the signature of the method to ensure that the method call is legitimate. All the classes that could possibly be called for the aforementioned method need to share a common super class, so that the method can be defined in the super class and overridden by the individual subclasses. If you want to force every subclass to have its own implementation of the method, the method can be defined as an abstract one. Chances are you will want to move the method definition higher and higher up the inheritance hierarchy, so that more and more classes can override the same method.

Because of single inheritance, any Java class has only a single super class. It inherits variables and methods from all superclasses above it in the hierarchy. This makes sub-classing easier to implement and design, but it also can be restricting when you have similar behavior that must be duplicated across different branches of class hierarchy. Java solves the problem of shared behavior by using interfaces. An interface is a collection of method signatures (without implementations) and constant values. Interfaces are used to define a protocol behavior that can be implemented by any class hierarchy. Interfaces are abstract classes that are left completely unimplemented**.** That means, no methods in the class has been implemented. Using an interface, you can specify what a class must do, but not how it does.

What is Interface?

- **A Java interface definition looks like a class definition that has only abstract methods, although the abstract keyword need not appear in the definition**

```
public interface Testable {
void method1();
void method2(int i, String s);
int x=10;
}
```

note no implementation for the methods, public by default

Static final variable

June 11, 2015 | Proprietary and Confidential | - 7 -

IGATE Speed.Agility.Imagination

Interface (contd.):

Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body. Additionally, interface data members are limited to **static final variables**, which means that they are constant. An object variable can be declared as an interface type, and all the constants and methods declared in the interface can be accessed from this variable. All objects, whose class types implement the interface, can then be assigned to this variable. Therefore, to solve the problem of how to decide implementation which method is to be executed at runtime, you can define an interface with the method be shared among classes. Reference to this commonly implemented method from the interface object variable will then be resolved at runtime.

An interface definition consists of both interface declaration and interface body.

```
// Interface Declaration:
<access> interface <name>  {
        return type <method_name> ( <parameter_list> ) ;
        <type> <variable name> = <value>;
}
```

The interface declaration informs about various attributes of the interface such as its name and whether it extends to another interface.

Declaring and Using Interfaces

```
public interface SimpleCalc {
        int add(int a, int b);
        int i = 10;
}
//Interfaces are to be implemented.
class Calc implements SimpleCalc {
        int add(int a, int b){
                return a + b;
        }
}
```

abstract method

By default is public, static and final

June 11, 2015 | Proprietary and Confidential | - 8 -

IGATE Speed.Agility.Imagination

An interface declaration can have two other components:
- The public access specifier
- The list of super interfaces

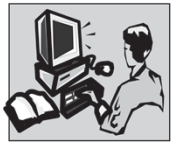
While a class can only extend one other class, an interface can extend any number of interfaces, and an interface cannot extend classes. An interface inherits all constants and methods from its super interface.
The interface body contains method declarations for the methods defined within the interface and constant declarations. All constant values defined in an interface are implicitly public, static and final. Similarly, all methods declared in an interface are implicitly public and abstract. One class can implement more than one interface at a time by separating them using commas.

Note: Once, a class implements an interface it has to override all the methods in that interface; otherwise, a class has to be declared as an abstract class.

7.2: Interfaces

Demo

- **Execute the Interface Implementation.java program**



IGATE
Speed.Agility.Imagination

The example shows how to declare and use an interface

7.3: Interfaces

Default Methods

- **Starting from Java SE 8, interfaces can define default methods**
- **A default method in an interface is a method with implementation**
- **Use "default " keyword in method signature to make it default.**

```
interface xyz {
  default return-type method-name(argument-list) {
      ------------------
      ------------------
  }
}
```

- **A class which implements the interface doesn't need to implement default methods**

June 11, 2015 | Proprietary and Confidential | - 10 -

IGATE Speed.Agility.Imagination

To enhance Java API with lambda expressions, many existing interfaces needs to be modified. Adding new methods to interface leads to break the existing implementation.

To avoid this, Java has added default methods to interfaces. When you add default method to existing interface, it doesn't break the classes which implements the interface.

Note: Default interfaces concept is strictly meant for backward compatibility. It doesn't mean you create java application only with interfaces with default methods and not classes.

Rules for default methods in Interface inheritance:

While extending an Interface having default methods, there are few points to ponder:

1. Child Interface can use the default method of parent interface.
2. Child Interface can re-declare the default method without default keyword to make it abstract.
3. Child Interface can override the default method by keeping the same signature as of parent interface.

7.4: Interfaces

Static Methods

- **Along with the default methods an Interface can also have static methods**
- **The syntax of static method is similar to default method, where static keyword will replace default**

```
interface xyz {
  static return-type method-name(argument-list) {
      ------------------
      ------------------
  }
}
```

June 11, 2015 | Proprietary and Confidential | - 11 -

IGATE Speed.Agility.Imagination

Both default and static interface method now allows developer to extend the functionality of existing system without breaking the code.

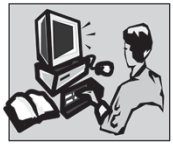
If you are designing the system from scratch, it is recommended not to use these features. However, these features can be handy to modify existing application to add new functionality with ease.

The default and static methods are extensively used by Java SE 8 language developers to add new methods to existing API. For example, the **forEach()** default method added to Collection API for iterating elements in collection.

7.4: Interfaces

Demo

- **Interface with default and static methods**



IGATE
Speed.Agility.Imagination

The example shows how to declare and use an interface with default and static methods.

7.5: Interfaces

Interface - Rules

- **Methods other than default and static in an interface are always public and abstract.**
- **Static methods in interface are always public .**
- **Data members in a interface are always public, static and final.**
- **Interfaces can extend other interfaces.**
- **A class can inherit from a single base class, but can implement multiple interfaces.**

IGATE
Speed.Agility.Imagination

Add the notes here.

7.6: Abstract Class vs Interface

Abstract Classes and Interfaces

Abstract classes	Interfaces
Abstract classes are used only when there is a "is-a" type of relationship between the classes.	Interfaces can be implemented by classes that are not related to one another.
You cannot extend more than one abstract class.	You can extend more than one interface.
Abstract class can contain abstract as well as implemented methods.	Interfaces contain only abstract, default and static methods.
With abstract classes, you grab away each class's individuality.	With Interfaces, you merely extend each class's functionality.

7.7: Runtime Polymorphism

Runtime Polymorphism

- **Runtime polymorphism enables a method can do different things based on the object used for invoking method at runtime**
- **Runtime polymorphism is implemented by doing method overriding**

```
class Parent {
    public String sayHello() {
        return "Hello from Parent";
    }
}
class Child extends Parent {
    public String sayHello() {
        return "Hello from Child";
    }
}
```

```
Parent object = new Child();
object.sayHello();
```

Hello from Child

June 11, 2015 | Proprietary and Confidential | - 15 -

IGATE Speed.Agility.Imagination

In Java, it is legal to up cast the reference of child object to parent class reference. Also in case of interfaces, you can assign reference of implementer class to an interface object.

Parent Class reference = Child Class Object;
Interface reference = Interface Implementer class object;

In above cases, parent class reference can be used to call methods in child class which are overridden. It cannot be used to access methods owned by child class.

The code snippet shown in the slide has two classes, where child class overrides the sayHello() method of parent. Therefore parent class reference is able to access the overridden method of child class.

Note: Polymorphism does not work with static methods because they are early-bound.

7.7: Runtime Polymorphism

Accessing Implementations through Interface Reference

```
class sample implements TestInterface {
// Implement Callback's interface
public void interfacemthod() {
        System.out.println("From interface method"); }
public void noninterfacemthod() {
        System.out.println("From interface method"); }
 }
```

```
class Test {
  public static void main(String args[]) {
      TestInterface t = new sample();
       t.interfacemethod()     //valid
       t.noninterfacemethod()      //invalid }
```

June 11, 2015 | Proprietary and Confidential | - 16 -

IGATE Speed.Agility.Imagination

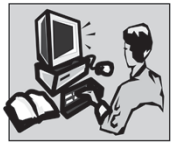
Accessing Implementations through Interface Reference:

Object references can be declared which uses an interface rather than a classtype. Any instance of any class that implements the declared interface can be referred to by such a variable. When you call a method through one of these references, the correct version will be called based on the actual instance of the interface being referred to. This is one of the key features of interfaces. The method to be executed is looked up dynamically at run time, allowing classes to be created later than the code which calls methods on them. The calling code can dispatch through an interface without having to know anything about the "callee."
The variable t is declared to be of the interface type TestInterface, yet it was assigned an instance of sample. Although, t can be used to access the interfacemethod() method, it cannot access any other members of the sample class. An interface reference variable only has knowledge of the methods declared by its interface declaration. Thus, t could not be used to access noninterfacemethod( ) since it is defined by sample but not the TestInterface.

7.7: Runtime Polymorphism

Demo

- **Runtime polymorphism**



IGATE
Speed.Agility.Imagination

The example shows how to declare and use an interface with default and static methods.

7.9: Abstract Classes and Interfaces

Lab

- **Lab 5: Abstract classes and Interfaces**

IGATE
Speed.Agility.Imagination

Add the notes here.

Summary

- **In this lesson, you have learnt about:**
  - Abstract class
  - Interfaces
  - default methods
  - static methods on Interface
  - Runtime Polymorphism

Review Question

- **Question 1: All variables in an interface are :**
  - **Option 1:** Constant instance variables
  - **Option 2:** Static and final
  - **Option 3:** Constant instance variables
- **Question 2: Will this code throw a compilation error?**

```
interface sample
{
 int x;
}
```

  - **Option 1:** True
  - **Option 2:** False

Core Java 8 with JAXB and Development Tools

Regular Expressions

IGATE
Speed.Agility.Imagination

Lesson Objectives

- **After completing this lesson, participants will be able to:**
  - Understand concept of Regular Expressions
  - Use the java.util.regex package
  - Validate input data

Lesson Outline:

8.1: Regular Expressions
8.2: Validating data
8.3: Best Practices

8.1: Regular Expressions

Text Processing using Regular Expression

- **Regular expressions or RegEx is a mechanism of allowing text processing. It is a special text string for performing search, edit, or manipulate text and data.**
- **Regex API is available in the** java.util.regex **package**
- **The String class in java also allows a regular expression operation with minimal code**
  - String.replaceAll()
  - String.matches()
  - String.split()

June 11, 2015 | Proprietary and Confidential | - 3 -

IGATE Speed.Agility.Imagination

Regular Expressions are basically patterns of characters which are used to perform certain useful operations on the given input. The operations include finding particular text, replacing the text with some other text, or validating the given text. For example, we can use Regular Expression to check whether the user input is valid for a field like Email-id or a telephone number.
Supported in most of the common languages like C, Java, Python, C# etc. Support is slightly different for regex in each of these languages though.
The Java String class has several methods that allow you to perform an operation using a regular expression on that string in a minimal amount of code. Some of them are listed above. For example, myString.matches("regex") returns true or false depending whether the string can be matched entirely by the regular expression.

8.1: Regular Expressions

java.util.regex package

- **The java.util.regex package primarily consists of the following three classes:**
  - Pattern
  - Matcher
  - PatternSyntaxException

June 11, 2015 | Proprietary and Confidential | - 4 -

IGATE Speed.Agility.Imagination

The java.util.regex package primarily consists of the following three classes:
Pattern Class: An instance of this class is a compiled representation of a regular expression. This class provides no public constructors. To create a pattern, you must first invoke one of its public static compile methods, which will then return a Pattern object. These methods accept a regular expression as the first argument.
Matcher Class: An instance of this class is the engine that interprets the pattern and performs match operations against an input string. Like the Pattern class, Matcher defines no public constructors. A Matcher instance is created by invoking the matcher() method on a Pattern object.
PatternSyntaxException: A PatternSyntaxException object is an unchecked exception that indicates a syntax error in a regular expression pattern. It has various methods like getDescription(), getIndex(), getMessage() and getPattern() that provide details of the error.

8.1: Regular Expressions

Pattern class

- **java.util.regex.Pattern precompiles regular expressions so they can be executed more efficiently. Example:**
  - String consisting of 'a' in the beginning and 'b' in the end with any number of characters in between
    - Pattern pattern = Pattern.compile("a*b");
  - Number consisting of one or more digits
    - Pattern pattern = Pattern.compile("(\\d+)");
- **Some methods of the Pattern class are compile(), matches(), matcher()**

June 11, 2015 | Proprietary and Confidential | - 5 -

IGATE Speed.Agility.Imagination

Some methods of the Pattern class:

compile (String regex) : This method returns a new Pattern object which is a compiled form of regular expression pattern. Note that compile() is a static method.

matcher (String input) : This method is used to create new Matcher object for an input for a given pattern, which can be used to perform matching operations.

matches (pattern, inputSequence) : This method returns true only if the entire input text matches the pattern. This method internally depends on the compile() and matcher() methods of the Pattern object. Note that matches() is a static method.

8.1: Regular Expressions

Pattern class : Example

```
public class RegExpTest {
   public static void main(String[] args) {
      String inputStr = "Test String";
      String pattern = "Test String";
      boolean patternMatched =
                        Pattern.matches(pattern, inputStr);
      System.out.println(patternMatched);
   }
}
```

Output: true

IGATE Speed.Agility.Imagination

8.1: Regular Expressions

Matcher class

- **java.util.regex.Matcher interprets the pattern and performs match operations against an input string.**
- **It provides a full set of methods to do the scanning.**

```
String input = "Shop,Mop,Hopping,Chopping";
Pattern pattern = Pattern.compile("hop");
Matcher matcher = pattern.matcher(input);
System.out.println(matcher.matches());
while (matcher.find()){
  System.out.println(matcher.group() + ": " +matcher.start() + ": " +
matcher.end());
}
```

Displays : false

Displays:
hop: 1: 4
hop: 18: 21

June 11, 2015 | Proprietary and Confidential | - 7 -

IGATE Speed.Agility.Imagination

Please refer to Javadocs for details about the methods of this class.
In the code snippet above, the first output returns false since the entire input string " Shop,Mop,Hopping,Chopping " does not match the regular expression pattern “hop" and hence matches() method returns false.

8.1: Regular Expressions

Regular Expression guide

Construct	Matches
\d	A digit
\D	A non digit
\s	A white space character
\S	A non-whitespace character
^	Beginning of a line
$	The end of a line
.	Any character
*	Any no of characters
\	Escape character

June 11, 2015 | Proprietary and Confidential | - 8 -

IGATE Speed.Agility.Imagination

This API supports a number of special characters (metacharacters) that affect the way a pattern is matched.
To treat a metacharacter as an ordinary character, precede it with a backslash (\)

8.1: Regular Expressions

Regular Expression guide

construct	Matches
[abc]	a, b, or c
[^abc]	Any character except a, b, or c
[a-zA-Z]	a through z or A through Z, inclusive
[a-d[m-p]]	a through d, or m through p:
[a-z&&[def]]	d, e, or f
[a-z&&[^bc]]	a through z, except for b and c
[a-z&&[^m-p]]	a through z, and not m through p:

IGATE Speed.Agility.Imagination

Please browse through the Pattern class specification in Javadocs. There are tables summarizing the supported regular expression constructs.

8.1: Regular Expressions

Example

```
public static void validateCode(String args) throws Exception{
   String input = “Ex01";
 //Checks for string that start with upper case alphabet and end with digit.
    Pattern p = Pattern.compile(“^[A-Z][0-9]&");
    Matcher m = p.matcher(input);
    if (!m.find()) {
      System.err.println(“Enter  code which  start with upper case alphabet and
end with a digit");
    }
  }
```


IGATE Speed.Agility.Imagination

Demo : Regular Expression

- **Execute the RegularExMatcher .java program**

IGATE
Speed.Agility.Imagination

Summary

- **In this lesson, you have learnt the following:**
  - What are Regular Expressions
  - Use the java.util.regex package
  - Use regular expressions for manipulating strings

Review Question

- **Question 1 : To suppress the special meaning of metacharacters, use _____**
- **Question 2 : This method returns a new Pattern object :**
  - **Option 1 :** compile()
  - **Option 2 :** matches()
  - **Option 3** : matcher()

Core Java 8 with JAXB and Development Tools

Exception Handling

IGATE
Speed.Agility.Imagination

Lesson Objectives

- **On completion of this lesson, you will be able to:**
  - Explain the concept of Exception
  - Describe types of Exceptions
  - Handle Exception in Java
  - Create your own Exceptions
  - State best practices on Exception

This lesson deals with Java's exception-handling mechanism. An exception is an abnormal condition that arises in a code sequence at run time. In other words, an exception is a run-time error.

Lesson Outline:

9.1: Introduction
9.2: Exception Types
9.3: Exception Hierarchy
9.4: Try-catch-finally
9.5: Try-with-resources
9.6: Multi catch blocks
9.7: Throwing exceptions using throw
9.8: Declaring exceptions using throws
9.9: User defined Exceptions
9.10: Best Practices

5.1: Exception Handling – Fundamentals

Why is exception handling used?

- **No matter how well-designed a program is, there is always a chance that some kind of error will arise during its execution, for example:**
  - Attempting to divide by 0
  - Attempting to read from a file which does not exist
  - Referring to non-existing item in array
- **An exception is an event that occurs during the execution of a program that disrupt its normal course.**

June 11, 2015 | Proprietary and Confidential | - 3 -

IGATE Speed.Agility.Imagination

Why Exception Handling?

Java was designed with the understanding that errors occur, that unexpected events happened and the programmer should always be prepared for the worst. The preferred way of handling such conditions is to use exception handling, an approach that separates a program's normal code from its error-handling code.

5.1: Exception Handling – Fundamentals

Exception Handling

- **Exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions:**
  - Eg: Hard disk crash, Out of bounds array access, Divide by zero etc
- **When an exception occurs, the executing method creates an Exception object and hands it to the runtime system —"throwing an exception"**
- **The runtime system searches the runtime call stack for a method with an appropriate handler, to handle/catch the exception.**

June 11, 2015 | Proprietary and Confidential | - 4 -

IGATE Speed.Agility.Imagination

Exception Handling:
An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions. Exceptions are used as a way to report occurrence of some exceptional condition. Exception provides a means of communicating information about errors up through a chain of methods until one of them handles it.
Note: Java exception handling is similar to the one used in C++.

The exception mechanism is built around the throw-and-catch paradigm. When an error occurs within a Java method, the method creates an exception object and hands it off to the runtime system. This process is known as throwing an exception. The exception object contains information about the exception, including its type and the state of the program when the error occurred. To catch an exception is to take appropriate action to deal with the exception.
Java's Exception handling mechanism is managed by five keywords: try, catch, throw, throws and finally.

5.1: Exception Handling – Fundamentals

Exception Handling

- **The general form of exception handling block:**

```
try {
        //code to be monitored.
  }
  catch (Exception1 e1 ) {
      //exception handler for Type Exception1
  }
  catch (Exception2 e2 ) {
      //exception handler for Type Exception2
  }
  finally {
      // code that must be executed.
  }
```

June 11, 2015 | Proprietary and Confidential | - 5 -

IGATE Speed.Agility.Imagination

Exception Handling (Contd.):
Normally, the program code that you want to observe for exceptions is written in the try block. If an exception occurs within a try block, it is thrown. Your code can catch this exception (using the catch block), handle the situation gracefully and continue to be in a program. Any code, that absolutely must be executed, regardless of whether exception has occurred or not, can be put into finally block.

In the above code fragment, Exception1 and Exception2 are being caught. The default handler ultimately processes an exception that is not caught by your program. The default handler displays a string describing the exception, and prints a stack trace from the point at which the exception occurred.

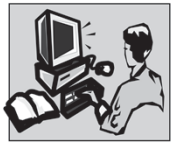
.

5.1: Exception Handling – Fundamentals

Demo

- **Execute the DefaultDemo.java program**

```
class DefaultDemo {
  public static void main(String a[]) {
    String str = null;
    str.equals("Hello");
  }
}
```



```
Output:
Exception in thread "main"
java.lang.NullPointerException at
      com.igatepatni.lesson5.DefaultDemo.main(DefaultDemo.java:6)
```

June 11, 2015 | Proprietary and Confidential | - 6 -

IGATE Speed.Agility.Imagination

The code throws a Exception:
Exception in thread "main" java.lang.NullPointerException
    at DefaultDemo.main(DefaultDemo.java:5)

This is because the String Object is not created and is therefore Null. When methods are invoked on such referenced objects, a NullPointerException is thrown!

5.1: Exception Handling – Fundamentals

Advantages of Exceptions

- **Separating Error-Handling Code from "Regular" Code:**

Code without Exception handling

```
readFile() {
   open the file;
   determine its size;
   allocate that much memory;
     read the file into memory;
     close the file;
}
```

Code with Exception handling

```
readfile() {
 try {
       open the file;
       determine its size;
       allocate that much memory;
       read the file into memory;
       close the file;
 } catch (fileOpenFailed) { doSomething; }
 catch (sizeDeterminationFailed) { doSomething; }
 catch (memoryAllocationFailed) { doSomething; }
 catch (readFailed) { doSomething; }
 catch (fileCloseFailed) { doSomething; }
 }
```

Note that error handling code and regular code are separate

June 11, 2015 | Proprietary and Confidential | - 7 -

IGATE Speed.Agility.Imagination

Advantages of Exceptions:

Exceptions provide the means to separate the details of what to do when something exceptional happens, which differs from the main logic of a program. Refer to the code snippet without Exception handling. At first glance, this function seems simple enough, but it ignores all the following potential errors.

What happens if the file can't be opened?
What happens if the length of the file can't be determined?
What happens if enough memory can't be allocated?
What happens if the read fails?
What happens if the file can't be closed?

Advantages of Exceptions (Contd.):

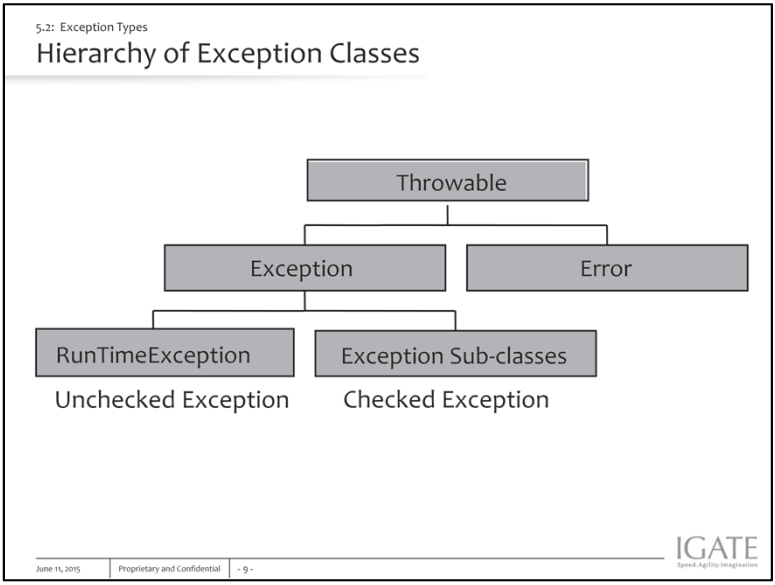
To handle such cases, the **readFile** function must have more code to do error detection, reporting, and handling. Here is an example of the function code.

```
errorCodeType readFile() {
Initialize errorCode = 0;
open the file;
 if (theFileIsOpen) {
        determine the length of the file;
        if (gotTheFileLength) {
                        allocate that much memory;
        if (gotEnoughMemory) {
                        read the file into memory;
         if (readFailed) {
                        errorCode = -1;
        } }
        else { errorCode = -2; } }
        else { errorCode = -3; }
        close the file;
        if (theFileDidntClose && errorCode == 0) {
        errorCode = -4;  }
else { errorCode = errorCode and -4; }
 } else { errorCode = -5; } return errorCode; }
```

There is so much error detection, reporting, and returning here that the original seven lines of code are lost in the clutter. Moreover, the logical flow of the code has also been lost. This maks it difficult to check whether the code is performing the right function: Is the file really being closed if the function fails to allocate enough memory? It's even more difficult to ensure that the code continues to carry out the right function when you modify the method three months after writing it. Many programmers solve this problem by simply ignoring it — errors are reported when their programs crash.

Exceptions enable you to write the main flow of your code and to deal with the exceptional cases elsewhere. If the **readFile** function used exceptions instead of traditional error-management techniques, it would be coded as shown in box #2 in the previous slide.

Note that exceptions do not spare you the effort involved in detecting, reporting, and handling errors; however, they help you organize the work more effectively.



Hierarchy of Exception Classes:

All exceptions are derived from the java.lang.Throwable class. Throwable is at top of the execution class hierarchy. Immediately below Throwable are two subclasses, Error and Exception, which categorize exceptions into two distinct branches.

Exception class: This class is used for exceptional conditions that user programs should catch. This is also the class that you use as a subclass to create your own custom exception types. There is an important subclass of Exception, called RuntimeException.
Error class: This class defines exceptions that are not expected to be caught under normal circumstances by your program. Exceptions of type Error are used by the Java run-time system to indicate errors having to do with the run-time environment, itself. Stack overflow is an example of such an error.

5.2.1: Exception Types -> Errors

Error

- **An Error is a subclass of Throwable that indicates serious problems that a reasonable application should not try to catch.**
- **Most such errors are abnormal conditions.**
- **Exceptions of type Error are used by the Java run-time system to indicate errors having to do with the run-time**
- **environment, itself.**
  - Stack overflow is an example of such an error.

June 11, 2015 | Proprietary and Confidential | - 10 -

IGATE Speed.Agility.Imagination

Error:

Instances of Error are internal errors in Java runtime environment. These are rare and usually fatal and therefore not supposed to be handled by the program. Instances of error are thrown, when the Java Virtual Machine faces some memory leakage problem, insufficient memory problem, dynamic linking failure or when some other "hard" failure in the virtual machine occurs. Obviously, in case of an error, the program stops executing.

5.2.2: Exception Types

Exception

- **The Exception class and its subclasses are a form of Throwables. They indicate conditions, which a reasonable application may want to catch.**
- **Two Types:**
  - Checked Exception
  - UnChecked Exception

June 11, 2015 | Proprietary and Confidential | - 11 -

IGATE Speed.Agility.Imagination

Java distinguishes between two categories of exceptions:
Checked exceptions
Unchecked exceptions

This distinction is important, because the Java compiler enforces a catch-or-declare requirement for checked exceptions. An exception's type determines whether the exception is checked or unchecked. All classes that inherit from the Exception class but not the RuntimeException class are considered to be checked exceptions. The compiler checks each method call and method declaration to determine whether the method throws checked exceptions. If so, the compiler ensures that the checked exception is caught or is declared in a throws clause.

5.2: Exception types

Checked/Compile Time Exceptions

- **Characteristics of Checked Exceptions:**
  - They are checked by the compiler at the time of compilation.
  - They are inherited from the core Java class Exception.
  - They represent exceptions that are frequently considered "non-fatal" to program execution.
  - They must be handled in your code, or passed to parent classes for handling.
  - Some examples of Checked exceptions include: IOException, SQLException, ClassNotFoundException

June 11, 2015 | Proprietary and Confidential | - 12 -

IGATE Speed.Agility.Imagination

Example:

```
try    {
       DriverManager.registerDriver (new
oracle.jdbc.driver.OracleDriver());
       Connection conn =
DriverManager.getConnection (
            "jdbc:oracle:thin:@DbServer:trgdb",
          "scott", "tiger");
         ……
       } catch (SQLException e) {

       System.out.println(e.getMessage());
       }
```

An SQLException can be thrown while attempting to connect to database. Hence must be caught.

5.2: Exception Types

Unchecked/Runtime Exceptions

- **Unchecked exceptions represent error conditions that are considered "fatal" to program execution.**
  - Runtime exceptions are exceptions which are not detected at the time of Compilation.
  - They are encountered only when the program is in execution.
  - It is called unchecked exception because the compiler does not check to see if a method handles or throws these exceptions.

June 11, 2015 | Proprietary and Confidential | - 13 -

IGATE Speed.Agility.Imagination

Unchecked/Runtime Exceptions:
These kind of exceptions are encountered only while the program is in execution. Program may take actions to handle them explicitly or are propagated till java run-time handles them. In the latter case, program is terminated abruptly.
Before you learn to handle exceptions in a program, it is useful to understand what happens when they are not handled. This program includes a code that causes NullPointerException because an object of the String class is not created.

```
class DefaultDemo {
        public static void main(String a[]) {
                        String str = null;
                        str.equals("Hello");
        }
}
```

This program will compile properly. When java run-time system detects the attempt to call equals() method on null object, it constructs a new exception object and throws this exception. This halts the execution of program, because once an exception has been thrown, it must be caught by an exception handler and dealt with immediately. As there is no explicit exception handler defined, it is handled by the default handler. The default handler displays a string describing the exception; prints stack trace and terminates the program.

Unchecked/Runtime Exceptions (Contd.):

```
Exception in thread "main" java.lang.NullPointerException
    at DefaultDemo.main(DefaultDemo.java:4)
```

In this, the class name **DefaultDemo**; the method name **main**, the file name **DefaultDemo.java** and line number **4**, are all included in simple stack trace. The type of thrown exception is **NullPointerException**.
Many exception types are in this category. They are always thrown automatically by Java and they need not be included in your exception specifications.
Conveniently enough, they're all grouped together by putting them under a single base class called **RuntimeException**, which is a perfect example of inheritance: it establishes a family of types that have some characteristics and behaviors in common.
In addition, you never need to write an exception specification saying that a method might throw a **RuntimeException**, since that's just assumed. Because they indicate bugs, you virtually never catch a **RuntimeException** – it's dealt with automatically. If you are forced to check for **RuntimeExceptions**, your code could get messy. Even though you don't typically catch **RuntimeExceptions**, in your own packages you might choose to throw some of the **RuntimeExceptions**.

What happens when you don't catch such exceptions? Since the compiler doesn't enforce exception specifications for these, it's quite plausible that a **RuntimeException** could percolate all the way to your **main( )** method without being detected.

5.3: Handling Exceptions

Keywords for handling Exceptions

- try : This marks the start of a block associated with a set of exception handlers.
- catch : The control moves here if an exceptions is generated.
- finally :This is called irrespective of whether an exception has occurred or not.
- throws : This describes the exceptions which can be raised by a method.
- throw : This raises an exception to the first available handler in the call stack, unwinding the stack along the way.

IGATE
Speed.Agility.Imagination

Mnemonic Hint: Throw two and try to catch finally.

5.3: Handling Exceptions

Why to handle exceptions?

- **Without Exception handling**

```
class WithoutException {
  public static void main(String args[]) {
    int d = 0;
    int a = 42 / d;
    System.out.println("Will not be printed"); } }
```

- **With Exception handling**

```
class WithExceptionHandling {
  public static void main(String args[]) {
    int d=0, a;
    try {
      a = 42 / d;
      System.out.println("This will not be printed.");
    } catch (ArithmeticException e) {
      System.out.println("Division by zero.");
    }
System.out.println("This will get printed"); } }
```

June 11, 2015 | Proprietary and Confidential | - 16 -

IGATE Speed.Agility.Imagination

Why to handle exceptions?
The code in box #1includes an expression that intentionally causes a divide-by-zero error.
When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then throws this exception. This causes the execution of WithoutException to stop, because once an exception has been thrown, it must be caught by an exception handler and dealt with immediately.
In this example, exception handlers haven't been supplied, so the exception is caught by the default handler provided by the Java run-time system. Any exception that is not caught by this program is ultimately processed by the default handler. The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program. Here is the output generated when this example is executed.

```
java.lang.ArithmeticException: / by zero
at WithoutException.main(Exc0.java:4)
```

Handling an Exception yourself provides two benefits. First, it allows you to fix the error. Second, it prevents the program from automatically terminating. The user of your application would be confused if your program stopped running and printed a stack trace whenever an error occurred.

Why to handle exceptions? (Contd.)

Fortunately, it is quite easy to prevent this. To guard against and handle a run-time error, simply enclose the code that you want to monitor inside a try block. Immediately following the try block, include a catch clause that specifies the exception type that you wish to catch. The code in box #2 includes a try block and a catch clause which processes the ArithmeticException generated by the division-by-zero error.
This program generates the following output:
Division by zero.
This will get printed
Notice that the call to println( ) inside the try block is never executed. Once an exception is thrown, program control transfers out of the try block into the catch block. Thus, the line "This will not be printed." is not displayed. Once the catch statement has executed, program control continues with the next line in the program following the entire try/catch mechanism.

5.3: Handling Exceptions

Try and Catch

- **The** try **structure has three parts:**
  - The try block : Code in which exceptions are thrown
  - One or more catch blocks : Respond to different Exceptions
  - An optional finally block : Contains code that will be executed regardless of exception occurring or not
- **The** catch **Block:**
  - If exception occurs in try block, program flow jumps to the catch blocks.
  - Any catch block matching the caught exception is executed.

June 11, 2015 | Proprietary and Confidential | - 18 -

IGATE Speed.Agility.Imagination

Syntax for the try and catch block is shown on page 5-05.
The default exception handler provided by java run-time system is useful for debugging. However, it's a good programming practice for the user to handle the exceptions.
The same example with exception handling is shown below. Observe the difference in the output. A try statement must be accompanied by at least one catch block and if required, one finally block.

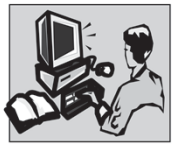
```
class TryCatchDemo {
  public static void main(String a[]) {
      String str = null;
       try {
                    str.equals("Hello");
       } catch(NullPointerException ne) {
                    str = new String("Hello");
                    System.out.println(str.equals("Hello"));
         }
      System.out.println("Continuing in the program");
      }
}
```

The output is:
True
Continuing in the program…

5.3: Handling Exceptions

Using Try and Catch

- Execute the TryCatchDemo.java program



IGATE
Speed.Agility.Imagination

The code is shown in the previous page.

5.3: Handling Exceptions

Try-with-resources

- **Resources used by java programs like file or database connection needs to be closed properly**
- **To close resource automatically in exception handling, Java 7 has added try-with-resources:**

```
try (resource1; resource 2; ….. resource n) {
    //resource related work
}
catch (Exception e) {
   //handle exception
}
```

- **It can be used to close resources that implement java.lang.AutoCloseable**

June 11, 2015 | Proprietary and Confidential | - 20 -

IGATE Speed.Agility.Imagination

Many times Java programs need to work with resources like file, database connection or network socket etc. After work, such resources must be closed gracefully to avoid loss of data. There are two places in exception handling where resource can be closed.

Finally Block (will be discussed later)
try-with-resources

A try-with-resources is a new feature added in java 7, where resources are closed automatically. Any block after try (either catch or finally block) will be executed only after the resource is closed.

5.3: Handling Exceptions

Multiple Catch Blocks

- **If you include multiple catch blocks, the order is important.**

```
public void divide(int x,int y)
{
    int ans=0;
    try{
            ans=x/y;
    }catch(Exception e) {
        //handle }
    catch(ArithmeticException f) {
        //handle}
```

Error!

You must catch subclasses before their ancestors

IGATE Speed.Agility.Imagination

Multiple Catch Blocks:
The following example shows the uses of multiple catch clauses.

```
class MultiCatch {
  public static void main(String args[]) {
      try {
          int a = args.length;
          System.out.println("a = " + a);
          int b = 42 / a;
          int c[] = { 1 };
          c[42] = 99;
      } catch(ArithmeticException e) {
            System.out.println("Divide by 0: " + e);        }
     catch(ArrayIndexOutOfBoundsException e) {
            System.out.println("Array index oob: " + e); }
        catch(Exception e) {
           System.out.println("Generic Exception: " + e); }
  System.out.println("After try/catch blocks.");
}
```

Output:

```
a = 0
Divide by 0: java.lang.ArithmeticException: / by zero
After try/catch blocks.
```

5.3: Handling Exceptions

Multi-Catch Blocks

- **Starting from Java 7, a single catch block can be used to catch multiple exceptions**
- **Multi-catch block separates handled exceptions by vertical bar (|)**

```
try {
}
catch (exception 1 | exception 2 | …..| exception n) {
}
```

- **Alternatives in multi-catch block is not allowed**

June 11, 2015 | Proprietary and Confidential | - 22 -

IGATE Speed.Agility.Imagination

Multiple-catch blocks:

Java 7 allows a single catch block to catch multiple exceptions. The only precaution required is not to list multiple exceptions which are related by sub classing. It means you cannot catch child and parent exception in multi-catch block. The following example throws an error as ArithmeticException is subclass of Throwable and hence cannot be combined in multi-catch block.

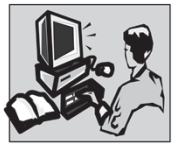
```
class MultiCatch {
   public static void main(String args[]) {
        try {
            int a = args.length;
            System.out.println("a = " + a);
            int b = 42 / a;
            int c[] = { 1 };
            c[42] = 99;
        } catch(ArithmeticException | Throwable e) {
              System.out.println("Exception: " + e);
        }
       System.out.println("After try/catch blocks.");
}
```

Error!

5.3: Handling Exceptions

Multiple Catch Blocks

- Execute the MultiCatch.java class



June 11, 2015 | Proprietary and Confidential | - 23 -

IGATE Speed.Agility.Imagination

The code for this example is shown in the previous page.
The last catch block is a generic catch block, which can catch all kinds of exceptions. This is a generalized catch block and should always appear as a last block in the list of catch blocks. If first two blocks cannot handle the exception thrown, it will be definitely handled by a generic block.
Try and catch blocks can be nested and if inner catch blocks are unable to handle an exception, it's escalated to the outer catch blocks. This continues until either one of the catch blocks handles the exception or all the try statements are exhausted.

5.3: Handling Exceptions

Nested Try Catch Block

```
try {
 int a = arg.length;
 int b = 10 / a;
 System.out.println("a = " + a);
    try {
            if(a==1)
              a = a/(a-a);
            if(a==2) {
            int c[] = { 1 };
               c[42] = 99;
            }
    } catch(ArrayIndexOutOfBoundsException e) {
    System.out.println("Array index out-of-bounds: " + e); }
 } catch(ArithmeticException e) {
       System.out.println("Divide by 0: " + e); }
```

June 11, 2015 | Proprietary and Confidential | - 24 -

IGATE Speed.Agility.Imagination

Nested Try Catch Block:
The try statement can be nested. If an inner try statement does not have a catch handler for a particular exception, the stack is unwound and the next try statement's catch handlers are inspected for a match. This continues until one of the catch statements succeeds, or until all of the nested try statements are exhausted. If no catch statement matches, then the Java run-time system handles the exception.

```
class Nesteddemo {
 public static void main(String arg[]) {
   try {
                int a = arg.length;
                int b = 10 / a;
                System.out.println("a = " + a);
          try {
             if(a==1)
                a = a/(a-a);
                  if(a==2) {
                  int c[] = { 1 };
                     c[42] = 99;
                    }
             } catch(ArrayIndexOutOfBoundsException e){
                   System.out.println("Array index out-of-bounds:"+ e); }
          } catch(ArithmeticException e) {
                     System.out.println("Divide by 0: " + e); } } }
```

5.3: Handling Exceptions

The Finally Clause

- **The finally block is optional.**
- **It is executed whether or not exception occurs.**

```
public void divide(int x,int y)
{
    int ans;
      try{
          ans=x/y;
      }
      catch(Exception e) {
          ans=0;  }
      finally{
          return ans;  // This is always executed }
}
```

June 11, 2015 | Proprietary and Confidential | - 25 -

IGATE Speed.Agility.Imagination

The Finally Clause:

When exceptions are thrown, execution in a method takes a rather abrupt, nonlinear path that alters the normal flow through the method. Depending upon how the method is coded, the method may return prematurely. For example, if a method opens a database connection on entry and closes it upon exit, then you will not want the code that closes the file to be bypassed by the exception-handling mechanism. The finally keyword is designed to address this contingency.

Finally creates a block of code that is executed after try/catch block has completed and before the code following the try/catch block. The finally block executes whether or not an exception is thrown.

If an exception is thrown, the finally block executes even if no catch statements matches the exception. Finally is guaranteed to execute, even if no exceptions are thrown. The Finally block is an ideal position for closing the resources such as file handle or a database connection, and so on.

A finally block typically contains code to release resources acquired in its corresponding try block; this is an effective way to eliminate resource leaks. For example, the finally block should close any files opened in the try block.

5.3: Handling Exceptions

Demo : The Finally Clause

- **Execute the FinallyDemo.java program**

Output of this prg:

inside procA
procA's finally
Exception caught
inside procB
procB's finally
inside procC
procC's finally

IGATE Speed.Agility.Imagination

```
class FinallyDemo {
   static void procA() {
       try {
           System.out.println("inside procA");
           throw new RuntimeException("demo");
          } finally {   System.out.println("procA's finally"); }
       }
    static void procB() {    // Return from within a try block.
       try {
            System.out.println("inside procB");
            return;
        } finally {  System.out.println("procB's finally");  }
       }
static void procC() {    // Execute a try block normally.
   try {
           System.out.println("inside procC");
       } finally {   System.out.println("procC's finally");  }
}
public static void main(String args[]) {
    try {
           procA();
       } catch (Exception e) { System.out.println("Exception caught"); }
          procB(); procC();
       }  }
```

5.3: Handling Exceptions

Throwing an Exception

- **You can throw your own runtime errors:**
  - To enforce restrictions on use of a method
  - To "disable" an inherited method
  - To indicate a specific runtime problem
- **To throw an error, use the** throw **Statement**
  - throw ThrowableInstance
    where ThrowableInstance is any Throwable Object

June 11, 2015 | Proprietary and Confidential | - 27 -

IGATE Speed.Agility.Imagination

Throwing an Exception:

It is possible for your program to throw an exception explicitly, using the throw statement.

Throwing an Exception

```
class ThrowDemo {
  void proc() {
     try {
          throw new ArithmeticException("From Exception");
     } catch(ArithmeticException e) {
          System.out.println("Caught inside demoproc.");
          throw e; // rethrow the exception
  } }
public static void main(String args[]) {
   ThrowDemo t=new ThrowDemo();
    try {
          t.proc();
     } catch(ArithmeticException e) {
          System.out.println("Recaught: " + e);
      }
} }
```


IGATE Speed.Agility.Imagination

Throwing an Exception:

This program gets two chances to deal with the same error. First, main( ) sets up an exception context and then calls proc( ). The proc( ) method then sets up another exception-handling context and immediately throws a new instance of ArithmeticException, which is caught on the next line. The exception is then rethrown. Here is the resulting output:
Caught inside demoproc.
Recaught: java.lang.ArithmeticException: From Exception

The program also illustrates how to create one of Java's standard exception objects.
Example:

```
throw new ArithmeticException();
```

Here, new is used to construct an instance of ArithmeticException. All of Java's built-in run-time exceptions have at least two constructors: one with no parameter and one that takes a string parameter.

5.3: Handling Exceptions

Using The Throws Clause

- **If a method might throw an exception, you may declare the method as "throws" that exception and avoid handling the exception yourself.**

```
class ThrowsDemo {
  public static void main(String args[]) {
    try {
      doWork();
    } catch (ArithmeticException e) {
        System.out.println("Exception: " + e.getMessage());
  } }
  static void doWork() throws ArithmeticException {
      int array[] = new int[100];
      array[100] = 100; } }
```

June 11, 2015 | Proprietary and Confidential | - 29 -

IGATE Speed.Agility.Imagination

Using The Throws Clause:

If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception. This can be achieved by using the Throws clause in the method declaration.
The Throws clause can throw multiple exceptions separated by commas. It lists the type of exceptions that a method might throw.
Here is the output generated by running this example program:

```
Exception in thread "main"
      java.lang.ArrayIndexOutOfBoundsException: 100
    at ThrowsDemo.doWork(ThrowsDemo.java:11)
    at ThrowsDemo.main(ThrowsDemo.java:4)
```

The Throws clause is added to an application to indicate to the rest of the program that this method may throw an ArithmeticException. Clients of method doWork() are thus informed that the method may throw an ArithmeticException and that the exception should be caught.

5.4: Creating Application specific Exception

User specific Exception

- **To create exceptions:**
  - Write a class that extends(indirectly) Throwable.
  - What Superclass to extend?
    - For unchecked exceptions: RuntimeException
    - For checked exceptions: Any other Exception subclass or the Exception itself

```
class AgeException extends Exception {
   private int age;
   AgeException(int a) {
      age = a;
   }
  public String toString() {
     return age+" is an invalid age";
} }
```

June 11, 2015 | Proprietary and Confidential | - 30 -

IGATE Speed.Agility.Imagination

User-Specific Exception:

To create your own exception class, you must inherit from an existing exception class, preferably one that is close in meaning to your new exception.
Define a subclass of Exception. Your subclasses don't need to actually implement anything—it is their existence in the type system that allows you to use them as exceptions. The Exception class does not define any methods of its own. It does, of course, inherit those methods provided by Throwable. Thus, all exceptions, including those that you create, have the methods defined by Throwable available to them. The following are some of the useful methods.
String toString( ): This exception returns a String object containing a description of the exception. This method is called by println( ) when outputting a Throwable object.
String getMessage( ): This exception returns a description of the exception.
void printStackTrace( ) : This exception displays the stack trace.

User Specific Exception (Contd.):

Example:

```
import java.util.*;
        class AgeException extends Exception {
            private int age;
            AgeException(int a) {
                age = a; }
            public String toString() {
                return age+" is an invalid age"; } }
        class emp {
            String name;
            int age;
            void getDetails() throws AgeException {
                 System.out.println("Enter your name:");
                 Scanner sc=new Scanner(System.in);
                 name=sc.next();
                 System.out.println("Enter your age:");
                 age=sc.nextInt();
                 if (age<16)
                    throw new AgeException(age);
         }}
        class ExceptionDemo {
            public static void main(String args[]) {
                try {
                    emp e=new emp();
                    e.getDetails();
                }catch (AgeException e) {
                     System.out.println( e); }
        } }
```

This example defines a subclass of **Exception** called **AgeException**. This subclass is quite simple: it has only a constructor and an overloaded **toString( )** method that displays the value of the exception. The **ExceptionDemo** class invokes **getDetails()** method of **emp** class. The **getDetails** method throws **AgeException** object if age is less than 16. The **main( )** method sets up an exception handler for **AgeException**, then calls **getDetails().**

The output generated is as follows:
Enter your name:
Suman
Enter your age:
12
12 is an invalid age

5.4: Creating Application specific Exception

Demo : User Specific Exception

- **Execute UserException.java program**

June 11, 2015 | Proprietary and Confidential | - 32 -

IGATE Speed.Agility.Imagination

```
class ApplicationException extends Exception {
    private int detail;
    ApplicationException(int a) { detail = a;   }
    ApplicationException(String args) { super(args);  }
    public String toString() {return "ApplicationException["+detail+"]"; }
}
class UserException {
   static void compute(int a) throws ApplicationException {
        System.out.println("called compute("+a+")");
        if (a>10)  throw new ApplicationException(a);
        System.out.println("Normal Exit");
    }
   public static void main(String arg[]) {
       try {
           compute(1);
           compute(20);
       } catch (ApplicationException e) { System.out.println("caught  "+e); }
       }
}
```

Output:
called compute(1)
Normal Exit
called compute(20)
caught ApplicationException[20]

5.4: Creating Application specific Exception

Lab : Exception

- **Lab 6: Exception Handling**

IGATE
Speed.Agility.Imagination

5.5: Best Practices on Exception Handling

The Best Practices

- Avoid empty catch blocks.
- Avoid throwing and catching a generic exception class.
- Pass all the pertinent data to exceptions.
- Use the finally block to release the resources

IGATE
Speed.Agility.Imagination

The Best Practices:
Avoid empty catch blocks:
Most contend that it is usually not preferable to have an empty catch block. When the exception occurs, nothing happens, and the program fails for unknown reasons.

For example, if a problem with user input is detected and an exception is thrown as a result, then merely informing the user of the problem might be all that is required. For example, a message might read : Age must be in range 0..120

Avoid throwing and catching generic exception class:
If the exception is known then, catch a specific exception class.
In the Throws clause of a method header, be as specific as possible. Do not group together related exceptions in a generic exception class - that would represent a loss of possibly important information.

Use the finally block to release the resources like a database connection, closing a file or socket connection, and others.
This prevents resource leaks even if an exception occurs. The Finally block always execute irrespective of whether exception is thrown or not.

Best Practices

- Avoid throwing unnecessary exceptions.
- Finalize method is not reliable.
- Exception thrown by finalizers are ignored.
- While using method calls, always handle the exceptions in the method where they occur.
- Do not use loops for exception handling.

June 11, 2015 | Proprietary and Confidential | - 35 -

IGATE Speed.Agility.Imagination

Best Practices:

Avoid Throwing Unnecessary Exceptions: Creating and throwing an exception is a time-consuming process, so you should avoid throwing exceptions when it's not necessary to do so. For example, you might define a method such as the one shown here that returns an object from a list until there are no more in the list. If the method is invoked when there are no objects remaining in the list, a NoMoreObjectsException is thrown:

```
public Object getNextObject() throws NoMoreObjectsException {
  // ...
}
```

Instead of throwing a NoMoreObjectsException when there are no objects remaining in the list, you might change this method so that it returns a null value, which will allow your code to execute more quickly when that occurs.

Finalize method is not reliable: Since the Garbage collector is JVM specific and it is not always sure when the garbage collector is invoked, it is thus not sure when the finalize method is executed. So, avoid releasing resources which are important and are to be referred again in the code using the finalize method.

Best Practices

- **When deciding on checked exceptions vs. unchecked exceptions, ask yourself, "What action can the client code take when the exception occurs?"**

IGATE
Speed.Agility.Imagination

Best Practices:
Always remember that exceptions thrown by finalizers are ignored.

While using method calls, always handle the exceptions in the method where they occur. Do not allow them to propagate to the calling method unless it is specifically required. It is efficient to handle them locally since allowing them to propagate to the calling method takes more execution time.

Do not use Exception handling in loops. It is better to place loops inside the try or catch block than vice versa.

Using Check or Uncheck Exception:
If the client can take some alternate action to recover from the exception, make it a checked exception. If the client cannot do anything useful, then make the exception unchecked. Being useful means, able to take steps to recover from the exception and not just log the exception. To summarize:

Client's reaction when exception happens	**Exception type**
1.Client code cannot do anything	1.Make it an unchecked exception
2.Client code will take some useful recovery	2. Make it a checked action based on information in exception

Summary

- **Exceptions are powerful error handling mechanisms.**
- **A program can catch exceptions by using a combination of the try, catch, and finally blocks:**
  - The try block identifies a block of code in which an exception can occur.
  - The catch block identifies a block of code, known as an exception handler.
  - The finally block identifies a block of code that is guaranteed to execute.
  - Try-with-resources
  - Multi-catch blocks
- **Throw is used to throw an exception by the user.**

IGATE
Speed.Agility.Imagination

Add the notes here.

Review – Match the Following format

1. CheckedException	A. Compulsory to use if a method throws a checked exception and doesn't handle it
2. finally	B. Inherited from RuntimeException
3. throws	C. Can have any number of catch blocks
4. Unchecked Exception	D. Used to avoid "resource leak"
5. try	E. Inherited from Exception

Core Java 8 with JAXB and Development Tools

Arrays

IGATE
Speed.Agility.Imagination

Lesson Objectives

- **After completing this lesson, participants will be able to**
  - Understand the different types of Arrays
  - Implement one and multi dimensional arrays
  - Iterate arrays using loops
  - Use varargs
  - Work with java.util.Arrays

June 11, 2015 | Proprietary and Confidential | - 2 -

IGATE
Speed.Agility.Imagination

This lesson discusses about how to work java arrays.

Lesson outline:

10.1: One dimensional array
10.2: Multidimensional array
10.3: Using varargs
10.4: Using Arrays class
10.5: Best Practices

10.1: Introduction to Array

Arrays

- **Arrays are used to group elements of either of primitive or reference types**
- **Array in java is created as Object:**
  - This object will help developers to find size of array
  - Using this object developers can manipulate array
  - Can be compared with null
- **All elements of array of same type**
- **Array is a fixed-length data structure having zero-based indexing**

June 11, 2015 | Proprietary and Confidential | - 3 -

IGATE Speed.Agility.Imagination

Arrays can be created for either primitive or reference type elements. Array in java is created as Object using new operator. Once array is created, individual elements can be accessed using index number enclosed in square brackets.

Arrays indexing is zero based, it means the first element of array start at index 0, second element is at 1, and so on. The last element of array is indexed as one minus size of an array.

Array size can be captured by using public final instance variable called length. This feature will avoid any runtime exceptions resulted due to out of bounds access.

10.1: Arrays

Arrays

- **A group of like-typed variables referred by a common name**
- **Array declaration and initialization:**
  - int arr [];
    arr = new int[10];
  - int arr[] = {2,3,4,5};
  - int twoDim [][] = new int[4][5];

June 11, 2015 | Proprietary and Confidential | - 4 -

IGATE Speed.Agility.Imagination

Syntax wherein the array is declared and initialized in the same statement:

```
strWords = { "quiet", "success", "joy", "sorrow", "java" };
```

10.1: Arrays

Creating Array Objects

- **Arrays of objects too can be created:**
  - Example 1:

```
Box barr[] = new Box[3];
barr[0] = new Box();
barr[1] = new Box();
barr[2] = new Box();
```

  - Example 2:

```
String[] Words = new String[2];
Words[0]=new  String("Bombay");
Words[1]=new  String("Pune");
```

June 11, 2015 | Proprietary and Confidential | - 5 -

IGATE Speed.Agility.Imagination

Use new operator or directly initialize an array. When you create an array object using new, all its slots are initialized for you (0 for numeric arrays, false for boolean, '\0' for character arrays, and null for objects).
Like single dimensional arrays, we can form multidimensional arrays as well. Multidimensional arrays are considered as array of arrays in java and hence can have asymmetrical arrays. See an example next.

10.1: Arrays Demo

Demo

- **Executing the ArrayDemo.java program**

June 11, 2015 | Proprietary and Confidential | - 6 -

IGATE Speed.Agility.Imagination

We have seen how to pass parameters to program during compile time using parameter passing. One can pass parameters to a program at runtime too. The args parameter (a String array) in main() receives command line arguments.

```
class ArrayDemo  {
   int intNumbers[];
   ArrayDemo(int i) {
      intNumbers = new int[i];
   }
   void populateArray() {
       for(int i = 0; i < intNumbers.length; ++i) intnumbers[i] = i;
    }
   void displayContents() {
        for(int i = 0; i <intNumbers.length; ++i)
             System.out.println("Number " + i + ": " + intNumbers[i]);
   }
   public static void main(String[] args)                    {
      //Accepting array length as command line argument.
       int intArg = Integer.parseInt(args[0]);
       ArrayDemo ad = new ArrayDemo(intArg);
       ad.displayContents();
       ad.populateArray();
       ad.displayContents();
   } }
```

10.2: Iterating Array

Enhanced for Loop (foreach)

- **New feature introduced in Java 5**
- **Iterate through a collection or array**
  - Syntax:

```
for (variable : collection)
{ //code}
```

  - Example

```
int sum(int[] intArray)
{
    int result = 0;
    for (int index : intArray)
        result += index;
    return result;
}
```

June 11, 2015 | Proprietary and Confidential | - 7 -

IGATE Speed.Agility.Imagination

Enhanced for loop works with collections and arrays. Notice the difference between the old code where the three standard steps of initialization, conditional check and re-initialization are explicitly required to be mentioned.

Old code

```
public class OldForArray {
  public static void main(String[] args){
     int[] squares = {0,1,4,9,16,25};
     for (int i=0; i< squares.length; i++){
         System.out.printf("%d squared is %d.\n",i, squares[i]);
     }
  }}
```

New Code

```
public class NewForArray {
   public static void main(String[] args) {
      int j = 0;
      int[] squares = {0, 1, 4, 9, 16, 25};
      for (int i : squares) {
         System.out.printf("%d squared is %d.\n", j++, i);
      }
   }
}
```

10.3 : Method with Variable Argument Lists

Variable Argument List

- **New feature added in J2SE5.0**
- **Allows methods to receive unspecified number of arguments**
- **An argument type followed by ellipsis(…) indicates variable number of arguments of a particular type**
  - *Variable-length* argument can take from *zero* to *n* arguments
  - Ellipsis can be used only once in the parameter list
  - Ellipsis must be placed at the end of the parameter list

June 11, 2015 | Proprietary and Confidential | - 8 -

IGATE Speed.Agility.Imagination

J2SE5 has added a new feature that simplifies the creation of methods that need to take a variable number of arguments. This feature is called varargs and it is short for variable-length arguments.

10.3 : Method with Variable Argument Lists

Variable Argument List (contd..)

```
//Valid Code
void print(int a,int b,String...c)
{
    //code
}
```

```
//Invalid  Code
void print(int a, int b… ,float c)
{
    //code
}
```

- **The above print function can be invoked using any of the invocations:**
  - print(1,1,"XYZ")
  - print(2,5)
  - print(5,6,"A","B")

Varargs can be used only in the final argument position.

IGATE Speed.Agility.Imagination

Note that the ellipses (…) must come only after the last parameter. Putting it anywhere else is invalid.
The three periods indicate that the final argument may be passed as an array or as a sequence of arguments.

10.3 : Method with Variable Argument Lists

Demo

- **Execute the varargs.java program**

June 11, 2015 | Proprietary and Confidential | - 10 -

IGATE Speed.Agility.Imagination

```
import static java.lang.System.*;
        public class varargs {
            static void print(int a,int y,String...s) {
                out.println(a+""+y);
                for(int i=0;i<s.length;i++) out.print(s[i]+"\t");
                    out.println();
            }
            public static void main(String[] arg) {
                print(3,2,"java","java 5");
                out.println("Next invoke");
                print(1,2,"a","b","c","d","e");
            } }
```

Java Arrays expose a property called length that returns the length of the array.

O/P:
32
java java 5
Next invoke
12
a b c d e

10.4: Arrays Class

Using java.util.Arrays Class

- **This class contains lots of useful methods to manipulate contents of array**

Method Name	Use
asList	Creates a new List from array
binarySearch	Use to search an element in an array
copyOf(array,n)	Creates new array of n size and copy all elements from array to new one
copyOfRange(array,n,from,to)	Creates new array of n size and copy specified elements from array to new one
sort	Sort elements of an array
equals	Compare two array elements
fill	Inserts specified value to each element of an array
stream(array)	Creates stream from an array

June 11, 2015 | Proprietary and Confidential | - 11 -

IGATE Speed.Agility.Imagination

Arrays Class:

Java provides utility Arrays class to manipulate arrays. This class is provided in the java.util package and includes lots of static methods to deal with array.

Please refer the java documentation of this class to know more about method signature and use.

Note: stream() method will be discussed in later chapter.

10.4 : Arrays Class

Demo

- **Execute the UsingArrays.java program**

June 11, 2015 | Proprietary and Confidential | - 12 -

IGATE Speed.Agility.Imagination

```
import java.util.Arrays;
        public class UsingArrays{
            static void sort(int...s) {
                Arrays.sort(s);
                for(int i=0;i<s.length;i++) out.print(s[i]+" " );
                    out.println();
            }
            public static void main(String[] arg) {
                sort(15,20,42,3,56,34);
            } }
```

Summary

- **In this lesson, you have learnt about:**
  - Creating and using array
  - Manipulating array
  - Iterating array
  - Varargs
  - Using java.util.Arrays class

Review Question

- **Question 1: If a display method accepts an integer array and returns nothing , is following call to display method is correct? State true or false.**
  - display({10,20,30,40,50})
- **Question 2: All methods in java.util.Arrays class are static (excluding Object class methods).**
  - True/False

Core Java 8 with JAXB and Development Tools

Collection

IGATE
Speed.Agility.Imagination

Lesson Objectives

- **After completing this lesson, participants will be able to**
  - Understand collection framework
  - Implement and use collection classes
  - Iterate collections
  - Create collection of user defined type

June 11, 2015 | Proprietary and Confidential | - 2 -

IGATE
Speed.Agility.Imagination

This lesson discusses about collection framework in Java.

Lesson outline:

11.1: Collections Framework
11.2: Collection Interfaces
11.3: Implementing Classes
11.4: Iterating Collections
11.5: Comparable and Comparator
11.6: Best Practices

11.1: Collections Framework

Collections Framework

- **A Collection is a group of objects.**
- **Collections framework provides a set of standard utility classes to manage collections.**
- **Collections Framework consists of three parts:**
  - Core Interfaces
  - Concrete Implementation
  - Algorithms such as searching and sorting

June 11, 2015 | Proprietary and Confidential | - 3 - 3 IGATE Speed.Agility.Imagination

Collections Framework:
A Collection (sometimes called a container) is an object that groups multiple elements into a single unit. Collection is used to store, retrieve objects, and to transmit them from one method to another.
The Collections API (also called the Collections framework) standardizes the way in which groups of objects are handled by your programs. It presents a set of standard utility classes to manage such collections. This framework is provided in the java.util package and comprises three main parts:

The core interfaces, which allow collections to be manipulated independent of their implementation. These interfaces define the common functionality exhibited by collections, and facilitate data exchange between collections.

A small set of implementations, which are concrete implementations of the core interfaces, providing data structures that a program can use. Eg LinkedLists, Arrays etc

An assortment of algorithms, which can be used to perform various operations on collections, such as sorting & searching.

The collection classes are the fundamental building blocks of the more complicated data structures used in the other Java packages in your own applications. There are several types of collections. They vary in storage mechanisms used, in the way they access data, and in the rules about what data may be stored.
Note: The Java Collection technology is similar to the Standard Template Library (STL) defined by C++.

11.1: Collections Framework

Advantages of Collections

- **Collections provide the following advantages:**
  - Reduces programming effort
  - Increases performance
  - Provides interoperability between unrelated APIs
  - Reduces the effort required to learn APIs
  - Reduces the effort required to design and implement APIs
  - Fosters Software reuse

June 11, 2015 | Proprietary and Confidential | - 4 - 4 IGATE Speed.Agility.Imagination

Collections Framework:

Advantages of Collections:

Collections provide the following advantages:

Reduces programming effort by providing useful data structures and algorithms so you do not have to write them yourself.

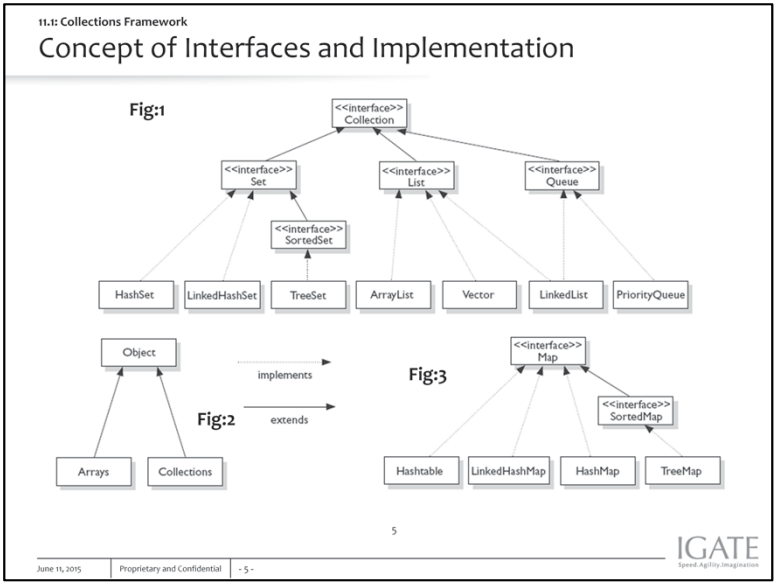
Increases performance by providing high-performance implementations of useful data structures and algorithms. Since the various implementations of each interface are interchangeable, programs can be easily tuned by switching implementations.

Provides interoperability between unrelated APIs by establishing a common language to pass collections back and forth.

Reduces the effort required to learn APIs by eliminating the need to learn multiple ad hoc collection APIs.

Reduces the effort required to design and implement APIs by eliminating the need to produce ad hoc collections APIs.

Fosters software reuse by providing a standard interface for collections and algorithms to manipulate them.



Interfaces and Implementation:
The core collection interfaces (shown in figure above) are the interfaces used to manipulate collections, and to pass them from one method to another. The basic purpose of these interfaces is to allow collections to be manipulated independently of the details of their representation.
Not all collections in the Collections Framework actually implement the Collection interface. Specifically, none of the Map-related classes and interfaces extend from Collection. So while SortedMap, Hashtable, HashMap, TreeMap, and LinkedHashMap are all thought of as collections, none are actually extended from Collection.
Note: Collections is a class, with static utility methods, while Collection is an interface with declarations of the methods common to most collections including add(), remove(), contains(), size(), and iterator().

11.2: Collection Interfaces

Collection Interfaces

- **Let us discuss some of the collection interfaces:**

Interfaces	Description
Collection	A basic interface that defines the operations that all the classes that maintain collections of objects typically implement.
Set	Extends the Collection interface for sets that maintain unique element.
SortedSet	Augments the Set interface or Sets that maintain their elements in sorted order.
List	Collections that require position-oriented operations should be created as lists. Duplicates are allowed.
Queue	Things arranged by the order in which they are to be processed.
Map	A basic interface that defines operations that classes that represent mapping of keys to values typically implement.
SortedMap	Extends the Map interface for maps that maintain their mappings in the key order.

June 11, 2015 | Proprietary and Confidential | - 6 - 6 IGATE Speed.Agility.Imagination

Interfaces and Implementation:
Collection Interfaces:
Following are the four major interfaces:

Set Interface: holds only unique values and rejects duplicates.
List Interface: represents an ordered list of objects, meaning the elements of a List can be accessed in a specific order, and by an index too. List can hold duplicates.
Queue Interface: represents an ordered list of objects just like a List. However, a queue is designed to have elements inserted at the end of the queue, and elements removed from the beginning of the queue. Just like a queue in a supermarket!
Map Interface: represents a mapping between a key and a value. The Map interface is not a subtype of the Collection interface. A Map cannot contain duplicate keys; each key can map to at most one value. The Map implementations let you do things like search for a value based on the key, ask for a collection of just the values, or ask for a collection of just the keys.
SortedSet Interface: is a Set that maintains its elements in ascending order. Several additional operations are provided to take advantage of the ordering.
SortedMap Interface: is a Map that maintains its mappings in ascending key order. This is the Map analog of SortedSet. Sorted maps are used for naturally ordered collections of key/value pairs, such as dictionaries and telephone directories.

11.2: Collection Interfa+ces

Collection Implementations

➢ **Collection Implementations:**

		Implementations				
		Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Interfaces	Set	HashSet		TreeSet		LinkedHashSet
	List		ArrayList		LinkedList	
	Map	HashMap		TreeMap		LinkedHashMap

Collection Implementations:
The Java Collections Framework provides several general-purpose implementations of the Set, List, and Map interfaces. The general purpose implementations are summarized in the table above.

HashSet: is an unsorted, unordered Set. It uses the hashcode of the object being inserted, so the more efficient your hashCode() implementation is, the better access performance you will get. Use this class when you want a collection with no duplicates and you do not care about order when you iterate through it. Implements the Set interface.
LinkedHashSet: differs from HashSet by guaranteeing that the order of the elements during iteration is the same as the order they were inserted into the LinkedHashSet.
TreeSet: implements the SortedSet interface. Like LinkedHashSet, TreeSet also guarantees the order of the elements when iterated, but the order is the sorting order of the elements. This order is determined either by their natural order (if they implement Comparable), or by a specific Comparator implementation.
ArrayList: Think of this as a growable array. It gives you fast iteration and fast random access. It is an ordered collection (by index). However, it is not sorted. ArrayList now implements the new RandomAccess interface — a marker interface (meaning it has no methods) that says, “this list supports fast (generally constant time) random access.” Choose this over a LinkedList when you need fast iteration but are not as likely to be doing a lot of insertion and deletion.
LinkedList: A LinkedList is ordered by index position, like ArrayList, except that the elements are doubly-linked to one another.

11.2: Collection Interfaces

Collection Implementations

- **Notes Page**

Collection Implementations (contd.):
This linkage gives you new methods (beyond what you get from the List interface) for adding and removing from the beginning or end. This makes it an easy choice for implementing a stack or queue. Keep in mind that a LinkedList may iterate more slowly than an ArrayList. However, it is a good choice when you need fast insertion and deletion. As of Java 5, the LinkedList class has been enhanced to implement the java.util.Queue interface. As such, it now supports the common queue methods: peek(), poll(), and offer().
HashMap: The HashMap gives you an unsorted, unordered Map. When you need a Map and you do not care about the order (when you iterate through it), then HashMap is the way to go. The other maps add a little more overhead. Where the keys land in the Map is based on the key's hashcode. So, like HashSet, the more efficient your hashCode() implementation, the better access performance you will get. HashMap allows one null key and multiple null values in a collection.
TreeMap: TreeMap is a sorted Map.
LinkedHashMap: Like its Set counterpart, LinkedHashSet, the LinkedHash-Map collection maintains insertion order (or, optionally, access order). Although it will be somewhat slower than HashMap for adding and removing elements, you can expect faster iteration with a LinkedHashMap.

11.2: Collection Interfaces

Collection Interface methods

Method	Description
int size();	Returns number of elements in collection.
boolean isEmpty();	Returns true if invoking collection is empty.
boolean contains(Object element);	Returns true if element is an element of invoking collection.
boolean add(Object element);	Adds element to invoking collection.
boolean remove(Object element);	Removes one instance of element from invoking collection
Iterator iterator();	Returns an iterator fro the invoking collection
boolean containsAll(Collection c);	Returns true if invoking collection contains all elements of c; false otherwise.
boolean addAll(Collection c);	Adds all elements of c to the invoking collection.
boolean removeAll(Collection c);	Removes all elements of c from the invoking collection
boolean retainAll(Collection c);	Removes all elements from the invoking collection except those in c.
void clear();	Removes all elements from the invoking collection
Object[] toArray();	Returns an array that contains all elements stored in the invoking collection
Object[] toArray(Object a[]);	Returns an array that contains only those collection elements whose type matches that of a.

June 11, 2015 | Proprietary and Confidential | - 9 - 9

IGATE Speed.Agility.Imagination

Collection Interface Methods:
The Collection Interface is the foundation on which the collection framework is built. It declares the core methods that all collections will have. Some of these methods are summarized in the table given in the above slide.
The bulk operations perform some operation on an entire Collection in a single shot. They are done through the following methods, namely: containsAll(), addAll(), removeAll(), retainAll(), clear().

11.5: AutoBoxing with Collections

AutoBoxing with Collections

- **Boxing conversion converts primitive values to objects of corresponding wrapper types.**

```
int intVal = 11;
Integer iReference = new Integer(i); // prior to Java 5, explicit Boxing
iReference = intVal;                 // In Java 5,Automatic Boxing
```

- **Unboxing conversion converts objects of wrapper types to values of corresponding primitive types.**

```
int intVal = iReference.intValue();   // prior to Java5, explicit unboxing
intVal = iReference;                  // In Java 5, Automatic Unboxing
```

June 11, 2015 | Proprietary and Confidential | - 10 -

IGATE Speed.Agility.Imagination

AutoBoxing with Collections:
J2SE 5 adds to the Java language autoboxing and auto-unboxing.
Primitive types and their corresponding wrapper classes can now be used interchangeably. For example: The following lines of code are legitimate in Java 5:

```
int intval1 = 0;
Integer intval2 = intval1;
int intval3 = new Integer(intval2);
```

This is often referred to as automatic boxing or unboxing.
If an int is passed where an Integer is expected, then the compiler will automatically insert a call to the Integer constructor. Conversely, if an Integer is provided where an int is required, then there will be an automatic call to the IntValue method.
Autoboxing is the process by which a primitive type is automatically encapsulated into its equivalent type wrapper whenever an object of that type is needed.
Auto-unboxing is the process by which the value of a boxed object is automatically extracted (unboxed) from type wrapper when its value is needed.

6.2: Collection Interfaces

Iterating through a collection

- **Iterator is an object that enables you to traverse through a collection.**
- **It can be used to remove elements from the collection selectively, if desired.**

```
public interface Iterator<E>
{
boolean hasNext();
E next();
void remove();
}
```

- **Iterable is an superinterface of Collection interface, allows to iterate the elements using foreach method**

```
Collection.forEach(Consumer<? super T> action)
```

June 11, 2015 | Proprietary and Confidential | - 11 - 11

IGATE Speed.Agility.Imagination

Iterators:
Java provides two interfaces that define the methods by which you can access each element of a collection: Enumeration and Iterator.

Enumeration is a legacy interface and is considered obsolete for new code. It is now superseded by the iterator interface.

The iterator() method of every collection returns an iterator to a collection. It is similar to an Enumeration, but differs in two respects:

Iterator allows the caller to remove elements from the underlying collection during the iteration with well-defined semantics.

Method names have been improved.

There is no safe way to remove elements from a collection while traversing it with an Enumeration.

In the example in the above slide, the following parameters are used:

boolean hasNext() : returns true if there are more elementsObject next() : It returns next element. It throws NoSuchElementException if there is no next element.

void remove() : It removes current element. Throws IllegalStateException if an attempt is made to call remove() that is not preceded by a call to next()

Note 1: The hasNext() method is identical in function to Enumeration.hasMoreElements(), and the next() method is identical in function to Enumeration.nextElement().

Note 2: Iterator.remove() is the only safe way to modify a collection during iteration. The behavior is unspecified if the underlying collection is modified in any other way while the iteration is in progress.

6.4: Enhanced for loop

Enhanced for loop

- **Iterating over collections looks cluttered:**

```
void printAll(Collection<Emp> employees) {
   for (Iterator<Emp> iterator = employees.iterator(); iterator.hasNext(); )
      System.out.println(iterator.next()); } }
```

- **Using enhanced** for loop**, we can do the same thing as:**

```
void printAll(Collection<Emp> employees) {
   for (Emp empObj : employees) )
      System.out.println( empObj ); }}
```

  - When you see the colon (:) read it as "in."
  - The loop above reads as "for each emp 't' in collection 'e'."

June 11, 2015 | Proprietary and Confidential | - 12 -

IGATE Speed.Agility.Imagination

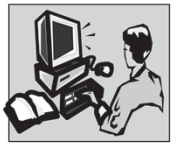
Enhanced for loop:
The enhanced for loop can be used for both Arrays and Collections:

```
class Enhancedforloop {
  static void printArray(int intArr[]) {
      for (int arrayindex : intArr )
         System.out.println(arrayindex);
  }
  static void printCollection(ArrayList arrList) {
     for (Object object : arrList)
        System.out.println(object);
  }

public static void main(String arg[]) {
   int intArr[] = { 1, 2, 3, 4, 5 };
   printArray(intArr);
   ArrayList arraylist = new ArrayList();
   arraylist.add(10);
   arraylist.add(30);
   arraylist.add(20);
   printCollection(arraylist);
   }
}
```

6.2: Collection Interfaces

Demo :Concept of Iterators

- **Execute:**
  - MailList.java
  - ItTest.java program



IGATE
Speed.Agility.Imagination

ArrayList Class

- **An ArrayList Class can grow dynamically.**
- **It provides more powerful insertion and search mechanisms than arrays.**
- **It gives faster Iteration and fast random access.**
- **It uses Ordered Collection (by index), but not Sorted.**

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(0, new Integer(42));
int total = list.get(0).intValue();
```

June 11, 2015 | Proprietary and Confidential | - 14 -

IGATE Speed.Agility.Imagination

ArrayList Class:
Let us check the power of ArrayList with an example:

```
List<String> myList = new ArrayList<String>();
```

In many ways, ArrayList<String> is similar to a String[] in that it declares a container that can hold only Strings. However, it is more powerful than a String[].
Let us look at some of the capabilities that an ArrayList has:

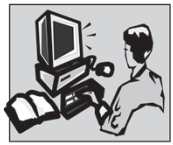
```
import java.util.*;
    public class ArrayListTest {
      public static void main(String[] args) {
        List<String> list = new ArrayList<String>();
        String str = "hi";
        list.add("string");
        list.add(str);
        list.add(str + str);
        System.out.println(list.size());
        System.out.println(list.contains(42));
        System.out.println(list.contains("hihi"));
        list.remove("hi");
        System.out.println(list.size());
      }
    }
```

```
output :
3
false
true
2
```

6.6: Implementing Classes

Demo : ArrayList Class

- **Execute the ArrayListDemo.java program**



IGATE
Speed.Agility.Imagination

6.6: Implementing Classes

HashSet Class

- **HashSet Class does not allow duplicates.**
- **A HashSet is an unsorted, unordered Set.**
- **It can be used when you want a collection with no duplicates and you do not care about the order when you iterate through it.**

June 11, 2015 | Proprietary and Confidential | - 16 -

IGATE Speed.Agility.Imagination

HashSet Class:
Remember that Sets are used when you do not want any duplicates in your collection. If you attempt to add an element to a set that already exists in the set, then the duplicate element will not be added, and the add() method will return false.
Remember, HashSets tend to be very fast because they use hashcodes.

```
import java.util.*;
    class SetTest {
      public static void main(String[] args) {
        boolean[] boolArr = new boolean[5];
        Set<Integer> set = new HashSet<Integer>();
        boolArr[0] = set.add(1);
        boolArr[1] = set.add(2);
        boolArr[2] = set.add(3);
        boolArr[3] = set.add(4);
        boolArr[4] = set.add(5);
        for (Integer index : set)
            System.out.print(index + " ");
      }
    }
```

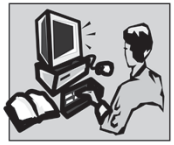
O/P: 2 4 1 3 5

Note: The order of the objects printed are not predictable

6.6: Implementing Classes

Demo : HashSet Class

- **Execute the HashSetDemo.java program**



June 11, 2015 | Proprietary and Confidential | - 17 -

IGATE Speed.Agility.Imagination

```
import java.util.*;
class HashSetDemo {
  public static void main(String args[]) {
  // create a hash set
     HashSet hs = new HashSet();
     // add elements to the hash set
        hs.add("B");
        hs.add("A");
        hs.add("D");
        hs.add("E");
        hs.add("C");
        hs.add("F");
       System.out.println(hs);
     }
}
```

Output :
[D, A, F, C, B, E]

TreeSet class

- **TreeSet does not allow duplicates.**
- **It iterates in sorted order.**
- **Sorted Collection:**
  - By default elements will be in ascending order.
- **Not synchronized:**
  - If more than one thread wants to access it at the same time, then it must be synchronized externally.

June 11, 2015 | Proprietary and Confidential | - 18 -

IGATE Speed.Agility.Imagination

TreeSet:
TreeSet implements the Set interface, backed by a TreeMap instance. This class guarantees that the sorted set will be in ascending element order, sorted according to the natural order of the elements, or by the comparator provided at set creation time, depending on which constructor is used.

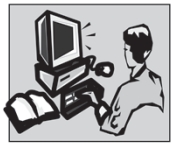
```
class TreeSetDemo {
 public static void main(String args[]) {
   TreeSet<String> treeSet = new TreeSet<String>();
   treeSet.add("One");
   treeSet.add("Two");
   treeSet.add("Three");
   treeSet.add("Four");
   treeSet.add("Five");
       System.out.println("Contents of treeset");
   Iterator iterator = treeSet.iterator(); // obtaining iterator object
   while (iterator.hasNext()) { // to iterate thru collection.
      Object object = iterator.next();
       System.out.print(object + "\t");
   }
 }}
```

O/P: Five Four One Three Two

6.6: Implementing Classes

Demo : TreeSet class

- **Execute the Treeset.java program**



IGATE
Speed.Agility.Imagination

You can also refer to the

6.6: Implementing Classes

Comparator Interface

- **The** java.util.Comparator **interface can be used to sort the elements of an Array or a list in the required way.**
- **It gives you the capability to sort a given collection in any number of different ways.**
- **Methods defined in Comparator Interface are as follows:**
  - int compare(Object o1, Object o2)
    - It returns true if the iteration has more elements.
  - boolean equals(Object obj)
    - It checks whether an object equals the invoking comparator.

June 11, 2015 | Proprietary and Confidential | - 20 - 20

IGATE Speed.Agility.Imagination

Comparator Interface:
The Comparator interface defines two methods: compare( ) and equals( ).
The compare( ) method, shown here, compares two elements for order:

int compare(Object obj1, Object obj2)

obj1 and obj2 are the objects to be compared. This method returns zero if the objects are equal. It returns a positive value if obj1 is greater than obj2. Otherwise, a negative value is returned. The method can throw a ClassCastException if the types of the objects are not compatible for comparison.
By overriding compare( ), you can alter the way that objects are ordered. For example, to sort in reverse order, you can create a comparator that reverses the outcome of a comparison.

The equals( ) method, shown here, tests whether an object equals the invoking comparator:

boolean equals(Object obj)

obj is the object to be tested for equality. The method returns true if obj and the invoking object are both Comparator objects and use the same ordering. Otherwise, it returns false. Overriding equals( ) is unnecessary, and most simple comparators will not do so.

6.6: Implementing Classes

Comparable Interface

- **Java.util.Comparable interface imposes a total ordering on the objects of each class that implements it.**
- **This ordering is referred to as the class's *natural ordering*, and the class's compareTo method is referred to as its *natural comparison method*.**
- **Methods defined in Comparable Interface are as follows:**
  - public int compareTo(Object o)
  - Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

June 11, 2015 | Proprietary and Confidential | - 21 - 21

IGATE Speed.Agility.Imagination

Comparable Interface:
This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's natural ordering, and the class's compareTo method is referred to as its natural comparison method.

Lists (and arrays) of objects that implement this interface can be sorted automatically by Collections.sort (and Arrays.sort). Objects that implement this interface can be used as keys in a sorted map or elements in a sorted set, without the need to specify a comparator.

public int compareTo(Object o)

Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

6.6: Implementing Classes

Comparable Interface Example

```
class Emp implements Comparable {
   int empID;
   String empName;
   double empSal;
    public Emp(String ename, double sal) { ...  }
    public String toString() {  ...  }

     public int compareTo(Object o) {
        if (this.empSal == ((Emp) o).empSal)    return 0;
         else if (this.empSal > ((Emp) o).empSal)   return 1;
            else   return -1;
   }}
```

6.6: Implementing Classes

Comparable Interface Example (ctnd…)

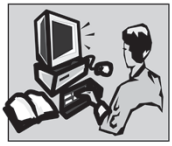
```
class Comparable Demo {
    public static void main(String[] args) {
      TreeSet tset = new TreeSet();
      tset.add(new Emp("harry", 40000.00));
      tset.add(new Emp("Mary", 20000.00));
      tset.add(new Emp("Peter", 50000.00));

      Iterator iterator = tset.iterator();
      while (iterator.hasNext()) {
        Object empObj = iterator.next();
        System.out.println(empObj + "\n");
      }
}}
```

Output:

```
Ename : Mary   Sal : 20000.0
Ename : harry  Sal : 40000.0
Ename : Peter  Sal : 50000.0
```

Demo : Concept of Comparator & Comparable Interface

- **Execute:**
  - ComparatorExample.java
  - ComparableDemo.java



IGATE
Speed.Agility.Imagination

6.6: Implementing Classes

HashMap Class

- HashMap uses the hashcode value of an object to determine how the object should be stored in the collection.
- Hashcode is used again to help locate the object in the collection.
- HashMap gives you an unsorted and unordered Map.
- It allows one null key and multiple null values in a collection.

June 11, 2015 | Proprietary and Confidential | - 25 -

IGATE Speed.Agility.Imagination

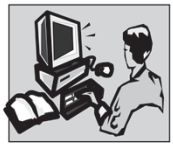
HashMap Class:
Map is an object that stores key/value pairs. Given a key, you can find its value. Keys must be unique, values may be duplicated. The HashMap class implements the map interface. The HashMap class uses a hash table to implement Map interface.
The following example maps names to account balances.

```
import java.util.*;
class HashMapDemo {
public static void main(String args[]) {
HashMap<String,Double> hm = new HashMap<String,Double>();
hm.put("John Doe", new Double(3434.34));
hm.put("Tom Smith", new Double(123.22));
hm.put("Jane Baker", new Double(1378.00));
hm.put("Tod Hall", new Double(99.22));
hm.put("Ralph Smith", new Double(-19.08));
Set set = hm.entrySet();    // Get a set of the entries
Iterator i = set.iterator();     // Get an iterator
    while(i.hasNext()) {          // Display elements
        Map.Entry me = (Map.Entry)i.next();
        System.out.println(me.getKey() + ": "+ me.getValue());
    }
// Deposit 1000 into John Doe's account
double balance =  ((Double)hm.get("John Doe")).doubleValue();
hm.put("John Doe", new Double(balance + 1000));
System.out.println("John Doe's new balance: " + hm.get("John Doe")); } }
```

6.6: Implementing Classes

Demo : HashMap Class

- **Execute the HashMapDemo.java program**



June 11, 2015 | Proprietary and Confidential | - 26 -

IGATE Speed.Agility.Imagination

The example is provided on previous page.
The output of the program is as follows:
Ralph Smith: -19.08
Tom Smith: 123.22
John Doe: 3434.34
Tod Hall: 99.22
Jane Baker: 1378.0
John Doe's new balance: 4434.34
The above program first populates the HashMap object. Then the contents of the map are displayed using a set-view, obtained by calling entrySet(). The keys and values are displayed by calling getKey() and getValue() methods of the Map.Entry interface.

Note: TreeMap instead of HashMap will have given a sorted output.

6.7: The Legacy Classes and Interfaces

Vector Class

- **The** java.util.Vector **class implements a growable array of Objects.**
- **It is same as ArrayList. However, Vector methods are synchronized for thread safety.**
- **New java.util.Vector is implemented from List Interface.**
- **Creation of a Vector:**
  - Vector v1 = new Vector(); // allows old or new methods
  - List v2 = new Vector(); // allows only the new (List) methods.

June 11, 2015 | Proprietary and Confidential | - 27 -

IGATE Speed.Agility.Imagination

Vector Class:
Vectors (the java.util.Vector class) are commonly used instead of arrays. This is because they expand automatically when new data is added to them. The Java 2 Collections API introduced a similar ArrayList data structure.
ArrayLists are unsynchronized and therefore faster than Vectors. However, they are less secure in a multithreaded environment. The Vector class was changed in Java 2 to add the additional methods supported by ArrayList. The description below is for the (new) Vector class.
Vectors can hold only Objects and not primitive types (for example: int). If you want to put a primitive type in a Vector, put it inside an object (for example: to save an integer value use the Integer class or define your own class). If you use the Integer wrapper, you will not be able to change the integer value, so it is sometimes useful to define your own class.
Constructor summary:
Vector()
Constructs an empty vector so that its internal data array has size 10 and its standard capacity increment is zero.
Vector(Collection<? extends E> c)
Constructs a vector containing the elements of the specified collection, in the order they are returned by the collection's iterator.
Vector(int initialCapacity)
Constructs an empty vector with the specified initial capacity and with its capacity increment equal to zero.
Vector(int initialCapacity, int capacityIncrement)
Constructs an empty vector with the specified initial capacity and capacity increment.

6.7: The Legacy Classes and Interfaces

Hashtable Class

- **It is a part of** java.util package**.**
- **It implements a hashtable, which maps keys to values.**
  - Any non-null object can be used as a key or as a value.
  - The Objects used as keys must implement the **hashcode** and the **equals method**.
- **Synchronized class**

June 11, 2015 | Proprietary and Confidential | - 28 -

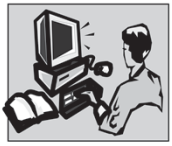
IGATE Speed.Agility.Imagination

Hashtable Class:
The Hashtable was a part of the original java.util package.
Hashtable is synchronized, and stores a key/value pair using the hashing technique.
While using a Hashtable, you specify an object that is used as a key, and the value that you want linked to that key. The key is then hashed. Subsequently, the resulting hash code is used as the index at which the value is stored within the table. The Hashtable class only stores objects that override the hashCode() and equals() methods that are defined by Object.

6.7: The Legacy Classes and Interfaces

Demo : Hashtable Class

- **Execute the HashTableDemo.java program**



June 11, 2015 | Proprietary and Confidential | - 29 -

IGATE Speed.Agility.Imagination

```
import java.util.*;
class HashTableDemo {
  public static void main(String args[]) {
      Hashtable<String,Double> balance = new
                                    Hashtable<String,Double>();
      Enumeration names;
      String str;
      double bal;
      balance.put("Arun", new Double(3434.34));
      balance.put("Radha", new Double(123.22));
      balance.put("Ram", new Double(99.22));
      // Show all balances in hash table.
     names = balance.keys();
      while(names.hasMoreElements()) {
          str = (String) names.nextElement();
          System.out.println(str + ": " +
          balance.get(str));
      }
    // Deposit 1,000 into Zara's account
        bal = ((Double)balance.get("Ram")).doubleValue();
        balance.put("Ram", new Double(bal+1000));
        System.out.println("Ram's new balance: " +
        balance.get("Ram"));
  }
}
```

Lab

- **Lab 7: Arrays and Collections**

6.8: Common Best Practices on Collections

Best Practices

- **Let us discuss some of the best practices on Collections:**
  - Use for-each liberally.
  - Presize collection objects.
  - Note that Vector and HashTable is costly.
  - Note that LinkedList is the worst performer.

June 11, 2015 | Proprietary and Confidential | - 31 -

IGATE Speed.Agility.Imagination

Common Best Practices on Collections:
Use for-each liberally : When there is a choice, the for-each loop should be preferred over the for loop, since it increases legibility.
Presize collection objects.

This is necessary because whenever the collection size has reached the maximum, internally whole array is copied to a new array with new increased size. This takes considerable time.

Try to presize any collection object to be as big as it will need to be. It is better for the object to be slightly bigger than necessary than to be smaller. This recommendation really applies to collections that implement size increases in such a way that objects are discarded.

For example: Vector grows by creating a new larger internal array object, copying all the elements from and discarding the old array. Most collection implementations work similarly, so presizing a collection to its largest potential size reduces the number of objects discarded.

Vector and HashTable is costly.

Usage of vector is very costly especially in code which heavily uses Vector to store lots of elements. Avoid using that if the elements in it are of same type. This is because elements are stored as Object so while accessing them one has to cast them into relevant classes which is very costly. Use ArrayList instead.

HashTable has the same reason as in the case of Vector. Moreover, the problem is compounded because of the use of Key and Value. Use HashMap class instead.

Never use linked List while accessing the objects : Sequentially access the elements.

Best Practices

- Choose the right Collection.
- Note that adding objects at the beginning of the collections is considerably slower than adding at the end.
- Encapsulate collections.
- Use thread safe collections when needed.

June 11, 2015 | Proprietary and Confidential | - 32 -

IGATE Speed.Agility.Imagination

Common Best Practices on Collections:

Choosing the right Collection:

We can select the appropriate collection based on the different implementation of the collection interfaces. As we know, there are different collection classes available such as ArrayList, LinkedList, HashSet, TreeSet, HashMap, TreeMap, and so on.

Principal features of non-primary implementations:

HashMap has slightly better performance than LinkedHashMap. However, its iteration order is undefined.

HashSet has slightly better performance than LinkedHashSet. However its iteration order is undefined.

TreeSet is ordered and sorted, but slow.

TreeMap is ordered and sorted, but slow.

LinkedList has fast adding to the start of the list, and fast deletion from the interior via iteration.

Common Best Practices on Collections (contd.):

- **Iteration order for above implementations:**
  - HashSet - *undefined*
  - HashMap - *undefined*
  - LinkedHashSet - insertion order
  - LinkedHashMap - insertion order of keys (by default), or "access order"
  - ArrayList - insertion order
  - LinkedList - insertion order
  - TreeSet - ascending order, according to Comparable / Comparator
  - TreeMap - ascending order of keys, according to Comparable / Comparator
- For LinkedHashSet and LinkedHashMap, the re-insertion of an item does not affect insertion order.
- For LinkedHashMap, "access order" is from the least recent access to the most recent access. In this context, only calls to **get**, **put**, and **putAll** constitute an access, and only calls to these methods affect access order.
- While being used in a Map or Set, these items must not change state (hence, it is recommended that these items be immutable objects):
  - keys of a Map
  - items in a Set
- **Adding objects at the beginning of the collections is considerably slower than adding at the end**
  - **For example:** Adding 50000 objects (strings) in ArrayList at the end takes around 195 millis and adding at the beginning of ArrayList takes over 5000 millis.
  - JDK1.3 gives the best and very optimized performance when it comes to collections. Similarly removing objects from end takes less time than removing from start.
- **Encapsulate collections:**
  - In general, Collections are not immutable objects. As such, one must often exercise care that collection fields are not unintentionally exposed to the caller.
  - One technique is to define a set of related methods which prevent the caller from directly using the underlying collection, such as :
  - addThing(Thing)
  - removeThing(Thing)
  - getThings() - return an unmodifiable Collection
- **Use thread safe collections when needed:**
  - Collection classes which have the synchronized methods are always thread safe. Next slide gives the list of the synchronized and non-synchronized collections.

Summary

- The various Collection classes and Interfaces
- Generics
- Best practices in Collections

Review Questions

- **Question 1:** Consider the following code:

```
TreeSet map = new TreeSet();
  map.add("one");
  map.add("two");
  map.add("three");
  map.add("one");
  map.add("four");
  Iterator it = map.iterator();
  while (it.hasNext() )
      System.out.print( it.next() + " " );
```

  - **Option 1:** Compilation fails
  - **Option 2:** four three two one
  - **Option 3:** one two three four
  - **Option 4:** four one three two

Review Questions

- **Question 2:** Which of the following statements are true for the given code?

```
public static void before() {
  Set set = new TreeSet();
  set.add("2");
  set.add(3);
  set.add("1");
  Iterator it = set.iterator();
   while (it.hasNext())
       System.out.print(it.next() + " ");
}
```

  - **Option 1:** The before() method will print 1 2
  - **Option 2:** The before() method will print 1 2 3
  - **Option 3:** The before() method will not compile.
  - **Option 4:** The before() method will throw an exception at runtime.

Core Java 8 with JAXB and Development Tools

Generics

IGATE
Speed.Agility.Imagination

Lesson Objectives

- **After completing this lesson, participants will be able to**
  - Understand concept of Generics
  - Implement generic based collections

June 11, 2015 | Proprietary and Confidential | - 2 -

IGATE
Speed.Agility.Imagination

This lesson discusses about generics feature in Java.

Lesson outline:

12.1: Generics
12.2: Writing Generic Classes
12.3: Using Generics with Collections
12.4: Best Practices

12.1: Introduction to Generics

Generics

- **Generics is a mechanism by which a single piece of code can manipulate many different data types without explicitly having a separate entity for each data type.**

June 11, 2015 | Proprietary and Confidential | - 3 -

IGATE Speed.Agility.Imagination

What and Why of Generics:
JDK 1.5 introduces several extensions to the Java programming language. One of these is generics. Generics allow you to abstract over types. The most common examples are container types, such as those in the Collection hierarchy.
Here is a typical usage of that sort:

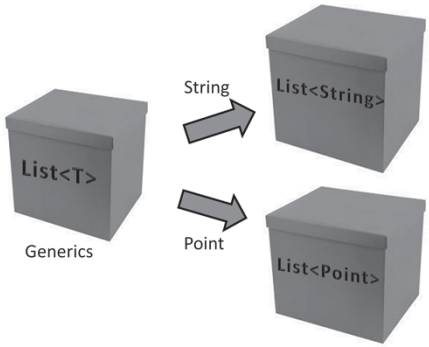
```
List myIntegerList = new LinkedList(); // 1
myIntegerList.add(new Integer(0)); // 2
Integer intObj = (Integer) myIntegerList.iterator().next(); // 3
```

The cast on line 3 is slightly annoying. Typically, the programmer knows what kind of data has been placed into a particular list. However, the cast is essential. The compiler can only guarantee that an Object will be returned by the iterator. To ensure the assignment to a variable of type Integer is type safe, the cast is required. Of course, the cast not only introduces clutter, it also introduces the possibility of a run time error, since the programmer might be mistaken. 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.

12.1: Introduction to Generics

Generics

- Generics allows programmer to create parameterized types
- Instances of such types can be created by passing reference types



What is Generics?:

Generics allows to write Parameterized type like List<T> and allows us to pass references to create instance of that type. Like when we pass the reference of String to List<T> as a reference type it creates List of strings.

Why generics?

Use of generics enables stricter compiler check. It means when List is created of type string, compiler only allows to add string elements to the list.

No need to perform casting. In case of generics enabled collections, no need to perform explicit casting.

12.2: Writing Generic Classes

Generics Fundamentals

- **Consider the class given to send the message of type String**
- **Can we reuse the same class to send message of type Employee?**

```
public class Sender{
   private String message;
   public setMessage(String message) {
        this.message = message;
   }
   public sendMessage() {
        //logic to send message
    }
}
```

June 11, 2015 | Proprietary and Confidential | - 5 -

IGATE Speed.Agility.Imagination

Generics Fundamentals:

Consider the example given in the slide, if you want to reuse the same class to send employee object as a message, then we need to create one more additional class as shown below:

```
public class Sender{
   private Employee message;
   public setMessage(Employee message) {
        this.message = message;
   }
   public sendMessage() {
            //logic to send message
    }
}
```

Is it possible to create a class with generic type of message? Yes. Generics enables us to create a parameterized class and later we can instantiate it as per our required reference type.

12.2: Writing Generic Classes

Writing Generic Types

- How to create a sender class to send generic type of message?

```
public class Sender<T>{
   private T message;
   public setMessage(T message) {
       this.message = message;
   }
   public sendMessage() {
       //logic to send message
   }
}
```

```
Sender<String> stringSender = new Sender<String>();
Sender<Employee> empSender = new Sender<Employee>();
```

June 11, 2015 | Proprietary and Confidential | - 6 -

IGATE Speed.Agility.Imagination

Generics Fundamentals:

As shown in the slide example, The sender class is declared as parameterized generic type of one parameter as "T". The "T" in diamond operator refers as generic type of message.

In case of String message sender, the class instance would be initialized as:

```
Sender<String> stringSender = new Sender<String>() ;
```

In the above instance creation, the type T is replaced by String reference type. The same generic sender can be used to send message of type employee as shown below:

```
Sender<Employee> stringSender = new Sender<Employee>() ;
```

12.2: Writing Generic Classes

Generics Terminology

- **Below listed are different conventions used in generics**

Syntax	Meaning
<T>	T denotes instance of any reference type
<?>	? denotes object of any type
<? super T>	? denotes lower bound object of type T
<? extends T>	? denotes upper bound object of type T (class)
<K, V>	K and V denotes instance of any type (same as T)

June 11, 2015 | Proprietary and Confidential | - 7 -

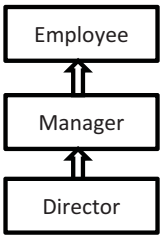
IGATE Speed.Agility.Imagination

Generics Terminology:

In generics, different conventions are used to indicate the applicable reference type. For example, class Sender<T> indicates, the allowed reference type to create instance of Sender are:
Any reference type T
Subclass of T

The wildcard ? is used to indicate any type. There are two variations in using wildcard.
? super T: indicates lower bound meaning, any reference types which are superclass of T are allowed.
? extends T: indicated upper bound meaning, any reference types which are subclass of T are allowed.

Consider the given example for inheritance relationship.



List<? super Manager> means, list can be created of Manager, Employee etc. That is all superclass's of Manager.

List<? extends Manager> means, list can be created of Manager, Director etc. That is all subclasses of Manager.

12.3: Using Generics With Collections

Using Generics with Collections

- Before Generics:

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

- After Generics:

Note: Line no 3 if not properly typecasted will throw runtime exception

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


IGATE Speed.Agility.Imagination

What and Why of Generics:
Observe the program fragment given above (in box #2) using generics:
Notice the type declaration for the variable myIntegerList. It specifies that this is not just an arbitrary List, but a List of Integer, written as List<Integer>. We say that List is a generic interface that takes a type parameter - in this case, Integer. We also specify a type parameter while creating the list object. Notice that the cast is gone from line 3. It may seem that all that's accomplished is just moving the clutter around. Instead of a cast to Integer on line 3, we have Integer as a type parameter on line 1.
However, there is a very big difference here. The compiler can now check the type correctness of the program at compile-time. When we say that myIntegerList is declared with type List<Integer>, this tells us something about the variable myIntegerList, which holds true wherever and whenever it is used, and the compiler will guarantee it. 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.

12.3: Using Generics with Collections

What problems does Generics solve?

- **Problem: Collection element types:**
  - Compiler is unable to verify types.
  - Assignment must have type casting.
  - ClassCastException can occur during runtime.
- **Solution: Generics**
  - Tell the compiler type of the collection.
  - Let the compiler fill in the cast.
    - **Example:** Compiler will check if you are adding Integer type entry to a String type collection (compile time detection of type mismatch).

What and Why of Generics:
What problems does Generics solve?
Type cast not being checked at compile-time leads to a major problem occurring at application's runtime.
Biggest achievement of the Generics is to avoid the runtime exceptions.

12.3: Using Generics with Collections

Using Generic Classes: 1

- **You can instantiate a generic class to create type specific object.**
- **In J2SE 5.0, all collection classes are rewritten to be generic classes.**
  - Example:

```
Vector<String> vector = new Vector<String>();
vector.add(new Integer(5)); // Compile error!
vector.add(new String("hello"));
String string = vector.get(0); // No casting needed
```

June 11, 2015 | Proprietary and Confidential | - 10 - 10

IGATE Speed.Agility.Imagination

Usage of Generics:
Usage of Generic classes is pertaining to the type that is required.
Once the Generic class is used for specific type, compile-time and runtime errors can be avoided.

12.3: Using Generics with Collections

Using Generic Classes: 2

- **Generic class can have multiple type parameters.**
- **Type argument can be a custom type.**
  - Example:

```
HashMap<String, Mammal> map =
                    new HashMap<String, Mammal>();
map.put("wombat", new Mammal("wombat"));
Mammal mammal = map.get("wombat");
```

June 11, 2015 | Proprietary and Confidential | - 11 -

11

IGATE Speed.Agility.Imagination

Usage of Generics:
Generic classes in use can have multiple arguments. Arguments can be standard as well as custom types.

12.3: Using Generics with Collections

Generics

- **Using generics, you can do this:**
  Object object = new Integer(5);
- **You can even do this:**
  Object[] objArr = new Integer[5];
- **So you would expect to be able to do this: *ArrayList<Object> arraylist = new ArrayList<Integer>();***
  But you can't do it!!
  - This is counter-intuitive at the first glance.

IGATE Speed.Agility.Imagination

Note: Generic classes cannot be assigned according to the super or subclass hierarchy of them.

12.3: Using Generics with Collections

Generics

- **Why does this compile error occur?**
  - It is because if it is allowed, ClassCastException can occur during runtime – this is not type-safe.

```
ArrayList<Integer> ai = new ArrayList<Integer>();
ArrayList<Object> ao = ai;   // If it is allowed at compile time,
ao.add(new Object());Integer i = ao.get(0);   // will result in runtime
 ClassCastException
```

- **There is no inheritance relationship between type arguments of a generic class.**

June 11, 2015 | Proprietary and Confidential | - 13 - 13

IGATE Speed.Agility.Imagination

Note: Generic classes do not support inheritance relationship between type arguments.

12.3: Using Generics with Collections

Generics

- **The following code works:**

```
ArrayList<Integer> ai = new ArrayList<Integer>();
List<Integer> li = new ArrayList<Integer>();
Collection<Integer> ci = new ArrayList<Integer>();
Collection<String> cs = new Vector<String>(4);
```

- **Inheritance relationship between Generic classes themselves still exists.**

June 11, 2015 | Proprietary and Confidential | - 14 - 14

IGATE Speed.Agility.Imagination

Note: Although the inheritance relationship between the type arguments of the generic classes does not exist, Inheritance relationship between Generic classes themselves still exist.

12.3: Using Generics with Collections

Generics

- **The following code works:**

```
ArrayList<Number> an = new ArrayList<Number>();
an.add(new Integer(5));
an.add(new Long(1000L));
an.add(new String("hello")); // compile error
```

- **The entries maintain inheritance relationship.**

Summary

- Generics
- Best practices in Generics

Review Questions

- **Question 1: If a method created to accept argument of List<Object>, then which of the following are valid options to pass? Ex: void printList(List<Object> list)**
  - Option1: List<Object>
  - Option2: List<Integer>
  - Option3: List<Float>
  - Option 4: All of the above

Core Java 8 with JAXB and Development Tools

File IO

IGATE
Speed.Agility.Imagination

Lesson Objectives

- **After completing this lesson, participants will be able to**
  - Understand concept of Java I/O API
  - Implements byte and character streams to perform I/O
  - Work with utility classes like File and Path

June 11, 2015 | Proprietary and Confidential | - 2 -

IGATE Speed.Agility.Imagination

This lesson covers the Java platform classes used for basic I/O. It focuses primarily on I/O Streams, a powerful concept that greatly simplifies I/O operations. The lesson also looks at serialization, which lets a program write whole objects out to streams and read them back again. Most of the classes covered are in the java.io package.

Lesson outline:

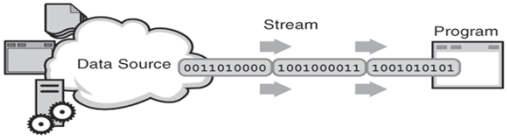
13.1: Overview of I/O Streams
13.2: Types of Streams
13.3: The Byte-stream I/O hierarchy
13.4: Character Stream Hierarchy
13.5: Buffered Stream
13.6: The File class
13.7: Exploring NIO
13.8: Object Stream
13.9: Best Practices

13.1 : Overview of I/O Streams

Overview

- **Most programs need to access external data.**
- **Data is retrieved from an input source. Program results are sent to output destination.**

Figure 7-1: A program uses an input stream to read data from a source, one item at a time



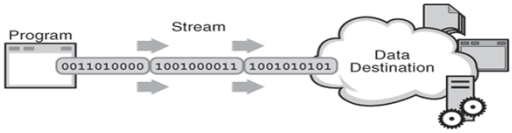


Figure 7-2:A program uses an output stream to write data to a destination, one item at time

June 11, 2015 | Proprietary and Confidential | - 3 -

IGATE Speed.Agility.Imagination

Most programs need to use data. To read some data, a Java program opens a stream to a data source, such as a file or remote socket, and reads the information serially. To write some data, a program opens a stream to a data source and writes to it in a serial fashion.

Whether you are reading from a file or from a socket, the concept of serially reading from, and writing to different data sources is the same.

The java.io package provides an extensive library of classes dealing with input and output. Each class has a variety of member variables & methods. java.io is layered. i.e. it does not attempt to put too much capability into one class. Instead, you can get the features you want, by layering (chaining streams) one class over another.

13.1: Overview of I/O Streams

What is a Stream?

- **Stream:**
  - Abstraction that consumes or produces information.
  - Linked to source and destination.
  - Implemented within class hierarchies defined in java.io package.
  - An input stream acts as a source of data.
  - An output stream acts as a destination of data.

Figure 7-3: (a) Input Stream

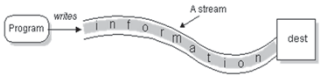


Figure 7-3:(b) Output stream

June 11, 2015 | Proprietary and Confidential | - 4 -

IGATE Speed.Agility.Imagination

The source and destination of data for a java program can be anything – a network connection, local files, memory buffer etc. All are handled in the same way using streams. Streams implement sequential access of data. Java implements streams within class hierarchies defined in the java.io package.

An input stream is an object that an application can use, to read a sequence of data. An output stream is an object that an application can use to write a sequence of data.

An input stream acts as a source of data, and an output stream acts as a destination of data.

Some streams simply pass on data; others manipulate and transform the data in useful ways.

13.2: Types of Streams

Different Types of I/O Streams

- **Byte Streams: Handle I/O of raw binary data.**
- **Character Streams: Handle I/O of character data. Automatic translation handling to and from a local character.**
- **Buffered Streams: Optimize input and output with reduced number of calls to the native API.**
- **Data Streams: Handle binary I/O of primitive data type and String values.**
- **Object Streams: Handle binary I/O of objects.**
- **Scanning and Formatting: Allows a program to read and write formatted text.**

June 11, 2015 | Proprietary and Confidential | - 5 -

IGATE Speed.Agility.Imagination

There are different types of I/O (Input/Output) Streams:

Byte Streams: They provide a convenient means for handling input and output of bytes. Programs use byte streams to perform input and output of 8-bit bytes. All byte stream classes descend from InputStream and OutputStream class.

Character streams: They provide a convenient means for handling input and output of characters. They use Unicode and, therefore, can be internationalized.

Buffered Streams: Buffered input streams read data from a memory area known as a buffer; the native input API is called only when the buffer is empty. Similarly, buffered output streams write data to a buffer, and the native output API is called only when the buffer is full.

Data Streams: Data streams support binary I/O of primitive data type values (boolean, char, byte, short, int, long, float, and double) as well as String values.

Object Streams: Just as data streams support I/O of primitive data types, object streams support I/O of objects.

Scanning and Formatting: It allows a program to read and write formatted text.

13.3: Byte Stream I/O Hierarchy

Byte Stream I/O Hierarchy

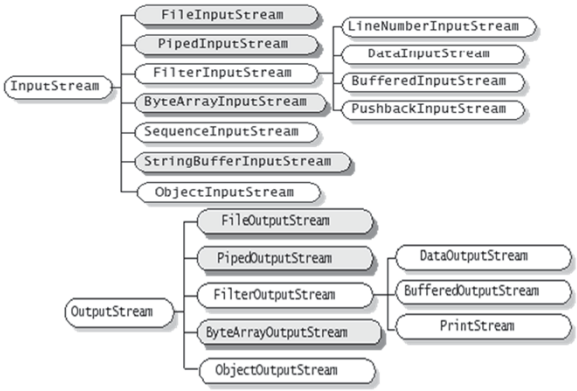


Figure 7-4:Byte-stream I/O hierarchy

June 11, 2015 | Proprietary and Confidential | - 6 -

IGATE Speed.Agility.Imagination

At the top of the hierarchy are two abstract classes: InputStream and OutputStream.

Each of these abstract classes serves as base class for all other concretely implemented I/O classes. Each of the abstract classes defines several key methods that the other stream classes implement.

13.3: Byte Stream I/O Hierarchy

Methods of InputStream Class

Method	Description
close()	Closes this input stream and releases any system resources associated with the stream.
int read()	Reads the next byte of data from the input stream.
int read(byte[] b)	Reads some number of bytes from the input stream and stores them into the buffer array *b*.
int read(byte[] b, int off, int len)	Reads up to *len* bytes of data from the input stream into an array of bytes.

Table 7-1: Methods of class InputStream

IGATE Speed.Agility.Imagination

All byte stream classes are descended from InputStream and OutputStream.

Note: Refer to Java documentation for more methods.

13.3: Byte Stream I/O Hierarchy

Methods of OutputStream Class

Method	Description
close()	Closes this output stream and releases any system resources associated with this stream.
flush()	Flushes this output stream and forces any buffered output bytes to be written out.
write(byte[] b)	Writes *b.length* bytes from the specified byte array to this output stream.
write(byte[] b, int off, int len)	Writes *len* bytes from the specified byte array starting at offset off to this output stream.
write(int b)	Writes the specified byte to this output stream.

Table 7-2: Methods of class OutputStream

IGATE Speed.Agility.Imagination

Closing an output stream automatically flushes the stream, meaning that data in its internal buffer is written out. An output stream can also be manually flushed by calling the flush() method.

Note: Refer to Java documentation for more methods.

13.3: Byte Stream I/O Hierarchy

InputStream Subclasses

Classname	Description
DataInputStream	A filter that allows the binary representation of java primitive values to be read from an underlying inputstream
BufferedInputStream	A filter that buffers the bytes read from an underlying input stream. The buffer size can be specified optionally.
FilterInputStream	Superclass of all input stream filters. An input filter must be chained to an underlying inputstream.
ByteArrayInputStream	Data is read from a byte array that must be specified
FileInputStream	Data is read as bytes from a file. The file acting as the input stream can be specified by File object, or as a String
PushBackInputStream	A filter that allows bytes to be "unread " from an underlying stream. The number of bytes to be unread can be optionally specified.
ObjectInputStream	Allows binary representation of java objects and java primitives to be read from a specified inputstream.
PipedInputStream	It reads many bytes from PipedOutputStream to which it must be connected.
SequenceInputStream	Allows bytes to be read sequentially from two or more input streams consecutively.

IGATE
Speed.Agility.Imagination

The InputStream class allows several classes to derive from it. Some of these classes are described in the table above.
Note: All of the above mentioned classes have corresponding output stream classes except SequenceInputStream.

13.3: Byte Stream I/O Hierarchy

The predefined streams

- **The** java.lang.System **class encapsulates several aspects of the run-time environment.**
- **Contains three predefined stream variables: in, out & err.**
- **These fields are declared as public and static within System.**
  - *System.out* :refers to the standard output stream
  - *System.err* :refers to standard error stream
  - *System.in* : refers to standard input

June 11, 2015 | Proprietary and Confidential | - 10 -

IGATE Speed.Agility.Imagination

System.out refers to the standard output stream and System.err refers to standard error stream (Both, by default, the console). These are objects of type PrintStream class.
System.in refers to standard input (keyboard by default). This is an object of the InputStream class.

13.3: Byte Stream I/O Hierarchy

Example : Reading Console input

```
import java.io.*;
class ReadKeys {
 public static void main (String args[]) {
        StringBuffer sb = new StringBuffer();
        char c;
    System.out.println("Enter a String:");
        try {
              while((c =(char)System.in.read()) != '\n')
                 sb.append(c);
    }catch(Exception e){
         System.out.println("Error while reading" + e.getMessage()); }
 String s = new String(sb);
 System.out.println("You entered : " + s);     }}
```

June 11, 2015 | Proprietary and Confidential | - 11 -

IGATE Speed.Agility.Imagination

In Java, console input is accomplished by reading from System.in. In the above example, read() methods reads a byte from the input stream (here, the keyboard), and returns an integer. Therefore, casting to char type need to be done.

13.3: Byte Stream I/O Hierarchy

Example: FileInputStream & FileOutputStream

```
class CopyFile  {
      FileInputStream fromFile;  FileOutputStream toFile;
      public void init(String arg1, String arg2) {      //pass file names
            try{
              fromFile = new FileInputStream(arg1);
              toFile = new FileOutputStream(arg2);
            } catch (Exception fnfe) {… }
    }
    public void copyContents() {// copy bytes
      try {
            int i = fromFile.read();
            while ( i != -1) {                              //check the end of file
                  toFile.write(i);
                  i = fromFile.read(); }
      } catch (IOException ioe) { System.out.println("Exception: " + ioe);}
  }
```

June 11, 2015 | Proprietary and Confidential | - 12 -

IGATE Speed.Agility.Imagination

The remainder of the code follows:

```
      public void closeFiles() {   //close the file
          try{
                  fromFile.close();
                  toFile.close();
            } catch (IOException ioe){
                  System.out.println("Exception: " + ioe);
                  }
      }
      public static void main(String[] args){
                  CopyFile c1 = new CopyFile();
                  c1.init(args[0], args[1]);
                  c1.copyContents();
                  c1.closeFiles();
      } }
```

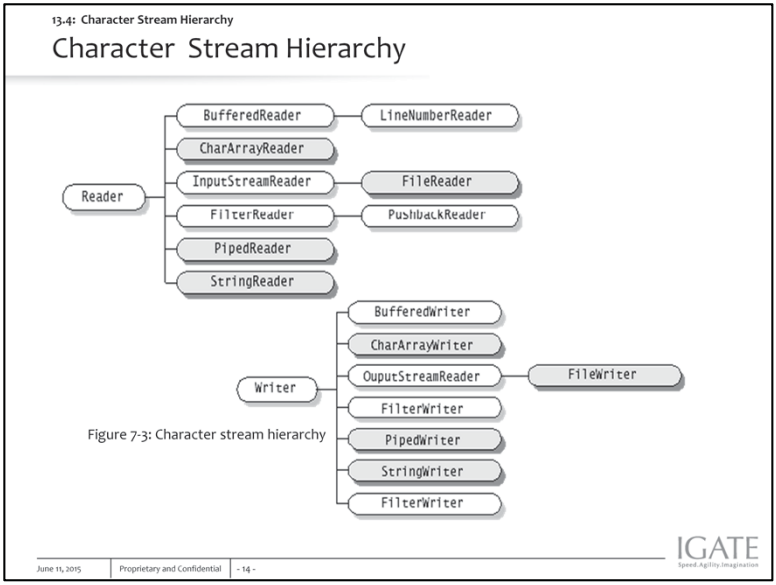
The FileInputStream and FileOutputStream classes define byte input and output streams that are connected to files. Data can only be read or written as a sequence of bytes. The above example demonstrates the use of File InputStream and FileOutputStream.

13.3: Byte Stream I/O Hierarchy

Demo : FileInput/OutputStream

- **Execute:**
  - ReadKeys.java
  - CopyFile.java program

IGATE
Speed.Agility.Imagination



The byte stream classes support only 8-bit byte streams and doesn't handle 16-bit Unicode characters well. A character encoding is a scheme for representing characters. Java represents characters internally in the 16-bit Unicode character encoding, but the host platform might use different character encoding.
The abstract classes Reader and Writer are the roots of the inheritance hierarchies for streams that read and write Unicode characters using a specific character encoding.
A reader is an input character stream that reads a sequence of Unicode characters, and a writer is an output character stream that writes a sequence of Unicode characters.

13.4: Character Stream Hierarchy

Reader Class Methods

Method	Description
int read() throws IOException	reads a byte and returns as an int
int read(char b[])throws IOException	reads into an array of chars b
int read(char b[], int off, int len) throws IOException	reads *len* number of characters into char array *b*, starting from offset *off*
long skip(long n) throws IOException	Can skip n characters.

Table 7-4: Reader Methods

Note: Refer to Java documentation for more methods.

13.4: Character Stream Hierarchy

Writer Class Methods

Method	Description
void write(int c)throws IOException	writes a byte.
void write(char b[])throws IOException	writes from an array of chars b
void write(char b[], int off, int len) throws IOException	writes len number of characters from char array b, starting from offset off
void write(String b, int off, int len) throws IOException	writes len number of characters from string b, starting from offset off

Table 7-5: Writer Methods

IGATE Speed.Agility.Imagination

Note: Refer to Java documentation for more methods.

13.4: Character Stream Hierarchy

Example: FileReader, FileWriter Classes

```
public class CopyCharacters {
   public static void main(String[] args) throws IOException {
 try(FileReader inputStream = new FileReader("sampleinput.txt");
      FileWriter outputStream = newFileWriter("sampleoutput.txt"))
{
          int c;
          while ((c = inputStream.read()) != -1) {
              outputStream.write(c);
          }
  } catch(IOException ex) {
              System.out.println(ex.getMessage());
  }
}}
```


IGATE Speed.Agility.Imagination

13.5: Buffered Stream

Buffered Input Output Stream

- **An unbuffered I/O means each read or write request is handled directly by the underlying OS.**
  - Makes a program less efficient.
    - Each such request often triggers disk access, network activity, or some other relatively expensive operation.
- **Java's buffered I/O Streams reduce this overhead.**
  - Buffered streams read/write data from a memory area known as a buffer; the native input API is called only when the buffer is empty.

IGATE
Speed.Agility.Imagination

13.5: Buffered Stream

Using buffered streams

- **A program can convert a unbuffered stream into buffered using the *wrapping idiom:***
  - Unbuffered stream object is passed to the constructor of a buffered stream class.
  - Example

```
inputStream = new BufferedReader(new FileReader("input.txt"));
outputStream = new BufferedWriter(new FileWriter("output.txt"));
```


IGATE Speed.Agility.Imagination

There are four buffered stream classes used to wrap unbuffered streams.

BufferedInputStream and BufferedOutputStream - create buffered byte streams.

BufferedReader and BufferedWriter - create buffered character streams.

Flushing Buffered Streams

It often makes sense to write out a buffer at critical points, without waiting for it to fill. This is known as flushing the buffer.

Some buffered output classes support autoflush, specified by an optional constructor argument. When autoflush is enabled, certain key events cause the buffer to be flushed. For example, an autoflush PrintWriter object flushes the buffer on every invocation of println or format.

To flush a stream manually, invoke its flush() method. The flush() method is valid on any output stream, but has no effect unless the stream is buffered.

13.5: Buffered Stream

Example of Buffered stream

```
class LineNumberReaderDemo{
    public static void main(String args[]) {
        String s;
        try(FileReader fr = new FileReader("names.txt");
        BufferedReader br = new BufferedReader(fr);
        LineNumberReader lr = new LineNumberReader(br);)  {
            while((s = lr.readLine()) != null)
                System.out.println(lr.getLineNumber()+" " +s);
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }
}}
```

Names.txt contains
Anita
Bindu
Cindy
Diana

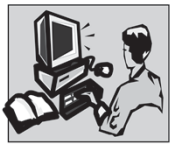
Output is:
1 Anita
2 Bindu
3 Cindy
4 Diana

IGATE Speed.Agility.Imagination

13.5: Buffered Stream

Demo : FileReader/FileWriter

- **Execute the**
  - LineNumberReaderDemo.java
  - CharEncode.java



IGATE
Speed.Agility.Imagination

13.6: File class

The File Class

- File class doesn't operate on streams
- Represents the pathname of a file or directory in the host file system
- Used to obtain or manipulate the information associated with a disk file, such as permissions, time, date, directory path etc
- An object of File class provides a handle to a file or directory and can be used to create, rename or delete the entry

June 11, 2015 | Proprietary and Confidential | - 22 -

IGATE Speed.Agility.Imagination

Support for File/Directory Operations are provided by java.io.File. This class makes it easier to write platform-independent code that examines and manipulates files.
Provides methods
To obtain basic information about the file/directory
To Create / Delete Files and Directories

13.6: File class

The File Class

- **Some methods**
  - canRead()
  - exists()
  - isFile()
  - isDirectory()
  - getAbsolutePath()
  - getName()
  - getPath()
  - getParent()
  - length() : returns length of file in bytes as long
  - lastModified()
  - mkdir()
  - list() : obtain listings of directory contents

IGATE
Speed.Agility.Imagination

The File Class

```
class FileDemo {
  String fname;
  public static void main(String args[]) {
    String fname = args[0];
    File f = new File(fname);
    System.out.println("File name : "+f.getName());
    System.out.println("Parent dir name : "+f.getParent());
    System.out.println("Absolute path name : "+f.getAbsolutePath());
    System.out.println("File modified last :
                        "+String.valueOf(f.lastModified()));
    System.out.println("File length : "+f.length());
    System.out.println("File Readable? : " + (f.canRead()? "true":"false"));
  } }
```


Output :
File name : books.xml
Parent directory name : null
Absolute path name : D:\G-drive contents\java-demo\day5(filesIO)\demo\file handling\books.xml
File modified last on : 0
File length : 0
File Readable? : false

13.7: Exploring NIO

Path Interface

- **Java 7 provides new improved features over traditional File class**
- **Files and directories in file system can be uniquely identified by Path**
- **A path can be absolute or relative**
- **Paths class can be used to create a path reference**

```
Path javaHome = Paths.get("C:/Program Files/Java/jdk1.8.0_25");
System.out.println(javaHome.getNameCount()); //3 (doesn't count root)
System.out.println(javaHome.getRoot()); // C:\
System.out.println(javaHome.getName(0));// Program Files
System.out.println(javaHome.getName(1)); // Java
System.out.println(javaHome.getFileName()); //jdk1.8.0_25
System.out.println(javaHome.getParent()); //C:\Program Files\Java
```

June 11, 2015 | Proprietary and Confidential | - 25 -

IGATE Speed.Agility.Imagination

Path interface introduced in java.nio.file package to support better file handling and to overcome few drawbacks of traditional File class.

Path instance is used as reference to File or Directories as either relative or absolute path. Paths class is used to create a non-existance reference to file or directory. It means, creating reference of Path doesn't create new file or directory.

Few methods of Path interface are shown in slide, the getNameCount() method is used to return count of path parts. Individual part of path can be retrieved by using getName(index); the index start from 0.

The getFileName() returns last part of path and getParent() returns parent path.

13.7: Exploring NIO

Files Class

- **Introduced in java.nio.file for better file and directory manipulation**
  - File/Directory creation and deletion
  - Perform different checks with File/Directory
  - Used to create streams objects

Method	Meaning
createFile	Used to create a file
createDirectory	Used to create a directory
delete	Used to delete the file/directory
deleteIfExists	Check before deleting file/directory
newDirectoryStream	Used to fetch directory contents
copy	Copies the file/directory
move	Moves the file/directory
readAllLines/readAllBytes	Used to read file in stream
write	Used to write in file

June 11, 2015 | Proprietary and Confidential | - 26 -

IGATE Speed.Agility.Imagination

Files class contains lots of static methods to perform manipulation on files and directories. It also helps to retrieve streams from file/directory for reading or writing. The code snippet shown below is used to list all contents of directory.

```
Path javaHome= Paths.get("C:/Program Files/Java/jdk1.8.0_25");
DirectoryStream<Path> contents = Files.newDirectoryStream(javaHome);
for(Path content: contents) {
        System.out.println(content.getFileName());
}
contents.close();
```

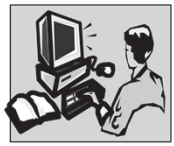
Below listed snippet shows how to read the contents of a textual file with ease.

```
Path file = Paths.get("D:/output.txt");
List<String> lines = Files.readAllLines(file);
for(String line:lines) {
    System.out.println(line);
}
System.out.println("End of File....");
```

13.7: Exploring NIO

Demo : Path and Files

- **Execute the**
  - PathDemo.java
  - ListingDirectory.java
  - ListingFile.java



IGATE
Speed.Agility.Imagination

13.8:Object Stream

ObjectInputStream, ObjectOutputStream

- **Object streams support I/O of objects:**
  - Support I/O of primitive data types.
  - Object has to be *Serializable* type.
  - *Object Classes:* ObjectInputStream, ObjectOutputStream
    - Implement ObjectInput and ObjectOutput, which are subinterfaces of DataInput and DataOutput.
  - An object stream can contain a mixture of primitive and object values.

June 11, 2015 | Proprietary and Confidential | - 28 -

IGATE Speed.Agility.Imagination

Only objects that support the java.io.Serializable or java.io.Externalizable interface can be read from streams.

The method readObject is used to read an object from the stream. Java's safe casting should be used to get the desired type. In Java, strings and arrays are objects and are treated as objects during serialization. When read they need to be cast to the expected type.

Primitive data types can be read from the stream using the appropriate method on DataInput.

13.8: Serialize Objects

Serializing Objects

- ***Object Serialization:***
  - Process to read and write objects.
  - Provides ability to read or write a whole object to and from a raw byte stream.
  - Use object serialization in the following ways:
    - *Remote Method Invocation (RMI):* Communication between objects via sockets.
    - *Lightweight persistence:* Archival of an object for use in a later invocation of the same program.

June 11, 2015 | Proprietary and Confidential | - 29 -

IGATE Speed.Agility.Imagination

Object Serialization allows an object to be transformed into a sequence of bytes that can be later re-created (deserialized) into an original object.
Java provides this facility through ObjectInput and ObjectOutput interfaces, which allow the reading and writing of objects from and to streams. These interfaces extend DataInput and DataOutput respectively
The concrete implementation of ObjectOutput and ObjectInput interfaces is provided in ObjectOutputStream and ObjectInputStream classes respectively.
These two interfaces have the following methods:
final void writeObject(Object obj) throws IOException.
final Object readObject() throws IOException, ClassNotFoundException
The writeObject() method can be used to write any object to a stream, including strings and arrays, as long as an object supports java.io.Serializable interface, which is a marker interface with no methods.
Serializing an object requires only that it meets one of two criteria. The class must either implement the Serializable interface (java.io.Serializable) which has no methods that you need to write or the class must implement the Externalizable interface which defines two methods. As long as you do not have any special requirements, making a serializable is as simple as adding the ``implements Serializable'' clause.

13.8: Serialize Objects

Example : Object Serialization

```
class Student implements Serializable{
   int roll;
   String sname;
   public Student(int r, String s){
       roll = r;
       sname = s;    }
    public String toString(){
                    return "Roll no is : "+roll+"   Name is : "+sname;
   } }
```

```
public class demo{
   public static void main(String args[]){
   try{ Student s1 = new Student (100,"Varsha");
       System.out.println("s1 object : "+s1);
```


IGATE Speed.Agility.Imagination

13.8: Serialize Objects

Example : Object Serialization (contd..)

```
FileOutputStream fos = new FileOutputStream("student");
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeObject(s1);
oos.flush();
oos.close();
 }  catch(Exception e){ }
 try{
Student s2;
FileInputStream fis = new FileInputStream("student");
ObjectInputStream ois = new ObjectInputStream(fis);
s2 = (Student)ois.readObject();
ois.close();
System.out.println("s2 object : "+s2);  }
catch(Exception e){ } }
```

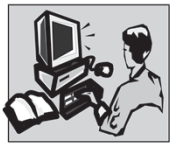

IGATE Speed.Agility.Imagination

Output :
s1 object : Roll no is : 100 Name is : Varsha
s2 object : Roll no is : 100 Name is : Varsha

13.8: Serialize Objects

Demo : Object Serialization

- **Execute the :**
  - Student.java and ObjectSerializationDemo.java
  - EmpObjectSerializationDemo.java



IGATE
Speed.Agility.Imagination

Lab : Files IO

- **Lab 8: Files IO**

13.9: Best Practices in I/O

Best Practices in I/O

- **Always close streams:**

```
try{
    file = new FileOutputStream( "emp.ser" );
    OutputStream buffer = new BufferedOutputStream( file);
    ObjectOutput output = new ObjectOutputStream( buffer );
     try{ output.writeObject(emp); }
    finally{ output.close(); } }
```

- **Use buffering when reading and writing text files.**
- **FileInputStream and DataInputStream are very slow.**

June 11, 2015 | Proprietary and Confidential | - 34 -

IGATE Speed.Agility.Imagination

Always close streams
Streams represent resources which you must always clean up explicitly, by calling the close method. Some java.io classes (apparently just the output classes) include a flush method. When a close method is called on a such a class, it automatically performs a flush. There is no need to explicitly call flush before calling close.
One stream is chained to another by passing it to the constructor of some second stream. When this second stream is closed, then it automatically closes the original underlying stream as well.
If multiple streams are chained together, then closing the one which was the last to be constructed, and is thus at the highest level of abstraction, will automatically close all the underlying streams. So, one only has to call close on one stream in order to close (and flush, if applicable) an entire series of related streams.
Reading and Writing text files : When reading and writing text files:
it is almost always a good idea to use buffering (default size is 8K)
Unbuffered input and output classes operate only on one byte at a time.
Using a buffer will often increase performance by large factors.

FileInputStream & DataInputStream is very slow : since they call read() for every character. Use FileReader and BufferedReader instead. Reader objects use Large Buffer. Unbuffered input/output classes operate only on one byte at a time. Using a buffer will often increase performance by large factors.

13.9: Best Practices in I/O

Best Practices in I/O (contd..)

- **Do not implement Serializable unless needed.**
- **Serialization and Subclassing**

June 11, 2015 | Proprietary and Confidential | - 35 -

IGATE Speed.Agility.Imagination

Implementing Serializable
Do not implement Serializable lightly, since it restricts future flexibility, and publicly exposes class implementation details which are usually private.

Serialization and Subclassing
Interfaces and classes designed for inheritance should rarely implement Serializable, since this would force a significant and (often unwanted) task on their implementors and subclasses.

However, even though most abstract classes do not implement Serializable, they may still need to allow their subclasses to do so, if desired.

There are two cases. If the abstract class has no state, then it can simply provide a no-argument constructor to its subclasses. If the abstract class has state, however, then there is more work to be done

Summary

- **In this lesson you have learnt:**
  - Different types of I/O Streams supported by Java
  - Important classes in java.io package
  - Object Serialization
  - Best Practices in Java I/O

Review Question

- **Question 1: What is a buffer?**
  - **Option 1 :** Section of memory used as a staging area for input or output data.
  - **Option 2 :** Cable that connects a data source to the bus.
  - **Option 3 :** Any stream that deals with character IO.
  - **Option 4 :** A file that contains binary data.
- **Question 2: Can data flow through a given stream in both directions?**
  - True
  - False
- **Question 3: _____ is the name of the abstract base class for streams dealing with *character input***

Core Java 8 with JAXB and Development Tools

Introduction to Junit 4

IGATE
Speed.Agility.Imagination

Lesson Objectives

- **After completing this lesson, participants will be able to**
  - Understand importance of unit testing
  - Install and use JUnit 4
  - Use JUnit Within Eclipse

June 11, 2015 | Proprietary and Confidential | - 2 -

IGATE
Speed.Agility.Imagination

This lesson covers the one of the important development tool called JUnit. It explains how to work with JUnit, install, configure tests, run test with use of an IDE.

Lesson outline:

14.1: Introduction
14.2: JUnit
14.3: Installing and Running JUnit
14.4: Testing with JUnit
14.5: Testing Exceptions
14.6: Test Fixtures
14.7: Best Practices

14.1: Introduction

Why is Testing Necessary

- **To test a program implies adding value to it.**
  - Testing means raising the reliability and quality of the program.
  - One should not test to show that the program works rather that it does not work.
  - Therefore testing is done with the intent of finding errors.
- **Testing is a costly activity.**

June 11, 2015 | Proprietary and Confidential | - 3 -

IGATE Speed.Agility.Imagination

Why Testing?
Why is Testing Necessary?
In SDLC, testing plays a vital role. When one tests a program one adds value to the program, in turn raising the quality and reliability of the program.

When we say "reliable", it implies finding and removing errors. Hence one should not test a program to show that it works, but to show that program does not work.

Testing cannot guarantee against software problems or even failures but it can minimize the risks of faults developing once the software is put to use.

Typically when testing one should start with assumptions that the program contains errors and the test the program to find as many errors as possible.
Testing is a costly activity. A test which does not find an error is a waste of time and money. "A test case that finds an error is a valuable investment".

14.1: Introduction

What is Unit Testing

- **The process of testing the individual subprograms, subroutines, or procedures to compare the function of the module to its specifications is called Unit Testing.**
  - Unit Testing is relatively inexpensive and an easy way to produce better code.
  - Unit testing is done with the intent that a piece of code does what it is supposed to do.

June 11, 2015 | Proprietary and Confidential | - 4 -

IGATE Speed.Agility.Imagination

Why Testing?
What is Unit Testing?
There are various phases in testing. However, in this course we are mainly concentrating on unit testing. Testing individual subprograms or procedure to compare the functions of the module to its required specifications is Unit Testing. This is an inexpensive activity and a very easy way to produce better code. In other words, unit test is a small piece of code that is written by the developer that will exercise a specific area of functionality of the code being tested.
For example: You write a functionality to sort a list and then check if the sort works. You then modify the functionality to accept the sort order, as well, and then you test this newly added functionality.
The point to note here is that while doing Unit testing, the developers are not worried about verification and validation of the program. It is just that the functionality should be running as required.
Unit Testing is also called as Test Driven Development (TDD). A significant advantage of TDD is that it enables you to take small steps while writing software. Most of you will be already doing some amount of unit testing in an ad hoc manner.

14.1: Introduction

What is Test-Driven Development (TDD)

- **Test-Driven Development, also called Test-First Development, is a technique in which you write unit tests before writing the application functionality.**
  - Tests are **non-production code** written in the same language as the application.
  - Tests return a simple **pass** or **fail**, giving the developer immediate feedback.

June 11, 2015 | Proprietary and Confidential | - 5 -

IGATE Speed.Agility.Imagination

Test Driven Development (TDD) requires developers to create automated unit tests that define code requirements before writing the code itself.

14.1: Introduction

Why Unit Testing

- **You can cite following reasons for doing a Unit Test:**
  - Unit testing helps developers find errors in code.
  - It helps you write better code.
  - Unit testing saves time later in the production/development cycle.
  - Unit testing provides immediate feedback on the code.

June 11, 2015 | Proprietary and Confidential | - 6 -

IGATE Speed.Agility.Imagination

Why Testing?
Why Unit Testing?
Why should a developer do unit testing? Some of the reasons that can be identified are given below:

Unit Testing helps developers find errors in code: When a developer starts writing unit tests, they can actually be surprised with how many errors are encountered in a small function that is written. It makes life of the developer easy. In case errors are "not found in time", and are delayed till the end, then the entire module may fail.

Testing helps you write better code: Unit testing can help developer during the initial phase of development. Unit testing makes designs better and drastically reduce the time required while debugging.

Unit testing will save time later: To understand this concept, let us consider an example. Suppose you have written a small piece of code and you start testing. You identify that it is likely to fail in one tricky situation and the test finds out the error. You fix it, and continue the development. Since you have already fixed the bug, there is a little chance that the module might fail. "Hence following the test driven development approach is beneficial".

Unit testing provides immediate feedback on the code: When a developer is writing a critical module / functionality for a user interface, testing at that point will provide immediate feedback. There is no need to wait till the code becomes part of the application and then test it to check whether it works.

Overall Unit testing activity benefits the developer.

14.2: JUnit

Need for Testing Framework

- **Testing without a framework is mostly ad hoc.**
- **Testing without framework is difficult to reproduce.**
- **Unit testing framework provides the following advantages:**
  - It allows to organize and group multiple tests.
  - It allows to invoke tests in simple steps.
  - It clearly notifies if a test has passed or failed.
  - It standardizes the way tests are written.

June 11, 2015 | Proprietary and Confidential | - 7 -

IGATE Speed.Agility.Imagination

Why use JUnit?
Need for Testing Framework:
After having understood the need for Unit testing, there is a requirement to understand how to do Unit testing.
Mostly the Unit Testing done by the developers is ad hoc. The tests done in this manner are not put across in code at all. If at all they are put up, then they are written in such a manner that they cannot be reused in future. If they can be used in future, then typically they might be reproduced differently every time they are used. “Hence it is said that testing without a framework is difficult to reproduce”. With the help of a framework, the tests get documented in code and are reproduced in the same manner, whenever required.

14.2: JUnit

What is JUnit

- **JUnit is a free, open source, software testing framework for Java.**
- **It is a library put in a jar file.**
- **It is not an automated testing tool.**
- **JUnit tests are Java classes that contain one or more unit test methods.**

IGATE
Speed.Agility.Imagination

Why use JUnit?
What is JUnit?
JUnit is an open source, software testing framework for Java developed by Kent Beck and Erich Gamma. JUnit allows developers to write Unit test cases for your Java code. It is library put in a jar file.
JUnit is not an automated testing tool. The developer has to write the test files and execute. JUnit offers some support so that the developer can easily write test files. It includes a tool which is called test runner to run your test files.
JUnit provides an easy way to state how the code should work. By expressing your intentions in code, you can use the JUnit test runners to verify that your code behaves according to your intentions.

14.2: JUnit

Why JUnit

- JUnit allows you to write tests faster while increasing quality and stability.
- It is simple, elegant, and inexpensive.
- The tests check their own result and provide feedback immediately.
- JUnit tests can be put together in a hierarchy of test suites.
- The tests are written in Java.

June 11, 2015 | Proprietary and Confidential | - 9 -

IGATE Speed.Agility.Imagination

Why use JUnit?
Why JUnit?
Many developers will agree to the fact that the code should be tested before it is delivered. We have already seen the reasons why a testing framework should be used. However, it is also necessary to understand why JUnit should be used. Some of the reasons are defined below:

JUnit allows you to write tests faster while increasing quality and stability: Using JUnit, a developer spends less time debugging, and confidently makes changes to the code. With constant testing, any new functionality that is added can be verified for whether it is working or not. Hence the developer can be more positive about adding new features, because the developer now knows that it is less likely to fail. If a bug is detected while running tests, then the source code is fresh in your mind, so the bug is easily found. Also tests written in JUnit help you write code at a fast pace, and identify defects quickly. Writing tests builds the stability of the code and ensures that any changes that are made are working. As a result of the change, there is no effect on the software.

JUnit is simple, elegant, and inexpensive: "Simplicity" is the keyword while writing a test. Developers should not find it time consuming or difficult to write tests. With JUnit, the TDD can be followed very easily. It is simple and easy to put JUnit in practice. Developers can incrementally write tests as they increment their code. JUnit tests are such that they can be executed easily and frequently and it does not disturb the development process. This framework offers a cheap way of testing since it is an open source and freely downloadable ware.

JUnit tests check their own result and provide feedback immediately: Manual Unit testing is a tedious task and obviously it is time consuming to compare the expected and the actual result. As a result, developers tend to do away with Unit testing. JUnit tests can be run easily and they check their own results. The developer immediately gets a feedback if the tests have passed or failed. Hence any manual intervention is not required while executing the tests.

JUnit tests can be put together in a hierarchy of test suites: JUnit tests can be organized into test suites containing test cases and even other test suites. The composite behavior of JUnit tests allows you to assemble collections of tests and automatically regression test the entire test suite in one go. You can also run the tests for any layer within the test suite hierarchy.

(Test Suites will be covered later in detail)

JUnit tests are written in Java: Testing Java software using Java tests forms a "seamless bond" between the test and the code under test. The tests become an extension to the overall software and code can be refactored from the tests into the software under test. The Java compiler helps the testing process by performing static syntax checking of the unit tests and ensuring that the software interface contracts are being obeyed.

14.3: Installing and Running JUnit

Steps for Installing JUnit

- **Following are the steps for installing and running JUnit:**
  - Download JUnit from www.junit/org. You can download either the jar file or the zip file.
    - Unzip the JUnit zip file
  - Add the jar file to the CLASSPATH.
    - Set CLASSPATH=.;%CLASSPATH%;junit-4.3.1.jar

June 11, 2015 | Proprietary and Confidential | - 10 -

IGATE Speed.Agility.Imagination

Installing and Running JUnit:
To start writing tests using JUnit, you will require the jar file for the latest version of JUnit. You can download the jar file or the zip file from the website. The zip file contains the source code and the documentation, as well.
Once downloaded, unzip the zip file that has downloaded. Add the jar file in the classpath. Alternatively the classpath can also be set through the Environment Variables option.
Once done you are ready to use JUnit.

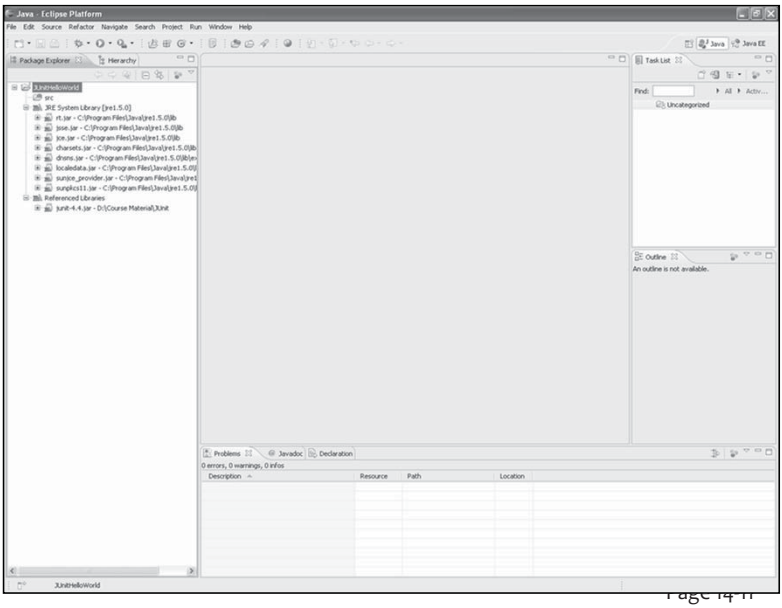
14.3: JUnit within Eclipse

Using JUnit within Eclipse

- **JUnit can be easily plugged in with Eclipse.**
- **Let us understand how JUnit can be used within Eclipse.**
  - Consider a simple "Hello World" program.
  - The code is tested using JUnit and Eclipse IDE.
- **Steps for using JUnit within JUnit:**
  - Open a new Java project.
  - Add junit.jar in the project Build Path.

IGATE
Speed.Agility.Imagination

Note: The following figure depicts JUnit being added to the build path of the project.



14.3: JUnit within Eclipse

Using JUnit within Eclipse (Contd.)

- **Write the Test Case as follows:**

```
import org.junit.Test;
import static org.junit.Assert.*;
public class TestHelloWorld {
  @Test
   public void testSay()
    {
      HelloWorld hi = new HelloWorld();
      assertEquals("Hello World!", hi.say());
    } }
```

```
class  HelloWorld{
   String say(){
      return "Hello World!"}
}
```

June 11, 2015 | Proprietary and Confidential | - 12 -

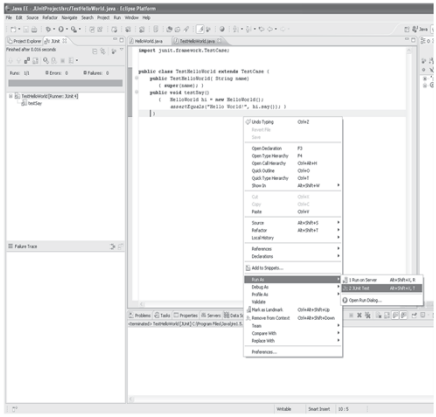
IGATE Speed.Agility.Imagination

Note:
Test method must be annotated with @Test annotation then only Junit framework and eclipse will consider it as test method.
assertEquals(expected,actual) is performing the Test .This function internally uses Java assert feature which throws AssertionError exception when assertion fails.

14.3: JUnit within Eclipse

Using JUnit within Eclipse (Contd.)

- **Run the Test Case.**
  - Right-click the Project → Run As → JUnit Test
- **The output of the test case is seen in Eclipse.**

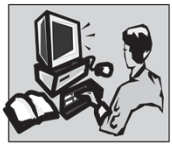


IGATE Speed.Agility.Imagination

14.3: JUnit within Eclipse

Demo

- **Demo on:**
  - Using JUnit with Eclipse
    - HelloWorld.java
    - TestHelloWorld.java



IGATE
Speed.Agility.Imagination

14.4: Testing with JUnit

Annotation Types in JUnit4.x

- **JUnit4.x introduces support for the following annotations:**
  - @Test – used to signify a method is a test method
  - @Before – can do initialization task before each test run
  - @After – cleanup task after each test is executed
  - @BeforeClass – execute task before start of tests
  - @AfterClass – execute cleanup task after all tests have completed
  - @Ignore – to ignore the test method

June 11, 2015 | Proprietary and Confidential | - 15 -

IGATE Speed.Agility.Imagination

Testing with JUnit – Annotations:
Annotation Types in JUnit4.x:
JUnit4.x introduced annotations. They are as follows:

@Test : It is used to signify a method as a test method. There are some options which can be mentioned with this annotation. For example: @Test(timeout=100) fails if the test takes longer than 100 milliseconds for execution. This can be used to test infinite loops. Earlier you would have to start every method with 'test' which is now not required. The method can be named whatsoever. You have to simply prefix it with @Test annotation.

@Before : It is used to carry some task before each test is run. This can be used for initialization required before the test.

@After : It is used to perform cleanup after each test is executed.

@BeforeClass : It will execute the method before the start of the tests. This can be used for initialization of intensive resources like database connection.

@AfterClass : It will execute the method after all tests have finished. Cleanup activities are carried out.

@Ignore : It will ignore the test method. This can be useful when the base code has been changed but the test is yet to be revised. You can also used this annotation when you do not want to run a test currently because it takes too long.

(As we go ahead, we will understand each of the annotation in detail.)

14.4: JUnit Framework

Simple Example using Junit4.x

- **Consider the following code snippet:**

```
import static org.junit.Assert.*;
import org.junit.Test;
 public class FirstJUnitTest {
  @Test
      public void simpleAdd() {
        int result = 1;
        int expected = 1;
        assertEquals(expected, actual);
 } }
```

June 11, 2015 | Proprietary and Confidential | - 16 -

IGATE Speed.Agility.Imagination

Understanding JUnit Framework:
Before we move any further, let us understand a very simple test case. Notice the static import. Since JUnit4.x is closely bound to Java 5, the static imports are permitted. JUnit4.x uses org.junit.* package, which provides JUnit core classes and annotations. As mentioned earlier, the TestCase class is not required to be inherited by the test class.
The class, which includes at least one @Test annotation, is treated as the test class. In JUnit4.x, the assert methods are static. Hence you need to do either of the following:

Call assert methods using Assert.assertEquals().

Do a static import as we have done in the snippet shown on the slide.

You also need not tag test methods by starting their name with "test". You instead only need to specify the @Test annotation.

14.4: Testing with JUnit

Assert Statements in JUnit

- **Following are the methods in Assert class :**
  - Fail(String)
  - assertTrue(boolean)
  - assertEquals([String message],expected,actual)
  - assertNull([message],object)
  - assertNotNull([message],object)
  - assertSame([String],expected,actual)
  - assertNotSame([String],expected,actual)
  - assertThat(String,T actual, Matcher<T> matcher)

June 11, 2015 | Proprietary and Confidential | - 17 -

IGATE Speed.Agility.Imagination

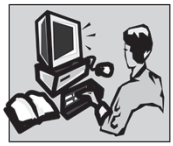
Testing with JUnit – Assertions:
Assert Statements in JUnit:
The org.junit.Assert class provides a set of useful assertions methods. You can call these methods by either using Assert.assertEquals() or by referencing through static import. Any failed assertions will be only recorded. Some assertion methods are as follows:

Fail([String]) : It signals the failure of a test. This method has two formats. If the String argument is not provided, then no message is displayed. Else the String argument message is displayed.
assertTrue(boolean) : It asserts if the condition is true. Similarly assertFalse(boolean) asserts if the condition is false.
assertEquals([String message],expected,actual) : It asserts whether the two objects passed as arguments are equal. This method can accept any kind of values for comparison like double long, etc..
assertNull([message],object) : It asserts that an object is null.
assertNotNull([message],object) : It asserts that an object is not null.
assertSame([String],expected,actual) : It asserts that two objects refer to the same object and assertNotSame([String],expected,actual) asserts that two objects do not refer to the same object.
assertThat(String, T actual, Matcher <T> matcher) : It asserts that "actual" satisfies the condition specified by the "matcher".

14.4: Testing with JUnit

Demo

- **Demo on:**
  - Using @Test Annotation
  - Using Assert Methods
    - Counter.java & Testcounter.java
    - Person.java & TestPerson.java



June 11, 2015 | Proprietary and Confidential | - 18 -

IGATE Speed.Agility.Imagination

Note:
Refer to JUnitProject.
Demo 1: Counter.java and testCounter.java
Demo 2: Person.java and TestPerson.java

In both the examples, we have used the @Test annotation and the assert methods to evaluate if the methods in the underlying class are working properly.

Remove the @Test annotation from one of the Test methods, and then observe the output. That particular method is not considered as a test method.

14.4: Testing with JUnit

Using @Before and @After

- **Test fixtures help in avoiding redundant code when several methods share the same initialization and cleanup code.**
- **Methods can be annotated with @Before and @After.**
  - @Before: This method executes before every test.
  - @After: This method executes after every test.
- **Any number of @Before and @After methods can exist.**
- **They can inherit the methods annotated with @Before and @After.**

June 11, 2015 | Proprietary and Confidential | - 19 -

IGATE Speed.Agility.Imagination

Testing with JUnit – Test Fixtures:
Using @Before and @After:
Many a times a set of common resources or data that you need to run one or more tests are required. To avoid the redundant code when several methods share the same initialization and cleanup code, you can use @Before and @After annotations. Prefix the methods with the respective annotations. In the previous version, you had to use setup() and teardown() methods to perform this task. @Before annotated methods run before every test, and @After prefixed methods run after every test.
You can also have any number of @Before and @After prefixed methods as you need. It is possible to inherit the @Before and @After methods. The @Before methods in the superclass will be executed prior to the derived class @Before methods. The @After in the subclass are executed before the superclass @After methods. The superclass @Before and @After methods execute automatically and there is no need to call them explicitly. Also a logger can be initialized in @Before method of a Test Case.
When inherited, the overall execution process will be as follows:

@Before methods in the superclass
@Before methods in the current class
@Test methods in the current class
@After methods in the current class
@After methods in the superclass

14.4: Testing with JUnit

Using @Before and @After

- **Example of @Before:**

```
@Before
public void beforeEachTest() {
Calculator cal=new Calculator();
Calculator cal1=new Calculator("5", "2"); }
```

- **Example of @After:**

```
@After
public void afterEachTest() {
Calculator cal=null;
Calculator cal1=null; }
```

June 11, 2015 | Proprietary and Confidential | - 20 -

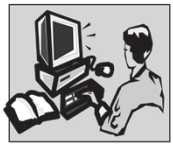
IGATE Speed.Agility.Imagination

Note: The methods annotated with @Before and @After should be declared as public void.

14.4: Testing with JUnit

Demo

- **Demo on:**
  - Using the @Before and @After annotations
    - TestPersonFixture.java



June 11, 2015 | Proprietary and Confidential | - 21 -

IGATE Speed.Agility.Imagination

Note:
Demo 5: TestPersonFixture.java
Notice the output of the execution of @Before and @After methods.

14.5: Testing Exceptions

Testing Exceptions

- It is ideal to check that exceptions are thrown correctly by methods.
- Use the expected parameter in @Test annotation to test the exception that should be thrown.
- For example:

```
@Test(expected = ArithmeticException.class)
public void divideByZeroTest() {
calobj.divide(15,0);
}
```

June 11, 2015 | Proprietary and Confidential | - 22 -

IGATE Speed.Agility.Imagination

Testing with JUnit – Testing Exceptions:
Testing using the Exceptions:
At times you would want to check that an exception is thrown correctly under certain circumstances. In previous versions, it was a laborious task which involved writing a try block around the code that throws the exception, then adding a fail() statement at the end of the try block.
For example:

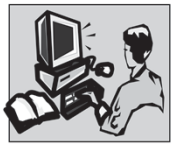
```
public void TestDivideByZero()
{
  try {
        calobj.divide( 15, 0 );

  }
   catch (ArithmeticException e) {
     fail(e.getMessage);
   }
}
```

In this case, if the exception is not thrown or a different exception is thrown, then the test will fail.
Now, you can use the expected parameter in the @Test annotation to test the exception that should be thrown as shown on the slide.
However, you may still need to use the traditional try-catch block if you want to test the exception's detail message or other properties.

14.5: Testing Exceptions

Demo

- **Demo on:**
  - Exception Testing
    - Person.java & TestPerson2.java



IGATE
Speed.Agility.Imagination

.

14.6: Test Fixtures

Using @BeforeClass and @AfterClass

- **Suppose some initialization has to be done and several tests have to be executed before the cleanup.**
- **Then methods can be annotated by using the @BeforeClass and @AfterClass.**
  - @BeforeClass: It is executed once before the test methods.
  - @AfterClass: It is executed once after all the tests have executed.

June 11, 2015 | Proprietary and Confidential | - 24 -

IGATE Speed.Agility.Imagination

Testing with JUnit – Test Fixtures:
Using @BeforeClass and @AfterClass:
Sometimes the requirement can be wherein the initialization code is run, and then several tests are executed. In this scenario, we need to use @BeforeClass and @AfterClass annotated methods. This can also be termed as "one time setup and teardown".

@BeforeClass : The method annotated with this will be executed once before the tests are run. It means if you have just one test or ten tests, then this will run one time before the very first test executed. In case of inheritance, the base class @BeforeClass method will execute once.
@Afterclass : The method annotated with this will run once after all the tests have finished execution. In case of inheritance, the @AfterClass in the derived class will execute first and then the method from base class will be executed.

Unlike @Before and @After, only one set of @BeforeClass and @AfterClass annotated methods are allowed.

14.6: Test Fixtures

Using @BeforeClass and @AfterClass

- **Example of @BeforeClass:**

```
@BeforeClass
public static void beforeAllTests() {
Connection conn=DriverManager.getConnection(….);}
```

- **Example of @AfterClass:**

```
@AfterClass
public static void afterAllTests() {
conn.close; }
```

- **The methods using this annotation should be public static void**

June 11, 2015 | Proprietary and Confidential | - 25 -

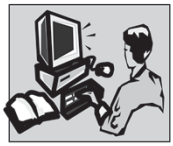
IGATE Speed.Agility.Imagination

The methods using this annotation should be public static void. Because @BeforeClass gets called before the class is created and @Afterclass complements the @BeforeClass annotated method.

14.6: Test Fixtures

Demo

- **Demo on:**
  - Using the @BeforeClass and @AfterClass annotations



June 11, 2015 | Proprietary and Confidential | - 26 -

IGATE
Speed.Agility.Imagination

Note:
Demo 6: TestPersonFixture.java

14.6: Test Fixtures

Using @Ignore

- The @Ignore annotation notifies the runner to ignore a test.
- The runner reports that the test was not run.
- Optionally, a message can be included to indicate why the test should be ignored.
- This annotation should be added either in before or after the @Test annotation.

June 11, 2015 | Proprietary and Confidential | - 27 -

IGATE Speed.Agility.Imagination

Testing with JUnit – Ignoring Tests:
Using @Ignore:
In JUnit4.x, if you need to temporarily ignore a test from being executed, then annotate them with @Ignore.

The @Ignore annotation can be included either in front or after the @Test annotation. The runner then ignores these tests and reports how many tests were not run along with the pass and fail tests.
Optionally, a message can be included to indicate why a test should be ignored for a particular test that is run. Though it is optional it is recommended that the message should be given because you might forget why a particular test has been ignored

In addition to this, even a class can be annotated with @Ignore, and all the tests in that class will be ignored.
The @Ignore annotation is should ideally be used when

the method cannot be tested in some form and it is documented in the code. This should be a special case and warrant a discussion or code review to see if there is any way to test it.
The test is not yet built - this should ideally happen only for legacy code. This should be also subject to code review and tasks should be put to add tests.

14.6: Test Fixtures

Using @Ignore

- **Example of @Ignore for a method:**

```
@Ignore ("The network resource is not currently available")
@Test
 public void multiplyTest() {
   ……}
```

- **Example of @Ignore for a class:**

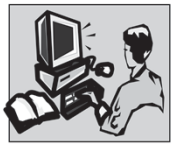
```
@Ignore public class TestCal {
@Test public void addTest(){ …. }
@Test public void subtractTest(){…..}
}
```


IGATE Speed.Agility.Imagination

14.6: Test Fixtures

Demo

- **Demo on:**
  - Using the @Ignore
    - Student.java & TestStudent.java



June 11, 2015 | Proprietary and Confidential | - 29 -

IGATE Speed.Agility.Imagination

Note:
Demo 7: TestStudent.java and Student.java
The Ignore annotation included in the program will not allow the respective test case to execute. The JUnit panel also indicates that one test has been ignored.

14.7: Best Practices

Unit Testing

- Start with writing tests for methods having the fewest dependencies and then work your way up.
- Ensure that tests are simple, preferably with no decision making.
- Use constant, expected values in the assertions instead of computed values wherever possible.

June 11, 2015 | Proprietary and Confidential | - 30 -

IGATE Speed.Agility.Imagination

Best Practices in JUnit:
Unit Testing Best Practices:
Unit Testing is the execution and validation of a block of code, which is created by a developer, in isolation. Some best practices to keep in mind while doing unit testing with JUnit are elaborated as follows:

Start with writing tests for methods having the fewest dependencies and then work your way up. If the testing process starts from a higher level method, then there is a probability that the test may fail since the subordinate method returns an incorrect value to the high level method. This increases the time required to find the cause of the problem, and to do this you need to test the subordinate method. Hence start with a bottom up approach

Ensure that tests are simple with no decision making. Decision making in the tests add to the number of ways a test method can be executed. Normally the requirement is to change the input of the method only.

Use constant expected values in the assertions instead of computed values wherever possible. Consider the following example:

```
returnVal=methodToTest(input);
assertEquals(returnVal,computeExpected(input));
assertEquals(returnVal,12);
```

In this snippet, the computeExpected method will have to implement the similar logic as the method which is being tested. The test class will tend to grow lengthy. Moreover, the second assertion statement is easy to implement, understand, and maintain.

14.7: Best Practices

Unit Testing

- Ensure that each unit test is independent of all other tests.
- Clearly document all the tests.
- Test all methods whether public, protected, or private.
- Test the exceptions.

June 11, 2015 | Proprietary and Confidential | - 31 -

IGATE Speed.Agility.Imagination

Best Practices in JUnit:
Unit Testing Best Practices:
Ensure that each unit test is independent of all other tests. The test method ideally executes one specific behavior for a single method. Placing too many assertions in one test case may cause a problem. This is because even if one of the assertion fails, then the entire test fails. Strive for one assertion per test case.
Clearly document each unit test. The name of the test method should ideally indicate which method is being tested. This helps in easy maintenance and reduced efforts in refactoring. Provide proper supporting comments to describe any special conditions.
Test all methods whether public, protected, or private.
Create unit tests which specifically check for exceptions. If a method throws more than one exception, then appropriate unit tests must be created to simulate those situations when the exceptions will be thrown.

Note: Testing exceptions will be discussed in Lesson 17.

14.7: Best Practices

JUnit

- Do not use the constructor of test case to setup a test case, instead use an @Before annotated method.
- Do not assume the order in which tests within a test case should run.
- Place tests and the source code in the same location.
- Put non-parameterized tests in a separate class.

June 11, 2015 | Proprietary and Confidential | - 32 -

IGATE Speed.Agility.Imagination

Best Practices in JUnit:

JUnit Best Practices:

Some more best practices to be followed while working with JUnit are elaborated as follows:

Do not use the constructor of test case to setup a test case, instead use an @Before method to do the setup task. The reason for this is incase the constructor fails to do the setup, JUnit simply throws an "AssertionFailsError" which indicates that the test could not be instantiated. Also the error stack trace is not very informative. As a result, it makes it hard to figure out the exceptions underlying cause. Using the @Before annotated method is handy because any exception thrown within this method is reported correctly and the error stack trace is informative. Hence tracking the probable cause of error is easy.

Never assume that tests will be called in a specific order. If the tests are dependent on each other, then it is better to compose them in test suites. In this way, it is controlled that the tests run in the order in which they were mentioned.

Keep both the tests and source code in the same location. This will compile both test and class during a build. In this case, the tests and class are synchronized during the development.

Put non-parameterized tests in a separate class to avoid running of that test since parameterized tests create new instance of the class every time. This leads to running the non-parameterized tests also every time.

14.7: Best Practices

JUnit

- **When writing tests consider the following questions:**
  - When do I write tests?
  - Do I test everything?
  - How often the tests should be run?
  - Why use JUnit, instead why not use println() or a debugger?

June 11, 2015 | Proprietary and Confidential | - 33 -

IGATE Speed.Agility.Imagination

Best Practices in JUnit:
JUnit Best Practices:
While writing test cases using JUnit, you should consider certain important aspects:

When do I write tests?

Tests should be ideally written before the code. Follow the test driven development approach instead of keeping the test to be written at the very end. The bugs can then be corrected immediately as they are reported.

Do I test everything?

At least test everything that can possibly break. Maximize the testing investment since it is equivalent to investing in design. If something is difficult to test, then relook at your design. Improve the design so that it is easier to test.

How often the tests should be run?

Run all your unit tests as often as possible that is whenever there is a change in code. Running the test cases often gives the confidence that the changes did not cause anything to break or fail.

Why use JUnit, instead why not use println() or a debugger?

Inserting println or debugging statements into the code causes the code to become lengthy. Also you need to manually scan the output every time your program runs to ensure that the program does what it is expected to do. Using debugger is a manual process that also requires visual inspections. It takes less time in the long run to codify expectations in the form of an automated JUnit test that retains its value over time.

Lab : Introduction to Junit

- **Lab 9: Introduction to Junit**

Review Question

- **Question 1: Why should one do Unit Testing?**
  - Option 1: Helps to write code better
  - Option 2: Provides immediate feedback on the code
  - Option 3: Because it is one of the testing methods that has to be carried out
- **Question 2: JUnit is a licensed product and can be purchased with Java.**
  - True / False

Review Question

- **Question 3: To start working with JUnit in Eclipse, you need to add junit.jar in ___.**
  - Option 1: CLASSPATH
  - Option 2: BUILDPATH
  - Option 3: Project Settings
- **Question 4: You can add a number of tests using the <batchtest> element.**
  - True / False

Core Java 8 with JAXB and Development Tools

Lambda Expressions

IGATE
Speed.Agility.Imagination

Lesson Objectives

- **After completing this lesson, participants will be able to**
  - Understand the concept of Lambda expressions
  - Work with lambda expressions
  - Use method references and functional interfaces

June 11, 2015 | Proprietary and Confidential | - 2 -

IGATE
Speed.Agility.Imagination

This lesson covers new feature in Java 8, lambda expressions. It also covers concepts of functional interfaces and method references.

Lesson outline:

15.1: Introduction
15.2: Writing Lambda Expressions
15.3: Functional Interfaces
15.4: Types of Functional Interfaces
15.5: Method reference
15.6: Best practices

15.1: Functional Interfaces

Functional Interface

- **Functional Interface is an interface having exactly one abstract method**
- **Such interfaces are marked with optional @FunctionalInterface annotation**

```
@FunctionalInterface
interface xyz {
        //single abstract method
}
```

June 11, 2015 | Proprietary and Confidential | - 3 -

IGATE Speed.Agility.Imagination

Before we get into the discussion of Lambda expressions, lets first have a look at functional interfaces. When an interface is declared with only one abstract method, then it is referred as Functional Interface. The method in functional interface is called as functional method. Along with the functional method, you can also add default and static method to functional interface. Optionally such interfaces can be annotated with @FunctionalInterface annotation. This annotation is not a requirement for the compiler to recognize an interface as a functional interface, but merely an aid to capture design intent and enlist the help of the compiler in identifying accidental violations of design intent.

Note: An empty interface is called as “Marker Interface”.

15.1: Functional Interfaces

Functional Interface : Example

```
@FunctionalInterface
public interface MaxFinder {
    //single abstract method to find max between two numbers
    public int maximum(int num1,int num2);
}
```

How to implement this interface?

June 11, 2015 | Proprietary and Confidential | - 4 -

IGATE Speed.Agility.Imagination

As shown in the slide example, a functional interface is annotated with @FunctionalInterface so as it instructs compiler to rectify any rules violation.

15.1: Functional Interfaces

Functional Interface : Implementation

- **Class Implementation:**

```
public class MaxFinderImpl implements MaxFinder {
    @Override
    public int maximum(int num1, int num2) {
        return num1>num2?num1:num2;
    }
}
```

```
MaxFinder finder = new MaxFinderImpl();
int result = finder.maximum(10, 20);
```

Want to know more concise way for implementation?

June 11, 2015 | Proprietary and Confidential | - 5 -

IGATE Speed.Agility.Imagination

The slides shows one way to implement the functional interfaces. Is it worthy to create separate class for single method implementation?

15.1: Functional Interfaces

Functional Interface : Implementation

- **Lambda Expression:**

```
public class MaxFinderImpl implements MaxFinder {
    @Override
    public int maximum(int num1, int num2) {
            return num1>num2?num1:num2;
    }
}
```

```
MaxFinder finder = (num1,num2) -> num1>num2?num1:num2;
int result = finder.maximum(10, 20);
```

Return type of "λE" is Functional Interface!

June 11, 2015 | Proprietary and Confidential | - 6 -

IGATE Speed.Agility.Imagination

The slides shows how to implement the functional interfaces using lambda expression. This way is more concise as we are implementing functional interface without creating additional class.

Lets discover the lambda expression in detail.

15.2: Lambda Expressions

Lambda Expression

- **Lambda expression represents an instance of functional interface**
- **A lambda expression is an anonymous block of code that encapsulates an expression or a block of statements and returns a result**
- **Syntax of Lambda expression:**

```
(argument list) -> { implementation }
```

- **The arrow operator -> is used to separate list of parameters and body of lambda expression**

June 11, 2015 | Proprietary and Confidential | - 7 -

IGATE Speed.Agility.Imagination

What is Lambda expression?

Lambda expression allows for creation and use of single method anonymous classes instead of creating separate concrete class for functional interface implementation.

A lambda expression is an anonymous block of code that encapsulates an expression or a series of statements and returns a result. They can accept zero or more parameters, any of which can be passed with or without type specification since the type can be automatically derived from the context.

The parameter list for a lambda expression can include zero or more arguments. If there are no arguments, then an empty set of parentheses can be used. No parenthesis is required for single argument.

Why Lambda Expressions?

Lambda expressions are an important addition to Java that greatly improves overall maintainability, readability, and developer productivity. They can be applied in many different contexts, ranging from simple anonymous functions to sorting and filtering Collections. Moreover, lambda expressions can be assigned to variables and then passed into other objects.

15.2: Lambda Expressions

Lambda Expression

- **Sample Lambda Expressions**

Functional Method	Lambda Expression
int fun(int arg);	(int num) -> num + 10
int fun(int arg0,int arg1);	(int num1,int num2) -> num1+num2

int fun(int arg0,int arg1);	(int num1,int num2) -> {
	int min = num1>num2?num2:num1;
	return min; }

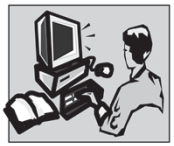
String fun();	() -> “Hello World!”
void fun();	() -> { }
int fun(String arg);	(String str) -> str.length()
int fun(String arg);	str -> str.length()

IGATE Speed.Agility.Imagination

15.2: Lambda Expressions and Functional Interface

Demo

- **Execute the :**
  - CalculatorDemo



IGATE
Speed.Agility.Imagination

15.3: Functional Interfaces

Built-in Functional Interfaces

- **Java SE 8 provides a rich set of 43 functional interfaces**
- **All these interfaces are included under package java.util.function**
- **This set of interfaces can be utilized to implement lambda expressions**
- **All functional interfaces are categorized into four types:**
  - Supplier
  - Consumer
  - Predicate
  - Function

June 11, 2015 | Proprietary and Confidential | - 10 -

IGATE Speed.Agility.Imagination

Built-in Functional Interfaces

As we have learnt so far, functional interfaces can be implemented by writing lambda expressions. Does it means, we have to write functional interface every time if we want to work with Lambda expressions?

Certainly no. As Java 8 comes with dozens of built-in functional interfaces, all are written and kept in java.util.function package. These interfaces can be useful when implementing lambda expressions.

All of these functional interfaces are written in generics format and hence can be applied in many different contexts. Use of such interfaces can greatly reduce the amount of code.

Functional Interfaces are categories into four types:

Supplier
Consumer
Predicate
Function

Lets discover the each type in detail!

15.3: Functional Interfaces

Supplier

- **A Supplier<T> represents a function that takes no argument and returns a result of type T.**
- **This is an interface that doesn't takes any object but provides a new one**

```
@FunctionalInterface
public interface Supplier<T> {
        T get();
}
```

- **List of predefined Suppliers:**
  - BooleanSupplier
  - IntSupplier
  - LongSupplier
  - DoubleSupplier etc.

June 11, 2015 | Proprietary and Confidential | - 11 -

IGATE Speed.Agility.Imagination

Supplier<T>

As the name suggest, Supplier<T> interface contains only one method get() which accepts no arguments but supplies a new object of type T. There are few predefined suppliers which returns object of specified type. For example, BooleanSupplier supplies Boolean valued results.

15.3: Functional Interfaces

Consumer

- **A Consumer<T> represents a function that takes an argument and returns no result**
- **A BiConsumer<T,U> takes two objects which can be of different type and returns nothing**

```
@FunctionalInterface
public interface Consumer<T> {
        void accept(T t);
}
```

```
@FunctionalInterface
public interface BiConsumer<T,U> {
        void accept(T t, U,u);
}
```

- **List of predefined Consumer:**
  - IntConsumer
  - LongConsumer
  - ObjIntConsumer
  - ObjLongConsumer etc.

IGATE Speed.Agility.Imagination

Consumer<T>/BiConsumer<T,U>

Consumers are of two types, a Consumer<T> accepts a single object and returns nothing, while the another BiConsumer<T> accepts two objects of any type and returns nothing. It contains a single method called accept() as shown in the slide.

15.3: Functional Interfaces

Predicate

- **A Predicate<T> represents a function that takes an argument and returns true or false result**
- **A BiPredicate<T,U> takes two objects which can be of different type and returns result as either true or false**

```
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
}
```

```
@FunctionalInterface
public interface BiPredicate<T,U> {
    boolean test(T t, U,u);
}
```

- **List of predefined Predicates:**
  - IntPredicate
  - LongPredicate
  - DoublePredicate etc.

June 11, 2015 | Proprietary and Confidential | - 13 -

IGATE Speed.Agility.Imagination

Predicate<T>/BiPredicate<T,U>

In mathematics, a predicate is a boolean-valued function that takes an argument and returns true or false. The function represents a condition that returns true or false for the specified argument. The other type BiPredicate is a predicate of two arguments which returns a Boolean value.

15.3: Functional Interfaces

Function

- **A Function<T> represents a function that takes an argument and returns another object**
- **A BiFunction<T,U> takes two objects which can be of different type and returns one object**

```
@FunctionalInterface
public interface Function<T,R> {
        R apply(T t);
}
```

```
@FunctionalInterface
public interface BiFunction<T,U,R> {
        R apply(T t, U,u);
}
```

- **List of predefined Functions:**
  - DoubleFunction<R>
  - IntFunction<R>
  - IntToDoubleFunction
  - DoubleToIntFunction
  - DoubleToLongFunction etc.

June 11, 2015 | Proprietary and Confidential | - 14 -

IGATE Speed.Agility.Imagination

Function<T,R>/BiFunction<T,U,R>

Function<T> represents a function that takes an argument of type T and returns a result of type R. BiFunction<T,U> represents a function that takes two arguments of types T and U, and returns a result of type R.

UnaryOperator<T,T>

Inherits the Function<T,T>, where it accepts and return a result of type T.

BinaryOperator<T>

Inherits from BiFunction<T,T,T>. Represents a function that takes two arguments of the same type and returns a result of the same.

Having discussed the different types of functional interfaces, let see now how to use them to write lambda expressions.

15.3: Lambda Expressions

Lambda Expression for Function Interfaces

- **Writing Lambda Expressions for Predefined Functional Interfaces**

Functional Interface	Functional Method	Lambda Expression
Supplier<String>	String get();	() -> “Hello World”;
BooleanSupplier	Boolean get();	() -> { return true; }
Consumer<String>	void accept(String str);	(msg) -> syso(msg);
IntConsumer	void accept(Integer num);	(num) -> syso(num);
Predicate<Integer>	boolean test(Integer num);	(num) -> num>0;
Function<String,Integer>	Integer apply(String str);	(str) -> str.length;
UnaryOperator<Integer>	Integer apply(Integer num);	(num) -> num +10;
BiFunction<String,String, Boolean>	Boolean apply(String user,String pass);	(user,pass) -> { //functionality to validate user }

IGATE Speed.Agility.Imagination

15.3: Functional Interfaces

Using Built-in Functional Interfaces

```
Consumer<String> consumer = (String str)-> System.out.println(str);
consumer.accept("Hello LE!");
Supplier<String> supplier = () -> "Hello from Supplier!";
consumer.accept(supplier.get());
//even number test
Predicate<Integer> predicate = num -> num%2==0;
System.out.println(predicate.test(24));
System.out.println(predicate.test(15));
//max test
BiFunction<Integer, Integer, Integer> maxFunction = (x,y)->x>y?x:y;
System.out.println(maxFunction.apply(25, 14));
```


Using Built-in Functional Interfaces

The first example show a consumer of type string which accepts a string and return nothing but print the accepted string value.

The supplier is of type String which supplies an object of String to a consumer which prints it.

The predicate is used to accept integer objects and returns true or false based on even test on given number.

The last example shows, a BiFunction which accepts two integer objects and return an integer which is maximum. The same expression can also be written as:

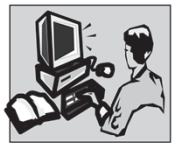
BinaryOperator<Integer> maxFunction = (x,y) -> x>y?x:y;

The functional interfaces described so far are utilized throughout the JDK, and they can also be utilized in developer applications.

15.3: Functional Interface

Demo

- **Execute the :**
  - FunctionalInterfaces



IGATE
Speed.Agility.Imagination

15.4: Method Reference

Method References

- **Method reference is a shorthand way to write lambda expressions**
- **It is a new way to refer a method by its name instead of calling it directly**
- **Consider the below lambda expression, which call println method of System.out object:**

```
Consumer<String> consumer = (String str)-> System.out.println(str);
```

- **Such lambda expressions are candidate for method references as it just calling a method for some functionality**
- **The same expression can be written as with method reference:**

```
Consumer<String> consumer = System.out :: println;
```

June 11, 2015 | Proprietary and Confidential | - 18 -

IGATE Speed.Agility.Imagination

Method Reference

If a lambda expression is written only to invoke a single method by name, then such expression can be shorthanded by using method reference. The syntax of method reference is:

<class or instance name> :: <method name>

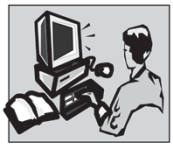
The double colon operator specifies the method reference. The method which is referred must reflect pre-defined or custom functional interface.

The slide example shows method println() which accept object to print and returns nothing, suits perfectly with predefined functional interface called Consumer<T> and matches with abstract method void accept(T) signature. Hence lambda expression can be short handed by refereeing it with method reference.

15.4: Method reference

Demo

- **Execute the :**
  - MethodReference



IGATE
Speed.Agility.Imagination

Summary

- **In this lesson, you have learnt:**
  - Writing Lambda Expressions
  - Functional Interfaces
  - Method reference

Review Question

- **Question 1 : Which of the following Lambda expressions are valid to perform addition of two numbers?**
  - **Option 1 :** (x, y) -> x +y;
  - **Option 2 :** (Integer x,Integer y) - > {return x+y;}
  - **Option 3 : (**Integer x, Integer y) -> (x + y);
  - **Option 4**: All of above
- **Question 2 : ____ is a predicate of two arguments.**
- **Question 3 : A method reference is shorthand to create a lambda expression using an existing method.**
  - True/False

Core Java 8 with JAXB and Development Tools

Stream API

IGATE
Speed.Agility.Imagination

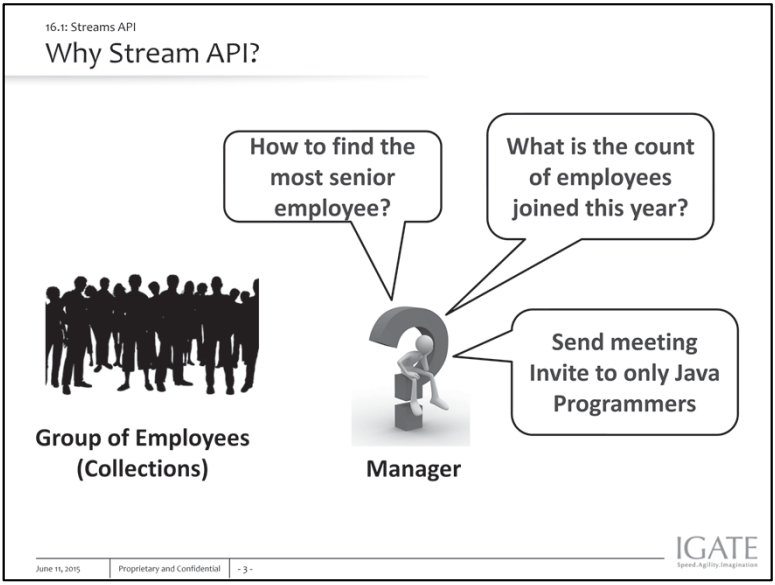
Lesson Objectives

- **After completing this lesson, participants will be able to**
  - Understand concept stream API
  - Use stream API with collections
  - Perform different stream opeations

This lesson covers new feature in Java 8, Stream API . It also covers processing collections using stream API .

Lesson outline:

16.1: Introduction
16.2: Stream API with Collections
16.3: Stream Operations
164: Best practices



Collections API in Java mainly focuses on storage and retrieval of its elements. Many times there is need to perform advanced retrieval operations. For example, Consider a collection of employees and the required operations to be performed:

1. Find the employees having age > 35
2. Group the employees based on their verticals and count.
3. and many more….

How many times do you find yourself re-implementing these operations using loops over and over again?

16.1: Stream API

Why Stream API?

- Stream API allows developers process data in a declarative way.
- Streams can leverage multicore architectures without writing a single line of multithread code
- Enhances the usability of Java Collection types, making it easy to iterate and perform tasks against each element in the collection
- Supports sequential and parallel aggregate operations

Stream API

June 11, 2015 | Proprietary and Confidential | - 4 -

IGATE Speed.Agility.Imagination

Stream API to rescue

Streams follow the "what, not how" principle. While working with Stream API, as a developer, we only need to specify what needs to be done.

Stream can be sequential or parallel. A parallel stream is especially useful if the computer the program is running on has a multicore CPU.

The main reason for using a stream is for its supports for sequential and parallel aggregate operations. It allows developer to traverse over a collection of elements and perform aggregate operations, pipeline two or more operations, perform parallel execution, and more.

16.1: Stream API

Stream API

- **Characteristics of Stream API**
  - Not a data structure
  - Designed for lambdas
  - Do not support indexed access
  - Can easily be output as arrays or Lists
  - Lazy
  - Parallelizable
  - Can be unbounded

June 11, 2015 | Proprietary and Confidential | - 5 -

IGATE Speed.Agility.Imagination

Stream API Characteristics

Not a data structures:
Streams have no storage. They carry values from a source through a pipeline of operations. They also never modify the underlying data structure.

Designed for lambdas
All Stream operations take lambdas as arguments

Do not support indexed access
You can ask for the first element, but not the second or third or last element.

Can easily be output as arrays or Lists
Simple syntax to build an array or List from a Stream

Lazy
Many Stream operations are postponed until it is known how much data is eventually needed E.g., if you do a 10-second-per-item operation on a 100 element list, then select the first entry, it takes 10 seconds, not 1000 seconds.

Parallelizable
If you designate a Stream as parallel, then operations on it will automatically be done concurrently, without having to write explicit multi-threading code

Can be unbounded
Unlike with collections, you can designate a generator function, and clients can consume entries as long as they want, with values being generated on the fly

16.1: Stream API

Stream Operations

- **Stream defines many operations, which can be grouped in two categories**
  - Intermediate operations
  - Terminal Operations
- **Stream operations that can be connected are called intermediate operations. They can be connected together because their return type is a Stream.**
- **Operations that close a stream pipeline are called terminal operations.**
- **Intermediate operations are "lazy"**

Intermediate operations **Terminal operation**

IGATE Speed.Agility.Imagination

Stream Operations

Some of the Stream methods perform intermediate operations and some others perform terminal operations.

Intermediate operations do not perform any processing until a terminal operation is invoked on the stream pipeline.

As shown in the slide, lets consider you want to travel by air which involves series of operations. Few of them are intermediate operations like travelling by taxi, take ticket from ticket counter, do check-in and finally you do terminal operations which is boarding the plane. The plane doesn't fly during the intermediate operations cause intermediate operations are lazy !

16.2: Working with Stream

Working with Stream: Step - 1

- **To perform a computation, first we need to define source of stream**
- **To create a stream source from values, use "of " method**

```
Stream<Integer> stream = Stream.of(10,20,30);
```

- **A stream can be obtained from sources like arrays or collections using "stream" method**
- **To obtain steam from array, use java.util.Arrays class**
  - stream()

```
Integer[] values = new Integer[] {10,20,30};
Stream<Integer> stream = Arrays.stream(values);
```

- **To obtain stream from collections, use java.util.Collection interface**
  - stream()
  - parallelStream()

June 11, 2015 | Proprietary and Confidential | - 7 -

IGATE Speed.Agility.Imagination

Working with Stream: Step – 1

As stream doesn't store data, we need to define the source to perform stream operations. This is done by either creating stream or obtaining stream from array/collections.

16.2: Working with Stream

Working with Stream: Step - 2

- **A stream pipeline consist of source, zero or more intermediate operations and a terminal operation**
- **A stream pipeline can be viewed as a query on the stream source**
- **Operations on stream are categories as:**
  - Filter
  - Map
  - Reduce
  - Search
  - Sort

June 11, 2015 | Proprietary and Confidential | - 8 -

IGATE Speed.Agility.Imagination

Working with Stream: Step – 2

After obtaining stream we can perform different operations on streams which are categorized into:

Filter
Map
Reduce

Kindly note a steam pipeline may consist of zero or more intermediate operations but it need only one terminal operation. A stream pipeline after terminal operation is closed automatically, hence its not available for further processing. A stream implementation may throw IllegalStateException if it detects that the stream is being reused.

16.2: Working with Stream

Stream Interface

- The Stream API consists of the types in the java.util.stream package
- The "Stream" interface is the most frequently used stream type
- A Stream can be used to transfer any type of objects
- Few important method of Stream Interface are:

concat	count
collect	filter
forEach	limit
map	max
min	of
reduce	Sorted
Intermediate	Terminal

June 11, 2015 | Proprietary and Confidential | - 9 -

IGATE Speed.Agility.Imagination

Stream API

Stream API introduced in Java 8 and provided under package java.util.stream. The Stream interface is most frequently used type in Stream API. The slide list few important method of the Stream Interface.

Methods such as filter, map and sorted are examples of methods that perform intermediate operations.

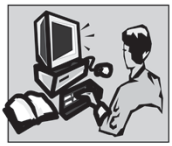
Methods such as count and forEach perform terminal operations.

An intermediate operation always returns a stream, where as the terminal operations performs an action.

16.2: Working with Stream

Demo

- **Execute the :**
  - BasicStream



IGATE
Speed.Agility.Imagination

16.3: Stream Operations

Mapping

- The Stream interface's map method maps each element of stream with the result of passing the element to a function.
- Map() takes a function (java.util.function.Function) as an argument to project the elements of a stream into another form.
- The function is applied to each element, "mapping" it into a new element.
- Syntax:

```
<R> Stream<R> map(java.util.function.Function<? super T, ? extends R> mapper)
```

- The map method returns a new Stream of elements whose type may be different from the type of the elements of the current stream.

IGATE Speed.Agility.Imagination

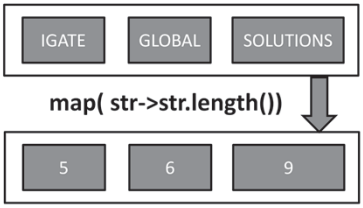
Mapping

Map method maps each element of stream with the result of passing the elements to a function.

16.3: Stream Operations

Mapping Example

```
List<String> words = Arrays.asList("IGATE","GLOBAL","SOLUTIONS");
words.stream().map(str->str.length()).forEach(System.out : : println);
```



June 11, 2015 | Proprietary and Confidential | - 12 -

IGATE Speed.Agility.Imagination

The mapping method returns a new stream of elements which may be different type. Here as shown in the slide example, the words stream is mapped based on length of each element. Therefore it returns a new stream of same size but the element type is integer.

Note: After any intermediate operation on stream, the processed items in resultant stream can be collected in separate collection if required.

```
List<String> words = Arrays.asList("IGATE","GLOBAL","SOLUTIONS");
List<Integer> counts = words.stream()
                                .map(str->str.length())
                                .collect(Collectors.toList());
```

16.3: Stream Operations

Filtering

- **There are several operations that can be used to filter elements from a stream:**

Operation	What ?
filter(Predicate)	Takes a predicate (java.util.function.Predicate) as an argument and returns a stream including all elements that match the given predicate
distinct	Returns a stream with unique elements (according to the implementation of equals for a stream element)
limit(n)	Returns a stream that is no longer than the given size n
skip(n)	Returns a stream with the first n number of elements discarded

June 11, 2015 | Proprietary and Confidential | - 13 -

IGATE Speed.Agility.Imagination

Filtering

A stream can be filtered out by using filter() method on Stream interface. Its takes predicate as argument which is usually a criteria for filtering, and returns new filtered stream containing elements which satisfies the criteria.

16.3: Stream Operations

Filtering Examples

- **filter(predicate)**

```
List<Integer>  listInt = Arrays.asList(11,3,44,5,66,33,44);
listInt.stream().filter(num -> num > 10).forEach(num->System.out.println(num));
```

11 44 66 33 44

- **distinct()**

```
List<Integer>  listInt = Arrays.asList(11,3,44,5,66,33,44);
listInt.stream().distinct().forEach(System.out : : println);
```

11 3 44 5 66 33

- **limit(size)**

```
List<Integer>  listInt = Arrays.asList(11,3,44,5,66,33,44);
listInt.stream().limit(4).forEach(System.out : : println);
```

11 3 44 5

June 11, 2015 | Proprietary and Confidential | - 14 -

IGATE Speed.Agility.Imagination

As shown in the slide examples, the filer operation filters a stream and reduces elements which less than 10.

The distinct() method removes duplicate from the stream.

The limit() method takes the new size of stream as argument and reduces the stream.

Note: All these operations are intermediate operations, whereas the forEach is terminal operation.

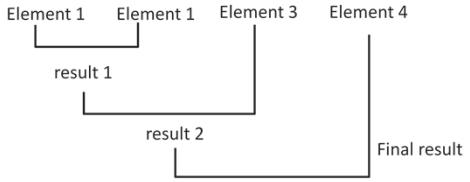
16.3: Stream Operations

Reducing

- **The reduce operation on streams, which repeatedly applies an operation on each element until a result is produced.**
- **It's often called a fold operation in functional programming**
- **Syntax:**

```
java.util.Optional<T> reduce(java.util.function.BinaryOperator<T> accumulator))
```

- **The reduce() method takes a BinaryOperator as argument and returns an Optional instance**



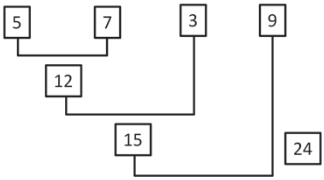
16.3: Stream Operations

Reducing Example

```
List<Integer> intList = Arrays.asList(5,7,3,9);
Optional<Integer> result = intList.stream().reduce((a,b)->a+b);
if(result.isPresent()) {
        System.out.println("Result:"+result.get());
}
```

Reduction of elements by adding them

Result: 24

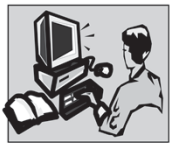


IGATE Speed.Agility.Imagination

16.3: Stream Operations

Demo

- **Execute the :**
  - StreamMap
  - StreamFiter
  - StreamReduce



IGATE
Speed.Agility.Imagination

16.5: Stream API

Lab

- **Lab 10: Lambda Expressions and Stream API**

IGATE
Speed.Agility.Imagination

Summary

- **In this lesson, you have learnt:**
  - Working with Stream API
  - Using Stream Operations on Collections

Review Question

- **Question 1 : Which of the following stream reduce call is valid to find max of given stream?**
  - **Option 1 :** stream.reduce((a,b)->a>b?a:b)
  - **Option 2 :** stream.max()
  - **Option 3 :** stream.map((a,b)->a>b)
- **Question 2 : ____ is a pipe for transferring data.**
- **Question 3 : Resource-intensive tasks can be done efficiently by using parallel stream.**
  - True/False

Core Java 8 with JAXB and Development Tools

Property Files

IGATE
Speed.Agility.Imagination

Lesson Objectives

- **After completing this lesson, participants will be able to**
  - Define property files and use them
  - Use properties and its methods
  - Define and use user specific properties

June 11, 2015 | Proprietary and Confidential | - 2 -

IGATE Speed.Agility.Imagination

This lesson covers the usage of Property files in your application. It explains how to create user specific property file. The important Properties class methods are also explained

Lesson outline:

17.1: What are Property Files?
17.2: Types of Property files
17.3: User defined Properties

17.1: What are Property Files?

Property Files

- **Property files**
  - have .properties extension
  - are used to store the configuration parameters
  - each parameter is stored as key/value pair
- **The java.util. Properties class**
  - represents a persistent set of key/value properties
  - are subclasses of Hashtables
  - provides methods to store and retrieve values from **properties files**. Example ->

```
#Properties File to the Test Application
password=tiger
username=scott
```

June 11, 2015 | Proprietary and Confidential | - 3 -

IGATE Speed.Agility.Imagination

Property Files:
Property files come with .properties extension and are used to store the configuration parameters or setting. Each parameter is stored as a pair of strings, one storing the name of the parameter (called the key), and the other storing the value.
java.util.Properties represents a persistent set of properties, ie a "key=value" pair. Each key and its corresponding value in the property list is a string. Properties are subclasses of Hashtables that can be backed to disk in human-readable format. You lookup by property name and get a value.
The Properties class provides methods to store and retrieve values from properties files.
The following are some of the points to be noted about Properties file:

Comments begin with #.

The keywords can contain dots and underscores but not spaces or =. You can use _ (underscore) in key names to represent a space.

The values can contain dots, underscores, spaces, and =.

17.2: Types of Property files

Categories of Property Files

- User Specific Properties
- System Properties

June 11, 2015 | Proprietary and Confidential | - 4 -

IGATE Speed.Agility.Imagination

Categories of Property Files:

User Specific Properties
These properties are part of the Application.properties containing a key value pair, which can be mentioned by the program in run. User-specific properties are generally used for configuring the application.
Our focus in this lesson will be on user specific property files.

System Properties
The Java platform itself uses a Properties object to maintain its own configuration. The System class maintains a Properties object that describes the configuration of the current working environment. System properties include information about the current user, the current version of the Java runtime, and the character used to separate components of a file path name.
You may read up on System properties in Appendix-A.

17.3: User Specific Properties

The java.util.Property Class

- **To manage properties, create instances of java.util.Properties class.**
- **This class provides methods for the following:**
  - Loading key/value pairs into a **Properties** object from a stream
  - Retrieving a value from its key
  - Listing the keys and their values
  - Saving the properties to a stream

June 11, 2015 | Proprietary and Confidential | - 5 -

IGATE Speed.Agility.Imagination

The java.util.Property Class
Some of the widely used methods of the java.util.Properties class:-
load
public synchronized void load(InputStream inStream) throws IOException : reads a property list (key and element pairs) from the input stream.
getProperty
public String getProperty(String key) : Searches for the property with the specified key in this property list. If the key is not found in this property list, the default property list, and its defaults, recursively, are then checked. The method returns null if the property is not found.
list
public void list(PrintStream out) : Prints this property list out to the specified output stream. This method is useful for debugging.
save
public synchronized void save(OutputStream out, String header) : Calls the store(OutputStream out, String header) method and suppresses IOExceptions that were thrown.

17.3: Setting Properties

Setting Properties

- **setProperty(String key, String value)**
  - Puts the key/value pair in the Properties object.
- **remove(Object key)**
  - Removes the key/value pair associated with key.

June 11, 2015 | Proprietary and Confidential | - 6 -

IGATE Speed.Agility.Imagination

Setting Properties:

A user's interaction with an application during its execution may impact property settings. These changes should be reflected in the Properties object so that they are saved when the application exits (and calls the store method).

The following methods change the properties in a Properties object:

setProperty(String key, String value)

Puts the key/value pair in the Properties object.

remove(Object key)

Removes the key/value pair associated with key.

Note: Some of the methods described above are defined in Hashtable, and thus, they accept key and value argument types other than String. Always use Strings for keys and values, even if the method allows other types. Also, do not invoke Hashtable.set or Hastable.setAll on Properties objects; always use Properties.setProperty.

17.3: Getting Property information

Getting Property Information

- contains(Object value)
- containsKey(Object key)
- getProperty(String key)
- getProperty(String key, String default)
- list(PrintStream s)
- list(PrintWriter w)
- elements()
- keys()
- propertyNames()
- stringPropertyNames()
- size()

June 11, 2015 | Proprietary and Confidential | - 7 -

IGATE Speed.Agility.Imagination

Getting Property Information

contains(Object value) , containsKey(Object key)

This returns TRUE if the value or the key is in the Properties object. Properties inherits these methods from Hashtable. Thus, they accept Object arguments, but only String values should be used.

getProperty(String key) , getProperty(String key, String default)

This returns the value for the specified property. The second version provides for a default value. If the key is not found, the default is returned.

list(PrintStream s) , list(PrintWriter w)

This writes all of the properties to the specified stream or writer. This is useful for debugging.

elements() , keys() , propertyNames()

This returns an Enumeration containing the keys or values (as indicated by the method name) contained in the Properties object. The keys method only returns the keys for the object itself; the propertyNames method returns the keys for default properties as well.

string PropertyNames()

This functions like propertyNames, but returns a Set<String>, and only returns names of properties where both key and value are strings. Note that the Set object is not backed by the Properties object, so changes in one do not affect the other.

size()

This returns the current number of key/value pairs.

17.3: Demo: User Specific Property

Demo: User Specific Properties

```
private static void saveProperties(Properties p) {
   try { OutputStream propsFile = new
FileOutputStream(fileName);
         p.store(propsFile, "Properties File to the Test Application");
         propsFile.close();
      } catch (IOException ioe) {… }
}
private static Properties loadProperties(String fileName) {
    Properties tempProp = new Properties();
      try { InputStream  propsFile = new FileInputStream(fileName);
         tempProp.load(propsFile);
         propsFile.close();
      } catch (IOException ioe) {… }
      return tempProp;    }
```

June 11, 2015 | Proprietary and Confidential | - 8 -

IGATE Speed.Agility.Imagination

Add the notes here.

17.3: Demo: User Specific Property

Demo: User Specific Properties

```
private static Properties createDefaultProperties() {
    Properties tempProp = new Properties();
        /* Database connection parameter properties are set */
    tempProp.setProperty("url",
            "jdbc:oracle:thin:@192.168.12.16:1521:oracle8i");

tempProp.setProperty("driver","oracle.jdbc.driver.OracleDriver");
    tempProp.setProperty("username", "trg1");
    tempProp.setProperty("password", "tiger");
    return tempProp;
}
private static void printProperties(Properties p, String s) {
    p.list(System.out);
}
```

IGATE Speed.Agility.Imagination

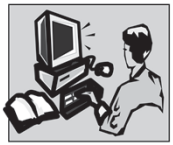
June 11, 2015 | Proprietary and Confidential | - 9 -

Add the notes here.

17.3: Demo: User Specific Property

Demo : Concept of properties

- **Execute the DatasourcePropertyfiles.java program**



IGATE
Speed.Agility.Imagination

Summary

- **In this lesson, you have learnt the following:**
  - What are Property files and their usage?
  - User specific properties

Add the notes here.

Review Question

- **Question 1: load(_____) throws IOException**
  - **Option 1 :** InputStream
  - **Option 2 :** OutputStream
- **Question 2: Is this a valid key value pair?**
  fruit apple
  - True/False.

Review Question

- **Question 3: If the property file contains**

```
fruits=apple,\mango
```

What will be the output of

```
System.out.println(p.getProperty("fruits"));
```

- **Where p is a properties object**
  - **Option 1 :** apple, mango
  - **Option 2 :** apple,\mango
  - **Option 3 :** null

Core Java 8 with JAXB and Development Tools

Java Database Connectivity (JDBC 4.0)

IGATE
Speed.Agility.Imagination

June 11, 2015 | Proprietary and Confidential | - 1 -

Lesson Objectives

- **After completing this lesson, participants will be able to**
  - Understand concept database connectivity architecture
  - Work with JDBC API 4.0
  - Access database through Java programs
  - Understand advance features of JDBC API

June 11, 2015 | Proprietary and Confidential | - 2 -

IGATE
Speed.Agility.Imagination

This lesson covers JDBC API, used to work with database.

Lesson outline:

18.1: Java Database Connectivity - Introduction
18.2: Database Connectivity Architecture
18.3: JDBC APIs
18.4: Database Access Steps
18.5: Calling database procedures/functions
18.6: Using Transaction
18.7: Best Practices

18.1: Java Database Connectivity

JDBC – an introduction

- **Java Database Connectivity (JDBC) is a standard SQL database access interface, providing uniform access to a wide range of relational databases.**
- **JDBC allows us to construct SQL statements and embed them inside Java API calls.**
- **JDBC provides different set of APIs to perform operations related to database; allows us to:**
  - Establish a connection with a database.
  - Send SQL statements.
  - Process the results

June 11, 2015 | Proprietary and Confidential | - 3 -

IGATE Speed.Agility.Imagination

What is JDBC?
JDBC is used to allow Java applications to connect to the database and perform different data manipulation operations such as insertion, modification, deletion, and so on.

18.1: Java Database Connectivity

JDBC Features

- **JDBC exhibits the following features:**
  - Java is a write once, run anywhere language.
  - Java based clients are thin clients.
  - It is suited for network centric models.
  - It provides a clean, simple, uniform vendor independent interface.
  - JDBC supports all the advanced features of latest SQL version
  - JDBC API provides a rich set of methods.

June 11, 2015 | Proprietary and Confidential | - 4 -

IGATE Speed.Agility.Imagination

Why JDBC?

JDBC Features:

With JDBC technology, businesses are not locked in any proprietary architecture, and can continue to use their installed databases and access information easily – even if it is stored on different database management systems.

The combination of the Java API and the JDBC API makes application development easy and economical.

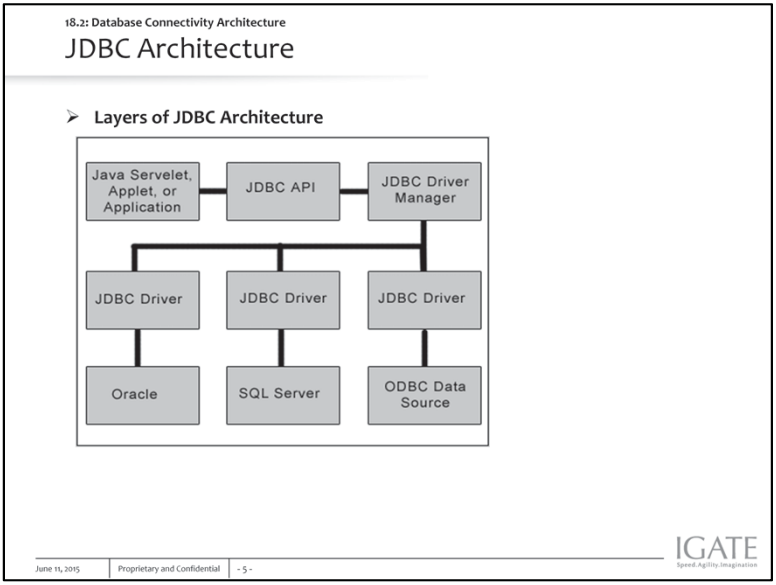
JDBC hides the complexity of many data access tasks, doing most of the "heavy lifting" for the programmer behind the scenes.

The JDBC API is simple to learn, easy to deploy, and inexpensive to maintain.

With the JDBC API, no configuration is required on the client side.

With a driver written in the Java programming language, all the information needed to make a connection is completely defined by the JDBC URL or by a DataSource object registered with a Java Naming and Directory Interface (JNDI) naming service.

Zero configuration for clients supports the network computing paradigm and centralizes software maintenance.



JDBC Architecture:

The JDBC Architecture can be classified as follows:

1. Type 1 – JDBC-ODBC Bridge
2. Type 2 – Java Native API
3. Type 3 – Java to Network Protocol
4. Type 4 – Java to Database Protocol

A JDBC driver translates standard JDBC calls into a network or database protocol or into a database library API call that facilitates communication with the database.

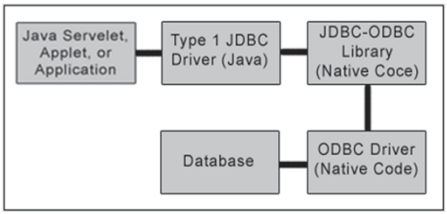
This translation layer provides JDBC applications with database independence.

If the back-end database changes, then only the JDBC driver needs to be replaced with few code modifications required. There are four distinct types of JDBC drivers.

18.2: Database Connectivity Architecture

JDBC Driver

- **Type 1 – JDBC-ODBC Bridge**



June 11, 2015 | Proprietary and Confidential | - 6 -

IGATE Speed.Agility.Imagination

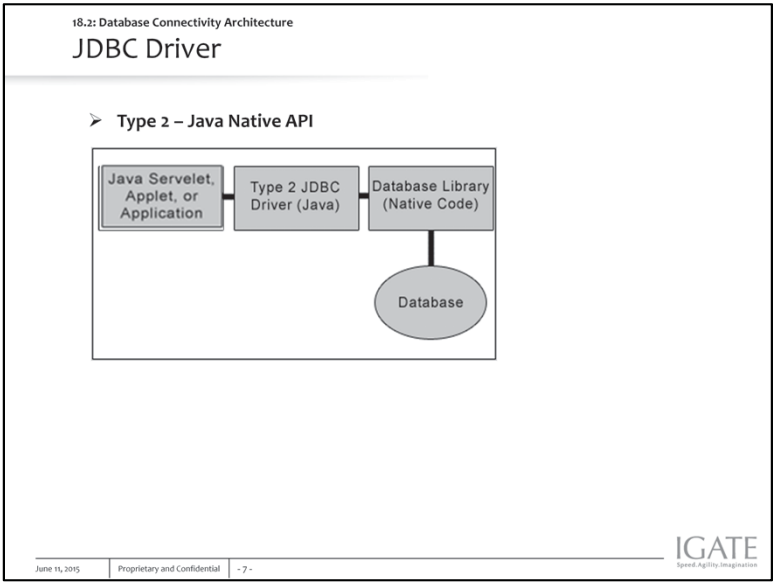
Type 1 JDBC-ODBC Bridge:

Type 1 drivers act as a "bridge" between JDBC and another database connectivity mechanism such as ODBC.

The JDBC- ODBC bridge provides JDBC access using most standard ODBC drivers.

This driver is included in the Java 2 SDK within the sun.jdbc.odbc package. In this driver the Java statements are converted to a JDBC statements.

JDBC statements call the ODBC by using the JDBC-ODBC Bridge. And finally the query is executed by the database. This driver has serious limitation for many applications.



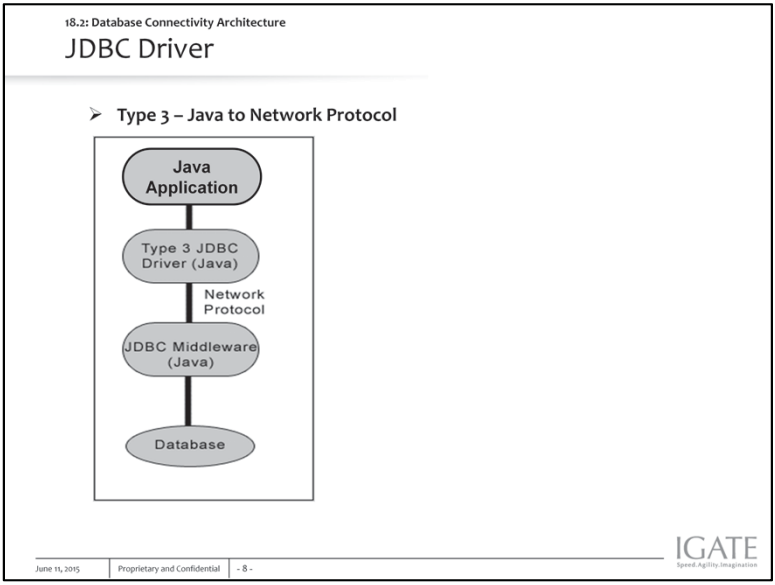
Type 2 Java to Native API:

Type 2 drivers use the Java Native Interface (JNI) to make calls to a local database library API.

This driver converts the JDBC calls into a database specific call for databases such as SQL, ORACLE, and so on. This driver communicates directly with the database server.

It requires some native code to connect to the database. Type 2 drivers are usually faster than Type 1 drivers.

Like Type 1 drivers, Type 2 drivers require native database client libraries to be installed and configured on the client machine.



Type 3 Java to Network Protocol Or All- Java Driver:

Type 3 drivers are pure Java drivers that use a proprietary network protocol to communicate with JDBC middleware on the server.

The middleware then translates the network protocol to database-specific function calls.

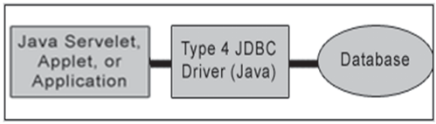
Type 3 drivers are the most flexible JDBC solution because they do not require native database libraries on the client and can connect to many different databases on the back end.

Type 3 drivers can be deployed over the Internet without client installation.

18.2: Database Connectivity Architecture

JDBC Driver

- **Type 4 – Java to Database Protocol**



June 11, 2015 | Proprietary and Confidential | - 9 -

IGATE Speed.Agility.Imagination

Type 4 Java to Database Protocol:

Type 4 drivers are pure Java drivers that implement a proprietary database protocol to communicate directly with the database.

Like Type 3 drivers, they do not require native database libraries and can be deployed over the Internet without client installation.

One drawback to Type 4 drivers is that they are database specific. Unlike Type 3 drivers, if your back-end database changes, you may save to purchase and deploy a new Type 4 driver (some Type 4 drivers are available free of charge from the database manufacturer).

However, since Type 4 drivers communicate directly with the database engine rather than through middleware or a native library, they are usually the fastest JDBC drivers available.

This driver directly converts the Java statements to SQL statements.

18.3: JDBC APIs

JDBC Packages

- **JDBC packages:**
  - java.sql.*
  - javax.sql.*

June 11, 2015 | Proprietary and Confidential | - 10 -

IGATE Speed.Agility.Imagination

JDBC APIs:

The JDBC API provides universal data access from the Java programming language.

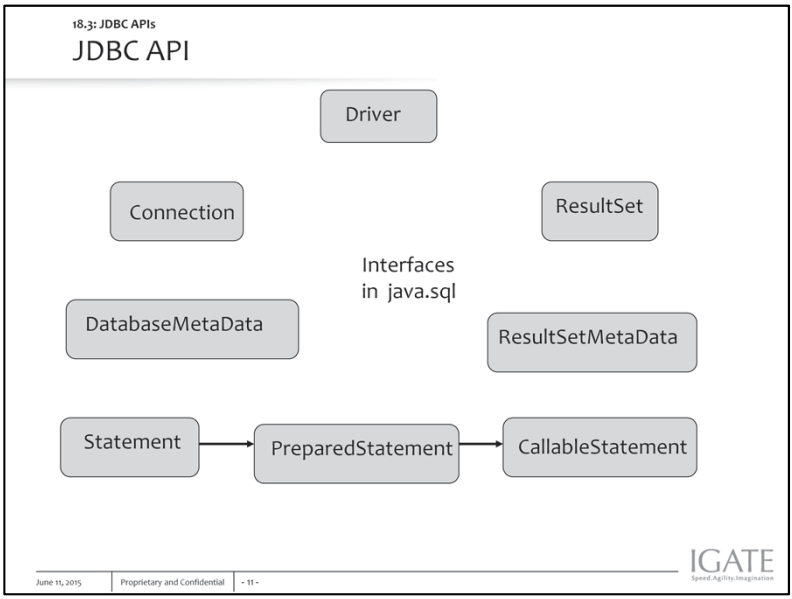
Using the JDBC 3.0 API, you can access virtually any data source, from relational databases to spreadsheets and flat files.

JDBC technology also provides a common base on which tools and alternate interfaces can be built.

The JDBC 3.0 API comprises of two packages:

1. java.sql package
2. javax.sql package

You automatically get both packages when you download the Java 2 Platform Standard Edition 5.0.



Java.sql package:

The **java.sql package** provides the API for accessing and processing data stored in a data source (usually a relational database) using the Java programming language.

This API includes a framework whereby different drivers can be installed dynamically to access different data sources.

Although the JDBC API is mainly geared to passing SQL statements to a database, it provides for reading and writing data from any data source with a tabular format.

The **reader/writer** facility, available through the **javax.sql.RowSet** group of interfaces, can be customized to use and update data from a spread sheet, flat file, or any other tabular data source.

Please note: Here we are not discussing about javax.sql package.

18.4: Database Access Steps

JDBC Database Access

- **Database Access takes you through the following steps:**
  - Import the packages
  - Register/load the driver
  - Establish the connection
  - Creating JDBC Statements
  - Executing the query
  - Closing resources

June 11, 2015 | Proprietary and Confidential | - 12 -

IGATE Speed.Agility.Imagination

13.4: Database Access Steps
- 13.4.1: Import the packages
- 13.4.2: Register the driver
- 13.4.3: Establishing the connection
- 13.4.4: Creating JDBC Statements
  - 13.4.4.1: Simple Statement
  - 13.4.4.2: Prepared Statement
  - 13.4.4.3: Callable Statement
- 13.4.5: Getting Data from a Table
- 13.4.6: Insert data into Table
- 13.4.7: Update table data

18.4: Database Access Steps

JDBC Database Access: Step-1

- **Step 1:** Import the java.sql and javax.sql packages
  - These packages provides the API for accessing and processing data stored in a data source.
  - They include:
    - import java.sql.*;
    - import javax.sql.*;

IGATE Speed.Agility.Imagination

Import Packages:
The first step in accessing the data from database using JDBC APIs is importing the packages java.sql and javax.sql.
These packages provides the set of APIs which are used in accessing the database like Connection, Statement, and so on.

18.4: Database Access Steps

JDBC Database Access: Step-2

- **Step 2:** Register/load the driver
  - Class.forName("oracle.jdbc.driver.OracleDriver");
    OR
  - DriverManager.registerDriver (new oracle.jdbc.driver.OracleDriver());
- **DriverManager class is used to load and register appropriate database specific driver.**
- **The registerDriver() method is used to register driver with DriverManager**
- **In JDBC 4.0, this step is not required, as when getConnection() method is called, the DriverManager will attempt to locate a suitable driver**

IGATE Speed.Agility.Imagination

Register Driver:

The second step is to register/load the driver for the respective database. There are two ways of loading the driver:

By using Class.forName() method

By calling DriverManager's registerDriver() method

Using Class.forName() you can load any class while DriverManger.registerDriver() is specific to JDBC driver class.

Note: In JDBC 4.0, we don't need the Class.forName() line. We can simply call getConnection() to get the database connection.

18.4: Database Access Steps

JDBC Database Access: Step-3

- **Step 3:** Establish the connection with the database using registered driver
  - String url = "jdbc:oracle:thin:@hostname:1521:database";
  - Connection conn = DriverManager.getConnection (url, "scott", "tiger");
- **Once driver is loaded, Connection object is used to establish a connection with database.**

June 11, 2015 | Proprietary and Confidential | - 15 -

IGATE Speed.Agility.Imagination

Establish the Connection:

The third step is to establish a connection with the database. There is a method DriverManager.getConnection() which returns Connection object.

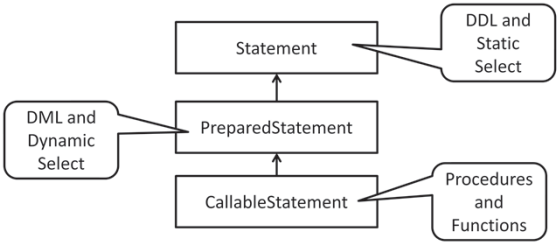
This method take three arguments,

URL: This contains the information about host on which database server, database name, and the port used for communication.
Username: database userid
Password: database password

18.5: Database Access Steps

JDBC Database Access: Step-4

- **Once connection is established with database, statement object can be used to execute different types of SQL queries**
- **JDBC API support different statement interfaces depending upon type of query to be executed**



June 11, 2015 | Proprietary and Confidential | - 16 -

IGATE Speed.Agility.Imagination

Using Statements:

java.sql.Statement interface in JDBC enables to send the SQL queries to database and retrieve data. Statement interface is suitable for executing DDL queries and Select queries which has no input.

Statement interface is further extended as PreparedStatement, which is recommended for DML and select queries that involves input parameters.

PreparedStatement is in turn extended as CallableStatement, which is suitable for calling database stored procedures and functions.

18.4: Database Access Steps

JDBC Database Access: Step-4

- **Step 4**: Create Statement
  - Statement:
    - Statement st=conn.createStatement();
  - PreparedStatement:
    - PreparedStatement pst=conn.prepareStatement("SELECT* FROM emp WHERE eno=?");
      pst.setInt(100);
  - CallableStatement:
    - CallableStatement cs=conn.prepareCall("{ call add() }") ;
- **Once statement object is created, use one of the following method to execute the query**

Operation	Method
Select	ResultSet executeQuery(String query)
DML	int executeUpdate(String query)
DDL	boolean execute(String query)

June 11, 2015 | Proprietary and Confidential | - 17 -

IGATE Speed.Agility.Imagination

Create Statement:

The fourth step is to create a statement which indicates the details that you want to access from database.

There are three types of statements that you can create:

Statement:

Statement object is created by calling createStatement() method on Connection object. The query is not associated with Statement object at the time of statement creation, rather it needs to be specified at the time of execution of the statement.

2. PreparedStatement:

PreparedStatement is used when you want to create parameterized query. Each parameter is represented by symbol "?" and set by setXXX() methods of PreparedStatement. The above slide shows a simple example to get more clear idea.

3. CallableStatement:

When you want to call any stored procedure from database from Java application, the CallableStatement will help you in doing that. The above example shows, how you can call add() procedure using CallableStatement.

18.4: Database Access Steps

JDBC Database Access: Step-5 (Querying Data)

- **Step 5**: Retrieve data from table
  - Statement:

```
Statement st=conn.createStatement();
ResultSet rs=st.executeQuery("SELECT * FROM emp");
while(rs.next()){
System.out.println("Emo No = "+rs.getInt("eno"));
System.out.println("Emo Name = "+rs.getString("ename"));
```

- Output:
  - Display the number and name of all employees

June 11, 2015 | Proprietary and Confidential | - 18 -

IGATE Speed.Agility.Imagination

Retrieve data from table:

Once your statement object is ready, you need to execute that statement to get the records from the database.

The executeQuery() method of the Statement returns the ResultSet object which represent the front end table of database records.

Further, you can traverse/iterate through ResultSet to retrieve the records one by one.

Use getXXX() methods to retrieve each data field value, for example rs.getInt("eno") will retrieve the employee number of the current record from rs.

Types of ResultSet:

Resultset contains results of the SQL query. There are three basic types of resultset.

Forward-only

As the name suggests, this type can only move forward and are non-scrollable.

Scroll-insensitive

This type is scrollable which means the cursor can move in any direction. It is insensitive which means any change to the database will not show change in the resultset while it opens.

Scroll-sensitive

This type allows cursor to move in any direction and also propagates the changes done to the database.

18.4: Database Access Steps

JDBC Database Access: Step-5 (inserting data)

- **Step 5:** Insert data into table
  - PreparedStatement:

```
String query = "INSERT INTO emp VALUES(?,?,?)"
PreparedStatement st=conn.prepareStatement(query);
st.setInt(1, 110);
st.setString(2, "xyz");
St.setString(3, "Pune");
int rec = st.executeUpdate();
System.out.println(rec + " record is inserted"));
```

June 11, 2015 | Proprietary and Confidential | - 19 -

IGATE Speed.Agility.Imagination

Insert data into table:

PreparedStatement is used to execute dynamic SQL queries with values being changed during runtime. These statements are pre-compiled and hence are faster as compared to Statement where every call need to be parsed and compiled before it is executed.

The "?" in query string are called as input parameters. These parameters indicates the value is not specified at compile time. A value for every "?" should be set before executing query. To set a value of parameter, setXXX() methods are used which accepts position of input parameter and value to replace before query execution.

The executeUpdate() method is used to insert and update records of the database.

This method returns int type value indicating the number of records affected in the database.

18.4: Database Access Steps

JDBC Database Access: Step-5 (updating data)

- **Step 5**: Update table data
  - Statement:

```
String query = "update emp set ecity=? where eno<?"
PreparedStatement st=conn.prepareStatement(query);
st.setString(1,"Mumbai");
st.setInt(2,1000);
int res = st.executeUpdate);
System.out.println(red + " records updated"));
```

June 11, 2015 | Proprietary and Confidential | - 20 -

IGATE Speed.Agility.Imagination

Update Table Data:

The executeUpdate() method is used to update records from the database table.

This method returns integer type value indicating the number of records affected in the database table.

18.4: Database Access Steps

JDBC Database Access: Step-5 (deleting data)

- **Step 5**: Delete table data
  - PreparedStatement:

```
String query = "delete from emp where eno<?"
PreparedStatement st=conn.prepareStatement(query);
st.setInt(2,1000);
int res = st.executeUpdate);
System.out.println(red + " records deleted"));
```

June 11, 2015 | Proprietary and Confidential | - 21 -

IGATE Speed.Agility.Imagination

Update Table Data:

The executeUpdate() method is also used to delete records from the database table.

This method returns integer type value indicating the number of records affected in the database table.

18.4: Database Access Steps

JDBC Database Access: Step-6

- **Step 6:** Closing resources
  - Once done with data access following resources needs to be closed in order to free the underlying processes and release the memory
  - resultSet.close();
  - statement.close();
  - connection.close();

June 11, 2015 | Proprietary and Confidential | - 22 -

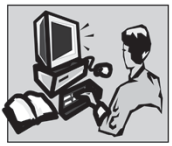
IGATE Speed.Agility.Imagination

Closing resources:
The last step in database access is closing used resources in program. Closing statements and result sets is required in order to free the underlying processes and memory.
Connection object is also required to be closed. Failure in closing connection object results in database server running out of connections.
All of these interfaces provides close() method, which is used to close the respective resource.

18.4: Database Access Steps

Demo

- **Execute the :**
  - Select.java
  - Insert.java
  - Delete.java
  - PreparedStat.java programs



IGATE
Speed.Agility.Imagination

18.5: Statements

Calling Stored Procedures and Functions

- **CallableStatement is used to invoke either procedure or function**
- **CallableStatement interface extends PreparedStatement so as to support input and output parameters**
- **Steps to call procedure/function:**
  - Create callable statement object
  - Call setXXX() method to set IN parameters
  - Call registerOutParameter() method to register OUT parameters/function return value
  - Call execute() to invoke the procedure/function
  - Call getXXX() method to retrieve results from OUT parameters/function return value

June 11, 2015 | Proprietary and Confidential | - 24 -

IGATE Speed.Agility.Imagination

Callable Statements:

PreparedStatement is in turn extended as CallableStatement, which is suitable for calling database stored procedures and functions.

18.5: Callable Statement

Calling Stored Procedures

```
String query = " { call getStudentName (?, ?) } " ;
CallableStatement st  = connection.prepareCall(query);
st.setInt(1,121);
st.registerOutParameter(2,java.sql.Types.VARCHAR);
st.execute();
String studName = st.getString(2);
System.out.println(studName);
```

Procedure which accepts roll number and returns name of the student

Registering OUT parameter "name" which is in VARCHAR format

Setting IN parameter "roll number" of student

Retrieving value of OUT parameter

June 11, 2015 | Proprietary and Confidential | - 25 -

IGATE Speed.Agility.Imagination

Callable Statements:

The above example shows how to use callable statement to invoke stored procedure.

18.5: Callable Statement

Calling Stored Functions

```
String query = " { ? = call getStudentName (?) } " ;
CallableStatement st  = connection.prepareCall(query);
st.setInt(2,121);
st.registerOutParameter(1,java.sql.Types.VARCHAR);
st.execute();
String studName = st.getString(1);
System.out.println(studName);
```

Function which accepts roll number returns name of the student

Registering return parameter "name" which is in VARCHAR format

Setting IN parameter "roll number" of student

Retrieving return value of function

June 11, 2015 | Proprietary and Confidential | - 26 -

IGATE Speed.Agility.Imagination

Callable Statements:

The above example shows how to use callable statement to invoke stored function.

18.6: Using Transaction

Transaction

- A transaction is a set of one or more statements that are executed together as a unit, so either all of the statements are executed, or none of the statements is executed.
- Java application uses Java Transaction API(JTA) to manage the transactions.

June 11, 2015 | Proprietary and Confidential | - 27 -

IGATE Speed.Agility.Imagination

Using Transaction:

Transaction management in Java application is normally done using methods of the Connection interface.

Listed below are the Connection methods for transaction management:

setAutoCommit()
commit()
rollback()
setSavePoint()
releaseSavePoint()

18.6: Using Transaction

Transaction (cntd…)

- **By default, a connection object is in** auto-commit **mode ie conn.setAutoCommit(true);**
- **Disable the auto-commit mode to allow two or more statements to be grouped into a transaction:**
- **Example: con.setAutoCommit(false);**

```
try{
    con.setAutoCommit(false);
  sql-statement-1;
    sql-statement-1;
    ….
  conn.commit();
}
```

Transactional unit

commit the transaction

June 11, 2015 | Proprietary and Confidential | - 28 -

IGATE Speed.Agility.Imagination

Disabling Auto-commit Mode:

When Connection is created, by default, transaction mode in Java application is true.

It means that as soon as the statement gets executed, it reflects the changes permanently in the database. This might violate the atomicity property of the transaction. Due to this reason before transaction start, you must set the auto-commit to false.

Once you disable auto-commit, it will not reflect the effect of queries permanently in the database until you call conn.commit() method explicitly.

In the code snippet above, auto-commit mode is disabled for the connection conn. This means that the enclosed sql statements are committed together when the commit method is called.

Whenever the commit method is called (either automatically when auto-commit mode is enabled or explicitly when it is disabled), all changes resulting from statements in the transaction are made permanent.

18.6: Using Transaction

Transaction (cntd…)

- **The** rollback() **method plays an important role in preserving data integrity in the transaction.**
- **When you want to undo the changes of half done transaction due to SQLException:**

```
try {
        conn.setAutoCommit(false);
          // perform transactions
        conn.commit();

      } catch (SQLException e) {
        conn .rollback() ;
      } finally {
        con.setAutoCommit(true);
      }
```

June 11, 2015 | Proprietary and Confidential | - 29 -

IGATE Speed.Agility.Imagination

When do you call the Rollback Method:

Calling the rollback method aborts a transaction and returns any values that were modified to their previous values. If you are trying to execute one or more statements in a transaction and get an SQLException, you should call the rollback method to abort the transaction and start the transaction all over again. That is the only way to be sure of what has been committed and what has not been committed.

Catching an SQLException tells you that something is wrong. However, it does not tell you what was or was not committed. Since you cannot count on the fact that nothing was committed, calling the method rollback is the only way to be sure.

The example in the above slide, demonstrates a transaction and includes a catch block that invokes the rollback method. In this particular situation, it is not really necessary to call rollback and we do it mainly to illustrate how it is done.

However, if the application continued and used the results of the transaction, it would be necessary to include a call to rollback in the catch block in order to protect against using possibly incorrect data.

18.6: Using Transaction

Transaction (cntd…)

- Savepoint **is marking to roll back the transaction till the particular statement in the transaction.**
- **You can set the** multiple save points **in the transaction.**
- **Example of Setting Savepoint:**

```
Savepoint svpt1 = conn.setSavepoint("SAVEPOINT_1");
……
con.rollback(svpt1);
```

- **Example of Releasing Savepoint:**

```
conn.releaseSavepoint(svpt1);
```

IGATE Speed.Agility.Imagination

June 11, 2015 | Proprietary and Confidential | - 30 -

Setting and Releasing the Savepoint:

The JDBC 3.0 API adds the method Connection.setSavepoint, which sets a savepoint within the current transaction.

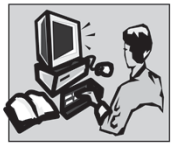
The Connection.rollback() method has been overloaded to take a savepoint argument.

The method Connection.releaseSavepoint() takes a Savepoint object as a parameter and removes it from the current transaction. Once a savepoint has been released, attempting to reference it in a rollback operation causes an SQLException to be thrown.

Any savepoints that have been created in a transaction are automatically released and become invalid when the transaction is committed, or when the entire transaction is rolled back. Rolling a transaction back to a savepoint automatically releases and makes invalid any other savepoints that were created after the savepoint.

Demo : Transaction

- **Execute the Transaction.java program**



IGATE
Speed.Agility.Imagination

18.7: JDBC Best Practices

Best Practices

- **Some of the best practices in JDBC:**
  - Selection of Driver
  - Close resources as soon as you're done with them
  - Turn-Off Auto-Commit – group updates into a transaction
  - Business identifiers as a String instead of number
  - Do not perform database tasks in code
  - Use JDBC's PreparedStatement instead of Statement when possible

June 11, 2015 | Proprietary and Confidential | - 32 -

IGATE Speed.Agility.Imagination

JDBC Best Practices:

Following are some of the best practices used in JDBC:

Selection of Driver

Select a certified, high performance type 2 (Thin) JDBC driver and use the latest drivers release.

Close resources as soon as you are done with them (in finally)

For example: Statements, Connections, Resultsets, and so on.

Failure to do so will eventually cause the application to "hang", and fail to respond to user actions.

Turn-Off Auto-Commit

It is best practice to execute the group of the statement together which are the part of the same transaction. It helps to avoid the overhead of data inconsistency.

JDBC Best Practices:

4. Business identifiers as a String instead of number

Many problem domains use numbers as business identifiers. Credit card numbers, bank account numbers, and the like, are often simply that - numbers. *It should always be kept in mind, however, that such items are primarily identifiers, and their numeric character is usually completely secondary*. Their primary function is to identify items in the problem domain, not to represent quantities of any sort.

For example: It almost never makes sense to operate on numeric business identifiers as true numbers - adding two account numbers, or multiplying an account number by -1, are meaningless operations. That is, one can strongly argue that modeling an account number as a Integer is inappropriate, simply because it does not behave as an Integer.

Furthermore, new business rules can occasionally force numeric business identifiers to be abandoned in favor of alphanumeric ones. It seems prudent to treat the content of such a business identifier as a business rule, subject to change. As usual, a program should minimize the ripple effects of such changes.

In addition, Strings can contain leading zeros, while numeric fields will remove them.

5. Do not perform database tasks in code

Databases are a mature technology, and they should be used to do as much work as possible. Do not do the following in code, if it can be done in SQL instead:

ordering (ORDER BY)
filtering based on criteria (WHERE)
joining tables (WHERE, JOIN)
summarizing (GROUP BY, COUNT, AVG, STDDEV)

Any corresponding task implemented entirely in code would very likely:

be *much* less robust and efficient
take longer to implement
require more maintenance effort

6. Use JDBC's PreparedStatement instead of Statement when possible for the following reasons:

It is in general more secure. When a Statement is constructed dynamically from user input, it is vulnerable to SQL injection attacks. PreparedStatement is not vulnerable in this way.

There is usually no need to worry about escaping special characters if repeated compilation is avoided, its performance is usually better.

In general, it seems safest to use a Statement only when the SQL is of fixed, known form, with no parameters

18.8: Java Database Connectivity

Lab : JDBC

- **Lab 11: Property Files and JDBC 4.0**

IGATE
Speed.Agility.Imagination

Summary

- **In this lesson, you have learnt:**
  - Establishing the connection with database to perform database operations
  - Different types of statement creation
  - Different ways of executing statements
  - Transaction management using JDBC APIs
  - Use of connection pooling to increase the performance of application
  - And best practices for JDBC applications

Review Question

- **Question 1 : Which Statement is used when you want to pass parameter to the query?**
  - **Option 1 :** Statement
  - **Option 2 :** PreparedStatement
  - **Option 3 :** CallableStatement
- **Question 2 : ____ method is best suited to execute DDL Queries.**
- **Question 3 : By default, a connection object is in auto-commit mode**
  - True/False

Core Java 8 with JAXB and Development Tools

Introduction to Layered Architecture

IGATE
Speed.Agility.Imagination

Lesson Objectives

- **After completing this lesson, participants will be able to**
  - Understand the concept of Layered Architecture
  - Implement layers in Java applications
  - Testing layered classes
  - Advanced testing concepts

This lesson covers Layered architecture and advanced testing concepts.

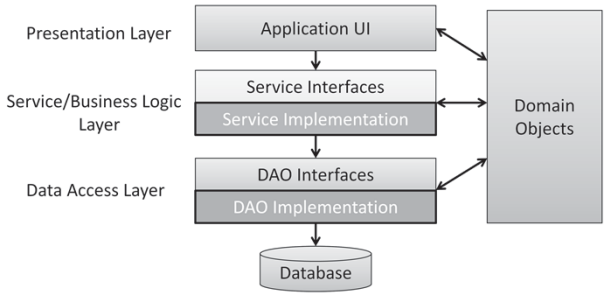
Lesson outline:

19.1: Introduction
19.2: Layered Architecture

19.1: Introduction

What is Layered Architecture?

- **Layered architecture is one of the architectural pattern based on call-and-return style**
- **In layered architecture, business rules, behavior, and data are obtained and manipulated, based on activity via the user interface.**
- **Layered architecture provides a clean separation between the business implementation, presentation and data-access logic.**



June 11, 2015 | Proprietary and Confidential | - 3 -

IGATE Speed.Agility.Imagination

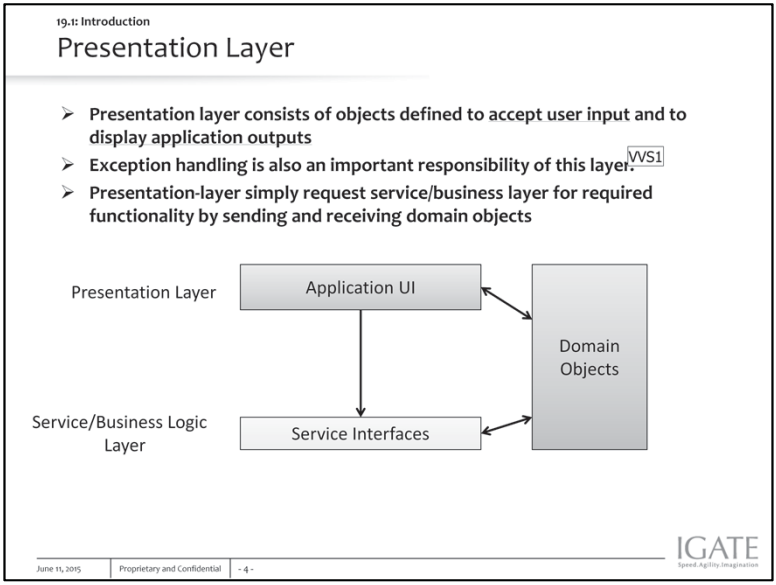
What is Layered Architecture?

Layering partitions the functionality of an application into separate layers that are stacked vertically. As shown in the figure above, layered architecture enables developer to make changes on one layer without having any side effects on others.

Why Layered Architecture?

Object technology encouraged the abstraction and reuse of not only presentation logic but also business processes and data. Therefore, decoupling application logic from application presentation is encouraged.

With the explosion of the Internet and related technologies, requirements to scale, rapidly develop, deploy, and react to business changes have made the existence of layered application architecture imperative.

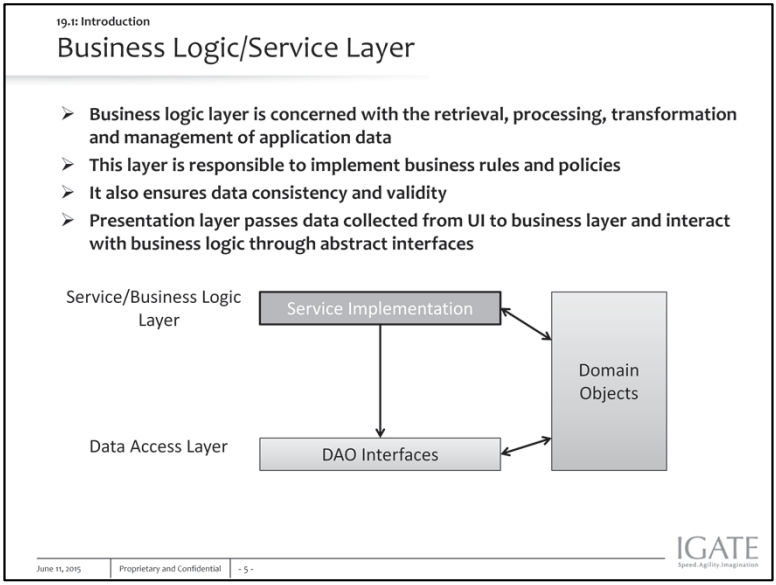


Presentation Layer:

The presentation layer contains the components that implement and display the user interface and manage user interaction. This layer includes controls for user input and display,

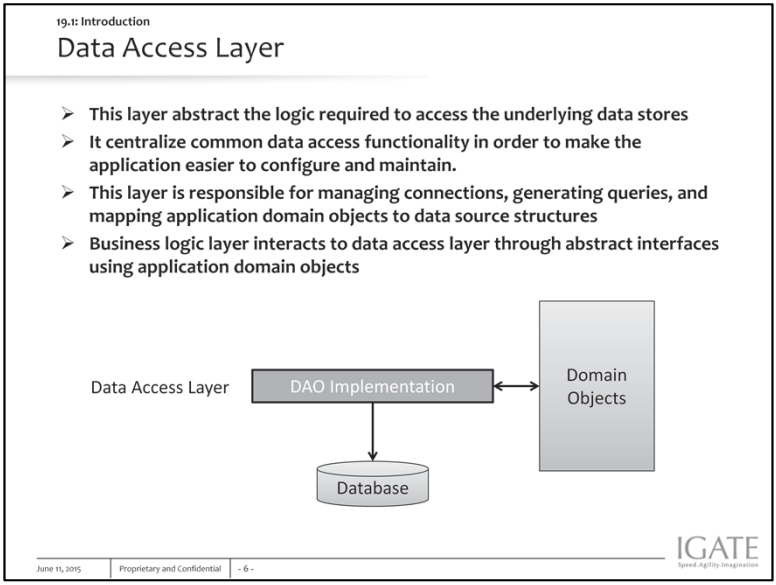
Slide 4

VVS1 Needs reframe.
Vinod V Satpute, 4/23/2015



Business Logic/Service Layer:

After the presentation layer collect the required data from the user and pass it to the business layer, the application can use this data to perform a business process. Use a business layer to centralize common business logic functions and promote reuse.



Data Access Layer

The data access layer should hide the details of data source access. It should be responsible for managing connections, generating queries, and mapping application domain objects to data source structures.

Consumers (Business logic layer) of the data access layer interact through abstract interfaces using application domain objects.

19.1: Introduction

Data Transfer Objects

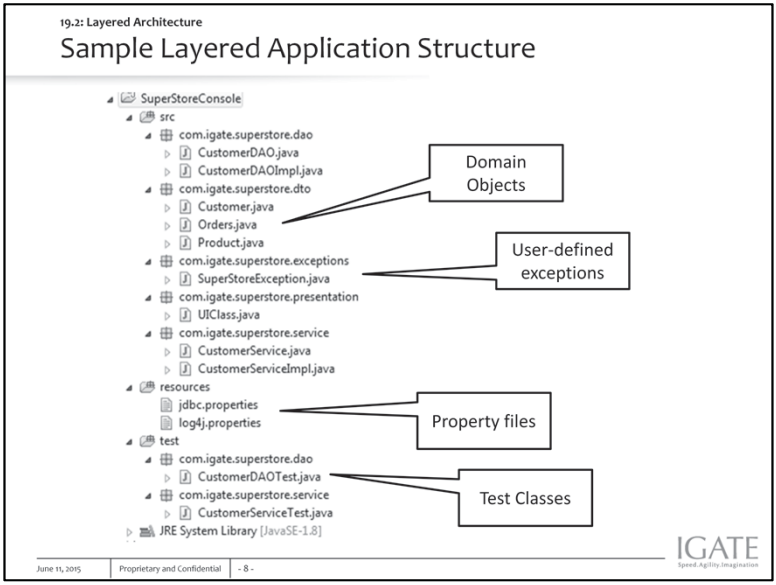
- **Data transfer objects (DTO) or Value Objects (VO) encapsulates business data necessary to represent real world elements, such as Customers or Orders**
- **These object are POJO's to store data values and expose them through properties**
- **They contain and manage business data used by the entire application**

Data Transfer Objects

IGATE
Speed.Agility.Imagination

Data Transfer Objects

DTO's are simply POJO classes that represent application real world entities. These objects are meant to share data between layers of application. These object contains data values and expose them through properties.



Designing Layered Application:

Create separate packages for all the layers including presentation, business logic and data access layer. All POJO based domain object needs to be stored in separate package.

Application specific exceptions must be separated from layered classes as shown in the slide.

Ensure the test classes must be written for DAO layers. Many business processes involve multiple steps that must be performed in the correct order. To ensure correctness of such business process, its better to test business layer classes too.

Database specific properties like username, password, URL etc. must be stored in separate property file. All property files must be store in resources folder of application.

Lab

- **Lab 12: Introduction to Layered Architecture**

IGATE
Speed.Agility.Imagination

Summary

- **In this lesson, you have learnt:**
  - Layered architecture for Java applications

Review Question

- **Question 1: _____ layer abstract the logic required to access the underlying data stores**
  - Option 1: Service
  - Option 2: Data Access
  - Option 3: Presentation
- **Question 2: Layered architecture is one of the architectural pattern based on call-wait-process pattern style**
  - True / False

Core Java 8 with JAXB and Development Tools

Advanced Testing Concepts

IGATE
Speed.Agility.Imagination

Lesson Objectives

- **After completing this lesson, participants will be able to**
  - Understand advanced testing concepts
  - Work with test suites
  - Implement parameterized tests and mocking concepts

This lesson covers advanced testing concepts.

Lesson outline:

20.1: Advanced Testing Concetps
20.2: Best practices

20.1: Advanced Testing Concepts

Composing Test into Test Suites

- **A testsuite comprises of multiple tests and is a convenient way to group the tests, which are related.**
- **It also helps in specifying the order for executing the tests.**
- **JUnit provides the following:**
  - org.junit.runners.Suite class : It runs a group of test cases.
  - @RunWith : It specifies runner class to run the annotated class.
  - @Suite.SuiteClasses : It specifies an array of test classes for the Suite.Class to run.
    - The annotated class should be an empty class but may contain initialization and cleanup code.

June 11, 2015 | Proprietary and Confidential | - 3 -

IGATE Speed.Agility.Imagination

Testing with JUnit – Test Suites:
Composing Test into Test Suites:
In a software development environment, a collection of test cases that test a software program is called a test suite. The tests in the test suite are normally related. For example: All the tests are testing for mathematical functionality. The test suite runs a collection of test cases. Prior to this version, you create a test suite using an instance of junit.framework.TestSuite. So the code looks as follows:

```
public static TestSuite()
{
  TestSuite RunTests = new TestSuite();
  RunTests.addTest(new MyTest("testFirstMethod"));
  RunTests.addTest(new MyTest("testSecondMethod"));
  return RunTests;
 }
```

However, the current version of JUnit encourages you to make use of annotations that have been introduced to build a Test Suite.
JUnit provides you with:

org.junit.runners.Suite : This class runs a group of test classes. Using this class as a runner lets you manually build a suite containing tests from several classes. To use it, annotate a class with @RunWith and @SuiteClasses.

20.1: Advanced Testing Concepts

Composing Test into Test Suites

- **Example:**

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
@RunWith(Suite.class)
@Suite.SuiteClasses({ TestCalAdd.class, TestCalSubtract.class,
TestCalMultiply.class, TestCalDivide.class })
public class CalSuite {
// the class remains completely empty,
// being used only as a holder for the above annotations
}
```

June 11, 2015 | Proprietary and Confidential | - 4 -

IGATE Speed.Agility.Imagination

Testing with JUnit – Test Suites:
Composing Test into Test Suites:
JUnit provides you with (contd.):

@RunWith : JUnit invokes the class it references to run the tests when this annotation is used instead of the runner that is built into JUnit. In JUnit4.x, suites are built using @RunWith and a custom runner named Suite. The @RunWith is telling JUnit to use org.junit.runner.Suite class. This runner allows you to manually build suite containing tests from many classes. All classes are defined in @Suite.SuiteClasses

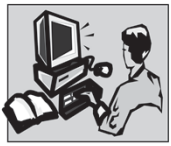
@Suite.SuiteClasses : This annotation is used to specify an array of test classes for the runner.

The CalSuite class is simply a place holder for the suite annotations. The @RunWith is the annotation which tells the JUnit runner to use the org.junit.runner.Suite class for running a particular class. The @Suite annotation instructs the suite runner about the test classes which have to be included in this suite and the sequence in which they should be introduced.

20.1: Advanced Testing Concepts

Demo

- **Demo on:**
  - Composing tests into Test Suites
    - TestPersonSuite.java



IGATE
Speed.Agility.Imagination

Note:
TestPersonSuite.java
In this demo example, the three classes, namely TestPerson, TestPerson2, and TestPersonFixture, have been put together for execution. The class itself does not have any methods for testing.

20.1: Advanced Testing Concepts

Reusing Tests

- **Parameterized tests allow you to run the same test with different data.**
- **To specify parameterized tests:**
  - Annotate class with @RunWith(Parameterized.class).
  - Add a public static method that returns a Collection of data.
    - Each element of the collection must be an Array of the various parameters used for the test.
  - Add a public constructor that uses the parameters.

June 11, 2015 | Proprietary and Confidential | - 6 -

IGATE Speed.Agility.Imagination

Testing with JUnit – Parameterized Tests:
Reusing Tests:
One of the significant features in JUnit4.x is the ability to run parameterized tests. This feature allows you to run the same test with different data. Essentially it means that you create a generic test, and run it multiple times with different parameters.
To create parameterized tests, use the specification as given on the slide.

20.1: Advanced Testing Concepts

Reusing Tests

- **Example:**

```
@RunWith(Parameterized.class)
public class SomethingTest {
@Parameters
public static Collection<Object[]> data() { …. }
public SomethingTest()
{…..}
@Test
public void testValue()
{…..}
}
```

June 11, 2015 | Proprietary and Confidential | - 7 -

IGATE Speed.Agility.Imagination

Testing with JUnit – Parameterized Tests:
Reusing Tests:
As per the syntax given on the slide, to create parameterized tests annotate:

The class with @RunWith
The method, which returns the collection to be annotated, with @Parameters

20.1: Advanced Testing Concepts

Testing in Isolation

- **Unit Testing of any method should be ideally done in isolation from other methods.**
- **For testing in isolation, you need to be independent of expensive resources.**
- **Use mock objects to perform testing in isolation.**
- **Mock object is created to represent an object that your code will be collaborating with.**

June 11, 2015 | Proprietary and Confidential | - 8 -

IGATE Speed.Agility.Imagination

Testing with JUnit – Isolated Testing:
Testing in Isolation:
Unit testing any method should ideally be done in isolation from other methods and it is certainly a nice objective. However, testing in isolation can get difficult in certain scenarios.

For example: The business logic of an application is built into servlet and without running the server you cannot use the functionality.

Hence you need an object that can be used in a test instead of an expensive resource or a difficult resource. That is instead of using a real database, we can use a mock object representing the database. The usage of mock objects allows you to unit test at a fine grained level by allowing you to write unit tests for all methods.
Hence while defining a mock object we can say, “a mock object is created to represent an object that your code will be collaborating with”.

20.1: Advanced Testing Concepts

Advantages of Using Mock Objects

- **There are obvious advantages of using mock objects:**
  - You get the ability to test code that is not yet written.
  - They help teams to unit test one part of the code independently.
  - They help to write focused tests that will test only a single method.
  - They are helpful when the application integrates with expensive external resources.

June 11, 2015 | Proprietary and Confidential | - 9 -

IGATE Speed.Agility.Imagination

Testing with JUnit – Isolated Testing:
Advantages of Using Mock Objects:
There are some noticeable benefits of using mock objects:

You can test the code which is not yet written but at the minimum an interface should be available to work with.
Testing in isolation helps teams to test one part of the code independently without waiting for all other parts of the code.
Also you can write focus tests that test only a single method without side effects resulting from other objects that are called from the method.
Small focused tests are easy to understand and they do not break when other parts of the code are changed.
Mock Objects replace the objects with which your method under test collaborates, thus offering a layer of isolation.
Mock objects are quite helpful to write a test wherein that part of the code integrates with expensive external resources.

20.1: Advanced Testing Concepts

Mock Objects in JUnit

- **Mock objects can either program these classes manually or use EasyMock to simulate these classes.**
  - EasyMock provides mock objects for interfaces in JUnit tests.
  - EasyMock is an open source software that is available under the terms of the Apache 2 license.

June 11, 2015 | Proprietary and Confidential | - 10 -

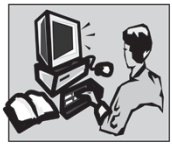
IGATE Speed.Agility.Imagination

Testing with JUnit – Isolated Testing:
Mock Objects in JUnit:
Other mock frameworks available are DynaMock and JMock.
In this course, we have a simple demo to understand representation of mock objects using EasyMock. Here we do not go into much details of EasyMock. However, you need easymock.jar in your classpath.

20.1: Advanced Testing Concepts

Demo

- **Demo on:**
  - Using Mock Object in JUnit
    - demo.mock.LoginTest.java



June 11, 2015 | Proprietary and Confidential | - 11 -

IGATE Speed.Agility.Imagination

Note:
Refer to demo.mock package for this demo
The UserDAO functionality has to be tested. No concrete implementation of this interface exist. We use a mock object to check the functionality of the method in UserDAO
The test depends on the provided methods.

The expect method tells EasyMock to expect certain method with some arguments and return method defines the return value of this method.
The replay method needs to be called to make mock objects available.
The verify method tells EasyMock to validate that all expected method calls were executed and in the correct order

Summary

- **In this lesson, you have learnt:**
  - Advanced Testing Concepts

Review Question

- **Question 1: While writing unit tests you should test ___.**
  - Option 1: Only public methods
  - Option 2: Only constructors
  - Option 3: Should test all methods
- **Question 2: Use constant expected values in assertions.**
  - True / False

Core Java 8 with JAXB and Development Tools

Logging with Log4J

IGATE
Speed.Agility.Imagination

Lesson Objectives

- **After completing this lesson, participants will be able to**
  - Understand the necessity of Logging
  - Work with log4j components
  - Configure java application for logging

Every Programmer is familiar with the process of inserting calls to System.out.println() into troublesome code to gain insight into program behavior. Of course, once you have figured out the cause of trouble, you remove the print statements, only to put them back in when the next problem surfaces. The logging API is designed to overcome the problem.
This lesson talks about Log4j, a package used to output log statements to a variety of output targets

Lesson outline:

21.1 Introduction
21.2 Log4J Concepts
21.3 Installation of Log4J
21.4 Configuring Log4J

21.1: Introduction

Overview of Logging

- Logging is writing the state of a program at various stages of its execution to some repository such as a log file.
- By logging, simple yet explanatory statements can be sent to text file, console, or any other repository.
- Using logging, a reliable monitoring and debugging solution can be achieved.

June 11, 2015 | Proprietary and Confidential | - 3 -

IGATE Speed.Agility.Imagination

What is Logging?
Logging, or writing the state of a program at various stages of its execution to some repository such as a log file, is an age-old method used for debugging and monitoring applications.

21.1: Introduction

Logging Requirements

- **Logging is used due to the following reasons:**
  - It can be used for debugging.
  - It is cost effective than including some debug flag.
  - There is no need to recompile the program to enable debugging.
  - It does not leave your code messy.
  - Priority levels can be set.
  - Log statements can be appended to various destinations such as file, console, socket, database, and so on.
  - Logs are needed to quickly target an issue that occurred in service.

June 11, 2015 | Proprietary and Confidential | - 4 -

IGATE Speed.Agility.Imagination

Logging Requirements:
One of the challenges in any programming environment is to be able to debug the code effectively. Suppose an application that is running in unattended mode on the server is not behaving properly. Everything seems to work fine in your development and test environment; yet in the production environment, something seems not to be "working" correctly. If you decide to print to standard output or to a log file, as an application developer, then you need to worry about commenting out the code in production to reduce the overhead associated with the calls.
Another approach is to define a Boolean variable, say debug, and if the value of the variable is true, then the application prints a whole set of debug messages. You compile by switching the flag one way or the other to get the necessary behavior. This is computationally expensive in addition to being cumbersome. With the logging API, however, you do not need to recompile your program every time you want to enable debugging. Furthermore you can set different levels for logging messages without incurring too much computational expense. You can even specify the kind of messages you want to log. Using a configuration file, you can change the runtime level of the logging information. This information can be written to a file, a screen console, a socket, a database, or any combination. It can be very detailed or very sparse, based on the level set at runtime, and may differ for various consumers of the information. For detailed analysis, examine the log file to discover when and where a problem occurs.

21.2: Log4J Concepts

Concept of Log4J

- **Log4j is an open source logging API for Java.**
  - It handles inserting log statements in application code, and manages them externally without touching application code, by using external configuration files.
  - It categorizes log statements according to user-specified criteria and assigns different priority levels to these log statements.
  - It lets users choose from several destinations for log statements, such as console, file, database, SMTP servers, GUI components.
  - It facilitates creation of customized formats for log output and provides default formats.

June 11, 2015 | Proprietary and Confidential | - 5 -

IGATE Speed.Agility.Imagination

What is Log4J?
Log4j is an opensource logging API for Java. This logging API became so popular that it has been ported to other languages such as C, C++, Python, and even C# to provide logging framework for these languages.
Log4j categorizes log statements according to user-specified criteria and assigns different priority levels to these log statements. These priority levels decide which log statements are important enough to be logged to the log repository.
Log4j lets users choose from several destinations for log statements, such as console, file, database, SMTP servers, GUI components; with option of assigning different destinations to different categories of log statements.
These log destinations can be changed anytime by simply changing log4j configuration files.
Advantages of Log4J are as follows:

It organizes the log output in separate categories, makes it easy to pinpoint the source of an error.
Log4j facilitates external configuration at runtime, makes the management and modification of log statements very simple.
Log4j assigns priority levels to loggers and log requests, helps weed out unnecessary log output, and allows only important log statements to be logged.

21.2: Log4J Concepts

Components

- **Log4j comprises of three main components:**
  - Logger
  - Appender
  - Layout
- **Users can extend these basic classes to create their own loggers, appenders, and layouts.**

June 11, 2015 | Proprietary and Confidential | - 6 -

IGATE Speed.Agility.Imagination

Log4J Concepts:
Logically, log4j can be viewed as being comprised of three main components: logger, appender, and layout.
The functionalities of each of these components are accessible through Java classes of the same name. Users can extend these basic classes to create their own loggers, appenders, and layouts.

21.2: Log4J Concepts

Logger

- **The component logger accepts log requests generated by log statements.**
- **Logger class provides a static method getLogger(name).**
  - This method:
    - Retrieves an existing logger object by the given name (or)
    - Creates a new logger of given name if none exists.
  - It then sends their output to appropriate destination called appenders.

IGATE Speed.Agility.Imagination

Logger:
The component logger accepts or enables log requests generated by log statements (or printing methods) during application execution. Subsequently, it sends their output to appropriate destination, that is appender(s), specified by user.
The logger component is accessible through the Logger class of the log4j API.

21.2: Log4J Concepts

Logger

- **The logger object is then used to … .**
  - set properties of logger component
  - invoke methods which generate log requests, namely:
    - debug(), info(), warn(), error(), fatal(), and log()
- **Each class in the Java application being logged can have an individual logger assigned to it or share a common logger with other classes.**
- **Any number of loggers can be created for the application to suit specific logging needs.**

June 11, 2015 | Proprietary and Confidential | - 8 -

IGATE Speed.Agility.Imagination

Logger:
One can create any number of loggers for the application to suit specific logging needs.
It is a common practice to create one logger for each class, with a name same as the fully-qualified class name. This practice helps to:

organize the log outputs in groups by the classes they originate from,
identify origin of log output, which is useful for debugging

21.2: Log4J Concepts

Logger

- **Log4j provides a default root logger that all user-defined loggers inherit from.**
  - Root logger is at the top of the logger hierarchy of all logger objects that are created.
  - If an application class does not have a logger assigned to it, it can still be logged using the root logger.
  - Root logger always exists and cannot be retrieved by name.

IGATE
Speed.Agility.Imagination

21.2: Log4J Concepts

Ways to create a Logger

- **Retrieve the root logger:**

```
Logger logger = Logger.getRootLogger();
```

- **Create a new logger:**

```
Logger logger = Logger.getLogger("MyLogger");
```

- **Instantiate a static logger globally, based on the name of the class:**

```
static Logger logger = Logger.getLogger(test.class);
```

- **Set the level with:**

```
logger.setLevel((Level)Level.WARN);
```


IGATE Speed.Agility.Imagination

21.2: Log4J Concepts

Logger Priority Levels

- **Loggers can be assigned different levels of priorities.**
- **Priority levels in ascending order of priority are as follows:**
  - DEBUG
  - INFO
  - WARN
  - ERROR
  - FATAL

June 11, 2015 | Proprietary and Confidential | - 11 -

IGATE Speed.Agility.Imagination

Logger Priority Levels:
Loggers can be assigned different levels of priorities.
These priority levels decide which log statement is going to be logged.
There are five different priority levels: DEBUG, INFO, WARN, ERROR, and FATAL; in ascending order of priority.

21.2: Log4J Concepts

Logger Priority Levels

- **There are printing methods for each of these priority levels.**
- **mylogger.info("logstatement1");**
  - generates log request of priority level INFO for logstatement1

June 11, 2015 | Proprietary and Confidential | - 12 -

IGATE Speed.Agility.Imagination

Logger Priority Levels (contd.):
As we can see, log4j has corresponding printing methods for each of these priority levels. These printing methods are used to generate log requests of corresponding priority level for log statements.
For example: mylogger.info("logstatement-1"); generates log request of priority level INFO for logstatement-1.

21.2: Log4J Concepts

Logger Priority Levels

- **In addition, there are two special levels of logging available:**
  - ALL: It has lowest possible rank. It is intended to turn on all logging.
  - OFF: It has highest possible rank. It is intended to turn off logging.
- **The behavior of loggers is hierarchical.**
- **The root logger is assigned the default priority level DEBUG.**

IGATE
Speed.Agility.Imagination

21.2: Log4J Concepts

Logger Priority Levels

- **The behavior of loggers is hierarchical. The following table illustrates this situation.**
  - Logger Output Hierarchy:

Will Output Messages Of Level

Logger Level	DEBUG	INFO	WARN	ERROR	FATAL
DEBUG					
INFO					
WARN					
ERROR					
FATAL					
ALL					
OFF					

June 11, 2015 | Proprietary and Confidential | - 14 -

IGATE Speed.Agility.Imagination

Logger Priority Levels (contd.):

In log4j, there are five normal levels of logger available:

DEBUG: It prints the debugging information and is helpful in the development stage.

INFO: It prints informational messages that highlight the progress of the application.

ERROR: It prints error messages that might still allow the application to continue running.

WARN: It prints information related to some faulty and unexpected behavior of the system, which needs attention in near future or else can cause malfunctioning of the application.

FATAL: It prints system critical information, which are causing the application to crash.

ALL: It has the lowest possible rank and is intended to turn on all logging.

OFF: It has the highest possible rank and is intended to turn off all the logging.

21.2: Log4J Concepts

Logger Priority Levels

- A logger will only output messages that are of a level greater than or equal to it.
- If the level of a logger is not set, it will inherit the level of the closest ancestor.
- If a logger is created in the package com.foo.bar and no level is set for it, then it will inherit the level of the logger created in com.foo.

IGATE
Speed.Agility.Imagination

21.2: Log4J Concepts

Logger Priority Levels

- If no logger was created in com.foo, then the logger created in com.foo.bar will inherit the level of the root logger.
- The root logger is always instantiated and available. The root logger is assigned the level DEBUG.

IGATE
Speed.Agility.Imagination

21.2: Log4J Concepts

Examples on Priority Levels

- /* Instantiate a logger named MyLogger */
  Logger mylogger = Logger.getLogger("MyLogger");
- /* Set logger priority I*/
  mylogger.setLevel(Level.INFO);
- /* statement logged, since INFO = INFO*/
  mylogger.info(" values ");
- /* statement not logged, since DEBUG < INFO*/ mylogger.debug("not logged");
- /* statement logged, since ERROR >INFO*/ mylogger.error("logged");

IGATE Speed.Agility.Imagination

21.2: Log4J Concepts

Appender

- **Appender component is interface to the destination of log statements, a repository where the log statements are written/recorded.**
- **A logger receives log request from log statements being executed, enables appropriate ones, and sends their output to the appender(s) assigned to it.**
- **The appender writes this output to repository associated with it.**
  - Examples: ConsoleAppender, FileAppender, WriterAppender, RollingFileAppender, DailyRollingFileAppender

June 11, 2015 | Proprietary and Confidential | - 18 -

IGATE Speed.Agility.Imagination

Appender:
There are various appenders available such as:

ConsoleAppender: It appends log events to System.out or System.err using a layout specified by the user
FileAppender: It appends log events to a file.
WriterAppender: It appends log events to a Writer or an OutputStream depending on the user's choice.
RollingFileAppender: It extends FileAppender to backup the log files when they reach a certain size.
DailyRollingFileAppender: It extends FileAppender so that the underlying file is rolled over at a user chosen frequency.
SMTPAppender: It sends an e-mail when a specific logging event occurs, typically on errors or fatal errors
SyslogAppender: It sends messages to a remote syslog daemon.
TelnetAppender: It specializes in writing to a read-only socket.
SocketAppender: It sends LoggingEvent objects to a remote a log server, usually a SocketNode.
SocketHubAppender: It sends LoggingEvent objects to a set of remote log servers, usually a SocketNodes.
An appender is assigned to a logger using the addAppender( ) method of the Logger class, or through external configuration files. A logger can be assigned one or more appenders that can be different from appenders of another logger. This is useful for sending log outputs of different priority levels to different destinations for better monitoring.

21.2: Log4J Concepts

Layout

- **The Layout component defines the format in which the log statements are written to the log destination by "appender".**
- **Layout is used to specify the style and content of the log output to be recorded.**
  - This is accomplished by assigning a layout to the appender concerned.
- **Layout is an abstract class in log4j API.**
  - It can be extended to create user-defined layouts.

IGATE Speed.Agility.Imagination

Layout:
The Layout component defines the format in which the log statements are written to the log destination by appender.
Layout is used to specify the style and content of the log output to be recorded, such as inclusion/exclusion of date and time of log output, priority level, info about the logger, line numbers of application code from where log output originated, and so forth. This is accomplished by assigning a layout to the appender concerned.

21.2: Log4J Concepts

Types of Layout

- **Let us discuss the different types of layouts:**
  - HTMLLayout: It formats the output as a HTML table.
  - PatternLayout: It formats the output based on a conversion pattern specified. If none is specified, then it uses the default conversion pattern.
  - SimpleLayout: It formats the output in a very simple manner, it prints the Level, then a dash "-", and then the log message.
  - XMLLayout: It formats the output as a XML.

June 11, 2015 | Proprietary and Confidential | - 20 -

IGATE Speed.Agility.Imagination

Note: The sample program's are given later (after installation of log4j).

21.3: Installation of Log4J

Steps

- **Let us see the steps for installation of Log4J:**
  - Download log4j-1.2.4.jar from http://logging.apache.org/log4j
  - Extract the log4j-1.2.4.jar at any desired location and include its absolute path in the application's CLASSPATH.
  - Now, log4j API is accessible to user's application classes and can be used for logging.

June 11, 2015 | Proprietary and Confidential | - 21 -

IGATE Speed.Agility.Imagination

Installation of Log4J:
Let us see the steps for installation of Log4J:
Download and unzip logging-log4j-1.2.9.zip.
Set the LOG4J_HOME system variable to the directory where you installed Log4j.
For example: LOG4J_HOME=C:\Tools\logging-log4j-1.2.9
Edit the CLASSPATH system variable:
For example: CLASSPATH=%CLASSPATH%;%LOG4J_HOME%\dist\lib\log4j-1.2.9.jar
Note: The jar file is located in Log4j_HOME\lib\log4j-1.2.9.jar
Create the following directories:
c:\demo
c:\demo\com
c:\demo\com\igate
Create the file c:\demo\com\igate\Log4jDemo.java.
import org.apache.log4j.Logger;

```
public class Log4jDemo {
    static Logger log = Logger.getLogger("Log4jDemo");
    public static void main(String args[]) {
        log.debug("This is my debug message.");
        log.info("This is my info message.");
        log.warn("This is my warn message.");
        log.error("This is my error message.");
        log.fatal("This is my fatal message.");
    }
```

Steps for Installation of Log4J

- **Notes are there.**

June 11, 2015 | Proprietary and Confidential | - 22 -

IGATE Speed.Agility.Imagination

Installation of Log4J (contd.):
Open a dos window and type:
cd c:\demo

Compile the java code, type:
c:\demo>javac Log4jDemo.java

Create file c:\demo\log4j.properties

Note: The log4j.properties file must be put in the directory where you issued the java command.

Run your application:
java Log4jDemo

The following log messages will be displayed:
[ERROR] 04:27 (Log4jDemo.java:main:12)
This is my error message.

[FATAL] 04:27 (Log4jDemo.java:main:13)
This is my fatal message.

Installation of Log4J (contd.):

There are additional parameters that you can add to the java command line:
[Note: This program can be referred after doing the configuration slides.]

To debug Log4j use parameter: -Dlog4j.debug.
For example:

```
log4j: Trying to find [log4j.xml] using context classloader
sun.misc.Launcher$AppClassLoader@11b86e7.
log4j: Trying to find [log4j.xml] using
sun.misc.Launcher$AppClassLoader@11b86e7class loader.
log4j: Trying to find [log4j.xml] using
ClassLoader.getSystemResource().
log4j: Trying to find [log4j.properties] using context classloader
sun.misc.Launcher$AppClassLoader@11b86e7.
log4j: Using URL [file:/C:/demo/log4j.properties] for automatic log4j
configuration.
log4j: Reading configuration from URL
file:/C:/demo/log4j.properties
log4j: Parsing for [root] with value=[ERROR, stdout].
log4j: Level token is [ERROR].
log4j: Category root set to ERROR
log4j: Parsing appender named "stdout".
log4j: Parsing layout options for "stdout".
log4j: Setting property [conversionPattern] to [[%5p] %d{mm:ss}
(%F:%M:%L)%n%m%n%n].
log4j: End of parsing for "stdout".
log4j: Parsed "stdout" options.
log4j: Finished configuring.
[ERROR] 06:29 (Log4jDemo.java:main:12)
This is my error message.
[FATAL] 06:29 (Log4jDemo.java:main:13)
This is my fatal message.
```

```
java -Dlog4j.configuration=test.properties Log4jDemo
```

If you want to use another filename instead of log4j.properties:
-Dlog4j.configuration.
For example:
Rename log4j.properties to test.properties

21.4: Configuring Log4J

Process

- **The log4j can be configured both programmatically and externally using special configuration files.**
- **External configuration is most preferred.**
  - This is because to take effect it does not require change in application code, recompilation, or redeployment.
- **Configuration files can be XML files or Java property files that can be created and edited using any text editor or xml editor, respectively.**

June 11, 2015 | Proprietary and Confidential | - 24 -

IGATE Speed.Agility.Imagination

Configuring Log4J:
Inserting log requests into the application code requires a fair amount of planning and effort. Observation shows that approximately 4 percent of code is dedicated to logging. Consequently, even moderately sized applications will have thousands of logging statements embedded within their code. Given their number, it becomes imperative to manage these log statements without the need to modify them manually.
The log4j environment is fully configurable programmatically.
The simplest configuration file will contain following specifications that can be modified, both programmatically and externally, to suit specific logging requirements:

priority level and name of appender assigned to root logger
Appender's type
layout assigned to the appender

21.4: Configuring Log4J

Using Property File

- **The root logger is assigned priority level DEBUG and an appender named myAppender:**
  - log4j.rootLogger=debug, myAppender
- **The appender's type specified as ConsoleAppender, that is logs output to console:**
  - log4j.appender.myAppender=org.apache.log4j.ConsoleAppender
- **# The appender is assigned a layout SimpleLayout:**
  - log4j.appender.myAppender.layout=org.apache.log4j.SimpleLayout

IGATE
Speed.Agility.Imagination

21.4: Configuring Log4J

Using Configuration Property Files

- **Example 1:**

```
package com.igate.sampleapp;
import org.apache.log4j.*;
public class MyClass
{     static Logger myLogger =
Logger.getLogger(MyClass.class.getName( ));
      //no constructor required
      public void do_something( int a, float b)
      {       myLogger.info("Logged since INFO=INFO"); ...
              myLogger.debug("not enabled, since DEBUG < INFO");
      ...                                          contd.
```


IGATE Speed.Agility.Imagination

21.4: Configuring Log4J

Using Configuration Property Files

- **Example 1 (contd.):**

```
                    if (x == null)
                    {myLogger.error("enabled & logged ,since ERROR
> INFO");...}
        }
    }
```

  - //Execute the file as
    // java -Dlog4j.configuration=config-sample.properties
    com.igate.sampleapp.MyClass

IGATE Speed.Agility.Imagination

21.4: Configuring Log4J

Using Configuration Property Files

- **Example 2:**

```
package com.igate.sampleapp;
import org.apache.log4j.*;
import org.apache.log4j.PropertyConfigurator;
import java.net.*;
public class MyClass
{      static Logger myLogger =
Logger.getLogger(MyClass.class.getName( ));
       public MyClass()
     {
                    PropertyConfigurator.configure("config-
sample.properties");
     }
                    contd.
```

June 11, 2015 | Proprietary and Confidential | - 28 -

IGATE Speed.Agility.Imagination

Note: This version of MyClass instructs PropertyConfigurator to parse a configuration file config-sample.properties and set up logging accordingly.

21.4: Configuring Log4J

Using Configuration Property Files

- **Example 2 (contd.):**

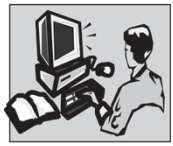
```
public void do_something( int a, float b)
{           myLogger.info("Logged since INFO=INFO"); ...
            myLogger.debug("not enabled, since DEBUG < INFO");
...
    if (x == null) {
      myLogger.error("enabled & logged ,since ERROR > INFO");...
    }
}
```


IGATE Speed.Agility.Imagination

21.4: Configuring Log4J

Demo

- **MyClass_Property.java**



IGATE
Speed.Agility.Imagination

21.5: Pros and Cons

Log4j

- **Advantages:**
  - Log4j print methods do not incur heavy process overhead.
  - External configuration makes management and modification of log statements simple and convenient.
  - Priority level of log statements helps to weed out unwanted logging.
- **ShortComings:**
  - Appender additivity may result in log requests being sent to unwanted appenders.

June 11, 2015 | Proprietary and Confidential | - 31 -

IGATE Speed.Agility.Imagination

Pros and Cons of Log4J:
The advantages of using log4j are listed below:
The log4j printing methods used for logging can always remain in application code because they do not incur heavy process overhead for the application and assist in ongoing debugging and monitoring of application code, thus proving useful in the long term.
Log4j organizes the log output in separate categories by the name of generating loggers that in turn are same as the names of the classes they log. This approach makes pinpointing the source of an error easy.
Log4j facilitates external configuration at runtime. This makes the management and modification of log statements very simple and convenient as compared to performing the same tasks manually.
Log4j assigns priority levels to loggers and log requests. This approach helps weed out unnecessary log output and allows only important log statements to be logged.
The shortcomings of log4j are listed below:
Appender additivity may result in the log requests being unnecessarily sent to many appenders and useless repetition of log output at an appeneder. Appender additivity is countered by preventing a logger from inheriting appenders from its ancestors by setting the additivity flag to false.
When configuration files are being reloaded after configuration at runtime, a small number of log outputs may be lost in the short time between the closing and reopening of appenders. In this case, Log4j will report an error to the stderr output stream, informing that it was unable send the log outputs to the appender(s) concerned. But the possibility of such a situation is minute. Also, this can be easily patched up by setting a higher priority level for loggers.

Lab : Log4J

- **Lab 13: Log4J**

Summary

- **In this lesson, you have learnt**
  - Log 4j and its components
  - The method to log messages using Log4J API
  - Configuring Log4j applications

Review Question

- **Question 1: Log4j is a product of ___.**
  - Option1: RedHat
  - Option2: Sun
  - Option3: Apache
  - Option4: None of the above

Review Question

- **Question 2: Configuration of Log4j applications can be done through:**
  - Option1: Property files
  - Option2: XML files
  - Option 3: Both Option1 and Option2 are right
  - Option4: None of the above is true

Review Question

- **Question 3: If the loglevel is set to INFO, will the following message be logged?**
  Message: logger.warn(" From warning");
  - Yes / No

Core Java 8 with JAXB and Development Tools

XML Processing using JAXB

IGATE
Speed.Agility.Imagination

Lesson Objectives

- **After completing this lesson, participants will be able to**
  - Understand the concept of JAXB architecture
  - Work with different JAXB bindings
  - Use JAXB to process XML files

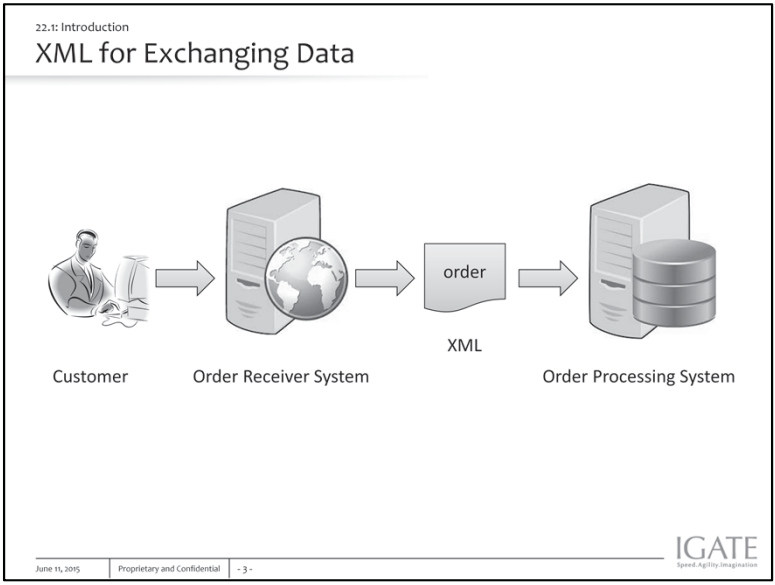
June 11, 2015 | Proprietary and Confidential | - 2 -

IGATE
Speed.Agility.Imagination

Java API for XML Binding (JAXB) is the much anticipated Java API designed to provide a programmatic association between XML and Java. This chapter focus on how to use JAXB for XML processing.

Lesson outline:

22.1: Introduction
22.2: JAXB Architecture
22.2: JAXB Bindings
22.3: Java to Schema binding



One of the most time-consuming challenges for developers is to exchange data between incompatible systems over the Internet.

Exchanging data as XML greatly reduces this complexity, since the data can be read by different incompatible applications.

Consider an example shown in slide, where there are two different applications (might be written in different languages) integrated to process orders from customer. While exchanging order, an order receiver system needs to convert object information into XML, transfer XML document to order processing system and finally the order processing system converts XML into object so as to persist the information in database.

To serialize objects into XML documents and de-serialize objects from XML documents, Java provides an standard API called JAXB. This lesson covers JAXB API support for bindings between XML documents and Java classes.

22.1: Introduction

Why JAXB?

- **Traditional API (DOM, SAX or JAXP) for XML processing is much complex and centered around working with different parsers**
- **DOM XML processing uses tree-based approach which is also inadequate**
- **Developers used to write custom beans to interpret XML schema which requires more efforts and extra coding**

June 11, 2015 | Proprietary and Confidential | - 4 -

IGATE Speed.Agility.Imagination

Traditional XML processing API like DOM, SAX or JAXP requires more efforts and extra coding. In those approaches XML schema was less significant and more work was centered around working with parsers.

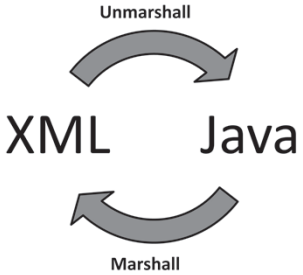
The transition from XML to Java code was inadequate as developers need to create POJO bases classes to interpret XML files into Java. Changes in XML schema requires Java classes to be updated.

To address all such issues and to save time while working with XML, there was a need for enhanced API.

22.1: Introduction

JAXB – Java API for XML Binding

- **JAXB 2.0 developed under JSR 222 and is included in Java EE 5 and Java SE 6**
- **JAXB defines a standard to bind Java representations to XML and vice versa.**
- **Provides API for reading and writing Java objects to and from XML documents**
- **JAXB abstracts developer from working with XML and XSD documents**



June 11, 2015 | Proprietary and Confidential | - 5 -

IGATE Speed.Agility.Imagination

JAXB stands for Java Architecture for XML Binding. The purpose of the API and JAXB specification is to facilitate the handling of an XML document by generating a set of classes.

These classes provides a higher level of abstraction than using JAXP (SAX or DOM). JAXB provides a tool that parses an XML schema, and generates set of classes.

JAXB enables developers to handle an XML documents without knowing XML processing technologies. All operations are done through java objects.

Marshalling is the action of transforming an object into XML. Unmarshalling would take this XML document as an input and instantiate java object.

Note: JAXB is ideally used when there is need to handle XML files directly into code. For example, working with web services.

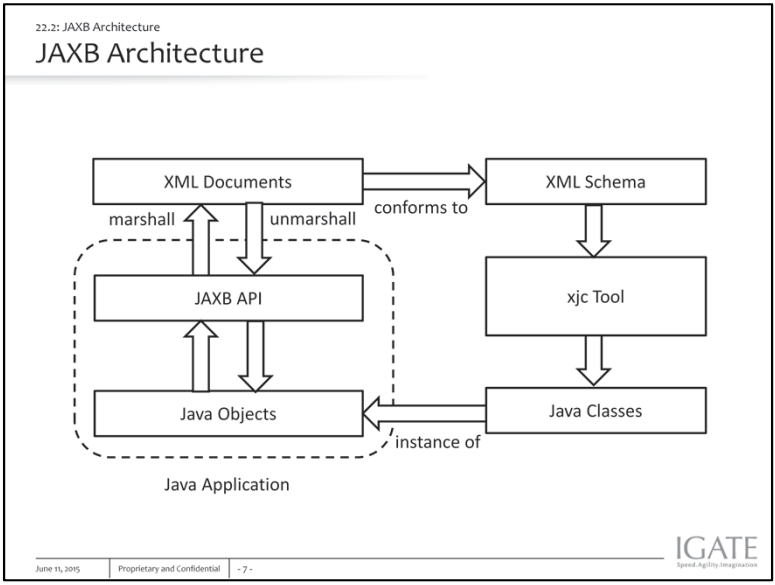
22.2: JAXB Architecture

JAXB 2.0 Features

- **JAXB 2.0 has made lots of improvements over the previous version**
  - Only XML schemas are supported (No DTD)
  - Implementation via annotations
  - Bidirectional mapping between XML schema and corresponding java classes
  - Support Generics and enumerations
- **No more XML processing knowledge required**

June 11, 2015 | Proprietary and Confidential | - 6 -

IGATE Speed.Agility.Imagination



JAXB architecture uses a design similar to the other JAX APIs . Developers use standard interfaces and classes provided by the specifications, and vendors provide the implementation which is .

As shown in the slide, developers write application code and business logic that uses Java classes and interfaces representing XML constructs. These classes are generated from XML schemas using JAXB provided tool called xjc.

The JAXB API is invoked to unmarshall the XML, and the data represented therein is populated in its corresponding Java classes.

To change the Java classes to the corresponding XML representation, developers invoke the JAXB API, and the Java classes are marshalled into the schema compliant XML format.

22.3: JAXB Bindings

JAXB Bindings

- **JAXB provides two ways of bindings between XML and Java**
  - Marshalling
  - Unmarshalling
- **JAXB provides an abstract class JAXBContext which manages the bindings**
- **Unmarshaller class is used to transform XML in Object graph**
- **Marshaller class is used to transform Object graph into XML document**
- **JAXB 2.0 provides many annotations which can be used during the marshalling or unmarshalling operations**

IGATE
Speed.Agility.Imagination

22.3: JAXB Bindings

JAXB Annotations

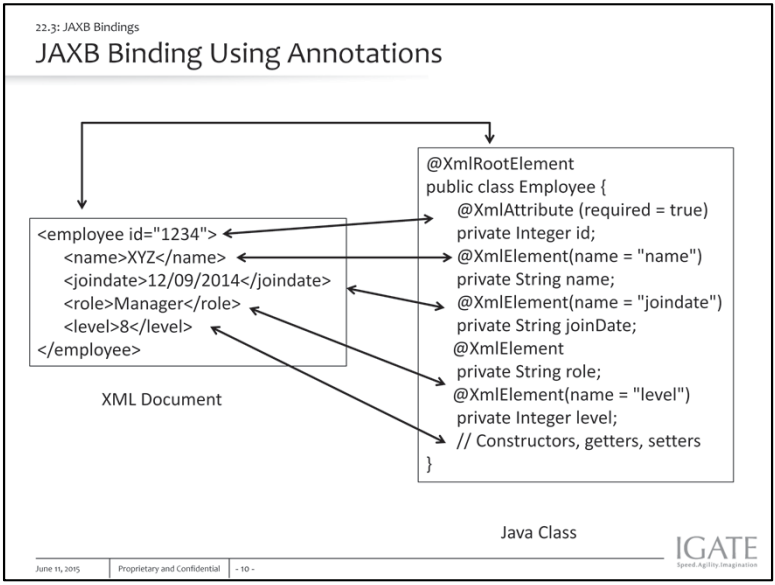
- **JAXB 2.0 provides many annotations under the javax.xml.bin.annotation**
- **Following table lists few important:**

Annotation	Meaning
@XmlAccessorType	Controls whether fields or Javabean properties are serialized by default (FIELD, NONE, PROPERTY, PUBLIC_MEMBER)
@XmlRootElement	Class level annotation to represent root element of XML file
@XmlAttribute	Maps attribute or getter to an XML attribute
@XmlElement	Maps nonstatic, nontransient attribute to XML attribute
@XmlTransient	Attribute not considered for XML binding
@XmlList	Maps a property to list
@XmlType	Annotates a class as being a complex type in XML Schema
@XmlValue	Allows the mapping of a class to a simple schema type

IGATE
Speed.Agility.Imagination

JAXB provides xjc (XML to Java compiler) tool for generating java classes from XML schema and vice versa. However, starting from JAXB 2.0, developers use annotations to create and control all aspects of Java binding.

The slides shows few important annotations provided in JAXB which are used to bind java classes with XML documents.



The @XmlRootElement annotation notifies JAXB that the Employee class is the root element of the XML document. If this annotation is missing, JAXB will throw an exception when trying to convert XML to Java or vice versa.

As an XML document is made of elements and attributes, JAXB uses two annotations to differentiate them: @XmlAttribute and @XmlElement. Each annotation has a set of parameters that allows you to rename an attribute, allow null values or not, give default values, and so forth.

Once binding is ready, the next step is to marshall or unmarshall XML documents.

22.3: JAXB Bindings

Unmarshalling XML to Java Object

- **JAXB provides an abstract class JAXBContext, which is used to get instance of Unmarshaller interface**
- **JAXBContext requires reference of binding configured annotated class**
- **Use unmarshal() method of Unmarshaller to read the XML document into an Java object**

```
// create an instance of JAXBContext
JAXBContext context = JAXBContext.newInstance(Employee.class);

// create an instance of unmarshaller from JAXBContext
Unmarshaller unmarshaller = context.createUnmarshaller();

// create source stream from where unmarshaller will read xml contents
File file = new File("resources/rajat.xml");

// unmarshall the XML file into and object
employee = (Employee) unmarshaller.unmarshal(file);
```


IGATE Speed.Agility.Imagination

22.3: JAXB Bindings

Marshalling Java Object to XML

- **Use marshal() method of Marshaller interface to read the XML document into an Java object**
- **This method needs an handler argument which defines the destination of XML document**

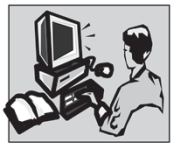
```
// create an instance of JAXBContext
JAXBContext context = JAXBContext.newInstance(Employee.class);

//create an instance of marshaller from JAXBContext
Marshaller marshaller = context.createMarshaller();

//create destination stream where marshaller will write xml contents
FileWriter writer = new FileWriter(new File("resources/rajat.xml"));

//marshall the object into an XML file
marshaller.marshal(employee, writer);
```


IGATE
Speed.Agility.Imagination

22.3: JAXB Bindings

Demo

- **Execute the :**
  - MarshallerDemo
  - UnMarshallerDemo



IGATE
Speed.Agility.Imagination

22.4: JAXB Bindings

Lab : JAXB

- **Lab 14: JAXB**

IGATE
Speed.Agility.Imagination

Summary

- **In this lesson, you have learnt:**
  - Architecture of JAXB
  - How to use JAXB for XML processing

Review Question

- **Question 1 : ____ annotation maps an attribute or a getter to an XML attribute of simple type.**
- **Question 2 : JAXB defines a standard to bind Java representations to XML and vice versa**
  - True/False

Core Java 8 with JAXB and Development Tools

Java Security Concepts

IGATE
Speed.Agility.Imagination

Lesson Objectives

- **After completing this lesson, participants will be able to**
  - Understand Java Security Architecture

This lesson introduces the JVM model of Security and memory management.

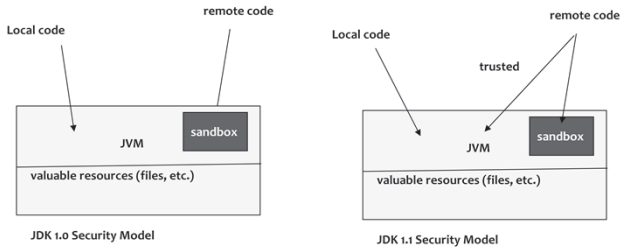
Lesson outline:

23.1: Java Security Architecture
23.2: Certificate
23.3: Security related tools

23.1: Java Security Architecture

Introduction-Original Security Model

- **Java security provides a secure ready-built platform on which Java-enabled applications can run in a secured manner.**



June 11, 2015 | Proprietary and Confidential | - 3 -

IGATE Speed.Agility.Imagination

Original Security Model:

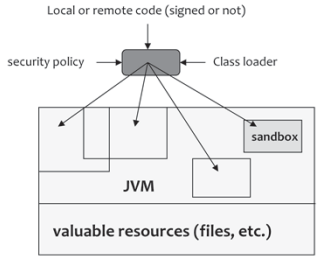
In JDK 1.0 and JDK 1.1, Java platform provided a sandbox model as the security model.

The concept of this security model was that only local code was fully trusted and hence was allowed to access the vital resources (such as the file system) . Whereas the remote code (for example, applet) was considered as untrusted, hence not allowed to access the vital resources and runs in a sandbox which provides limited access to resources.

In JDK 1.1, if the remote code is signed (signed applet) then it is allowed to run out of the sand box and thus, can access the system resources.

23.1: Java Security Architecture

Java 2 Security Model



JDK 1.5 Security Model

No code is trusted in this security model. Every code whether local or remote runs as per the set permissions.

Java 2 Security Model:

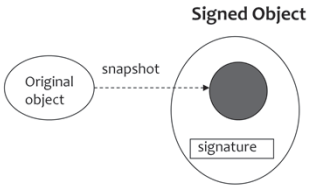
The new Java 2 Platform Security Architecture, illustrated in the slide, is introduced primarily for the following purposes:
Fine-grained access control: This was possible in the earlier versions of JDK by subclassing the SecurityManager and ClassLoader classes which results in substantial amount of programming. The new security architecture has made things easier and safer for the developer.
Easily configurable security policy: The security policy can be configured, thus making the life of the programmers easy.
Easily extensible access control structure: Earlier, the SecurityManager class needed a change (addition of a check method) to create a new access permission. However, in the new model, this is avoided with the introduction of typed permissions (each representing an access to a system resource) and automatic handling of permissions.

The most important aspect of this new model is no code is trusted. Any Java code (local or remote, signed or unsigned) is subjected to the same security constraints. However, it is possible to declare the local code as trusted through configuration.

23.1: Java Security Architecture

Signed Object

- **SignedObject contains another serializable object and its signature.**



SignedObject can be used as an authorization token which can be passed around without getting modified. The SignedObject class is immutable.

June 11, 2015 | Proprietary and Confidential | - 5 -

IGATE Speed.Agility.Imagination

Signed Object:

The java.security.SignedObject class is an essential building block for other security primitives. The SignedObject contains another serializable object and its signature as shown in the figure on the slide.

The signed object is a 'deep copy' (in serialized form) of the original object. The manipulation of the original object does not affect the copy created because the SignedObject is immutable.

SignedObject can be used internally as an authorization token which can be passed around without getting maliciously modified.

23.2: Certificates

X.509 Certificate

- **A public-key certificate is a digitally signed statement from one entity, saying that the public key (and some other information) of another entity has some specific value.**
- **JDK contains certificate interfaces and classes for parsing and managing certificates**
- **"Keytool" java tool can be used to generate, display, import and export X.509 certificates**

June 11, 2015 | Proprietary and Confidential | - 6 -

IGATE Speed.Agility.Imagination

X.509 Certificate:
The terms related to a certificate are explained below:
Public Key : This is a number/finger print associated with a particular entity and it is the one that is shared with everyone who intends to use it for verifying the signatures on the certificate.

Digitally Signature : A certificate normally contains a digital signature. The signature contains the "identity" of an entity and proves that the entity is trusted. The data (code or document) to be trusted is signed with the entity's private key. The digital signature is also computed using the entity's private key.

Private Key: This is also a number/finger print anly known to the entity who is the owner of the data (code or document). It is not shared with anyone. It is used to sign the data that needs to be secured. Private and public keys exist in pairs in all public key cryptography systems (also referred to as "public key crypto systems").

Entity: An entity can be a person, organization, program, computer, business, bank, or something else you are trusting to some degree.

Basically, public key cryptography requires access to users' public keys. Certificates were invented to establish a secure relationship between entities who need to exchange data in a secure fashion.
The java tool used to generate, display, import and export X.509 certificates is "keytool". The X.509 is a standard that defines what information can go into a certificate, and describes how to write it down (the data format) – for details refer to Java docs.

23.3: Certificates

KeyStore

- KeyStore **is a protected database/repository of keys (public/private) and certificates.**
- Keytool **allows creation of the keys and certificates and stores it in the** KeyStore**.**
- **The keys stored in the** keySstore **can also be password-protected.**
- **The keys and certificates stored in the keystore can be used to digitally sign the code using the** jarsigner **tool.**

June 11, 2015 | Proprietary and Confidential | - 7 -

IGATE Speed.Agility.Imagination

KeyStore:

A keystore is a protected database that holds keys and certificates for an entity. The password for the keystore can be defined during its creation and it can be changed at later stage. The private key stored in the key store can also be password protected.

There is a tool named keytool which can be used to create public/private key pairs and self-signed X.509 v1 certificates, and to manage keystores.
The private Keys and certificates are used to digitally sign your Java applications and applets using the jarsigner tool.

As mentioned earlier, keytool can be used to generate, display, import, and export X.509 certificates which are stored as files.

23.3: Tools

Security Related Tools

- **Keytool**
- **jarsigner tool**
- **Policy Tool**

June 11, 2015 | Proprietary and Confidential | - 8 -

IGATE Speed.Agility.Imagination

Security Related Tools:

Keytool

The keytool is used to create pairs of public and private keys, to import and display certificate chains, to export certificates, and to generate X.509 v1 self-signed certificates and certificate requests that can be sent to a certification authority.

jarsigner tool

The jarsigner signs JAR (Java ARchive format) files and verifies the authenticity of the signature(s) of signed JAR files.

Policy Tool

The policy tool creates and modifies the policy configuration files that define your installation's security policy.

Summary

- **In this lesson, you have learnt:**
  - JVM Security Architecture
  - Certificate
  - Security related tools

Review Question

- **Question 1 : JDK 1.0 & JDK 1.1 security model is known as _____ model**
- **Question 2 : No code is trusted in Java 2 security model**
  - True/False

IGATE
Speed.Agility.Imagination

Core Java 8 with JAXB and Development Tools

Lab Book

Copyright © 2015IGATE Corporation. All rights reserved. No part of this publication shall be reproduced in any way, including but not limited to photocopy, photographic, magnetic, or other record, without the prior written permission of IGATE Corporation.

IGATE Corporation considers information included in this document to be Confidential and Proprietary.

Document Revision History

Date	Revision No.	Author	Summary of Changes
17-11-2013	1.0	Rathnajothi P	As of updated module content, designed lab book
28-05-2015	2.0	Vinod Satpute	Updated to include new features of Java SE 8, Junit 4 and JAXB 2.0

IGATE
Speed.Agility.Imagination

Table of Contents

Document Revision History 2
Table of Contents 3
Getting Started 4
Overview 4
Setup Checklist for Core Java 4
Instructions 4
Learning More (Bibliography if applicable) 4
Problem Statement/ Case Study (If applicable) 5
Lab 1. *Working with Java and Eclipse IDE* 6
Lab 2. *Language Fundamentals, Classes and Objects* 14
Lab 3. *Exploring Basic Java Class Libraries* 16
Lab 4. *Inheritance and Polymorphism* 19
Lab 5. *Abstract classes and Interfaces* 19
Lab 6. *Exception Handling* 21
Lab 7. *Arrays and Collections* 21
Lab 8. *File IO* 22
Lab 9. *Introduction to Junit* 33
Lab 10. *Lambda Expressions and Stream API* 30
Lab 11. *Property Files and JDBC 4.0* 32
Lab 12. *Introduction to Layered Architecture* 33
Lab 13. *Log4J* 35
Lab 14. *JAXB* 44
Appendices 45
Appendix A: Naming Conventions 45
Appendix B: Table of Figures 46

IGATE
Speed.Agility.Imagination

Getting Started

Overview

This lab book is a guided tour for learning Core Java version 8 with JAXB and development tools. It comprises of assignments to be done. Refer the demos and work out the assignments given by referring the case studies which will expose you to work with Java applications.

Setup Checklist for Core Java

Here is what is expected on your machine in order to work with lab assignment.

Minimum System Requirements

- Intel Pentium 90 or higher (P166 recommended)
- Microsoft Windows 7 or higher.
- Memory: (1GB or more recommended)
- Internet Explorer 9.0 or higher or Google Chrome 43 or higher
- Connectivity to Oracle database

Please ensure that the following is done:

- A text editor like Notepad or Eclipse is installed.
- JDK 1.8 or above is installed. (This path is henceforth referred as <java_home>)

Instructions

- For all Naming conventions, refer Appendix A. All lab assignments should adhere to naming conventions.
- Create a directory by your name in drive <drive>. In this directory, create a subdirectory java_assignments. For each lab exercise create a directory as lab <lab number>.

Learning More (Bibliography if applicable)

- https://docs.oracle.com/javase/8/docs/
- Java, The Complete Reference; by Herbert Schildt
- Thinking in Java; by Bruce Eckel
- Beginning Java 8 Fundamentals by KishoriSharan

Problem Statement/ Case Study (If applicable)

1. **Bank Account Management System:**

➢ Funds Bank needs an application to feed new Account Holder information. AccountHolder will be a person. There are two types of accounts such as SavingsAccount, CurrentAccount.

2. **Employee Medical Insurance Scheme:**

➢ By default, all employees in an organization will be assigned with a medical insurance scheme based on the salary range and designation of the employee. Refer the below given table to find the eligible insurance scheme specific to an employee.

Salary	Designation	Insurance scheme
>5000 and < 20000	System Associate	Scheme C
>=20000 and <40000	Programmer	Scheme B
>=40000	Manager	Scheme A
<5000	Clerk	No Scheme

Lab 1: Working with Java and Eclipse IDE

Goals	Learn and understand the process of: ➢ Setting environment variables ➢ Creating a simple Java Project using Eclipse 3.0 or above
Time	45 minutes

1.1: Setting environment variables from CommandLineSolution:

Step 1: Set **JAVA_HOME** to Jdk1.8 using the following command:

- **Set JAVA_HOME=**C:\Program Files\Java\jdk1.8.0_25

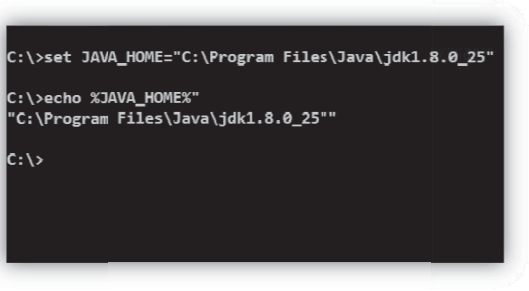


Figure 1: Java program

Step 2: Set PATH environment variable:

- **Set PATH=%PATH%;%JAVA_HOME%\bin;**

Step 3: Set your current working directory and set classpath.

- Set CLASSPATH=.

Note: Classpath searches for the classes required to execute the command. Hence it must be set to the directory containing the class files or the names of the jars delimited by ;

For example: C:\Test\myproject\Class;ant.jar

Alternatively follow the following steps for setting the environment variables

Alternate approach:

IGATE
Speed.Agility.Imagination

Step 1: Right click **My Computers**, and select **Properties→Environment Variables**.

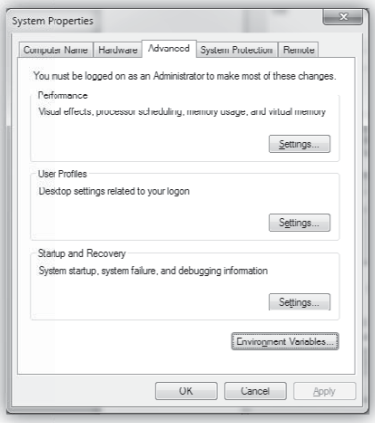


Figure 2: System Properties

Step 2: Click **Environment Variables**. The Environment Variables window will bedisplayed.

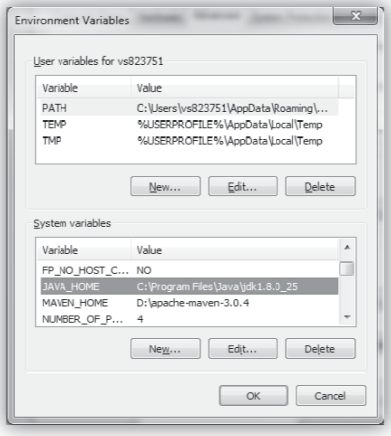


Figure 3: Environment Variables

Step 3: Click **JAVA_HOME** System Variable if it already exists, or create a new one and set the path of JDK1.8 as shown in the figure.

IGATE
Speed.Agility.Imagination

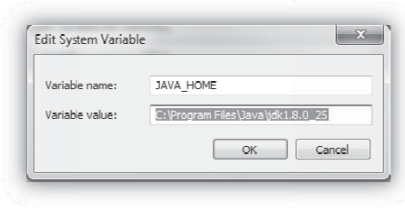


Figure 4: Edit System Variable

Step 4: Click **PATH** System Variable and set it as ***%PATH%;%JAVA_HOME%\bin***.

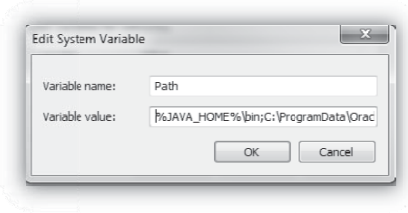


Figure 5: Edit System Variable

Step 5: Set **CLASSPATH** to your working directory in the **User Variables** tab.

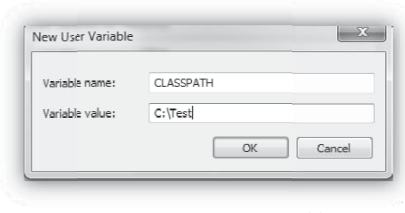


Figure 6: Edit User Variable

1.2: Create Java Project

Create a simple java project named 'MyProject'.

Solution:

Step 1: Open **eclipse 4.4**(or above)

Step 2: Select **File→New→Project →Java project**.

IGATE
Speed.Agility.Imagination

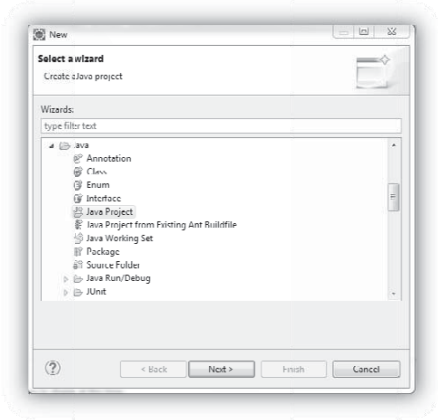


Figure 7: Select Wizard

Step 3: Click **Next** and provide name for the project.

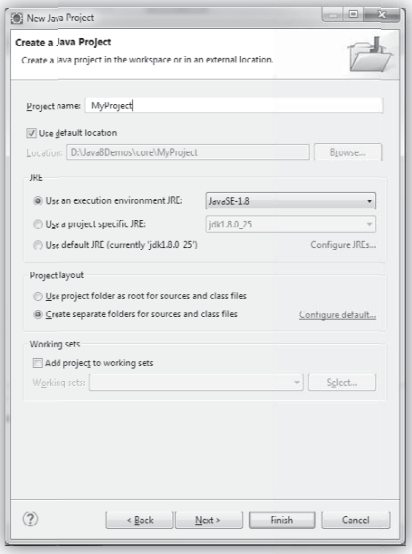


Figure 8: New Java Project

IGATE
Speed.Agility.Imagination

Step 4: Click **Next** and select build options for the project.

Figure 9: Java Settings

Step 5: Click **Finish** to complete the project creation.

Step 6: Right-click **myproject**, and select resource type that has to be created.

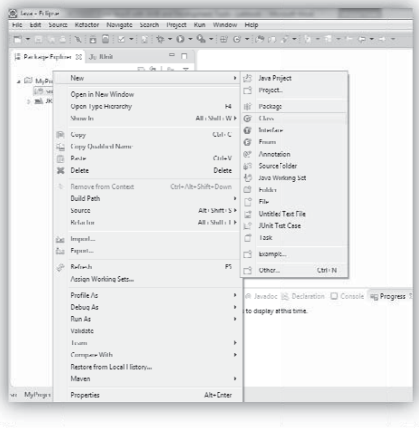


Figure 10: Select Resource

IGATE
Speed.Agility.Imagination

Step 7: Providename and other details for the class, and click **Finish**.

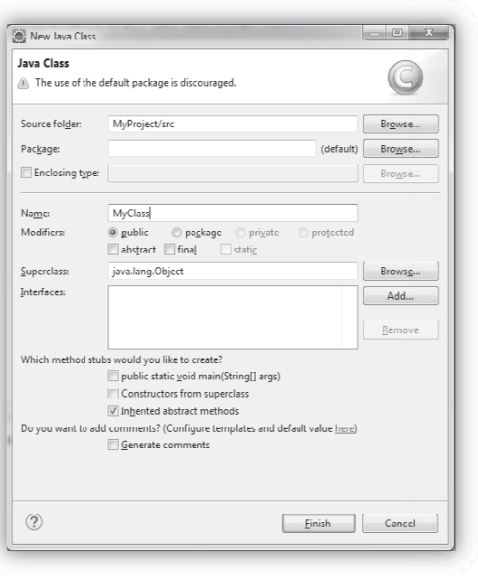


Figure 11: Java Class

This will open **MyClass.java** in the editor, with ready skeleton for the class, default constructor, **main()** method, and necessary **javadoc** comments.

To run this class, select **Run** from toolbar, or select **Run As →Java application**. Alternatively, you can select **Run..**and you will be guided through a wizard, for the selection of class containing **main()** method.

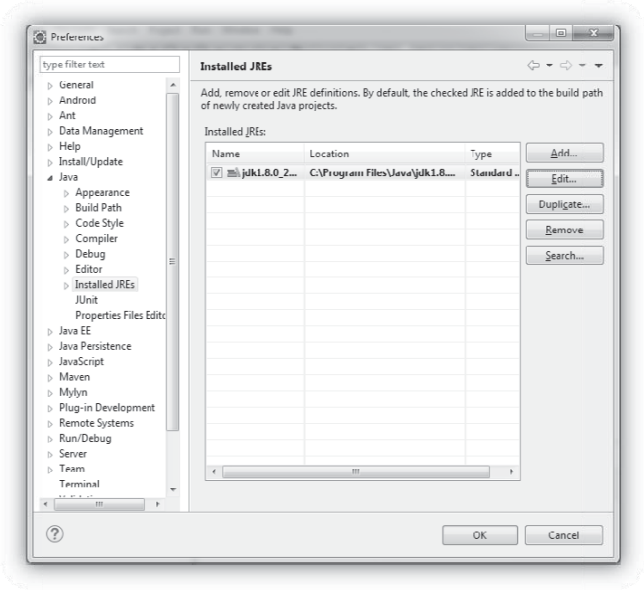
Console window will show the output.

1.3: Using offline Javadoc API in Eclipse

Step 1: Open **eclipse 4.4**(or above)

Step2: From eclipse Window → Preferences →Java → "Installed JREs" select available JRE (jdk1.8.0_25 for instance) and click Edit.

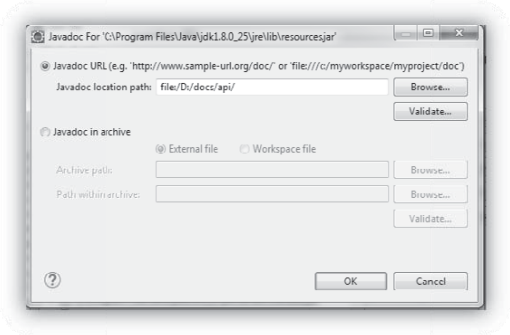
IGATE
Speed.Agility.Imagination



Step3: Select all the "JRE System libraries" using Control+A.

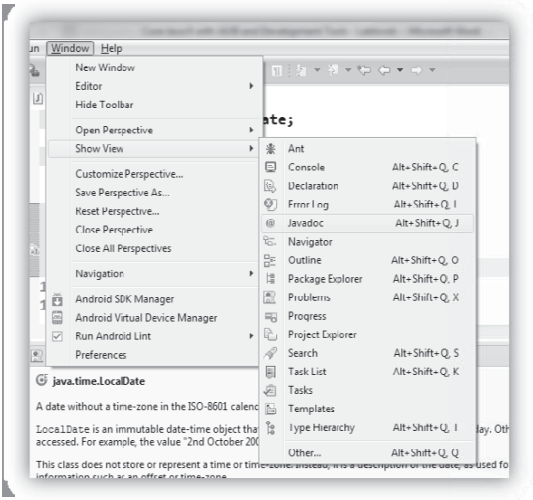
Step 4: Click "Javadoc Location"

Step 5: Change "Javadoc location path:" from http://download.oracle.com/javase/8/docs/api/ to "file:/E:/Java/docs/api/".

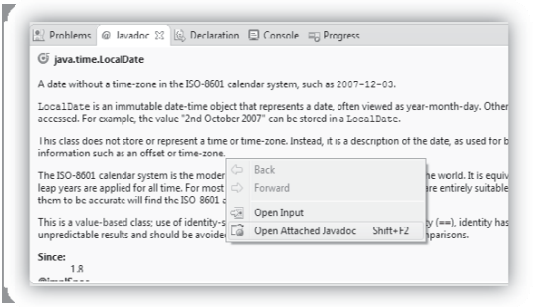


Step 6: Close all windows by either clicking on ok/apply.

Step 7: Open the Javadoc view from Window → Show View → Javadoc.



Note: Henceforth whenever you select any class or method in Editor Window, it Javadoc view will display the reference documentation.



If you want to open the Java documentation for specified resource as html page, right click in the Javadoc view → Open Attached Javadoc.

IGATE
Speed.Agility.Imagination

Lab 2: Language Fundamentals, Classes and Objects

Goals	At the end of this lab session, you will be able to: ➢ Write a Java program that displays person details ➢ Working with Conditional Statements ➢ Create Classes and Objects
Time	120 minutes

2.1 Write a java program to print person details in the format as shown below:

```
Person Details:
_____

First Name: Divya
Last Name: Bharathi
Gender: F
Age: 20
Weight: 85.55
```

Figure 12: Sample output of Person details

2.2: Write a program to accept a number from user as a command line argument and check whether the given number is positive or negative number.

2.3: Refer the class diagram given below and createa personclass.

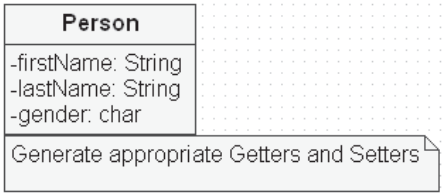


Figure 13: Class Diagram of Person

Create default and parameterized constructor for Person class.

IGATE
Speed.Agility.Imagination

Also Create "PersonMain.java" program and write code for following operations:

a) Create an object of Person class and specify person details through constructor.

b) Display the details in the format given in Lab assignment 2.1

2.4: Modify Lab assignment 2.3 to accept phone number of a person. Create a newmethod to implement the same and also define method for displaying persondetails.

2.5: Modify the above program, to accept only 'M' or 'F' as gender field values. Use Enumeration for implementing the same.

Lab 3: Exploring Basic Java Class Libraries

Goals	At the end of this lab session, you will be able to: ➢ Working with Basic Java Class Libraries ➢ Working with Strings and Date and Time API
Time	100 minutes

3.1: Create a method which can perform a particular String operation based on the user's choice. The method should accept the String object and the user's choice and return the output of the operation. Options are

- Add the String to itself
- Replace odd positions with #
- Remove duplicate characters in the String
- Change odd characters to upper case

3.2: Create a method that accepts a String and checks if it is a positive string. A string is considered a positive string, if on moving from left to right each character in the String comes after the previous characters in the Alphabetical order.For Example: ANT is a positive String (Since T comes after N and N comes after A). The method should return true if the entered string is positive.

3.3: Create a method to accept date and print the duration in days, months and years with regards to current system date.

3.4: Revise exercise 3.3 to accept two LocalDates and print the duration between dates in days, months and years.

3.5: Create a method to accept product purchase date and warrantee period (in terms of months and years). Print the date on which warrantee of product expires.

3.6: Create a method which accept zone id and print the current date and time with respect to given zone. (Hint: Few zones to test your code. America/New_York, Europe/London, Asia/Tokyo, US/Pacific, Africa/Cairo, Australia/Sydney etc.)

3.7: Modify Lab assignment 2.3 to perform following functionalities:

a) Add a method called calculateAge which should accept person's date of birth and calculate age of a person.

b) Add a method called getFullName(String firstName, String lastName) which should return full name of a person

Display person details with age and fullname.

IGATE
Speed.Agility.Imagination

Lab 4:Inheritance and Polymorphism

Goals	At the end of this lab session, you will be able to: ➢ Write a Java program that manipulates person details ➢ Working with Inheritance, Polymorphism
Time	120 minutes

4.1: Refer the case study 1in Page No: 5 and create Account Class as shown below in class diagram. Ensure minimum balance of INR 500 in a bank account is available.

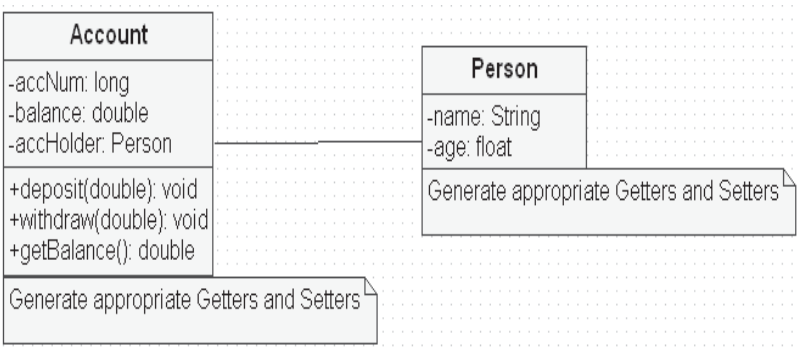


Figure 14: Association of person with account class

a) Create Account for smith with initial balance as INR 2000 and for Kathy with initial balance as 3000.(accNum should be auto generated).

b) Deposit 2000 INR to smith account.

c) Withdraw 2000 INR from Kathy account.

d) Display updated balances in both the account.

e) Generate toString() method.

IGATE
Speed.Agility.Imagination

4.2: Extend the functionality through Inheritanceand polymorphism (Maintenance)

Inherit two classes Savings Account and Current Account from account class. Implement the following in the respective classes.

a) Savings Account

a. Add a variable called minimum Balance and assign final modifier.

b. Override method called withdraw (This method should check for minimum balance and allow withdraw to happen)

b) Current Account

a. Add a variable called overdraft Limit

b. Overridemethod called withdraw (checks whether overdraft limit is reached and returns a boolean value accordingly)

IGATE
Speed.Agility.Imagination

Lab 5: Abstract classes and Interfaces

Goals	At the end of this lab session, you will be able to: ➢ Use of abstract classes and interfaces
Time	90 minutes

5.1: Refer the case study 2 in page no: 5 and create an application for that requirement by creating packages and classes as given below:

a) **com.igate.eis.bean**

In this package, create "Employee" class with different attributes such as id, name, salary, designation, insuranceScheme.

b) **com.igate.eis.service**

This package will contain code for services offered in Employee Insurance System. The service class will have one EmployeeService Interface and its corresponding implementation class.

c) **com.igate.eis.pl**

This package will contain code for getting input from user, produce expected output to the user and invoke services offered by the system.

The services offered by this application currently are:

i) Get employee details from user.

ii) Find the insurance scheme for an employee based on salary and designation.

iii) Display all the details of an employee.

5.2: Use overrides annotation for the overridden methods available in a derived class of an interface of all the assignments.

5.3: Refer the problem statement 4.1. Modify account class as abstract class and declare withdraw method.

IGATE
Speed.Agility.Imagination

Lab 6: Exception Handling

Goals	At the end of this lab session, you will be able to: ➢ Create and use application specific exceptions
Time	120 minutes

6.1: Modify the Lab assignment 2.3 to validate the full name of an employee. Create and throw a user defined exception if firstName and lastName is blank.

6.2: Validate the age of a person in Lab assignment 4.2 and display proper message by using user defined exception. Age of a person should be above 15.

6.3: Modify the Lab assignment 5.1 to handle exceptions. Create an Exception class named as "EmployeeException"(User defined Exception) in a package named as "com.igate.eis.exception" and throw an exception if salary of an employee is below than 3000. Use Exception Handling mechanism to handle exception properly.

Lab 7: Arrays and Collections

Goals	At the end of this lab session, you will be able to: ➢ Use Comparator interface ➢ Use Collections ➢ Use Generics with Collection Classes ➢ Use iterators to iterate through Collections
Time	180 minutes

7.1: Write a program to store product names in a string array and sort strings available in an array.

7.2: Modify the above program to store product names in anArrayList, sort strings available in an arrayList and display the names using for-each loop.

7.3: Modify the lab assignment 5.1 to accept multiple employee details and store all employee objects in a HashMap. The functionalities need to be implemented are:

i) Add employee details to HashMap.

ii) Accept insurance scheme from user and display employee details based on Insurance scheme

iii) Delete an employee details from map.

iv) Sort the employee details based on salary and display it.

Note: Use generics and Comparable/comparator interface.

Sample code Snippet of EmployeeServiceImpl class:

```
public class EmployeeServiceImpl {

HashMap<String,Employee> list = new HashMap<String,Employee>();

public void addEmployee(Employee emp)  {
                                        //code to add employee
}
public boolean deleteEmployee(int id)      {
//    code to delete a employee whose id is passed as parameter
}
……................
}
```


IGATE
Speed.Agility.Imagination

Lab 8: Files IO

Goals	At the end of this lab session, you will be able to: ➢ Read and write data using streams. ➢ Serialize and Deserialize objects
Time	75 minutes

8.1: Write a program to read content from file, reverse the content and write the reversed content to the file. (Use Reader and Writer APIs).

8.2: Create a file named as "numbers.txt" which should contain numbers from 0 to 10 delimited by comma. Write a program to read data from numbers.txt using Scanner class API and display only even numbers in the console.

8.3: Enhance the lab assignment 6.3 by adding functionality in service class to write employee objects into a File. Also read employee details from file and display the same in console. Analyze the output of the program.

Lab 9: Introduction to Junit

Goals	At the end of this lab session, you will be able to: ➢ Configuring JUnit in Eclipse ➢ Using JUnit to write TestCase for standalone Java Applications
Time	120 minutes

9.1: Configuration of JUnit in Eclipse

Step 1: Create a Java project.

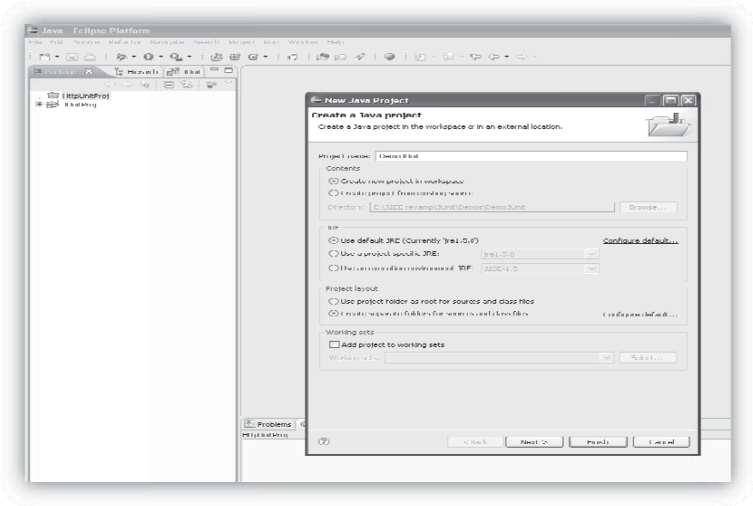


Figure 155: Creating Java Project in Eclipse

Step 2: Add **junit4.4.jar** in the build path of the project.

IGATE
Speed.Agility.Imagination

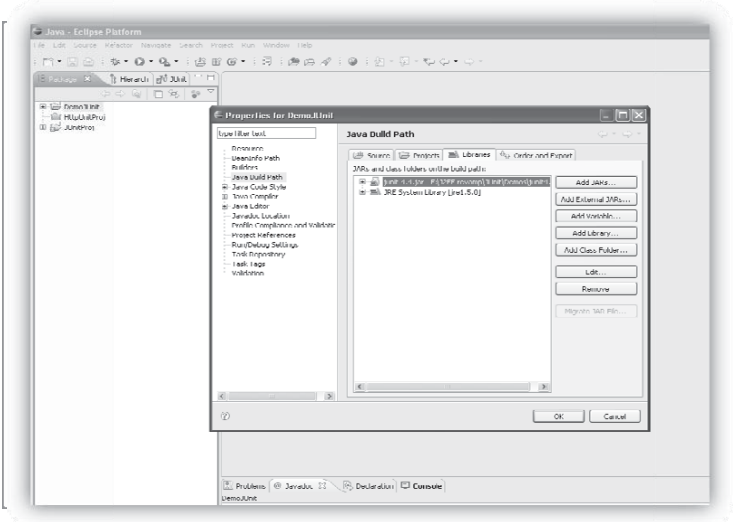


Figure 16: Adding junit4.4.jar in the build path

Step 3: Write the java class as follows:

```
public class  Person
{
        private String firstName;
        private String lastName;

        public Person(String fname,Stringlname)
        {
                if(fname == null &&lname==null){
                        throw new IllegalArgumentException("Both Names Cannot be
NULL");
                }
                this.firstName=fname;
                this.lastName = lname;
        }

public String getFullName()
{
        String first=(this.firstName != null)? this.firstName:"?";
        String last=(this.lastName != null)? this.lastName:"?";
        return first + " " + last;
```

```
}

public String getFirstName(){
        return this.firstName;
}

public String getLastName(){
        return this.lastName;
}
public static void main(String args[])
{
        Person p=new Person("a","b");
        System.out.println(p.getFirstName());
}
}
```

Example 1: Person.java

Step 4: Write the JUnit test class.

- Create a JUnit test case in Eclipse.

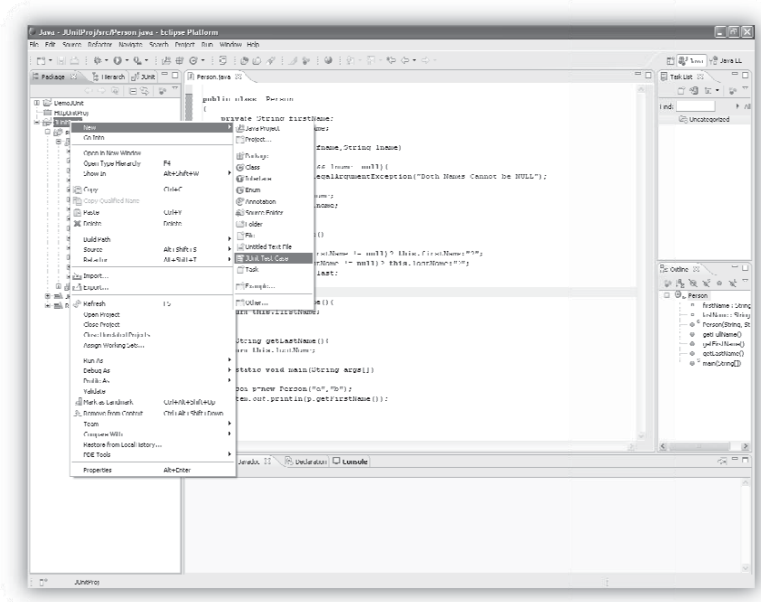


Figure 17: Adding the JUnit test case to the project

IGATE
Speed.Agility.Imagination

- A dialog box opens, where you need to specify the following details:
  - The Junit version that is used
  - The package name and the class name
  - The class under test
  - You can also specify the method stubs that you would like to create

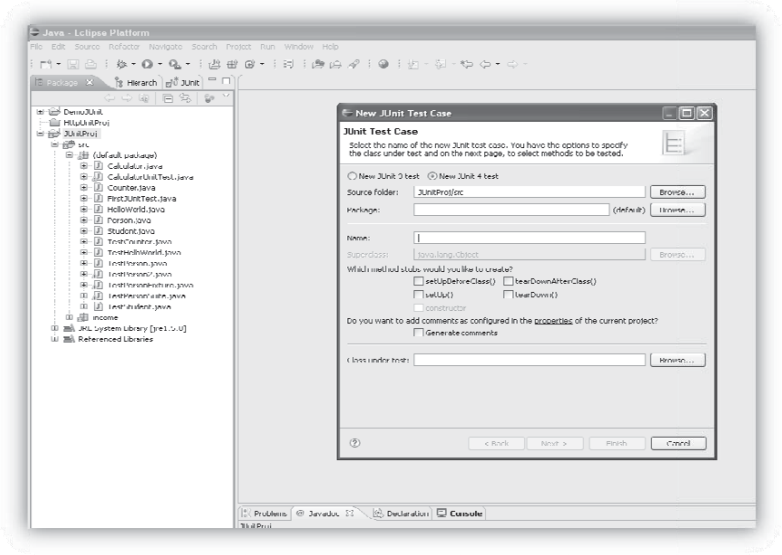


Figure 18: Specifying information for the test case

- Write the code as follows:

```
importorg.junit.*;
import org.junit.Test;
import static org.junit.Assert.*;

public class  TestPerson2
{
        @Test
        public void testGetFullName()
        {
                System.out.println("from TestPerson2");
```


IGATE
Speed.Agility.Imagination

```
                Person per = new Person("Robert","King");
                assertEquals("Robert King",per.getFullName());
        }

        @Test (expected=IllegalArgumentException.class)
        public void testNullsInName()
                {
                System.out.println("from TestPerson2 testing exceptions");
                Person per1 = new Person(null,null);
                }
}
```

Example 2: TestPerson2.java

Step 5: Run the test case.

- Right click the test case class, and select **RunAs→ JUnit Test**.
- The output will be displayed as shown below:

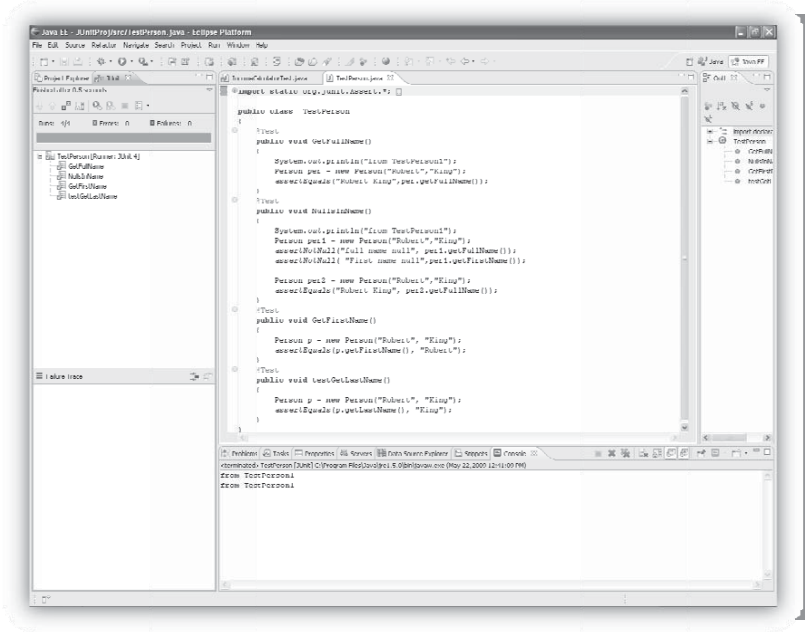


Figure 19: Output of JUnit text case execution

IGATE
Speed.Agility.Imagination

9.2: Writing JUnit tests

Consider the following Java program. Write tests for testing various methods in the class

Solution:

Step 1: Write the following Java Program **Date.java.**

```
class Date
   {
    intintDay, intMonth, intYear;
    // Constructor
    Date(int intDay, int intMonth, int intYear)  {
                this.intDay = intDay;
this.intMonth = intMonth;
this.intYear = intYear;
    }
    // setter and getter methods
    voidsetDay(int intDay)
    {
                this.intDay = intDay;
    }
                intgetDay( )
    {
                return  this.intDay;
    }

    voidsetMonth(int intMonth)
    {
                this.intMonth = intMonth;
    }

    intgetMonth( )
    {
                return  this.intMonth;
    }

    voidsetYear(int intYear)
    {
                this.intYear=intYear;
    }



    intgetYear( )
    {
                return  this.intYear;
    }
    public String toString() //converts date obj to string.
```


IGATE
Speed.Agility.Imagination

```
{
        return      "Date is "+intDay+"/"+intMonth+"/"+intYear;
}


} // Date class
```

Example 3: Date.java

Step 2: Write test class for testing all the methods of the above program and run it using the eclipse IDE.

9.2.1: Consider the Person class created in lab assignment 2.3. This class has some members and corresponding setter and getter methods. Write test case to check the functionality of getter methods and displaydetails method.

9.2.2: Consider the lab assignment 6.3 from Exception Handling Lab. Create a new class ExceptionCheck.javawhich handles an exception. Write a test case to verify if the exception is being handled correctly.

Lab 10: Lambda Expressions and Stream API

Goals	At the end of this lab session, you will be able to: ➢ Work with lambda expressions and stream API
Time	180 minutes

10.1: Write a lambda expression which accepts x and y numbers and return x^y.

10.2: Write a method that uses lambda expression to format a given string, where a space is inserted between each character of string. For ex., if input is "IGATE", then expected output is"I G A T E".

10.3: Write a method that uses lambda expression to accept username and password and return true or false. (**Hint:** Use any custom values for username and password for authentication)

10.4: Write a class with main method to demonstrate instance creation using method reference. (**Hint:** Create any simple class with attributes and getters and setters)

10.5: Write a method to calculate factorial of a number. Test this method using method reference feature.

<u>Case Study for Steam API:</u>

Refer the classes given below to represent employees and their departments.

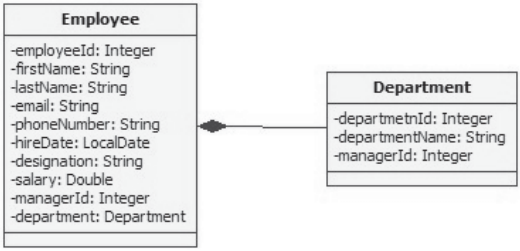


Figure 20: Class Diagram used for Stream API

Also refer an EmployeeRepository class which is used to create and populate employee's collection with sample data.

IGATE
Speed.Agility.Imagination

Create an EmployeeService class which queries on collections provided by EmployeeRepository class for following requirements. Create separate method for each requirement. (**Note:** Each requirement stated below must be attempted by using lambda expressions/stream API. It's mandatory to solve at least 5 questions from following set. However, it is recommended to solve all questions to understand stream API thoroughly).

10.6: Find out the sum of salary of all employees.

10.7: List out department names and count of employees in each department.

10.8:Find out the senior most employee of an organization.

10.9: List employee name and duration of their service in months and days.

10.10: Find out employees without department.

10.11: Find out department without employees.

10.12: Find departments with highest count of employees.

10.13: List employee name, hire date and day of week on which employee has started.

10.14: Revise exercise 10.13 to list employee name, hire date and day of week for employee started on Friday. (Hint: Accept the day name for e.g. FRIDAY and list all employees joined on Friday)

10.15: List employee's names and name of manager to whom he/she reports. Create a report in format "employee name reports to manager name".

10.16: List employee name, salary and salary increased by 15%.

10.17: Find employees who didn't report to anyone (**Hint:** Employees without manager)

10.18: Create a method to accept first name and last name of manager to print name of all his/her subordinates.

10.19: Sort employees by their

- Employee id
- Department id
- First name

IGATE
Speed.Agility.Imagination

Lab 11: Property Files and JDBC 4.0

Goals	At the end of this lab session, you will be able to: ➢ Understand how property files can be used. ➢ Use JDBC for connecting to the Database through DriverManager and DataSource
Time	240 minutes

11.1: Write a program to store a person details in a properties file named as "PersonProps.properties" and also do the following tasks:

a) Read data from properties file, load the data into Properties object and display the data in the console.

b) Read data from properties file(using getProperties method) and print data in the console.

11.2: Extend the assignment 7.3 by persisting data into database instead of hashmap and display/delete data from database. Use DriverManager for connecting to the database.

IGATE
Speed.Agility.Imagination

Lab 12: Introduction to Layered Architecture

Goals	At the end of this lab session, you will be able to: ➢ Develop an complete Java application in layered architecture
Time	300 minutes

12.1: Develop a Mobile Purchase system for a Mobile Sales shop. This application is a part of the system. Consider customer is doing full payment by cash, so payment details are not in the scope of our system.Assume mobile details are available in the table (TableName: mobiles). Each mobile detail have unique id and many quantity is available for each mobile. In this system,administratorshould be able to do the following process:

a) Insert the customer and purchase details into database

➢ Before inserting into database, do check that the quantity of the mobile should be greater than 0, else display error message.

b) Update the mobile quantity in mobiles table, once mobile is purchased by a customer.

c) View details of all mobiles available in the shop.

d) Delete a mobile details based on mobile id.

e) Search mobiles based on price range.

f) Write a test case for insert and search mobile service functionalities.

When a customer purchased a mobile, the customer and purchase details have to be inserted to the database through system. Perform the following validations while accepting customer details:

- **Customer name:** Valid value should contain maximum 20 alphabets. Out of 20 Characters, first character should be in UPPERCASE.
- **MailId:** should be valid mail id.
- **Phone number:** Valid value should contain 10 digits exactly.
- **MobileId:** Valid value should contain only 4 digits and it should be one of the mobileid available in mobiles table.
- **PurchaseId:** Generate automatically using sequence.
- **Purchasedate:** Should be the current system date.

IGATE
Speed.Agility.Imagination

Note:

1. Use layered architecture while implementing application
2. Handle all exceptions as a user defined exception.
3. Use Datasource for connecting to the database.
4. Read database details from properties file.
5. Use RegEx for performing validations.
6. Adhere to the coding standards and follow best practices.
7. Application should provide the menu options for the above requirements.

Assume mobile details are already available in the database.

Table Script to be used:

CREATE TABLE mobiles (mobileid NUMBER PRIMARY KEY, name VARCHAR2 (20), price NUMBER(10,2),quantity VARCHAR2(20));

INSERT INTO mobiles VALUES(1001,'Nokia Lumia 520',8000,20);
INSERT INTO mobiles VALUES(1002,'Samsung Galaxy IV',38000,40);
INSERT INTO mobiles VALUES(1003,'Sony xperia C',15000,30);
//TO DO – INSERT few more mobile details.

CREATE TABLE purchasedetails(purchaseid NUMBER, cname vARCHAR2(20), mailid VARCHAR2(30),phoneno VARCHAR2(20), purchasedate DATE, mobileid references mobiles(mobileid));

IGATE
Speed.Agility.Imagination

Lab 13: Log4J

Goals	At the end of this lab session, you will be able to: ➢ Use Loggers ➢ Use categories ➢ Use appenders ➢ Load Log4j properties file
Time	120 minutes

13.1: Use Loggers.

Solution:

Step 1: Create a directory structure as follows: **c:\demo\com\sample**

Step 2: Create the file c:\demo\com\sample\Log4jDemo.java.

```
packagecom.sample;
import org.apache.log4j.Logger;

public class Log4jDemo {

//create a logger for Log4jDemo class

public static void main(String args[]) {



        // create log messages for each priority level
    }
}
```

Example 4: Sample code

Step 3: Compile the Java code.

Step 4: Create file c:\demo\log4j.properties.

```
log4j.rootLogger=ERROR, stdout
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=[%5p] %d{mm:ss} (%F:%M:%L)%n%m%n%n
```

Example 5: Sample code

IGATE
Speed.Agility.Imagination

Step 5: Run your application:

The following log messages will be displayed:

```
[ERROR] 08:34 (Log4jDemo.java:main:13)
This is my error message.

[FATAL] 08:34 (Log4jDemo.java:main:14)
This is my fatal message.
```

Step 6: Change the following line in the log4j.properties file:

```
log4j.rootLogger=ALL, stdout
```

or

```
log4j.rootLogger=DEBUG, stdout
```

Step 7: Run your application.

The following log messages will be displayed:

```
[DEBUG] 27:42 (Log4jDemo.java:main:10)
This is my debug message.

[ INFO] 27:42 (Log4jDemo.java:main:11)
This is my info message.

[ WARN] 27:42 (Log4jDemo.java:main:12)
This is my warn message.

[ERROR] 27:42 (Log4jDemo.java:main:13)
This is my error message.

[FATAL] 27:42 (Log4jDemo.java:main:14)
This is my fatal message.
```

Step 8: Change the following line in the log4j.properties file:

log4j.rootLogger=OFF, stdout

Step 9: Run your application:

There will be no log messages displayed.

Step 10: Change the following line in the log4j.properties file:

```
log4j.rootLogger=FATAL, stdout
```


IGATE
Speed.Agility.Imagination

Step 11: Run your application.

The following log messages will be displayed:

[FATAL] 27:42 (Log4jDemo.java:main:14)
This is my fatal message.

13.2: Working with logger priority levels.

Solution:

Step 1: Create one more directory bean under sample

c:\demo\com\sample\bean.

Step 2: Create the file c:\demo\com\sample\bean\Message.java.

```
packagecom.sample.bean;
import org.apache.log4j.Logger;

public class Message {

//create a logger for Message class

private String msg;

public void setMessage(String msg) {
        this.msg = msg;

        //log the messages for each priority level
    }
public String getMessage() {

        //log messages for each priority level
returnmsg;
    }
}
```

Example 6: Sample code

Step 3: Create the file c:\demo\com\sample\Log4jDemo3.java.

```
packagecom.sample;
importcom.sample.bean.Message;
import org.apache.log4j.Logger;

public class Log4jDemo3 {

//create a logger for Log4jDemo3 class
public static void main(String args[]) {
```


IGATE
Speed.Agility.Imagination

```
//create an instance of Message class
//call setMessage() method
//print the log messages using getMessage() method
// write log message statements for each priority level
    }
}
```

Example 7: Sample code

Step 4: Compile the Java code for **Message.java** and **Log4Demo3.java**.

Step 5: Create file c:\demo\log4j.properties.

```
log4j.rootLogger=DEBUG, stdout

# Global Threshold - overridden by any Categories below.
log4j.appender.stdout.Threshold=WARN

# Categories
log4j.category.com.sample=FATAL
#log4j.category.com.sample.bean=INFO

log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%5p [%c{1}] %M - %m%n
```

Example 8: Sample code

Step 6: Run your application.

The following log messages will be displayed:

```
FATAL [Message] setMessage - This is my fatal message.
FATAL [Message] getMessage - This is my fatal message.
Hello World
FATAL [Log4jDemo3] main - This is my fatal message.
```

The category **com.sample** in line "log4j.category.com.sample=FATAL" is the parent of category **com.sample.bean**. Only FATAL messages are logged.

Step 7: Uncomment the following line in the log4j.properties file:

```
log4j.category.com.sample.bean=INFO
```

Step 8: Run your application **Log4jDemo3**:

IGATE
Speed.Agility.Imagination

The following log messages will be displayed:

```
WARN [Message] setMessage - This is my warn message.
ERROR [Message] setMessage - This is my error message.
FATAL [Message] setMessage - This is my fatal message.
WARN [Message] getMessage - This is my warn message.
ERROR [Message] getMessage - This is my error message.
FATAL [Message] getMessage - This is my fatal message.
Hello World
FATAL [Log4jDemo3] main - This is my fatal message.
```

INFO messages in category com.sample.bean are NOT logged. This is because of line "log4j.appender.stdout.Threshold=WARN"

This appender will not log any messages with priority lower than WARN even if the category's priority is set lower (INFO).

Step 9: Comment out the following lines in the **log4j.properties** file:

```
#log4j.category.com.sample=FATAL
#log4j.category.com.sample.bean=INFO
```

Step 10: Run your application **Log4jDemo3.**

The following log messages will be displayed:

```
WARN [Message] setMessage - This is my warn message.
ERROR [Message] setMessage - This is my error message.
FATAL [Message] setMessage - This is my fatal message.
WARN [Message] getMessage - This is my warn message.
ERROR [Message] getMessage - This is my error message.
FATAL [Message] getMessage - This is my fatal message.
Hello World
FATAL [Log4jDemo3] main - This is my fatal message.
```

All log messages should be displayed due to line "log4j.rootLogger=DEBUG, stdout". However because of line "log4j.appender.stdout.Threshold=WARN" only messages with priority WARN or higher are logged.

Step 11: Comment out the following lines in the log4j.properties file:

```
#log4j.appender.stdout.Threshold=WARN
#log4j.category.com.sample=FATAL
#log4j.category.com.sample.bean=INFO
```

Step 12: Run your application **Log4jDemo3.**

The following log messages will be displayed:

```
DEBUG [Message] setMessage - This is my debug message.
 INFO [Message] setMessage - This is my info message.
 WARN [Message] setMessage - This is my warn message.
```


IGATE
Speed.Agility.Imagination

```
ERROR [Message] setMessage - This is my error message.
FATAL [Message] setMessage - This is my fatal message.
DEBUG [Message] getMessage - This is my debug message.
 INFO [Message] getMessage - This is my info message.
 WARN [Message] getMessage - This is my warn message.
ERROR [Message] getMessage - This is my error message.
FATAL [Message] getMessage - This is my fatal message.
Hello World
DEBUG [Log4jDemo3] main - This is my debug message.
 INFO [Log4jDemo3] main - This is my info message.
 WARN [Log4jDemo3] main - This is my warn message.
ERROR [Log4jDemo3] main - This is my error message.
FATAL [Log4jDemo3] main - This is my fatal message.
```

13.3: Use Appenders.

Solution:

Step 1: Create a directory structure as follows: **c:\demo\com\sample**

Step 2: Create the file c:\demo\com\sample\Log4jDemo2.java.

```
packagecom.sample;
import org.apache.log4j.Logger;

public class Log4jDemo2 {

//create a logger for Log4jDemo2 class

public static void main(String args[]) {

for(int i=1 ; i<50000; i++) {
System.out.println("Counter = " + i);
log.debug("This is my debug message. Counter = " + i);
            // write log message statements for remaining priority levels
                        //in the same way
        }
    }
}
```

Example 9: Sample code

Step 3: Compile the java code for **Log4jDemo2.java**

Step 4: Create file c:\demo\log4j.properties and define several appenders:

```
log4j.rootLogger=ERROR, A2

########## Appender A1
log4j.appender.A1=org.apache.log4j.ConsoleAppender
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=[%5p] %d{mm:ss} (%F:%M:%L)%n%m%n%n
```


IGATE
Speed.Agility.Imagination

```
########## Appender A2
log4j.appender.A2=org.apache.log4j.FileAppender
log4j.appender.A2.File=c:/demo/app_a2.log

# Append to the end of the file or overwrites the file at start.

log4j.appender.A2.Append=false
log4j.appender.A2.layout=org.apache.log4j.PatternLayout
log4j.appender.A2.layout.ConversionPattern=[%5p] %d{mm:ss} (%F:%M:%L)%n%m%n%n

########## Appender A3

log4j.appender.A3=org.apache.log4j.RollingFileAppender
log4j.appender.A3.File=c:/demo/app_a3.log
# Set the maximum log file size (use KB, MB or GB)
log4j.appender.A3.MaxFileSize=3000KB
# Set the number of log files (0 means no backup files at all)
log4j.appender.A3.MaxBackupIndex=5
# Append to the end of the file or overwrites the file at start.
log4j.appender.A3.Append=false
log4j.appender.A3.layout=org.apache.log4j.PatternLayout
log4j.appender.A3.layout.ConversionPattern=[%5p] %d{mm:ss} (%F:%M:%L)%n%m%n%n

########## Appender A4

log4j.appender.A4=org.apache.log4j.DailyRollingFileAppender
log4j.appender.A4.File=c:/demo/app_a4.log
# Roll the log file at a certain time
log4j.appender.A4.DatePattern='.'yyyy-MM-dd-HH-mm
# Append to the end of the file or overwrites the file at start.
log4j.appender.A4.Append=false
log4j.appender.A4.layout=org.apache.log4j.PatternLayout
log4j.appender.A4.layout.ConversionPattern=[%5p] %d{mm:ss} (%F:%M:%L)%n%m%n%n
```

Example 10: Sample code

Note: Use forward slashes in log4j.appender.A2.File=c:/demo/app_a2.log.

Step 5: Demonstrate **FileAppender**, and run your application:

By using **appender A2** no log messages are displayed on the console, and all log messages are written to one large log file C:\demo\app_a2.log:

Step 6: Demonstrate **FileAppender**, and run your application:

Demonstrate RollingFileAppender, change the following line in c:\demo\log4j.properties:
log4j.rootLogger=ERROR, A3

IGATE
Speed.Agility.Imagination

Step 7: Run your application.
By using **appender A3**, the log file app_a3.log will be rolled over when it reaches 3000KB. When the roll-over occurs, the app_a3.log is automatically moved to app_a3.log.1. When app_a3.log again reaches 3000KB, app_a3.log.1 is moved to app_a3.log.2 and app_a3.log is moved to app_a3.log.1.

The maximum number of backup log files is set to MaxBackupIndex=5, which means app_a3.log.5 is the last file created.

Step 8: Demonstrate **DailyRollingFileAppender**, change the following line in c:\demo\log4j.properties:

log4j.rootLogger=ERROR, A4

Step 9: Run your application.
By using **appender A4**, the log file app_a4.log will be rolled over depending on the date format (= Java SimpleDateFormat) used.

<<TO DO>>

13.3.1: Assign a layout to an appender in the log4j.properties configuration file and see the results.

13.3.2: Define the appenders in **log4j.properties** and use it to get the desired result.

13.4: Loading Log4J.properties file.

Solution:

Step 1: In a standalone application the **log4j.properties** must be put in the directory where you issued the java command.

Step 2: Run your application.

```
public class HelloWorld {
static final Logger logger = Logger.getLogger(HelloWorld.class);
public static void main(String[] args) {
PropertyConfigurator.configure("log4j.properties");
logger.debug("Hello World!");
logger.warn("Sample warn message");
logger.error("Sample error message");
}
}
```

Example 11: Sample code

If you rename **log4j.properties** file to something else (for example: **test.properties**), you must add the following line to your Java runtime command:

IGATE
Speed.Agility.Imagination

-Dlog4j.configuration=test.properties

<<TO DO>>

Assignment 3: Rename the **Log4j.properties** to **testfile.properties** and execute the application.

13.5 Refer the Mobile Purchase layered application from lab 12.1. Configure the logger for following functionalities:

13.5.1: Log details of customer and mobile when mobile is purchased successfully.

13.5.2: Log message when the mobile deleted.

13.5.3: Log search criteria details upon each search request from user.

13.5.3: Log all error messages/exceptions

Lab 14: JAXB

Goals	At the end of this lab session, you will be able to: ➢ To process XML contents using JAXB.
Time	90 minutes

14.1: Write a program to do the following operations using JAXB:

- Read mobile details from database table and write it into XML file.
- Read mobile details from XML and display it in console.
- Get mobile id from user, search mobile details in XML file based on mobile id and display details if mobile id is found.

14.2: Refer the above Employee case study given in Stream API lab. Export data of employees in XML format, storedepartment wise employees in separate XML file with file name as name of department.

IGATE
Speed.Agility.Imagination

Appendices

Appendix A: Naming Conventions

Package names are written in all lower case to avoid conflict with the names of classes or interfaces.Companies use their reversed Internet domain name to begin their package names—for example, com.igate.mypackage for a package named mypackage created by a programmer at igate.com.

Packages in the Java language itself begin with **java. orjavax**.

Classes and interfaces The first letter should be capitalized, and if several words are linked together to form the name, the first letter of the inner words should be uppercase (a format that's sometimes called "camelCase").

For classes, the names should typically be nouns. For example:

Dog

Account

PrintWriter

For interfaces, the names should typically be adjectives like

Runnable

Serializable

Methods The first letter should be lowercase, and then normal camelCase rules should be used. In addition, the names should typically be verb-noun pairs. For example:

getBalance

doCalculation

setCustomerName

Variables Like methods, the camelCase format should be used, starting with a lowercase letter. Sun recommends short, meaningful names, which sounds good to us. Some examples:

buttonWidth

accountBalance

myString

Constants Java constants are created by marking variables static and final. They should be named using uppercase letters with underscore characters as separators:

MIN_HEIGHT

IGATE
Speed.Agility.Imagination

Appendix B: Table of Figures

Figure 1: Java program 6
Figure 2: System Properties 7
Figure 3: Environment Variables 7
Figure 4: Edit System Variable 8
Figure 5: Edit System Variable 8
Figure 6: Edit User Variable 8
Figure 7: Select Wizard 9
Figure 8: New Java Project 9
Figure 9: Java Settings 10
Figure 10: Select Resource 10
Figure 11: Java Class 11
Figure 12: Sample output of Person details 14
Figure 13: Class Diagram of Person 14
Figure 14: Association of person with account class 17