

Fundamental Programming in Java

Fundamental Programming in Java Trainer's Guide

© 2017 Aptech Limited

All rights reserved.

No part of this book may be reproduced or copied in any form or by any means – graphic, electronic or mechanical, including photocopying, recording, taping, or storing in information retrieval system or sent or transferred without the prior written permission of copyright owner Aptech Limited.

All trademarks acknowledged.

APTECH LIMITED

Contact E-mail: customercare@aptech.ac.in

First Edition - 2017



“

A little learning is a dangerous thing,
but a lot of ignorance is just as bad.

”

For Aptech Centre Use Only

Preface

The book 'Fundamental Programming in Java' Trainer's Guide aims to teach the basics of Java programming language. The book teaches the various Java features and development of Java applications based on object-oriented programming methodology. The book also teaches the enhancements in the Java Standard Edition (Java SE) version 8.

The faculty/trainer should teach the concepts in the theory class using the slides. This Trainer's Guide will provide guidance on the flow of the session and also provide tips and additional examples wherever necessary. The trainer can ask questions to make the session interactive and also to test the understanding of the students.

This book is the result of a concentrated effort of the Design Team, which is continuously striving to bring you the best and the latest in Information Technology. The process of design has been a part of the ISO 9001 Certification for Aptech-IT Division, Education Support Services. As part of Aptech's quality drive, this team does intensive research and curriculum enrichment to keep it in line with industry trends.

We will be glad to receive your suggestions.

Design Team

“ Practice is the best of
all instructors.

For Aptech Centre Use Only

Table of Contents

Sessions

1. Introduction to Java
2. Application Development in Java
3. Variables and Operators
4. Decision-Making Constructs
5. Looping Constructs
6. Classes and Objects
7. Methods and Access Specifiers
8. Arrays and Strings
9. Modifiers and Packages
10. Inheritance and Polymorphism
11. Interfaces and Nested Classes
12. Exceptions
13. New Date and Time API
14. Annotations and Base64 Encoding
15. Functional Programming in Java
16. Stream API
17. More on Functional Programming
18. Additional Features of Java 8

**“ The future depends on what
we do in the present.**

For Aptech Centre Use Only

Session 1 – Introduction to Java

1.1 Pre-Class Activities

Before you commence the session, you should familiarize yourself with the topics of this session in-depth.

1.1.1 Objectives

By the end of this session, the learners will be able to:

- Explain the structured programming paradigm
- Explain the object-oriented programming paradigm
- Explain the features of Java as a OOP language
- Describe Java platform and its components
- List the different editions of Java
- Explain the evolution of Java Standard Edition (Java SE)
- Describe the steps for downloading and installing Java Development Kit (JDK)

1.1.2 Teaching Skills

To teach this session, you should be well-versed with object-oriented paradigm which is used as a solution to develop applications that are modeled to real world entities called objects and classes. Also, you should be aware with Java which is an Object-Oriented Programming Language (OOP) to develop platform independent applications.

You should teach the concepts in the theory class using the images provided. For teaching in the class, you are expected to use slides and LCD projectors.

Tips:

It is recommended that you test the understanding of the students by asking questions in between the class.

In-Class Activities:

Follow the order given here during In-Class activities.

Overview of the Session:

Give the students the overview of the current session in the form of session objectives. Show the students slide 2 of the presentation.

Objectives
<ul style="list-style-type: none">◆ Explain the structured programming paradigm◆ Explain the object-oriented programming paradigm◆ Explain the features of Java as a OOP language◆ Describe Java platform and its components◆ List the different editions of Java◆ Explain the evolution of Java Standard Edition (Java SE)◆ Describe the steps for downloading and installing Java Development Kit (JDK)

Tell the students that this session introduces them to Java as an OOP language as well as a platform for executing Java applications. This session introduces them to the features of OOP language. They will understand the two most important entities of OOP namely class and object which form the base for application designing in OOP languages. Further, the session explains them about the different editions of Java and components of Java Development Kit. Finally, they will learn how to install JDK and set its path on the system.

1.2 In-Class Explanations

Slide 3

Let us understand software applications in computers.

Introduction 1-3

- ◆ The most prominent use of computers is to solve problems quickly and accurately.
- ◆ The solution adopted to solve a problem is provided as a sequence of instructions or specifications of activity which enables a user to achieve the desired result.

Software Applications

- The solution for solving a problem in the field of information technology is achieved by developing software applications.
- A software application can be defined as a collection of programs that are written in high-level programming languages to solve a particular problem.

3

Using slide 3, explain the use of software development as a solution in information technology.

The process of problem solving using computers is complex and needs a proper planning and logical thinking. The solution adopted to solve a problem is provided as a sequence of instructions or specifications of activity which enables a user to achieve the desired result.

Software Applications

The solution for solving a problem in the field of information technology is achieved by developing software applications. A software application can be defined as a collection of programs that are written in high-level programming languages to solve a particular problem.

Discuss with the students about the different types of high-level programming languages and their features in developing software applications. During discussion, you can provide examples of different languages used for application development for different purposes. For example, to develop a Web site, you work with Web languages, such as HTML, JavaScript, PHP, or JSP. To develop a Product-Order form to be executed on Windows platform, you can use Visual Basic as it provided rich user interface controls such as text boxes, command buttons, check boxes, and so on. To develop compilers or driver software for the devices, C language is preferred.

Similarly, Java is a fundamental object-oriented programming language used for developing applications that can be executed on any platform.

In-Class Question:

After you finish explaining, the concept of software applications, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



Mention some of the examples where C language can be used?

Answer:

- Operating systems
- Compilers
- Text editors
- Network drivers
- Utility programs

Slide 4

Let us discuss on domain and its knowledge in application development.

Introduction 2-3

◆ **Knowledge of Domain:**

- ❖ It plays an important role while developing software applications.
- ❖ It can be defined as the field of business or technology to which a problem belongs.
- ❖ Following figure shows the development of software application as a solution for managing various operations in a banking domain:

© Aptech Ltd. Introduction to Java/Session 1 4

Using slide 4, explain the knowledge of domain.

Domain plays an important role while developing software applications. It can be defined as the field of business or technology to which a problem belongs. For example, in ordering the food from a restaurant, the knowledge of all the restaurants in nearby area forms the domain knowledge.

Domain – Restaurant.

Domain knowledge - Type of food served.

The knowledge about the restaurant as well as the type of food served can also be considered as domain knowledge in this case. Having a good knowledge of the domain helps to solve the problem quickly. In the scenario described, if the restaurant specializes in Mexican cuisine and you call them to order Chinese food, then it would be a waste of time and effort. Here, if you had known the name of the nearest restaurant that serves only Chinese food, it would have saved your time and effort. Hence, having adequate domain knowledge before designing a problem solution is always recommended.

Similarly, explain another scenario for domain and its knowledge to the students. Tell them about the different processes in bank related to accounts, customer, payments, and so on.

The software application developed to handle the bank operations based on the domain needs will be the solution.

Slide 5

Let us understand on different programming language paradigms.

Introduction 3-3

Programming Languages:

- ❖ The development of software application is done using a programming language.
- ❖ A programming language is used as a medium for communicating the instruction to the computer.
- ❖ The programming language enforces a particular style of programming that is referred to as a programming paradigm.
- ❖ Following are the two types of programming paradigm:

Structured Programming Paradigm

Object-oriented Programming Paradigm

© Aptech Ltd. Introduction to Java/Session 1 5

Using slide 5, explain the use of programming languages and their programming style or methodology to develop an application.

The development of software application is done using a programming language. A programming language is used as a medium for communicating the instruction to the computer. They enforce a particular style or approach to be followed while application development referred to as a programming paradigm.

Normally, there are two types of programming paradigms:

- Structured programming paradigm
- Object-oriented programming paradigm

Tell them earlier programming language such as C, Pascal, and COBOL followed structured programming paradigm where application development is decomposed into hierarchy of sub-programs.

The modern programming languages such as Smalltalk, Java, .Net languages such as C# are based on object-oriented paradigm or methodology. The object-oriented methodology enables application designers and developers to think in terms of objects. For example, in the banking domain, each entity such as customer, account, or so on are the objects similar to real-world objects.

Tips:

All object-oriented programming languages, such as C++ use the concepts of class and objects.

Slides 6 and 7

Let us understand structured programming paradigm and its use.

Structured Programming Paradigm 1-2

Structured Programming

- In structured programming paradigm, the application development is decomposed into a hierarchy of subprograms.
- The subprograms are referred to as procedures, functions, or modules in different structured programming languages.
- Each subprogram is defined to perform a specific task.
- Some of structured programming languages are C, Pascal, and Cobol.

Following figure displays bank application activities broken down into subprograms:

```

graph TD
    A[Manage Bank Activities] --> B[Raise Interest]
    A --> C[Customer Transaction]
    B --> D[Raise Interest for Saving Accounts]
    B --> E[Raise Interest for Other Accounts]
    C --> F[Deposit]
    C --> G[Withdraw]
    C --> H[Fixed Deposit]
  
```

© Aptech Ltd. Introduction to Java / Session 1 6

Structured Programming Paradigm 2-2

Main disadvantage of structured programming languages are as follows:

- Data is shared globally between the subprograms.
- Efforts are spent on accomplishing the solution rather than focusing on problem domain.

This often led to a software crisis, as the maintenance cost of complex applications became high and availability of reliable software was reduced.

A central blue circle labeled "Data" contains binary code. Three arrows point from this central circle to three separate subprograms: SubProgram 1, SubProgram 2, and SubProgram 3. Each subprogram has a variable X assigned a value (X=5, X=10, and X=15 respectively).

© Aptech Ltd. Introduction to Java / Session 1 7

Using slides 6 and 7, explain structured programming paradigm.

A software application developed as a solution can break the problem into smaller tasks which can be developed separately and then assembled to solve a particular problem. Example, suppose you want to design a loan calculator for the banking domain which will help its customers to get the loan amount and its interest rate to be paid by him. Each task in developing the loan calculator can be divided into modules or procedures or function, based on which structured language is used for developing the application.

Structured Programming

Structured programming paradigm aimed on improving the clarity, quality, and development time of computer program by making extensive use of sub-routines and block structures. It mainly used to solve complex problems, however, it had paid very less attention on how data is used between the sub-routines. Efficiency of programs is measured on time and memory occupancy. The application development is decomposed into a hierarchy of subprograms. The subprograms are referred to as procedures, functions, or modules in different structured programming languages.

Each subprogram is defined to perform a specific task. Some of structured programming languages are C, Pascal, and COBOL.

Main disadvantage of structured programming languages are as follows:

- Data is shared globally between the subprograms.
- Efforts are spent on accomplishing the solution rather than focusing on problem domain.

This often led to a software crisis, as the maintenance cost of complex applications became high and availability of reliable software was reduced.

Slides 8 to 11

Let us understand object-oriented programming paradigm and its features.

Object-oriented Programming Paradigm 1-4



- ◆ Growing complexity of software required change in programming style.
- ◆ Some of the features that were aimed are as follows:
 - ◆ Development of reliable software at reduced cost.
 - ◆ Reduction in the maintenance cost.
 - ◆ Development of reusable software components.
 - ◆ Completion of software development with the specified time interval.
- ◆ These features resulted in the evolution of **object-oriented programming paradigm**.

© Aptech Ltd. Introduction to Java / Session 1 8

Object-oriented Programming Paradigm 2-4



- ◆ The software applications developed using object-oriented programming paradigm is:
 - ◆ Designed around data, rather than focusing only on the functionalities.
- ◆ Following shows different activities involved in the object-oriented software development:

Object Oriented Development =

Object Oriented Analysis

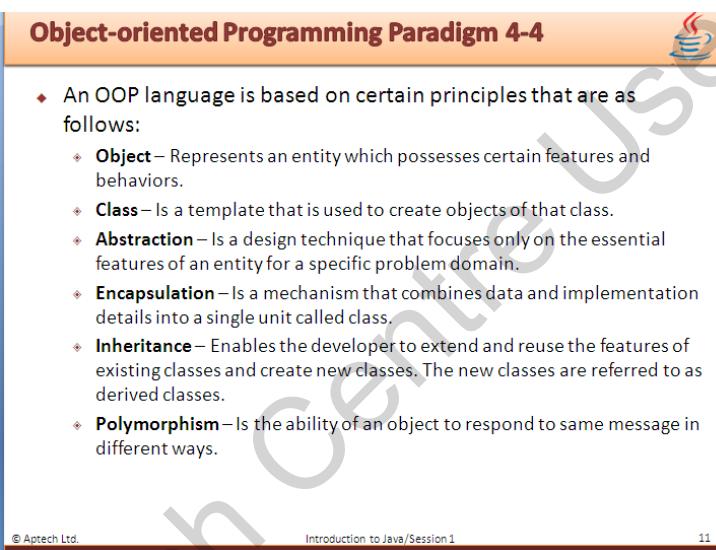
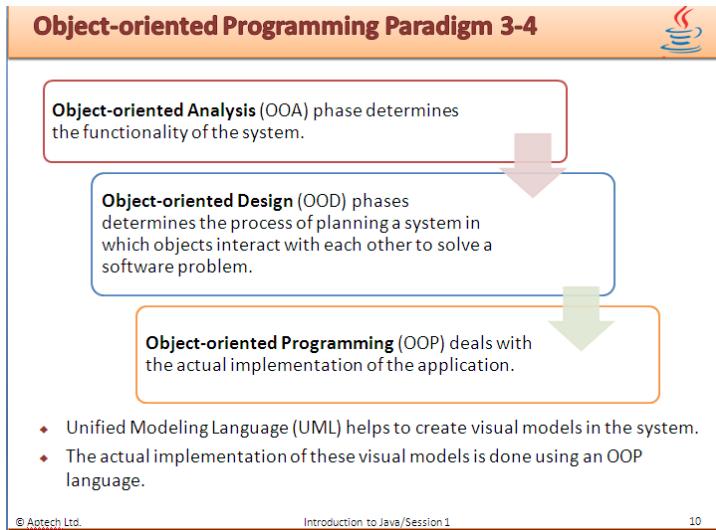
+

Object Oriented Design

+

Object Oriented Programming

© Aptech Ltd. Introduction to Java / Session 1 9



Using slides 8 to 11, explain the object-oriented programming paradigm and its features.

Object-oriented programming paradigm represents the concept of objects that have data fields and associated procedures known as methods. It is an approach to design modular, reusable software components. It integrates code and data by using the concept of object. Growing complexity of software required change in programming style. Some of the features that were aimed are as follows:

- Development of reliable software at reduced cost.
- Reduction in the maintenance cost.
- Development of reusable software components.
- Completion of software development with the specified time interval.

These features resulted in the evolution of object-oriented programming paradigm.

The software applications developed using object-oriented programming paradigm is designed around data, rather than focusing only on the functionalities.

There are different types of activities involved in the object-oriented software development. They are: object-oriented analysis, object-oriented design, and object-oriented programming.

- Object-oriented Analysis (OOA) phase determines the functionality of the system.
- Object-oriented Design (OOD) phases determine the process of planning a system in which objects interact with each other to solve a software problem.
- Object-oriented Programming (OOP) deals with the actual implementation of the application.

Unified Modeling Language (UML) helps to create visual models in the system, while performing the analysis of the OOP project. The actual implementation of these visual models is done using an OOP language.

Then, explain the features of OOP languages which are based on certain principles that are as follows:

- **Object** – Represents an entity which possesses certain features and behaviors.
- **Class** – Is a template that is used to create objects of that class.
- **Abstraction** – Is a design technique that focuses only on the essential features of an entity for a specific problem domain.
- **Encapsulation** – It refers to the creation of self-contained module that binds processing functions to the data. It is a mechanism that combines data and implementation details into a single unit called class.
- **Inheritance** – Enables the developer to extend and reuse the features of existing classes and create new classes. The new classes are referred to as derived classes.
- **Polymorphism** – Is the ability of an object to respond to same message in different ways.

In-Class Question:

After you finish explaining Object-oriented programming paradigm, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



What is known as abstraction?

Answer:

An abstraction is a design technique that focuses only on the essential features of an entity based on the problem domain. It helps the designer to focus only on the necessary features of the object relevant to its domain.

Slides 12 to 15

Let us understand concept of an object.

Concept of an Object 1-4

- ◆ An object represents a real-world entity.
- ◆ Any tangible or touchable entity in the real-world can be described as an object.
- ◆ Following figure shows some real-world entities:

Car Flower
Bicycle Person

© Aptech Ltd. Introduction to Java/Session 1 12

Concept of an Object 2-4

- ◆ Each object has:
 - ◆ **Characteristics** – Defined as attributes, properties, or features describing the object.
 - ◆ **Actions** – Defined as activities or operations performed by the object.
- ◆ Example of an object, Dog.
 - ◆ **Properties** – Breed, Color, and Age
 - ◆ **Actions** – Barking, Eating, and Running
- ◆ The concept of objects in the real-world can be extended to the programming world where software 'objects' can be defined.

© Aptech Ltd. Introduction to Java/Session 1 13

Concept of an Object 3-4

- ◆ A software object has state and behavior.
- ◆ 'State' refers to object's characteristics or attributes.
- ◆ 'Behavior' of the software object comprises its actions.
- ◆ Following figure shows a software object, a Car with its state and behavior:

State	Behavior
Color	Drive
Make	Change gear
Model	Increase speed
	Apply Brakes

© Aptech Ltd. Introduction to Java/Session 1 14

Concept of an Object 4-4



- ◆ The advantages of using objects are as follows:

1
2

- They help to understand the real-world.
- They map attributes and actions of real-world entities with state and behavior of software objects.

© Aptech Ltd. Introduction to Java/Session 1 15

Using slides 12 to 15, explain the concept of object and its anatomy.

An object represents a real-world entity. Any tangible or touchable entity in the real-world can be described as an object. In object-oriented programming paradigm an object refers to a particular instance of class where it can be combination of variables, functions, and data structures.

Each object has:

- **Characteristics** – Defined as attributes, properties, or features describing the object.
- **Actions** – Defined as activities or operations performed by the object.

The concept of objects in the real-world can be extended to the programming world where software ‘objects’ can be defined. Software object has state and behavior.

- ‘**State**’ refers to object’s characteristics or attributes.
- ‘**Behavior**’ of the software object comprises its actions.

The advantages of using objects are as follows:

- They help to understand the real-world.
- They map attributes and actions of real-world entities with state and behavior of software objects.
- A program can carry out various tasks such as implementing GUI, sending, and receiving information over network.

In the real-world, several objects have common state and behavior. For Example: all car objects have attributes, such as color, make, or model.

In-Class Question:

After you finish explaining, the concept of software object, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



Identify some of the real-world objects around you.

Answer:

Desk, Books, Pen, and so on.

Slides 16 to 18

Let us understand designing a class.

Defining a Class 1-3

- ◆ In the real-world, several objects:
 - ❖ Have common state and behavior.
 - ❖ Can be grouped under a single class.
 - ❖ Example: All car objects have attributes, such as color, make, or model.
- ◆ Class:
 - ❖ Can be defined that a class is a template or blueprint which defines the state and behavior for all objects belonging to that class.
 - ❖ Following figure shows a car as a template and a Toyota car as an object or instance of the class:

© Aptech Ltd. Introduction to Java / Session 1 16

Defining a Class 2-3

- ◆ Class comprises fields and methods, collectively called as members.
 - ❖ Fields – Are variables that depict the state of objects.
 - ❖ Methods – Are functions that depict the behavior of objects.

© Aptech Ltd. Introduction to Java / Session 1 17

Defining a Class 3-3

- ◆ Following table shows the difference between a class and an object:

Class	Object
Class is a conceptual model	Object is a real thing
Class describes an entity	Object is the actual entity
Class consists of fields (data members) and functions	Object is an instance of a class

© Aptech Ltd. Introduction to Java / Session 1 18

Using slides 16 to 18, explain the class design and its members.

Several objects having common state and behavior and can be grouped under a single class. For example, the class **Car** can be a general class with many features, whereas **Toyota Car** is an object of the class which is specific in features.

Tell the students that a class is a template which defines the state and behavior for all objects belonging to that class. A class comprises fields and methods which can be mapped to state and behavior respectively. These fields and methods are collectively known as members of a class.

- **Fields** – Are variables that depict the state of objects. There is one field for each object of a class.
- **Methods** – Are functions that depict the behavior of objects.

Tell the students that the class is a conceptual model, is used for describing an entity and consists of fields and methods. On the other hand, an object is a real thing, it is an actual entity, and it is an instance of a class.

Tips:

A class is a general concept whereas an object is a specific embodiment of that class and has a limited lifespan. A class provides the template for something more specific that the programmer has to define.

Slide 19

Let us understand Java.

Java

- ◆ It is one of the most popular OOP language.
- ◆ It helps programmers to develop wide range of applications that can run on various hardware and Operating System (OS).
- ◆ It is also a platform that creates an environment for executing Java application.
- ◆ It caters to small-scale to large-scale problems across the Internet.
- ◆ Java applications are built on variety of platforms that range from:
 - ◆ Embedded devices to desktop applications
 - ◆ Web applications to mobile phones
 - ◆ Large business applications to supercomputers

© Aptech Ltd. Introduction to Java/Session 1 19

Using slide 19, explain the Java programming language.

Java is object-oriented programming language which is specifically designed to have implementation dependencies as possible. It is one of the most popular OOP languages. It helps programmers to develop wide range of applications that can run on various hardware and Operating System (OS). It is also a platform that creates an environment for executing Java application.

Java caters the need of small-scale to large-scale business problems across the Internet. It is compiled to byte code and most popular for the client-server Web applications.

Java applications are built on variety of platforms that range from:

- Embedded devices to desktop applications
- Web applications to mobile phones
- Large business applications to supercomputers

Java platform is a name for bundle of programs that allow developing and running programs in Java programming language.

Slides 20 to 22

Let us understand history of Java.

History of Java 1-3

◆ **Java Origins: Embedded Systems**

1991

- Team of engineers from Sun Microsystems wanted to design a language for consumer devices.
- Project was named as 'Green Project'.
- Team included: James Gosling, Mike Sheridan, and Patrick Naughton.
- Efforts were taken to produce portable and a platform independent language that can run on any machine.
- Result was evolution of Java.
- Initially called 'OAK' and later renamed to Java.

© Aptech Ltd. Introduction to Java/Session 1 20

History of Java 2-3

◆ **Java Wonder: Internet**

1995

- Internet and Web started emerging and was used worldwide.
- Sun Microsystems turned Java into an Internet programming language.
- It emerged as a Web technology that added dynamic capabilities to the Web pages.

◆ **Java Moved: Middle-tier**

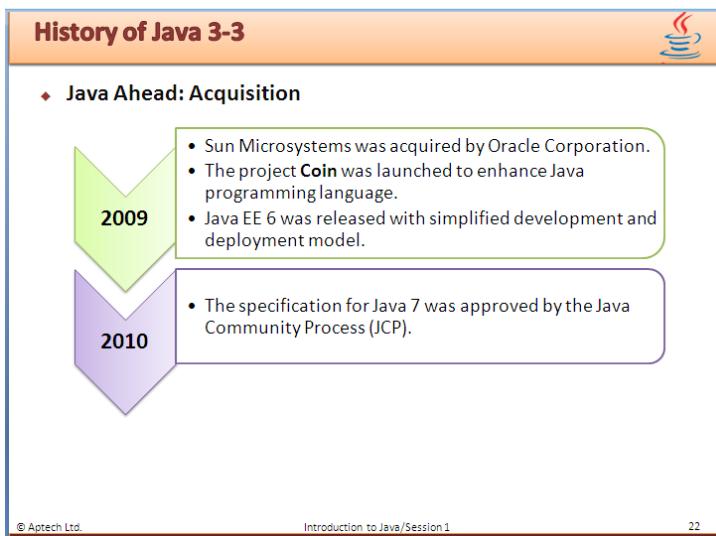
1997

- Sun Microsystems defined Servlets API to generate dynamic HTML for Web pages and Enterprise JavaBeans for developing business logics.

2006

- Sun Microsystems released three versions for free under General Public License (GPL)
- These are: Java Platform, Standard Edition (Java SE), Java Platform, Enterprise Edition (Java EE), and Java Platform Micro Edition (Java ME)

© Aptech Ltd. Introduction to Java/Session 1 21



Using slides 20 to 22, describe the history of Java.

Java Origins: Embedded Systems

◆ 1991

Mention that in the period from 1991–1994, the basic objective behind the development of the language, was to create software that could be embedded in consumer electronics. In C and C++, different compilers were required for a particular CPU which was expensive. Thus, efforts were made to develop a portable, platform independent language that could run on different CPUs under different environments. Team included: James Gosling, Mike Sheridan, and Patrick Naughton. Efforts were taken to produce portable and a platform independent language that can run on any machine. Result was evolution of Java. Initially called 'OAK' and later renamed to Java.

Java Wonder: Internet

◆ 1995

Sun Microsystems releases the first implementation as Java. Internet and Web started emerging and was used worldwide. Sun Microsystems turned Java into an Internet programming language. It emerged as a Web technology that added dynamic capabilities to the Web pages. Later, Java became an ideal software for network computers.

Java Moved: Middle-tier

Later, that is in late 90's, Sun introduced middle-tier capabilities to Java to ensure that it runs on Web/Application servers. In 1999, Sun offered middle-tier solution for Java called Java 2 Enterprise Edition, also known as J2EE.

◆ 1997

Sun Microsystems defined Servlets API to generate dynamic HTML for Web pages and Enterprise JavaBeans for developing business logics.

◆ 2006

Sun Microsystems released three versions for free under General Public License (GPL). These are: Java Platform, Standard Edition (Java SE), Java Platform, Enterprise Edition (Java EE), and Java Platform Micro Edition (Java ME).

Java Ahead: Acquisition**◆ 2009**

Sun Microsystems was acquired by Oracle Corporation. The project Coin was launched to enhance Java programming language. Java EE 6 was released with simplified development and deployment model.

◆ 2010

The specification for Java 7 was approved by the Java Community Process (JCP). Java software runs on everything from laptops to data centers and game consoles.

In-Class Question:

After you finish explaining history of Java, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



Why was servlets API defined by Sun Microsystems?

Answer:

It was defined to generate dynamic HTML for Web pages and Enterprise JavaBeans for developing business logics.

Slides 23 to 27

Let us understand the features of Java language.

Java: Object-oriented Programming Language 1-5

Java is a high level OOP language as well a platform used for developing applications that can be executed on different platforms.

It is characterized by following features:

Simple

- Inherits its syntax from predecessor programming languages, such as C/C++.
- Helps programmers to adapt to Java language without any extra skills and extensive trainings.
- Eliminates the use of pointers, operator overloading, and multiple inheritance features supported by predecessor languages.

© Aptech Ltd. Introduction to Java/Session 1 23

Java: Object-oriented Programming Language 2-5

Object-oriented

- Java is a pure OOP language that uses classes and objects that address to the real-world problem domains.
- Even, the application development in Java starts with a class designing.

Robust

- C and C++ languages** - Dynamic memory allocation/deallocation is done manually through pointers that resulted in memory related errors.
- Java incorporates:
 - Strong memory management** - Handles memory allocation and deallocation using **Garbage Collection Mechanism** that destroys unused objects in memory automatically.
 - Exception handling mechanism** - Stops abnormal termination of code at runtime.
 - Compile-time checking** - Ensures variables are declared which contain specific type of data.
 - Run-time checking** - Ensures purity of Java code during execution.

Java: Object-oriented Programming Language 3-5

Secure

- Security checks applied at different layers ensures that the Java programs are protected against malicious codes.
- Java programs that are accessed on the network are known as applets.
- Java applies security to applets by placing them in a sandbox that ensures it should not have direct access to files or resources available on the local system.
- Java Virtual Machine (JVM) which is a runtime environment for executing Java programs applies its own security features to Java language.
- JVM ensures that the code loaded for execution is well-formed and conforms to Java standards.

Java: Object-oriented Programming Language 4-5

Architecture Neutral and Portable

- Supports portability by converting application into architecture neutral bytecode during compilation.
- Java has defined language specifications, such as size of primitive data types and operators that are independent of the hardware platform.
- For example, an integer variable in Java always occupies 32 bits on whichever machine the code is executed.
- These features satisfies the major goal of Java language which is 'Write once, run anywhere'.

Multithreaded

- Allows a single program to perform multiple tasks simultaneously with different threads.

The slide has a header "Java: Object-oriented Programming Language 5-5" and a logo of a coffee cup with steam. It contains two main sections: "Distributed" (purple background) and "Dynamic" (blue background). The "Distributed" section lists: "Supports distributed programming in which resources can be accessed across the network." The "Dynamic" section lists: "At runtime an application can dynamically decide which classes it requires and loads them accordingly.", "This gives new perspective to Java for designing and developing applications." The footer includes "© Aptech Ltd.", "Introduction to Java/Session 1", and "27".

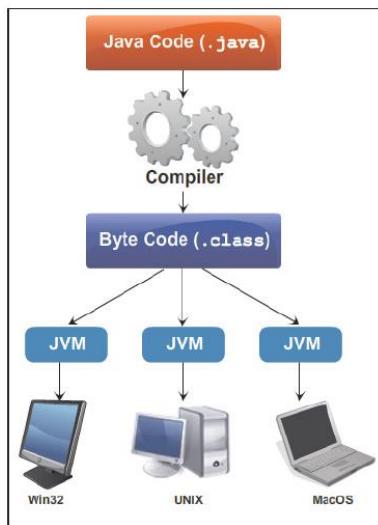
Using slides 23 to 27, explain the features of Java language.

Explain the features of Java programming language. Tell them these features have helped Java to become a popular programming language.

Mention that Java besides providing a platform-independent environment is also a high level programming language. Tell the students that Java is purely object-oriented language and has no standalone constants, variables, or functions. All of these are accessed through classes and objects. Explain to the students that Java is based on the object model. Object model is defined by means of classes and objects that are used to address the solutions closer to the real-world problem domains. In other words, it is a pure OOP language that uses classes and objects. Even, the application development in Java starts with a class designing.

Next, inform the students that Java is platform-independent which means the same Java code can be executed on any platform. Java achieves platform-independence by using bytecode.

You can explain the execution of same bytecode with different implementations of JVM on various platforms with the help of following figure (draw simplified version of the figure).



Then, tell the students that it is designed for writing highly reliable or robust software. It requires explicit method declaration and checks for syntax error at the time of compilation, and also at the time of interpretation. Java does not support pointers and pointer arithmetic.

Next, explain the security features of Java. Java provides a controlled environment for the execution of the program. It never assumes that the code is safe for execution. Java is secured and provides several layers of security control. In the first layer, the data and methods are accessed through interfaces that the class provides. In the second layer, the compiler ensures that the code is safe and follows the protocols set by Java before compiling the code. The third layer is safety and is provided by the interpreter. The fourth layer loads the class and ensures that the class does not violate the access restrictions, before loading it into the system.

Mention that Java is used for developing applications that are portable across multiple platforms, operating systems, and graphical user interfaces.

Next, explain to the students that since Java supports multi-threading, an application can perform multiple tasks. For example, in a Word application program, a user can type the content at the same time, the Word application performs spell-check for the typed content. Thus, Word application can perform multiple tasks at the same time.

Multithreading allows a single program having different threads to be executed independently at the same time. It is easy to work with threads in Java as it has built-in language support. Threads are also known as light-weight processes. Each thread runs independently without disturbing the execution of other threads.

Java is a dynamic language designed to adapt to an evolving environment. Tell them that Java programs are compiled to an architecture neutral bytecode format to transport the code efficiently to multiple hardware and software platforms. Bring to the notice of the students that Java technology takes portability, a stage further, by explicitly specifying the size of its basic data types to eliminate implementation dependence.

Lastly, explain to the students that Java is high-performance. Compared to those high-level, fully-interpreted scripting languages, Java is high-performance. The automatic garbage collector runs as a low-priority background thread, ensuring a high probability.

In-Class Question:

After you finish explaining, the features of Java, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



Which is the feature that allows a single program having different threads to be executed independently at the same time?

Answer:

Multithreading.

Slide 28

Let us understand Java platform.

The slide has a title 'Java Platform' at the top left and a coffee cup icon at the top right. The main content is a bulleted list:

- ◆ Is a software-only platform that runs on top of other hardware-based platforms.
- ◆ Contains **Java Runtime Environment (JRE)** with components namely:
 - ◆ Java Virtual Machine (JVM)
 - ◆ Java class library also referred to as Java Programming Interface (Java API)

At the bottom left is the copyright notice '© Aptech Ltd.', in the center is 'Introduction to Java/Session 1', and at the bottom right is the page number '28'.

Using slide 28, explain the Java platform.

Java is portable which means computer program written in Java must run on any hardware-based platforms or operating systems. The platform is not specific to any one processor, but it is an execution engine called virtual machine and compiler with a set of libraries. It is a software-only platform that runs on top of other hardware-based platforms. It contains Java Runtime Environment (JRE) with components namely:

- Java Virtual Machine (JVM)
- Java class library also referred to as Java Programming Interface (Java API)

Slides 29 and 30

Let us understand Java Virtual Machine (JVM).

Java Virtual Machine (JVM) 1-2

- ◆ It is an executable engine that creates an environment for executing Java compiled code, that is, bytecode.
- ◆ It is known as a virtual machine because it is an imitation of a Java processor on the physical machine.
- ◆ There are different implementations of JVM available for different platforms, such as Windows, Unix, and Solaris.

© Aptech Ltd. Introduction to Java / Session 1 29

Java Virtual Machine (JVM) 2-2

- ◆ **Bytecode:**
 - ◆ Is an intermediate form closer to machine representation.
 - ◆ Is an optimized set of instructions executed by the Java runtime environment.
 - ◆ This environment is known as JVM.
 - ◆ The same bytecode can be executed by different implementations of JVM on various platforms.

```

graph TD
    A[Java Code (.java)] --> B[Compiler]
    B --> C[Byte Code (.class)]
    C --> D[JVM]
    C --> E[JVM]
    C --> F[JVM]
    D --> G[Win32]
    E --> H[UNIX]
    F --> I[MacOS]
  
```

© Aptech Ltd. Introduction to Java / Session 1 30

Using slides 29 and 30, explain the working of JVM.

Explain to the students that JVM is the heart of Java programming language. Mention to them that the Java environment consists of five elements namely, Java language, bytecode definitions, Java/Sun libraries, JVM, and structure of **.class** files.

Tell the students that the portability feature of **.class** file helps the execution of the class files on any computer or chip set having an implementation of the JVM. Mention that the virtual machine is based on the idea of an imaginary computer which has a set of instructions. These instructions define the operations of the imaginary computer.

JVM is a processed virtual machine that can execute Java byte code. It is an executable engine that creates an environment for Java compiled code, that is, byte code. It is a code execution component for Java platform. It is known as a virtual machine because it is an imitation of a Java processor on the physical machine. JVM can be used to implement interpreters for dynamic languages. There are different implementations of JVM available for different platforms, such as Windows, Unix, and Solaris.

Bytecode:

It is an intermediate form closer to machine representation. It is an optimized set of instructions executed by the Java runtime environment. Each byte code is comprised of one or two bytes that represent instructions. This environment is known as JVM. The same byte code can be executed by different implementations of JVM on various platforms.

Mention that it forms a layer of abstraction for the underlying hardware platform, operating system, and compiled code. Also mention that different versions of JVMs are available for different operating systems.

In-Class Question:

After you finish explaining Java Virtual Machine (JVM), you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



Why Java Virtual Machine (JVM) is known as virtual machine?

Answer:

JVM is known as virtual machine because it is an imitation of a Java processor on the physical machine.

Tips:

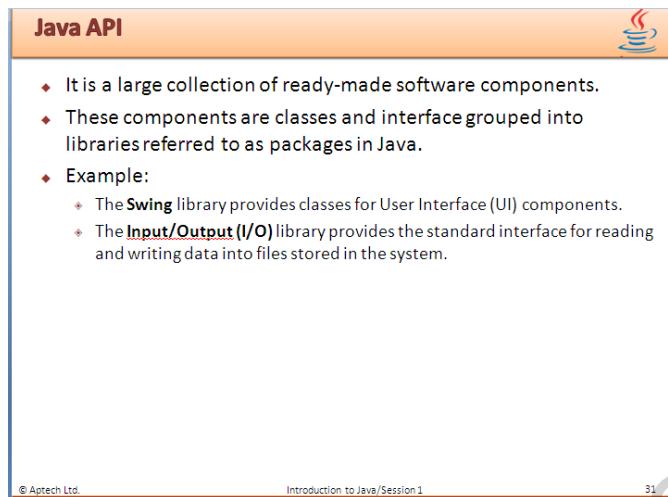
JVM is included with a Java interpreter and Just In Time (JIT) compiler. JIT comes with virtual machine and is used optionally. Sun Microsystems, suggest that, it is usually faster to select the JIT compiler option, especially if the method executable is used repeatedly.

JIT's main purpose is to convert the bytecode instruction set to machine code instructions targeted for a particular microprocessor. These instructions are stored and used, whenever a call is made to a particular method.

Bring to the notice of the students that once the code has been recompiled by the JIT compiler, it will usually run more quickly in the computer.

Slide 31

Let us understand Java API.



Java API

- ◆ It is a large collection of ready-made software components.
- ◆ These components are classes and interface grouped into libraries referred to as packages in Java.
- ◆ Example:
 - ◆ The **Swing** library provides classes for User Interface (UI) components.
 - ◆ The **Input/Output (I/O)** library provides the standard interface for reading and writing data into files stored in the system.

© Aptech Ltd. Introduction to Java/Session 1 31

Using slide 31, explain the Java APIs.

Java API is a set of classes included with the Java development environment. It is a large collection of ready-made software components. These components are classes and interface grouped into libraries referred to as packages in Java.

Some of the examples of Java API are as follows:

- The Swing library provides classes for User Interface (UI) components.
- The Input/Output (I/O) library provides the standard interface for reading and writing data into files stored in the system.
- The network library to read and write the data on the sockets.
- The collection library to store the objects in the collection, such as linked list, set, or as a map.

Tips:

There are three types of Java API:

- **Official core Java API:** It contains JDK and JRE.
- **Optional official API:** They can be downloaded separately.
- **Unofficial API:** They are developed by third parties. For example, NetBeans platform.

In-Class Question:

After you finish explaining Java APIs, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



What are the components of Java API?

Answer:

An Application Programming Interface (API), in the context of Java, is a collection of prewritten packages, classes, and interfaces with their respective methods, fields, and constructors. Similar to a user interface, which facilitates interaction between humans and computers, an API serves as a software program interface facilitating interaction.

Slide 32

Let us understand editions of Java.

Editions of Java



- ◆ **Java Standard Edition (Java SE)** - Is a base platform and enables to develop console and networking applications for desktop computers.
- ◆ **Java Enterprise Edition (Java EE)** - Is built on top of Java SE platform and provides a standard specification for developing and deploying distributed, scalable, and multi-tier enterprise applications.
- ◆ **Java Mobile Edition (Java ME)** - Is a robust platform for developing embedded Java applications for consumer electronic devices, such as mobiles, Personal Digital Assistants (PDAs), and TV set-top boxes.

© Aptech Ltd. Introduction to Java/Session 1 32

Using slide 32, explain the editions of Java.

Java Standard Edition (Java SE) – It is a widely used platform for development and deployment of portable applications for desktop and server environment. It is a base platform and enables to develop console and networking applications for desktop computers. It is a platform specification.

Java Enterprise Edition (Java EE) – It provides an API and run time environment for developing and running enterprise software including network and Web services. It is built on top of Java SE platform and provides a standard specification for developing and deploying distributed, scalable, and multi-tier enterprise applications.

Java Mobile Edition (Java ME) - Is a robust platform for developing embedded Java applications for consumer electronic devices, such as mobiles, Personal Digital Assistants (PDAs), and TV set-top boxes.

Slides 33 and 34

Let us understand components of Java SE platform.

Components of Java SE Platform 1-2



- ◆ Provides two software components:
 - ◆ **JRE**
 - JRE provides JVM and Java libraries that are used to run a Java program.
 - ◆ **JDK**
 - Known as Java Development Kit (JDK).
 - Is a binary software development kit released by Oracle Corporation.
 - Is an implementation of Java and distributed for different platforms, such as Windows, Linux, Mac OS X, and so on.
 - Contains a comprehensive set of tools, such as compilers and debuggers that are used to develop Java applications.

© Aptech Ltd. Introduction to Java/Session 1 33

Components of Java SE Platform 2-2



- ◆ **Development Tools** – Include tools used for compiling, running, debugging, and documenting a Java application.
- ◆ **API** - Provides the core functionality of the Java programming language.
- ◆ **Deployment Tools** – Provides software for deploying the developed applications to end-users.
- ◆ **User Interface Toolkits** - Enables the developer to create graphical interfaces in a Java application.
- ◆ **Integration libraries** - Enable developers to access and manipulate database and remote objects in an application.

© Aptech Ltd. Introduction to Java/Session 1 34

Using slides 33 and 34, explain the components of Java SE platform.

There are two software components of Java SE platform:

JRE

JRE provides JVM and Java libraries that are used to run a Java program. JRE is part of the Java Development Kit (JDK), but can be downloaded separately. JRE was originally developed by Sun Microsystems Inc., a wholly-owned subsidiary of Oracle Corporation. It is also known as Java runtime.

JDK

It is known as Java Development Kit (JDK). It includes JRE and command line development tools such as compilers and debuggers that are necessary for developing applications. It is a binary software development kit released by Oracle Corporation. It is an implementation of Java and distributed for different platforms, such as Windows, Linux, Mac OS X, and so on.

It contains a comprehensive set of tools, such as compilers and debuggers that are used to develop Java applications.

Description of some of these tools is as follows:

- **Development Tools** – Include tools used for compiling, running, debugging, and documenting a Java application.
- **API** - Provides the core functionality of the Java programming language.
- **Deployment Tools** – Provides software for deploying the developed applications to end-users.
- **User Interface Toolkits** - Enables the developer to create graphical interfaces in a Java application.
- **Integration libraries** - Enable developers to access and manipulate database and remote objects in an application.

Slide 35

Let us understand evolution of Java.

Evolution of Java	
Releases	Implementation
JDK 1.0	Creation of packages with classes in the standard library
JDK 1.1	Included an event delegation model for Graphical User Interface (GUI) package AWT, JavaBeans, and Java Database Connectivity (JDBC) API
JDK 1.2 (Java 2)	Included a new graphical API, named Swing. Also, added APIs for reflection and collection framework (based on data structure)
JDK 1.3	Included a directory interface to lookup for components, named, Java Naming and Directory Interface (JNDI)
JDK 1.4	Included regular expression API, assertions, exception chaining, channel-based I/O API, an XML API for parsing and processing
JDK 1.5	Included new features in the language such as for-each loop, generics, annotations, and auto-boxing
JDK 1.6	Included script language, visual basic language support, and improvements in the GUI

© Aptech Ltd.

Introduction to Java/Session 1

35

Using slide 35, explain the evolution of Java.

The Java language has undergone several changes since JDK 1.0 and various additions of classes and packages to the standard libraries.

Explain the features and improvements of different JDK versions as listed in the slide.

Slide 36

Let us understand Java SE 7.

Java SE 7	
<ul style="list-style-type: none"> ◆ Java SE 7 is the new major release of Java with internal version number as 1.7. ◆ The lists of new features that are incorporated in the language are as follows: <ul style="list-style-type: none"> ◆ Supports the use of <code>String</code> class in the <code>switch</code> decision-making construct. ◆ Integer types can be assigned with a binary number value. ◆ Supports the use of underscore character (<code>_</code>) between the digits of a numeric value. ◆ An expandable <code>try</code> statement called <code>try-with-resources</code> statement used for automatic resource management. ◆ Constructor of a generic class declaration can be replaced with an empty set of parameters (<code><></code>). ◆ Enhancement in exception handling mechanism. In exception handling code, a single catch can be used to handle one or more exceptions. ◆ Compiler warnings generated for <code>varargs</code> methods has been improved. 	

© Aptech Ltd.

Introduction to Java/Session 1

36

Using slide 36, explain the features of Java SE 7.

Java SE 7 is the new major release of Java with internal version number as 1.7. It includes JVM support for dynamic languages.

The lists of new features that are incorporated in the language are as follows:

- Supports the use of `String` class in the `switch` decision-making construct.
- Integer types can be assigned with a binary number value.

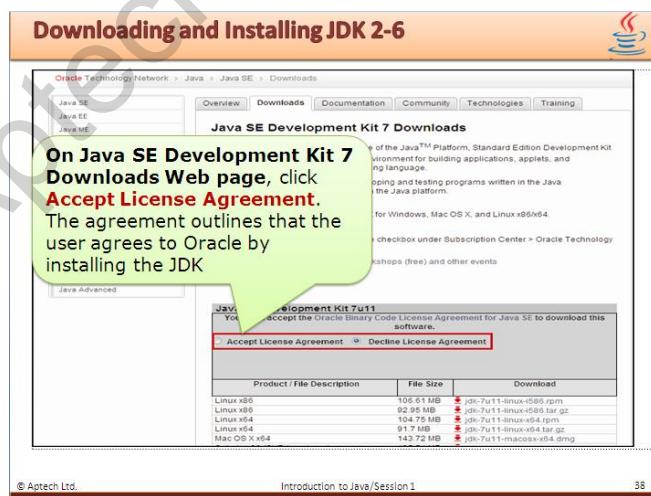
- Supports the use of underscore character (_) between the digits of a numeric value.
- An expandable try statement called try-with-resources statement used for automatic resource management.
- Constructor of a generic class declaration can be replaced with an empty set of parameters (<>).
- Enhancement in exception handling mechanism. In exception handling code, a single catch can be used to handle one or more exceptions.
- Compiler warnings generated for varargs methods has been improved.

Tips:

Check the following link for examples on each of the features added in Java SE 7:
<http://radar.oreilly.com/2011/09/java7-features.html>.

Slides 37 to 42

Let us understand downloading and installing of JDK on Windows system.



Downloading and Installing JDK 3-6

◆ Click to add text

To download JDK 7 for Windows 32-bit OS, click **jdk-7u11-windows-i586.exe** under Download

The screenshot shows the Java SE Development Kit 7u11 download page. It includes a license acceptance dialog, a product table with columns for Product / File Description, File Size, and Download, and another license acceptance dialog at the bottom.

Product / File Description	File Size	Download
Linux x86	108.51 MB	jdk-7u11-linux-i586.rpm
	92.95 MB	jdk-7u11-linux-i586.tar.gz
	104.75 MB	jdk-7u11-linux-x64.rpm
	91.7 MB	jdk-7u11-linux-x64.tar.gz
	143.72 MB	jdk-7u11-macosx-x64.dmg
(package)	136.54 MB	jdk-7u11-solaris-i586.tar.Z
	91.92 MB	jdk-7u11-solaris-i586.tar.gz
(package)	22.52 MB	jdk-7u11-solaris-x64.tar.Z
	14.95 MB	jdk-7u11-solaris-x64.tar.gz
Solaris SPARC	136.57 MB	jdk-7u11-solaris-sparc.tar.Z
Solaris SPARC 64-bit (S/R4 package)	95.25 MB	jdk-7u11-solaris-sparc64.tar.Z
Solaris SPARC 64-bit	17.77 MB	jdk-7u11-solaris-sparc64tar.gz
Windows x86	88.77 MB	jdk-7u11-windows-i586.exe
Windows x64	90.41 MB	jdk-7u11-windows-x64.exe

Java SE Development Kit 7u10
You must accept the Oracle Binary Code License Agreement for Java SE to download this software.

© Aptech Ltd. Introduction to Java / Session 1 39

Downloading and Installing JDK 4-6

In the Save As dialog box, select the location to save the JDK installer and then, click **Save**

The screenshot shows a 'Save As' dialog box from a Windows file explorer. It displays the file name as 'jdk-7u11-windows-i586.exe' and the save type as 'Application'. A callout bubble points to the 'Save' button.

© Aptech Ltd. Introduction to Java / Session 1 40

Downloading and Installing JDK 5-6

- ◆ Double-click the installer icon and follow the instructions provided by the JDK installer.
- ◆ The installer installs development tools, source code, and the JRE in the default directory, C:\Program Files\Java.
- ◆ Following figure shows the directory structure of the installed JDK on the system:

The diagram illustrates the directory structure of the installed JDK 1.7. It shows a main folder 'JDK 1.7' which contains subfolders 'bin', 'lib', 'db', 'include', and 'jre'.

© Aptech Ltd. Introduction to Java / Session 1 41

Downloading and Installing JDK 6-6



- Following table lists the common directories that are part of the typical JDK installation:

Directory	Description
bin	Contains tools that are used for developing a Java application, such as compiler and JVM
db	Contains a relational database named Apache Derby
include	Contains header files that are used to interact with C applications
jre	Represents the JRE used by the JDK

© Aptech Ltd. Introduction to Java/Session 1 42

Using slides 37 to 42, explain the steps to download and install JDK.

Explain the steps presented on slides 37 to 41 on downloading and installing JDK on the Windows system.

Then, explain the created directory structure of JDK within the installed directory listed on slide 42.

Tips:

The following figure shows the JDK 1.7 files and directories:



The explanation is as follows:

- /jdk1.7.0 – Root directory of the JDK software installation. Contains copyright, license, and README files. Also contains src.zip, the archive of source code for the Java platform.

- **/jdk1.7.0/bin** – Executables for all the development tools contained in the JDK. The **PATH** environment variable should contain an entry for this directory. For more information on the tools, see [JDK Tools](#).
- **/jdk1.7.0/lib** – Files used by the development tools. Includes `tools.jar`, which contains non-core classes for support of the tools and utilities in the JDK. Also includes `dt.jar`, the DesignTime archive of BeanInfo files that tell interactive development environments (IDE's) how to display the Java components and how to let the developer customize them for an application.
- **/jdk1.7.0/jre** – Root directory of the Java runtime environment used by the JDK development tools. The runtime environment is an implementation of the Java platform. This is the directory referred to by the `java.home` system property.
- **/jdk1.7.0/jre/bin** – Executable files for tools and libraries used by the Java platform. The executable files are identical to files in `/jdk1.7.0/bin`. The **java** launcher tool serves as an application launcher (and replaced the old **jre** tool that shipped with 1.1 versions of the JDK). This directory does not need to be in the **PATH** environment variable.
- **/jdk1.7.0/jre/lib** – Code libraries, property settings, and resource files used by the Java runtime environment. For example:
- `rt.jar` -- the *bootstrap* classes (the RunTime classes that comprise the Java platform's core API).
- `charsets.jar` -- character-conversion classes.
 - Aside from the `ext` subdirectory (described here) there are several additional resource subdirectories not described here.
- **/jdk1.7.0/jre/lib/ext** – Default installation directory for [Extensions](#) to the Java platform. This is where the `JavaHelp.jar` file goes when it is installed, for example.
- `localedata.jar` -- locale data for `java.text` and `java.util`.
- **/jdk1.7.0/jre/lib/security** – Contains files used for security management. These include the security policy (`java.policy`) and -security properties (`java.security`) files.
- **/jdk1.7.0/jre/lib/sparc** – Contains the `.so` (shared object) files used by the Solaris version of the Java platform.
- **/jdk1.7.0/jre/lib/sparc/client** – Contains the `.so` file used by the Java HotSpot™ Client Virtual Machine, which is implemented with Java HotSpot™ technology. This is the default VM.
- **/jdk1.7.0/jre/lib/sparc/server** – Contains the `.so` file used by the Java HotSpot™ Server Virtual Machine.
- **/jdk1.7.0/jre/lib/applet** – Jar files containing support classes for applets can be placed in the `lib/applet/` directory. This reduces startup time for large applets by allowing applet classes to be pre-loaded from the local file system by the applet class loader, providing the same protections as if they had been downloaded over the net.
- **/jdk1.7.0/jre/lib/fonts** – Font files for use by platform.

Slides 43 to 45

Let us understand configuring JDK.

Configuring JDK 1-3

- ◆ To work with JDK and Java programs, certain settings need to be made to the environment variables.
- ◆ Environment variables are pointers pointing to programs or other resources. The variables to be configured are as follows:
 - ❖ **PATH** - Set to point to the location of Java executables (javac.exe and java.exe).
 - ❖ **CLASSPATH** - Specifies the location of the class files and libraries needed by the Java compiler to compile applications.

© Aptech Ltd. Introduction to Java/Session 1 43

Configuring JDK 2-3

- ◆ To set the value for the **PATH** variable, perform the following steps in Windows 7:
 - ❖ Right-click **My Computer** icon on the desktop and then, click **Properties** from the context menu.
 - ❖ Click **Advanced system** settings link on the left tab.
 - ❖ Under **Advanced** tab, click **Environment Variables**.
 - ❖ In the System variables area, select the **PATH** variable and then, click **Edit** button to enter the JDK installation folder path.
 - ❖ Type path of the bin folder in the **Variable Value** text box.
 - ❖ For example, the path can be:
`C:\WINDOWS\system32;C:\WINDOWS;C:\Program Files\Java\jdk1.7.0\bin`

© Aptech Ltd. Introduction to Java/Session 1 44

Configuring JDK 3-3

- ◆ To set **CLASSPATH** variable in Windows 7, perform the following steps:
 - ❖ Right-click **My Computer** icon on the desktop and click **Properties** from the context menu.
 - ❖ Click **Advanced system** settings link on the left tab.
 - ❖ Under **Advanced** tab, click **Environment Variables**.
 - ❖ In the System variables area, click **New** button.
 - ❖ Type **CLASSPATH** in **Variable Name** and then, type `C:\<jdk安装目录>` in the **Variable Value**.

© Aptech Ltd. Introduction to Java/Session 1 45

Using slides 43 to 45, explain the steps to use **PATH** and **CLASSPATH** environment variables on Windows platform.

To work with JDK and Java programs, certain settings need to be made to the environment variables. Environment variables are pointers pointing to programs or other resources.

The variables to be configured are as follows:

- **PATH** - Set to point to the location of Java executables (`javac.exe` and `java.exe`). The PATH environment variables are a series of directories separated by semicolons (`;`).
- **CLASSPATH** - Specifies the location of the class files and libraries needed by the Java compiler to compile applications. On Windows platform, you can check if the `CLASSPATH` is set on the command line using `echo %CLASSPATH%`.

Then, explain the steps to set the `PATH` and `CLASSPATH` variables on Windows 7 as explained on slides 44 and 45.

Tips:

The `CLASSPATH` can also be specified using the `-cp` command line switch which allows the `CLASSPATH` to be set individually for each application without affecting other applications. The default value of the class path is `". "`, meaning that only the current directory is searched. Specifying either the `CLASSPATH` variable or the `-cp` command line switch overrides this value.

To get more information on the `CLASSPATH`, read the technical documentation on this link:
<http://docs.oracle.com/javase/8/docs/technotes/tools/windows/classpath.html>.

In-Class Question:

After you finish explaining configuring JDK, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



What is `CLASSPATH`?

Answer:

It specifies the location of the class files and libraries needed by the Java compiler to compile applications.

Slides 46 and 47

Let us summarize the session.

Summary 1-2



- ◆ The development of application software is performed using a programming language that enforces a particular style of programming, also referred to as programming paradigm.
- ◆ In structured programming paradigm, the application development is decomposed into a hierarchy of subprograms.
- ◆ In object-oriented programming paradigm, applications are designed around data, rather than focusing only on the functionalities.
- ◆ The main building blocks of an OOP language are classes and objects. An object represents a real-world entity and a class is a conceptual model.

© Aptech Ltd. Introduction to Java/Session 1 46

Summary 2-2



- ◆ Java is an OOP language as well a platform used for developing applications that can be executed on different platforms. Java platform is a software-only platform that runs on top of the other hardware-based platforms.
- ◆ The editions of Java platform are Java SE, Java EE, and Java ME.
- ◆ The components of Java SE platform are JDK and JRE. JRE provides JVM and Java libraries that are used to run a Java program. JDK includes the necessary development tools, runtime environment, and APIs for creating Java programs.

© Aptech Ltd. Introduction to Java/Session 1 47

Using slides 46 and 47, summarize the session. End the session with a brief summary of what has been taught in the class.

1.3 Post Class Activities for Faculty

You should familiarize yourself with the topics of the next session which is based on developing Java applications in NetBeans IDE.

Tips:

You can also check the Articles/Blogs/Expert Videos uploaded on the Online Varsity site to gain additional information related to the topics covered in the next session. You can also connect to online tutors on the Online Varsity site to ask queries related to the sessions.

Session 2 – Application Development in Java

2.1 Pre-Class Activities

Before you commence the session, you should familiarize yourself with the topics of this session in-depth.

Here, you can discuss the key points with the students that were covered in the previous session. Prepare a question or two which will help you to relate the current session objectives.

2.1.1 Objectives

By the end of this session, the learners will be able to:

- Explain the structure of a Java class
- List and explain steps to write a Java program
- Identify the benefits of NetBeans IDE
- Describe the various elements of NetBeans IDE
- Explain the steps to develop, compile, and execute Java program using NetBeans IDE
- Explain the various components of JVM
- Describe comments in Java

2.1.2 Teaching Skills

To teach this session, you should be well-versed with basic structure of designing a class in Java. You have to familiarize yourself with the NetBeans Integrated Development Environment (IDE) and its components used for developing the Java applications in the course.

You should develop and execute a simple Java program in NetBeans IDE to be covered in the session. Also, comment the code, which will help you to explain the use of comments for Java documentation.

You should teach the concepts in the theory class using the images provided. For teaching in the class, you are expected to use slides and LCD projectors.

Tips:

It is recommended that you test the understanding of the students by asking questions in between the class.

In-Class Activities:

Follow the order given here during In-Class activities.

Overview of the Session:

Give the students the overview of the current session in the form of session objectives. Show the students slide 2 of the presentation.

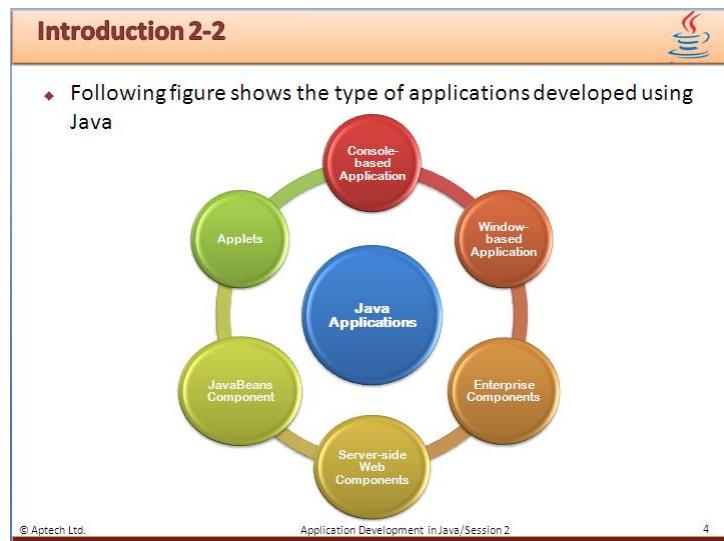
Objectives
<ul style="list-style-type: none">◆ Explain the structure of a Java class◆ List and explain steps to write a Java program◆ Identify the benefits of <u>NetBeans IDE</u>◆ Describe the various elements of <u>NetBeans IDE</u>◆ Explain the steps to develop, compile, and execute Java program using <u>NetBeans IDE</u>◆ Explain the various components of JVM◆ Describe comments in Java

Tell the students that this session introduces the basic structure of Java class and NetBeans integrated development environment (IDE) and its components. Explain the execution of Java program using NetBeans IDE.

2.2 In-Class Explanations**Slides 3 and 4**

Let us understand about the environment used for Java development.

Introduction 1-2
<ul style="list-style-type: none">◆ Java is a popular OOP language that supports developing applications for different requirements and domain areas.◆ All types of applications can be developed:<ul style="list-style-type: none">◆ In a simple text editor, such as <u>Notepad</u>.◆ In an environment that provides necessary tools to develop a Java application.◆ The environment is called as Integrated Development Environment (IDE).



Using slides 3 and 4, explain the environment used for Java development.

Java helps the programmers to develop wide range of applications that can run on various hardware and Operating System (OS). Java is a popular OOP language that supports developing applications for different requirements and domain areas.

Different types of Java applications can be developed using various editors of Java available in the market. Some of the editors include: a simple text editor, such as Notepad, a full equipped environment with Java development tools such as NetBeans, Eclipse, JCreator, Oracle JDeveloper, and so on. These editors provide necessary tools, such as compiler, debugger, Java runtime integrated in them to develop Java applications. Hence, they are called as Integrated Development Environment (IDE).

Then, explain the type of applications that can be developed using Java as shown on slide 4.

Slides 5 to 9

Let us understand structure of a Java class.

Structure of a Java Class 1-5

- ◆ The Java programming language is designed around object-oriented features and begins with a class design.
- ◆ The class represents a template for the objects created or instantiated by the Java runtime environment.
- ◆ The definition of the class is written in a file and is saved with a .java extension.
- ◆ Following figure shows the basic structure of a Java class.

```
package <package_name>
import <other_packages>;
public class ClassName {
    <variables(also known as fields)>;
    <constructor method(s)>;
    <other methods>;
}
```

© Aptech Ltd. Application Development In Java/Session 2 5

Structure of a Java Class 2-5

package

```
package <package_name>;
```

- Defines a namespace that stores classes with similar functionalities in them.
- The package keyword identifies:
 - Name of the package to which the class belongs.
 - Visibility of the class within the package and outside the package.
- The concept of package is similar to folder in the OS.
- In Java, all classes belongs to a package. If the package statement is not specified, then the class belongs to the default package.
- Example: All the user interface classes are grouped in the `java.awt` or `java.swing` packages.

© Aptech Ltd. Application Development In Java/Session 2 6

Structure of a Java Class 3-5

import

```
import <other_packages>;
```

- Identifies the classes and packages that are used in a Java class.
- Helps to narrow down the search performed by the Java compiler by informing it about the classes and packages.
- Mandatory to import the required classes, before they are used in the Java program.
- Some exceptions wherein the use of **import** statement is not required are as follows:
 - If classes are present in the `java.lang` package.
 - If classes are located in the same package.
 - If classes are declared and used along with their package name.
For example, `java.text.NumberFormat nf = new java.text.NumberFormat();`

© Aptech Ltd. Application Development In Java/Session 2 7

Structure of a Java Class 4-5

```

class ClassName {
    public class ClassName {
        • class keyword identifies a Java class.
        • Precedes the name of the class in the declaration.
        • public keyword indicates the access modifier that decides the visibility of the class.
        • Name of a class and file name should match.

    Variables
    <variables(also known as fields)>;
    • Are also referred to as instance fields or instance variables.
    • Represent the state of objects.
}

```

© Aptech Ltd. Application Development in Java/Session 2 8

Structure of a Java Class 5-5

```

Methods
<other methods>;
• Are functions that represent some action to be performed on an object.
• Are also referred to as instance methods.

Constructors
<constructor method(s)>;
• Are also methods or functions that are invoked during the creation of an object.
• Used to initialize the objects.

```

© Aptech Ltd. Application Development in Java/Session 2 9

Using slides 5 to 9, explain the structure of Java class.

The Java program begins with a class design. Everything has to be included inside the class and this makes code easier to reuse. Java class is like a container which holds codes. The class represents a template for the objects created or instantiated by the Java runtime environment. The definition of the class is written in a file and is saved with a .Java extension. In order to compile Java application successfully, at least one class and one method are required in the project.

Tips:

A class with a main () method is a tester class in which objects can be instantiated. The functional testing of the methods can be done by invoking them on the object.

Different parts of the Java class are:

package

Package contains classes and methods. It defines a namespace that stores classes with similar functionalities in them. The package keyword identifies the name of the package to which the class belongs and the visibility of the class within the package and outside the package. The concept of package is similar to folder in the OS. In Java, all classes belong to a package. If the package statement is not specified, then the class belongs to the default package.

Example: All the user interface classes are grouped in the Java.awt or Java.swing packages.

import

It allows specifying the classes from other packages that can be referenced without qualifying them with their package. It identifies the classes and packages that are used in a Java class. It helps to narrow down the search performed by the Java compiler by informing it about the classes and packages.

It is mandatory to import the required classes, before they are used in the Java program. Some exceptions wherein the use of **import** statement is not required are as follows:

- If classes are present in the `Java.lang` package.
- If classes are located in the same package. For example, if a package named `accountpack` contains class `Account` and class `Loan`. Then, importing the classes is not required.
- If classes are declared and used along with their package name. For example,
`Java.text.NumberFormat nf = new Java.text. NumberFormat();`

class

`Class` keyword identifies a Java class. It precedes the name of the class in the declaration. `Public` keyword indicates the access modifier that decides the visibility of the class. Name of a class and file name should match.

Variables

They are also referred to as instance fields or instance variables. They represent the state of objects.

Methods

Java programs mostly made up of methods. A main method is where main code of the program is executed. Methods are functions that represent some action to be performed on an object. They are also referred to as instance methods.

Constructors

Constructors are also methods or functions that are invoked during the creation of an object. It is used to initialize the objects. It is a special method in a class which is used to construct an instance of the class.

In-Class Question:

After you finish explaining structure of Java class, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



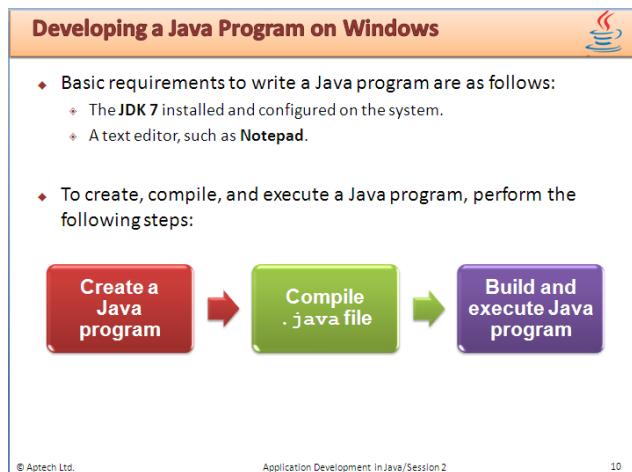
What does the package contain?

Answer:

Package contains classes and interfaces.

Slide 10

Let us understand developing a Java program on Windows.



Using slide 10, explain the requirements for developing Java programs on Windows platform.

Java is write once, run anywhere language which means that the same Java class can be executed on any platform containing the JVM.

However, to develop a Java program, the basic requirements are as follows:

- The JDK 7 installed and configured on the system.
- A text editor, such as Notepad.

Then, explain the steps required to create, compile, and execute a Java program.

Slides 11 to 14

Let us understand to create a Java program in the notepad.

Create a Java Program 1-4

- ◆ Following code snippet demonstrates a simple Java program:

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Welcome to the world of Java");
    }
}
```

- ◆ class is a keyword and **HelloWorld** is the name of the class.
- ◆ The entire class definition and its members must be written within the opening and closing curly braces {}.
- ◆ The area between the braces is known as the class body and contains the code for that class.

© Aptech Ltd. Application Development in Java/Session 2 11

Create a Java Program 2-4

- ◆ Following code snippet demonstrates a simple Java program:

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Welcome to the world of Java");
    }
}
```

- ◆ main () - method is the entry point for a java-based console application.
- ◆ public - Is a keyword that enables the JVM to access the main () method.
- ◆ static - Is a keyword that allows a method to be called from outside a class without creating an instance of the class.
- ◆ void - Is a keyword that represents the data type of the value returned by the main () method. It informs the compiler that the method will not return any value.
- ◆ args - Is an array of type String and stores command line arguments. String is a class in Java and stores group of characters.

© Aptech Ltd. Application Development in Java/Session 2 12

Create a Java Program 3-4

- ◆ Following code snippet demonstrates a simple Java program:

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Welcome to the world of Java");
    }
}
```

- ◆ **System.out.println()** statement displays the string that is passed as an argument.
- ◆ **System** is the predefined class and provides access to the system resources, such as console.
- ◆ **out** is the output stream connected to the console.
- ◆ **println()** is the built-in method of the output stream that is used to display a string.

© Aptech Ltd. Application Development in Java/Session 2 13

Create a Java Program 4-4

Following code snippet demonstrates a simple Java program:

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Welcome to the world of Java");
    }
}
```

- ◆ Save the file as **HelloWorld.java**.
- ◆ The file name is same as class name, as the compilation of a Java code results in a class. Hence, the class name and the file name should be same.

© Aptech Ltd. Application Development in Java/Session 2 14

Using slides 11 to 14, explain how to create a Java program in the notepad.

To program in Java, user needs a compiler which is a program to convert Java source code to bytecode. Also a Java virtual machine is required which understands Java bytecode and translate them into machine language. The basic unit of Java program is class. Class is a keyword and **HelloWorld** is the name of the class. The entire class definition and its members must be written within the opening and closing curly braces {}.

The area between the braces is known as the class body and contains the code for that class.

- **main ()** - method is the entry point for a Java-based console application. It invokes when Java program starts from the command line.
- **public** - Is a keyword that enables the JVM to access the main() method.
- **static** - Is a keyword that allows a method to be called from outside a class without creating an instance of the class.
- **void** - Is a keyword that represents the data type of the value returned by the main() method. It informs the compiler that the method will not return any value. It is not a type in Java. It represents the absence of a type.
- **args** - Is an array of type String and stores command line arguments. String is a class in Java and stores group of characters.

`System.out.println ()` statement displays the string that is passed as an argument. `System` is the predefined class and provides access to the system resources, such as console. `out` is the output stream connected to the console. The `println ()` is the built-in method of the output stream that is used to display a string. Save the file as `HelloWorld.java`. The file name is same as class name, as the compilation of a Java code results in a class. Hence, the class name and the file name should be same.

In-Class Question:

After you finish explaining the Java program creation, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



What is the use of void keyword?

Answer:

It is a keyword that represents the data type of the value returned by the main () method.

Slides 15 to 18

Let us understand compile .java file.

Compile .java File 1-4

- Following figure shows the compilation process of the Java program.

The HelloWorld.java file is known as source code file.
 It is compiled by invoking tool named javac.exe, which compiles the source code into a .class file.
 The .class file contains the bytecode which is interpreted by java.exe tool.
 java.exe interprets the bytecode and runs the program.

© Aptech Ltd. Application Development in Java/Session 2 15

Compile .java File 2-4

- The syntax to use the javac.exe command:

Syntax

```
javac [option] source
```

where,
 source - Is one or more file names that end with a .java extension.

© Aptech Ltd. Application Development in Java/Session 2 16

Compile .java File 3-4

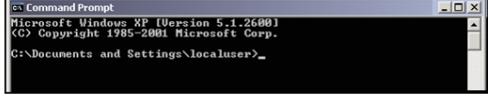
- Following table lists some of the options that can be used with the javac command:

Option	Description
-classpath	Specifies the location for the imported classes (overrides the CLASSPATH environment variable)
-d	Specifies the destination directory for the generated class files
-g	Prints all debugging information instead of the default line number and file name
-verbose	Generates message while the class is being compiled
-version	Displays version information
Sourcepath	Specifies the location of the input source file
-help	Prints a synopsis of standard options

- For example, `javac -d c:\ HelloWorld.java` will create and save `HelloWorld.class` file in the C:\ drive.

© Aptech Ltd. Application Development in Java/Session 2 17

Compile .java File 4-4



- ◆ To compile the `HelloWorld.java` program from the Windows platform, the user can:
 - ❖ Click **Start** menu.
 - ❖ Choose **Run**.
 - ❖ Enter the `cmd` command to display the **Command Prompt** window.
 - ❖ Following figure shows the **Command Prompt** window:

Set the drive and directory path to the directory containing `.java` file.
For example, `cd H:\Java`.

Type the command, `javac HelloWorld.java` and press **Enter**.

© Aptech Ltd. Application Development in Java/Session 2 18

Using slides 15 to 18, explain the compilation process in Java.

Before the JVM can run a Java program, the program source code can be compiled into bytecode. Java bytecode is a platform independent version of machine code. Once the Java source code compiled successfully, user can invoke Java Virtual Machine to run the application byte-code. The `HelloWorld.java` file is known as source code file. It is compiled by invoking tool named `Javac.exe`, which compiles the source code into a `.class` file. The `.class` file contains the bytecode which is interpreted by `java.exe` tool. `java.exe` interprets the bytecode and runs the program.

Explains the lists of the options that can be used with the `.Javac` command shown on slide 17.

Then, explain the steps to compile the `HelloWorld.java` program from the Windows platform on slide 18.

Tips:

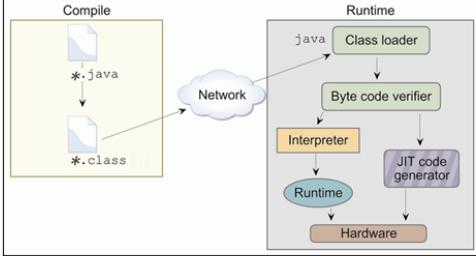
Check the following site for more information on the `Javac` command:

<http://www.cis.upenn.edu/~bcpierce/courses/629/jdkdocs/tooldocs/win32/Javac.html>.

Slides 19 to 22

Let us understand to build and execute Java program.

Build and Execute Java Program 1-4



The diagram illustrates the flow from compilation to execution:

- Compile:** A Java source file (*.java) is compiled into a class file (*.class).
- Network:** The class file is sent over a network to a runtime environment.
- Runtime:** The runtime environment includes a **Class loader**, **Byte code verifier**, **Interpreter**, **Runtime**, and **Hardware**.
- Execution Path:** The class loader loads the class, the byte code verifier checks it, and the interpreter executes it. If the interpreter finds performance issues, it triggers the JIT code generator to compile the code into native or platform-specific code before handing it back to the interpreter.

© Aptech Ltd. Application Development in Java/Session 2 19

Build and Execute Java Program 2-4

The **class loader** component of JVM loads all the necessary classes from the runtime libraries required for execution of the compiled bytecode.

The **bytecode verifier** then checks the code to ensure that it adheres to the JVM specification.

The bytecode is executed by the **interpreter**.

To boost the speed of execution, in Java version 2.0, a Hot Spot **Just-in-Time (JIT)** compiler was included at runtime.

During execution, the **JIT** compiler compiles some of the code into native code or platform-specific code to boosts the performance.

© Aptech Ltd. Application Development in Java/Session 2 20

Build and Execute Java Program 3-4

- The Java interpreter command, `java` is used to interpret and run the Java **bytecode**.
- The syntax to use the `java.exe` command is as follows:

Syntax

```
java [option] classname [arguments]
```

where,

classname: Is the name of the class file.

arguments: Is the arguments passed to the main function.

- To execute the **HelloWorld** class, type the command, `java HelloWorld` and press **Enter**.

© Aptech Ltd. Application Development in Java/Session 2 21

Build and Execute Java Program 4-4

Following table lists some of the options that can be used with the `java` command:

Option	Description
<code>classpath</code>	Specifies the location for the imported classes (overrides the CLASSPATH environment variable)
<code>-v</code> or <code>-verbose</code>	Produces additional output about each class loaded and each source file compiled
<code>-version</code>	Displays version information and exits
<code>-jar</code>	Uses a JAR file name instead of a class name
<code>-help</code>	Displays information about help and exits
<code>-X</code>	Displays information about non-standard options and exits

© Aptech Ltd. Application Development in Java/Session 2 22

Using slides 19 to 22, explain the process to build and execute the Java program.

Java programs are run or interpret by another program called the Java VM. The JVM is at the heart of the Java programming language. It is responsible for executing the .class file or bytecode file. The .class file can be executed on any computer or device that has the JVM implemented on it. The class loader component of JVM loads all the necessary classes from the runtime libraries required for execution of the compiled bytecode. The bytecode verifier then checks the code to ensure that it adheres to the JVM specification. The bytecode is executed by the interpreter.

To boost the speed of execution, in Java version 2.0, a Hot Spot Just-in-Time (JIT) compiler was included at runtime. During execution, the JIT compiler compiles some of the code into native code or platform-specific code to boosts the performance. The Java interpreter command, `java` is used to interpret and run the Java bytecode. Once the Java source code compiled successfully, user can invoke Java Virtual Machine to run the application byte-code.

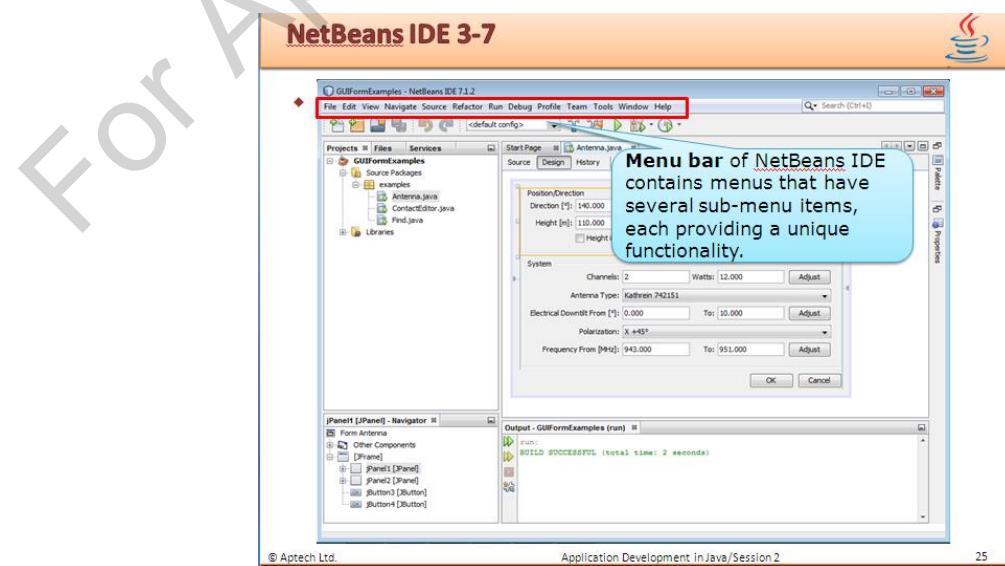
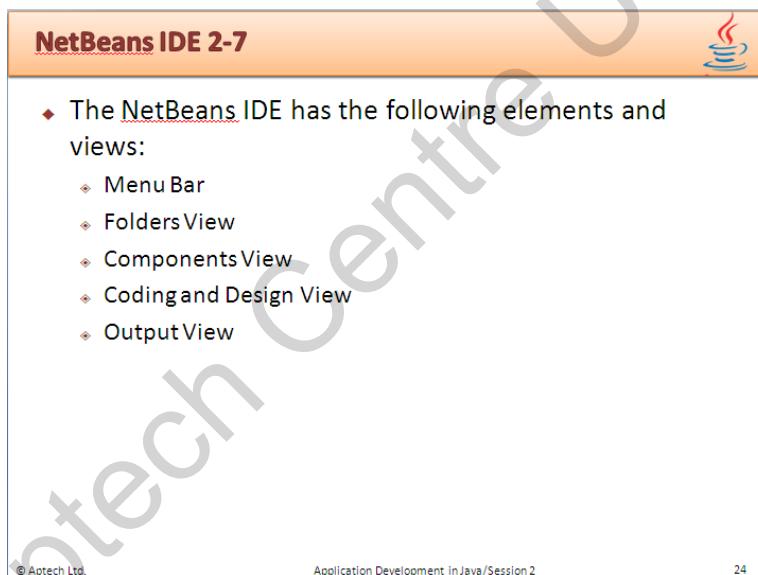
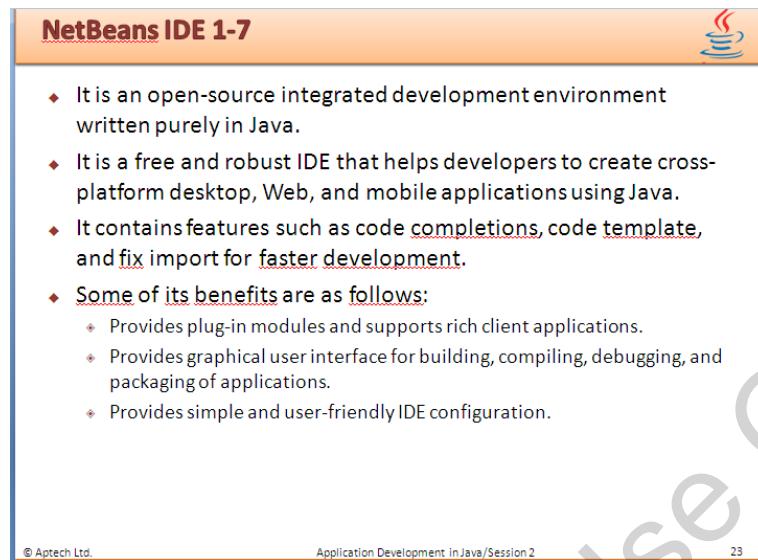
Using slide 22, explain the lists of the options that can be used with the Java command.

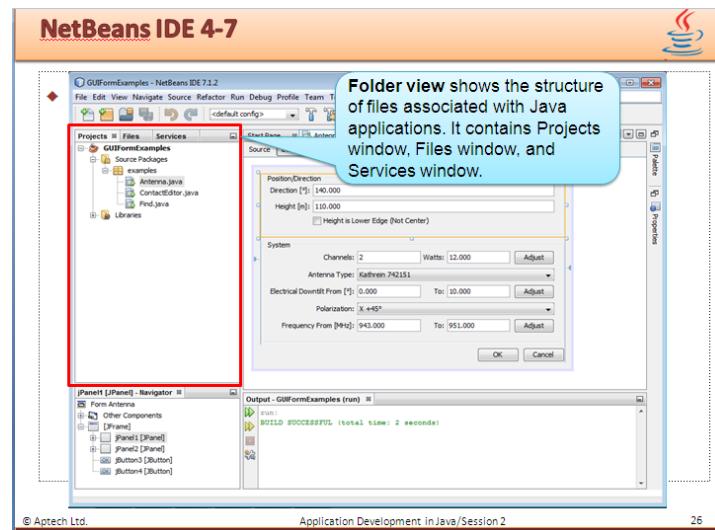
Tips:

Check this site for more information on Java command options to execute the Java code:
<http://docs.oracle.com/javase/7/docs/technotes/tools/windows/Java.html>.

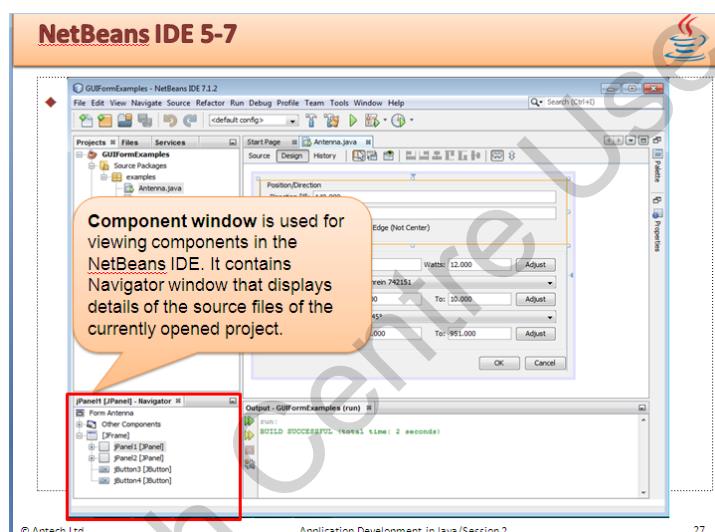
Slides 23 to 29

Let us understand NetBeans IDE.

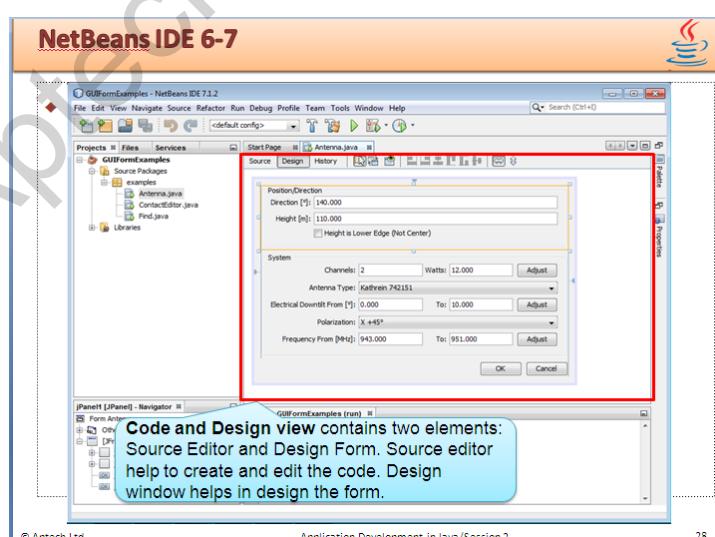




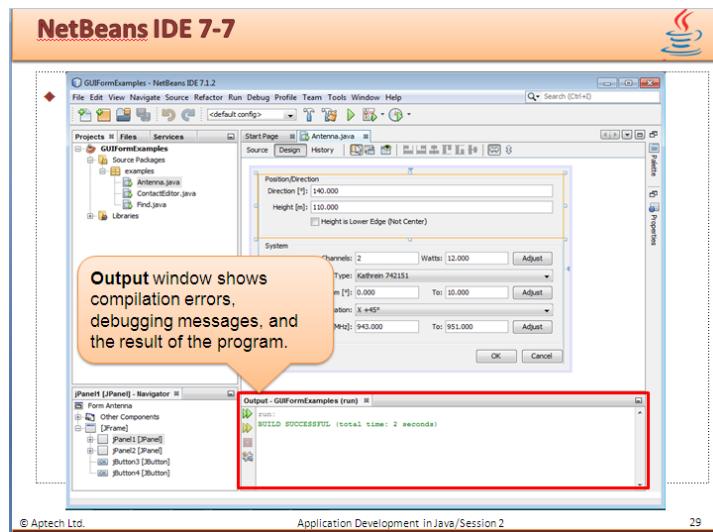
26



27



28



Using slides 23 to 29, explain the NetBeans IDE.

NetBeans is an open-source integrated development environment written purely in Java. It is a free and robust IDE that helps developers to create cross-platform desktop, Web, and mobile applications using Java. It is an application platform framework for Java desktop applications and others. The NetBeans platform allows application to be developed from a set of modular software components called modules. It contains features such as code completion, code template, and fix import for faster development.

Some of its benefits are as follows:

- Provides plug-in modules and supports rich client applications.
- Provides graphical user interface for building, compiling, debugging, and packaging of applications.
- Provides simple and user-friendly IDE configuration.

Then, using slides 25 to 29, explain all the NetBeans IDE elements.

Tips:

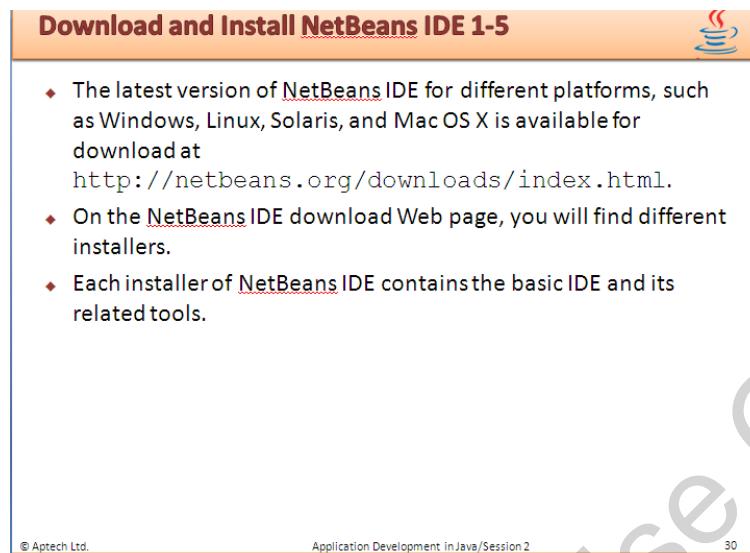
Some of the tips to work with the source editor in NetBeans IDE are as follows:

- You can right-click into the Source Editor and write code. When you write code this way, editor automatically highlights code in color as appropriate and as you type, it provides suggestions for code-completion.
- You can customize colors in the highlighting by going to Tools → Options → Fonts and Colors.
- Code completion finishes package names, classes, interfaces, and common methods. To deactivate this feature, press Esc or you can deactivate it from Tools → Options → Editor → General. Under Code Completion, deselecting the Auto Popup Completion Window checkbox.
- You can also save time by assigning abbreviations that the Source Editor expands for you. To set, go to Tools → Options → Editor → Code Templates. Type the first few letters of an abbreviation and press the spacebar. The Source Editor then expands the abbreviation.
- To turn the line numbering, right-click the left sidebar of the Source Editor's and select Show Line Numbers.
- To add commonly-used code templates for your code, press Ctrl-I in the Source Editor to view the Insert Code pop-up menu. Add properties to classes, create constructors, generate accessor methods, and override methods from superclasses.

Slides 30 to 34

Let us understand download and install NetBeans IDE.

Download and Install NetBeans IDE 1-5

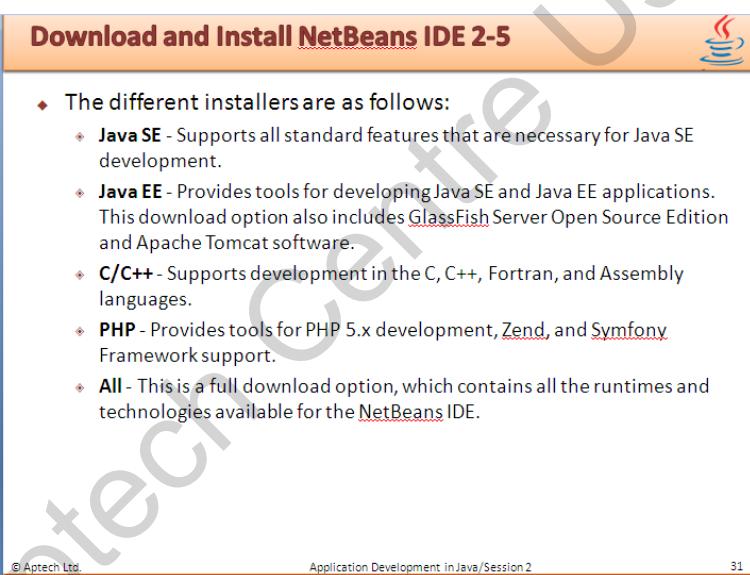


The slide shows a screenshot of the NetBeans IDE 1-5 download page. The title is "Download and Install NetBeans IDE 1-5". The content lists three bullet points about the download process:

- ◆ The latest version of NetBeans IDE for different platforms, such as Windows, Linux, Solaris, and Mac OS X is available for download at <http://netbeans.org/downloads/index.html>.
- ◆ On the NetBeans IDE download Web page, you will find different installers.
- ◆ Each installer of NetBeans IDE contains the basic IDE and its related tools.

At the bottom of the slide, there is a watermark "For Aptech Certified Use Only".

Download and Install NetBeans IDE 2-5



The slide shows a screenshot of the NetBeans IDE 2-5 download page. The title is "Download and Install NetBeans IDE 2-5". The content lists several bullet points about the different download options:

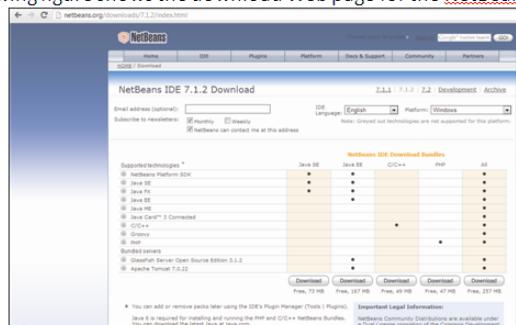
- ◆ The different installers are as follows:
 - ◆ **Java SE** - Supports all standard features that are necessary for Java SE development.
 - ◆ **Java EE** - Provides tools for developing Java SE and Java EE applications. This download option also includes GlassFish Server Open Source Edition and Apache Tomcat software.
 - ◆ **C/C++** - Supports development in the C, C++, Fortran, and Assembly languages.
 - ◆ **PHP** - Provides tools for PHP 5.x development, Zend, and Symfony Framework support.
 - ◆ **All** - This is a full download option, which contains all the runtimes and technologies available for the NetBeans IDE.

At the bottom of the slide, there is a watermark "For Aptech Certified Use Only".

Download and Install NetBeans IDE 3-5



- ◆ To download the NetBeans IDE 7.1.2, perform the following steps:
 - ◆ Type <http://netbeans.org/downloads/7.1.2/index.html> in the **Address bar** of the Web browser.
 - ◆ Following figure shows the download Web page for the NetBeans 7.1.2 IDE:



© Aptech Ltd.

Application Development in Java/Session 2

32

Download and Install NetBeans IDE 4-5



- ◆ Select **IDE language** as **English** from the drop-down list. Also, select the **Platform** as **Windows** from the drop-down list.
- ◆ Click **Download** under the installer **All**. The **Save As** dialog box is opened with **netbeans-7.1.2-ml-windows.exe** installer file. This installer will support development of all technologies in the NetBeans IDE.
- ◆ Click **Save** to save the installer file on the local system.

- ◆ To install the NetBeans IDE, perform the following steps:



- Double-click **netbeans-7.1.2-ml-windows.exe** to run the installer.
- Click **Next** at the **Welcome** page of the installation wizard.
- Click **Next** in the **License Agreement** page after reviewing the license agreement, and select the acceptance check box.

© Aptech Ltd.

Application Development in Java/Session 2

33

Download and Install NetBeans IDE 5-5



- 4 • At the [JUnit License Agreement page](#), decide if you want to install JUnit and click the appropriate option, click **Next**.
- 5 • Select either the default installation directory or specific directory where the NetBeans IDE needs to be installed. Set the path of the **default JDK** installation and click **Next**.
- 6 • The [GlassFish Server Source Edition 3.1.2](#) installation page is displayed. You can either select the default location or specify another location to install the GlassFish Server.
- 7 • To install the [Apache Tomcat Server](#), on the installation page, either select the default location or specify another location and then, click **Next**.
- 8 • The **Summary** page is opened. The list of components that are to be installed is displayed,
- 9 • Click **Install** to install the NetBeans IDE on the system.
- 10 • After the installation is completed, click **Finish** to complete and close the setup page.

Using slides 30 to 34, explain the steps to download and install NetBeans IDE.

The latest version of NetBeans IDE for different platforms, such as Windows, Linux, Solaris, and Mac OS X is available for download at <http://netbeans.org/downloads/index.html>.

On the NetBeans IDE download Web page, you will find different installers. Each installer of NetBeans IDE contains the basic IDE and its related tools. The different installers are namely, Java SE, Java EE, C/C++, PHP, All.

Then using slides 32, 33, and 34, explain the steps to download the NetBeans IDE 7.1.2.

Slide 35

Let us understand writing a Java program using NetBeans IDE.

Writing a Java Program Using NetBeans IDE



- ◆ The basic requirements to write a Java program using the NetBeans IDE is as follows:
 - ◆ The **JDK 7** installed and configured on the system
 - ◆ The **NetBeans IDE**
- ◆ To develop a Java program in NetBeans IDE, perform the following steps:

Create a project in IDE



Add code to the generated source files



Build and execute Java program

Using slide 35, explain that the basic requirements to write a Java program using the NetBeans IDE is as follows:

- The **JDK 7** installed and configured on the system – It is a development environment for developing applications, applets, and components.

- The NetBeans IDE - It is an open-source integrated development environment written purely in Java.

Slides 36 and 37

Let us understand to create a project in IDE.

Create a Project in IDE 1-2



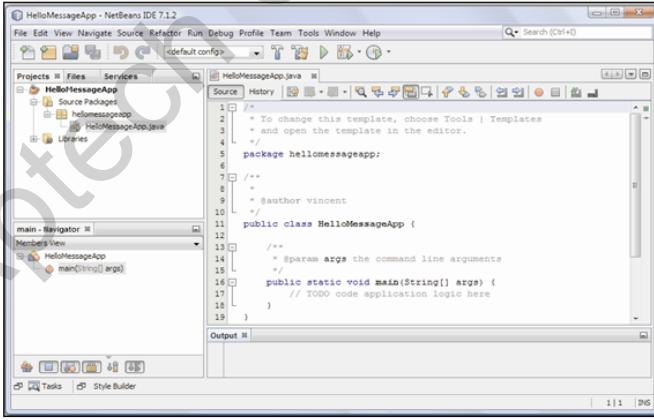
- To create a project in IDE, perform the following steps:
 - To launch NetBeans IDE, click **Start → All Programs → NetBeans** and select **NetBeans IDE 7.1.2**.
 - To create a new project, click **File → New → Project**. This opens the **New Project** wizard.
 - Under **Categories**, expand **Java** and then, select **Java Application** under Projects.
 - Click **Next**. This displays the **Name and Location** page in the wizard.
 - Type **HelloMessageApp** in the **Project Name** box.
 - Click **Browse** and select the appropriate location on the system.
 - Click **Finish**.

© Aptech Ltd. Application Development in Java/Session 2 36

Create a Project in IDE 2-2



- Following figure shows the **HelloMessageApp** project in the **NetBeans IDE**:



© Aptech Ltd. Application Development in Java/Session 2 37

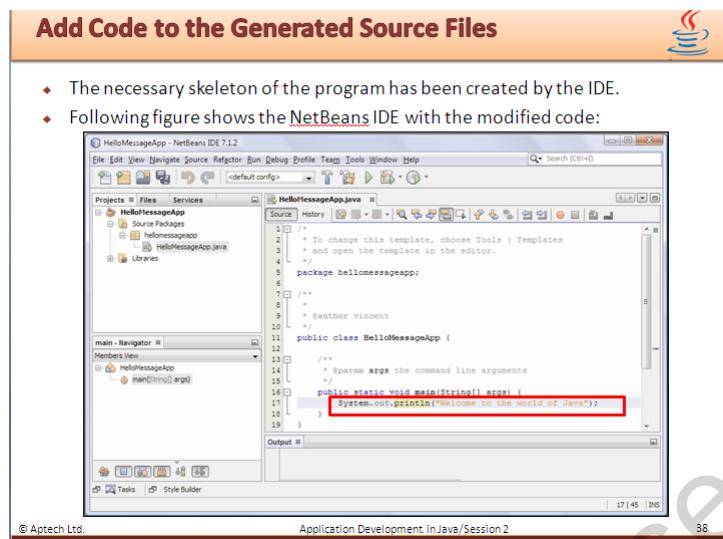
Using slides 36 and 37, explain the steps to create project in NetBeans IDE.

In setting the project, the application created has two projects which are a Java class library project in which a utility class is created whereas a Java application project with a main class that implements a method from the library project utility class.

Explain the steps for creating a project in NetBeans IDE.

Slide 38

Let us understand how to add code to the generated source files.



Using slide 38, explain to add code to the generated source files.

When creating projects using **New project wizard**, the **Create main class** checkbox is selected to create the project with a skeleton. The necessary skeleton of the program has been created by the IDE.

Using slide 38, explain how to add the statement, `System.out.println()`, in the **HelloMessageApp** class. This statement will print the provided string on the console.

Tips:

The 'out' is a variable. There are two possibilities – it could be a static or an instance variable. Because 'out' is being called with the 'System' class name itself, and not an instance of a class (an object), then we know that 'out' must be a static variable, since only static variables can be called with just the class name itself.

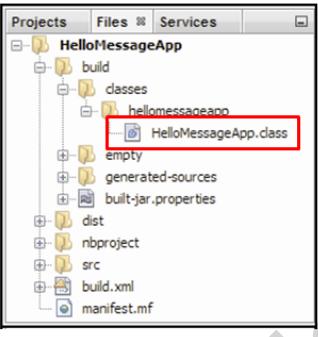
So 'out' is a static member variable belonging to the `System` class. It belongs to the `PrintStream` class and provides the `println()` method.

Slides 39 and 40

Let us understand to build and execute Java program in NetBeans IDE.

Build and Execute Java Program in NetBeans IDE 1-2

- ◆ To compile the source file, `HelloMessageApp.java`, click Run→Build Main Project in the NetBeans IDE menu bar.
- ◆ Following figure shows Files window that shows the generated bytecode file, `HelloMessageApp.class`, after the project is build successfully:



The screenshot shows the NetBeans IDE's Files window for the project "HelloMessageApp". The "build" folder contains a "classes" folder which holds the "HelloMessageApp.class" file, which is highlighted with a red box. Other files in the "build" folder include "empty", "generated-sources", "built-jar.properties", and "nbproject". The "src" folder contains "build.xml" and "manifest.mf".

© Aptech Ltd. Application Development in Java/Session 2 39

Build and Execute Java Program in NetBeans IDE 2-2

- ◆ To execute the program, click Run→Run Main Project.
- ◆ Following figure shows the Output window that displays the output of the `HelloMessageApp` program.



The screenshot shows the NetBeans IDE's Output window titled "Output - HelloMessageApp (run)". It displays the following text:
 run:
 Welcome to the world of Java
 BUILD SUCCESSFUL (total time: 0 seconds)

© Aptech Ltd. Application Development In Java/Session 2 40

Using slides 39 and 40, explain how to build and execute the Java program in NetBeans IDE.

Explain the steps to build and execute the code in NetBeans IDE. In Build process, the bytecode file is generated.

Slide 41

Let us understand comments in Java.

Comments in Java

- ◆ Are placed in a Java program source file.
- ◆ Are used to document the Java program and are not compiled by the compiler.
- ◆ Are added as remarks to make the program more readable for the user.
- ◆ Are of three types:
 - ◆ Single-line comments
 - ◆ Multi-line comments
 - ◆ Javadoc comments

© Aptech Ltd. Application Development in Java/Session 2 41

Using slide 41, explain comments in Java.

Comments allow inserting text that will not be compiled or interpreted. Comments in Java are placed in a Java program source file. They are used to document the Java program and are not by the compiler and are added as remarks to make the program more readable for the user. It is useful to explain the adopted technical choice and to give the required explanations to understand the code.

Comments in Java are of three types:

- Single-line comments
- Multi-line comments
- Javadoc comments

Slides 42 and 43

Let us understand single-line comments.

Single-line Comments 1-2

- ◆ A single-line comment is used to document the functionality of a single line of code.
- ◆ There are two ways of using single-line comments that are as follows:
 - Beginning-of-line comment**
 - This type of comment can be placed before the code (on a different line).
 - End-of-line comment**
 - This type of comment is placed at the end of the code (on the same line).
- ◆ The syntax for applying the comments is as follows:

Syntax

```
// Comment text
```

© Aptech Ltd. Application Development in Java/Session 2 42

Single-line Comments 2-2

- Following code snippet shows the different ways of using single-line comments in a Java program:

```
...
// Declare a variable
int a = 32;
int b // Declare a variable
...
```

- Conventions for using single-line comments are as follows:
 - Insert a space after the forward slashes.
 - Capitalize the first letter of the first word.

© Aptech Ltd. Application Development in Java/Session 2 43

Using slides 42 and 43, explain the single-line comment.

Java single-line comment starts with two forward slashes with no white spaces and lasts till the end of line. A single-line comment is used to document the functionality of a single line of code. It is useful to provide short explanations for variables, functions, and expressions.

There are two ways of using single-line comments that are as follows:

- Beginning-of-line comment - This type of comment can be placed before the code (on a different line).
- End-of-line comment - This type of comment is placed at the end of the code (on the same line).

Then, explain the conventions for using single-line comments in Java code.

Slide 44

Let us understand multi-line comments.

Multi-line Comments

- Is a comment that spans multiple lines.
- Starts with a forward slash and an asterisk (*).
- Ends with an asterisk and a forward slash (* /).
- Anything that appears between these delimiters is considered to be a comment.
- Following code snippet shows a Java program that uses multi-line comments:

```
...
/*
 * This code performs mathematical
 * operation of adding two numbers.
 */
int a = 20;
int b = 30;
int c;
c = a + b;
...
```

© Aptech Ltd. Application Development in Java/Session 2 44

Using slide 44, explain the multi-line comments.

Multi-line Comments is a comment that spans multiple lines. It starts with a forward slash and an asterisk (/*) and ends with an asterisk and a forward slash (* /). Anything that appears between these delimiters is considered to be a comment. It is useful when the comment text does not fit in one line or can be used when the comment need to be inserted in the middle of the code.

In-Class Question:

After you finish explaining the multi-line comments, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



Why multi-line comments are useful?

Answer:

Multi-line comments are useful when the comment text does not fit in one line or can be used when the comment need to be inserted in the middle of the code.

Slides 45 and 46

Let us understand Javadoc comments.

Javadoc Comments 1-2

- ◆ Is used to document public or protected classes, attributes, and methods.
- ◆ Starts with /** and ends with */.
- ◆ Everything between the delimiters is a comment.
- ◆ The javadoc command can be used for generating Javadoc comments.
- ◆ Following code snippet demonstrates the use of Javadoc comments in the Java program:

```
/**  
 * The program prints the welcome message  
 * using the println() method.  
 */  
package hellomessageapp;
```

© Aptech Ltd. Application Development In Java/Session 2 45

Javadoc Comments 2-2



```

/**
 *
 * @author vincent
 */
public class HelloMessageApp {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {

        // The println() method displays a message on the screen
        System.out.println("Welcome to the world of Java");

    }
}

```

© Aptech Ltd. Application Development in Java/Session 2 46

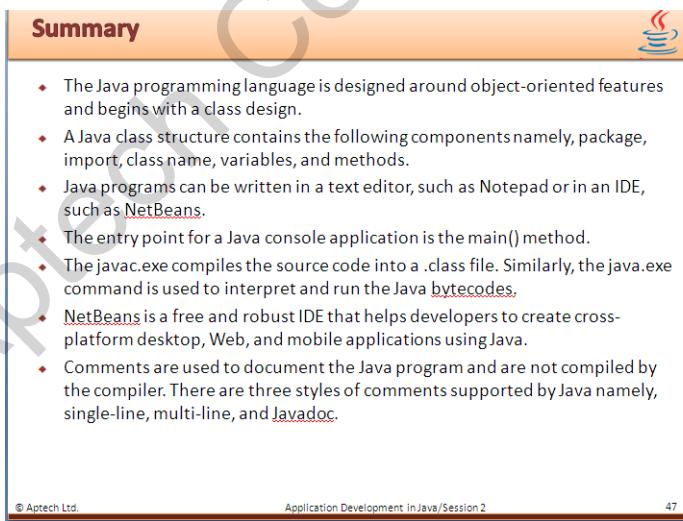
Using slides 45 and 46, explain Javadoc comments.

Javadoc comment is used to document public or protected classes, attributes, and methods. Also to document the API which is developed as a part of source code and kept in source file. It starts with `/**` and ends with `*/`. Everything between the delimiters is a comment. The Javadoc command can be used for generating Javadoc comments. The Javadoc is a tool which comes as a part of JDK. By using this comment, classes, fields, constructors, and methods are documented.

Slides 47

Let us summarize the session.

Summary



- ◆ The Java programming language is designed around object-oriented features and begins with a class design.
- ◆ A Java class structure contains the following components namely, package, import, class name, variables, and methods.
- ◆ Java programs can be written in a text editor, such as Notepad or in an IDE, such as NetBeans.
- ◆ The entry point for a Java console application is the `main()` method.
- ◆ The `javac.exe` compiles the source code into a `.class` file. Similarly, the `java.exe` command is used to interpret and run the Java `bytecodes`.
- ◆ NetBeans is a free and robust IDE that helps developers to create cross-platform desktop, Web, and mobile applications using Java.
- ◆ Comments are used to document the Java program and are not compiled by the compiler. There are three styles of comments supported by Java namely, single-line, multi-line, and Javadoc.

© Aptech Ltd. Application Development in Java/Session 2 47

In slide 47, you will summarize the session. End the session with a brief summary of what has been taught in the session.

2.3 Post Class Activities for Faculty

You should familiarize yourself with the topics of the next session that is based on how to deal with variables and operators in Java.

Tips:

You can also check the Articles/Blogs/Expert Videos uploaded on the Online Varsity site to gain additional information related to the topics covered in the next session. You can also connect to online tutors on the Online Varsity site to ask queries related to the sessions.

Session 3 – Variables and Operators

3.1 Pre-Class Activities

Before you commence the session, you should familiarize yourself with the topics of this session in-depth.

Here, you can discuss the key points with the students that were covered in the previous session. Prepare a question or two which will help you to relate the current session objectives.

3.1.1 Objectives

By the end of this session, the learners will be able to:

- Explain variables and their purpose
- State the syntax of variable declaration
- Explain the rules and conventions for naming variables
- Explain data types
- Describe primitive and reference data types
- Describe escape sequence
- Describe format specifiers
- Identify and explain different type of operators
- Explain the concept of casting
- Explain implicit and explicit conversion

3.1.2 Teaching Skills

To teach this session, you should be well-versed with usage of variables, literals, data types, and constants in Java. You should be familiar with different types of operators and their usage in writing the expressions in the Java program.

You should teach the concepts in the theory class using the images provided. For teaching in the class, you are expected to use slides and LCD projectors.

Tips:

It is recommended that you test the understanding of the students by asking questions in between the class.

In-Class Activities:

Follow the order given here during In-Class activities.

Overview of the Session:

Give the students the overview of the current session in the form of session objectives. Show the students slide 2 of the presentation.

Objectives	
<ul style="list-style-type: none">◆ Explain variables and their purpose◆ State the syntax of variable declaration◆ Explain the rules and conventions for naming variables◆ Explain data types◆ Describe primitive and reference data types◆ Describe escape sequence◆ Describe format specifiers◆ Identify and explain different type of operators◆ Explain the concept of casting◆ Explain implicit and explicit conversion	

© Aptech Ltd. Variables and Operators/Session 3 2

Tell them that Java is an object-oriented programming language used for developing various types of applications that can be executed on any platform. Tell the student that this session will focus on the usage of variables and different data types that are present in Java programming language.

They will also learn about the various operators and their implementation in Java programs. The session also explains the different type of conversions used in Java to convert the value from one data type onto another.

3.2 In-Class Explanations

Slide 3

Let us understand the use of variables in Java.

Introduction



- ◆ The core of any programming language is the way it stores and manipulates the data.
- ◆ The Java programming language can work with different types of data, such as number, character, boolean, and so on.
- ◆ To work with these types of data, Java programming language supports the concept of variables.
- ◆ A variable is like a container in the memory that holds the data used by the Java program.
- ◆ A variable is associated with a data type that defines the type of data that will be stored in the variable.
- ◆ Java is a strongly-typed language which means that any variable or an object created from a class must belong to its type and should store the same type of data.
- ◆ The compiler checks all expressions variables and parameters to ensure that they are compatible with their data types.

Using slide 3, explain the concept of variables in Java.

The core of any programming language is the way it stores and manipulates the data. Thus, to work with the data in a program, you need variables. They act as a container that holds data to be processed by the Java program. Tell the students that in a Java program variables store data that changes during the execution of the program. After declaring a variable, compiler reserves an area of the memory to put appropriate data into it.

The type of data that a variable can hold depends upon the data type assigned to the variable. In Java every variable must have a data type, hence, the variable is associated with a data type that defines the type of data that will be stored in the variable.

A variety of data types are built into the Java language to work with different types of data, such as number, character, boolean, and so on. Every variable must be declared to use a data type. For instance, a variable could be declared to use one of the eight primitive data types namely, byte, short, int, long, float, double, char, or boolean. Every variable must be given an initial value, before it can be used.

Java is a strongly-typed language which means that any variable or an object created from a class must belong to its type and should store the same type of data. The compiler checks all expressions variables and parameters to ensure that they are compatible with their data types.

Slides 4 and 5

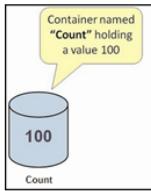
Let us understand the declaration of variables.

Variables 1-2



A variable is a location in the computer's memory which stores the data that is used in a Java program.

- Following figure depicts a variable that acts as a container and holds the data in it:



- Variables
 - are used in a Java program to store data that changes during the execution of the program.
 - are the basic units of storage in a Java program.
 - can be declared to store values, such as names, addresses, and salary details.
 - must be declared before they can be used in the program.
- A variable declaration begins with data type and is followed by variable name and a semicolon.

© Aptech Ltd. Variables and Operators/Session 3 4

Variables 2-2



- The data type can be a primitive data type or a class.
- The syntax to declare a variable in a Java program is as follows:

Syntax

```
datatype variableName;
```

where,

- datatype: Is a valid data type in Java.
- variableName: Is a valid variable name.

- Following code snippet demonstrates how to declare variables in a Java program:

```
...
int rollNumber;
char gender;
...
```

- In the code, the statements declare an integer variable named rollNumber, and a character variable called gender.
- These variables will hold the type of data specified for each of them.

© Aptech Ltd. Variables and Operators/Session 3 5

Using slides 4 and 5, explain the syntax and example to work with the variables.

The variable is a location in the computer's memory, where the data is stored and from which the value can be retrieved later. Tell the students that in a Java program variables store data that changes during the execution of the program. They are the basic units of storage in a Java program. Discuss some of the example for declaring variables in a class. For instance to store the employee details in an application, you need to declare variables such as name, address, salary, and so on.

Explain to them the syntax for declaring variables. Variables must be declared before they can be used in the program. A variable declaration begins with data type and is followed by variable name and a semicolon. The data type can be a primitive data type or a class.

For example,

```
int rollNumber;  
char gender;
```

Tell the students that these statements declare an integer variable called **rollNumber**, and a character variable called **gender**. These statements instruct the JVM to allocate the required amount of memory to each variable in order to hold the type of data specified for each.

Explain to them that the name can be used within the program to access the values stored in each of the variables. Values can be assigned to variables by using the assignment operator (=) as follows:

```
rollNumber = 101;  
gender = 'M';
```

Also, tell the students that values can be assigned to a variable upon creation, as shown:

```
int rollNumber = 101;
```

Tips:

Multiple variables also can be declared and initialized at the same time as shown:

```
int a=5, b, c=10;
```

The statement initializes **a** and **c**, but declares three variables.

In-Class Question:

After you finish explaining variables, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



How to assign the value of one variable to another variable?

Answer:

Using assignment operator (=).

For example, int a =10; int b = a;

Slides 6 and 7

Let us understand rules for naming variables.

Rules for Naming Variables 1-2



- Variable names may consist of Unicode letters and digits, underscore (_), and dollar sign (\$).
- A variable's name must begin with a letter, the dollar sign (\$), or the underscore character (_).
 - The convention, however, is to always begin your variable names with a letter, not 'S' or '_'.
- Variable names must not be a keyword or reserved word in Java.
- Variable names in Java are case-sensitive.
 - For example, the variable names `number` and `Number` refer to two different variables.
- If a variable name comprises a single word, the name should be in lowercase.
 - For example, `velocity` or `ratio`.
- If the variable name consists of more than one word, the first letter of each subsequent word should be capitalized.
 - For example, `employeeNumber` and `accountBalance`.

© Aptech Ltd. Variables and Operators/Session 3

Rules for Naming Variables 2-2



- Following table shows some examples of valid and invalid Java variable names:

Variable Name	Valid/Invalid
<code>rollNumber</code>	Valid
<code>a2x5_w7t3</code>	Valid
<code>\$yearly_salary</code>	Valid
<code>_2010_tax</code>	Valid
<code>\$\$</code>	Valid
<code>amount#Balance</code>	Invalid and contains the illegal character #
<code>double</code>	Invalid and is a keyword
<code>4short</code>	Invalid and the first character is a digit

© Aptech Ltd. Variables and Operators/Session 3

Using slides 6 and 7, explain the naming conventions for variables in Java.

Tell them that every programming language has its own set of rules and conventions for the kinds of names that you're allowed to use, and the Java programming language is also not different. For instance, variable names should be short and meaningful. Not adhering to the rules will result in syntax errors.

Explain the naming conventions that should be followed while declaring variables in Java.

Tell them that the rules and conventions for naming your variables are as follows:

- Variable names may consist of Unicode letters and digits, underscore (_), and dollar sign (\$).
- A variable's name must begin with a letter, the dollar sign (\$), or the underscore character (_). The convention, however, is to always begin your variable names with a letter, not '\$' or '_'.
- Variable names must not be a keyword or reserved word in Java.
- Variable names in Java are case-sensitive (for example, the variable names, **number** and **Number** refer to two different variables).
- If a variable name comprises a single word, the name should be in lowercase (for example, velocity or ratio).
- If the variable name consists of more than one word, the first letter of each subsequent word should be capitalized (for example, **employeeNumber** and **accountBalance**).

Tips:

If your variable stores a constant value, such as static final int NUM_GEARs = 6, the convention changes slightly, capitalizing every letter and separating subsequent words with the underscore character. By convention, the underscore character is never used elsewhere.

Slides 8 to 10

Let us understand assigning value to a variable.

Assigning Value to a Variable 1-3



- ◆ Values can be assigned to variables by using the assignment operator (=).
- ◆ There are two ways to assign value to variables. These are as follows:

At the time of declaring a variable

- ◆ Following code snippet demonstrates the initialization of variables at the time of declaration:

```

...
int rollNumber = 101;
char gender = 'M'; Assigning Value to a Variable 1-
...

```

- ◆ In the code, variable **rollNumber** is an integer variable, so it has been initialized with a numeric value **101**.
- ◆ Similarly, variable **gender** is a character variable and is initialized with a character '**M**'.
- ◆ The values assigned to the variables are called as literals.

Literals are constant values assigned to variables directly in the code without any computation.

© Aptech Ltd.

Variables and Operators/Session 3

8

©Aptech Limited

Assigning Value to a Variable 2-3



After the variable declaration

- Following code snippet demonstrates the initialization of variables after they are declared:

```
int rollNumber; // Variable is declared
...
rollNumber = 101; //variable is initialized
...
```

- Here, the variable `rollNumber` is declared first and then, it has been initialized with the numeric literal `101`.
- Following code snippet shows the different ways for declaring and initializing variables in Java:

```
// Declares three integer variables x, y, and z
int x, y, z;

// Declares three integer variables, initializes a and c
int a = 5, b, c = 10;

// Declares a byte variable num and initializes its value to 20
byte num = 20;
```

© Aptech Ltd.

Variables and Operators/Session 3

9

Assigning Value to a Variable 3-3



```
// Declares the character variable c with value 'c'
char c = 'c';

// Stores value 10 in num1 and num2
int num1 = num2 = 10; //
```

- In the code, the declarations, `int x, y, z;` and `int a=5, b, c=10;` are examples of comma separated list of variables.
- The declaration `int num1 = num2 = 10;` assigns same value to more than one variable at the time of declaration.

© Aptech Ltd.

Variables and Operators/Session 3

10

Using slides 8 to 10, explain how to assign the values to a variable.

The values can be assigned to the variable in two ways, that is, at the time of variable declaration or after the variable is declared. The value assigned to the variable is also called as literal. Literals are constant values assigned to variables directly in the code without any computation.

At the time of declaring a variable

```
...
int rollNumber = 101;
```

In the code, variable `rollNumber` is an integer variable, so it has been initialized with a numeric value `101`.

After the variable declaration

```
int rollNumber; // Variable is declared
...
rollNumber = 101; //variable is initialized
...
```

Here, the variable **rollNumber** is declared first and then, it has been initialized with the numeric literal 101.

Slides 11 to 13

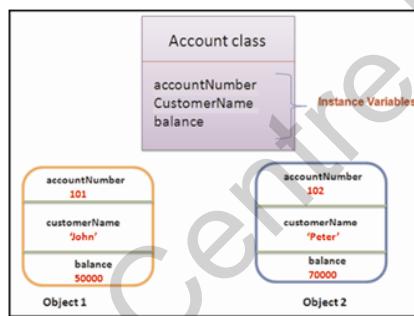
Let us understand different kinds of variables in Java.

Different Types of Variables 1-3

- Java programming language allows you to define different kind of variables that are categorized as follows:

Instance variables

- The state of an object is represented as fields or attributes or instance variables in the class definition.
- Each object created from a class will have its own copy of instance variables.
- Following figure shows the instance variables declared in a class template:



© Aptech Ltd.

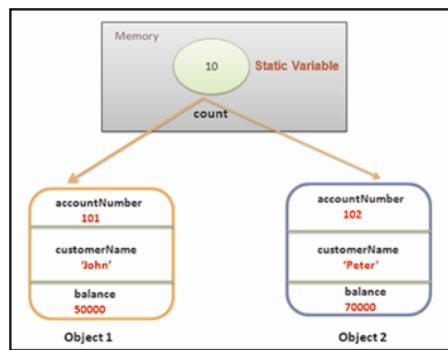
Variables and Operators/Session 3

11

Different Types of Variables 2-3

Instance variables

- These are also known as class variables.
- Only one copy of static variable is maintained in the memory that is shared by all the objects belonging to that class.
- These fields are declared using the `static` keyword.
- Following figure shows the static variables in a Java program:



© Aptech Ltd.

Variables and Operators/Session 3

12

Different Types of Variables 3-3



Local variables

- ◆ The variables declared within the blocks or methods of a class are called local variables.
- ◆ A method represents the behavior of an object.
- ◆ The local variables are visible within those methods and are not accessible outside them.
- ◆ A method stores its temporary state in local variables.
- ◆ There is no special keyword available for declaring a local variable, hence, the position of declaration of the variable makes it local.

Using slides 11 to 13, explain that Java programming language allows you to define different kind of variables that are categorized as follows:

Instance variables

They are declared in class but outside methods, constructor or any block. The state of an object is represented as fields or attributes or instance variables in the class definition.

They are created when an object is created with the use of 'new' keyword and destroyed when the object is destroyed. Each object created from a class will have its own copy of instance variables.

Instance variables can be accessed directly by calling the variable name inside the class.

Static variables

These are also known as class variables and declared with the static keyword in a class. Only one copy of static variable is maintained in the memory that is shared by all the objects belonging to that class. Static variables are created when the program starts and destroyed when the program stops. They are stored in static memory.

Local variables

The variables declared within the blocks or methods of a class are called local variables. A method represents the behavior of an object. The local variables are visible within those methods and are not accessible outside them. A method stores its temporary state in local variables. There is no special keyword available for declaring a local variable, hence, the position of declaration of the variable makes it local. There is no default value for local variable, so local variables can be declared and an initial value can be assigned before the first use.

In-Class Question:

After you finish explaining the different types of variables, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



Where are the instance variable declared and when are they created?

Answer:

Instance variables are declared within the class definition and they are created when an object of the class is instantiated in the memory. The object is created with the use of 'new' keyword.

Slides 14 to 16

Let us understand scope and lifetime of variables.

Scope and Lifetime of Variables 1-3

- ◆ In Java, variables can be declared within a class, method, or within any block.
- ◆ A scope determines the visibility of variables to other part of the program.

```

graph TD
    C[C/C++]
    L1[Local]
    G1[Global]
    J[Java]
    C2[Class]
    M[Method]

    C --> L1
    C --> G1
    J --> C2
    J --> M
  
```

Class scope

- ◆ The variables declared within the class can be instance variables or static variables.
- ◆ The instance variables are owned by the objects of the class and their existence or scope depends upon the object creation.
- ◆ Static variables are shared between the objects and exists for the lifetime of a class.

© Aptech Ltd. Variables and Operators/Session 3 14

Scope and Lifetime of Variables 2-3

Method scope

- ◆ The variables defined within the methods of a class are local variables.
- ◆ The lifetime of these variables depends on the execution of methods.
- ◆ This means memory is allocated for the variables when the method is invoked and destroyed when the method returns.
- ◆ After the variables are destroyed, they are no longer in existence.
- ◆ Methods parameters values passed to them during method invocation.
- ◆ The parameter variables are also treated as local variables which means their existence is till the method execution is completed.

© Aptech Ltd. Variables and Operators/Session 3 15

Scope and Lifetime of Variables 3-3



- Following figure shows the scope and lifetime of variables **x** and **y** defined within the Java program:

```
public class ScopeOfVariables {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // Known to code within main() method
        int x; ← Variable x is accessible within the
        x = 10;

        { // Starts a block with new scope
            int y = 20; ← Variable y is visible only within the
            System.out.println("x and y: " + x + " " + y);
            // Calculates value for variable x
            x = y * 2;
        } // End of the block

        // y = 100; // Error! y not known here
        // x is accessible
        System.out.println("x is: " + x);
    }
}
```

© Aptech Ltd.

Variables and Operators/Session 3

16

Using slides 14 to 16, explain the scope and lifetime of the variables.

In Java, variables can be declared within a class, method, or within any block. A scope determines the visibility of variables to other part of the program. The scope of a variable defines the section of the code in which the variable is visible.

The lifetime of a variable refers to how long the variable exists before it destroyed. Destroying variables means deallocated the memory that was allotted to the variables when declaring it.

Class scope

The variables declared within the class can be instance variables or static variables. The instance variables are owned by the objects of the class and their existence or scope depends upon the object creation. In this, any method in the class definition can access these variables. The static variables are shared between the objects and exist for the lifetime of a class.

Tell them that if any variable is declared with the `static` modifier. This tells the compiler that there is exactly one copy of this variable in existence, regardless of how many times the class has been instantiated.

Method scope

The variables defined within the methods of a class are local variables. The lifetime of these variables depends on the execution of methods. This means memory is allocated for the variables when the method is invoked and destroyed when the method returns. After the variables are destroyed, they are no longer in existence.

Methods parameters values passed to them during method invocation. The parameter variables are also treated as local variables which means their existence is till the method execution is completed.

Then, explain the example provided on slide 16 to understand the scope of variables within the method.

Slide 17

Let us understand data types.

Data Types

- When you define a variable in Java, you must inform the compiler what kind of a variable it is.
- That is, whether it will be expected to store an integer, a character, or some other kind of data.
- This information tells the compiler how much space to allocate in the memory depending on the data type of a variable.
- Thus, the data types determine the type of data that can be stored in variables and the operation that can be performed on them.

In Java, data types fall under two categories that are as follows:

Primitive data types **Reference data types**

© Aptech Ltd. Variables and Operators/Session 3 17

Using slide 17, explain to the students about the data types in Java.

Tell them that Java programming language is statically-typed. This means that all variables need to be declared before they can be used.

Explain to the students that when you define a variable in Java, you must tell the compiler what kind of a variable it is. That is, whether it will be expected to store an integer, a character, or some other kind of data.

This information is required for the following reasons:

- It tells the compiler how much space to allocate in the memory depending on the data type of the variable
- It also determines the type of data that can be stored in the variable
- It determines the type of operations that can be performed on this data

Mention to the students that Java is a strongly typed language. This means that every variable has a type, every expression has a type, and every type is strictly defined. It also means that all assignments, whether explicit or via parameter passing in method calls, are checked for type compatibility by the Java compiler. Any type mismatches are considered as errors and need to be corrected before compilation is over.

There are two data types available in Java programming language:

- Primitive Data Types
- Reference/Object Data Types

Slide 18

Let us understand primitive data types.

Primitive Data Types



The Java programming language provides eight primitive data types to store data in Java programs.

A primitive data type, also called built-in data type, stores a single value at a time, such as a number or a character.

The size of each data type will be same on all machines while executing a Java program.

The primitive data types are predefined in the Java language and are identified as reserved words.

- ◆ Following figure shows the primitive data types that are broadly grouped into four groups:

```

graph TD
    P[Primitive Data Type] --> I[Integer]
    P --> F[Float]
    P --> C[Character]
    P --> B[Boolean]
    I --> byte[byte]
    I --> short[short]
    I --> int[int]
    I --> long[long]
    F --> float[float]
    F --> double[double]
    C --> char[char]
    B --> boolean[boolean]
  
```

© Aptech Ltd.

Variables and Operators/Session 3

18

Using slide 18, explain the primitive data types in Java.

Every variable has a typed declared in the source code. Primitive type directly contains value. The Java programming language provides eight primitive data types to store data in Java programs. A primitive data types are predefined by language and named by a reserved keyword. A primitive data type, also called built-in data type, stores a single value at a time, such as a number or a character. Primitive value does not share state with other primitive value. The size of each data type will be same on all machines while executing a Java program. The primitive data types are predefined in the Java language and are identified as reserved words.

Slide 18 shows the primitive data types that are broadly grouped into four groups. They are integer, float, character, Boolean. Integer data type is further sub-divided into four types. They are byte, short, int, long. Float data type is further sub-divided into two types. They are float and double. Character data type is named by the keyword char, whereas boolean data type is named by the keyword boolean.

In-Class Question:

After you finish explaining the data types, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



Which primitive data type is useful for saving memory in large areas?

Answer:

byte and short

Slides 19 and 20

Let us understand integer types and floating-point types.

Integer Types



- ◆ The integer data types supported by Java are `byte`, `short`, `int`, and `long`.
- ◆ These data type can store signed integer values.
- ◆ Signed integers are those integers, which are capable of representing positive as well as negative numbers, such as -40.
- ◆ Java does not provide support for unsigned integers.
- ◆ Following table lists the details about the integer data types:

<code>byte</code>	<code>short</code>	<code>int</code>	<code>long</code>
A signed 8-bit type. Range: -128 to 127	A signed 16-bit type. Range: -32,768 to 32,767	Signed 32-bit type. Range: -2,147,483,648 to 2,147,483,647	Signed 64-bit type. Range: 9,223,372,036,854,775, 808 to 9,223,372,036,854,775, 807
Useful when working with raw binary data.	Used to store smaller numbers, for example, employee number.	Used to store the total salary being paid to all the employees of the company.	Used to store very large values such as population of a country.
Keyword: <code>byte</code>	Keyword: <code>short</code>	Keyword: <code>int</code>	Keyword: <code>long</code>

© Aptech Ltd.

Variables and Operators/Session 3

19

Floating-point Types



- ◆ The floating-point data types supported by Java are `float` and `double`.
- ◆ These are also called real numbers, as they represent numbers with fractional precision.
- ◆ For example, calculation of a square root or PI value is represented with a fractional part.
- ◆ The brief description of the floating-point data types is given in the following table:

<code>float</code>	<code>byte</code>
A single precision value with 32-bit storage.	A double precision with 64-bit storage.
Useful when a number needs a fractional component, but with less precision.	Useful when accuracy is required to be maintained while performing calculations.
Keyword: <code>float</code>	Keyword: <code>double</code>
For example, <code>float sguRoot, cubeRoot;</code>	For example, <code>double bigDecimal;</code>

© Aptech Ltd.

Variables and Operators/Session 3

20

Using slides 19 and 20, explain the integer and floating-point types.

Integer and floating-point values are the basic building blocks of arithmetic and computation.

Integer Types

The integer data types supported by Java are `byte`, `short`, `int`, and `long`. These data type can store signed integer values. Signed integers are those integers, which are capable of representing positive as well as negative numbers, such as -40. Java does not provide support for unsigned integers. Byte data type is used to save space in large arrays since byte is four times smaller than `int`. While short data type is also used to save memory, but it is two times smaller than `int`.

Floating-point Types

The floating-point data types supported by Java are `float` and `double`. These are also called real numbers, as they represent numbers with fractional precision. For example, calculation of a square root or PI value is represented with a fractional part. `Float` data type is mainly used to save memory in large arrays of floating-point numbers and should not be used for precise values such as currency. `Double` data type is used as a default data type for decimal values.

Slides 21 to 24

Let us understand character and boolean types.

Character and Boolean Types 1-4

- ◆ `char` data type belongs to this group and represents symbols in a character set like letters and numbers.
- ◆ `char` data type stores 16-bit Unicode character and its value ranges from 0 ('`\u0000`') to 65,535 ('`\uffff`').
- ◆ Unicode is a 16-bit character set, which contains all the characters commonly used in information processing.
- ◆ It is an attempt to consolidate the alphabets of the world's various languages into a single and international character set.
- ◆ `boolean` data type represents `true` or `false` values.
- ◆ This data type is used to track `true/false` conditions.
- ◆ Its size is not defined precisely.
- ◆ Apart from primitive data types, Java programming language also supports strings.
- ◆ A string is a sequence of characters.
- ◆ Java does not provide any primitive data type for storing strings, instead provides a class `String` to create string variables.
- ◆ The `String` class is defined within the `java.lang` package in Java SE API.

Character and Boolean Types 2-4

- ◆ Following code snippet demonstrates the use of `String` class as primitive data type:


```
. . .
String str = "A String Data";
. . .
```
- ◆ The statement, `String str` creates an `String` object and is not of a primitive data type.
- ◆ When you enclose a string value within double quotes, the Java runtime environment automatically creates an object of `String` type.
- ◆ Also, once the `String` variable is created with a value '`A String Data`', it will remain constant and you cannot change the value of the variable within the program.
- ◆ However, initializing string variable with new value creates a new `String` object.
- ◆ This behavior of strings makes them as immutable objects.

Character and Boolean Types 3-4



- Following code snippet demonstrates the use of different data types in Java:

```
public class EmployeeData {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {

        // Declares a variable of type integer
        int empNumber;
        // Declares a variable of type decimal
        float salary;
        // Declares and initialize a decimal variable
        double shareBalance = 456790.897;
        // Declare a variable of type character
        char gender = 'M';
        // Declares and initialize a variable of type boolean
        boolean ownVehicle = false;
        // Variables, empNumber and salary are initialized
        empNumber = 101;
        salary = 6789.50f;
```

© Aptech Ltd.

Variables and Operators/Session 3

23

Character and Boolean Types 4-4



```
// Prints the value of the variables on the console
System.out.println("Employee Number: " + empNumber);
System.out.println("Salary: " + salary);
System.out.println("Gender: " + gender);
System.out.println("Share Balance: " + shareBalance);
System.out.println("Owns vehicle: " + ownVehicle);
}
```

- Here, a `float` value needs to have the letter `f` appended at its end.
- Otherwise, by default, all the decimal values are treated as `double` in Java.
- The output of the code is shown in the following figure:

```
Output - Session 3 (run)
run:
Employee Number: 101
Salary: 6789.5
Gender: M
Share Balance: 456790.897
Owns vehicle: false
BUILD SUCCESSFUL (total time: 0 seconds)
```

© Aptech Ltd.

Variables and Operators/Session 3

24

Using slides 21 to 24, explain the character and boolean types.

`char` is used to store any character, while `boolean` has only two possible values which is `true` or `false`. `char` represents symbols in a character set such as letters and numbers. `char` stores 16-bit Unicode character and its value ranges from 0 ('`\u0000`') to 65,535 ('`\uffff`'').

Unicode is a 16-bit character set, which contains all the characters commonly used in information processing. It is an attempt to consolidate the alphabets of the world's various languages into a single and international character set.

Apart from primitive data types, Java programming language also supports strings. A string is a sequence of characters. Java does not provide any primitive data type for storing strings, instead provides a class `String` to create string variables. The `String` class is defined within the `java.lang` package in Java SE API. Though, `String` is a class in Java, however, you can use it as data type in the program. For example, `String str = "A String Data"`, it will remain constant and you cannot change the value of the variable within the program.

However, initializing string variable with new value creates a new `String` object. This behavior of strings makes them as immutable objects.

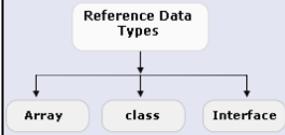
Slide 25

Let us understand reference data types.

Reference Data Types



- ◆ In Java, objects and arrays are referred to as reference variables.
- ◆ Reference data type is an address of an object or an array created in memory.
- ◆ Following figure shows the reference data types supported in Java:



```

graph TD
    A[Reference Data Types] --> B[Array]
    A --> C[class]
    A --> D[Interface]
  
```

- ◆ Following table lists and describes the three reference data types:

Data Type	Description
Array	It is a collection of several items of the same data type. For example, names of students in a class can be stored in an array.
Class	It is encapsulation of instance variables and instance methods.
Interface	It is a type of class in Java used to implement inheritance.

© Aptech Ltd

Variables and Operators/Session 3

25

Using slide 25, explain the reference data type in Java.

Tell them that, in Java, a reference data type is a variable that can contain the reference or an address of dynamically created object. These types of data type are not predefined such as primitive data type.

Explain it to them that a variable of a reference data type holds the reference to the actual value or a set of values represented by the variable. The reference data types are arrays, classes, and interfaces. They are described as follows:

Array

It is a collection of several items of the same data type. For example, names of students in a class can be stored in an array. It is used to store collection of data but here the collection of variable is of same size.

Class

It is encapsulation of instance variables and instance methods. It can be defined as a blue print that describes the behavior that an object of its type supports.

Interface

It is a type of class in Java used to implement inheritance. It is a sequence of one or more character.

Slides 26 to 29

Let us understand literals.

Literals 1-4

- ◆ A literal represents a fixed value assigned to a variable.
- ◆ It is represented directly in the code and does not require computation.
- ◆ Following figure shows some literals for primitive data types:

Integer	Float	Character	Boolean
50	35.7F	'C'	true

- ◆ A literal is used wherever a value of its type is allowed.
- ◆ However, there are several different types of literals as follows:

Integer Literals

- ◆ Integer literals are used to represent an int value, which in Java is a 32-bit integer value.
- ◆ Integers literals can be expressed as:
 - Decimal values have a base of 10 and consist of numbers from 0 through 9. For example, `int decNum = 56;`
 - Hexadecimal values have a base of 16 and consist of numbers 0 through 9 and letters A through F. For example, `int hexNum = 0X1c;`

© Aptech Ltd. Variables and Operators/Session 3 26

Literals 2-4

- Binary values have a base of 2 and consist of numbers 0 and 1. Java SE 7 supports binary literals. For example, `int binNum = 0b0010;`
- An integer literal can also be assigned to other integer types, such as byte or long.
- When a literal value is assigned to a byte or short variable, no error is generated, if the literal value is within the range of the target type.
- Integer numbers can be represented with an optional uppercase character ('L') or lowercase character ('l') at the end.
- This will inform the computer to treat that number as a long (64-bit) integer.

Floating-point Literals

- Floating-point literals represent decimal values with a fractional component.
- Floating-point literals have several parts:
 - Whole number component, for example 0, 1, 2,....., 9.
 - Decimal point, for example 4.90, 3.141, and so on.

© Aptech Ltd. Variables and Operators/Session 3 27

Literals 3-4



Exponent is indicated by an E or e followed by a decimal number, which can be positive or negative. For example, e+208, 7.436E6, 23763E-05, and so on.

Type suffix D, d, F, or f.

- ◆ Floating-point literals in Java default to double precision.
- ◆ A float literal is represented by F or f appended to the value, and a double literal is represented by D or d.

Boolean Literals

- ◆ Boolean literals are simple and have only two logical values - true and false.
- ◆ These values do not convert into any numerical representation.
- ◆ A true boolean literal in Java is not equal to one, nor does the false literal equals to zero.
- ◆ They can only be assigned to boolean variables or used in expressions with boolean operators.

© Aptech Ltd.

Variables and Operators/Session 3

28

Literals 4-4



Character Literals

- ◆ Character literals are enclosed in single quotes.
- ◆ All the visible ASCII characters can be directly enclosed within quotes, such as 'g', '\$', and 'z'.
- ◆ Single characters that cannot be enclosed within single quotes are used with escape sequence.

Null Literals

- ◆ When an object is created, a certain amount of memory is allocated for that object.
- ◆ The starting address of the allocated memory is stored in an object variable, that is, a reference variable.
- ◆ However, at times, it is not desirable for the reference variable to refer that object.
- ◆ In such a case, the reference variable is assigned the literal value null. For example, Car toyota = null;.

String Literals

- ◆ String literals consist of sequence of characters enclosed in double quotes. For example, "Welcome to Java", "Hello\nWorld".

© Aptech Ltd.

Variables and Operators/Session 3

29

Using slides 26 to 29, explain literals in Java.

Explain to the students the meaning of literal. Tell them that by literal we mean any number, text, or other information that represents a value. In other words, it means that what you type is what you get.

Explain to the students that a literal signifies a fixed value and is represented directly in the code without requiring computation. For instance,

```
int val = 50;
float num = 35.7f;
```

In the statement, the literal is an integer value that is **50**. The literal is **50** because it directly represents the integer value.

A literal is used wherever a value of its type is allowed.

Tips:

A *literal* is the source code representation of a fixed value. Literal does not require computation.

integer literals are used to represent an `int` value, which in Java is a 32-bit integer value.

Any whole number value is an integer literal. Integers can be expressed as:

- Decimal values, expressed in base 10
- Octal values, expressed in base 8
- Hexadecimal values, expressed in base 16

Mention to the students that values of the integral types `byte`, `short`, `int`, and `long` can be created from `int` literals. Explain to them that an integer literal is of type `long` if it ends with the letter `L` or `l`, otherwise it is of type `int`.

Tips:

Tell the students that as discussed integers can either be in decimal, hexadecimal, or octal format.

To indicate a **decimal format**, put the left most digit as nonzero. Similarly, put the characters as `ox` to the left of at least one hexadecimal digit to indicate **hexadecimal format**. Also, we can indicate the **octal format** by a zero digit followed by the digits 0 to 7.

Floating-point numbers are same as real numbers in mathematics.

Explain to them that floating-point literals represent decimal values with a fractional component.

Floating-point literals have several parts such as:

- Whole number component, for example, 0, 1, 2, ..., 9.
- Decimal point, for example, 4.90, 3.141, and so on.
- Exponent is indicated by an E or e followed by a decimal number, which can be positive or negative. For example, `e+208`, `7.436E6`, `23763E-05`, and so on.
- Type suffix D, d, F, or f.

Tell the students that Java has two kinds of floating-point numbers, `float` and `double`. The default type when you write a floating-point literal is `double`. A `float` literal is represented by `F` or `f` appended to the value, and a `double` literal is represented by `D` or `d`.

Tell the students that boolean literals are simple and have only two logical values, `true` and `false`. These values do not convert into any numerical representation. A true boolean literal in Java is neither equal to one nor does the false literal equals to zero. They can only be assigned to variables declared as `boolean`, or used in expressions with boolean operators.

Explain to the students about character literals. Tell them that character literal are represented as a single printable character in a pair of single quote characters such as '`a`', '`#`', and '`3`'. The ASCII character set includes 128 characters including letters, numerals, punctuations, and so on. There are few character literals which are not readily printable through a keyboard. Single characters that cannot be directly entered are hyphen (`-`) and backslash (`\`).

Tell them that one specifies the `Null` literal in the source code as '`null`'. To reduce the number of references to an object, one can use `null` literal.

Explain to them that when an object is created, a certain amount of memory is allocated for that object. The starting address of this memory is stored in an object, that is, a reference variable.

However, at times, it is not desirable for the reference variable to refer that object. In such a case, the reference variable is assigned the literal value `null` as shown:

```
obj = null;
```

In this statement, you reduce the number of references to an object by assigning `null` to `obj`. Now, as in this example, the object is no longer referenced so it will be available for the garbage collection that is the compiler will destroy it and the free memory will be allocated to the other object.

Explain to the students about string literals. Tell them that string literals consists of sequence of characters enclosed in double quotes. The characters can be printable or non-printable. However, backslash, double quote, and non-printable characters are represented using escape sequences.

Tips:

In Java, a string is not a basic data type, rather it is an object. These strings are not stored in arrays as in C language. Literals of types `char` and `String` may contain any Unicode (UTF-16) characters. Always use 'single quotes' for `char` literals and 'double quotes' for `String` literals.

In-Class Question:

After you finish explaining, the variables and literals in Java, you will ask the students the following In-Class question:



Can variable names begin with underscore character?

Answer:

Yes.

Slides 30 and 31

Let us understand underscore character in numeric literals.

Underscore Character in Numeric Literals 1-2



Java SE 7 allows you to add underscore characters (_) between the digits of a numeric literal.

The underscore character can be used only between the digits.

- ◆ In integral literals, underscore characters can be provided for telephone numbers, identification numbers, or part numbers, and so on.
- ◆ Similarly, for floating-point literals, underscores are used between large decimal values.
- ◆ Restrictions for using underscores in numeric literals are as follows:

A number cannot begin or end with an underscore.

In the floating-point literal, underscore cannot be placed adjacent to a decimal point.

Underscore cannot be placed before a suffix, L or F.

Underscore cannot be placed before or after the binary or hexadecimal identifiers, such as b or x.

© Aptech Ltd. Variables and Operators / Session 3 30

Underscore Character in Numeric Literals 2-2



- ◆ Following table shows the list of valid and invalid placement of underscore character:

Numeric Literal	Valid/Invalid
1234_9876_5012_5454L	Valid
_8976	Invalid, as underscore placed at the beginning
3.14_15F	Valid
0b11010000_11110000_00001111	Valid
3_.14_15F	Invalid, as underscore is adjacent to a decimal point
0x_78	Invalid, an underscore is placed after the hexadecimal

© Aptech Ltd. Variables and Operators / Session 3 31

Using slides 30 and 31, explain the use of underscore character in numeric literals.

Java SE 7 allows you to add underscore characters (_) between the digits of a numeric literal. This feature helps to separate groups of digit in numeric literals which can improve the readability of the code. The underscore character can be used only between the digits. In integral literals, underscore characters can be provided for telephone numbers, identification numbers, or part numbers, and so on. Similarly, for floating-point literals, underscores are used between large decimal values.

Restrictions for using underscores in numeric literals are as follows:

- A number cannot begin or end with an underscore.
- In the floating-point literal, underscore cannot be placed adjacent to a decimal point.
- Underscore cannot be placed before a suffix, L or F.
- Underscore cannot be placed before or after the binary or hexadecimal identifiers, such as b or x.

Slide 31 shows the list of valid and invalid placement of underscore character.

In-Class Question:

After you finish explaining the underscore character, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



Why underscore character is used in numeric literals?

Answer:

1. The underscore character can be used only between the digits.
2. It helps to separate groups of digit in numeric literals which can improve the readability of the code.

Slides 32 to 34

Let us understand escape sequences.

Escape Sequences 1-3

An escape sequence is a special sequence of characters that is used to represent characters, which cannot be entered directly into a string.

- ◆ For example, to include tab spaces or a new line character in a line or to include characters which otherwise have a different notation in a Java program (\ or "), escape sequences are used.

An escape sequence begins with a backslash character (\), which indicates that the character (s) that follows should be treated in a special way.

The output displayed by Java can be formatted with the help of escape sequence characters.

Following table lists escape sequence characters in Java:

Escape Sequence	Character Value
\\b	Backspace character
\\t	Horizontal Tab character
\\n	New line character
\\'	Single quote marks
\\\\	Backslash
\\r	Carriage Return character
\\"	Double quote marks
\\f	Form feed
\\xxx	Character corresponding to the octal value xxx, where xxx is between 000 and 0377
\\xxxxx	Unicode character with encoding xxxx, where xxxx is one to four hexadecimal digits. Unicode escapes are distinct from the other escape types

© Aptech Ltd. Variables and Operators/Session 3 32

Escape Sequences 2-3

- Following code snippet demonstrates the use of escape sequence characters:

```
public class EscapeSequence {
    /*
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // Uses tab and new line escape sequences
        System.out.println("Java \t Programming \n Language");
        // Prints Tom "Dick" Harry string
        System.out.println("Tom \"Dick\" Harry");
    }
}
```

- The output of the code is shown in the following figure:

Output - Session 3 (run) ■

run:
Java Programming
Language
Tom "Dick" Harry
BUILD SUCCESSFUL (total time: 0 seconds)

© Aptech Ltd.

Variables and Operators/Session 3

33

Escape Sequences 3-3

- To represent a Unicode character, \u escape sequence can be used in a Java program.
- A Unicode character can be represented using hexadecimal or octal sequences.
- Following code snippet demonstrates the Unicode characters in a Java program:

```
public class UnicodeSequence {
    /*
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // Prints 'Hello' using hexadecimal escape sequence characters
        System.out.println("\u0048\u0065\u006C\u006F" + "!\\n");
        // Prints 'Blake' using octal escape sequence character for 'a'
        System.out.println("Bl\u141ke\"2007\" ");
    }
}
```

- The output of the code is shown in the following figure:

Output - Session 3 (run) ■

run:
Hello!
Blake"2007"
BUILD SUCCESSFUL (total time: 0 seconds)

© Aptech Ltd.

Variables and Operators/Session 3

34

Using slides 32 to 34, explain the use of escape sequence.

Tell them that a character preceded by a backslash (\) is an *escape sequence* and has special meaning to the compiler.

Mention that an escape sequence is a special sequence of characters that is used to represent characters, which cannot be entered directly into a string. For example, to include tab spaces or a new line character or to include characters which otherwise have a different connotation in a Java program (such as \ or "), escape sequences are used.

An escape sequence begins with a backslash character (\), which indicates that the character (s) that follows should be treated in a special way. The output displayed by Java can be formatted with the help of escape sequence characters.

Tips:

The hexadecimal escape sequence starts with \u followed by 4 hexadecimal digits. The octal escape sequence comprises 3 digits after back slash.

For example,

\xxyy

where, x can be any digit from 0 to 3 and y can be any digit from 0 to 7.

Slides 35 to 39

Let us understand constants and enumerations.

Constants and Enumerations 1-5

- ◆ Constants in Java are fixed values assigned to identifiers that are not modified throughout the execution of the code.
- ◆ In Java, the declaration of constant variables is prefixed with the `final` keyword.
- ◆ The syntax to initialize a constant variable is as follows:

Syntax

```
final data-type variable-name = value;
```

where,

`final`: Is a keyword and denotes that the variable is declared as a constant.

- ◆ Following code snippet demonstrates the code that declares the constant variables:

```
public class AreaOfCircle {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // Declares constant variable
        final double PI = 3.14159;
        double radius = 5.87;
        double area;
    }
}
```

© Aptech Ltd.

Variables and Operators/Session 3

35

Constants and Enumerations 2-5

```
// Calculates the value for the area variable
area = PI * radius * radius;
System.out.println("Area of the circle is: " + area);
}
}
```

- ◆ In the code, a constant variable `PI` is assigned the value `3.14159`, which is a fixed value.
- ◆ The output of the code is shown in the following figure:

```
Output - Session 3 (run) %
run:
Area of the circle is: 108.24945247100001
BUILD SUCCESSFUL (total time: 0 seconds)
```

© Aptech Ltd.

Variables and Operators/Session 3

36

Constants and Enumerations 3-5



- Java SE 5.0 introduced enumerations.
- An enumeration is defined as a list that contains constants.
- Unlike C++, where enumeration was a list of named integer constants, in Java, enumeration is a class type.
- This means it can contain instance variables, methods, and constructors.
- The enumeration is created using the `enum` keyword.

- ◆ The syntax for declaring a method is as follows:

Syntax

```
enum enum-name {
    constant1, constant2, . . . , constantN
}
```

Constants and Enumerations 4-5



- ◆ Though, enumeration is a class in Java, you do not use `new` operator to instantiate it.
- ◆ Instead, declare a variable of type enumeration to use it in the Java program.
- ◆ This is similar to using primitive data types.
- ◆ The enumeration is mostly used with decision-making constructs, such as switch-case statement.
- ◆ Following code snippet demonstrates the declaration of enumeration in a Java program:

```
public class EnumDirection {

    /**
     * Declares an enumeration
     */
    enum Direction {
        East, West, North, South
    }

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
    }
}
```

Constants and Enumerations 5-5



```
// Declares a variable of type Direction
Direction direction;

// Instantiate the enum Direction
direction = Direction.East;

// Prints the value of enum
System.out.println("Value: " + direction);
}
```

- The output of the code is shown in the following figure:

The screenshot shows the Eclipse IDE's Output window titled "Output - Session 3 (run)". It displays the following text:
run:
Value: East
BUILD SUCCESSFUL (total time: 0 seconds)

© Aptech Ltd.

Variables and Operators/Session 3

39

Using slides 35 to 39, explain the use of constants and enumerations in Java.

Constants are a way to use meaningful names in place of a value that does not change. Constants in Java are fixed values assigned to identifiers that are not modified throughout the execution of the code. In Java, the declaration of constant variables is prefixed with the final keyword. Constant can be used to provide meaningful names to make code more readable.

Then explain the syntax to initialize a constant variable in the code.

Java SE 5.0 introduced enumerations. An enumeration is defined as a list that contains constants. The enumeration instance defined the methods by which user can enumerate the elements in a collection of objects. Unlike C++, where enumeration was a list of named integer constants, in Java, enumeration is a class type. This means it can contain instance variables, methods, and constructors. The enumeration is created using the enum keyword.

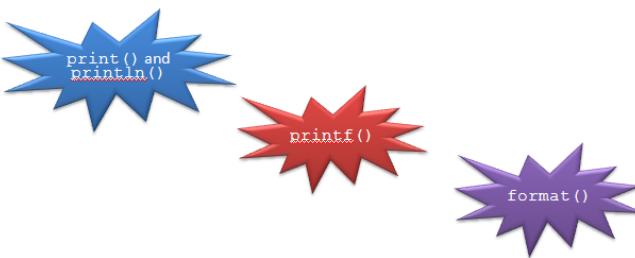
An enum type is a special data type that enables for a variable to be a set of predefined constants. Though, enumeration is a class in Java, you do not use new operator to instantiate it. Instead, declare a variable of type enumeration to use it in the Java program. This is similar to using primitive data types. The enumeration is mostly used with decision-making constructs, such as switch-case statement.

Explain the code to define enumeration in the Java program using slides 38 and 39.

Slide 40

Let us understand formatted output and input.

Formatted Output and Input



- Whenever an output is to be displayed on the screen, it needs to be formatted.
- Formatting can be done using three ways that are as follows:

- print() and println()
- printf()
- format()

- These methods behave in a similar manner.
- The format() method uses the `java.util.Formatter` class to do the formatting work.

© Aptech Ltd. Variables and Operators/Session 3 40

Using slide 40, explain formatted output and input methods used in Java.

Programming simple input/output is easy, which involves only few classes and methods. Whenever an output is to be displayed on the screen, it needs to be formatted. Formatting can be done using three ways that are as follows:

- print() and println()
- printf()
- format()

These methods behave in a similar manner. The format() method uses the `java.util.Formatter` class to do the formatting work. It formats multiple arguments based on a format string.

Slides 41 and 42

Let us understand `print()` and `println()` methods.

'print()' and 'println()' Methods 1-2

- ◆ These methods convert all the data to strings and display it as a single value.
- ◆ The methods uses the appropriate `toString()` method for conversion of the values.
- ◆ These methods can also be used to print mixture combination of strings and numeric values as strings on the standard output.
- ◆ Following code snippet demonstrates the use of `print()` and `println()` methods:

```
public class DisplaySum {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        " + sum + "); int num1 = 5;
        int num2 = 10;
        int sum = num1 + num2;
        System.out.print("The sum of ");
        System.out.print(num1);
        System.out.print(" and ");
        System.out.print(num2);
        System.out.print(" is ");
        System.out.print(sum);
    }
}
```

© Aptech Ltd. Variables and Operators/Session 3 41

'print()' and 'println()' Methods 2-2

```
System.out.println(".");
int num3 = 2;
sum = num1 + num2 + num3;
System.out.println("The sum of " + num1 + ", " + num2 + " and " +
num3 + " is
")
}
}


```

- ◆ The sum variable is formatted twice.
- ◆ In the first case, the `print()` method is used for each instruction which prints the result on the same line.
- ◆ In the second case, the `println()` method is used to convert each data type to string and concatenate them to display as a single result.
- ◆ The output of the code is shown in the following figure:

The screenshot shows the Java IDE's output window titled "Output - Session 3 (run)". It displays the following text:
run:
The sum of 5 and 10 is 15.
The sum of 5, 10 and 2 is 17.
BUILD SUCCESSFUL (total time: 0 seconds)

© Aptech Ltd. Variables and Operators/Session 3 42

Using slides 41 and 42, explain `print()` and `println()` methods.

The `println()` methods print the string and moves the cursor to a new line. It can also be used without parameters, to position the cursor on the next line. The `print()` method print the string, but does not move the cursor to the new line. These methods convert all the data to strings and display it as a single value. The methods uses the appropriate `toString()` method for conversion of the values. These methods can also be used to print mixture combination of strings and numeric values as strings on the standard output.

Explain the code snippet which demonstrates the use of `print()` and `println()` methods.

Slides 43 and 44

Let us understand `printf()` method.

'printf()' Method 1-2

- The `printf()` method introduced in J2SE 5.0 can be used to format the numerical output to the console.
- Following table lists some of the format specifiers in Java:

Format Specifier	Description
<code>%d</code>	Result formatted as a decimal integer
<code>%f</code>	Result formatted as a real number
<code>%o</code>	Results formatted as an octal number
<code>%e</code>	Result formatted as a decimal number in scientific notation
<code>%n</code>	Result is displayed in a new line

- Following code snippet demonstrates the use of `printf()` methods to format the output:

```

public class FormatSpecifier {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        int i = 55 / 22;
        // Decimal integer
        System.out.printf("55/22 = %d\n", i);
    }
}

```

43

'printf()' Method 2-2

```

// Pad with zeros
double q = 1.0 / 2.0;
System.out.printf("1.0/2.0 = %09.3f\n", q);
// Scientific notation
q = 5000.0 / 3.0;
System.out.printf("5000/3.0 = %7.2e\n", q);
// Negative infinity
q = -10.0 / 0.0;
System.out.printf("-10.0/0.0 = %7.2e\n", q);
// Multiple arguments, PI value, E-base of natural logarithm
System.out.printf("pi = %5.3f, e = %5.4f\n", Math.PI, Math.E);
}

```

- The output of the code is shown in the following figure:

44

Using slides 43 and 44, explain the `printf()` method.

Tell them that whenever an output is to be displayed on the screen, it needs to be formatted. The formatting can be done with the help of format specifiers in Java. Tell the students that the `printf()` method introduced in J2SE 5.0 can be used to format the numerical output to the console.

All format specifiers begin with a % and end with a 1- or 2-character conversion that specifies the kind of formatted output being generated.

The format specifiers are as follows:

- `d` formats an integer value as a decimal value.
- `f` formats a floating point value as a decimal value.
- `n` outputs a platform-specific line terminator.
- `o` outputs a value as an octal number.
- `e` outputs value as a decimal number in scientific notation.

Then, explain the code snippet for the `printf()` method. Tell the students that in the code snippet, '`%09.3f`' indicates that there will be total 9 digits including decimal point and three places of decimals. If the number of digits is less than 9, then it will be padded with zeroes. If '0' is omitted from '`%09.3f`', then it will be padded with spaces.

Slides 45 to 47

Let us understand `format()` method.

'format()' Method 1-3



- ◆ This method formats multiple arguments based on a format string.
- ◆ The format string consists of the normal string literal information associated with format specifiers and an argument list.
- ◆ The syntax of a format specifier is as follows:

Syntax

`*[arg_index$] [flags] [width] [.precision] conversion character`

where,

`arg_index$`: Is an integer followed by a \$ symbol. The integer indicates that the argument should be printed in the mentioned position.

`flags`: Is a set of characters that format the output result. There are different flags available in Java.

- ◆ Following table lists some of the flags available in Java:

Flag	Description
" <code>-</code> "	Left justify the argument
" <code>+</code> "	Include a sign (+ or -) with this argument
" <code>0</code> "	Pad this argument with zeros

© Aptech Ltd.

Variables and Operators / Session 3

45

©Aptech Limited

'format()' Method 2-3



Flag	Description
"'	Use locale-specific grouping separators
"("	Enclose negative numbers in parenthesis

width: Indicates the minimum number of characters to be printed and cannot be negative.

precision: Indicates the number of digits to be printed after a decimal point. Used with floating-point numbers.

conversion character: Specifies the type of argument to be formatted. For example, b for boolean, c for char, d for integer, f for floating-point, and s for string.

- ◆ The values within '[]' are optional.
- ◆ The only required elements of format specifier are the % and a conversion character.
- ◆ Following code snippet demonstrates the format() method:

```
public class VariableScope {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
```

© Aptech Ltd.

Variables and Operators/Session 3

46

'format()' Method 3-3



```
int num = 2;
double result = num * num;
System.out.format("The square root of %d is %f.%n", num, result);
}
```

- ◆ The output of the code is shown in the following figure:

The screenshot shows the Java IDE's output window titled "Output - Session 3 (run)". It displays the command "run:", the output "The square root of 2 is 4.000000.", and the message "BUILD SUCCESSFUL (total time: 0 seconds)".

© Aptech Ltd.

Variables and Operators/Session 3

47

Using slides 45 to 47, explain the format() method.

The format() method is used to format string in Java. This method formats multiple arguments based on a format string. It works in same way as printf(), but it does not print the string, it returns a formatted string. The format string consists of the normal string literal information associated with format specifiers and argument list.

Explain the syntax of a format specifier presented on slide 45. Then, explain the code snippet presented on slides 46 and 47 to demonstrates the format() method.

Slides 48 to 50

Let us understand formatted Input.

Formatted Input 1-3

- ◆ The Scanner class allows the user to read or accept values of various data types from the keyboard.
 - ◆ It breaks the input into tokens and translates individual tokens depending on their data type.
 - ◆ To use the Scanner class, pass the `InputStream` object to the constructor as follows:
- ```
Scanner input = new Scanner(System.in);
```
- ◆ Here, input is an object of Scanner class and `System.in` is an input stream object.
  - ◆ Following table lists the different methods of the Scanner class that can be used to accept numerical values from the user:

| Method                    | Description                              |
|---------------------------|------------------------------------------|
| <code>nextByte()</code>   | Returns the next token as a byte value   |
| <code>nextInt()</code>    | Returns the next token as an int value   |
| <code>nextLong()</code>   | Returns the next token as a long value   |
| <code>nextFloat()</code>  | Returns the next token as a float value  |
| <code>nextDouble()</code> | Returns the next token as a double value |

**Formatted Input 2-3**

- ◆ Following code snippet demonstrates the Scanner class methods and how they can be used to accept values from the user:

```
public class FormattedInput {
 /**
 * @param args the command line arguments
 */
 public static void main(String[] args) {

 // Creates an object and passes the InputStream object
 Scanner s = new Scanner(System.in);
 System.out.println("Enter a number:");
 // Accepts integer value from the user
 int intValue = s.nextInt();
 System.out.println("Enter a decimal number:");
 // Accepts integer value from the user
 float floatValue = s.nextFloat();
 System.out.println("Enter a String value");
 // Accepts String value from the user
 String strValue = s.next();
 }
}
```

### Formatted Input 3-3



```
System.out.println("Values entered are: ");
System.out.println(intValue + " " + floatValue + " " + strValue);
}
```

- The output of the code is shown in the following figure:

```
Output - Session 3 (run) #2
run:
Enter a number:
23456
Enter a decimal number:
9876.76545
Enter a String value
John
Values entered are:
23456 9876.766 John
BUILD SUCCESSFUL (total time: 19 seconds)
```

© Aptech Ltd.

Variables and Operators/Session 3

50

Using slides 48 to 50, explain the `Scanner()` class.

Tell the students that the `java.util.Scanner` class is a simple text scanner which can parse primitive types and strings using regular expressions.

Tell the students that the `Scanner` class allows the user to read values of various types. To use the `Scanner` class, pass the `InputStream` object to the constructor.

For example,

```
Scanner input = new Scanner(System.in);
```

Here, `input` is an object of `Scanner` class and `System.in` is an input stream object.

Explain to them the methods of the `Scanner` class. Tell them that `Scanner` breaks its input into tokens using a delimiter pattern, which is by default matches whitespace. The `nextByte()`, `nextInt()`, `nextLong()`, `nextFloat()`, and `nextDouble()` methods returns the next token as byte, int, long, float, and double value respectively.

Then, demonstrate the `Scanner` class methods and how they can be used to accept values from the user and finally, print the output.

#### Tips:

- A scanning operation may block resources waiting for input and a `Scanner` is not safe for multithreaded, use without external synchronization.
- A token is a series of characters that ends with delimiters. A delimiter can be a whitespace (default delimiter for tokens in `Scanner` class), a tab character, a carriage return, or the end of the file. Thus, if we read a line that has a series of numbers separated by whitespaces, the scanner will take each number as a separate token.

**Slide 51**

Let us understand operators.

**Operators**

Operators are set of symbols used to indicate the kind of operation to be performed on data.

- Consider the expression:  $Z = X + Y$ ;
- Here,
  - $+$  is called the Operator and the operation performed is addition.
  - $X$  and  $Y$ , the two variables, on which addition is performed, are called as Operands.
  - $Z = X + Y$  a combination of both the operator and the operands, is known as an Expression.
- Java provides several categories of operators and they are as follows:

Categories of Operators:

- Assignment Operator
- Arithmetic Operator
- Unary Operator
- Conditional Operator
- Logical Operator
- Assignment Operator
- Bitwise Operator

© Aptech Ltd. Variables and Operators/Session 3 51

Using slide 51, explain the operators in Java.

Tell them that operators are special symbols that perform specific operations on one, two, or three *operands*, and then return a result.

Tell the students that all programming languages provide some mechanism for performing various operations on the data stored in variables. Also mention that the simplest form of operations involves arithmetic (such as adding, dividing, or comparing between two or more variables). A set of symbols is used to indicate the kind of operation to be performed on data. These symbols are called **operators**.

For instance,

$$Z = X + Y$$

The ' $+$ ' symbol in the statement is called the **operator** and the operation performed is **addition**. This operation is performed on the two variables  $X$  and  $Y$ , which are called **operands**. The combination of both the operator and the operands,  $Z = X + Y$ , is known as an **Expression**.

Mention to the students about the different categories of operators such as:

- Arithmetic
- Relational
- Logical
- Assignment
- Bitwise
- Bit Shift

**Tips:**

In general-purpose programming, certain operators tend to appear more frequently than others. For example, the assignment operator ( $=$ ) is more common than the unsigned right shift operator ( $>>>$ ).

## Slides 52 and 53

Let us understand assignment operators.

### Assignment Operators 1-2



- The basic assignment operator is a single equal to sign, '='.
- This operator is used to assign the value on its right to the operand on its left.
- Assigning values to more than one variable can be done at a time.
- In other words, it allows you to create a chain of assignments.

◆ Consider the following statements:

```
int balance = 3456;
char gender = 'M';
```

◆ The value **3456** and '**M**' are assigned to the variables, **balance** and **gender**.

© Aptech Ltd. Variables and Operators/Session 3 52

### Assignment Operators 2-2



- ◆ In addition to the basic assignment operator, there are combined operators that allow you to use a value in an expression, and then, set its value to the result of that expression.

```
X = 3;
X += 5;
```

- ◆ The second statement stores the value **8**, the meaning of the statement is that **X = X + 5**.
- ◆ Following code snippet demonstrates the use of assignment operators:

```
...
x = 10; // Assigns the value 10 to variable x
x += 5; // Increments the value of x by 5
x -= 5; // Decrements the value of x by 5
x *= 5; // Multiplies the value of x by 5
x /= 2; // Divides the value of x by 2
x %= 2; // Divides the value of x by 2 and the remainder is returned
```

© Aptech Ltd. Variables and Operators/Session 3 53

Using slides 52 and 53, explain the assignment operator.

Tell the students that one of the most common operators is the assignment operator (=).

Explain that it is the simple assignment operator (=) and is used for assigning the value on its right to the operand on its left. Assigning values to more than one variable can be done at a time. In other words, it allows you to create a chain of assignments.

For instance,

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

In the statement, it assigns value 5 to a and b integer variables.

Tell the students that, in addition to the basic assignment operator, there are combined operators that allow you to use a value in an expression, and then set its value to the result of that expression.

For instance,

```
X = 3;
X+=5;
```

The second statement stores the value 8 in the variable X, the meaning of the statement is that,  $X = X + 5$ .

Explain to the students about the different types of combined assignment operators such as  $+=$ ,  $-=$ ,  $*=$ ,  $/=$ , and  $\% =$  as demonstrated in the code snippet.

#### Tips:

Simple assignment operator (=) can also be used on objects to assign *object references*.

#### Slide 54

Let us understand arithmetic operators.

### Arithmetic Operators



- ◆ Arithmetic operators manipulate numeric data and perform common arithmetic operations on the data.
- ◆ Operands of the arithmetic operators must be of numeric type.
- ◆ Boolean operands cannot be used, but character operands are allowed.
- ◆ The operators mentioned here are binary in nature that is, these operate on two operands, such as **X+Y**. Here, + is a binary operator operating on **X** and **Y**.
- ◆ Following table lists the arithmetic operators:

| Operator | Description                                                 |
|----------|-------------------------------------------------------------|
| +        | Addition - Returns the sum of the operands                  |
| -        | Subtraction - Returns the difference of two operands        |
| *        | Multiplication - Returns the product of operands            |
| /        | Division – Returns the result of division operation         |
| %        | Remainder - Returns the remainder from a division operation |

- ◆ Following code snippet demonstrates the use of arithmetic operators:

```

...
x = 2 + 3; // Returns 5
y = 8 - 5; // Returns 3
x = 5 * 2; // Returns 10
x = 5/2; // Returns 2
y = 10 % 3; // Returns 1
...

```

© Aptech Ltd.

Variables and Operators/Session 3

54

Using slide 54, explain the arithmetic operators in Java.

Tell them that arithmetic operators are used in mathematical expressions in the same way that they are used in algebra.

Tell them that arithmetic operators manipulate numeric data and perform common arithmetic operations on the data. Operands of the arithmetic operators must be of numeric type. Mention that boolean operands cannot be used, but character operands are allowed. The operators

mentioned here are binary in nature, that is, these operate on two operands, such as  $X+Y$ . Here,  $+$  is a binary operator operating on  $X$  and  $Y$ .

Explain the different arithmetic operators such as  $+$ ,  $-$ ,  $*$ ,  $/$ , and  $\%$  which perform addition, subtraction, multiplication, division, and modulus operations respectively.

Then, explain the code snippet showing the use of arithmetic operators.

#### Tips:

The modulus operator returns the remainder after performing the division.

#### Slide 55

Let us understand unary operator.

### Unary Operator



- ◆ Unary operators require only one operand.
- ◆ They increment/decrement the value of a variable by 1, negate an expression, or invert the value of a boolean variable. Following table lists the unary operators:

| Operator        | Description                                                  |
|-----------------|--------------------------------------------------------------|
| <code>+</code>  | Unary plus - Indicates a positive value                      |
| <code>-</code>  | Unary minus - Negates an expression                          |
| <code>++</code> | Increment operator - Increments the value of a variable by 1 |
| <code>--</code> | Decrement operator - Decrements the value of a variable by 1 |
| <code>!</code>  | Logical complement operator - Inverts a boolean value        |

- ◆ The prefix version (`++variable`) will increment the value before evaluating.
- ◆ The postfix version (`variable++`) will first evaluate and then, increment the original value.
- ◆ Following code snippet demonstrates the use of unary operators:

```

...
int i = 5;
int j = i++; // i=6, j=5
int k = ++i; //i=6,k=6
i = -i; //now i is -6
boolean result = false; //result is false
result = !result; //now result is true
...

```

© Aptech Ltd. Variables and Operators/Session 3

55

Using slide 55, explain the unary operator.

Tell the students that the unary operators require only one operand, they perform various operations such as incrementing/decrementing a value by one, negating an expression, or inverting the value of a boolean.

The various type of unary operators are `+`, `-`, `++`, `--`, and `!` represents unary plus, unary minus, increment operator, decrement operator, and logical complement operator respectively.

Tell them that the `++` and `--` operators can be applied before (prefix) or after (postfix) the operand. The statements `++variable` and `variable++` both result in incrementing the variable value by 1.

Mention that the only difference is that the prefix version (`++variable`) will increment the value before evaluating, whereas the postfix version (`variable++`) will first evaluate and then, increment the original value.

Explain to the student the code snippet demonstrating the use of prefix and postfix increment and decrement operator.

**Tips:**

Tell the students that, if you are just performing a simple increment/decrement, it doesn't really matter which version (prefix or postfix) you choose. However, if you use this operator in part of a larger expression, the one that you choose may make a significant difference.

**Slides 56 and 57**

Let us understand conditional operators.

### Conditional Operators 1-2

- ◆ The conditional operators test the relationship between two operands.
- ◆ An expression involving conditional operators always evaluates to a boolean value (that is, either true or false).
- ◆ Following table lists the various conditional operators:

| Operator           | Description                                                                                                   |
|--------------------|---------------------------------------------------------------------------------------------------------------|
| <code>==</code>    | Equal to - Checks for equality of two numbers                                                                 |
| <code>!=</code>    | Not Equal to - Checks for inequality of two values                                                            |
| <code>&gt;</code>  | Greater than - Checks if value on left is greater than the value on the right                                 |
| <code>&lt;</code>  | Less than - Checks if the value on the left is lesser than the value on the right                             |
| <code>&gt;=</code> | Greater than or equal to - Checks if the value on the left is greater than or equal to the value on the right |
| <code>&lt;=</code> | Less than or equal to - Checks if the value on the left is less than or equal to the value on the left        |

- ◆ Following code snippet demonstrates the use of conditional operators:

```
public class TestConditionalOperators {
 /**
 * @param args the command line arguments
 */
 public static void main(String[] args) {
 int value1 = 10;
 int value2 = 20;
 }
}
```

© Aptech Ltd. Variables and Operators/Session3 56

### Conditional Operators 2-2

```
// Use of conditional operators
System.out.print("value1 == value2: ");
System.out.println(value1 == value2);
System.out.print("value1 != value2: ");
System.out.println(value1 != value2);
System.out.print("value1 > value2: ");
System.out.println(value1 > value2);
System.out.print("value1 < value2: ");
System.out.println(value1 < value2);
System.out.print("value1 <= value2: ");
System.out.println(value1 <= value2);
System.out.print("value1 >= value2: ");
System.out.println(value1 >= value2);
```

- ◆ The output of the code is shown in the following figure:

© Aptech Ltd. Variables and Operators/Session3 57

Using 56 and 57, explain the conditional operator.

The conditional operators test the relationship between two operands. An expression involving conditional operators always evaluates to a boolean value (that is, either true or false). Here, the expression is evaluated so that truth and falsehood can be determined.

Explain the various relational operators such as `==`, `!=`, `>`, `<`, `>=`, and `<=`.

**Tips:**

Remember that you must use '==' , not '=' , when testing, if two primitive values are equal.

**Slides 58 and 59**

Let us understand logical operators.

**Logical Operators 1-2**

- ♦ Logical operators (`&&` and `||`) work on two boolean expressions.
- ♦ These operators exhibit short-circuit behavior, which means that the second operand is evaluated only if required.
- ♦ Following table lists the two logical operators:

| Operator                | Description                                                                                        |
|-------------------------|----------------------------------------------------------------------------------------------------|
| <code>&amp;&amp;</code> | Conditional AND - Returns true only if both the expressions are true                               |
| <code>  </code>         | Conditional OR - Returns true if either of the expression is true or both the expressions are true |

- ♦ Following code snippet demonstrates the use of logical operators:

```
public class TestLogicalOperators {
 /**
 * @param args the command line arguments
 */
 public static void main(String[] args) {
 int first = 10;
 int second = 20;

 // Use of logical operator
 System.out.println((first == 30) && (second == 20));
 }
}
```

© Aptech Ltd.

Variables and Operators/Session 3

58

**Logical Operators 2-2**

```
System.out.println((first == 30) || (second == 20));
}
}
```

- ♦ The output of the code is shown in the following figure:

The screenshot shows the Java Output window with the title "Output - Session 3 (run)". It displays the following text:  
run:  
false  
true  
BUILD SUCCESSFUL (total time: 0 seconds)

© Aptech Ltd.

Variables and Operators/Session 3

59

Using slides 58 and 59, explain the logical operators.

Logical operators (`&&` and `||`) work on two boolean expressions. These operators exhibit short-circuit behavior, which means that the second operand is evaluated only if required.

Explain to them that the operator `&&` will return true, only if both expressions are true. Similarly, the conditional operator `||` will return true, if either one expression is true or both the expressions are true.

### Slides 60 and 61

Let us understand bitwise operators.

### Bitwise Operators 1-2

- Bitwise operators work on binary representations of data.
- These operators are used to change individual bits in an operand.
- Following table lists the various bitwise operators:

| Operator              | Description                                                                                                           |
|-----------------------|-----------------------------------------------------------------------------------------------------------------------|
| <code>&amp;</code>    | Bitwise AND - compares two bits and generates a result of 1 if both bits are 1; otherwise, it returns 0               |
| <code> </code>        | Bitwise OR - compares two bits and generates a result of 1 if the bits are complementary; otherwise, it returns 0     |
| <code>^</code>        | Exclusive OR - compares two bits and generates a result of 1 if either or both bits are 1; otherwise, it returns 0    |
| <code>~</code>        | Complement operator - used to invert all of the bits of the operand                                                   |
| <code>&gt;&gt;</code> | Shift Right operator - Moves all bits in a number to the right by one position retaining the sign of negative numbers |
| <code>&lt;&lt;</code> | Shift Left operator - Moves all the bits in a number to the left by the specified position                            |

- Following code snippet demonstrates the use of bitwise operators:

```
public class TestBitwiseOperators {
 /**
 * @param args the command line arguments
 */
 public static void main(String[] args) {
 int x = 23;
 int y = 12;
 //23 = 10111 , 12 = 01100
 }
}
```

Variables and Operators/Session 3

50

### Bitwise Operators 2-2

```
System.out.print("x & y: ");
System.out.println(x & y); // Returns 4 , i.e, 4 = 00100
System.out.print("x | y: ");
System.out.println(x | y); // Returns 31, i.e 31 = 11111
System.out.print("x ^ y: ");
System.out.println(x ^ y); // Returns 27, i.e 31 = 11011
int a = 43;
int b = 1;
System.out.print("a >> b: ");
System.out.println(a >> b); // returns 21 , i.e, 21 = 0010101
System.out.print("a << b: ");
System.out.println(a << b); //returns 86 , i.e, 86 = 1010110
}
```

- The output of the code is shown in the following figure:

The screenshot shows the Java IDE's Output window titled "Output - Session 3 (run)". It displays the following text:  
run:  
x & y: 4  
x | y: 31  
x ^ y: 27  
a >> b: 21  
a << b: 86  
BUILD SUCCESSFUL (total time: 0 seconds)

Variables and Operators/Session 3

51

Using slides 60 and 61, explain the bitwise operators.

Tell the students that the Java programming language also provides operators that perform bitwise and bit shift operations on integral types.

Tell them that the bitwise operators work on binary representations of data. These operators are used to change individual bits in an operand. Bitwise shift operators are used to move all the bits in the operand left or right, a given number of times.

Mention to the students that the unary bitwise complement operator (`~`) inverts a bit pattern. In other words, it can be applied to any of the integral types, converting every '0' to '1' and every '1' to '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'.

Also tell them about the signed left shift operator (`<<`). This 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 be shifted is provided 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.

The bitwise `&` operator performs a bitwise AND operation. It converts two bits and generates a result of 1 if both bits are 1, otherwise, it returns 0.

The bitwise `^` operator performs a bitwise exclusive OR operation. It compares two bits and generates a result of 1 if either or both bits are 1, otherwise, it returns 0.

The bitwise `|` operator performs a bitwise inclusive OR operation. It compares two bits and generates a result of 1 if the bits are complementary, otherwise, it returns 0.

Then explain the code snippet to explain the concept of bitwise and bit shift operators. Explain the output as specified in comments.

**In-Class Question:**

After you finish explaining Bitwise operators, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



What are bitwise operators?

**Answer:**

These operators are used to change individual bits in an operand. It provides a way to manipulate individual bits into integral primitive data type.

## Slides 62 and 63

Let us understand ternary operator.

### Ternary Operator 1-2



- The ternary operator (`? :`) is a shorthand operator for an if-else statement.
- It makes your code compact and more readable.
- The syntax to use the ternary operator is as follows:

#### Syntax

```
expression1 ? expression2 : expression3
```

where,

`expression1`: Represents an expression that evaluates to a `boolean` value of true or false.

`expression2`: Is executed if `expression1` evaluates to true.

`expression3`: Is executed if `expression1` evaluates to false.

- Following code snippet demonstrates the use of ternary operator:

```
public class VariableScope {
 /**
 * @param args the command line arguments
 */
 public static void main(String[] args) {
 int value1 = 10;
```

© Aptech Ltd.

Variables and Operators/Session 3

62

### Ternary Operator 2-2



```
int value2 = 20;
int result;
boolean someCondition = true;
result = someCondition ? value1 : value2;
System.out.println(result);
}
```

- As `someCondition` variable evaluates to `true`, the value of `value1` variable is assigned to the `result` variable.
- Thus, the program prints `10` on the console.

© Aptech Ltd.

Variables and Operators/Session 3

63

Using 62 and 63, explain the ternary operator.

The ternary operator (`? :`) is a shorthand operator for an if-else statement. It is used to control the flow of execution in Java. It works with three operands producing a value depending on the truth or falsehood of a Boolean assertion. It makes your code compact and more readable.

Then, explain the syntax for ternary operator and the corresponding code snippet.

**Slides 64 and 65**

Let us understand operator precedence.

**Operator Precedence 1-2**

- ◆ Expressions that are written generally consist of several operators.
- ◆ The rules of precedence decide the order in which each operator is evaluated in any given expression.
- ◆ Following table lists the order of precedence of operators from highest to lowest in which operators are evaluated in Java:

| Order | Operator                                                          |
|-------|-------------------------------------------------------------------|
| 1.    | Parentheses like ()                                               |
| 2.    | Unary Operators like +, -, ++, --, ~, !                           |
| 3.    | Arithmetic and Bitwise Shift operators like *, /, %, +, -, >>, << |
| 4.    | Relational Operators like >, >=, <, <=, ==, !=                    |
| 5.    | Conditional and Bitwise Operators like &, ^,  , &&,   ,           |
| 6.    | Conditional and Assignment Operators like ?:, =, *=, /=, +=, -=   |

- ◆ Parentheses are used to change the order in which an expression is evaluated.
- ◆ Any part of an expression enclosed in parentheses is evaluated first.
- ◆ For example, consider the following expression:

`2*3+4/2 > 3 && 3<5 || 10<9`

**Operator Precedence 2-2**

- ◆ The evaluation of the expression based on its operators precedence is as follows:

- 1 •  $(2*3+4/2) > 3 \&\& 3<5 || 10<9$   
• First the arithmetic operators are evaluated.
- 2 •  $((2*3)+(4/2)) > 3 \&\& 3<5 || 10<9$   
• Division and Multiplication are evaluated before addition and subtraction.
- 3 •  $(6+2) > 3 \&\& 3<5 || 10<9$
- 4 •  $(8>3) \&\& [3<5] || [10<9]$   
• Next to be evaluated are the relational operators all of which have the same precedence.
- 5 • These are therefore evaluated from left to right.  
•  $(\text{True} \&\& \text{True}) || \text{False}$
- 6 • The last to be evaluated are the logical operators. && takes precedence over || .  
•  $\text{True} || \text{False}$
- 7 •  $\text{True}$

Using 64 and 65, explain the ternary operator.

Tell the students that operators with higher precedence are evaluated before operators with relatively lower precedence. Also tell them that when operators of equal precedence appear in the same expression, a rule must govern which is evaluated first.

Mention that expressions that are written generally consist of several operators. The rules of precedence decide the order in which each operator is evaluated in any given expression.

Explain to them using the following table:

| Operators            | Precedence                             |
|----------------------|----------------------------------------|
| parenthesis          | ()                                     |
| postfix              | expr++ expr--                          |
| unary                | ++expr --expr +expr -expr ~ !          |
| multiplicative       | * / %                                  |
| additive             | + -                                    |
| shift                | << >> >>>                              |
| relational           | < > <= >= instanceof                   |
| equality             | == !=                                  |
| bitwise AND          | &                                      |
| bitwise exclusive OR | ^                                      |
| bitwise inclusive OR |                                        |
| logical AND          | &&                                     |
| logical OR           |                                        |
| ternary              | ? :                                    |
| assignment           | = += -= *= /= %= &= ^=  = <<= >>= >>>= |

Tell the students that the parentheses are used to change the order, in which an expression is evaluated. Any part of an expression enclosed in parentheses is evaluated first.

Explain to the students, the concept of operator precedence using the code snippet. Tell them that the evaluation will be as shown:

1. **(2\*3+4/2) > 3 && 3<5 || 10<9**
2. First, the arithmetic operators are evaluated.
3. **((2\*3)+(4/2)) > 3 && 3<5 || 10<9**
4. Division and Multiplication are evaluated before addition and subtraction.
5. **(6+2) >3 && 3<5 || 10<9**
6. **(8>3) && [3<5] || [10<9]**
7. Next to be evaluated are the relational operators, all of which have the same precedence. These are therefore, evaluated from left to right.
8. **(True && True) || False**
9. The last to be evaluated are the logical operators. && takes precedence over ||.
10. True || False
11. True

#### Tips:

When operators of same precedence appear in an expression, then a rule is followed. All binary operators except for the assignment operators are evaluated from left to right, assignment operators are evaluated right to left.

## Slides 66 and 67

Let us understand operator associativity.

### Operator Associativity 1-2



- When two operators with the same precedence appear in an expression, the expression is evaluated, according to its associativity.
- For example, in Java the `-` operator has left-associativity and `x - y - z` is interpreted as `(x - y) - z`, and `=` has right-associativity and `x = y = z` is interpreted as `x = (y = z)`.
- Following table shows the Java operators and their associativity:

| Operator                        | Description                                                                   | Associativity                        |
|---------------------------------|-------------------------------------------------------------------------------|--------------------------------------|
| <code>( ), ++, --</code>        | Parentheses, post increment/decrement                                         | Left to right                        |
| <code>++, --, +, -, !, ~</code> | Pre increment/decrement unary plus, unary minus, logical NOT, and bitwise NOT | unary minus logical NOT, bitwise NOT |
| Right to left                   | <code>*, /, %, +, -</code>                                                    | Multiplicative and Additive          |
| Left to right                   | <code>&lt;&lt;, &gt;&gt;</code>                                               | Bitwise shift                        |
| Left to right                   | <code>&lt;, &lt;, &gt;=, &lt;=, ==, !=</code>                                 | Relational and Equality operators    |
| Left to right                   | <code>&amp;, ^,  </code>                                                      | Bitwise AND, XOR, OR                 |
| Left to right                   | <code>&amp;&amp;,   </code>                                                   | Conditional AND, OR                  |
| Left to right                   | <code>?:</code>                                                               | Conditional operator (Ternary)       |

### Operator Associativity 2-2



- Consider the following expression:

$$2+10+4-5*(7-1)$$

1

- The `*` has higher precedence than any other operator in the equation.
- However, as `7-1` is enclosed in parenthesis, it is evaluated first.
- $2+10+4-5*6$

2

- Next, `*` is the operator with the highest precedence.
- Since there are no more parentheses, it is evaluated according to the rules.
- $2+10+4-30$

3

- As `+` and `-` have the same precedence, the left associativity works out.
- $12+4-30$

4

- Finally, the expression is evaluated from left to right.
- $6 - 30$
- The result is  $-14$ .

Using slides 66 and 67, explain the operator associativity.

Tell them that when two operators with the same precedence appear in an expression, the expression is evaluated, according to its associativity. For example, in Java, the `-` operator has left-associativity and `x - y - z` is interpreted as `(x - y) - z`, and `=` has right-associativity and `x = y = z` is interpreted as `x = (y = z)`.

Explain to the students about the associativity using the following table.

| Operator                                                                       | Description                                                                                         | Level | Associativity |
|--------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------|-------|---------------|
| [ ]<br>.()<br>++<br>--                                                         | access array element<br>access object member<br>invoke a method<br>post-increment<br>post-decrement | 1     | left to right |
| ++<br>--<br>+<br>-<br>!<br>~                                                   | pre-increment<br>pre-decrement<br>unary plus<br>unary minus<br>logical NOT<br>bitwise NOT           | 2     | right to left |
| ()<br>new                                                                      | cast<br>object creation                                                                             | 3     | right to left |
| *                                                                              | multiplicative                                                                                      | 4     | left to right |
| /                                                                              |                                                                                                     |       |               |
| %                                                                              |                                                                                                     |       |               |
| + -<br>+                                                                       | additive<br>string concatenation                                                                    | 5     | left to right |
| << >><br>>>>                                                                   | shift                                                                                               | 6     | left to right |
| < <=                                                                           | relational                                                                                          | 7     | left to right |
| > >=                                                                           | type comparison                                                                                     |       |               |
| instanceof                                                                     |                                                                                                     |       |               |
| ==                                                                             | equality                                                                                            | 8     | left to right |
| !=                                                                             |                                                                                                     |       |               |
| &                                                                              | bitwise AND                                                                                         | 9     | left to right |
| ^                                                                              | bitwise XOR                                                                                         | 10    | left to right |
|                                                                                | bitwise OR                                                                                          | 11    | left to right |
| &&                                                                             | conditional AND                                                                                     | 12    | left to right |
|                                                                                | conditional OR                                                                                      | 13    | left to right |
| ? :                                                                            | conditional                                                                                         | 14    | right to left |
| =      +=<br>-=<br>*=      /=<br>%=<br>&=      ^=<br> =<br><<=      >>=<br>>>= | assignment                                                                                          | 15    | right to left |

Explain to the students about this concept using the expression as shown:

$$2+10+4-5*(7-1)$$

- According to the rules of operator precedence, the '\*' has higher precedence than any other operator in the equation. Since  $7-1$  is enclosed in parenthesis, it is evaluated first.

$$2+10+4-5*6$$

2. Next, '\*' is the operator with the highest precedence. Since there are no more parentheses, it is evaluated according to the rules.

$2+10*4-30$

3. As '+' and '-' have the same precedence, the left associativity works out.

$12+4-30$

4. Finally, the expression is evaluated from left to right.

$16 - 30$

Finally, the result is -14.

**In-Class Question:**

After you finish explaining the operators, operator precedence, and operator associativity, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



Which operator is used to compare two values, = or ==, ?

**Answer:**

The == operator is used for comparison and = is used for assignment.



Which operator would you use to invert the value of a boolean?

**Answer:**

The == logical complement operator '!' .

**Slide 68**

Let us understand casting.

## Type Casting

Type conversion or typecasting refers to changing an entity of one data type into another.

- ◆ For instance, values from a more limited set, such as integers, can be stored in a more compact format.
- ◆ There are two types of conversion:
  - implicit ➤ The term for implicit type conversion is coercion.
  - explicit ➤ The most common form of explicit type conversion is known as casting.  
➤ Explicit type conversion can also be achieved with separately defined conversion routines such as an overloaded object constructor.

© Aptech Ltd. Variables and Operators/Session 3 68

Using slide 68, explain the concept of casting.

Tell them that in any application, there may be situations, where one data type may need to be converted into another data type. The type casting feature in Java helps in such conversion. Type conversion or typecasting refers to changing an entity of one data type into another. This is done to take advantage of certain features of type hierarchies. For instance, values from a more limited set, such as integers, can be stored in a more compact format. It can be converted later, to a different format enabling operations not previously possible, such as division with several decimal places worth of accuracy. In object-oriented programming languages, type conversion allows programs to treat objects of one type as one of their ancestor types to simplify interacting with them.

Mention that if a conversion results in the loss of precision, as in an `int` value converted to a `short`, then the compiler issues an error message unless an explicit cast is made.

Tell them that the two types of casting are as follows:

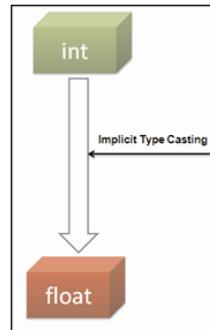
- Implicit casting
- Explicit casting

**Slides 69 to 71**

Let us understand implicit type casting.

### Implicit Type Casting 1-3

- ◆ When a data of a particular type is assigned to a variable of another type, then automatic type conversion takes place.
- ◆ It is also referred to as implicit type casting, provided it meets the conditions specified:
  - The two types should be compatible
  - The destination type should be larger than the source
- ◆ Following figure shows the implicit type casting:



© Aptech Ltd.

Variables and Operators/Session 3

69

### Implicit Type Casting 2-3

- ◆ The primitive numeric data types that can be implicitly cast are as follows:

- byte (8 bits) to short, int, long, float, double
- short (16 bits) to int, long, float, double
- int (32 bits) to long, float, double
- long (64 bits) to float, double

- ◆ This is also known as the type promotion rule.
- ◆ The type promotion rules are listed as follows:

- All byte and short values are promoted to int type.
- If one operand is long, the whole expression is promoted to long.
- If one operand is float then, the whole expression is promoted to float.
- If one operand is double then, the whole expression is promoted to double.

© Aptech Ltd.

Variables and Operators/Session 3

70

*FOR INTERNAL USE ONLY*

### Implicit Type Casting 3-3



- Following code snippet demonstrates implicit type conversion:

```

...
double dbl = 10;
long lng = 100;
int in = 10;
dbl = in; // assigns the integer value to double variable
lng = in; // assigns the integer value to long variable

```

Using slides 69 to 71, explain the implicit casting.

Tell them that when one type of data is assigned to a variable of another type, then automatic type conversion takes place, also referred to as implicit type casting, provided it meets the conditions specified:

- The two types should be compatible.
- The destination type should be larger than the source.

Explain to them that the primitive numeric data types that can be implicitly cast are as follows:

- byte (8-bits) to short, int, long, float, double
- short(16-bits) to int, long, float, double
- int (32-bits) to long, float, double
- long(64-bits) to float, double

This is also known as the type promotion rule.

#### Tips:

Type promotion rules are as follows:

- All byte and short values are promoted to int type.
- If one operand is long, then the whole expression is promoted to long.
- If one operand is float, then the whole expression is promoted to float.
- If one operand is double, then the whole expression is promoted to double.

**Slides 72 and 73**

Let us understand explicit casting.

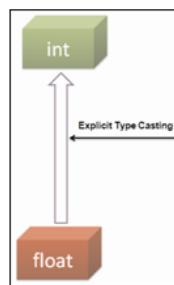
**Explicit Casting 1-2**

- ◆ A data type with lower precision, such as `short`, can be converted to a type of higher precision, such as `int`, without using explicit casting.
- ◆ However, to convert a higher precision data type to a lower precision data type, such as `float` to `int` data type, an explicit cast is required.
- ◆ The syntax for explicit casting is as follows:

**Syntax**

```
(target data type) value;
```

- ◆ Following figure shows the explicit type casting of data types:



© Aptech Ltd.

Variables and Operators/Session 3

72

**Explicit Casting 2-2**

- ◆ Following code snippet adds a `float` value to an `int` and stores the result as an integer:

```
...
float a = 21.3476f;
int b = (int) a + 5;
...
```

- ◆ The `float` value in `a` is converted into an integer value `21`.
- ◆ It is then, added to `5`, and the resulting value, `26`, is stored in `b`.
- ◆ This type of conversion is known as truncation.
- ◆ The fractional component is lost when a floating-point is assigned to an integer type, resulting in the loss of precision.

© Aptech Ltd.

Variables and Operators/Session 3

73

Using slides 72 and 73, explain the explicit type conversion.

Explain to the students that a data type with lower precision, such as `short`, can be converted to a type of higher precision, such as `int`, without using explicit casting. However, to convert a higher precision data type to a lower precision data type, such as `float` to `int` data type, an explicit cast is required. Otherwise, the compiler will display an error message.

Explain this concept with the help of the code snippet. Tell them that the float value in a variable is converted into an integer value **21**. It is then added to **5**, and the resulting value, **26**, is stored in **b**. This type of conversion is known as **truncation**. The fractional component is lost, when a floating point is assigned to an integer type, resulting in the loss of precision.

### Slides 74

Let us summarize the session.

### Summary



- ◆ Variables store values required in the program and should be declared before they are used. In Java, variables can be declared within a class, method, or within any block.
- ◆ Data types determine the type of values that can be stored in a variable and the operations that can be performed on them. Data types in Java are divided mainly into primitive types and reference types.
- ◆ A literal signifies a value assigned to a variable in the Java program. Java SE 7 supports the use of the underscore characters (\_) between the digits of a numeric literal.
- ◆ The output of the Java program can be formatted using three ways: `print()` and `println()`, `printf()`, `format()`. Similarly, the Scanner class allows the user to read or accept values of various data types from the keyboard.
- ◆ Operators are symbols that help to manipulate or perform some sort of function on data.
- ◆ Parentheses are used to change the order in which an expression is evaluated.
- ◆ The type casting feature helps in converting a certain data type to another data type. The type casting can be automatic or manual and should follow the rules for promotion.

© Aptech Ltd. | Variables and Operators / Session 3 | 74

In slide 74, you will summarize the session. End the session with a brief summary of what has been taught in the session.

### 3.3 Post Class Activities for Faculty

You should familiarize yourself with the topics of the next session which is how to implement decision-making constructs in the Java program.

#### Tips:

You can also check the Articles/Blogs/Expert Videos uploaded on the Online Varsity site to gain additional information related to the topics covered in the next session. You can also connect to online tutors on the Online Varsity site to ask queries related to the sessions.

# Session 4 – Decision-Making Constructs

## 4.1 Pre-Class Activities

Familiarize yourself with the topics of this session in-depth. You should revisit topics of the previous session for a brief review.

Here, you can ask students the key topics they can recall from previous session. Prepare a question or two which will be a key point to relate the current session objectives.

### 4.1.1 Objectives

By the end of this session, the learners will be able to:

- Identify the need for decision-making statements
- List the different types of decision-making statements
- Explain the if statement
- Explain the various forms of if statement
- Explain the switch-case statement
- Explain the use of strings and enumeration in the switch-case statement
- Compare the if-else and switch-case statement

### 4.1.2 Teaching Skills

To teach this session, you should be well-versed with different types of decision-making statements in Java programming language, comparison of data, and sequence of execution of statements. You should teach the concepts in the theory class using the images provided. For teaching in the class, you are expected to use slides and LCD projectors.

#### Tips:

It is recommended that you test the understanding of the students by asking questions in between the class.

#### In-Class Activities:

Follow the order given here during In-Class activities.

**Overview of the Session:**

Give the students the overview of the current session in the form of session objectives. Show the students slide 2 of the presentation.

| Objectives                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>◆ Identify the need for decision-making statements</li><li>◆ List the different types of decision-making statements</li><li>◆ Explain the if statement</li><li>◆ Explain the various forms of if statement</li><li>◆ Explain the switch-case statement</li><li>◆ Explain the use of strings and enumeration in the switch-case statement</li><li>◆ Compare the if-else and switch-case statement</li></ul> |

Tell the students that the ability to compare values and alter the course of a program, based on the result of the comparison. This ability of comparison gives a computer the power to solve problems.

There are two aspects to making decisions, the first is the comparison of data and the second is the sequence of execution. This session focuses more on decision-making statements.

## 4.2 In-Class Explanations

### Slide 3

Let us understand the introduction of control flow statements.

### Introduction



- ◆ A Java program consists of a set of statements which are executed sequentially in the order in which they appear.
- ◆ The change in the flow of statements is achieved by using different **control flow statements**.
- ◆ Three categories of control flow statements supported by Java programming language are as follows:

|                                   |                                                                                                                                   |
|-----------------------------------|-----------------------------------------------------------------------------------------------------------------------------------|
| <b>Conditional<br/>Statements</b> | <ul style="list-style-type: none"> <li>• These types of statements are also referred to as decision-making statements.</li> </ul> |
| <b>Iteration<br/>Statements</b>   | <ul style="list-style-type: none"> <li>• These types of statements are also referred to as looping constructs.</li> </ul>         |
| <b>Branching<br/>Statements</b>   | <ul style="list-style-type: none"> <li>• These types of statements are referred to as jump statements.</li> </ul>                 |

© Aptech Ltd.

Decision-Making Constructs/Session 4

3

Using slide 3, explain the purpose of decision making statements.

Tell the students that the statements inside your source code files are generally executed from top to bottom, in the order that they appear. *Control flow statements*, however, break up the flow of execution by employing decision making, looping, and branching, enabling your program to *conditionally* execute particular blocks of code.

Mention that a Java program is a set of statements, which are executed sequentially in the order in which they appear. However, in some cases, the order of execution of statements may change based on the evaluation of certain conditions. The Java programming language possesses decision-making capabilities.

Then, explain the three categories of control flow statements supported by Java programming language.

**Slide 4**

Let us understand decision-making statements.

**Decision-making Statements**

- ◆ Enable us to change the flow of execution of a Java program.
- ◆ Evaluates a condition and based on the result of evaluation, a statement or a sequence of statements is executed.
- ◆ Different types of decision-making statements supported by Java are as follows:

**if Statement**

**Switch-case Statement**

© Aptech Ltd. | Decision-Making Constructs / Session 4 | 4

Using slide 4, explain the types of decision-making statements in Java.

Decision-making statements enable the user to change the flow of execution of a Java program and evaluate a condition. Then, based on the result of evaluation, a statement or a sequence of statements is executed. It is an expression that produces true or false results. Boolean operators can be used in conditional statements.

Different types of decision-making statements supported by Java are, `if` Statement and `Switch-case` statement. The `if` statement executes a block of code only if the specified expression is true. If the value is false, then the `if` block is skipped and execution continues with the rest of the program. The `switch-case` statement is a multi-way branch with several choices.

**Slides 5 to 9**

Let us understand 'if' statement.

**'if' Statement 1-5**

- ◆ It is the most basic form of decision-making statement.
- ◆ It evaluates a given condition and based on the result of evaluation executes a certain section of code.
- ◆ If the condition evaluates to `true`, then the statements present within the `if` block gets executed.
- ◆ If the condition evaluates to `false`, the control is transferred directly to the statement outside the `if` block.

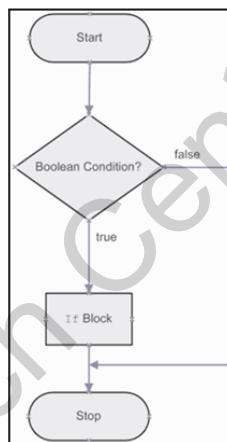
© Aptech Ltd.

Decision-Making Constructs/Session 4

5

**'if' Statement 2-5**

- ◆ Following figure shows the flow of execution for the `if` statement:



© Aptech Ltd.

Decision-Making Constructs/Session 4

6

**'if' Statement 3-5**

- The syntax for using the `if` statement is as follows:

**Syntax**

```
if (condition) {
 // one or more statements;
}
```

where,

**condition:** Is the boolean expression.

**statements:** Are instructions/statements enclosed in curly braces. These statements are executed when the boolean expression evaluates to true.

**'if' Statement 4-5**

- Following code snippet demonstrates the code that performs conditional check on the value of a variable:

```
public class CheckNumberValue {
 /**
 * @param args the command line arguments
 */
 public static void main(String[] args) {
 int first = 400, second = 700, result;
 result = first + second;

 // Evaluates the value of result variable
 if (result > 1000) {
 second = second + 100;
 }
 System.out.println("The value of second is " + second);
 }
}
```

- The program tests the value of the variable, `result` and accordingly calculates value for the variable, `second`.
- If the value of `result` is greater than 1000, then the value of the variable `second` is incremented by 100.

### 'if' Statement 5-5



- ◆ If the evaluation of condition is **false**, the value of the variable **second** is not incremented.
- ◆ Finally, the value of the variable **second** gets printed on the console.
- ◆ Following figure shows the output of the code:

The screenshot shows a terminal window titled "Output - Session4 (run)". It displays the command "run:" followed by the output "The value of second is 800". At the bottom, it says "BUILD SUCCESSFUL (total time: 0 seconds)".

- ◆ If there is only a single action statement within the body of the **if** block, then use of opening and closing curly braces is optional.

© Aptech Ltd.

Decision-Making Constructs/Session 4

9

Using slides 5 to 9, explain the **if** statement in Java.

Explain about the simple **if** statement. Tell that the simple **if** statement helps in decision-making based on the evaluation of a given condition to true or false. Explain that **if** the condition is true, then the statements in the **if** block get executed. On the other hand, if the condition evaluates to false, the control is transferred to the statement directly outside the **if** block.

Explain the syntax of the **if** statement.

Mention that the opening and closing braces are optional, provided that the **if** clause contains only one statement. Explain to them that deciding when to omit the braces is a matter of personal choice. Omitting them can make the code more brittle. If a second statement is later added to the **if** clause, a common mistake would be forgetting to add the newly required braces. The compiler cannot catch this sort of error and thus, you will get the wrong output or results.

#### Tips:

The **if** statement braces contain the statements that are to be executed only if the condition evaluates to true.

Explain to the student the concept of simple **if** statement using the code snippet.

Tell them that the program tests the value of **result** and accordingly calculates the value of **second** and prints it. If the **result** is greater than 1000, then the variable **second** gets increased by 100 and is printed. If not, the value of **second** is not increased and the original value is printed. The output of the code will be **800** for the **second** value.

**Slides 10 to 12**

Let us understand 'if-else' statement.

**'if-else' Statement 1-3**

- ◆ Sometimes, it is required to define a block of statements to be executed when a condition evaluates to `false`.
- ◆ This is done by using the `if-else statement`.
- ◆ `if-else Statement`:
  - ◆ Begins with the `if` block followed by the `else` block.
  - ◆ `else` block specifies a block of statements that are to be executed when a condition evaluates to `false`.
- ◆ The syntax for using the `if-else statement` is as follows:

**Syntax**

```
if (condition) {
 // one or more statements;
}
else {
 // one or more statements;
}
```

© Aptech Ltd.

Decision-Making Constructs/Session 4

10

**'if-else' Statement 2-3**

- ◆ Following code snippet demonstrates the code that checks whether a number is even or odd:

```
public class Number_Division {
 /**
 * @param args the command line arguments
 */
 public static void main(String[] args) {
 int number = 11, remainder;

 // % operator to return the remainder of the division
 remainder = number % 2;
 if (remainder == 0) {
 System.out.println("Number is even");
 } else {
 System.out.println("Number is odd");
 }
 }
}
```

- ◆ In the code, the variable, `number` is divided by 2 to obtain the remainder of the division.

© Aptech Ltd.

Decision-Making Constructs/Session 4

11

### 'if-else' Statement 3-3



- ❖ The % (modulus) operator which returns the remainder after performing the division.
- ❖ If the remainder is 0, the message **Number is even** is printed.  
Otherwise, the message **Number is odd** is printed.
- ❖ Following figure shows the output of the code:

The screenshot shows a Java IDE's output window titled "Output - Session4 (run)". It contains the following text:  
 run:  
 Number is odd  
 BUILD SUCCESSFUL (total time: 0 seconds)

© Aptech Ltd.

Decision-Making Constructs/Session 4

12

Using slides 10 to 12, explain the `if-else` statement in Java.

Tell them that the `if-else` statement provides a secondary path of execution, when an `if` clause evaluates to false.

Mention to the students that generally, it is not only important to specify an action to be performed when the condition is true, but also to specify what action is to be performed if the condition is false. To do this, the `if-else` statement can be used.

Explain to the students the syntax of the `if-else` statements.

Mention that the program checks whether a number is even or odd. If the remainder is 0, the message, 'Number is even' is printed. Else, the message, 'Number is odd' is printed.

The output of the program is the message '**Number is odd**'.

#### Tips:

Mention to the students that the `if` statement can be followed by an optional `if...else` statement, which is very useful to test various conditions using single `if...else` statement.

**Slides 13 to 16**

Let us understand nested-if statement.

**Nested-if Statement 1-4**

- ◆ An if statement can also be used within another if statement forming a nested-if.
- ◆ A nested-if statement is an if statement that is the target of another if or else statement.
- ◆ The syntax to use the nested-if statements is as follows:

**Syntax**

```
if(condition) {
 if(condition)
 true-block statement(s);
 else
 false-block statement(s);
}

else {
 false-block statement(s);
}
```

© Aptech Ltd.

Decision-Making Constructs/Session 4

13

**Nested-if Statement 2-4**

- ◆ Following code snippet checks whether a number is divisible by 3 as well as 5:

```
import java.util.*;
public class NumberDivisibility {
 /**
 * @param args the command line arguments
 */
 public static void main(String[] args) {
 // Scanner class is used to accept values from the user
 Scanner input = new Scanner(System.in);
 System.out.println("Enter a Number: ");
 int num = input.nextInt();

 // Checks whether a number is divisible by 3
 if (num % 3 == 0) {
 System.out.println("Inside Outer if Block");

 // Inner if statement checks if number is divisible by 5
 if (num % 5 == 0) {
 System.out.println("Number is divisible by 3 and 5");
 } else {
 System.out.println("Number is divisible by 3, but not by 5");
 } // End of inner if-else statement
 }
 }
}
```

© Aptech Ltd.

Decision-Making Constructs/Session 4

14

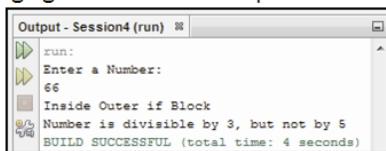
### Nested-if Statement 3-4

```

 } else {
 System.out.println("Number is not divisible by 3");
 } // End of outer if-else statement
 }
}

```

- ◆ The code declares a variable **num** to store an integer value accepted from the user.
- ◆ Initially, the outer **if** statement is evaluated. If it evaluate to:
  - **false**, then the inner **if-else** statement is skipped and the final **else** block is executed.
  - **true**, then its body containing the inner **if-else** statement is evaluated.
- ◆ Following figure shows the output of the code:



© Aptech Ltd.

Decision-Making Constructs/Session 4

15

### Nested-if Statement 4-4

- ◆ The important points to remember about nested-if statements are as follows:

• [Text]

An **else** statement should always refer to the nearest **if** statement.

The **if** statement must be within the same block as the **else** and it should not be already associated with some other **else** statement.

• [Text]

© Aptech Ltd.

Decision-Making Constructs/Session 4

16

Using slides 13 to 16, explain the nested-if statement.

Explain to the students that the **if-else** statement tests the result of a condition, that is, a boolean expression, and performs appropriate actions based on the result. The **if** statement can also be used inside another if. This is known as nested-if. Thus, a nested-if is the **if** statement that is the target of another if or else.

The important points to remember in nested-if statements are as follows:

- The else statement should always refer to the nearest if statement.
- It should be within the same block.

Explain to the students the syntax of the nested-if statements.

Then, explain the code snippet, a variable `num` is declared which stores an integer value accepted from the user. Then, using nested-if statements, it checks whether `num` is divisible by 3 and 5 or only by 3, and then prints an appropriate message. Bring to the notice of the students that the final `else` is associated with `if(num % 3 == 0)`. Tell them that the inner `else` refers to `if(num % 5 == 0)`, because it is closest to the inner `if` within the same block.

### Slides 17 to 21

Let us understand ‘if-else-if’ ladder.

#### ‘if-else-if’ Ladder 1-5



- ◆ The multiple `if` construct is known as the `if-else-if` ladder.
- ◆ The conditions are evaluated sequentially starting from the top of the ladder and moving downwards.
- ◆ When a condition controlling the `if` statement is evaluated as `true`, then the associated statements associated are executed and all other `else-if` statements are bypassed.
- ◆ If none of the condition is `true`, then the final `else` statement also referred as default statement is executed.

**'if-else-if' Ladder 2-5**

- The syntax for using the if-else-if statement is as follows:

**Syntax**

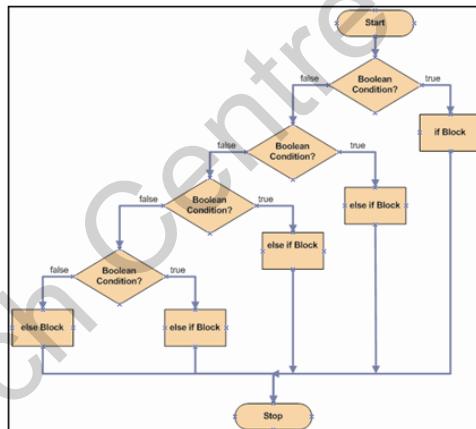
```
if(condition) {
 // one or more statements
}

else if (condition) {
 // one or more statements
}

else {
 // one or more statements
}
```

**'if-else-if' Ladder 3-5**

- Following figure shows the flow of execution for the if-else-if ladder:



**'if-else-if' Ladder 4-5**

- Following code snippet checks the total marks and prints the appropriate grade:

```
public class CheckMarks {
 /**
 * @param args the command line arguments
 */
 public static void main(String[] args) {
 int totalMarks = 59;
 /* Tests the value of totalMarks and accordingly transfers
 * control to the else if statement
 */
 if (totalMarks >= 90) {
 System.out.println("Grade = A+");
 } else if (totalMarks >= 60) {
 System.out.println("Grade = A");
 } else if (totalMarks >= 40) {
 System.out.println("Grade = B");
 } else if (totalMarks >= 30) {
 System.out.println("Grade = C");
 } else {
 System.out.println("Fail");
 }
 }
}
```

© Aptech Ltd.

Decision-Making Constructs/Session 4

20

**'if-else-if' Ladder 5-5**

- If the code satisfies a given condition, then:
  - The statements within that else if condition are executed.
  - After execution of the statements, the control breaks.
  - Remaining if conditions are bypassed for evaluation.
- If none of the condition is satisfied, then:
  - The final else statement, also known as the default else statement is executed.
- Following figure shows the output of the code:

© Aptech Ltd.

Decision-Making Constructs/Session 4

21

Using slides 17 to 21, explain the if-else-if ladder.

Tell them that the multiple if construct is also known as the if-else-if ladder. The conditions are evaluated sequentially starting from the top of the ladder and moving downwards. When a true condition is found, the statement associated with the true condition is executed.

Explain them the syntax of multiple if statements.

Explain to the students that when they are using if , else if , else statements, there are few points to be kept in mind. They are as follows:

- The if can have zero or one else's and it must come after any else if's.
- The if can have zero to many else if's and they must come before the else.
- Once the else if succeeds, none of the remaining else if's or else's will be tested.

Tell the students that in the code snippet, the value of `totalmarks` can satisfy the expression in the statement, `totalMarks >= 40`. However, once a condition is satisfied, the appropriate statements are executed (`Grade = B`) and the remaining conditions are not evaluated. The output will be, `Grade = B`.

### Slides 22 to 32

Let us understand ‘switch-case’ Statement.

#### ‘switch-case’ Statement 1-11



- ◆ Alternative for too many `if` statements representing multiple selection constructs.
- ◆ Contains a variable as an expression whose value is compared against different values.
- ◆ Results in better performance.
- ◆ Can have a number of possible execution paths depending on the value of expression provided with the `switch` statement.
- ◆ Can evaluate different primitive data types, such as `byte`, `short`, `int`, and `char`.

#### ‘switch-case’ Statement 2-11



##### Enhancements to `switch-case` statement in Java SE 7

- Supports the use of strings in the `switch-case` statement.
- String variable can be passed as an expression for the `switch` statement.
- Supports use of objects from classes present in the Java API.
- The classes whose objects can be used are `Character`, `Byte`, `Short`, and `Integer`.
- Supports the use of enumerated types as expression.

**'switch-case' Statement 3-11**

- The syntax for using the `switch-case` statement is as follows:

**Syntax**

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

where,

`switch`: The `switch` keyword is followed by an expression enclosed in parentheses.

**'switch-case' Statement 4-11**

`Case`: The `case` keyword is followed by a constant and a colon. Each `case` value is a unique literal.

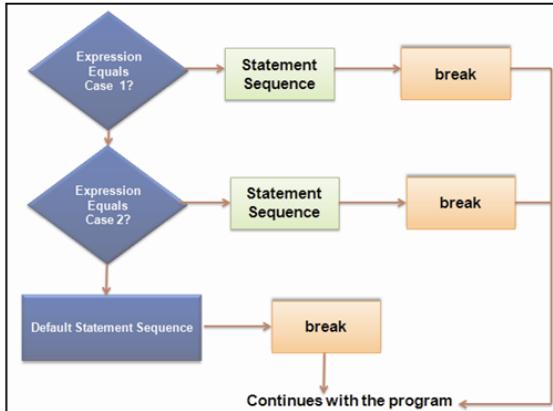
`default`: If no `case` value matches the `switch` expression value, execution continues at the `default` clause.

`break`: The `break` statement is used inside the `switch-case` statement to terminate the execution of the statement sequence. It is optional. If there is no `break` statement, execution flows sequentially into the next cases.

### 'switch-case' Statement 5-11



- Following figure shows the flow of execution for the `switch-case` statement:



© Aptech Ltd.

Decision-Making Constructs/Session 4

26

### 'switch-case' Statement 6-11



The value of the expression specified with the `switch` statement is compared with each case constant value.

If any case value matches, the corresponding statements in that `case` are executed.

When the `break` statement is encountered, it terminates the `switch-case` block and control switches to the statements following the block.

The `break` statement must be provided as without it, even after the matching case is executed; all other cases following the matching case are also executed.

If there is no matching case, then the `default` case is executed.

© Aptech Ltd.

Decision-Making Constructs/Session 4

27

### 'switch-case' Statement 7-11



- Following code snippet demonstrates the use of the switch-case statement:

```
public class TestNumericOperation {
 /**
 * @param args the command line arguments
 */
 public static void main(String[] args) {
 // Declares and initializes the variable
 int choice = 3;

 // switch expression value is matched with each case
 switch (choice) {
 case 1:
 System.out.println("Addition");
 break;
 case 2:
 System.out.println("Subtraction");
 break;
 case 3:
 System.out.println("Multiplication");
 break;
 }
 }
}
```

© Aptech Ltd.

Decision-Making Constructs/Session 4

28

### 'switch-case' Statement 8-11



```
case 4:
 System.out.println("Division");
 break;
default:
 System.out.println("Invalid Choice");
} // End of switch-case statement
}
```

- Value of the expression, `choice` is compared with the literal value in each of the case statement.
- Here, case 3 is executed, as its value is matching with the expression.
- The control moves out of the switch-case, due to the presence of the `break` statement.
- Following figure shows the output of the code:

© Aptech Ltd.

Decision-Making Constructs/Session 4

29

## 'switch-case' Statement 9-11



- ◆ Sometimes, it is required to have multiple case statements to be executed without a break statement.
- ◆ Following code snippet demonstrates the use of multiple case statements with no break statement:

```
public class NumberOfDays {
 /**
 * @param args the command line arguments
 */
 public static void main(String[] args) {

 int month = 5;
 int year = 2001;
 int numDays = 0;

 // Cases are executed until a break statement is encountered
 switch (month) {
 case 1:
 case 3:
 case 5:
 case 7:
```

## 'switch-case' Statement 10-11



```
case 8:
case 10:
case 12:
 numDays = 31;
 break;
case 4:
case 6:
case 9:
case 11:
 numDays = 30;
 break;
case 2:
 if (year % 4 == 0) {
 numDays = 29;
 } else {
 numDays = 28;
 }
 break;
default:
 System.out.println("Invalid Month");
} // End of switch-case statement
System.out.println("Month: " + month);
System.out.println("Number of Days: " + numDays);
}
```

**'switch-case' Statement 11-11**

The value of expression, `month` is compared through each case, till a `break` statement or end of the `switch-case` block is encountered.

Following figure shows the output of the code:

Output - Session4 (run)

```

run:
Month: 5
Number of Days: 31
BUILD SUCCESSFUL (total time: 0 seconds)

```

Using slides 22 to 32, explain the `switch-case` statement.

The `switch-case` statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case. The `switch-case` statement is alternative for too many `if` statements representing multiple selection constructs. It contains a variable as an expression whose value is compared against different values. Its results are better performance. It can have a number of possible execution paths depending on the value of expression provided with the `switch` statement. It can evaluate different primitive data types, such as `byte`, `short`, `int`, and `char`.

Tell them that the `switch-case` statement is used, when a variable needs to be compared against different values.

Explain to them about the syntax of the `switch-case` statements. Mention that the important keywords in `switch-case` statement are as follows:

- `switch`
- `case`
- `default`
- `break`

#### Tips:

Unlike `if-then` and `if-then-else` statements, the `switch` statement can have a number of possible execution paths. A `switch` works with the `byte`, `short`, `char`, and `int` primitive data types.

Explain to the students the keywords `switch` and `case`. Tell the students that the body of a `switch` statement is known as a *switch block*. Explain to them that the `switch` keyword is followed by an integer expression enclosed in parentheses. The expression must be of type `int`, `char`, `byte`, or `short`. Mention that each case value must be a unique literal. Thus, it must be a constant and not a variable.

The `switch` statement executes the case corresponding to the value of the expression. Also mention that the `case` keyword is followed by an integer constant and a colon. Each case value is a unique literal. The `case` statement might be followed by code sequences that are executed, when the `switch` expression and the `case` value match.

Then, explain to the students about the `default` and `break` clause. Mention to the student that a switch statement can have an optional `default` case, which must appear at the end of the `switch`.

The `default` case can be used for performing a task, when none of the case statements are true. No `break` is needed in the `default` case. In other words, explain that if no case value matches the `switch` expression value, execution continues at the `default` clause. This is the equivalent to the `else` clause for the `switch` statement.

Mention to the student the importance of `break` statement. Tell them that when a `break` statement is reached, the `switch` terminates, and the flow of control jumps to the next line following the `switch` statement. Therefore, the `break` statement is used inside the `switch-case` statement to terminate the execution of the statement sequence. The control is then transferred to the first statement, after the end of the `switch`. The `break` statement is optional. If there is no `break`, execution flows sequentially into the next case statement. Sometimes, multiple cases can be present without `break` statements between them.

**Tips:**

Each `break` statement terminates the enclosing `switch` statement. Control flow continues with the first statement following the `switch` block. The `break` statements are necessary because without them, statements in `switch` blocks fall through.

All statements after matching case label are executed in sequence, regardless of the expression of subsequent case labels, until a `break` statement is encountered.

Then, explain the code snippet on slides 28 and 29 which demonstrate the use of `switch-case` statement. Explain to the students that in the code snippet, case 3 will be executed because value of `choice` is 3. So, the message, `Multiplication` is displayed and in the next statement, when `break` is encountered the program exits.

Then, explain another code snippet that uses multiple case statements with single `break` statement.

**Slides 33 to 36**

Let us understand string-based 'switch-case' statement.

**String-based 'switch-case' Statement 1-4**

- ◆ Java SE 7 supports the use of strings in the switch-case statement.
- ◆ A String is not a primitive data type, but an object in Java.
- ◆ To use strings for comparison, a String object is passed as an expression in the switch-case statement.
- ◆ Following code snippet demonstrates the use of strings in the switch-case statement:

```
public class DayofWeek {
 /**
 * @param args the command line arguments
 */
 public static void main(String[] args) {
 String day = "Monday";

 // switch statement contains an expression of type String
 switch (day) {
 case "Sunday":
 System.out.println("First day of the Week");
 break;
```

**String-based 'switch-case' Statement 2-4**

```
case "Monday":
 System.out.println("Second Day of the Week");
 break;
case "Tuesday":
 System.out.println("Third Day of the Week");
 break;
case "Wednesday":
 System.out.println("Fourth Day of the Week");
 break;
case "Thursday":
 System.out.println("Fifth Day of the Week");
 break;
case "Friday":
 System.out.println("Sixth Day of the Week");
 break;
case "Saturday":
 System.out.println("Seventh Day of the Week");
 break;
default:
 System.out.println("Invalid Day");
} // End of switch-case statement
```

### String-based 'switch-case' Statement 3-4



- ◆ The statement `String day="Monday"` creates an object named day of type String and initializes it.
- ◆ The object is passed as an expression to the switch statement.
- ◆ The value of this expression, that is "Monday", is compared with the value of each case statement.
- ◆ If no matching statement is found, then the statement associated with the default clause is executed.
- ◆ Following figure shows the output of the code:

```
Output - Session4 (run) ॥
run:
Second Day of the Week
BUILD SUCCESSFUL (total time: 0 seconds)
```

### String-based 'switch-case' Statement 4-4



- ◆ Following points are to be considered while using strings with the switch-case statement:

#### Null Values

- A runtime exception is generated when a String variable is assigned a null value and is passed as an expression to the switch statement.

#### Case-sensitive values

- The value of String variable that is matched with the case literals is case sensitive.
- Example: a String value "Monday" when matched with the case labeled "MONDAY":, then it will not be treated as a matched value.

Using slides 33 to 36, explain the string-based switch statement in Java SE 7.

Allowing the use of strings in switch-case statement enables the program to incorporate a readable code. A string is not a primitive data type, but an object in Java. Thus, to use strings for comparison, a String object is passed as an expression in the switch-case statement.

It is preferable to use string with switch as Java compiler generates most efficient byte code from switch statements that use String objects.

Slides 33 to 35 shows the code snippet that demonstrates the use of strings in the `switch-case` statement.

Following points are to be considered while using strings with the `switch-case` statement:

#### Null Values

A runtime exception is generated when a String variable is assigned a null value and is passed as an expression to the switch statement.

#### Case-sensitive values

The comparison of string objects in switch statements is case-sensitive. The value of String variable that is matched with the case literals is case sensitive. Example: a String value “**Monday**” when matched with the case labeled “**MONDAY**”: then it will not be treated as a matched value.

#### In-Class Question:

After you finish explaining the string based switch case statements, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



When a runtime exception is generated?

#### Answer:

It is generated when a string variable is assigned a null value and is passed as an expression to the switch statement.

#### Slides 37 to 39

Let us understand enumeration-based ‘switch-case’ statement.

#### Enumeration-based ‘switch-case’ Statement 1-3



- ◆ The `switch-case` statement supports the use of an enumeration (`enum`) value in the expression.
- ◆ The constraint with an `enum` expression is that:
  - ◆ All case constants must belong to the same `enum` variable used with the `switch` statement.
- ◆ Following code snippet demonstrates the use of enumerations in the `switch-case` statement:

```
public class TestSwitchEnumeration {
 /**
 * An enumeration of Cards Suite
 */

 enum Cards {
 Spade, Heart, Diamond, Club
 }
 /**
 * Gparam args the command line arguments
 */
 public static void main(String[] args) {
 Cards card = Cards.Diamond;
 }
}
```

### Enumeration-based 'switch-case' Statement 2-3



```
// enum variable is used to control a switch statement
switch (card) {
 case Spade:
 System.out.println("SPADE");
 break;
 case Heart:
 System.out.println("HEART");
 break;
 case Diamond:
 System.out.println("DIAMOND");
 break;
 case Club:
 System.out.println("CLUB");
 break;
} // End of switch-case statement
```

- ◆ The `enum` `card` is passed as an expression to the `switch` statement.
- ◆ Each `case` statement has an enumeration constant associated with it and does not require it to be qualified by the enumeration name.

### Enumeration-based 'switch-case' Statement 3-3



- ◆ Following figure shows the output of the code:

```
Output - Session4 (run) ✘
run:
DIAMOND
BUILD SUCCESSFUL (total time: 0 seconds)
```

Using slides 37 to 39, explain the enumeration-based switch-case statement.

The switch-case statement supports the use of an enumeration value in the expression. It is one of the greatest new feature introduce from the version 5. Using Java enum in switch-case is straight forward. Just use enum reference variable in switch and enum constants and interfaces in case statement. The constraint with an enum expression is that all case constants must belong to the same enum variable used with the switch statement.

Then, explain the code snippet that demonstrates the use of enumerations in the switch-case statement.

### Slides 40 to 43

Let us understand nested 'switch-case' statement.

#### Nested 'switch-case' Statement 1-4



- ◆ A switch-case statement can be used as a part of another switch-case statement. This is referred to as nested switch-case statements.
- ◆ Following code snippet demonstrates the use of nested switch-case statements:

```
public class Greeting {
 /**
 * @param args the command line arguments
 */
 public static void main(String[] args) {
 // String declaration
 String day = "Monday";
 String hour = "am";

 // Outer switch statement
 switch (day) {
 case "Sunday":
 System.out.println("Sunday is a Holiday...");
 // Inner switch statement
 switch (hour) {

```

#### Nested 'switch-case' Statement 2-4



```
case "am":
 System.out.println("Good Morning");
 break;
case "pm":
 System.out.println("Good Evening");
 break;
} // End of inner switch-case statement
break; // Terminates the outer case statement

case "Monday":
 System.out.println("Monday is a Working Day...");
 switch (hour) {
 case "am":
 System.out.println("Good Morning");
 break;
 case "pm":
 System.out.println("Good Evening");
 break;
 } // End of inner switch-case statement
break;
default:
 System.out.println("Invalid Day");
} // End of the outer switch-case statement
}
```

### Nested 'switch-case' Statement 3-4



- ◆ The variable, **day** is used as an expression with the outer **switch** statement.
- ◆ If the value of **day** variable matches with “**Sunday**” or “**Monday**”, then the inner **switch-case** statement is executed.
- ◆ The inner **switch** statement compares the value of **hour** variable with case constants “**am**” or “**pm**”.

- ◆ Following figure shows the output of the code:

```
Output - Session4 (run) ✎
run:
Monday is a Working Day...
Good Morning
BUILD SUCCESSFUL (total time: 0 seconds)
```

© Aptech Ltd.

Decision-Making Constructs/Session 4

42

### Nested 'switch-case' Statement 4-4



- ◆ The three important features of **switch-case** statements are as follows:

The **switch-case** statement differs from the **if** statement, as it can only test for equality.

No two case constants in the same **switch** statement can have identical values, except the nested **switch-case** statements.

A **switch** statement is more efficient and executes faster than a set of nested-if statements.

© Aptech Ltd.

Decision-Making Constructs/Session 4

43

Using slides 40 to 43, explain the nested-switch statement.

A **switch-case** statement can be used as a part of another **switch-case** statement. This is referred to as nested **switch-case** statements and it is exactly same idea as nested **if-else** statement. Since, the **switch-case** statement defines its own blocks; no conflicts arise between the case constants present in the inner **switch** and those present in the outer **switch** because each **switch** statement defines its own block.

Nested-switch statement can be use in many applications as it require less work than nested-if statements and do not have compatibility issues.

Slides 40 to 42 shows the code snippet that demonstrates the use of nested **switch-case** statements. In the code, the variable, **day** is used as an expression with the outer **switch** statement. It is compared with the list of cases provided with the outer **switch-case** statements. If the value of **day** variable matches with “**Sunday**” or “**Monday**”, then the inner

switch-case statement is executed. The inner switch statement compares the value of **hour** variable with case constants “**am**” or “**pm**”.

The three important features of switch-case statements are as follows:

- The switch-case statement differs from the if statement, as it can only test for equality.
- No two case constants in the same switch statement can have identical values, except the nested switch-case statements.
- A switch statement is more efficient and executes faster than a set of nested-if statements.

#### In-Class Question:

After you finish explaining the Nested switch case statements, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



What is Nested switch case statement?

#### Answer:

A switch-case statement can be used as a part of another switch-case statement which is referred to as nested switch-case statements.

#### Slide 44

Let us understand comparison between if and switch-case statement.

| Comparison Between if and switch-case Statement                                   |                                                                                       |
|-----------------------------------------------------------------------------------|---------------------------------------------------------------------------------------|
| if                                                                                | switch-case                                                                           |
| Each if statement has its own logical expression to be evaluated as true or false | Each case refers back to the original value of the expression in the switch statement |
| The variables in the expression may evaluate to a value of any type               | The expression must evaluate to a byte, short, char, int, or String                   |
| Only one of the blocks of code is executed                                        | If the break statement is omitted, the execution will continue into the next block    |

Using slide 44, explain the difference between if and switch statement.

Tell them that the if-else-if and the switch-case decision-making statements have similar use in a program, but there are distinct differences between them.

They are as follows:

#### **if-else-if**

- Each `if` has its own logical expression to be evaluated as true or false.
- The variables in the expression may evaluate to a value of any type.
- Only one of the blocks of code is executed.

#### **switch**

- Each case refers back to the original value of the expression in the `switch` statement.
- The expression must evaluate to a `byte`, `short`, `char` or `int`.
- If the `break` statement is omitted, the execution will continue into the next block.

#### **In-Class Question:**

After you finish explaining the decision-making statements in Java, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



Which decision-making statement allows for any number of possible execution paths?

#### **Answer:**

The `switch` statement allows for any number of possible execution paths.

#### **Slide 45**

Let us summarize the session.

| Summary                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |  |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--|
| <ul style="list-style-type: none"> <li>◆ A Java program is a set of statements, which are executed sequentially in the order in which they appear.</li> <li>◆ The three categories of control flow statements supported by Java programming language include: conditional, iteration, and branching statements.</li> <li>◆ The <code>if</code> statement is the most basic decision-making statement that evaluates a given condition and based on result of evaluation executes a certain section of code.</li> <li>◆ The <code>if-else</code> statement defines a block of statements to be executed when a condition is evaluated to false.</li> <li>◆ The multiple <code>if</code> construct is known as the <code>if-else-if</code> ladder with conditions evaluated sequentially from the top of the ladder.</li> <li>◆ The <code>switch-case</code> statement can be used as an alternative approach for multiple selections. It is used when a variable needs to be compared against different values. Java SE 7 supports strings and enumerations in the <code>switch-case</code> statement.</li> <li>◆ A <code>switch</code> statement can also be used as a part of another <code>switch</code> statement. This is known as nested <code>switch-case</code> statements.</li> </ul> |  |

© Aptech Ltd.      Decision-Making Constructs/Session 4      45

In slide 45, you will summarize the session. End the session with a brief summary of what has been taught in the session.

#### **4.3 Post Class Activities for Faculty**

You should familiarize yourself with the topics of the next session which is based on iterative construct in Java.

**Tips:**

You can also check the Articles/Blogs/Expert Videos uploaded on the Online Varsity site to gain additional information related to the topics covered in the next session. You can also connect to online tutors on the Online Varsity site to ask queries related to the sessions.

# Session 5 – Looping Constructs

---

## 5.1 Pre-Class Activities

Before you commence the session, you should familiarize yourself with the topics of this session in-depth.

Here, you can discuss the key points with the students that were covered in the previous session. Prepare a question or two which will help you to relate the current session objectives.

### 5.1.1 Objectives

By the end of this session, the learners will be able to:

- List the different types of loops
- Explain the while statement and the associated rules
- Identify the purpose of the do-while statement
- State the need of for statement
- Describe nested loops
- Compare the different types of loops
- State the purpose of jump statements
- Describe break statement
- Describe continue statement

### 5.1.2 Teaching Skills

To teach this session, you should be well-versed with different types of looping statements available in Java programming language and the various jump statements which is called branching statements.

You should teach the concepts in the theory class using the images provided. For teaching in the class, you are expected to use slides and LCD projectors.

#### Tips:

It is recommended that you test the understanding of the students by asking questions in between the class.

#### In-Class Activities:

Follow the order given here during In-Class activities.

## Overview of the Session:

Give the students the overview of the current session in the form of session objectives. Show the students slide 2 of the presentation.

**Objectives**

- ◆ List the different types of loops
- ◆ Explain the while statement and the associated rules
- ◆ Identify the purpose of the do-while statement
- ◆ State the need of for statement
- ◆ Describe nested loops
- ◆ Compare the different types of loops
- ◆ State the purpose of jump statements
- ◆ Describe break statement
- ◆ Describe continue statement

© Aptech Ltd. Looping Constructs/Session 5 2

Tell the students that this session introduces the different types of looping statements available in Java programming language and the various jump statements which is called branching statements. Also it introduces the nested loops and compares the different types of loops.

## 5.2 In-Class Explanations

### Slide 3

Let us understand the concepts of loops.

**Introduction**

- ◆ A computer program consists of a set of statements, which are usually executed sequentially.
- ◆ However, in certain situations, it is necessary to repeat certain steps to meet a specified condition.
- ◆ Following figure shows the program that displays the multiples of 10:

|                                                                                                                                                                            |                                                                                                                                                                                                     |    |    |    |    |    |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|----|----|----|----|
| <pre>BEGIN     COMPUTE result1 AS 1 * 10     COMPUTE result2 AS 2 * 10     COMPUTE result3 AS 3 * 10     COMPUTE result4 AS 4 * 10     COMPUTE result5 AS 5 * 10 END</pre> | <b>Output</b> → <table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>10</td></tr> <tr><td>20</td></tr> <tr><td>30</td></tr> <tr><td>40</td></tr> <tr><td>50</td></tr> </table> | 10 | 20 | 30 | 40 | 50 |
| 10                                                                                                                                                                         |                                                                                                                                                                                                     |    |    |    |    |    |
| 20                                                                                                                                                                         |                                                                                                                                                                                                     |    |    |    |    |    |
| 30                                                                                                                                                                         |                                                                                                                                                                                                     |    |    |    |    |    |
| 40                                                                                                                                                                         |                                                                                                                                                                                                     |    |    |    |    |    |
| 50                                                                                                                                                                         |                                                                                                                                                                                                     |    |    |    |    |    |

- ◆ The same statement is repeating 5 times to display the multiple of 10 with 1, 2, 3, 4, and 5.
- ◆ Thus, a loop can be used in this situation.

© Aptech Ltd. Looping Constructs/Session 5 3

Using slide 3 explain, introduction of loops.

Tell the students that a computer program is a set of statements, which are usually executed sequentially. However, in certain situations, it is necessary to repeat certain steps to meet a specified condition.

For example, if the user wants to write a program that calculates and displays the sum of the first 10 numbers 1, 2, 3, ..., 10.

One way to calculate the same is as follows:

$$1+2=3$$

$$3+3=6$$

$$6+4=10$$

$$10+5=15$$

$$15+6=21$$

...

and so on.

This technique is suitable for relatively small calculations. However, if the program requires addition of the first 200 numbers, it would be tedious to add up all the numbers from 1 to 200 using the mentioned technique. In such situations, iterations or loops come to our rescue.

#### Slide 4

Let us understand looping statements.

**Looping Statements**

The loop statements supported by Java programming language are as follows:

- Loops enable programmers to develop concise programs, which otherwise would require thousands of program statements.
- Loops consists of statement or a block of statements that are repeatedly executed.
- Statements in the loops are executed until a condition evaluates to true or false.

**while**    **do-while**  
**for**    **for-each**

© Aptech Ltd.      Looping Constructs/Session 5      4

Using slide 4, explain Looping Statements.

Loops enable programmers to develop concise programs, which otherwise would require thousands of program statements. Loops consist of statement or a block of statements that are repeatedly executed. Statements in the loops are executed until a condition evaluates to true or false.

Mention to the students the different types of the loop statements supported by Java programming language are as follows:

- while
- do-while
- for
- for-each

#### Tips:

All programming languages follow these three basic types of loop statements. The syntax changes according to the programming language.

**In-Class Question:**

After you finish explaining looping statements, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



What does loop consists of?

**Answer:**

Loops consist of statement or a block of statements that are repeatedly executed, until a condition evaluates to true or false.

**Slides 5 to 9**

Let us understand the 'while' statement.

### 'while' Statement 1-5

**Syntax**

```
while (expression) {
 // one or more statements
}
```

where,

- expression: Is a conditional expression which must return a boolean value, that is, true or false.
- The use of curly braces ({ }) is optional and can be avoided, if there is only a single statement within the body of the loop. However, providing statements within the curly braces increases the readability of the code.

© Aptech Ltd. Looping Constructs/Session 5 5

### 'while' Statement 2-5

- Following figure shows the flow of execution of while loop:

The body of the if loop contains a set of statements.

Statements will be executed until the conditional expression evaluates to true.

When the conditional expression evaluates to false, the loop is terminated.

The control passes to the statement immediately following the loop.

```

graph TD
 Start([Start]) --> Cond{Evaluate Condition?}
 Cond -- true --> Execute[Execute Loop]
 Execute --> Cond
 Cond -- false --> Stop([Stop])

```

© Aptech Ltd. Looping Constructs/Session 5 6

### 'while' Statement 3-5



- Following code snippet demonstrates the code that displays multiples of 10 using the `while` loop:

```
public class PrintMultiplesWithWhileLoop {
 /**
 * @param args the command line arguments
 */
 public static void main(String[] args) {
 // Variable num acts as a counter variable
 int num = 1;
 // Variable product will store the result
 int product = 0;
```

- An integer variable, `num` is declared to store a number and is initialized to 1.
- It is used in the `while` loop to start multiplication from 1.

© Aptech Ltd.

Looping Constructs/Session 5

7

### 'while' Statement 4-5



```
// Tests the condition at the beginning of the loop
while (num <= 5) {
 product = num * 10;
 System.out.printf("\n %d * 10 = %d", num, product);
 num++; // Equivalent to n = n + 1
} // Moves the control back to the while statement

// Statement gets printed on loop termination
System.out.println("\n Outside the Loop");
}
```

- The conditional expression:
  - `num <= 5` is evaluated at the beginning of the `while` loop. The loop is executed only if the conditional expression evaluates to true.
  - In this case, as the value in the variable, `num` is less than 5, hence, the statements present in the body of the loop is executed.
- The first statement within the body of the loop calculates the product by multiplying `num` with 10.
- The next statement prints this value.
- The last statement `num++` increments the value of `num` by 1.
- The execution of the loop stops when condition becomes false, that is, when the value of `num` reaches 6.

© Aptech Ltd.

Looping Constructs/Session 5

8

### 'while' Statement 5-5



- Finally, the statement, 'Outside the Loop' is displayed.
- Following figure shows the output of the code:

```
Output - Session5 (run) ✘
run:
1 * 10 = 10
2 * 10 = 20
3 * 10 = 30
4 * 10 = 40
5 * 10 = 50
Outside the Loop
BUILD SUCCESSFUL (total time: 0 seconds)
```

© Aptech Ltd.

Looping Constructs/Session 5

9

Using slides 5 to 9, explain the 'while' statement.

Mention to the students that the `while` statement in Java is used to execute a statement or a block of statements, while a particular condition is true. Bring to the notice of the students that the condition is checked, before the statements are executed. The condition for the `while` statement can be any expression which returns a boolean value.

Explain to them the syntax of the `while` loop.

Tell them that when executing, if the *expression* result is true, then the actions inside the loop will be executed. This will continue as long as the expression result is true.

**Tips:**

The key point of the `while` loop is that the loop might not ever run.

Tell them that as shown in the code snippet, a variable of type integer, `num`, is declared to store the number. The variable `num` is initialized to 1 and used in the `while` loop to start multiplication from 1.

The condition `num <= 5` ensures that the `while` loop executes as long as the value in `num` is less than or equal to 5. The execution of the loop stops, when condition becomes false, that is, when the value of `num` reaches 6. The first statement within the body of the loop, calculates the value of product by multiplying `num` with 10. The next statement prints this value and the last statement within the body of the loop changes the value of `num` by incrementing it by 1.

**Tips:**

The key point of while loop is that when the expression is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

## Slides 10 and 11

Let us understand null statement in loops.

### Null Statement in Loops 1-2



- ◆ There are situations when it is required to write a loop without any action statement to delay a process.
- ◆ Such a loop is referred to as null statement loop.
- ◆ **Null statement loop:**
  - ◆ There are no statements in the body of the loop.
  - ◆ The loop is terminated with a semicolon.
- ◆ Following code snippet demonstrates a code that prints the midpoint of two numbers with an empty while loop:

```
public class TestWhileEmptyBody {
 /**
 * @param args the command line arguments
 */

 public static void main(String[] args) {
 int num1 = 1;
 int num2 = 30;
```

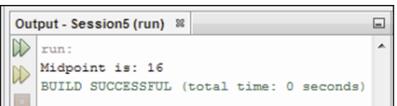
**Null Statement in Loops 2-2**



```
// An empty while loop with no statements
while (++num1 < --num2);

// The statement executes after the while loop is completed
System.out.println("Midpoint is: " + num1);
}
```

- ◆ The value of **num1** is incremented and the value of **num2** is decremented.
- ◆ The loop repeats till the value of **num1** is equal to or greater than **num2**.
- ◆ Thus, upon exit **num1** will hold a value that is midway between the original values of **num1** and **num2**.
- ◆ Following figure shows the output of the code:



Output - Session5 (run) ■  
run:  
Midpoint is: 16  
BUILD SUCCESSFUL (total time: 0 seconds)

© Aptech Ltd. Looping Constructs/Session 5 11

Using slides 10 and 11, explain the null statement in loops.

There are situations when it is required to write a loop without any action statement to delay a process. Such a loop is referred to as null statement loop.

#### Null statement loop:

- There are no statements in the body of the loop.
- The loop is terminated with a semicolon.

Explain the code snippet that demonstrates a code that prints the midpoint of two numbers with an empty `while` loop mentioned in the slides.

Tell them that a null statement is used to provide a null operation in situations where the grammar of the language requires a statement, but a program requires no work to be done.

#### Tips:

The most common use of null statement is in the loop operations in which all the loop activity is performed by the test portion of the loop. It is useful where a label is needed just before a brace that terminates compound statement.

#### In-Class Question:

After you finish explaining null statement, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



How null statement loop can be identified?

#### Answer:

- There are no statements in the body of the loop.
- The loop is terminated with a semicolon.

**Slide 12**

Let us understand rules for using 'while' loop.

### Rules for Using 'while' Loop

- ◆ The following points should be noted when using the `while` statement:
  - ◆ The value of the variables used in the expression must be set once before the execution of the loop. For example, `num = 1;`
  - ◆ The body of the loop must have an expression that changes the value of the variable which is a part of the loop's expression. For example, `num++;` or `num--;`

© Aptech Ltd.      Looping Constructs/Session 5      12

Using slide 12, explain rules for using 'while' loop.

Tell them that the rules to be followed while implementing the `while` statements are as follows:

- The values of the variables used in the expression must be set at some point, before the `while` loop is reached. This process is called the *initialization of variables* and has to be performed once, before the execution of the loop.
- The body of the loop must have an expression that changes the value of the variable that is a part of the loop's expression. A variable is said to be incremented if its value increases in the body of the loop, and is said to be decremented if its value decreases.

**Slide 13**

Let us understand infinite loop.

### Infinite Loop

- ◆ An infinite loop is one which never terminates.
- ◆ It runs infinitely when the conditional expression or the increment/decrement expression of the loop is missing.
- ◆ Following code snippet shows the implementation of an infinite loop using the `while` statement:

```
public class InfiniteWhileLoop {
 /**
 * @param args the command line arguments
 */
 public static void main(String[] args) {
 /*
 * Loop begins with a boolean value true and is executed
 * infinitely as the terminating condition is missing
 */
 while (true) {
 System.out.println("Welcome to Loops...");
 } //End of the while loop
 }
}
```

◆ The loop never terminates as the expression always returns a `true` value.

© Aptech Ltd.      Looping Constructs/Session 5      13

Using slide 13, explain infinite loop.

An infinite loop is one which never terminates. It runs infinitely when the conditional expression or the increment/decrement expression of the loop is missing.

Any type of loop can be infinite loop. The loop never terminates as the expression always returns a true value.

Explain the code snippet shows the implementation of an infinite loop using the while statement mentioned in slide 13. The condition is simply a boolean value **true**, which leads to an infinite loop.

Additionally, explain the code showing an infinite while loop.

```
int count = 0;
while(count < 100) {
 System.out.println("This goes on forever, HELP!!!");
 count = count + 10; \\Incrementing the value of count by 10.
 System.out.println("Count = " + count);
 count= count - 10; \\Decrementing the value of count by 10.
 System.out.println("Count = " + count);
}
```

Explain to them that in the code, the value of **count** is always **0**, which is less than **100**. So, the expression always returns a true value. Hence, the loop never ends.

A break statement can be used to terminate such programs. Thus, it will just go into the loop once and terminate, displaying the output as:

**This goes on forever, HELP!!!**

**Count = 10**

**Count = 0**

### Slides 14 to 17

Let us understand 'do-while' statement.

### 'do-while' Statement 1-4



- ◆ It checks the condition at the end of the loop rather than at the beginning.
- ◆ It ensures that the loop is executed at least once.
- ◆ It comprises a condition expression that evaluates to a boolean value.
- ◆ The syntax to use the do-while statement is as follows:

**Syntax**

```
do {
 statement(s);
} while (expression);
```

where,

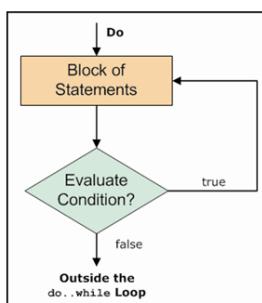
**expression:** A conditional expression which must return a boolean value, that is, true or false.

**statement (s):** Indicates body of the loop with a set of statements.

### 'do-while' Statement 2-4



- Following figure shows the flow of execution for the do-while loop:



For each iteration, the do-while loop first executes the body of the loop and then, the conditional expression is evaluated.

When the conditional expression evaluates to true, the body of the loop executes.

When the conditional expression evaluates to false, the loop terminates.

The statement following the loop is executed.

### 'do-while' Statement 3-4



- Following code snippet demonstrates the use of do-while loop for finding the sum of 10 numbers:

```

public class SumOfNumbers {
 /**
 * @param args the command line arguments
 */
 public static void main(String[] args) {
 int num = 1, sum = 0;

 /**
 * The body of the loop is executed first, then the condition is
 * evaluated
 */
 do {
 sum = sum + num;
 num++;
 } while (num <= 10);

 // Prints the value of variable after the loop terminates
 System.out.printf("Sum of 10 Numbers: %d\n", sum);
 }
}

```

### 'do-while' Statement 4-4



- Two integer variables, **num** and **sum** are declared and initialized to 1 and 0 respectively.
- The loop block begins with a **do** statement.
- The first statement in the body of the loop calculates the value of **sum** by adding the current value of **sum** with **num**.
- The next statement in the loop increments the value of **num** by 1.
- The condition, **num <= 10**, included in the **while** statement is evaluated.
- If the condition is met, the instructions in the loop are repeated.
- After the loop terminates, the value in the variable **sum** is printed.
- Following figure shows the output of the code:

Using slides 14 to 17, explain the 'do-while' statement.

Tell the students that the Java programming language also provides a **do-while** statement. Mention that the **do-while** statement checks the condition at the end of the loop rather than at the beginning, to ensure that the loop is executed at least once. The condition of the **do-while** statement usually comprises of an expression that evaluates to a boolean value.

Explain the syntax of **do-while** statement from slide.

**Tips:**

The `do-while` evaluates its expression at the bottom of the loop instead of the top.

The flow of execution for the `do-while` loop:

- For each iteration, the `do-while` loop first executes the body of the loop and then, the conditional expression is evaluated.
- When the conditional expression evaluates to true, the body of the loop executes.
- When the conditional expression evaluates to false, the loop terminates.
- The statement following the loop is executed.

For example, here's a loop that counts 1 to 10.

```
int number = 1;
do
{
 System.out.print(number + " ");
 number++;
} while (number <= 10);
```

In '`do-while`' loop, boolean expression appears at the end of the loop, so the statements in the loop execute once before the boolean is tested.

Tell them that in the code snippet, two integer variables, `num` and `sum` are declared and initialized to 1 and 0 respectively. The loop block begins with a `do` statement. The first statement in the body of the loop calculates the value of `sum` by adding the current value of `sum` with `num` and then, the next statement in the loop changes the value of `num` by incrementing it by 1.

Next, the condition, `num <= 10`, included in the `while` statement is evaluated. If the condition is met, the instructions in the loop are repeated. If the condition is not met (that is, when the value of `num` becomes 11), the loop terminates and the value in the variable, `sum` is printed.

The output will be, `Sum = 55`.

**Slides 18 to 22**

Let us understand 'for' statement.

### 'for' Statement 1-5

**'for' Statement**

- Used when the user knows the number of times the statements need to be executed.
- Statements within the body of the loop are executed as long as the condition is `true`.
- Condition is checked before the statements are executed.

© Aptech Ltd.      Looping Constructs/Session 5      18

## 'for' Statement 2-5



- The syntax to use the `for` statement is as follows:

### Syntax

```
for(initialization; condition; increment/decrement) {
 // one or more statements
}
```

where,

**initialization:** Is an expression that will set the initial value of the loop control variable.

**condition:** Is a `boolean` expression that tests the value of the loop control variable. If the condition expression evaluates to `true`, the loop executes, else terminates.

**increment/decrement:** Increments or decrements the value of control variable(s) in each iteration, till the condition specified in the condition section is reached. Typically, increment and decrement operators, such as `++`, `--`, and shortcut operators, such as `+=` or `-=` are used in this section.

## 'for' Statement 3-5



- Following figure shows the flow of execution for the `for` statement:

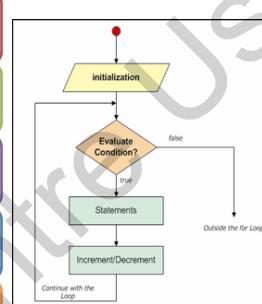
The initialization expression is executed only once, that is, when the loop starts.

Next, the `boolean` expression is evaluated and tests the loop control variable against a targeted value.

If the expression is `true`, then the body of the loop is executed and if the expression is `false`, then the loop terminates.

Lastly, the iteration portion of the loop is executed. This expression usually increments or decrements value of the control variable.

In the next iteration, again the condition section is evaluated and depending on the result of evaluation the loop is either continued or terminated.



## 'for' Statement 4-5



- Following code snippet demonstrates the use of `for` statement for displaying the multiples of 10:

```
public class PrintMultiplesWithForLoop {
 /*
 * @param args the command line arguments
 */
 public static void main(String[] args) {
 int num, product;

 // The for Loop with all the three declaration parts
 for (num = 1; num <= 5; num++) {
 product = num * 10;
 System.out.printf("\n %d * 10 = %d", num, product);
 } // Moves the control back to the for loop
 }
}
```

- In the initialization section of the `for` loop, the `num` variable is initialized to 1.
- The condition statement, `num <= 5`, ensures that the `for` loop executes as long as `num` is less than or equal to 5.
- The increment statement, `num++`, increments the value of `num` by 1.
- Finally, the loop terminates when the condition becomes `false`, that is, when the value of `num` becomes equal to 6.

**'for' Statement 5-5**

Following figure shows the output of the code:

```
Output - Session5 (run) ✘
run:
1 * 10 = 10
2 * 10 = 20
3 * 10 = 30
4 * 10 = 40
5 * 10 = 50 BUILD SUCCESSFUL (total time: 0 seconds)
```

© Aptech Ltd. Looping Constructs/Session 5 22

Using slides 18 to 22, explain the 'for' statement.

Tell them that a `for` loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

In other words, the `for` loops are especially used when the user knows how many times the statements need to be executed in the code block. It is similar to the `while` statement in its function. The statements within the body of the loop are executed as long as the condition is true. Here too, the condition is checked before the statements are executed.

Explain to the students the syntax as follows:

- **Initialization:** Initializes the variables that will be used in the condition.
- **Condition:** Comprises the condition that is tested, before the statements in the loop are executed.
- **Increment/decrement:** Comprises the statement that changes the value of the variable(s) to ensure that the condition specified in the section is reached. Typically, increment and decrement operators, such as `++`, `--`, and shortcut operators, such as `+=` or `-=` are used in this section. Note that there is **no semicolon** at the end of the increment/decrement expressions.

#### Tips:

A `for` loop is useful when you know how many times a task is to be repeated.

#### Working of the for loop

Tell them that the three declaration parts are separated by semicolons. When the loop starts, the initialization portion of the loop is executed. Generally, this is an expression that sets the value of the loop control variable and acts as a counter that controls the loop. The initialization expression is executed only once.

Next, point out that the boolean expression is evaluated. It usually tests the loop control variable against a targeted value. If the expression is true, then the body of the loop is executed and if the expression is false, then the loop terminates.

Lastly, tell them that the iteration portion of the loop is executed. This expression will usually increment or decrement the loop control variable.

In slide 21, you will explain the `for` statement using the code snippet. Tell them that in the initialization section of the `for` loop, the `num` variable is initialized to 1. The condition statement, `num <= 5`, ensures that the `for` loop executes as long as `num` is less than or equal to 5. The loop exits when the condition becomes false, that is, when the value of `num` becomes equal to 6. Finally, point out to the students that the increment statement, `num++` in the increment/decrement section of the `for` statement, increments the value of `num` by 1. The increment/decrement expression is evaluated after the first round of iteration.

The output will be as shown:

```
1 * 10 = 10
2 * 10 = 20
3 * 10 = 30
4 * 10 = 40
5 * 10 = 50
```

#### In-Class Question:

After you finish explaining for statement, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



How long the statements are executed in for loop?

#### Answer:

The statements within the body of the loop are executed as long as the condition is true.

#### Slides 23 and 24

Let us understand scope of control variable in 'for' statement.

#### Scope of Control Variable in 'for' Statement 1-2



##### ◆ Control Variables:

- ❖ Are used within the `for` loops and may not be used further in the program.
- ❖ It is possible to restrict the scope of variables by declaring them at the time of initialization.
- ❖ Following code snippet declares the counter variable inside the `for` statement:

```
public class ForLoopWithVariables {
 /**
 * @param args the command line arguments
 */
 public static void main(String[] args) {
 int product;
 // The counter variable, num is declared inside the for loop
 }
}
```

### Scope of Control Variable in 'for' Statement 2-2



```
for (int num = 1; num <= 5; num++) {
 product = num * 10;
 System.out.printf("\n %d * 10 = %d ", num, product);
} // End of the for loop
}
```

- In the code, the variable `num` has been declared inside the `for` statement.
- This restricts the scope of the variable, `num` to the `for` statement and completes when the loop terminates.

Using slides 23 and 24, explain scope of control variable in 'for' statement.

#### Control Variables

- Are used within the for loops and may not be used further in the program.
- It is possible to restrict the scope of variables by declaring them at the time of initialization.

A variable declared in for statement can only be used in that statement and in the body of the loop. The scope of variable extends from its declaration to the end of the block of the `for` statement, so it can be used in termination and increment expressions as well.

Explain the code snippet that declares the counter variable inside the `for` statement mentioned in slides 23 and 24.

#### Slides 25 to 27

Let us understand use of comma operator in 'for' statement.

### Use of Comma Operator in 'for' Statement 1-3



#### Expressions:

- The `for` statement can be extended by including more than one initialization or increment expressions.
- Expressions are separated by using the 'comma' (,) operator.
- Expressions are evaluated from left to right.
- The order of the evaluation is important, if the value of the second expression depends on the newly calculated value.
- Following code snippet demonstrates the use of `for` loop to print the addition table for two variables using the 'comma' operator:

```
public class ForLoopWithComma {
 /**
 * @param args the command line arguments
 */
 public static void main(String[] args) {
 int i, j;
 int max = 10;
```

### Use of Comma Operator in 'for' Statement 2-3



```
/*
 * The initialization and increment/decrement section includes
 * more than one variable
 */
for (i = 0, j = max; i <= max; i++, j--) {
 System.out.printf("\n%d + %d = %d", i, j, i + j);
}
```

- ◆ Three integer variables **i**, **j**, and **max** are declared.
- ◆ The variable **max** is assigned a value **10**. The **i** variable is assigned a value of **0** and **j** is assigned the value of **max**, that is, **10**. Thus, two parameters are initialized using a 'comma' operator.
- ◆ The condition statement, **i <= max**, ensures that the **for** loop executes as long as **i** is less than or equal to **max** that is **10**.
- ◆ Finally, the iteration expression again consists of two expressions, **i++**, **j--**.
- ◆ After each iteration, **i** is incremented by 1 and **j** is decremented by 1.
- ◆ The sum of these two variables which is always equal to **max** is printed.

### Use of Comma Operator in 'for' Statement 3-3



- ◆ Following figure shows the output of the code:

```
Output - Session5 (run) ✘
0 + 10 = 10
1 + 9 = 10
2 + 8 = 10
3 + 7 = 10
4 + 6 = 10
5 + 5 = 10
6 + 4 = 10
7 + 3 = 10
8 + 2 = 10
9 + 1 = 10
10 + 0 = 10BUILD SUCCESSFUL (total time: 0 seconds)
```

Using slides 25 to 27, explain use of comma operator in 'for' statement.

#### Expressions:

- The for statement can be extended by including more than one initialization or increment expressions.
- Expressions are separated by using the 'comma' (,) operator.
- Expressions are evaluated from left to right.
- The order of the evaluation is important, if the value of the second expression depends on the newly calculated value.

Explain the code snippet that demonstrates the use of for loop to print the addition table for two variables using the 'comma' operator mentioned in the slides.

Explain to them that in the code, three integer variables **i**, **j**, and **max** are declared. The variable, **max** is assigned a value **10**. Further, within the initialization section of the **for** loop, the **i** variable is assigned a value **0** and **j** is assigned the value of **max**, that is, **10**. Thus, two parameters are initialized using a 'comma' operator.

Bring to the notice of the student that the condition statement, **i <= max**, ensures that the **for** loop executes as long as **i** is less than or equal to **max** that is **10**. The loop exits when the condition becomes false, that is, when the value of **i** becomes equal to **11**.

Finally, explain that the iteration expression again consists of two expressions, **i++**, **j--**. After each iteration, **i** is incremented by 1 and **j** is decremented by 1. The sum of these two variables which is always equal to **max** is printed.

#### Tips:

Any or all expressions in the `for` loop may be left blank. If all the three expressions are left blank, an infinite loop will be created.

### Slides 28 and 29

Let us understand variation in 'for' loop.

**Variation in 'for' Loop 1-2**

- The most common variation involves the conditional expression which can be:
  - Tested with the targeted values, but, it can also be used for testing **boolean** expressions.
- Alternatively, the initialization or the iteration section in the `for` loop may be left empty, that is, they need not be present in the `for` loop.
- Following code snippet demonstrates the use of `for` loop without the initialization expression:

```
public class ForLoopWithNoInitialization {
 public static void main(String[] args) {
 /*
 * Counter variable declared and initialized outside for loop
 */
 int num = 1;
 /*
 * Boolean variable initialized to false
 */
 boolean flag = false;
```

© Aptech Ltd.      Looping Constructs/Session 5      28

**Variation in 'for' Loop 2-2**

```
/*
 * The for loop starts with num value 1 and continues till value of
 * flag is not true
 */
for (; !flag; num++) {
 System.out.println("Value of num: " + num);
 if (num == 5) {
 flag = true;
 }
} // End of for loop
```

- The `for` loop in the code continues to execute till the value of the variable **flag** is set to **true**.
- Following figure shows the output of the code:

© Aptech Ltd.      Looping Constructs/Session 5      29

Using slides 28 and 29, explain variation in 'for' loop.

The most common variation involves the conditional expressions which can be:

- Tested with the targeted values, but, it can also be used for testing **boolean** expressions.

Alternatively, the initialization or the iteration section in the `for` loop may be left empty, that is, they need not be present in the `for` loop.

Explain the code snippet that demonstrates the use of `for` loop without the initialization expression mentioned in the slides.

**Slide 30**

Let us understand infinite 'for' loop.

### Infinite 'for' Loop



- ◆ If all the three expressions are left empty, then it will lead to an infinite loop.
- ◆ The infinite for loop will run continuously because there is no terminating condition.
- ◆ Following code snippet demonstrates the code for the infinite for loop:

```
.....
for(; ;) {
 System.out.println("This will go on and on");
}
.....
```

- ◆ The code will print 'This will go on and on' until the loop is terminated manually.
- ◆ Infinite loops make the program run indefinitely for a long time resulting in the consumption of all resources and stopping the system.

© Aptech Ltd.      Looping Constructs/Session 5      30

Using slide 30, explain the Infinite 'for' Loop.

If all the three expressions are left empty, then it will lead to an infinite loop. The infinite for loop will run continuously because there is no terminating condition. Infinite loops make the program run indefinitely for a long time resulting in the consumption of all resources and stopping the system.

An infinite loop is a sequence of instructions which loops endlessly.

**Slides 31 and 32**

Let us understand enhanced 'for' loop.

### Enhanced 'for' Loop 1-2



- ◆ It is designed to retrieve or traverse through a collection of objects, such as an array.
- ◆ It is also used to iterate over the elements of the collection objects, such as `ArrayList`, `LinkedList`, `HashSet`, and so on defined in the collection framework.
- ◆ It continues till all the elements from a collection are retrieved.
- ◆ The syntax for using the enhanced for loop is as follows:

**Syntax**

```
for (type var: collection) {
 // block of statement
}
```

where,

`type`: Specifies the type of collection that is traversed.  
`var`: Is an iteration variable that stores the elements from the collection.

© Aptech Ltd.      Looping Constructs/Session 5      31

### Enhanced 'for' Loop 2-2



- Following table shows the method for retrieving elements from an array object using enhanced `for` loop and its equivalent `for` loop:

| for Loop                                                                                   | Enhanced for Loop                                                           |
|--------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------|
| <pre>type var; for (int i = 0; i &lt; arr.length; i++) {     var = arr[i];     ... }</pre> | <pre>for (type var : arr) {     ...     // Body of the loop     ... }</pre> |

© Aptech Ltd.

Looping Constructs/Session 5

32

Using slides 31 and 32, explain the enhanced 'for' loop.

It is designed to retrieve or traverse through a collection of objects, such as an array. It is also used to iterate over the elements of the collection objects, such as `ArrayList`, `LinkedList`, `HashSet`, and so on defined in the collection framework. It continues till all the elements from a collection are retrieved.

Then, explain the syntax for using the enhanced for loop. Assuming that in the syntax, `var` is defined to be an array of `String` objects, then each element is assigned to the `var` variable as it loops through the array.

**Tips:**

The enhanced for loop was introduced in Java 5 as a simpler way to iterate through all the elements of a collection. Enhanced for loops are simple, but inflexible. They can be used when you wish to step through the elements of the array in first-to-last order, and you do not need to know the index of the current element. In all other cases, the "standard" for loop should be preferred.

Then, explain how to retrieve the elements of an array using standard for as well as for-each loop discussed on slide 32.

The code can be completed to execute both the loops as shown:

- The usual way to step through all the elements of an array in order is with a standard `for` loop will be as follows:

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

- The enhanced `for` loop is a simpler way of retrieving elements from the array as follows:

```
for (int myValue : myArray) {
 System.out.println(myValue);
}
```

**Slides 33 to 35**

Let us understand nested-for loop.

**Nested Loop 1-3**

- ◆ The placing of a loop statement inside the body of another loop statement is called nesting of loops.
- ◆ There can be any number of combinations between the three types of loops.
- ◆ The most commonly nested loops are formed by `for` statements which can be nested within another `for` loop forming nested-for loop.
- ◆ Following code snippet demonstrates the use of a nested-for loop for displaying a pattern:

```
public class DisplayPattern {
 /**
 * param args the command line arguments
 */
 public static void main(String[] args) {
 int row, col;
 // The outer for loop executes 5 times
 for (row = 1; row <= 5; row++) {
 for (col = 1; col <= row; col++) {
 System.out.print("* ");
 }
 System.out.println();
 }
 }
}
```

© Aptech Ltd.

Looping Constructs/Session 5

33

**Nested Loop 2-3**

- ```
/*
 * For each iteration, the inner for loop will execute from col = 1
 * and will continue, till the value of col is less than or equal to row
 */
for (col = 1; col <= row; col++) {
    System.out.print(" * ");
} // End of inner for loop
System.out.println();
} // End of outer for loop
}
```
- ◆ The outer for loop starts with the counter variable `row` whose initial value is set to 1.
 - ◆ As the condition, `row < 5` is evaluated to `true`, the body of the outer for loop gets executed.
 - ◆ The body contains an inner for loop which starts with the counter variable's value `col` set to 1.

© Aptech Ltd.

Looping Constructs/Session 5

34

Nested Loop 3-3

- ◆ The iteration of the inner loop executes till the value of `col` is less than or equal to the value of `row` variable.
- ◆ Once the value of `col` is greater than `row`, the inner `for` loop terminates.
- ◆ For each iteration of the outer loop, the inner `for` loop is reinitialized and continues till the condition evaluates to `false`.
- ◆ Following figure shows the output of the code:

```
Output - Session5 (run)
run:
*
**
***
****
*****
BUILD SUCCESSFUL (total time: 1 second)
```

© Aptech Ltd.

Looping Constructs/Session 5

35

Using slides 33 to 35, explain the nested loop.

The placing of a loop statement inside the body of another loop statement is called nesting of loops. While nesting two loops, outer loop takes control of the inner loop. There can be any number of

combinations between the three types of loops. The most commonly nested loops are formed by for statements which can be nested within another for loop forming nested-for loop.

Explain the code snippet that demonstrates the use of a nested-for loop for displaying a pattern mentioned in the slides. Explain them in the nested loops, the outer loop changes only after the inner loop is completely finished.

Slide 36

Let us understand comparison of loops.

Comparison of Loops	
Following table lists the differences between while/for and do-while loops:	
while/for	do-while
Loop is pre-tested. The condition is checked before the statements within the loop are executed.	Loop is post-tested. The condition is checked after the statements within the loop are executed.
The loop does not get executed if the condition is not satisfied at the beginning.	The loop gets executed at least once even if the condition is not satisfied at the beginning.

Using slide 36, explain comparison of loops.

Explain to them that the type of loop, that is chosen while writing a program, depends on the good programming practice.

Mention that a loop written using the `while` statement can also be rewritten using the `for` statement and vice versa. The `do-while` statement can be rewritten using the `while` or the `for` statement. However, this is not advisable because the `do-while` statement is executed at least once. When the number of times the statements within the loop should be executed is known, the `for` statement is used.

Point out the differences between `while/for` and `do-while` loops:

`while/for`

- Loop is pre-tested. The condition is checked before the statements within the loop, are executed.
- The loop does not get executed if the condition is not satisfied at the beginning.

`do-while`

- Loop is post-tested. The condition is checked after the statements within the loop, are executed.
- The loop gets executed at least once even if the condition is not satisfied at the beginning.

In-Class Question:

After you finish explaining the loops in Java, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



How do you write an infinite loop using the `for` statement?

Answer:

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



Which of the loop statement is similar to the `while` statement, but evaluates its expression at the **bottom** of the loop?

Answer:

The `do-while` statement is similar to the `while` statement, but evaluates its expression at the bottom of the loop.

Slide 37

Let us understand jump statements.

Jump Statements

Java provides two keywords: `break` and `continue` that are used within loops to change the flow of control based on conditions.

© Aptech Ltd. Looping Constructs/Session 5 37

Using slide 37, explain the jump statements.

Tell the students that at times, the exact number of times the loop has to be executed is known only during runtime. In such a case, the condition to terminate the loop can be enclosed within the body of the loop. At some times, based on a condition, the remaining statements present in the body of the loop need to be skipped.

Java supports jump statements that unconditionally transfer control to locations within a program known as *target* of jump statements.

Java provides two keywords, `break` and `continue` that serve diverse purposes. However, both are used with loops to change the flow of control based on conditions.

Slides 38 to 40

Let us understand 'break' statement.

'break' Statement 1-3

- ◆ It can be used to terminate a case in the switch statement.
- ◆ It forces immediate termination of a loop, bypassing the loop's normal conditional test.
- ◆ When the break statement is encountered inside a loop, the loop is immediately terminated and the program control is passed to the statement following the loop.
- ◆ If used within a set of nested loops, the break statement will terminate the innermost loop.
- ◆ Following code snippet demonstrates the use of break statement:

```
public class AcceptNumbers {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        int cnt, number; // cnt variable is a counter variable
    }
}
```

© Aptech Ltd.

Looping Constructs/Session 5

38

'break' Statement 2-3

- ```
for (cnt = 1, number = 0; cnt <= 10; cnt++) {
 // Scanner class is used to accept data from the keyboard
 Scanner input = new Scanner(System.in);
 System.out.println("Enter a number:");
 number = input.nextInt();

 if (number == 0) {
 // break statement terminates the loop
 break;
 } // End if statement
} // End of for statement
}
```
- ◆ In the code, the user is prompted to enter a number, and this is stored in the variable, **number**.
  - ◆ However, if the user enters the number zero, the loop terminates and the control is passed to the next statement after the loop.

© Aptech Ltd.

Looping Constructs/Session 5

39

**'break' Statement 3-3**

- ◆ Following figure shows the output of the code:

```
Output - Session5 (run) 8
run:
Enter a number:
8
Enter a number:
6
Enter a number:
3
Enter a number:
0
BUILD SUCCESSFUL (total time: 8 seconds)
```

© Aptech Ltd.

Looping Constructs/Session 5

40

Using slides 38 to 40, explain the 'break' Statement.

Tell them that the **break** keyword is used to stop the entire loop. The **break** keyword must be used inside any loop or a **switch** statement.

Explain that the `break` statement in Java is used in two ways. First, it can be used to terminate a case in the `switch` statement. Second, it forces immediate termination of a loop, bypassing the loop's normal conditional test.

When the `break` statement is encountered inside a loop, the loop is immediately terminated and the program control is passed to the statement following the loop.

Explain the use of `break` as shown in the code snippet that demonstrates the use of `break` statement in the slides.

### Slides 41 and 42

Let us understand the 'continue' statement.

**'continue' Statement 1-2**

- ◆ It skips statements within a loop and proceeds to the next iteration of the loop.
- ◆ In while and do-while loops, a `continue` statement transfers the control to the conditional expression which controls the loop.
- ◆ Following code snippet demonstrates the code that uses `continue` statement in printing the square and cube root of a number:

```
public class NumberRoot {
 /**
 * @param args the command line arguments
 */

 public static void main(String[] args) {
 int cnt, square, cube;
 // Loop continues till the remainder of the division is 0

 for (cnt = 1; cnt < 300; cnt++) {
 if (cnt % 3 == 0) {
 continue;
 }
 }
 }
}
```

© Aptech Ltd.      Looping Constructs/Session 5      41

**'continue' Statement 2-2**

```
square = cnt * cnt;
cube = cnt * cnt * cnt;
System.out.printf("\nSquare of %d is %d and Cube is %d", cnt, square,
cube);
} // End of the for loop
}
```

- ◆ The code declares a variable `cnt` and uses the `for` statement which contains the initialization, termination, and increment expression.
- ◆ The value of `cnt` is divided by 3 and the remainder is checked.
- ◆ If the remainder is 0, the `continue` statement is used to skip the rest of the statements in the body of the loop.
- ◆ If remainder is not 0, the `if` statement evaluates to `false`, and the square and cube of `cnt` is calculated and displayed.
- ◆ Following figure shows the output of the code:

© Aptech Ltd.      Looping Constructs/Session 5      42

Using slides 41 and 42, explain the 'continue' statement.

Tell them that the `continue` keyword can be used in any of the loop control structures. It causes the loop to immediately jump to the next iteration of the loop.

Mention the following points:

- In a `for` loop, the `continue` keyword causes flow of control to immediately jump to the update statement.

- In a `while` loop or `do/while` loop, flow of control immediately jumps to the boolean expression.

Explain the code snippet. Tell them that the code declares a variable, `cnt` and uses the `for` statement which contains the initialization, termination, and increment expression. In the body of the loop, the value of `cnt` is divided by 3 and the remainder is checked. If the remainder is 0, the `continue` statement is used to skip the rest of the statements in the body of the loop. If remainder is not 0, then the `if` statement evaluates to false, square and cube of `cnt` is calculated and displayed.

### Slides 43 to 47

Let us understand the labeled statements.

### Labeled Statements 1-5

Java defines an expanded form of `break` and `continue` statements referred to as labeled statements.

**Labeled Statements:**

- Are expanded forms that can be used within any block that must be part of a loop or a switch statement.
- Can be used to precisely specify the point from which the execution should resume.
- Can be used to exit from a set of nested blocks.

The syntax to declare the labeled `break` statement is as follows:

**Syntax**

```
break label;
```

Where,

`label:` Is an identifier specified to put a name to a block. It can be any valid Java identifier followed by a colon.

© Aptech Ltd.      Looping Constructs/Session 5      43

### Labeled Statements 2-5

Following code snippet demonstrates the use of labeled `break` statement:

```
public class TestLabeledBreak {
 /**
 * @param args the command line arguments
 */
 public static void main(String[] args) {
 int i;

 outer:
 for (i = 0; i < 5; i++) {
 if (i == 2) {
 System.out.println("Hello");
 // Break out of outer loop
 break outer;
 }
 System.out.println("This is the outer loop.");
 }

 System.out.println("Good - Bye");
 }
}
```

© Aptech Ltd.      Looping Constructs/Session 5      44

### Labeled Statements 3-5



- In the code, the loop will execute for five times.
- The first two times it displays the sentence 'This is the outer loop'.
- In the third round of iteration the value of **i** is set to 2 and prints 'Hello'.
- Next, the **break** statement is encountered and the control passes to the label named **outer**:
- Thus, the loop terminates and the last statement is printed.
- Following figure shows the output of the code:

```
Output - Session5 (run) *
run:
This is the outer loop.
This is the outer loop.
Hello
Good - Bye
BUILD SUCCESSFUL (total time: 1 second)
```

© Aptech Ltd.

Looping Constructs/Session 5

45

### Labeled Statements 4-5



- Labeled continue Statement:**
  - Similar to labeled **break** statement, you can specify a label to enclose a loop that continues with the next iteration of the loop.
  - This is done using labeled **continue** statement.
- Following code snippet demonstrates the use of labeled **continue** statement:

```
public class NumberPyramid {
 /**
 * @param args the command line arguments
 */
 public static void main(String[] args) {
 outer:
 for (int i = 1; i < 5; i++) {
 for (int j = 1; j < 5; j++) {
 if (j > i) {
 System.out.println();
 } /* Terminates the loop counting j and continues the
 * next iteration of the loop counting I */
 continue outer;
 } // End of if statement
 }
 }
}
```

© Aptech Ltd.

Looping Constructs/Session 5

46

### Labeled Statements 5-5



```
System.out.print(j);
} // End of inner for loop
System.out.println("\nThis is the outer loop.");
} // End of outer for loop
System.out.println("Good-Bye");
}
```

- Following figure shows the output of the code:

```
Output - Session5 (run) *
run:
1
12
123
1234
This is the outer loop.
Good-Bye
BUILD SUCCESSFUL (total time: 0 seconds)
```

© Aptech Ltd.

Looping Constructs/Session 5

47

Using slides 43 to 47, explain the labeled statements.

Tell them that a labeled statement is used only in case of nested loops.

### **Labeled break Statement**

It is used to indicate the nested loop that is to be continued with the next iteration or the nested loop to break from. A `break` keyword, when used with a label, exits out of the labeled loop.

Tell them that in the code snippet, the loop is supposed to be executed five times. The first two times it displays the sentence, 'This is the outer loop'. In the third pass, the value of `i` is equal to 2, thus it enters the `if` statement and prints the message, 'Hello'.

Next, the `break` statement is encountered and the control passes to the label, `outer:`. Thus, the loop is terminated and the last statement is printed.

Mention that the output of the code will be:

```
This is the outer loop.
This is the outer loop.
Hello
Good-Bye
```

### **Labeled continue Statement**

Similar to labeled `break` statement, where user can specify a label to enclose a loop that continues with the next iteration of the loop.

Explain the code snippet demonstrates the use of labeled `continue` statement mentioned in slides 46 and 47.

#### **In-Class Question:**

After you finish explaining, the jump statements in Java, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



Use of which jump statement will allow the programmer to transfer the control to the statement outside the loop?

#### **Answer:**

Break

**Slide 48**

Let us summarize the session.

**Summary**



- ◆ Loops enable programmers to develop concise programs, which otherwise would require thousands of lines of program statements.
- ◆ The loop statements supported by Java are namely, while, do-while, and for.
- ◆ The while loop is used to execute a statement or a block of statements until the specified condition is true.
- ◆ The do-while statement checks for condition at the end of the loop rather than at the beginning to ensure that the loop is executed at least once.
- ◆ The for loop is especially used when the user knows the number of times the statements need to be executed in the code block of the loop. The three parts of for statement are initialization, condition, increment/decrement.
- ◆ The placing of a loop in the body of another loop is called nesting.
- ◆ Java provides two keywords namely, break and continue that serve diverse purposes. However, both are used with loops to change the flow of control.

© Aptech Ltd. Looping Constructs/Session 5 48

In slide 48, you will summarize the session. End the session with a brief summary of what has been taught in the session.

### 5.3 Post Class Activities for Faculty

You should familiarize yourself with the topics of the next session which is based on classes and objects in Java.

**Tips:**

You can also check the Articles/Blogs/Expert Videos uploaded on the Online Varsity site to gain additional information related to the topics covered in the next session. You can also connect to online tutors on the Online Varsity site to ask queries related to the sessions.

# Session 6 – Classes and Objects

## 6.1 Pre-Class Activities

Before you commence the session, you should familiarize yourself with the topics of this session in-depth.

Here, you can discuss the key points with the students that were covered in the previous session. Prepare a question or two which will help you to relate the current session objectives.

### 6.1.1 Objectives

By the end of this session, the learners will be able to:

- Explain the process of creation of classes in Java
- Explain the instantiation of objects in Java
- Explain the purpose of instance variables and instance methods
- Explain constructors in Java
- Explain the memory management in Java
- Explain object initializers

### 6.1.2 Teaching Skills

To teach this session, you should be well-versed with creation of classes in Java and the constructors in Java. Also familiarize yourself with the purpose of instance variables, instance methods, and object initializers in Java.

You should teach the concepts in the theory class using the images provided. For teaching in the class, you are expected to use slides and LCD projectors.

#### Tips:

It is recommended that you test the understanding of the students by asking questions in between the class.

#### In-Class Activities:

Follow the order given here during In-Class activities.

**Overview of the Session:**

Give the students the overview of the current session in the form of session objectives. Show the students slide 2 of the presentation.

| Objectives                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <br><ul style="list-style-type: none"> <li>◆ Explain the process of creation of classes in Java</li> <li>◆ Explain the instantiation of objects in Java</li> <li>◆ Explain the purpose of instance variables and instance methods</li> <li>◆ Explain constructors in Java</li> <li>◆ Explain the memory management in Java</li> <li>◆ Explain object initializers</li> </ul> |

© Aptech Ltd. Classes and Objects/Session 6 2

Tell them that Java is an object-oriented programming language used for developing various types of applications that can be executed on any platform. Explain to the student that this session will familiarize them with the concept of classes and objects. This session explains them the instance variables, instance methods, constructor methods, methods defined in the class. It also gives a brief explanation of object initializers.

**6.2 In-Class Explanations****Slide 3**

Let us understand the concept of class in Java.

| Introduction                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <br><ul style="list-style-type: none"> <li>◆ <b>Class in Java:</b> <ul style="list-style-type: none"> <li>◆ Is the prime unit of execution for object-oriented programming in Java.</li> <li>◆ Is a logical structure that defines the shape and nature of an object.</li> <li>◆ Is defined as a new data type that is used to create objects of its type.</li> <li>◆ Defines attributes referred to as fields that represents the state of an object.</li> </ul> </li> </ul> <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"> <b>Conventions to be followed while naming a class</b> <ul style="list-style-type: none"> <li>• Class declaration should begin with the keyword class followed by the name of the class.</li> <li>• Class name should be a noun.</li> <li>• Class name can be in mixed case, with the first letter of each internal word capitalized.</li> <li>• Class name should be simple, descriptive, and meaningful.</li> <li>• Class name cannot be Java keywords.</li> <li>• Class name cannot begin with a digit. However, they can begin with a dollar (\$) symbol or an underscore character.</li> </ul> </div> |

© Aptech Ltd. Classes and Objects/Session 6 3

Using slide 3, explain the concept of class in Java.

Class is the prime unit of execution for object-oriented programming in Java. The class is a logical construct that defines the shape and nature of an object. Each object created from the class contains its own copy of the attributes defined in the class.

Everything in Java is contained in the classes. It is a collection of data members and functions where data members are variable that are declared inside a class and member functions are the functions or methods defined inside the class.

Next, you explain to them the rules for declaring a class. Tell them that:

- A class declaration should contain the keyword `class` and the name of the class that is being declared.
- Class name should be a noun.
- Class name can be in mixed case, with the first letter of each internal word capitalized.
- Class name should be simple, descriptive, and meaningful.
- Class names cannot be Java keywords.
- Class names cannot begin with a digit. However, they can begin with a dollar (\$) symbol or an underscore character (\_).

#### Slides 4 and 5

Let us understand declaring a class.

### Declaring a Class 1-2

The syntax to declare a class in Java is as follows:

```
class <class_name> {
 // class body
}
```

The body of the class is enclosed between the curly braces {}.

In the class body, you can declare members, such as fields, methods, and constructors.

Following figure shows the declaration of a sample class:

```
class Student {
 String studentName;
 int studentAge;

 void initialise() {
 studentName = "James Anderson";
 studentAge = 24;
 }

 void display() {
 System.out.println("Student Name: " + studentName);
 System.out.println("Student Age: " + studentAge);
 }
}

public static void main(String[] args) {
 Student objStudent = new Student();
 objStudent.initialise();
 objStudent.display();
}
```

Fields or Instance Variables: studentName, studentAge

Functions or Instance Methods: initialise(), display()

© Aptech Ltd. Classes and Objects/Session 6 4

### Declaring a Class 2-2

Following code snippet shows the code for declaring a class **Customer**:

```
class Customer {
 // body of class
}
```

In the code:

- A class is declared that acts as a new data type with the name **Customer**.
- It is just a template for creating multiple objects with similar features.
- It does not occupy any memory.

**Creating Objects:**

- Objects are the actual instances of the class.

© Aptech Ltd. Classes and Objects/Session 6 5

Using slides 4 and 5, explain how to declare a class in Java.

Explain to them the syntax for declaring a class.

Mention to the students that this is a *class declaration*. The *class body* that is the area between the braces. It contains all the code that provides for the life cycle of the objects created from the class.

In other words, it contains the constructors for initializing new objects, declarations for the fields that provide the state of the class and its objects, and methods to implement the behavior of the class and its objects.

After explaining the syntax for declaring a class, explain to them the code snippet. Specify that the code declares a class named **Customer** and within the braces, you need to define the variables and methods of the class.

#### Tips:

The Javadoc comments for a class are as shown:

```
/**
 * class description
 *
 * @version 1.0
 * @authorName Robert Robbsen
 */
```

`@version` adds 'Version' subheading, for classes and interfaces, with version-text to the generated docs when the `-version` option is used with **Javadoc** utility. This tag holds the current version number of the software.

`@authorName` adds author entry with name-text specified to the generated docs.

#### Slides 6 and 7

Let us understand declaring and creating an object.

**Declaring and Creating an Object 1-2**

- ◆ An object is created using the `new` operator.
- ◆ On encountering the `new` operator:
  - ◆ JVM allocates memory for the object.
  - ◆ Returns a reference or memory address of the allocated object.
  - ◆ The reference or memory address is then stored in a variable called as reference variable.
- ◆ The syntax for creating an object is as follows:

**Syntax**

`<class_name> <object_name> = new <class_name> ();`

where,

`new`: Is an operator that allocates the memory for an object at runtime.  
`object_name`: Is the variable that stores the reference of the object.

6

## Declaring and Creating an Object 2-2



- Following code snippet demonstrates the creation of an object in a Java program:

```
Customer objCustomer = new Customer();
```

- The expression on the right side, `new Customer()` allocates the memory at runtime.
- After the memory is allocated for the object, it returns the reference or address of the allocated object, which is stored in the variable, `objCustomer`.

Using slides 6 and 7, explain declaring and creating an object of the class in Java.

Mention that an object is created using the `new` keyword. On encountering the `new` keyword, the JVM allocates memory for the object, invokes the constructor of the class, and returns a reference of that allocated memory. The reference is assigned to the object.

Next, explain the syntax for creating an object as shown:

Explain the code snippet. Tell them that it creates an object for the class **Customer** named, **objCustomer**.

### Tips:

Explain the following steps for creating an object from a class:

- Declaration:** A variable declaration with a variable name with an object type.
- Instantiation:** The `new` keyword is used to create the object.
- Initialization:** The `new` keyword is followed by a call to a constructor. This call initializes the new object.

### Slides 8 to 10

Let us understand creation of an object.

## Creation of an Object: Two Stage Process 1-3



- Alternatively, an object can be created using two steps that are as follows:

- Declaration of an object reference.
- Dynamic memory allocation of an object.

- Declaration of an object reference:**

- The syntax for declaring the object reference is as follows:

### Syntax

```
<class_name> <object_name>;
```

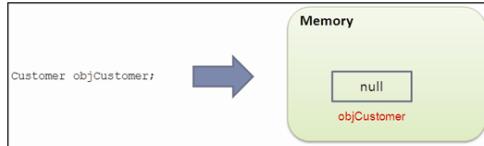
where,

`object_name`: Is just a variable that will not point to any memory location.

### Creation of an Object: Two Stage Process 2-3



- Following figure shows the effect of the statement, `Customer objCustomer;` which declares a reference variable:



- By default, the value `null` is stored in the object's reference variable which means reference variable does not point to an actual object.
- If `objCustomer` is used at this point of time, without being instantiated, then the program will result in a compile time error.

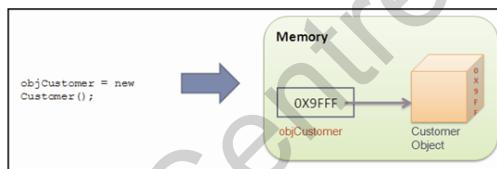
### Creation of an Object: Two Stage Process 3-3



◆ **Dynamic memory allocation of an object:**

- The object should be initialized using the `new` operator which dynamically allocates memory for an object.
- For example, the statement, `objCustomer = new Customer();` allocates memory for the object and memory address of the allocated object is stored in the variable `objCustomer`.

- Following figure shows the creation of object in the memory and storing of its reference in the variable, `objCustomer`:



Using slides 8 to 10, explain declaring and creating an object.

Discuss with the students that an object can be declared without using the `new` operator as shown:  
`<class_name> <object_name>;`

However, in this case, the object `object_name` will not point to any memory location and memory will not be allocated. Using an object, created without using the `new` operator, in the program will result in compile time error. Before using such an object, the object should be initialized using the `new` operator.

Explain the figure which shows the effect of the statement, `Customer objCustomer;` which declares a reference variable as mentioned in slide 9.

By default, the value `null` is stored in the object's reference variable which means reference variable does not point to an actual object.

#### **Dynamic memory allocation of an object:**

The object should be initialized using the `new` operator which dynamically allocates memory for an object.

The `new` operator instantiates a class by allocating memory to the object and returning a reference to that memory. The `new` operator returns a reference to an object it created.

The reference returned by a `new` operator does not have to be assigned to a variable. It can also be used directly in an expression.

Explain the figure that shows the creation of object in the memory and storing its reference in the variable, `objCustomer` as mentioned in slide 10.

### Slide 11

Let us understand members of a class.

**Members of a Class**

- ◆ The members of a class are fields and methods.

|                                                                                                                                                         |
|---------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Fields</b>                                                                                                                                           |
| <ul style="list-style-type: none"> <li>• Define the state of an object created from the class.</li> <li>• Referred to as instance variables.</li> </ul> |

|                                                                                                                                        |
|----------------------------------------------------------------------------------------------------------------------------------------|
| <b>Methods</b>                                                                                                                         |
| <ul style="list-style-type: none"> <li>• Implement the behavior of the objects.</li> <li>• Referred to as instance methods.</li> </ul> |

© Aptech Ltd. Classes and Objects/Session 6 11

Using slide 11, explain members of a class.

The members of a class are fields and methods.

- **Fields** - It defines the state of an object created from the class. It is referred to as instance variables.
- **Methods** - Methods implement the behavior of the objects. It is referred to as instance methods.

### Slides 12 to 18

Let us understand instance variables.

**Instance Variables 1-7**

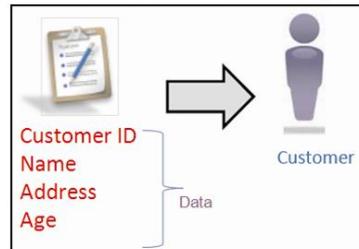
- ◆ They are used to store data in them.
- ◆ They are called instance variables because each instance of the class, that is, object of that class will have its own copy of the instance variables.
- ◆ They are declared similar to local variables.
- ◆ They are declared inside a class, but outside any method definitions.
- ◆ For example: Consider a scenario where the `Customer` class represents the details of customers holding accounts in a bank.
  - ◆ A typical question that can be asked is 'What are the different data that are required to identify a customer in a banking domain and represent it as a single object?'.

© Aptech Ltd. Classes and Objects/Session 6 12

### Instance Variables 2-7



- Following figure shows a **Customer** object with its data requirement:



- The identified data requirements for a bank customer includes: Customer ID, Name, Address, and Age.
- To map these data requirements in a **Customer** class, **instance variables** are declared.

© Aptech Ltd.

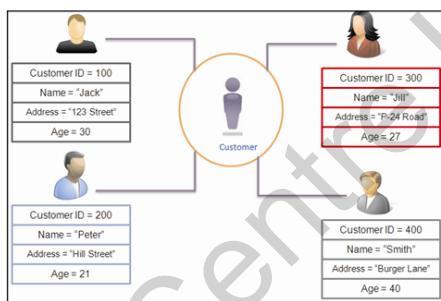
Classes and Objects/Session 6

13

### Instance Variables 3-7



- Each instance created from the **Customer** class will have its own copy of the instance variables.
- Following figure shows various instances of the class with their own copy of instance variables:



© Aptech Ltd.

Classes and Objects/Session 6

14

### Instance Variables 4-7



- The syntax to declare an instance variable within a class is as follows:

#### Syntax

```
[access_modifier] data_type instanceVariableName;
```

where,

access\_modifier: Is an optional keyword specifying the access level of an instance variable. It could be **private**, **protected**, and **public**.

data\_type: Specifies the data type of the variable.

instanceVariableName: Specifies the name of the variable.

- Instance variables are accessed by objects using the dot operator ( . ).

© Aptech Ltd.

Classes and Objects/Session 6

15

### Instance Variables 5-7



- Following code snippet demonstrates the declaration of instance variables within a class in the Java program:

```

1: public class Customer {
2: // Declare instance variables
3: int customerID;
4: String customerName;
5: String customerAddress;
6: int customerAge;
7: /* As main() method is a member of class, so it can access other
8: * members of the class */
9: public static void main(String[] args) {
10: // Declares and instantiates an object of type Customer
11: Customer objCustomer1 = new Customer();

```

- Lines 3 to 6 declares instance variables.
- Line 11 creates an object of type **Customer** and stores its reference in the variable, **objCustomer1**.

© Aptech Ltd.

Classes and Objects/Session 6

16

### Instance Variables 6-7



```

12: // Accesses the instance variables to store values
13: objCustomer1.customerID = 100;
14: objCustomer1.customerName = "John";
15: objCustomer1.customerAddress = "123 Street";
16: objCustomer1.customerAge = 30;

17: // Displays the objCustomer1 object details
18: System.out.println("Customer Identification Number: " +
19: objCustomer1.customerID);
20: System.out.println("Customer Name: " + objCustomer1.customerName);
21: System.out.println("Customer Address: " + objCustomer1.
customerAddress);
22: System.out.println("Customer Age: " + objCustomer1.customerAge);
}

```

- Lines 13 to 16 accesses the instance variables and assigns them the values.
- Lines 18 to 21 display the values assigned to the instance variables for the object, **objCustomer1**.

© Aptech Ltd.

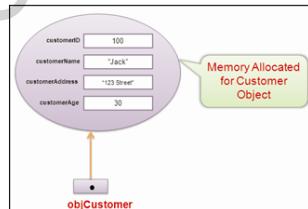
Classes and Objects/Session 6

17

### Instance Variables 7-7



- Following figure shows the allocation of **Customer** object in the memory:



- Following figure shows the output of the code:

```

Output - Session 6 (run) ⑧
run:
Customer Identification Number: 100
Customer Name: John
Customer Address: 123 Street
Customer Age: 30
BUILD SUCCESSFUL (total time: 1 second)

```

© Aptech Ltd.

Classes and Objects/Session 6

18

Using slides 12 to 18, explain instance variables.

Consider a scenario, wherein a car dealer wants to keep track of the price of various cars in stock. So, to store the prices of various cars, you need variables. Accordingly, you need several local variables to store prices. However, an alternative is to create a class and declare a variable named, **price** in it. This enables every instance created from this class to have its own copy of price variable referred to as *instance variable*.

Tell them that the figure from the slide demonstrates the instances of class with their own copy of instance variable.

**Tips:**

The benefit of using instance variables over local variables is that, you declare only one instance variable and use it for all instances of the class. The name of instance variable is shared across all instances, but each instance has its own copy of instance variable.

First, bring to the notice of the students that there are several kinds of variables:

- **Member variables in a class**—these are called fields or instance variables.
- **Variables in a method or block of code**—these are called local variables.
- **Variables in method declarations**—these are called parameters.

Tell them that instance variables are declared in the same way as local variables. The declaration comprises the data type and a variable name. An instance variable also has an access specifier associated with it. Tell them that an access modifier lets you control what other classes have access to a member field. Instance variables are declared inside a class, but outside any method definitions. They can be accessed only through an instance using the dot notation. Instance variables are accessed by objects using the dot operator ( . ).

Explain the syntax for declaring an instance variable within a class.

- `access_modifier` is an optional keyword specifying the access level of an instance variable. It could be private, protected, and public.
- `data_type` specifies the data type of the variable. It can be of primitive types such as int, float, boolean, and so on. Also it can be of reference types, such as strings, arrays, or objects.
- `instanceVariableName` specifies the name of the variable.

**Tips:**

It is common to make fields private. This means that they can only be directly accessed from the class.

Explain the code snippet that demonstrates the declaration of instance variables within a class in the Java program as mentioned in slides 16 and 17.

Finally, explain the pictorial representation of the object instantiated from the class along with the generated output.

**In-Class Question:**

After you finish explaining the Memory Management in Java, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



Where instance variables are declared?

**Answer:**

They are declared inside a class, but outside any method definitions.

After you finish explaining, the concept of instance variables in Java, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



How do you access instance variables?

**Answer:**

Instance variables are accessed using objects.

**Slides 19 to 24**

Let us understand instance methods.

### Instance Methods 1-6

- ◆ They are functions declared in a class.
- ◆ They implement the behavior of an object.
- ◆ They are used to perform operations on the instance variables.
- ◆ They can be accessed by instantiating an object of the class in which it is defined and then, invoking the method.
- ◆ For example: the class **Car** can have a method **Brake ()** that represents the 'Apply Brake' action.
  - + To perform the action, the method **Brake ()** will have to be invoked by an object of class **Car**.

**Conventions to be followed while naming a method are as follows:**

- Cannot be a Java keyword.
- Cannot contain spaces.
- Cannot begin with a digit.
- Can begin with a letter, underscore, or a '\$' symbol.
- Should be a verb in lowercase.
- Should be descriptive and meaningful.
- Should be a multi-word name that begins with a verb in lowercase, followed by adjectives, nouns, and so forth.

© Aptech Ltd.

Classes and Objects/Session 6

19

### Instance Methods 2-6

- ◆ Following figure shows the instance methods declared in the class, **Customer**:

**Customer**

```
int customerID;
String customerName;
String customerAddress;
int customerAge;
```

```
void changeCustomerAddress(String address)
void displayCustomerInformation()
```

→ Instance Methods

© Aptech Ltd.

Classes and Objects/Session 6

20

### Instance Methods 3-6



- The syntax to declare an instance method in a class is as follows:

#### Syntax

```
[access_modifier] <return type> <method_name> ([list
of parameters]) {

 // Body of the method

}
```

where,

**access\_modifier**: Is an optional keyword specifying the access level of an instance method. It could be **private**, **protected**, and **public**.

**returntype**: Specifies the data type of the value that is returned by the method.

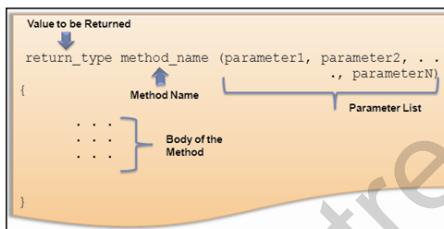
**method\_name**: Is the method name.

**list of parameters**: Are the values passed to the method.

### Instance Methods 4-6



- Following figure shows the declaration of an instance method within the class:



- Each instance of the class has its own instance variables, but the instance methods are shared by all the instances of the class during execution.

### Instance Methods 5-6



- Following code snippet demonstrates the code that declares instance methods within the class, **Customer**:

```
public class Customer {
 // Declare instance variables
 int customerId;
 String customerName;
 String customerAddress;
 int customerAge;

 /**
 * Declares an instance method changeCustomerAddress is created to
 * change the address of the customer object
 */
 void changeCustomerAddress(String address) {
 customerAddress = address;
 }
}
```

- The instance methods **changeCustomerAddress ()** method will accept a string value through parameter address.
- It then assigns the value of address variable to the **customerAddress** field.

### Instance Methods 6-6



```


/*
 * Declares an instance method displayCustomerInformation is created
 * to display the details of the customer object
 */
void displayCustomerInformation() {
 System.out.println("Customer Identification Number: " + customerID);
 System.out.println("Customer Name: " + customerName);
 System.out.println("Customer Address: " + customerAddress);
 System.out.println("Customer Age: " + customerAge);
}


```

- The method **displayCustomerInformation()** displays the details of the customer object.

Using slides 19 to 24, explain Instance Methods.

They are functions declared in a class. They implement the behavior of an object. They are used to perform operations on the instance variables. They can be accessed by instantiating an object of the class in which it is defined and then, invoking the method. Instance method can access instance variables and instance methods directly.

For example, the class **Car** can have a method **a Brake ()** that represents the 'Apply Brake' action. To perform the action, the method **Brake ()** will have to be invoked by an object of class **Car**.

Explain to them that following conventions have to be followed while naming a method:

- Cannot be a Java keyword.
- Cannot contain spaces.
- Cannot begin with a digit.
- Can begin with a letter, underscore, or a '\$' symbol.
- Should be a verb in lowercase.
- Should be descriptive and meaningful.
- Should be a multi-word name that begins with a verb in lowercase, followed by adjectives, nouns, and so forth.

Explain the syntax of defining the instance method in the class. More generally, method declarations have six components, in order:

- Modifiers** — such as **public**, **private**, and others you will learn about later.
- The return type** — the data type of the value returned by the method, or **void**, if the method does not return a value.
- The method name** — the rules for field names apply to method names as well, but the convention is a little different.
- The parameter list** in parenthesis — a comma-delimited list of input parameters, preceded by their data types, enclosed by parentheses **( )**. If there are no parameters, you must use empty parentheses.
- An exception list** — the list of the exception which the method can throw from its body.
- The method body**, enclosed between braces — the method's code, including the declaration of local variables, goes here.

Each instance of the class has its own instance variables, but the instance methods are shared by all the instances of the class during execution. As methods are part of class declaration, they can access the other class member, such as instance variables and method of the class.

Explain the code snippet that demonstrates the code that declares instance methods within the class, Customer as mentioned in slides 23 and 24.

#### Tips:

Instance methods should be placed in the class such that there is no needless duplication of it and it is accessed in all the subclasses of the class in which it is defined.

#### Slides 25 to 28

Let us understand invoking methods.

**Invoking Methods 1-4**

- ◆ To invoke a method, the object name is followed by the dot operator(.) and the method name.
- ◆ A method is always invoked from another method.
- ◆ The method which invokes a method is referred to as the **calling method**.
- ◆ The invoked method is referred to as the **called method**.
- ◆ After execution of all the statements within the code block of the invoked method, the control returns back to the **calling method**.

© Aptech Ltd. Classes and Objects/Session 6 25

**Invoking Methods 2-4**

- ◆ Following code snippet demonstrates a class with **main()** method which creates the instance of the class **Customer** and invokes the methods defined in the class:

```
public class TestCustomer {
 /**
 * @param args the command line arguments
 * The main() method creates the instance of class Customer
 * and invoke its methods
 */
 public static void main(String[] args) {
 // Creates an object of the class
 Customer objCustomer = new Customer();

 // Initialize the object
 objCustomer.customerID = 100;
 objCustomer.customerName = "Jack";
 objCustomer.customerAddress = "123 Street";
 objCustomer.customerAge = 30;
 }
}
```

- ◆ The code instantiates an object **objCustomer** of type **Customer** class and initializes its instance variables.

© Aptech Ltd. Classes and Objects/Session 6 26

**Invoking Methods 3-4**

```

/*
 * Invokes the instance method to display the details
 * of objCustomer object
 */
objCustomer.displayCustomerInformation();

/*
 * Invokes the instance method to
 * change the address of the objCustomer object
 */
objCustomer.changeCustomerAddress("123 Fort, Main Street");

/*
 * Invokes the instance method after changing the address field
 * of objCustomer object
 */
objCustomer.displayCustomerInformation();
}

```

- The method `displayCustomerInformation()` is invoked using the object `objCustomer` and displays the values of the initialized instance variables on the console.
- Then, the method `changeCustomerAddress("123 Fort, Main Street")` is invoked to change the data of the `customerAddress` field.

© Aptech Ltd.

Classes and Objects/Session 6

27

**Invoking Methods 4-4**

- Following figure shows the output of the code:

```

Output - Session 6 (run) ✘
run:
Customer Details
=====
Customer Identification Number: 100
Customer Name: Jack
Customer Address: 123 Street
Customer Age: 30

Modified Customer Details
=====
Customer Identification Number: 100
Customer Name: Jack
Customer Address: 123 Fort, Main Street
Customer Age: 30
BUILD SUCCESSFUL (total time: 0 seconds)

```

© Aptech Ltd.

Classes and Objects/Session 6

28

Using slides 25 to 28, explain invoking methods.

To invoke a method, the object name is followed by the dot operator (.) and the method name. A method is always invoked from another method. The method which invokes a method is referred to as the **calling method**. The invoked method is referred to as the **called method**. After execution of all the statements within the code block of the invoked method, the control returns back to the **calling method**.

Explain the code snippet that demonstrates a class with `main ()` method which creates the instance of the class **Customer** and invokes the methods defined in the class as mentioned in slides 26 to 28.

When a program invokes a method, the program control gets transferred to the called methods. When the method is invoked, all the statements that are part of the method would be executed. When the JVM invokes a class method, it selects the method to invoke based on the type of object reference which is always known as compile time.

**Tips:**

The Javadoc comments that are used for methods in Java are as follows:

- Method description: Javadoc comments on description of the method.
- @param tag: is followed by the name (not data type) and description of the parameter.
- @return tag: is used to specify the return type of methods. It is not used with methods having a void return type.

Statements in the current method after the return statement are skipped, and the control returns to the statement that invoked the method. With methods that are declared void, use the form of return statement that does not return any value.

```
return;
```

Sometimes, a void method has to return explicitly depending on a condition. In this situation, this form of return statement can be used.

**Slides 29 to 31**

Let us understand constructor.

### Constructor 1-3

- ◆ It is a method having the same name as that of the class.
- ◆ It initializes the variables of a class or perform startup operations only once when the object of the class is instantiated.
- ◆ It is automatically executed whenever an instance of a class is created.
- ◆ It can accept parameters and do not have return types.
- ◆ It is of two types:
  - + No-argument constructor
  - + Parameterized constructor
- ◆ Following figure shows the constructor declaration:

© Aptech Ltd. Classes and Objects/Session 6 29

### Constructor 2-3

- ◆ The syntax for declaring constructor in a class is as follows:

#### Syntax

```
<classname>() {
 // Initialization code
}
```

© Aptech Ltd. Classes and Objects/Session 6 30

### Constructor 3-3



◆ **No-argument Constructor:**

- Following code snippet demonstrates a class **Rectangle** with a constructor:

```
public class Rectangle {
 int width;
 int height;

 /**
 * Constructor for Rectangle class
 */
 Rectangle() {
 width = 10;
 height = 10;
 }
}
```

- The code declares a method named **Rectangle ()** which is a constructor.
- This method is invoked by JVM to initialize the two instance variables, **width** and **height**, when the object of type **Rectangle** is constructed.
- The constructor does not have any parameters; hence, it is called as **no-argument constructor**.

© Aptech Ltd.

Classes and Objects / Session 6

31

Using slides 29 to 31, explain constructor methods in a Java class.

Bring to the notice of the students the importance of constructors. Tell them that when discussing about classes, one of the most important sub topic would be constructors. Every class has a constructor. If we do not explicitly write a constructor for a class, the Java compiler builds a default constructor for that class.

Next, provide the definition of constructor. Constructors are methods in a class that create objects or instances of a class. Besides this, the constructors are used for initializing variables and invoking any methods that may be required for initialization.

The constructor is invoked when an object is created. They do not have return types, but do have parameters. A no-argument, default constructor is provided by the compiler for any class that does not have an explicit constructor.

**Tips:**

A class can have more than one constructor.

The main rule for constructors is that they should have the same name as the class.

Explain the syntax for declaring a constructor.

Java provides a default constructor which takes no argument and performs no special actions or initializations. The only action taken by the implicit default constructor is to call the superclass constructor using super () .

Slide 31 shows the code snippet that demonstrates a class **Rectangle** with a constructor.

**Slides 32 to 34**

Let us understand invoking constructor.

### Invoking Constructor 1-3

- The constructor is invoked immediately during the object creation which means:
  - Once the new operator is encountered, memory is allocated for the object.
  - Constructor method is invoked by the JVM to initialize the object.
- Following figure shows the use of new operator to understand the constructor invocation:

```
<class_name> <object_name> = new <class_name>();
```

The parenthesis after the class name indicates the invocation of the constructor.

© Aptech Ltd. Classes and Objects/Session 6 32

### Invoking Constructor 2-3

- Following code snippet demonstrates the code to invoke the constructor for the class **Rectangle**:

```
public class TestConstructor {
 /**
 * @param args the command line arguments
 */
 public static void main(String[] args) {
 // Instantiates an object of the Rectangle class
 Rectangle objRec = new Rectangle();

 // Accessed the instance variables using the object reference
 System.out.println("Width: " + objRec.width);
 System.out.println("Height: " + objRec.height);
 }
}
```

- The codes does the following:
  - Creates an object, **objRec** of type **Rectangle**.
  - Then, the constructor is invoked.
  - The constructor initializes the instance variables of the newly created object, that is, **width** and **height** to 10.

© Aptech Ltd. Classes and Objects/Session 6 33

### Invoking Constructor 3-3

- Following figure shows the output of the code:

© Aptech Ltd. Classes and Objects/Session 6 34

Using slides 32 to 34, explain how to invoke constructor method.

The constructor is invoked immediately during the object creation which means:

- Once the new operator is encountered, memory is allocated for the object.
- Constructor method is invoked by the JVM to initialize the object.

The standard way to create objects in Java using `new` keyword which allocates the memory for the new object, its instance variable is initialized, and constructor body is executed.

Explain the code snippet that demonstrates the code to invoke the constructor for the class `Rectangle` as mentioned in slides 33 and 34.

### Slides 35 to 39

Let us understand default constructor in the class.

**Default Constructor 1-5**



- Consider a situation, where the constructor method is not defined for a class.
- In such a scenario, an implicit constructor is invoked by the JVM for initializing the objects.
- This implicit constructor is also known as default constructor.
- Default Constructor:**
  - Created for the classes where explicit constructors are not defined.
  - Initializes the instance variables of the newly created object to their default values.

© Aptech Ltd. Classes and Objects/Session 6. 35

**Default Constructor 2-5**



- Following:

| Data Type                        | Default Value      |
|----------------------------------|--------------------|
| <code>byte</code>                | 0                  |
| <code>short</code>               | 0                  |
| <code>int</code>                 | 0                  |
| <code>long</code>                | 0L                 |
| <code>float</code>               | 0.0F               |
| <code>double</code>              | 0.0                |
| <code>char</code>                | '\u0000'           |
| <code>boolean</code>             | <code>False</code> |
| <code>String (any object)</code> | <code>Null</code>  |

© Aptech Ltd. Classes and Objects/Session 6. 36

### Default Constructor 3-5



- Following code snippet demonstrates a class **Employee** for which no constructor has been defined:

```
public class Employee {
 // Declares instance variables
 String employeeName;
 int employeeAge;
 double employeeSalary;
 boolean maritalStatus;

 /**
 * Accesses the instance variables and displays
 * their values using the println() method
 */
 void displayEmployeeDetails() {
 System.out.println("Employee Details");
 System.out.println("=====");
 System.out.println("Employee Name: " + employeeName);
 System.out.println("Employee Age: " + employeeAge);
 System.out.println("Employee Salary: " + employeeSalary);
 System.out.println("Employee MaritalStatus: " + maritalStatus);
 }
}
```

- The code declares a class **Employee** with instance variables and an instance method `displayEmployeeDetails()` that prints the value of the instance variables.

### Default Constructor 4-5



- Following code snippet demonstrates a class containing the `main()` method that creates an instance of the class **Employee** and invokes its methods:

```
public class TestEmployee {
 /**
 * @param args the command line arguments
 */
 public static void main(String[] args) {
 // Instantiates an Employee object and initializes it
 Employee objEmp = new Employee();

 // Invokes the displayEmployeeDetails() method
 objEmp.displayEmployeeDetails();
 }
}
```

- As the **Employee** class does not have any constructor defined for itself, a **default constructor** is created for the class at runtime.
- When the statement `new Employee()` is executed, the object is allocated in memory and the instance variables are initialized to their default values by the constructor.
- Then, the method `displayEmployeeDetails()` is executed which displays the values of the instance variables referenced by the object, `objEmp`.

### Default Constructor 5-5



- Following figure shows the output of the code:

```
Output - Session 6 (run) ■
run:
Employee Details
=====
Employee Name: null
Employee Age: 0
Employee Salary: 0.0
Employee MaritalStatus:false
BUILD SUCCESSFUL (total time: 2 seconds)
```

Using slides 35 to 39, explain constructors in the class.

Consider a situation, where the constructor method is not defined for a class. In such a scenario, an implicit constructor is invoked by the JVM for initializing the objects. This implicit constructor is also known as default constructor.

**Default Constructor:**

- Created for the classes where explicit constructors are not defined.
- Initializes the instance variables of the newly created object to their default values.

Explain the table which lists the default values assigned to instance variables of the class depending on their data types mentioned in slide 36.

Explain the code snippet that demonstrates a class containing the `main()` method that creates an instance of the class **Employee** and invokes its methods as mentioned in slides 37 to 39.

In Java, default constructors refer to a null constructor that is automatically generated by the compiler if no constructors have been defined for the class. This means, they are empty and contain nothing.

**In-Class Question:**

After you finish explaining, the concept of constructors in Java, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



Is it a must to declare a default constructor for a class?

**Answer:**

No. If we do not explicitly write a constructor for a class, then the Java compiler builds a default constructor for that class. The default constructor generated by the JVM is also called as vanilla constructor.

**Slides 40 to 44**

Let us understand parameterized constructor.

### Parameterized Constructor 1-5

- ◆ The parameterized constructor contains a list of parameters that initializes instance variables of an object.
- ◆ The value for the parameters is passed during the object creation.
- ◆ This means each object will be initialized with different set of values.
- ◆ Following code snippet demonstrates a code that declares parameterized constructor for the **Rectangle** class:

```

public class Rectangle {
 int width;
 int height;

 /**
 * A default constructor for Rectangle class
 */

 Rectangle() {
 System.out.println("Constructor Invoked...");
 width = 10;
 height = 10;
 }
}
```

### Parameterized Constructor 2-5



```
/*
 * A parameterized constructor with two parameters
 * @param wid will store the width of the rectangle
 * @param heig will store the height of the rectangle
 */
Rectangle (int wid, int heig) {
 System.out.println("Parameterized Constructor");
 width = wid;
 height = heig;
}

/*
 * This method displays the dimensions of the Rectangle object
 */
void displayDimensions() {
 System.out.println("Width: " + width);
 System.out.println("Width: " + height);
}
```

- ◆ The code declares a parameterized constructor, `Rectangle(int wid, int heig)`.
- ◆ During execution, the constructor will accept the values in two parameters and assigns them to `width` and `height` variable respectively.

© Aptech Ltd.

Classes and Objects/Session 6

41

### Parameterized Constructor 3-5



- ◆ Following code snippet demonstrates the code with `main()` method that creates objects of type `Rectangle` and initializes them with parameterized constructor:

```
public class RectangleInstances {

 /**
 * @param args the command line arguments
 */

 public static void main(String[] args) {
 // Declare and initialize two objects for Rectangle class
 Rectangle objRec1 = new Rectangle(10, 20);
 Rectangle objRec2 = new Rectangle(6, 9);

 // Invokes displayDimensions() method to display values
 System.out.println("\nRectangle1 Details");
 System.out.println("=====");
 objRec1.displayDimensions();
 System.out.println("\nRectangle2 Details");
 System.out.println("=====");
 objRec2.displayDimensions();
 }
}
```

© Aptech Ltd.

Classes and Objects/Session 6

42

### Parameterized Constructor 4-5



- ◆ The statement `Rectangle objRec1 = new Rectangle(10, 20);` instantiates an object.
- ◆ During instantiation, the following things happen in a sequence:
  - ◆ Memory allocation is done for the new instance of the class.
  - ◆ Values 10 and 20 are passed to the parameterized constructor, `Rectangle(int wid, int heig)` which initializes the object's instance variables `width` and `height`.
  - ◆ Finally, the reference of the newly created instance is returned and stored in the object, `objRec1`.
- ◆ Following figure shows the output of the code:

```
Output - Session 6 (run) ■
run:
Parameterized Constructor Invoked...
Parameterized Constructor Invoked...
=====
Rectangle1: Details
=====
Width: 10
Width: 20

Rectangle2: Details
=====
Width: 6
Width: 9
BUILD SUCCESSFUL (total time: 1 second)
```

© Aptech Ltd.

Classes and Objects/Session 6

43

**Parameterized Constructor 5-5**

The diagram illustrates the creation of two **Rectangle** objects, **objRec1** and **objRec2**, within a **main()** method. Each object has its own copy of instance variables **width** and **height**.   
**objRec1** is associated with a **Rectangle Object** containing **width = 10** and **height = 20**.   
**objRec2** is associated with a **Rectangle Object** containing **width = 6** and **height = 9**.   
A coffee cup icon is in the top right corner.

Using slides 40 to 44, explain parameterized constructor.

The parameterized constructor contains a list of parameters that initializes instance variables of an object. The value for the parameters is passed during the object creation. This means each object will be initialized with different set of values.

Explain the code snippet that demonstrates the code that declares parameterized constructor for the Rectangle class mentioned in slides 40 to 41.

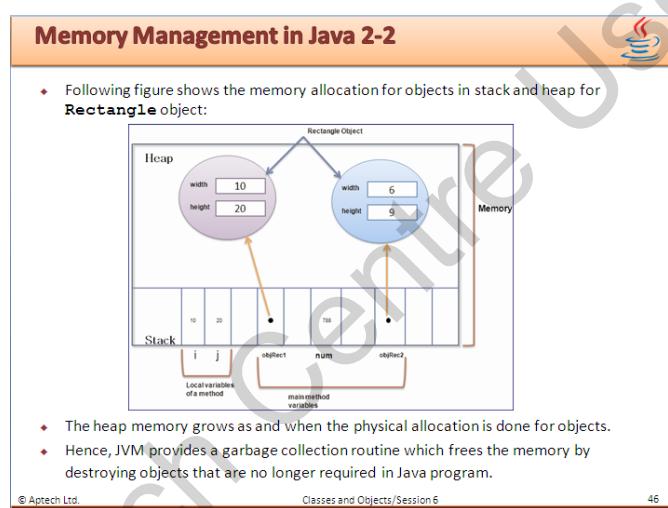
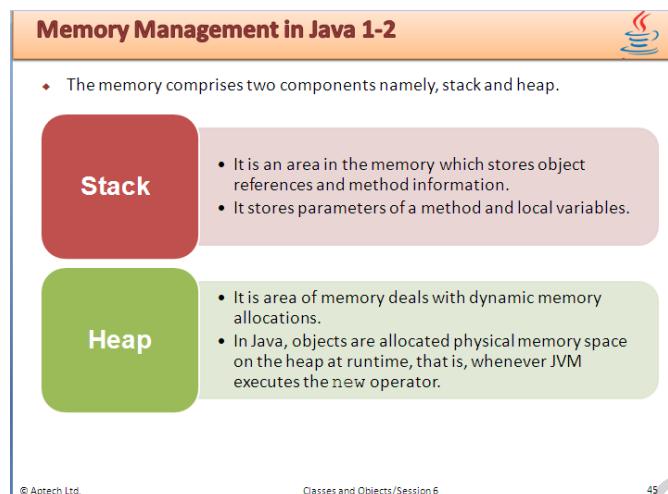
Explain the code snippet that demonstrates the code with **main()** method that creates objects of type **Rectangle** and initializes them with parameterized constructor as mentioned in slides 42 and 43.

Each object contains its own copy of instance variables that are initialized through constructor which is displayed in slide 44.

Tell them that in parameterized constructor, the number of parameters can be greater or equal to one. When an object is declared in a parameterized constructor, the initial values have to be passed as argument to constructor function.

## Slides 45 and 46

Let us understand memory management in Java.



Using slides 45 and 46, explain memory management in Java.

Memory management is one of the prominent features of Java platform. Memory management is the process of allocating new objects and removing unused objects to make space for those new object allocations.

Tell them that the memory comprises two areas namely, stack and heap.

### Stack

- It is an area in the memory which stores object references and method information.
- It stores parameters of a method and local variables.

### Heap

- It is area of memory deals with dynamic memory allocations.
- In Java, objects are allocated physical memory space on the heap at runtime, that is, whenever JVM executes the new operator.

The heap memory grows as and when the physical allocation is done for objects. Hence, JVM provides a garbage collection routine which frees the memory by destroying objects that are no longer required in Java program. This way the memory in the heap is re-used for objects allocation.

The figure shows the memory allocation for objects in stack and heap for **Rectangle** object in slide 46.

### In-Class Question:

After you finish explaining the memory management in Java, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



What is the use of garbage collection routine?

### Answer:

It frees the memory by destroying objects that are no longer required in Java program. Hence, the memory in the heap is re-used for objects allocation.

### Slides 47 to 50

Let us understand assigning object references.

### Assigning Object References 1-4

- ◆ Working with primitive data types:
  - ❖ The value of one variable can be assigned to another variable using the assignment operator.
  - ❖ For example, `int a = 10; int b = a;` copies the value from variable **a** and stores it in the variable **b**.
  - ❖ Following figure shows assigning of a value from one variable to another:

```

 graph TD
 a[("a
10
int a = 10;")]
 b[("b
int b;")]
 a -- "Copies the value" --> b

```

- ◆ Working with object references:
  - ❖ Similar to primitive data types, the value stored in an object reference variable can be copied into another reference variable.
  - ❖ Both the reference variables must be of same type, that is, both the references must belong to the same class.

© Aptech Ltd. Classes and Objects/Session 6 47

### Assigning Object References 2-4

- ◆ Following code snippet demonstrates assigning the reference of one object into another object reference variable:

```

public class TestObjectReferences {
 /**
 * @param args the command line arguments
 */
 public static void main(String[] args) {
 /* Instantiates an object of type Rectangle and stores
 * its reference in the object reference variable, objRec1
 */
 Rectangle objRec1 = new Rectangle(10, 20);
 // Declares a reference variable of type Rectangle
 Rectangle objRec2;
 }
}

```

- ❖ The **objRec1** points to the object that has been allocated memory and initialized to 10 and 20.
- ❖ The **objRec2** is an object reference variable that does not point to any object.

© Aptech Ltd. Classes and Objects/Session 6 48

### Assigning Object References 3-4



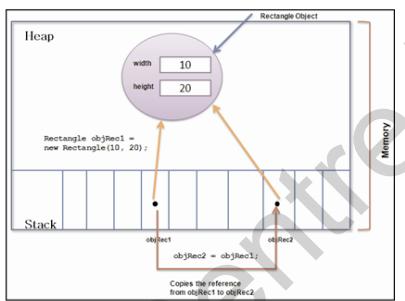
```
// Assigns the value of objRec1 to objRec2
objRec2 = objRec1;
System.out.println("\nRectangle1 Details");
System.out.println("=====");
/* Invokes the method that displays values of the
 * instance variables for object, objRec1
 */
objRec1.displayDimensions();
System.out.println("\nRectangle2 Details");
System.out.println("=====");
objRec2.displayDimensions();
}
```

- The statement, `objRec2 = objRec1;` copies the address in `objRec1` into `objRec2`.
- Thus, the references are copied between the variables created on the stack without affecting the actual objects created on the heap.

### Assigning Object References 4-4



- Following figure shows the assigning of reference for the statement, `objRec2 = objRec1;`



Using slides 47 to 50, explain assigning object references.

Explain them how the value of two variables in copied using assignment operators for primitive data types. Explain the figure presented on slide 47 as follows:

- `int a = 10; int b = a;` copies the value from variable **a** and stores it in the variable **b**.

Similarly, the values between the two objects can be copied. However, the object variable also known as reference variable contains the address of the object created in the heap. Thus, copying the values in the objects, copies the reference or address.

Point on copying reference in objects is as follows:

- Both the reference variables must be of same type, that is, both the references must belong to the same class.
- Thus, the references are copied between the variables created on the stack without affecting the actual objects created on the heap.

Explain the code snippet that demonstrates assigning the reference of one object into another object reference variable mentioned in slides 48 and 49.

The figure shows assigning of reference for the statement, `objRec2 = objRec1;` in slide 50.

As values between primitive data types can be copied, similarly the value stored in object reference type can be copied into another reference variable. As Java is strongly-typed language, both reference variables can be of same type. In other words, both the references must belong to the same class.

Reference variable can be used to store the address of the variable. Assigning reference will not create distinct copies of objects.

### Slide 51

Let us understand encapsulation in Java.

## Encapsulation



- ◆ In OOP languages, the concept of hiding implementation details of an object is achieved by applying the concept of encapsulation.

### Encapsulation

- It is a mechanism which binds code and data together in a class.
- Its main purpose is to achieve data hiding within a class which means:
  - Implementation details of what a class contains need not be visible to other classes and objects that use it.
  - Instead, only specific information can be made visible to the other components of the application and the rest can be hidden.
  - By hiding the implementation details about what is required to implement the specific operation in the class, the usage of operation becomes simple.

© Aptech Ltd. Classes and Objects/Session 6 51

Using slide 51, explain encapsulation in Java.

In OOP languages, the concept of hiding implementation details of an object is achieved by applying the concept of encapsulation. It is a mechanism which binds code and data together in a class.

The main purpose of encapsulation is to achieve data hiding within a class which means:

- Implementation details of what a class contains need not be visible to other classes and objects that use it.
- Instead, only specific information can be made visible to the other components of the application and the rest can be hidden.

By hiding the implementation details about what is required to implement the specific operation in the class, the usage of operation becomes simple.

Encapsulation is also called data hiding. It is said that in OOP languages, encapsulation covers the internal workings of Java object. The main purpose of data hiding within a class is to reduce the complexity in the software development.

Data encapsulation hides the instance variables that represent the state of an object. Thus, the only interaction and modification is performed through methods.

**Slides 52 to 54**

Let us understand access modifiers.

### Access Modifiers 1-3

- In Java, the data hiding is achieved by using **access modifiers**.
- Access Modifiers:**
  - Determine how members of a class, such as instance variable and methods are accessible from outside the class.
  - Decide the scope or visibility of the members.
  - Are of four types:

|                          |                                                                                                                                                                                                                                              |
|--------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>public</b>            | • Members declared as public can be accessed from anywhere in the class as well as from other classes.                                                                                                                                       |
| <b>private</b>           | • Members are accessible only from within the class in which they are declared.                                                                                                                                                              |
| <b>protected</b>         | • Members to be accessible from within the class as well as from within the derived classes.                                                                                                                                                 |
| <b>package (default)</b> | <ul style="list-style-type: none"> <li>Allows only the public members of a class to be accessible to all the classes present within the same package.</li> <li>This is the default access level for all the members of the class.</li> </ul> |

© Aptech Ltd. Classes and Objects/Session 6 52

### Access Modifiers 2-3

- As a general rule in Java, the details and implementation of a class is hidden from the other classes or external objects in the application.
- This is done by making instance variables as **private** and instance methods as **public**.
- Following code snippet demonstrates the use of the concept of encapsulation in the class **Rectangle**:

```

public class Rectangle {
 // Declares instance variables
 private int width;
 private int height;
 /* Declares a no-argument constructor */
 public Rectangle() {
 System.out.println("Constructor Invoked...");
 width = 10;
 height = 10;
 }
}

```

- The **access specifiers** of the instance variables, **width** and **height** are changed from **default** to **private** which means that the class fields are not directly accessible from outside the class.

© Aptech Ltd. Classes and Objects/Session 6 53

### Access Modifiers 3-3

```

/**
 * Declares a parameterized constructor with two parameters
 * @param wid
 * @param heig
 */
public Rectangle (int wid, int heig) {
 System.out.println("Parameterized Constructor Invoked...");
 width = wid;
 height = heig;
}

/**
 * Displays the dimensions of the Rectangle object
 */
public void displayDimensions(){
 System.out.println("Width: " + width);
 System.out.println("Height: " + height);
}

```

- The access modifiers for the methods are changed to **public**.
- Thus, the users can access the class members through its methods without impacting the internal implementation of the class.

© Aptech Ltd. Classes and Objects/Session 6 54

Using slides 52 to 54, explain access modifiers in Java.

In Java, the data hiding is achieved by using access modifiers. Access modifiers are the keywords in OOP languages that set the accessibility of classes, methods, and other members. They are specific part of programming language that facilitates the encapsulation of components.

Features of access modifiers are:

- Determine how members of a class, such as instance variable and methods are accessible from outside the class.
- Decide the scope or visibility of the members.
- Are of four types:
  - **public**
    - Members declared as public can be accessed from anywhere in the class as well as from other classes.
  - **private**
    - Members are accessible only from within the class in which they are declared.
  - **protected**
    - Members to be accessible from within the class as well as from within the derived classes.
  - **package (default)**
    - It allows only the public members of a class to be accessible to all the classes present within the same package. This is the default access level for all the members of the class.

As a general rule in Java, the details and implementation of a class is hidden from the other classes or external objects in the application. This is done by making instance variables as `private` and instance methods as `public`.

Explain the code snippet that demonstrates the use of the concept of encapsulation in the class `Rectangle` as mentioned in slides 53 and 54.

#### In-Class Question:

After you finish explaining the access modifiers, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



What will happen by making instance variables as `private` and instance methods as `public`?

#### Answer:

The details and implementation of a class is hidden from the other classes or external objects in the application.

#### Tips:

Ensure that errors cannot happen if other programmers use your class. This can be helped by access levels. Like avoid public fields except for constants as public fields tend to link you to a particular implementation and limit your flexibility in changing the code.

**Slides 55 to 60**

Let us understand object initializers.

**Object Initializers 1-6**

- ◆ They provide a way to create an object and initialize its fields.
- ◆ They complement the use of constructors to initialize objects.
- ◆ There are two approaches to initialize the fields or instance variables of the newly created objects:
  - + Using instance variable **initializers**.
  - + Using initialization block
- ◆ **Instance Variable Initializers:**
  - + In this approach, you specify the names of the fields and/or properties to be initialized, and give an initial value to each of them.
  - + Following code snippet demonstrates a Java program that declares a class, **Person** and initializes its fields:

```
public class Person {
 private String name = "John";
 private int age = 12;

 /**
 * Displays the details of Person object
 */
}
```

© Aptech Ltd.

Classes and Objects/Session 6

55

**Object Initializers 2-6**

```
void displayDetails() {
 System.out.println("Person Details");
 System.out.println("=====");
 System.out.println("Person Name: " + name);
}
```

- + The instance variables **name** and **age** are initialized to values '**John**' and **12** respectively.
- + Following code snippet shows the class with **main()** method that creates objects of type **Person**:

```
public class TestPerson {
 /**
 * @param args the command line arguments
 */
 public static void main(String[] args) {
 Person objPerson1 = new Person();
 objPerson1.displayDetails();
 }
}
```

© Aptech Ltd.

Classes and Objects/Session 6

56

**Object Initializers 3-6**

- ◆ The code creates an object of type **Person** and invokes the method to display the details.
- ◆ Following figure shows the output of the code:

- ◆ **Initialization Block:**
  - + In this approach, an initialization block is specified within the class.
  - + The initialization block is executed before the execution of constructors during an object initialization.

© Aptech Ltd.

Classes and Objects/Session 6

57

### Object Initializers 4-6



- Following code snippet demonstrates the class **Account** with an initialization block:

```
public class Account {
 private int accountID;
 private String holderName;
 private String accountType;

 /**
 * Initialization block
 */
 {
 accountID = 100;
 holderName = "John Anderson";
 accountType = "Savings";
 }
}
```

- In the code, the initialization blocks initializes the instance variables or fields of the class.

### Object Initializers 5-6



```
/*
 * Displays the details of Account object
 */
public void displayAccountDetails() {
 System.out.println("Account Details");
 System.out.println("=====");
 System.out.println("Account ID: " + accountID + "\nAccount
Type: " + accountType);
}
```

- Following code snippet shows the code with main () method to initialize the **Account** object through initialization block:

```
public class TestInitializationBlock {
 /**
 * @param args the command line arguments
 */
 public static void main(String[] args) {
 Account objAccount = new Account();
 objAccount.displayAccountDetails();
 }
}
```

### Object Initializers 6-6



- Following figure shows the output of the code:

```
Output - Session 6 (run) ✘
run:
Account Details
=====
Account ID: 100
Account Type: Savings
BUILD SUCCESSFUL (total time: 1 second)
```

Using slides 55 to 60, explain object initializers.

They provide a way to create an object and initialize its fields. They complement the use of constructors to initialize objects.

Tell them that object field initializers also known as **instance variable initializers** are declared inside a class. An instance variable initializer contains an equal sign and an expression.

Explain to them the syntax of object field initializer as shown:

```
class <classname>{
// Object field initialization with value/expression
<data_type> field1=<value>/<expression>
}
```

Explain the code snippet that demonstrates a Java program that declares a class, **Person** and initializes its fields as mentioned in slides 55 and 56.

Next, you explain to them object block initializer. Object block initializers also known as **instance initialization block** are declared within the class, but outside the constructor and method definitions. An instance initializer begins with a brace bracket, followed by one or more statements and ends with a brace bracket.

Explain the code snippet which shows the code with `main()` method to initialize the **Account** object through initialization block as mentioned in slides 59 and 60.

**In-Class Question:**

After you finish explaining the object initializers, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



When the initialization block is executed?

**Answer:**

It is executed before the execution of constructors, during an object initialization.

**Tips:**

It is not necessary to declare fields at the beginning of class declaration. But it is necessary that they be declared and initialized before they are used.

**Slide 61**

Let us summarize the session.

**Summary**



- ◆ The class is a logical construct that defines the shape and nature of an object.
- ◆ Objects are the actual instances of the class and are created using the new operator. The new operator instructs JVM to allocate the memory for the object.
- ◆ The members of a class are fields and methods. Fields define the state and are referred to as instance variables, whereas methods are used to implement the behavior of the objects and are referred to as instance methods.
- ◆ Each instance created from the class will have its own copy of the instance variables, whereas methods are common for all instances of the class.
- ◆ Constructors are methods that are used to initialize the fields or perform startup operations only once when the object of the class is instantiated.
- ◆ The heap area of memory deals with the dynamic memory allocations for objects, whereas the stack area holds the object references.
- ◆ Data encapsulation hides the instance variables that represents the state of an object through access modifiers. The only interaction or modification on objects is performed through the methods.

© Aptech Ltd. Classes and Objects/Session 6 61

In slide 61, you will summarize the session. End the session with a brief summary of what has been taught in the session.

### 6.3 Post Class Activities for Faculty

You should familiarize yourself with the topics of the next session which is based on methods and access specifiers.

#### Tips:

You can also check the Articles/Blogs/Expert Videos uploaded on the Online Varsity site to gain additional information related to the topics covered in the next session. You can also connect to online tutors on the Online Varsity site to ask queries related to the sessions.

# Session 7 – Methods and Access Specifiers

## 7.1 Pre-Class Activities

Before you commence the session, you should familiarize yourself with the topics of this session in-depth.

Here, you can discuss the key points with the students that were covered in the previous session. Prepare a question or two which will help you to relate the current session objectives.

### 7.1.1 Objectives

By the end of this session, the learners will be able to:

- Describe methods
- Explain the process of creation and invocation of methods
- Explain passing and returning values from methods
- Explain variable argument methods
- Describe the use of Javadoc to lookup methods
- Describe access specifiers and the types of access specifiers
- Explain the use of access specifiers with methods
- Explain the concept of method overloading
- Explain the use of this keyword

### 7.1.2 Teaching Skills

To teach this session, you should be well-versed with creation and invocation of methods, passing and returning values to and from methods and also the use of Javadoc to lookup methods. Also, familiarize yourself with the concept of access specifiers for restricting access to the class members.

You should teach the concepts in the theory class using the images provided. For teaching in the class, you are expected to use slides and LCD projectors.

#### Tips:

It is recommended that you test the understanding of the students by asking questions in between the class.

#### In-Class Activities:

Follow the order given here during In-Class activities.

**Overview of the Session:**

Give the students the overview of the current session in the form of session objectives. Show the students slide 2 of the presentation.

| Objectives                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"> <li>◆ Describe methods</li> <li>◆ Explain the process of creation and invocation of methods</li> <li>◆ Explain passing and returning values from methods</li> <li>◆ Explain variable argument methods</li> <li>◆ Describe the use of <u>Javadoc</u> to lookup methods</li> <li>◆ Describe access <u>specifiers</u> and the types of access <u>specifiers</u></li> <li>◆ Explain the use of access <u>specifiers</u> with methods</li> <li>◆ Explain the concept of method overloading</li> <li>◆ Explain the use of this keyword</li> </ul> |

© Aptech Ltd. Methods and Access Specifiers/Session 7 2

Tell the students that this session introduces the creation and invocation of methods, passing, and returning values to and from methods and also the use of Javadoc to lookup methods.

The session will explain the concept of access specifiers for restricting access to the class members and the concept of method and constructor overloading.

**7.2 In-Class Explanation****Slide 3**

Let us understand the role of access specifiers in accessing methods.

| Introduction                                                                                                                                                                                                                                                                                                                              |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"> <li>◆ Methods in Java are such a feature that allows grouping of statements and execution of a specific set of statements instead of executing the entire program.</li> <li>◆ Java provides a set of access <u>specifiers</u> that can help the user to restrict access to certain methods.</li> </ul> |

© Aptech Ltd. Methods and Access Specifiers/Session 7 3

Using slide 3, explain the role of access specifiers in accessing methods.

Methods in Java allows grouping of statements and execution of a specific set of statements, instead of executing the entire program.

Tell them what if the user wants to restrict access to certain methods in the class?

Java provides a set of access specifiers that can help the user to accomplish this task. As already learnt in the previous session, Java provides a set of access specifiers that can help the user to restrict access to certain methods.

### Slides 4 to 6

Let us understand methods in Java class.

### Methods 1-3

A Java method can be defined as a set of statements grouped together for performing a specific task.

For example, a call to the `main()` method which is the point of entry of any Java program, will execute all the statements written within the scope of the `main()` method.

- The syntax for declaring a method is as follows:

**Syntax**

```
modifier return_type method_name([list_of_parameters]) {
 // Body of the method
}
```

where,

- `modifier`: Specifies the visibility of the method. Visibility indicates which object can access the method. The values can be `public`, `private`, or `protected`.
- `return_type`: Specifies the data type of the value returned by the method.
- `method_name`: Specifies the name of the method.
- `list_of_parameters`: Specifies the comma-delimited list of values passed to the method.

© Aptech Ltd. Methods and Access Specifiers/Session 7 4

### Methods 2-3

- Generally, a method declaration has the following six components, in order:

|   |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|---|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1 | • Modifiers such as <code>public</code> , <code>private</code> , and <code>protected</code> .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| 2 | • A return type that indicates the data type of the value returned by the method.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| 3 | • The return type is set to <code>void</code> if the method does not return a value.<br><br>• The method name that is specified based on certain rules. A method name: <ul style="list-style-type: none"> <li>cannot be a Java keyword</li> <li>cannot have spaces</li> <li>cannot begin with a digit</li> <li>cannot begin with any symbol other than a \$ or _</li> <li>can be a verb in lowercase</li> <li>can be a multi-word name that begins with a verb in lowercase, followed by adjectives or nouns</li> <li>can be a multi-word name with the first letter of the second word and each of the following words capitalized</li> <li>should be descriptive and meaningful</li> </ul> |

© Aptech Ltd. Methods and Access Specifiers/Session 7 5

**Methods 3-3**

Some valid method names are `add`, `_view`, `$calc`, `add_num`, `setFirstName`, `compareTo`, `isValid`, and so on.

- 4 • Parameter list in parenthesis is separated with a comma delimiter.  
• Each parameter is preceded by its data type.  
• If there are no parameters, an empty parenthesis is used.
- 5 • An exception list that specifies the names of exceptions that can be thrown by the method.  
• An exception is an event encountered during the execution of the program, disrupting the flow of program execution.
- 6 • Method body consists of a set of statements enclosed between curly braces '{}'.  
• Method body can have variables, method calls, and even classes.

The two components of a method declaration namely, the method name and the parameter types comprise the method signature.

© Aptech Ltd. Methods and Access Specifiers/Session 7 5

Using slides 4 to 6, explain methods in Java class.

Tell them about the `main()` method which is the point of entry of any Java program will execute all the statements written within the scope of the `main()` method.

Then discuss the syntax of writing methods in Java. Any method such as `main()` method or instance method will have three parts,

- **modifier:** Specifies the visibility of the method. Visibility indicates which object can access the method. The values can be `public`, `private`, or `protected`.
- **return\_type:** Specifies the data type of the value returned by the method.
- **method\_name:** Specifies the name of the method.
- **list\_of\_parameters:** Specifies the comma-delimited list of values passed to the method.

Tell them that the two components of a method declaration namely, the method name and the parameter types comprise the method signature.

Then, discuss with them the rules for writing methods in Java and discuss some examples on valid method names allowed in Java such as `add`, `_view`, `$calc`, `add_num`, `setFirstName`, `compareTo`, `isValid`, and so on.

Then, discuss about the declaration of parameters and exceptions in the method signature.

#### **Parameters in method signature**

The parameter list in parenthesis is separated with a comma delimiter. Each parameter is preceded by its data type. If there are no parameters, an empty parenthesis is used.

Example: `void addContactDetails(int cust_id, String cust_name, int cust_age) { }`

#### **Exceptions in method signature**

An exception list specifies the names of exceptions that can be thrown by the method. An exception is an event encountered during the execution of the program, disrupting the flow of program execution.

Example: `void addContactDetails(int cust_id, String cust_name, int cust_age) throws CustomerException { }`

The method declaration shows the writing of exceptions with the method signature, which indicates that the method can throw an error while adding customer details.

### Slides 7 to 13

Let us understand creating and invoking methods.

**Creating and Invoking Methods 1-7**

- Methods help to segregate tasks to provide modularity to the program.
- A program is modular when different tasks in a program are grouped together into modules or sections.
- For example, to perform different types of mathematical operations such as addition, subtraction, multiplication, and so on, a user can create individual methods as shown in the following figure:

The diagram illustrates a central brown circle labeled 'obj' at the bottom, representing an object. Four arrows point from this object to four rectangular boxes above it, each containing a method name: 'add(a,b)', 'sub(a,b)', 'mul(a,b)', and 'div(a,b)'. These boxes are grouped under a larger box labeled 'Methods' at the top.

- The figure shows an object named `obj` accessing four different methods namely, `add (a,b)`, `sub (a,b)`, `mul (a,b)`, and `div (a,b)` for performing the respective operations on two numbers.

© Aptech Ltd. Methods and Access Specifiers/Session 7 7

**Creating and Invoking Methods 2-7**

- To create a method that adds two numbers, the user can write a method as depicted in the following code snippet:

```
public void add(int num1, int num2){
 int num3; // Declare a variable
 num3 = num1 + num2; // Perform the addition of numbers
 System.out.println("Addition is " + num3); // Print the result
}
```

- Defines a method named `add()` that accepts two integer parameters `num1` and `num2`.
- Has declared the method with the `public` access specifier which means that it can be accessed by all objects.
- Has set the return type to `void`, indicating that the method does not return anything.
- Statement '`int num3;`' is a declaration of an integer variable named `num3`.
- Statement '`num3 = num1 + num2;`' is an addition operation performed on parameters `num1` and `num2` using the arithmetic operator '+'.
- Result is stored in a third variable `num3` by using the assignment operator '='.
- Finally, '`System.out.println("Addition is "+ num3);`' is used to print the value of variable `num3`.
- Method signature is '`add(int, int)`'.

© Aptech Ltd. Methods and Access Specifiers/Session 7 8

**Creating and Invoking Methods 3-7**

- To use a method, it must be called or invoked. When a program calls a method, the control is transferred to the called method.
- The called method executes and returns control to the caller.
- The call is returned back after the return statement of a method is executed or when the closing brace is reached.
- A method can be invoked in one of the following ways:

If the method returns a value, then, a call to the method results in return of some value from the method to the caller. For example,

```
int result = obj.add(20, 30);
```

If the method's return type is set to `void`, then, a call to the method results in execution of the statements within the method without returning any value to the caller.

For example, a call to the method would be `obj.add(23, 30)` without anything returned to the caller.

© Aptech Ltd. Methods and Access Specifiers/Session 7 9

### Creating and Invoking Methods 4-7

Consider the project **Session7** created in the **NetBeans IDE** as shown in the following figure:

```

Session7 - NetBeans IDE 7.1.2
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
Projects Session7
Source Packages session7
Calculator.java
Source History ...
Calculator.java
1 /* To change this template, choose Tools | Templates
2 * and open the template in the editor.
3 */
4 package session7;
5
6 /**
7 * @author vincent
8 */
9
10 public class Calculator {
11
12 /**
13 * @param args the command line arguments
14 */
15 public static void main(String[] args) {
16 // TODO code application logic here
17 }
18 }

```

The project consists of a package named **session7** with the **Calculator** class that has the **main()** method.  
Several methods for mathematical operations can be added to the class.

© Aptech Ltd. Methods and Access Specifiers/Session 7 10

### Creating and Invoking Methods 5-7

Following code snippet demonstrates an example of creation and invocation of methods:

```

package session7;
public class Calculator {
 // Method to add two integers
 public void add(int num1, int num2) {
 int num3;
 num3 = num1 + num2;
 System.out.println("Result after addition is " + num3);
 }
 // Method to subtract two integers
 public void sub(int num1, int num2) {
 int num3;
 num3 = num1 - num2;
 System.out.println("Result after subtraction is " + num3);
 }
 // Method to multiply two integers
 public void mul(int num1, int num2) {
 int num3;
 num3 = num1 * num2;
 System.out.println("Result after multiplication is " + num3);
 }
}

```

© Aptech Ltd. Methods and Access Specifiers/Session 7 11

### Creating and Invoking Methods 6-7

```

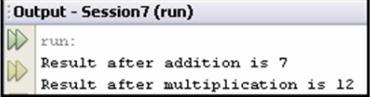
// Method to divide two integers
public void div(int num1, int num2) {
 int num3;
 num3 = num1 / num2;
 System.out.println("Result after division is " + num3);
}

/**
 * @param args the command line arguments
 */
public static void main(String[] args) {
 // Instantiate the Calculator class
 Calculator objCalc = new Calculator();
 // Invoke the methods with appropriate arguments
 objCalc.add(3, 4);
 objCalc.mul(3, 4);
}

```

© Aptech Ltd. Methods and Access Specifiers/Session 7 12

**Creating and Invoking Methods 7-7**



- Class **Calculator** consists of methods such as **add()**, **sub()**, **mul()**, and **div()** that are used to perform the respective operations.
- Each method accepts two integers as parameters.
- The **main()** method creates an object, **objCalc** of class **Calculator**.
- The object **objCalc** uses the dot '.' operator to invoke the **add()** and **mul()** methods.
- Following figure shows the output of the program:

© Aptech Ltd. Methods and Access Specifiers/Session 7 13

Using slides 7 to 13, explain creating and invoking methods.

Tell them that to perform different types of mathematical operations such as addition, subtraction, multiplication, and so on in a calculator class, a user can create individual methods as shown in the figure mentioned in slide 7.

Then explain how to create a method that adds two numbers, the user can write a method as depicted in the code snippet mentioned in slide 8. Tell them that a method named **add()** that accepts two parameters **num1** and **num2**, each of type **integer** is defined. Also, the method has been defined with the **public** access specifier which means that it can be accessed by all objects. The return type set to **void** indicates that the method does not return anything.

To use a method, it must be called or invoked. A method can be invoked in one of the following ways:

- If the method returns a value, then, a call to the method results in return of some value from the method to the caller. For example, `int result = obj.add(20, 30);`
- If the method's return type is set to **void**, then, a call to the method results in execution of the statements within the method without returning any value to the caller.
- For example, a call to the method would be `obj.add(23, 30)` without anything returned to the caller.

Explain the project **Session7** with the class **Calculator** created in the NetBeans IDE as shown in the figure on slide 10. Then, explain the code snippet demonstrates an example of creation and invocation of methods within the created class in the NetBeans IDE as mentioned in slides 11 to 13.

Explain the creation of the methods within the class. The class **Calculator** consists of methods such as **add()**, **sub()**, **mul()**, and **div()** that are used to perform the respective operations. Each method accepts two integers as parameters. The **main()** method creates an object, **objCalc** of class **Calculator**. The object **objCalc** uses the dot '.' operator to call or invoke the **add()** and **mul()** methods. When the program is executed in NetBeans IDE, first the **main()** method is executed by the JVM. The runtime creates the object **objCalc** of class **Calculator**. Next, the **add()** method is invoked with two values **3** and **4**. The **add()** method is processed and the print statement displays the result. Lastly, the **mul()** method is invoked with two parameters and prints the result.

**Slide 14**

Let us understand passing and returning values from methods.

### Passing and Returning Values from Methods



| Parameters                                                                                                                                                                                          | Arguments                                                                                                                                                                                                                                                     |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Parameters are the list of variables specified in a method declaration.                                                                                                                             | Arguments are the actual values that are passed to the method when it is invoked.                                                                                                                                                                             |
| When a method is invoked, the type and order of arguments that are passed must match the type and order of parameters declared in the method.                                                       |                                                                                                                                                                                                                                                               |
| A method can accept primitive data types such as <code>int</code> , <code>float</code> , <code>double</code> , and so on as well as reference data types such as arrays and objects as a parameter. |                                                                                                                                                                                                                                                               |
| Arguments can be passed by                                                                                                                                                                          |  <div style="display: flex; justify-content: space-around; align-items: center;"> <span style="color: green;">value</span> <span style="color: red;">reference</span> </div> |

© Aptech Ltd.      Methods and Access Specifiers/Session 7      14

Using slide 14, explain passing and returning values from methods.

Passing of parameters to a method can be through passing a value or passing a reference.

Passing of parameters to a method involves **Parameters list** that is specified with the method declaration and the **actual argument** values that are passed to the method when it is invoked.

When a method is invoked, the type and order of arguments that are passed must match the type and order of parameters declared in the method. A method can accept primitive data types such as `int`, `float`, `double`, and so on as well as reference data types such as arrays and objects as a parameter.

Arguments can be passed as value or reference to the invoked method.

**Slides 15 and 16**

Let us understand passing arguments by value.

### Passing Arguments by Value 1-2



- ◆ When arguments are passed by value it is known as call-by-value and it means that:
- A copy of the argument is passed from the calling method to the called method.
- Changes made to the argument passed in the called method will not modify the value in the calling method.
- Variables of primitive data types such as `int` and `float` are passed by value.
- ◆ Following code snippet demonstrates an example of passing arguments by value:

```

 package session7;
 public class PassByValue {
 // method accepting the argument by value
 public void setVal(int num1) {
 num1 = num1 + 10;
 }
 }

```

© Aptech Ltd.      Methods and Access Specifiers/Session 7      15

### Passing Arguments by Value 2-2

```
public static void main(String[] args) {
 // Declare and initialize a local variable
 int num1 = 10;
 // Instantiate the PassByValue class
 PassByValue obj = new PassByValue();
 // Invoke the setVal() method with num1 as parameter
 obj.setVal(num1);
 // Print num1 to check its value
 System.out.println("Value of num1 after invoking setVal is " + num1);
}
```

- Following figure shows the output of the code:

**Output - Session7 (run)**  
 run:  
 Value of num1 after invoking setVal is 10  
 BUILD SUCCESSFUL (total time: 0 seconds)

- Output shows that the value of `num1` is still **10** even after invoking `setVal()` method when the value had been incremented.
- This is because, `num1` was passed by value.

© Aptech Ltd.

Methods and Access Specifiers/Session 7

16

Using slides 15 and 16, explain passing arguments by value.

When arguments are passed by value it is known as call-by-value and it means that:

- A copy of the argument is passed from the calling method to the called method.
- Changes made to the argument passed in the called method will not modify the value in the calling method.
- Variables of primitive data types such as int and float are passed by value.

Explain the code snippet that demonstrates an example of passing arguments by value mentioned in slides 15 and 16.

#### Tips:

The Java programming language does not allow to pass methods into methods, but you can pass objects into methods and then invoke the object's method.

#### Slides 17 to 19

Let us understand passing arguments by reference.

### Passing Arguments by Reference 1-3

- When arguments are passed by reference it means that:

The actual memory location of the argument is passed to the called method and the object or a copy of the object is not passed.

The called method can change the value of the argument passed to it.

Variables of reference types such as objects are passed to the methods by reference.

There are two references of the same object namely, argument reference variable and parameter reference variable.

- Following code snippet demonstrates an example of passing arguments by reference:

```
package session7;
class Circle{
 // Method to retrieve value of PI
 public double getPI(){
 return 3.14;
 }
}
```

© Aptech Ltd.

Methods and Access Specifiers/Session 7

17

### Passing Arguments by Reference 2-3

```
// Define another class PassByRef
public class PassByRef{
 // Method to calculate area of a circle that
 // takes the object of class Circle as a parameter
 public void calcArea(Circle objPI, double rad){
 // Use getPI() method to retrieve the value of PI
 double area= objPI.getPI() * rad * rad;
 // Print the value of area of circle
 System.out.println("Area of the circle is "+ area);
 }
 public static void main(String[] args){
 // Instantiate the PassByRef class
 PassByRef p1 = new PassByRef();
 // Invoke the calcArea() method with object of class Circle as
 // a parameter
 p1.calcArea(new Circle(), 2);
 }
}
```

© Aptech Ltd.

Methods and Access Specifiers/Session 7

18

### Passing Arguments by Reference 3-3



- Following figure shows the output of the code:

```
Output - Session7 (run)
run:
Area of the circle is 12.56
BUILD SUCCESSFUL (total time: 0 seconds)
```

- Note that the value of PI is passed by reference and not by value.

© Aptech Ltd.

Methods and Access Specifiers/Session 7

19

Using slides 17 to 19, explain passing arguments by reference.

When arguments are passed by reference it means that:

- The actual memory location of the argument is passed to the called method and the object or a copy of the object is not passed.
- The called method can change the value of the argument passed to it.
- Variables of reference types such as objects are passed to the methods by reference.
- There are two references of the same object namely, argument reference variable and parameter reference variable.

Explain the code snippet that demonstrates an example of passing arguments by reference mentioned in slides 17 to 19.

**Slides 20 and 21**

Let us understand returning values from methods.

### Returning Values from Methods 1-2

**dt.** A method will return a value to the invoking method only when all the statements in the invoking method are complete, or when it encounters a return statement, or when an exception is thrown.

The `return` statement is written within the body of the method to return a value.

A `void` method will not have a return type specified in its method body.

A compiler error is generated when a `void` method returns a value.

You can store the value in a variable and specify the name of the variable with the `return` keyword.

© Aptech Ltd. Methods and Access Specifiers/Session 7 20

### Returning Values from Methods 2-2

- For example, the class `Circle` and its `getPI()` method can be modified as shown in code snippet:

```
public class Circle {
 // Declare and initialize value of PI
 private double PI = 3.14;
 // Method to retrieve value of PI
 public double getPI(){
 return PI;
 }
}
```

- In the modified class `Circle`, the value `3.14` is stored in a `private double` variable `PI`.
- Later, the method `getPI()` returns the value stored in the variable `PI` instead of the constant value `3.14`.

© Aptech Ltd. Methods and Access Specifiers/Session 7 21

Using slides 20 and 21, explain returning values from methods.

A method will return a value to the invoking method only when all the statements in the invoking method are complete, or when it encounters a return statement, or when an exception is thrown. The `return` statement is written within the body of the method to return a value.

A `void` method will not have a return type specified in its method body. A compiler error is generated when a `void` method returns a value. You can store the value in a variable and specify the name of the variable with the `return` keyword.

Explain the class `Circle` and its `getPI()` method can be modified as shown in code snippet mentioned in slide 21.

The type of data returned by a method must be compatible with the return type specified by the method. For example, if the return type of some method is Boolean, you could not return an integer. The variable receiving a value returned by the method must be compatible with the return type specified for the method. Parameters should be passed in sequence and they must be accepted by method in the same sequence.

**In-Class Question:**

After you finish explaining the returning values from methods, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



What happens when a `void` method returns a value?

**Answer:**

A compiler error is generated.

**Slides 22 to 24**

Let us understand declaring variable argument methods.

**Declaring Variable Argument Methods 1-3**

- Java provides a feature called `varargs` to pass variable number of arguments to a method.
- `varargs` is used when the number of a particular type of argument that will be passed to a method is not known until runtime.
- It serves as a shortcut to creating an array manually.
- To use `varargs`, the type of the last parameter is followed by ellipsis (...), then, a space, followed by the name of the parameter.
- This method can be called with any number of values for that parameter, including none.

◆ The syntax of a variable argument method is as follows:

**Syntax**

```
<method_name>(type ... variableName)
// method body
}
```

where,  
`...` : Indicates the variable number of arguments.

© Aptech Ltd. Methods and Access Specifiers/Session 7 22

**Declaring Variable Argument Methods 2-3**

- Following code snippet demonstrates an example of a variable argument method:

```
package session7;
public class Varargs {
 // Variable argument method taking variable number of integer arguments
 public void addNumber(int...num) {
 int sum=0;
 // Use for loop to iterate through num
 for(int i:num) {
 // Add up the values
 sum = sum + i;
 }
 // Print the sum
 System.out.println("Sum of numbers is "+sum);
 }
 public static void main(String[] args) {
 // Instantiate the Varargs class
 Varargs obj = new Varargs();
 // Invoke the addNumber() method with multiple arguments
 obj.addNumber(10,30,20,40);
 }
}
```

© Aptech Ltd. Methods and Access Specifiers/Session 7 23

### Declaring Variable Argument Methods 3-3



- Following figure shows the output of the code:

```

Output - Session7 (run)
run:
Sum of numbers is 100

```

- The class **Varargs** consists of a method called **addNumber(int num)**.
- The method accepts variable number of arguments of type integer.
- The method uses the enhanced for loop to iterate through the variable argument parameter **num** and adds each value with the variable **sum**.
- Finally, the method prints the value of **sum**.
- The **main()** method creates an object of the class and invokes the **addNumber()** method with multiple arguments of type integer.
- The output displays 100 after adding up the numbers.

© Aptech Ltd.

Methods and Access Specifiers /Session7

24

Using slides 22 to 24, explain declaring variable argument methods.

Java provides a feature called varargs to pass variable number of arguments to a method. varargs is used when the number of a particular type of argument that will be passed to a method is not known until runtime. It serves as a shortcut to creating an array manually.

To use varargs, the type of the last parameter is followed by ellipsis (...), then, a space, followed by the name of the parameter. This method can be called with any number of values for that parameter, including none.

Then, explain the syntax of a variable argument method as shown in the slide 22.

Explain the code snippet that demonstrates an example of a variable argument method as mentioned in slides 23 and 24.

Varargs is helper syntax and it enables the use of variable number of arguments in a method call. While invoking the varargs method, you can use any number of arguments and separated by comma. You can have only one varargs in a method and it must always be the last declared argument in the signature.

## Slides 25 to 35

Let us understand using **Javadoc** to lookup methods of a class.

### Using Javadoc to Lookup Methods of a Class 1-11

- Java provides a JDK tool named **Javadoc** that is used to generate API documentation as an HTML page from declaration and documentation comments.
- These comments are descriptions of the code written in a program.
- The different terminologies used while generating **Javadoc** are as follows:

|                                                                                                                                                                                                                                                                                                                                          |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>API documentation or API docs</b>                                                                                                                                                                                                                                                                                                     |
| <ul style="list-style-type: none"> <li>Are the online or hard copy descriptions of the API that are primarily intended for the programmers.</li> <li>API specification consists of all assertions for a proper implementation of the Java platform to ensure that the 'write once, run anywhere' feature of Java is retained.</li> </ul> |
| <b>Documentation comments or doc comments</b>                                                                                                                                                                                                                                                                                            |
| <ul style="list-style-type: none"> <li>Are special comments in the Java source code.</li> <li>Are written within the <code>/** ... */</code> delimiters.</li> <li>Are processed by the <b>Javadoc</b> tool for generating the API docs.</li> </ul>                                                                                       |

© Aptech Ltd.

Methods and Access Specifiers/Session 7

25

### Using Javadoc to Lookup Methods of a Class 2-11

- The four types of source files from which **Javadoc** tool can generate output are as follows:
  - Java source code files (`.java`) which consist of the field, class, constructor, method, and interface comments.
  - Package comment files that consist of package comments.
  - Overview comment files that contain comments about set of packages.
  - Miscellaneous files that are unprocessed such as images, class files, sample source codes, and any other file that is referenced from the previous files.
- A doc comment is written in HTML and it must precede a field, class, method, or constructor declaration.
- The doc comment consists of two parts namely, a description and block tags.
- For example, consider the class given in the following code snippet:

```
public class Circle {
 private double PI=3.14;
 public double calcArea(double rad) {
 return (3.14 * rad * rad);
 }
}
```

© Aptech Ltd.

Methods and Access Specifiers/Session 7

26

### Using Javadoc to Lookup Methods of a Class 3-11

- The code consists of a class **Circle**, a variable **PI**, and a method **calcArea()** that accepts radius as a parameter.
- Now, a doc comment for **calcArea()** method can be written as depicted in the following code snippet:

```
/*
 * Returns the area of a circle
 *
 * @param rad a variable indicating radius of a circle
 * @return the area of the circle
 * @see PI
 */

```

- First statement is the method description.
- @param**, **@return**, and **@see** are block tags that refer to the parameters and return value of the method.
- A blank comment line must be provided between the description line and block tags.

© Aptech Ltd.

Methods and Access Specifiers/Session 7

27

### Using Javadoc to Lookup Methods of a Class 4-11



- The HTML generated from running the **Javadoc** tool is as follows:

```
calcArea
public double calcArea(double rad)
Returns the area of a circle

Parameters:
 rad - a variable storing the radius of the circle

Returns:
 the area of a circle

See Also:
 PI
```

- To use **Javadoc** tool to lookup methods of a class, perform the following steps:

- 1 • Open the **Calculator** class created earlier.
- 2 • Open **Javadoc** widow by clicking **Window** → **Other** → **Javadoc**.

© Aptech Ltd.

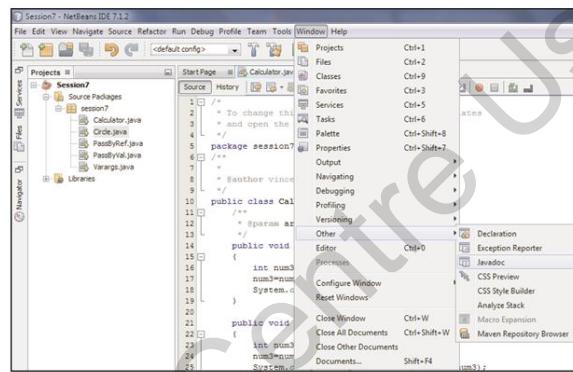
Methods and Access Specifiers/Session 7

28

### Using Javadoc to Lookup Methods of a Class 5-11



- This is shown in the following figure:



© Aptech Ltd.

Methods and Access Specifiers/Session 7

29

### Using Javadoc to Lookup Methods of a Class 6-11



- Following figure shows the **Javadoc** window opened at the bottom:



- The **Javadoc** window shows the description of the **Calculator** class.
- However, since **Javadoc** for the class is not generated still, it gives a message that '**Javadoc not found**'.

- 3 • Select the `println()` method of `System.out.println()` statement of the `add()` method and then, open the **Javadoc** window.

© Aptech Ltd.

Methods and Access Specifiers/Session 7

30

### Using Javadoc to Lookup Methods of a Class 7-11



- The window will show the built-in Java documentation of the `println()` method along with its description and parameters as defined in the `javadoc`.
- This is shown in the following figure:

The screenshot shows the NetBeans IDE interface. On the left, there's a tree view with 'Varargs.java' under 'Libraries'. The main editor area contains the following Java code:

```

10 public class Calculator {
11 /*
12 * @param args the command line arguments
13 */
14 public void add(int num1, int num2) {
15 int num3;
16 num3=num1+num2;
17 System.out.println("Result after addition is " + num3);
18 }
19
20 public void sub(int num1, int num2) {
21 int num3;
22 num3=num1-num2;
23 }
24 }
```

Below the code, a tooltip for `java.io.PrintStream.println()` is displayed:

**java.io.PrintStream**  
**public void println(String x)**  
 Prints a String and then terminate the line. This method behaves as though it invokes `PrintStream.print(String)` and then `PrintStream.println()`.  
**Parameters:**  
 x - The String to be printed.

© Aptech Ltd.

Methods and Access Specifiers/Session 7

31

### Using Javadoc to Lookup Methods of a Class 8-11



- 4 • Select the `add()` method. Again, no details about add method will be displayed since `javadoc` does not exist for it as shown in the following figure:

The screenshot shows the NetBeans IDE interface. The code editor displays the following Java code:

```

public void add(int num1, int num2)
```

- 5 • Type the following `javadoc` comments above the `add()` method:

```
/*
 * Displays the sum of two integers
 *
 * @param num1 an integer variable storing the value of first number
 * @param num2 an integer variable storing the value of second number
 * @return void
 */
```

© Aptech Ltd.

Methods and Access Specifiers/Session 7

32

### Using Javadoc to Lookup Methods of a Class 9-11



- 6 • Click Run → Generate Javadoc. The NetBeans IDE generates the `javadoc` for the project `Session7` and displays it in the browser as shown in the following figure:

The screenshot shows a web browser window titled 'Generated Documentation' with the URL 'file:///E:/Session7/dist/javadoc/index.html'. The page displays the 'session7' package structure and a 'Class Summary' table:

| Class      | Description |
|------------|-------------|
| Calculator |             |
| Circle     |             |
| PassByRef  |             |
| PassByVal  |             |
| Varargs    |             |

© Aptech Ltd.

Methods and Access Specifiers/Session 7

33

### Using Javadoc to Lookup Methods of a Class 10-11



- The **javadoc** lists the various classes available in the selected package.
- To move to the next or previous package, one can click the **Prev Package** and **Next Package** links on the page.

7

- Click the **Calculator** class under the **Class Summary** tab.
- The **javadoc** of the **Calculator** class is shown in the following figure:

```

public class Calculator
 extends java.lang.Object

Constructor Summary
Calculator()

Method Summary
Methods Modifier and Type Description
add(int num1, int num2) Displays the sum of two integers
subtract(int num1, int num2) Displays the difference of two integers
multiply(int num1, int num2) multiplies two integers

```

© Aptech Ltd.

Methods and Access Specifiers/Session 7

34

### Using Javadoc to Lookup Methods of a Class 11-11



- Similarly, **javadoc** for other classes can also be viewed.
- This shows the structure of the class, its constructor, and its various methods.

8

- Now, select the **add()** method in the **NetBeans** IDE and open the **Javadoc** window.
- The window will show the **javadoc** created for the **add()** method as shown in the following figure:

```

public void add(int num1, int num2)
 /*
 * Displays the sum of two integers
 *
 * @param num1 an integer variable storing the value of first number
 * @param num2 an integer variable storing the value of second number
 * @return void
 */
 int sum;
 sum=num1+num2;
 System.out.println("Total after addition is " + sum);
}

```

© Aptech Ltd.

Methods and Access Specifiers/Session 7

35

Using slides 25 to 35, explain how to use **Javadoc** to lookup methods of a class.

Java provides a JDK tool named **Javadoc** that is used to generate API documentation as an HTML page from declaration and documentation comments. These comments are descriptions of the code written in a program.

Then, explain the types of documentations that can be created using **Javadoc** namely API docs and doc comments.

The four types of source files from which **Javadoc** tool can generate output are as follows:

- Java source code files (**.java**) which consist of the field, class, constructor, method, and interface comments.
- Package comment files that consist of package comments.
- Overview comment files that contain comments about set of packages.
- Miscellaneous files that are unprocessed such as images, class files, sample source codes, and any other file that is referenced from the previous files.

A doc comment is written in HTML and it must precede a field, class, method, or constructor declaration. The doc comment consists of two parts namely, a description and block tags.

For example, explain the class given in the code snippet mentioned in slides 26 and 27.

Explain the HTML generated from running the **Javadoc** tool is mentioned in slide 28.

Then, explain the steps to use **Javadoc** tool to lookup methods of a class listed from slides 29 to 35.

The **Javadoc** tool parses the declaration and documentation comments in a set of Java source files and produces a corresponding set of HTML pages describing the public and protected classes, nested classes, interfaces, constructors, and methods.

#### Tips:

The **Javadoc** tool must be able to find all referenced classes like extensions or user classes because **Javadoc** tool builds its internal structure for the documentation and it loads all the referenced classes.

#### Slide 36

let us understand overview of access specifiers.



**Overview of Access Specifiers**

- ◆ Java provides a number of access specifiers or modifiers to set the level of access for a class and its members such as fields, methods, and constructors within the class.
- ◆ This is also known as the visibility of the class, field, and methods.
- ◆ When no access specifier is mentioned for a class member, the default accessibility is package or default.
- ◆ Using access specifiers provides the following advantages:

- Access specifiers help to control the access of classes and class members.
- Access specifiers help to prevent misuse of class details as well as hide the implementation details that are not required by other classes.
- The access specifiers also determine whether classes and the members of the classes can be invoked by other classes or interfaces.
- Accessibility affects inheritance and how members are inherited by the subclass.
- A package is always accessible by default.

© Aptech Ltd. Methods and Access Specifiers/Session 7 36

Using slide 36, explain overview of access specifiers.

Java provides a number of access specifiers or modifiers to set the level of access for a class and its members such as fields, methods, and constructors within the class. This is also known as the visibility of the class, field, and methods. When no access specifier is mentioned for a class member, the default accessibility is package or default.

Using access specifiers provides the following advantages:

- Access specifiers help to control the access of classes and class members.
- Access specifiers help to prevent misuse of class details as well as hide the implementation details that are not required by other classes.
- The access specifiers also determine whether classes and the members of the classes can be invoked by other classes or interfaces.
- Accessibility affects inheritance and how members are inherited by the subclass.
- A package is always accessible by default.

**Slides 37 to 39**

Let us understand types of access specifiers

### Types of Access Specifiers 1-3

- Java comes with four access specifiers namely, **public**, **private**, **protected**, and **default**.

|                  |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>public</b>    | <ul style="list-style-type: none"> <li>The <b>public</b> access specifier is the least restrictive of all access specifiers.</li> <li>A field, method, or class declared <b>public</b> is visible to any class in a Java application in the same package or in another.</li> </ul>                                                                                                                                                                                                                                                                                                                                                     |
| <b>private</b>   | <ul style="list-style-type: none"> <li>The <b>private</b> access specifier cannot be used for classes and interfaces as well as fields and methods of an interface.</li> <li>Fields and methods declared <b>private</b> cannot be accessed from outside the enclosing class.</li> </ul>                                                                                                                                                                                                                                                                                                                                                |
| <b>protected</b> | <ul style="list-style-type: none"> <li>The <b>protected</b> access specifier is used with classes that share a parent-child relationship which is referred to as inheritance.</li> <li>The <b>protected</b> keyword cannot be used for classes and interfaces as well as fields and methods of an interface.</li> <li>Fields and methods declared <b>protected</b> in a parent or super class can be accessed only by its child or subclass in another packages.</li> <li>Classes in the same package can also access protected fields and methods, even if they are not a subclass of the <b>protected</b> member's class.</li> </ul> |

© Aptech Ltd. Methods and Access Specifiers/Session 7 37

### Types of Access Specifiers 2-3

- Following figure shows the various access specifiers:

© Aptech Ltd. Methods and Access Specifiers/Session 7 38

### Types of Access Specifiers 3-3

- Following table shows the access level for different access specifiers:

| Access Specifier      | Class | Package | Subclass | World |
|-----------------------|-------|---------|----------|-------|
| <b>public</b>         | Y     | Y       | Y        | Y     |
| <b>protected</b>      | Y     | Y       | Y        | N     |
| No modifier (default) | Y     | Y       | N        | N     |
| <b>private</b>        | Y     | N       | N        | N     |

- The first column states whether the class itself has access to its own data members.
- As can be seen, a class can always access its own members.
- The second column states whether classes within the same package as the owner class (irrespective of their parentage) can access the member.
- As can be seen, all members can be accessed except **private** members.
- The third column states whether the subclasses of a class declared outside this package can access a member.
- In such cases, **public** and **protected** members can be accessed.
- The fourth column states whether all classes can access a data member.

© Aptech Ltd. Methods and Access Specifiers/Session 7 39

Using slides 37 to 39, explain types of access specifiers.

Java comes with four access specifiers namely, **public**, **private**, **protected**, and **default**.

Explain the access specifiers listed on slide 37 and also its related figure on slide 38. Explain the table that shows the access level for different access specifiers mentioned on slide 39.

Java allows you to control access to classes, methods, and fields with the help of access specifiers. This helps in encapsulation which concerns with hiding of data in a class and making this class available only through methods. In this way, the chance of making mistakes in changing values is minimized. Default access is more restricted than protected and protected members are accessible in subclass.

### In-Class Question:

After you finish explaining the types of access specifiers, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



What are the four Java access specifiers?

### Answer:

- public
- private
- protected
- default

### Slide 40

Let us understand rules for access control.

### Rules for Access Control



- ◆ Java has rules and constraints for usage of access specifiers as follows:
  - While declaring members, a **private** access specifier cannot be used with **abstract**, but it can be used with **final** or **static**.
  - No access specifier can be repeated twice in a single declaration.
  - A constructor when declared **private** will be accessible in the class where it was created.
  - A constructor when declared **protected** will be accessible within the class where it was created and in the inheriting classes.
  - private** cannot be used with fields and methods of an interface.
  - The most restrictive access level must be used that is appropriate for a particular member.
  - Mostly, a **private** access specifier is used at all times unless there is a valid reason for not using it.
  - Avoid using **public** for fields except for constants.

Using slide 40, explain the rules for access control.

**Slides 41 to 46**

Let us understand using access specifiers with variables and methods.

**Using Access Specifiers with Variables and Methods 1-6**

- ◆ The access specifiers discussed can be used with variables and methods of a class to restrict access from other classes.
- ◆ Following code snippet demonstrates an example of using access specifiers with variables and methods:

```
package session;
public class Employee {
 // Variables with default access
 int empID; // Variable to store employee ID
 String empName; // Variable to store employee name
 // Variables with private and protected access
 private String SSN; // Variable to store social security number
 protected String empDesig; // Variable to store designation
 /**
 * Parameterized constructor
 *
 * @param ID an integer variable storing the employee ID
 * @param name a String variable storing the employee name
 * @return void
 */
 public Employee(int ID, String name) {
 empID = ID;
 }
}
```

© Aptech Ltd.

Methods and Access Specifiers/Session 7

41

**Using Access Specifiers with Variables and Methods 2-6**

```
 empName = name;
}
// Define public methods
/**
 * Returns the value of SSN
 *
 * @return String
 */
public String getSSN(){ // Accessor for SSN
 return SSN;
}
/**
 * Sets the value of SSN
 *
 * @param ssn a String variable storing the social security number
 * @return void
 */
public void setSSN(String ssn) { // Mutator for SSN
 SSN = ssn;
}
```

© Aptech Ltd.

Methods and Access Specifiers/Session 7

42

**Using Access Specifiers with Variables and Methods 3-6**

```
 /**
 * Sets the value of Designation
 *
 * @param desig a String variable storing the employee designation
 * @return void
 */
 public void setDesignation(String desig) { // public method
 empDesig = desig;
 }

 /**
 * Displays employee details
 *
 * @return void
 */
 public void display(){ // public method
 System.out.println("Employee ID is "+ empID);
 System.out.println("Employee name is "+ empName);
 System.out.println("Designation is "+ empDesig);
 System.out.println("SSN is "+ SSN);
 }
}
```

© Aptech Ltd.

Methods and Access Specifiers/Session 7

43

### Using Access Specifiers with Variables and Methods 4-6



```
/*
 * @param args the command line arguments
 */
public static void main(String[] args) {
 // Instantiate the Employee class
 Employee objEmp1 = new Employee(1200,"Roger Stevens");
 // Assign values to public variables
 objEmp1.empDesig = "Manager";
 objEmp1.SSN = "281-72-3873";
 // Invoke the public method
 objEmp1.display();
}

◆ main() method creates an object of Employee class with values of empID and empName.
```

- ◆ Next, the values of `empDesig` and `SSN` are specified by directly accessing the variables with the object `objEmp1` even though `empDesig` is protected and `SSN` is private.
- ◆ This is because, `objEmp1` is an object present in the same class.
- ◆ So, `objEmp1` has access to data members with any access specifier.
- ◆ The `display()` method is used to display all the values as shown in the output.

© Aptech Ltd.

Methods and Access Specifiers/Session 7

44

### Using Access Specifiers with Variables and Methods 5-6



- ◆ Following figure shows the output of the code:

**Output - Session7 (run)**

```
run:
Employee ID is 1200
Employee name is Roger Stevens
Designation is Manager
SSN is 281-72-3873
```

- ◆ Following code snippet demonstrates the use of access specifiers in another class named `EmployeeDetails` but in the same package as `Employee` class:

```
public class EmployeeDetails {
 /*
 * @param args the command line arguments
 */
 public static void main(String[] args) {
 // Instantiate the Employee class within EmployeeDetails class
 Employee objEmp = new Employee(1300,"Clara Smith");
 // Assign value to protected variable
 objEmp.empDesig="Receptionist";
 // Use the mutator method to set the value of private variable
 objEmp.setSSN("282-72-3873");
 // Invoke the public method
 objEmp.display();
 }
}
```

© Aptech Ltd.

Methods and Access Specifiers/Session 7

45

### Using Access Specifiers with Variables and Methods 6-6



- ◆ Following figure shows the output of the code:

**Output - Session7 (run)**

```
run:
Employee ID is 1300
Employee name is Clara Smith
Designation is Receptionist
SSN is 282-72-3873
```

- ◆ The value of `empDesig` is specified by directly accessing the protected variable `empDesig`.
- ◆ This is because, a protected variable can be accessed by another class of the same package even if it is not a child class of `Employee`.
- ◆ However, to set the value of `SSN`, the `setSSN()` method is used.
- ◆ This is because `SSN` is a private member variable in `Employee` class and hence, cannot be directly accessed by other classes.
- ◆ Classes of other packages that are not child class of `Employee` can set the value of `empID` and `empName` by using the constructor, of `empDesig` and `SSN` by using `setDesignation()` and `setSSN()` methods.

© Aptech Ltd.

Methods and Access Specifiers/Session 7

46

Using slides 41 to 46, explain using access specifiers with variables and methods.

The access specifiers discussed can be used with variables and methods of a class to restrict access from other classes.

Explain the code snippet that demonstrates an example of using access specifiers with variables and methods mentioned in slides 41 to 46.

Access specifiers are used to control the visibility of members like classes, variables, and methods.

Accessibility of instance variables can be controlled with the access specifiers. When a variable is declared as public, it can be modified from all classes. If the variable is declared as private, access is restricted to class itself.

Access specifiers for methods works in same way as that of variables. Methods too can be defined using all the four access protection.

### Slide 47

Let us understand method overloading.

**Method Overloading**

- ◆ Consider the class **Calculator** created earlier.
  - ◆ Class has different methods for different operations.
  - ◆ However, all the methods add only integers.
- ◆ What if a user wants to add numbers of different types such as two floating-point numbers or one integer and other floating-point number?
- ◆ It would be more convenient to have a way to create different variations of the same **add()** method to add different types of values.
- ◆ The Java programming language provides the feature of method overloading to distinguish between methods with different method signatures.
- ◆ Using method overloading, multiple methods of a class can have the same name but with different parameter lists.
- ◆ Method overloading can be implemented in the following three ways:

**Changing the number of parameters**

**Changing the sequence of parameters**

**Changing the type of parameters**

© Aptech Ltd. Methods and Access Specifiers/Session 7 47

Using slide 47, explain how to overload methods.

Discuss the calculator scenario used for performing mathematical operations such as addition, subtraction, multiplication, and division. To add two numbers, the user normally selects the integer numbers. What if a user wants to add numbers of different types such as two floating-point numbers or one integer and other floating-point number?

It would be more convenient to have a way to create different variations of the same **add()** method to add different types of values. However, if calculator has to provide multiple buttons for each type of addition functionalities, then working with calculator will be cumbersome.

The Java programming language provides the feature of method overloading to distinguish between methods with different method signatures. Using method overloading, multiple methods of a class can have the same name, but with different parameter lists.

When the class has two or more methods of same name but different parameters, it is known as method overloading.

Method overloading can be implemented in the following three ways:

- Changing the number of parameters
- Changing the sequence of parameters
- Changing the type of parameters

If you have to perform only one operation, having same name of the methods increases the readability of the code.

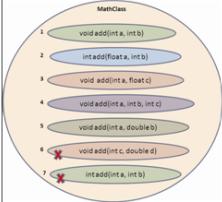
**Tips:**

In Java, method overloading is not possible by changing the return type of the method.

**Slide 48**

Let us understand overloading with different parameter list.

### Overloading with Different Parameter List



- ◆ Methods can be overloaded by changing the number or sequence of parameters of a method.
- ◆ Following figure shows an example of overloaded **add()** methods:

© Aptech Ltd.      Methods and Access Specifiers/Session 7      48

Using slide 48, explain overloading with different parameter list.

Methods can be overloaded by changing the number or sequence of parameters of a method. Explain the figure that shows an example of overloaded **add()** methods mentioned in slide 48.

When an overloaded method is called, Java look for a match between the arguments to call the method and the method's parameter. Sometimes, when the match is not found, Java automatic type conversion plays a vital role.

**Slides 49 to 53**

Let us understand overloading with different data types.

### Overloading with Different Data Types 1-5

- ◆ Methods can be overloaded by changing the data type of parameters of a method.
- ◆ Earlier figure shows an example of **add()** method overloaded by changing the type of parameters.
- ◆ Each of the **add()** methods numbered 1, 2, 3, and 5 accepts two parameters.
- ◆ However, they differ in the type of parameters that they accept.
- ◆ Notice that the **add()** method numbered 6 is similar to the method numbered 5.
  - ◆ Both the methods accept first an integer argument and then, a **float** argument as a parameter.
  - ◆ However, the parameter names are different.
  - ◆ This is not sufficient to make a method overloaded.
- ◆ The methods must differ in argument type and number and not simply in name of arguments.
- ◆ Similarly, the **add()** method numbered 7 has a signature similar to the method numbered 1.
  - ◆ Both the methods accept two integers, **a** and **b** as parameters.
  - ◆ However, the return type of method numbered 7 is **int** whereas that of method numbered 1 is **void**.

© Aptech Ltd.      Methods and Access Specifiers/Session 7      49

### Overloading with Different Data Types 2-5



- Following code snippet demonstrates an example of method overloading:

```
package session7;
public class MathClass {

 /**
 * Method to add two integers
 *
 * @param num1 an integer variable storing the value of first number
 * @param num2 an integer variable storing the value of second number
 * @return void
 */
 public void add(int num1, int num2) {
 System.out.println("Result after addition is "+ (num1+num2));
 }

 /**
 * Overloaded method to add three integers
 *
 * @param num1 an integer variable storing the value of first number
 * @param num2 an integer variable storing the value of second number
 * @param num3 an integer variable storing the value of third number
 * @return void
 */
}
```

© Aptech Ltd.

Methods and Access Specifiers/Session 7

50

### Overloading with Different Data Types 3-5



```
 /**
 * Overloaded method to add three integers
 *
 * @param num1 an integer variable storing the value of first number
 * @param num2 an integer variable storing the value of second number
 * @param num3 an integer variable storing the value of third number
 * @return void
 */
 public void add(int num1, int num2, int num3) {
 System.out.println("Result after addition is "+ (num1+num2+num3));
 }

 /**
 * Overloaded method to add a float and an integer
 *
 * @param num1 a float variable storing the value of first number
 * @param num2 an integer variable storing the value of second number
 * @return void
 */
 public void add(float num1, int num2) {
 System.out.println("Result after addition is "+ (num1+num2));
 }

 /**
 * Overloaded method to add a float and an integer accepting the values
 * in a different sequence
 *
 * @param num1 an integer variable storing the value of first number
 * @param num2 a float variable storing the value of second number
 * @return void
 */
}
```

© Aptech Ltd.

Methods and Access Specifiers/Session 7

51

### Overloading with Different Data Types 4-5



```
 * @return void
 */
 public void add(int num1, float num2) {
 System.out.println("Result after addition is "+ (num1+num2));
 }

 /**
 * Overloaded method to add two floating-point numbers
 *
 * @param num1 a float variable storing the value of first number
 * @param num2 a float variable storing the value of second number
 * @return void
 */
 public void add(float num1, float num2) {
 System.out.println("Result after addition is "+ (num1+num2));
 }

 /**
 * @param args the command line arguments
 */
 public static void main(String[] args) {

```

© Aptech Ltd.

Methods and Access Specifiers/Session 7

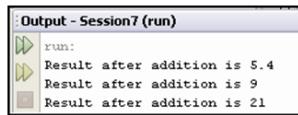
52

### Overloading with Different Data Types 5-5

```
//Instantiate the MathClass class
MathClass objMath = new MathClass();

//Invoke the overloaded methods with relevant arguments
objMath.add(3.4F, 2);
objMath.add(4, 5);
objMath.add(6, 7, 8);
}
```

- Following figure shows the output of the code:



- Compiler executes the appropriate **add()** method based on the type and number of arguments passed by the user.
- Output displays the result of addition of the different values.

© Aptech Ltd.

Methods and Access Specifiers/Session 7

53

Using slides 49 to 53, explain overloading with different data types.

Methods can be overloaded by changing the data type of parameters of a method. Earlier figure shows an example of **add()** method overloaded by changing the type of parameters. Each of the **add()** methods numbered 1, 2, 3, and 5 accepts two parameters. However, they differ in the type of parameters that they accept.

Then, explain the method numbered 6 and its similarities with method numbered 5. Discuss that the overloaded methods must differ in argument type and number and not simply in name of arguments. Similarly, the **add()** method numbered 7 has a signature similar to the method numbered 1.

Explain the code snippet demonstrates an example of method overloading mentioned in slides 50 to 53.

### Slides 54 to 59

Let us understand constructor overloading.

### Constructor Overloading 1-6



Constructor is a special method of a class that has the same name as the class name.

A constructor is used to initialize the variables of a class.

Similar to a method, a constructor can also be overloaded to initialize different types and number of parameters.

When the class is instantiated, the compiler invokes the constructor based on the number, type, and sequence of arguments passed to it.

- Following code snippet demonstrates an example of constructor overloading:

```
package session7;
public class Student {

 int rollNo; // Variable to store roll number
 String name; // Variable to store student name
 String address; // Variable to store address
 float marks; // Variable to store marks
```

© Aptech Ltd.

Methods and Access Specifiers/Session 7

54

### Constructor Overloading 2-6



```
/*
 * No-argument constructor
 *
 */
public Student(){
 rollNo = 0;
 name = "";
 address = "";
 marks = 0;
}

/*
 * Overloaded constructor
 *
 * @param rNo an integer variable storing the roll number
 * @param name a String variable storing student name
 */
public Student(int rNo, String sname) {
 rollNo = rNo;
 name = sname;
}
```

© Aptech Ltd.

Methods and Access Specifiers/Session 7

55

### Constructor Overloading 3-6



```
/*
 * Overloaded constructor
 *
 * @param rNo an integer variable storing the roll number
 * @param score a float variable storing the score
 */
public Student(int rNo, float score) {
 rollNo = rNo;
 marks = score;
}

/*
 * Overloaded constructor
 *
 * @param sName a String variable storing student name
 * @param addr a String variable storing the address
 */
public Student(String sName, String addr) {
 name = sName;
 address = addr;
}
```

© Aptech Ltd.

Methods and Access Specifiers/Session 7

56

### Constructor Overloading 4-6



```
/*
 * Overloaded constructor
 *
 * @param rNo an integer variable storing the roll number
 * @param sName a String variable storing student name
 * @param score a float variable storing the score
 */
public Student(int rNo, String sname, float score) {
 rollNo = rNo;
 name = sname;
 marks = score;
}

/*
 * Displays student details
 *
 * @return void
 */
public void displayDetails(){
 System.out.println("Rollno :" + rollNo);
 System.out.println("Student name:" + name);
 System.out.println("Address " + address);
}
```

© Aptech Ltd.

Methods and Access Specifiers/Session 7

57

### Constructor Overloading 5-6

```

 System.out.println("Score " + marks);
 System.out.println("-----");
 }

 /**
 * @param args the command line arguments
 */
 public static void main(String[] args) {

 // Instantiate the Student class with two string arguments Student
 objStud1 = new Student("David", "302, Washington Street");

 // Invoke the displayDetails() method
 objStud1.displayDetails();

 // Create other Student class objects and pass different
 // parameters to the constructor
 Student objStud2 = new Student(103, 46); objStud2.displayDetails();
 Student objStud3 = new Student(104, "Roger", 50);
 objStud3.displayDetails();
 }
}

```

© Aptech Ltd.

Methods and Access Specifiers/Session 7

58

### Constructor Overloading 6-6

- ◆ The class **Student** consists of member variables named **rollNo**, **name**, **address**, and **marks**.
- ◆ **Student ()** is the default or no-argument constructor of the **Student** class.
- ◆ The other constructors are overloaded constructors created by changing the number and type of parameters.
- ◆ Following figure shows the output of the program:

The screenshot shows the Java Output window with three entries. Each entry represents a student object with its roll number, name, address, and score. The first student has a name and address specified but a null roll number. The second student has a roll number and score specified but a null name and address. The third student has all values specified.

```

Output - Session7 (run)
=====
rollno :0
Student name:David
Address 302, Washington Street
Score 0.0

rollno :103
Student name:null
Address null
Score 46.0

rollno :104
Student name:Roger
Address null
Score 50.0

```

© Aptech Ltd.

Methods and Access Specifiers/Session 7

59

Using slides 54 to 59, explain constructor overloading.

Constructor is a special method of a class that has the same name as the class name. A constructor is used to initialize the variables of a class. Similar to a method, a constructor can also be overloaded to initialize different types and number of parameters. When the class is instantiated, the compiler invokes the constructor based on the number, type, and sequence of arguments passed to it.

Explain the code snippet demonstrates an example of constructor overloading mentioned in slides 54 to 59.

Constructor overloading is not much different than method overloading. Just like in method overloading you have multiple methods with same name but different signature, in constructor overloading you have multiple constructors with different signatures. The only difference is that the constructor does not have return type in Java.

One constructor can only be called from inside of another constructor and if called it must be first statement of that constructor. It means that Overloaded constructor must be called from other constructor only. The biggest advantage of overloaded constructor is flexibility which allows you to create object in different way.

**Slides 60 to 65**

Let us understand using 'this' keyword.

**Using 'this' Keyword 1-6**

- Java provides the keyword `this` which can be used in an instance method or a constructor to refer to the current object, that is, the object whose method or constructor is being called.

Any member of the current object can be referred from within an instance method or a constructor by using the `this` keyword.

The keyword `this` is not explicitly used in instance methods while referring to variables and methods of a class.

- For example, consider the method `calcArea()` of the following code snippet:

```
public class Circle {
 float area; // variable to store area of a circle

 /**
 * Returns the value of PI
 *
 * @return float
 */
 public float getPI(){
 return 3.14;
 }
}
```

© Aptech Ltd.

Methods and Access Specifiers/Session 7

60

**Using 'this' Keyword 2-6**

```
/**
 * Calculates area of a circle
 * @param rad an integer to store the radius
 * @return void
 */
public void calcArea(int rad) {
 this.area = getPI() * rad * rad;
}
}
```

- The method `calcArea()` calculates the area of a circle and stores it in the variable, `area`.
- It retrieves the value of `PI` by invoking the `getPI()` method.
- Here, the method call does not involve any object even though `getPI()` is an instance method.
- This is because of the implicit use of 'this' keyword.

© Aptech Ltd.

Methods and Access Specifiers/Session 7

61

**Using 'this' Keyword 3-6**

- For example, the method `calcArea()` can also be written as shown in the following code snippet:

```
public class Circle {
 float area; // Variable to store area of a circle
 /**
 * Returns the value of PI
 *
 * @return float
 */
 public float getPI(){
 return 3.14;
 }

 /**
 * Calculates area of a circle
 * @param rad an integer to store the radius
 * @return void
 */
 public void calcArea(int rad) {
 this.area = this.getPI() * rad * rad;
 }
}
```

© Aptech Ltd.

Methods and Access Specifiers/Session 7

62

### Using 'this' Keyword 4-6



- ◆ Notice the use of `this` to indicate the current object.
- ◆ The keyword `this` can also be used to invoke a constructor from within another constructor.
- ◆ This is also known as explicit constructor invocation as shown in the following code snippet:

```
public class Circle {
 private float rad; // Variable to store radius of a circle
 private float PI; // Variable to store value of PI

 /**
 * No-argument constructor
 *
 */
 public Circle() {
 PI = 3.14;
 }

 /**
 * Overloaded constructor
 *
 * @param r a float variable to store the value of radius

```

© Aptech Ltd.

Methods and Access Specifiers/Session 7

63

### Using 'this' Keyword 5-6



- ◆ The keyword `this` can be used to resolve naming conflicts when the names of actual and formal parameters of a method or a constructor are the same as depicted in the following code snippet:

```
public class Circle {
 // Variable to store radius of a circle
 private float rad; // line 1
 private float PI; // Variable to store value of PI

 /**
 * no-argument constructor
 *
 */
 public Circle(){
 PI = 3.14;
 }

 /**
 * Overloaded constructor
 * @param r a float variable to store the value of radius

```

© Aptech Ltd.

Methods and Access Specifiers/Session 7

64

### Using 'this' Keyword 6-6



- ◆ The code defines the constructor `Circle` with the parameter `rad` in line2 which is the formal parameter.
- ◆ Also, the parameter declared in line1 has the same name `rad` which is the actual parameter to which the user's value will be assigned at runtime.
- ◆ Now, while assigning a value to `rad` in the constructor, the user would have to write `rad = rad`.
- ◆ However, this would confuse the compiler as to which `rad` is the actual and which one is the formal parameter.
- ◆ To resolve this conflict, `this.rad` is written on the left of the assignment operator to indicate that it is the actual parameter to which value must be assigned.

© Aptech Ltd.

Methods and Access Specifiers/Session 7

65

Using slides 60 to 65, explain 'this' Keyword.

Java provides the keyword `this` which can be used in an instance method or a constructor to refer to the current object, that is, the object whose method or constructor is being called.

Any member of the current object can be referred from within an instance method or a constructor by using the `this` keyword. The keyword is not explicitly used in instance methods, while referring to variables and methods of a class.

For example, consider the method `calcArea()` of the following code snippet mentioned in slides 60 and 61.

For example, explain the method `calcArea()` can also be written in the code snippet mentioned in slides 62 and 63.

Explain the explicit constructor invocation as shown in the code snippet mentioned in slides 63 and 64.

Explain the keyword `this` can be used to resolve naming conflicts when the names of actual and formal parameters of a method or a constructor are the same as depicted in the code snippet mentioned in slides 64 and 65.

In Java, this is a reference variable that refers to the current object which means the object whose method or constructor being called. `this()` can be used to invoke current class constructor and implicitly invoke current class method. The `this()` constructor call should be used to reuse the constructor in the constructor.

**In-Class Question:**

After you finish explaining the ‘this’ Keyword, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



What is the use of `this` keyword?

**Answer:**

Any member of the current object can be referred from within an instance method or a constructor.

**Slide 66**

Let us summarize the session.

**Summary**



- ◆ A Java method is a set of statements grouped together for performing a specific operation.
- ◆ Parameters are the list of variables specified in a method declaration, whereas arguments are the actual values that are passed to the method when it is invoked.
- ◆ The variable argument feature is used in Java when the number of a particular type of arguments that will be passed to a method is not known until runtime.
- ◆ Java provides a JDK tool named [Javadoc](#), that is used to generate API documentation from documentation comments.
- ◆ Access specifiers are used to restrict access to fields, methods, constructor, and classes of an application.
- ◆ Java comes with four access specifiers namely, public, private, protected, and default.
- ◆ Using method overloading, multiple methods of a class can have the same name but with different parameter lists.
- ◆ Java provides the 'this' keyword which can be used in an instance method or a constructor to refer to the current object, that is, the object whose method or constructor is being invoked.

© Aptech Ltd. Methods and Access Specifiers/Session 7 66

In slide 66, you will summarize the session. End the session with a brief summary of what has been taught in the session.

### 7.3 Post Class Activities for Faculty

You should familiarize yourself with the topics of the next session which is teaching arrays and strings in Java.

**Tips:**

You can also check the Articles/Blogs/Expert Videos uploaded on the Online Varsity site to gain additional information related to the topics covered in the next session. You can also connect to online tutors on the Online Varsity site to ask queries related to the sessions.

# Session 8 – Arrays and Strings

## 8.1 Pre-Class Activities

Before you commence the session, you should familiarize yourself with the topics of this session in-depth.

Here, you can discuss the key points with the students that were covered in the previous session. Prepare a question or two which will help you to relate the current session objectives.

### 8.1.1 Objectives

By the end of this session, the learners will be able to:

- Describe an array
- Explain declaration, initialization, and instantiation of a single-dimensional array
- Explain declaration, initialization, and instantiation of a multi-dimensional array
- Explain the use of loops to process an array
- Describe ArrayList and accessing values from an ArrayList
- Describe String and StringBuilder classes
- Explain command line arguments
- Describe Wrapper classes, autoboxing, and unboxing

### 8.1.2 Teaching Skills

To teach this session, you should be well-versed with creation and use of arrays in Java and accessing values from ArrayList using loops. Also, familiarize yourself with command-line argument and working with String and StringBuilder classes.

You should teach the concepts in the theory class using the images provided. For teaching in the class, you are expected to use slides and LCD projectors.

#### Tips:

It is recommended that you test the understanding of the students by asking questions in between the class.

#### In-Class Activities:

Follow the order given here during In-Class activities.

**Overview of the Session:**

Give the students the overview of the current session in the form of session objectives. Show the students slide 2 of the presentation.

| Objectives                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"> <li>◆ Describe an array</li> <li>◆ Explain declaration, initialization, and instantiation of a single-dimensional array</li> <li>◆ Explain declaration, initialization, and instantiation of a multi-dimensional array</li> <li>◆ Explain the use of loops to process an array</li> <li>◆ Describe <u>ArrayList</u> and accessing values from an <u>ArrayList</u></li> <li>◆ Describe <u>String</u> and <u>StringBuilder</u> classes</li> <li>◆ Explain command line arguments</li> <li>◆ Describe Wrapper classes, <u>autoboxing</u>, and unboxing</li> </ul> |

© Aptech Ltd. Arrays and Strings/Session 8 2

Tell the students that this session introduces the creation and use of arrays in Java and accessing values from ArrayList using loops. Explain to the student that an array is a type of storage where same type of data can be stored or maintained in consecutive memory locations. This session focuses more on single-dimensional and two-dimensional arrays. The session also covers the various classes that are used to manipulate strings. String is a class that provides various methods to manipulate characters. Tell them session also describes wrapper classes, boxing, and un-boxing.

**8.2 In-Class Explanation****Slide 3**

Let us understand how to store multiple values in a variable.

| Introduction                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"> <li>◆ Consider a situation where in a user wants to store the marks of ten students.</li> <li>◆ For this purpose, the user can create ten different variables of type integer and store the marks in them.</li> <li>◆ What if the user wants to store marks of hundreds or thousands of students?</li> <li>◆ In such a case, one would need to create as many variables.</li> <li>◆ This can be a very difficult, tedious, and time consuming task.</li> <li>◆ Here, it is required to have a feature that will enable storing of all the marks in one location and access it with similar variable names.</li> <li>◆ Array, in Java, is a feature that allows storing multiple values of similar type in the same variable.</li> </ul> |

© Aptech Ltd. Arrays and Strings/Session 8 3

Using slide 3, explain how to store multiple values in a variable.

Consider a situation where in a user wants to store the marks of ten students. For this purpose, the user can create ten different variables of type integer and store the marks in them.

What if the user wants to store marks of hundreds or thousands of students? In such a case, one would need to create as many variables. This can be a very difficult, tedious, and time consuming task. Here, it is required to have a feature that will enable storing of all the marks in one location and access it with similar variable names.

Array, in Java, is a feature that allows storing multiple values of similar type in the same variable.

## Slides 4 to 6

Let us understand introduction to arrays.

### Introduction to Arrays 1-3

An array is a special data store that can hold a fixed number of values of a single type in contiguous memory locations.

It is implemented as objects.

The size of an array depends on the number of values it can store and is specified when the array is created.

After creation of an array, its size or length becomes fixed. Following figure shows an array of numbers:

The figure displays an array of ten integers storing values such as, 20, 100, 40, and so on.

© Aptech Ltd. Arrays and Strings/Session 8 4

### Introduction to Arrays 2-3

- Each value in the array is called an element of the array.
- The numbers 0 to 9 indicate the index or subscript of the elements in the array.
- The length or size of the array is 10. The first index begins with zero.
- Since the index begins with zero, the index of the last element is always length - 1.
- The last, that is, tenth element in the given array has an index value of 9.
- Each element of the array can be accessed using the subscript or index.
- Array can be created from primitive data types such as `int`, `float`, `boolean` as well as from reference type such as object.
- The array elements are accessed using a single name but with different subscripts.
- The values of an array are stored at contiguous locations in memory.
- This induces less overhead on the system while searching for values.

© Aptech Ltd. Arrays and Strings/Session 8 5

**Introduction to Arrays 3-3**



- The use of arrays has the following benefits:

Arrays are the best way of operating on multiple data elements of the same type at the same time.

Arrays make optimum use of memory resources as compared to variables.

Memory is assigned to an array only at the time when the array is actually used. Thus, the memory is not consumed by an array right from the time it is declared.

- Arrays in Java are of the following two types:



Single-dimensional arrays



Multi-dimensional arrays

© Aptech Ltd. Arrays and Strings/Session 8 6

Using slides 4 to 6, explain introduction to arrays.

An array is a special data store that can hold a fixed number of values of a single type in contiguous memory locations. It is implemented as objects. The size of an array depends on the number of values it can store and is specified when the array is created. After creation of an array, its size or length becomes fixed.

Explain the figure which shows an array of numbers mentioned in slide 4. Explain that each value in the array is called an element of the array. The numbers 0 to 9 indicate the index or subscript of the elements in the array. The length or size of the array is 10. The first index begins with zero. Since the index begins with zero, the index of the last element is always length - 1. The last, that is, tenth element in the given array has an index value of 9. Each element of the array can be accessed using the subscript or index. Array can be created from primitive data types such as `int`, `float`, `boolean` as well as from reference type such as `object`. The array elements are accessed using a single name but with different subscripts. The values of an array are stored at contiguous locations in memory. This induces less overhead on the system while searching for values.

The use of arrays has the following benefits:

- Arrays are the best way of operating on multiple data elements of the same type at the same time.
- Arrays make optimum use of memory resources as compared to variables.
- Memory is assigned to an array only at the time when the array is actually used. Thus, the memory is not consumed by an array right from the time it is declared.

Arrays in Java are of the following two types namely, Single-dimensional arrays and Multi-dimensional arrays.

An array is a container object that holds a fixed number of values of a single type. The length of an array is established when the array is created. After creation, its length is fixed. Each item in an array is called an element, and each element is accessed by numerical index.

Arrays are always fixed length abstracted data structure which cannot be altered when required. It helps the programmer to organize the same type of data into easily manageable format.

**Tips:**

An *array* is a container object that holds a fixed number of values of a single type. The length of an array is established when the array is created. After creation, its length is fixed.

**Slides 7 to 14**

Let us understand declaring, instantiating, and initializing single-dimensional array.

### Declaring, Instantiating, and Initializing Single-dimensional Array 1-8

A single-dimensional array has only one dimension and is visually represented as having a single column with several rows of data.

Each element is accessed using the array name and the index at which the element is located.

- Following figure shows the array named marks and its elements with their values and indices:

| marks[4] |       |
|----------|-------|
| Element  | Value |
| marks[0] | 65    |
| marks[1] | 47    |
| marks[2] | 75    |
| marks[3] | 50    |

- The size or length of the array is specified as 4 in square brackets '[]'.
- `marks[0]` indicates the first element in the array.
- `marks[3]`, that is, `marks[length-1]` indicates the last element of the array.
- Notice that there is no element with index 4.

© Aptech Ltd. Arrays and Strings/Session 8 7

### Declaring, Instantiating, and Initializing Single-dimensional Array 2-8

- An attempt to write `marks[4]` will issue an exception.

An exception is an abnormal event that occurs during the program execution and disrupts the normal flow of instructions.

- Array creation involves the following tasks:

#### Declaring an Array

- Declaring an array notifies the compiler that the variable will contain an array of the specified data type. It does not create an array.
- The syntax for declaring a single-dimensional array is as follows:

#### Syntax

```
datatype[] <array-name>;
```

where,

- `datatype`: Indicates the type of elements that will be stored in the array.
- `[]`: Indicates that the variable is an array.
- `array-name`: Indicates the name by which the elements of the array will be accessed.

© Aptech Ltd. Arrays and Strings/Session 8 8

### Declaring, Instantiating, and Initializing Single-dimensional Array 3-8

- For example,  
`int[] marks;`
- Similarly, arrays of other types can also be declared as follows:  
`byte[] byteArray;  
float[] floatsArray;  
boolean[] booleanArray;  
char[] charArray;  
String[] stringArray;`

#### Instantiating an Array

- Since array is an object, memory is allocated only when it is instantiated.
- The syntax for instantiating an array is as follows:

#### Syntax

```
datatype[] <array-name> = new datatype[size];
```

where,

- `new`: Allocates memory to the array.

© Aptech Ltd. Arrays and Strings/Session 8 9

## Declaring, Instantiating, and Initializing Single-dimensional Array 4-8



- size: Indicates the number of elements that can be stored in the array.
  - For example,  
`int[] marks = new int[4];`
- Initializing an Array**
- Since, array is an object that can store multiple values, array needs to be initialized with the values to be stored in it.
  - Array can be initialized in the following two ways:

**During creation:**

- To initialize a single-dimensional array during creation, one must specify the values to be stored while creating the array as follows: `int[] marks = {65, 47, 75, 50};`
- Notice that while initializing an array during creation, the new keyword or size is not required.
- This is because all the elements to be stored have been specified and accordingly the memory gets automatically allocated based on the number of elements.

## Declaring, Instantiating, and Initializing Single-dimensional Array 5-8



**After creation:**

- A single-dimensional array can also be initialized after creation and instantiation.
  - In this case, individual elements of the array need to be initialized with appropriate values.
  - For example,
    - `int[] marks = new int[4];`
    - `marks[0] = 65;`
    - `marks[1] = 47;`
    - `marks[2] = 75;`
    - `marks[3] = 50;`
  - Notice that in this case, the array must be instantiated and size must be specified.
  - This is because, the actual values are specified later and to store the values, memory must be allocated during creation of the array.
- Another way of creating an array is to split all the three stages as follows:
- ```
int marks[]; // declaration
marks = new int[4]; // instantiation
marks[0] = 65; // initialization
```

Declaring, Instantiating, and Initializing Single-dimensional Array 6-8



- Following code snippet demonstrates an example of single-dimensional array:

```
package session8;
public class OneDimension {
    //Declare a single-dimensional array named marks
    int marks[]; // line 1

    /**
     * Instantiates and initializes a single-dimensional array
     * @return void
     */
    public void storeMarks() {
        // Instantiate the array
        marks = new int[4]; // line 2
        System.out.println("Storing Marks. Please wait...");

        // Initialize array elements
        marks[0] = 65; // line 3
        marks[1] = 47;
        marks[2] = 75;
        marks[3] = 50;
    }
}
```

Declaring, Instantiating, and Initializing Single-dimensional Array 7-8

```


    /**
     * Displays marks from a single-dimensional array
     *
     * @return void
     */
    public void displayMarks() {
        System.out.println("Marks are:");

        // Display the marks
        System.out.println(marks[0]);
        System.out.println(marks[1]);
        System.out.println(marks[2]);
        System.out.println(marks[3]);
    }

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {

        //Instantiate class OneDimension
        OneDimension oneDimenObj = new OneDimension(); //line 4
    }
}


```

© Aptech Ltd.

Arrays and Strings/Session 8

13

Declaring, Instantiating, and Initializing Single-dimensional Array 8-8

```


        //Invoke the storeMarks() method
        oneDimenObj.storeMarks(); // line 5

        //Invoke the displayMarks() method
        oneDimenObj.displayMarks(); // line 6
    }
}

◆ The class OneDimension consists of an array named marks [] declared in line 1.
◆ To instantiate and initialize the array elements, the method storeMarks () is created.
◆ To display the array elements, the displayMarks () method is created.
◆ Following figure shows the output of the code:


```

```

Output - Session8 (run) x
run:
Storing Marks. Please wait...
Marks are:
65
47
75
50

```

© Aptech Ltd.

Arrays and Strings/Session 8

14

Using slides 7 to 14, explain declaring, instantiating, and initializing single-dimensional array.

A single-dimensional array has only one dimension and is visually represented as having a single column with several rows of data. Each element is accessed using the array name and the index at which the element is located.

Explain the figure which shows the array named marks and its elements with their values and indices mentioned in slide 7.

An exception is an abnormal event that occurs during the program execution and disrupts the normal flow of instructions.

Array creation involves the following tasks:

Declaring an Array

Declaring an array notifies the compiler that the variable will contain an array of the specified data type. It does not create an array.

Explain the syntax for declaring a single-dimensional array. Explain the following terms used in the syntax:

- datatype: specifies any valid data type in Java
- []: denotes that the variable is an array

- `arrayName`: denotes the array variable name that will be used for referring to the array
- `new`: allocates memory for objects
- `size`: denotes the number of elements that can be stored in an array

Tips:

An array having four columns is not called a four-dimensional array; it is still a two-dimensional array. The number of subscripts needed to access elements depends upon the dimensions of an array.

Tell them that arrays of other types can also be declared.

Array variable is declared in same way as any variable is declared. It has a type and a valid Java identifier. The type means the type of elements contain in an array. Also, it is essential to assign memory to an array when you declare it. Memory is assigned to set the size of declared array.

Instantiating an Array

Since array is an object, memory is allocated only when it is instantiated. Explain the syntax for instantiating the array using the `new` operator. The `new` operator is used for allocation of memory to the array object.

Initializing an Array

Since, array is an object that can store multiple values, array needs to be initialized with the values to be stored in it. Array can be initialized in the following two ways:

- **During creation:**

To initialize a single-dimensional array during creation, one must specify the values to be stored while creating the array as follows: `int[] marks = {65, 47, 75, 50};` Notice that while initializing an array during creation, the `new` keyword or size is not required. This is because all the elements to be stored have been specified and accordingly the memory gets automatically allocated based on the number of elements.

- **After creation:**

A single-dimensional array can also be initialized after creation and instantiation. In this case, individual elements of the array need to be initialized with appropriate values. For example,

```
int[] marks = new int[4];
marks[0] = 65;
marks[1] = 47;
marks[2] = 75;
marks[3] = 50;
```

Notice that in this case, the array must be instantiated and size must be specified.

This is because, the actual values are specified later and to store the values, memory must be allocated during creation of the array.

Explain the code snippet demonstrates an example of single-dimensional array mentioned in slides 12 to 14.

In-Class Question:

After you finish explaining the declaring, instantiating, and initializing single-dimensional array, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



Why there is a need to initialize array?

Answer:

Array is an object that can store multiple values hence, array needs to be initialized with the values to be stored in it.

Slides 15 to 21

Let us understand declaring, instantiating, and initializing multi-dimensional array.

Declaring, Instantiating, and Initializing Multi-dimensional Array 1-7

A multi-dimensional array in Java is an array whose elements are also arrays. This allows the rows to vary in length.

- The syntax for declaring and instantiating a multi-dimensional array is as follows:

Syntax

```
datatype[][] <array-name> = new datatype [rowsize][colszie];
```

where,

- datatype**: Indicates the type of elements that will be stored in the array.
- rowsize** and **colszie**: Indicates the number of rows and columns that the array will contain.
- new**: Keyword used to allocate memory to the array elements.

- For example,
`int[][] marks = new int[4][2];`
- The array named **marks** consists of four rows and two columns.

© Aptech Ltd. Arrays and Strings/Session 8 15

Declaring, Instantiating, and Initializing Multi-dimensional Array 2-7

- A multi-dimensional array can be initialized in the following two ways:

During creation

- To initialize a multi-dimensional array during creation, one must specify the values to be stored while creating the array as follows:
`int[][] marks = {{23,65}, {42,47}, {60,75}, {75,50}};`
- While initializing an array during creation, the elements in rows are specified in a set of curly brackets separated by a comma delimiter.
- Also, the individual rows are separated by a comma separator.
- This is a two-dimensional array that can be represented in a tabular form as shown in the following figure:

Rows	Columns	0	1
0	23	65	
1	42	47	
2	60	75	
3	75	50	

© Aptech Ltd. Arrays and Strings/Session 8 16

Declaring, Instantiating, and Initializing Multi-dimensional Array 3-7



After creation

A multi-dimensional array can also be initialized after creation and instantiation.

In this case, individual elements of the array need to be initialized with appropriate values.

Each element is accessed with a row and column subscript.

- For example,

```
int[][] marks = new int[4][2];
marks[0][0] = 23; // first row, first column
marks[0][1] = 65; // first row, second column
marks[1][0] = 42;
marks[1][1] = 47;
marks[2][0] = 60;
marks[2][1] = 75;
marks[3][0] = 75;
marks[3][1] = 50;
```

© Aptech Ltd.

Arrays and Strings/Session 8

17

Declaring, Instantiating, and Initializing Multi-dimensional Array 4-7



- The element 23 is said to be at position (0,0), that is, first row and first column.
- Therefore, to store or access the value 23, one must use the syntax `marks[0][0]`.
- Similarly, for other values, the appropriate row-column combination must be used.
- Similar to row index, column index also starts at zero. Therefore, in the given scenario, an attempt to write `marks[0][2]` would result in an exception as the column size is 2 and column indices are 0 and 1.
- Following code snippet demonstrates an example of two-dimensional array:

```
package session8;
public class TwoDimension {
    //Declare a two-dimensional array named marks
    int marks[][]; //line 1

    /**
     * Stores marks in a two-dimensional array
     *
     * @return void
     */
    public void storeMarks() {
```

© Aptech Ltd.

Arrays and Strings/Session 8

18

Declaring, Instantiating, and Initializing Multi-dimensional Array 5-7



```
// Instantiate the array
marks = new int[4][2]; // line 2
System.out.println("Storing Marks. Please wait...");

// Initialize array elements
marks[0][0] = 23; // line 3
marks[0][1] = 65;
marks[1][0] = 42;
marks[1][1] = 47;
marks[2][0] = 60;
marks[2][1] = 75;
marks[3][0] = 75;
marks[3][1] = 50;

/**
 * Displays marks from a two-dimensional array
 *
 * @return void
 */
public void displayMarks() {
```

© Aptech Ltd.

Arrays and Strings/Session 8

19

Declaring, Instantiating, and Initializing Multi-dimensional Array 6-7

```


    /**
     * Displays marks from a two-dimensional array
     *
     * @return void
     */
    public void displayMarks() {
        System.out.println("Marks are:");
        System.out.println("Roll no.1:" + marks[0][0] + "," + marks[0][1]);
        System.out.println("Roll no.2:" + marks[1][0] + "," + marks[1][1]);
        System.out.println("Roll no.3:" + marks[2][0] + "," + marks[2][1]);
        System.out.println("Roll no.4:" + marks[3][0] + "," + marks[3][1]);
    }

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        //Instantiate class TwoDimension
        TwoDimension twoDimenObj = new TwoDimension(); // line 4
    }
}


```

© Aptech Ltd.

Arrays and Strings/Session 8

20

Declaring, Instantiating, and Initializing Multi-dimensional Array 7-7

```


    //Invoke the storeMarks() method
    twoDimenObj.storeMarks();

    //Invoke the displayMarks() method
    twoDimenObj.displayMarks();
}


```

- Following figure shows the output of the code, that is, marks of four students are displayed from the array `marks[][]`:

```

Output - Session8 (run)
run:
Storing Marks. Please wait...
Marks are:
Roll no.1:23,65
Roll no.2:42,47
Roll no.3:60,75
Roll no.4:75,50
BUILD SUCCESSFUL (total time: 1 second)

```

© Aptech Ltd.

Arrays and Strings/Session 8

21

Using slides 15 to 21, explain declaring, instantiating, and initializing multi-dimensional array.

A multi-dimensional array in Java is an array whose elements are also arrays. This allows the rows to vary in length. When the elements of array are themselves arrays, it is multi-dimensional arrays.

Explain the syntax for declaring and instantiating a multi-dimensional array. Explain some of the terms in the syntax:

- datatype:** Indicates the type of elements that will be stored in the array.
- rowsize and colszie:** Indicates the number of rows and columns that the array will contain.
- new:** Keyword used to allocate memory to the array elements.

Then, explain the example, `int[][] marks = new int[4][2];` that declares an array named marks consists of four rows and two columns.

Then, explain how to initialize the multi-dimensional array in the following two ways:

During creation

To initialize a multi-dimensional array during creation, one must specify the values to be stored while creating the array as follows:

```
int[][] marks = {{23,65}, {42,47}, {60,75}, {75,50}};
```

While initializing an array during creation, the elements in rows are specified in a set of curly brackets separated by a comma delimiter. Also, the individual rows are separated by a comma separator.

After creation

A multi-dimensional array can also be initialized after creation and instantiation. In this case, individual elements of the array need to be initialized with appropriate values. Each element is accessed with a row and column subscript.

When you create multi-dimensional array using the `new` keyword, you always get a rectangular array: one in which all the array values for a given dimension have the same size.

Explain the code snippet demonstrates an example of two-dimensional array mentioned in slides 18 to 21.

Tips:

A quick way to initialize a two-dimensional array is to use nested-for loops.

Slides 22 to 27

Let us understand using loops to process and initialize an array.

Using Loops to Process and Initialize an Array 1-6



- ◆ A user can use loops to process and initialize an array.
- ◆ Following code snippet depicts the revised `displayMarks()` method of the single-dimensional array named `marks[]`:

```

...
public void displayMarks() {
    System.out.println("Marks are:");

    // Display the marks using for loop
    for(int count = 0; count < marks.length; count++) {
        System.out.println(marks[count]);
    }
}
...

```

- ◆ In the code, a `for` loop has been used to iterate the array from zero to `marks.length`.
- ◆ The property, `length`, of the array object is used to obtain the size of the array.
- ◆ Within the loop, each element is displayed by using the element name and the variable `count`, that is, `marks[count]`.

© Aptech Ltd. Arrays and Strings/Session 8 22

Using Loops to Process and Initialize an Array 2-6



- ◆ Following code snippet depicts the revised `displayMarks()` method of the two-dimensional array `marks[][]`:

```

...
public void displayMarks() {

    System.out.println("Marks are:");

    // Display the marks using nested for loop

    // outer loop
    for (int row = 0; row < marks.length; row++) {

        System.out.println("Roll no." + (row+1));

        // inner loop
        for (int col = 0; col < marks[row].length; col++) {
            System.out.println(marks[row][col]);
        }
    }
}
...
```

© Aptech Ltd. Arrays and Strings/Session 8 23

Using Loops to Process and Initialize an Array 3-6



- The outer loop keeps track of the number of rows and inner loop keeps track of the number of columns in each row.
- Following figure shows the output of the two-dimensional array `marks[][]`, after using the `for` loop:

```
Output - Session8 (run)
run:
Storing Marks. Please wait...
Marks are:
Roll no.1
23
65
Roll no.2
42
47
Roll no.3
60
75
Roll no.4
75
50
BUILD SUCCESSFUL (total time: 1 second)
```

© Aptech Ltd.

Arrays and Strings/Session 8

24

Using Loops to Process and Initialize an Array 4-6



- One can also use the enhanced `for` loop to iterate through an array.
- Following code snippet depicts the modified `displayMarks()` method of single-dimensional array `marks[]` using the enhanced `for` loop:

```
...
public void displayMarks() {
    System.out.println("Marks are:");
    // Display the marks using enhanced for loop
    for(int value:marks) {
        System.out.println(value);
    }
}
```

- The loop will print all the values of `marks[]` array till `marks.length` without having to explicitly specify the initializing and terminating conditions for iterating through the loop.

© Aptech Ltd.

Arrays and Strings/Session 8

25

Using Loops to Process and Initialize an Array 5-6



- Following code snippet demonstrates the calculation of total marks of each student by using the `for` loop and the enhanced `for` loop together with the two-dimensional array `marks[][]`:

```
...
public void totalMarks() {
    System.out.println("Total Marks are:");

    // Display the marks using for loop and enhanced for loop
    for (int row = 0; row < marks.length; row++) {
        System.out.println("Roll no." + (row+1));
        int sum = 0;

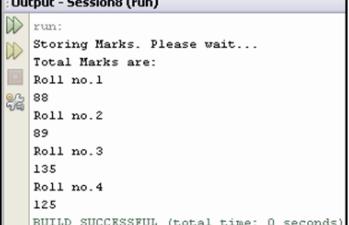
        // enhanced for loop
        for(int value:marks[row]) {
            sum = sum + value;
        }
        System.out.println(sum);
    }
}
```

© Aptech Ltd.

Arrays and Strings/Session 8

26

Using Loops to Process and Initialize an Array 6-6



- ◆ The enhanced `for` loop is used to iterate through the columns of the row selected in the outer loop using `marks[row]`.
- ◆ The code `sum = sum + value` will add up the values of all columns of the currently selected row.
- ◆ The selected row is indicated by the subscript variable `row`.
- ◆ Following figure shows the sum of the values of the two-dimensional array named `marks[][]` using `for` loop and enhanced `for` loop together:

Using slides 22 to 27, explain using loops to process and initialize an array.

A user can use loops to process and initialize an array. Explain the code snippet depicts the revised `displayMarks()` method of the single-dimensional array named `marks[]` mentioned in slide 22.

Explain the code snippet depicts the revised `displayMarks()` method of the two-dimensional array `marks[][]` mentioned in slides 23 and 24.

One can also use the enhanced `for` loop to iterate through an array.

Explain the code snippet depicts the modified `displayMarks()` method of single-dimensional array `marks[]` using the enhanced `for` loop mentioned in slide 25.

Explain the code snippet demonstrates the calculation of total marks of each student by using the `for` loop and the enhanced `for` loop together with the two-dimensional array `marks[][]` mentioned in slides 26 and 27.

Array can be processed using different types of loop. The main technique used to process array is `for` loop. A `for` loop is a way for processing each element of the array in a sequential manner.

Array can also be processed using `while` loops. `While` loop iterate through the loop body until termination condition evaluates to a false value. `While` loops are very useful when the data is not sequentially accessible with some sort of index.

Slides 28 to 35

Let us understand initializing an arraylist.

Initializing an ArrayList 1-8

One major disadvantage of an array is that its size is fixed during creation. The size cannot be modified later.

To resolve this issue, it is required to have a construct to which memory can be allocated based on requirement.

Also, addition and deletion of values can be performed easily. Java provides the concept of collections to address this problem.

A collection is a single object that groups multiple elements into a single unit.

- The core `Collection` interfaces that encapsulate different types of collections are shown in the following figure:

```

graph TD
    Collection --> Set
    Collection --> List
    Collection --> Queue
    Collection --> Map
    Set --> SortedSet
  
```

© Aptech Ltd. Arrays and Strings/Session 8 28

Initializing an ArrayList 2-8

- The general-purpose implementations are summarized in the following table:

Interfaces	Hash table	Resizable array	Tree	Linked list	Hash table + Linked list
Set	HashSet	-	TreeSet	-	LinkedHashSet
List	-	-	ArrayList	-	LinkedList
Queue	-	-	-	-	-
Map	HashMap	-	TreeMap	-	LinkedHashMap

- The `ArrayList` class is a frequently used collection that has the following characteristics:

- It is flexible and can be increased or decreased in size as needed.
- Provides several useful methods to manipulate the collection.
- Insertion and deletion of data is simpler.
- It can be traversed by using `for` loop, enhanced `for` loop, or other iterators.

© Aptech Ltd. Arrays and Strings/Session 8 29

Initializing an ArrayList 3-8

- `ArrayList` extends `AbstractList` and implements the interfaces such as `List`, `Cloneable`, and `Serializable`.
- The capacity of an `ArrayList` grows automatically.
- It stores all elements including null.
- The `ArrayList` collection provides methods to manipulate the size of the array.

- Following table lists the constructors of `ArrayList` class:

Constructor	Description
<code>ArrayList()</code>	Creates an empty array list.
<code>ArrayList(Collection c)</code>	Creates an array list initialized with the elements of the collection <code>c</code> .
<code>ArrayList(int capacity)</code>	Creates an array list with a specified initial capacity. Capacity is the size of the underlying array used to store the elements. The capacity can grow automatically as elements are added to an array list.

© Aptech Ltd. Arrays and Strings/Session 8 30

Initializing an ArrayList 4-8



- ◆ [ArrayList](#) consists of several methods for adding elements.
- ◆ These methods can be broadly divided into following two categories:

Methods that append one or more elements to the end of the list.

Methods that insert one or more elements at a position within the list.

- ◆ Following table lists the methods of [ArrayList](#) class:

Method	Description
<code>void add(int index, Object element)</code>	Inserts the specified element at the given index in this list. If <code>index >= size()</code> or <code>index < 0</code> , it throws <code>IndexOutOfBoundsException</code> .
<code>boolean add(Object o)</code>	Appends the specified element to the end of this list.
<code>boolean addAll(Collection c)</code>	Appends all elements in the specified collection to the end of this list. If the specified collection is null, it throws <code>NullPointerException</code> .
<code>boolean addAll(int index, Collection c)</code>	Inserts all of the elements in the specified collection into this list, starting at the specified index. If the collection is null, it throws <code>NullPointerException</code> .
<code>void clear()</code>	Removes all of the elements from this list.
<code>Object clone()</code>	Returns a copy of the <code>ArrayList</code> .

© Aptech Ltd.

Arrays and Strings/Session 8

31

Initializing an ArrayList 5-8



Method	Description
<code>boolean contains(Object o)</code>	Returns true if and only if the list contains the specified element.
<code>void ensureCapacity(int minCapacity)</code>	Increases the capacity of the <code>ArrayList</code> , if required, to ensure that it can store at least as many number of elements as indicated by the minimum capacity.
<code>Object get(int index)</code>	Returns the element at the specified index in this list. If <code>index >= size()</code> or <code>index < 0</code> , it throws <code>IndexOutOfBoundsException</code> .
<code>int indexOf(Object o)</code>	Returns the index of the first occurrence of the specified element in the list. If the element is not found, it returns -1.
<code>int lastIndexOf(Object o)</code>	Returns the index of the last occurrence of the specified element in this list. If the element is not found, it returns -1.
<code>Object remove(int index)</code>	Removes the element at the specified index in this list. If <code>index >= size()</code> or <code>index < 0</code> , it throws <code>IndexOutOfBoundsException</code> .
<code>protected void removeRange(int fromIndex, int toIndex)</code>	Removes all the elements between <code>fromIndex</code> , inclusive and <code>toIndex</code> , exclusive of the list.
<code>Object set(int index, Object element)</code>	Replaces the element at the specified index in this list with the newly specified element. If <code>index >= size()</code> or <code>index < 0</code> , it throws <code>IndexOutOfBoundsException</code> .

© Aptech Ltd.

Arrays and Strings/Session 8

32

Initializing an ArrayList 6-8



Method	Description
<code>int size()</code>	Returns the number of elements in this list.
<code>Object[] toArray()</code>	Returns an array containing all of the elements in the list in the correct order. If the array is null, it throws <code>NullPointerException</code> .
<code>Object[] toArray(Object[] a)</code>	Returns an array containing all of the elements in the list in the correct order. The type of the returned array is same as that of the specified array.
<code>void trimToSize()</code>	Trims the capacity of the <code>ArrayList</code> to the list's actual size.

- ◆ To traverse an `ArrayList`, one can use one of the following approaches:

➡ A **for loop**

➡ An **enhanced for loop**

➡ **Iterator**

➡ **ListIterator**

© Aptech Ltd.

Arrays and Strings/Session 8

33

Initializing an ArrayList 7-8

Iterator interface provides methods for traversing a set of data.

It can be used with arrays as well as various classes of the Collection framework.

- The **Iterator** interface provides the following methods for traversing a collection:
 - next ()** • This method returns the next element of the collection.
 - hasNext ()** • This method returns true if there are additional elements in the collection.
 - remove ()** • This method removes the element from the list while iterating through the collection.
- There are no specific methods in the **ArrayList** class for sorting.
- However, one can use the **sort ()** method of the **Collections** class to sort an **ArrayList**.

© Aptech Ltd. Arrays and Strings/Session 8 34

Initializing an ArrayList 8-8

- The syntax for using the **sort ()** method is as follows:

Syntax

```
Collections.sort(<list-name>);
```

- Following code snippet demonstrates instantiation and initialization of an **ArrayList**:

```
ArrayList marks = new ArrayList(); // Instantiate an ArrayList
marks.add(67); // Initialize an ArrayList
marks.add(50);
```

© Aptech Ltd. Arrays and Strings/Session 8 35

Using slides 28 to 35, explain initializing an arraylist.

One major disadvantage of an array is that its size is fixed during creation. The size cannot be modified later. To resolve this issue, it is required to have a construct to which memory can be allocated based on requirement. Also, addition and deletion of values can be performed easily. Java provides the concept of collections. A collection is a single object that groups multiple elements into a single unit.

Explain the figure of the core collection interfaces that encapsulate different types of collections mentioned on slide 28.

Then, explain the **ArrayList** class that can be increased or decreased in size as needed. It provides several useful methods to manipulate the collection and can be traversed by using for loop, enhanced for loop, or other iterators.

Explain the table lists the constructors of **ArrayList** class in slide 30.

`ArrayList` consists of several methods for adding elements. These methods can be broadly divided into following two categories:

- Methods that append one or more elements to the end of the list.
- Methods that insert one or more elements at a position within the list.

Explain the table lists the methods of `ArrayList` class on slides 31 to 33. To traverse an `ArrayList`, one can use one of the following approaches:

- A for loop
- An enhanced for loop
- Iterator
- ListIterator

Then, explain `Iterator` interface provides methods for traversing a set of data. It can be used with arrays as well as various classes of the Collection framework.

Explain the code snippet demonstrates instantiation and initialization of an `ArrayList` mentioned in slide 35.

Array lists are created with an initial size. When this size is exceeded, the collection is automatically enlarged. When objects are removed, the array may be shrunk. They can dynamically increase or decrease in size. The disadvantage of Array lists is that it holds only object type and not primitive type like int.

Tips:

If you want to reduce the size of the array that underlies an `ArrayList` object, so that it is precisely as large as the number of items that it is currently holding, call `trimToSize()` method.

In-Class Question:

After you finish explaining the initializing an `ArrayList`, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



Which methods provide by `Iterator` interface for traversing a collection?

Answer:

- `next()`
- `hasNext()`
- `remove()`

Slides 36 to 40

Let us understand accessing values in an ArrayList.

Accessing Values in an ArrayList 1-5

- An `ArrayList` can be iterated by using the `for` loop or by using the `Iterator` interface.
- Following code snippet demonstrates the use of `ArrayList` named `marks` to add and display marks of students:

```
package session8;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Iterator;
public class ArrayLists{
    // Create an ArrayList instance
    ArrayList marks = new ArrayList(); // line 1

    /**
     * Stores marks in ArrayList
     *
     * @return void
     */
    public void storeMarks(){
        System.out.println("Storing marks. Please wait...");
        marks.add(67); // line 2
    }
}
```

© Aptech Ltd.

Arrays and Strings/Session 8

36

Accessing Values in an ArrayList 2-5

```
marks.add(50);
marks.add(45);
marks.add(75);
}

/**
 * Displays marks from ArrayList
 *
 * @return void
 */
public void displayMarks() {
    System.out.println("Marks are:");

    // iterating the list using for loop
    System.out.println("Iterating ArrayList using for loop:");
    for (int i = 0; i < marks.size(); i++) {
        System.out.println(marks.get(i));
    }
    System.out.println("-----");
}
```

© Aptech Ltd.

Arrays and Strings/Session 8

37

Accessing Values in an ArrayList 3-5

```
// Iterate the list using Iterator interface
Iterator imarks = marks.iterator(); // line 3
System.out.println("Iterating ArrayList using Iterator:");
while (imarks.hasNext()) { // line 4
    System.out.println(imarks.next()); // line 5
}
System.out.println("-----");

// Sort the list
Collections.sort(marks); // line 6
System.out.println("Sorted list is: " + marks);
}

/**
 * @param args the command line arguments
 */
public static void main(String[] args) {

    //Instantiate the class OneDimension
    ArrayLists obj = new ArrayLists(); // line 7
}
```

© Aptech Ltd.

Arrays and Strings/Session 8

38

Accessing Values in an ArrayList 4-5



```
//Invoke the storeMarks() method
obj.storeMarks();

//Invoke the displayMarks() method
obj.displayMarks();
}
}
```

- ◆ The **Iterator** object **imarks** is instantiated in line 3 and attached with the marks **ArrayList** using **marks.iterator()**.
- ◆ It is used to iterate through the collection.
- ◆ The **Iterator** interface provides the **hasNext()** method to check if there are any more elements in the collection as shown in line 4.
- ◆ The method, **next()** is used to traverse to the next element in the collection.
- ◆ The retrieved element is then displayed to the user in line 5.
- ◆ The static method, **sort()** of **Collections** class is used to sort the **ArrayList** **marks** in line 6 and print the values on the screen.

Accessing Values in an ArrayList 5-5



- ◆ Following figure shows the output of the code:

```
Output - Session8 (run)
run:
Storing marks. Please wait...
Marks are:
Iterating ArrayList using for loop:
67
50
45
75
-----
Iterating ArrayList using Iterator:
67
50
45
75
-----
Sorted list is: [45, 50, 67, 75]
```

The values of an **ArrayList** can also be printed by simply writing `System.out.println("Marks are:" + marks)`.
In this case, the output would be: Marks are: [67, 50, 45, 75].

Using slides 36 to 40, explain accessing values in an **ArrayList**.

An **ArrayList** can be iterated by using the **for** loop or by using the **Iterator** interface.

Explain the code snippet demonstrates the use of **ArrayList** named **marks** to add and display marks of students mentioned in slides 36 to 40.

Iterator enables you to cycle through the collections, obtaining, or removing elements. Before you access a values through an iterator, you must obtain it. Each of the collection classes provides an **iterator()** method that returns an iterator to the start of the collection. By using this **iterator** object, you can access each element, one at a time.

Slide 41

Let us understand introduction to strings.

Introduction to Strings

- Consider a scenario, where in a user wants to store the name of a person.
- One can create a character array as shown in the following code snippet:

```
char[] name = {'J', 'u', 'l', 'i', 'a'}
```

- Similarly, to store names of multiple persons, one can create a two-dimensional array.
- However, the number of characters in an array must be fixed during creation.
- This is not possible since the names of persons may be of variable sizes.
- Also, manipulating the character array would be tedious and time consuming.
- Java provides the `String` data type to store multiple characters without creating an array.

© Aptech Ltd. Arrays and Strings/Session 8 41

Using slide 41, explain introduction to Strings. Consider a scenario, where in a user wants to store the name of a person. One can create a character array as shown in the following code snippet:

```
char[] name = {'J', 'u', 'l', 'i', 'a'}
```

Similarly, to store names of multiple persons, one can create a two-dimensional array. However, the number of characters in an array must be fixed during creation. This is not possible since the names of persons may be of variable sizes.

Also, manipulating the character array would be tedious and time consuming. Java provides the `String` data type to store multiple characters without creating an array.

Java strings are a series of characters gathered together. It stores text and a sequence of characters. Each character in a string is a 16-bit Unicode character, to provide rich set of characters.

Slides 42 to 44

Let us understand the features of strings.

Strings 1-3

- String literals such as "Hello" in Java are implemented as instances of the `String` class.
- Strings are constant and immutable, that is, their values cannot be changed once they are created.
- String buffers allow creation of mutable strings.
- A simple `String` instance can be created by enclosing a string literal inside double quotes as shown in the following code snippet:

```
...
String name = "Mary";
// This is equivalent to:
char name[] = {'M', 'a', 'r', 'y'};
...
```

- An instance of a `String` class can also be created using the `new` keyword, as shown in code snippet:


```
String str = new String();
```
- The code creates a new object of class `String`, and assigns its reference to the variable `str`.

© Aptech Ltd. Arrays and Strings/Session 8 42

Strings 2-3



- Java also provides special support for concatenation of strings using the plus (+) operator and for converting data of other types to strings as depicted in the following code snippet:

```
...
String str = "Hello"; String str1 = "World";
// The two strings can be concatenated by using the operator '+'
System.out.println(str + str1);

// This will print 'HelloWorld' on the screen
...
```

- One can convert a character array to a string as depicted in the following code snippet:

```
char[] name = {'J', 'o', 'h', 'n'}; String empName = new String(name);
```

The `java.lang.String` class is a final class, that is, no class can extend it.

The `java.lang.String` class differs from other classes, in that one can use '+=' and '+' operators with String objects for concatenation.

© Aptech Ltd.

Arrays and Strings/Session 8

43

Strings 3-3



- If the string is not likely to change later, one can use the `String` class.
- Thus, a `String` class can be used for the following reasons:

String is immutable and so it can be safely shared between multiple threads.

The threads will only read them, which is normally a thread safe operation.

- If the string is likely to change later and it will be shared between multiple threads, one can use the `StringBuffer` class.
- The use of `StringBuffer` class ensures that the string is updated correctly.
- However, the drawback is that the method execution is comparatively slower.
- If the string is likely to change later but will not be shared between multiple threads, one can use the `StringBuilder` class.

- The `StringBuilder` class can be used for the following reasons:

It allows modification of the strings without the overhead of synchronization.

Methods of `StringBuilder` class execute as fast as, or faster, than those of the `StringBuffer` class

© Aptech Ltd.

Arrays and Strings/Session 8

44

Using slides 42 to 44, explain the features of strings.

String literals such as "Hello" in Java are implemented as instances of the `String` class. Strings are constant and immutable, that is, their values cannot be changed once they are created. Thus, string buffers can be used, as they allow creation of mutable strings.

A simple `String` instance can be created by enclosing a string literal inside double quotes as shown in the code snippet in slide 42. Java also provides special support for concatenation of strings using the plus (+) operator and for converting data of other types to strings as depicted in the code snippet in slide 43.

The `java.lang.String` class is a final class, that is, no class can extend it. This class differs from other classes, in that one can use '+=' and '+' operators with `String` objects for concatenation.

If the string is not likely to change later, one can use the `String` class. The threads will only read them, which is normally a thread safe operation.

If the string is likely to change later and it will be shared between multiple threads, one can use the `StringBuffer` class. The use of `StringBuffer` class ensures that the string is updated correctly. However, the drawback is that the method execution is comparatively slower.

If the string is likely to change later, but will not be shared between multiple threads, one can use the `StringBuilder` class.

In Java, strings are objects and not a primitive type. Even string constants are string objects. As you know, `String` is immutable and modifying the contents of one string can cause adverse effects on other string.

JDK provides two classes to support mutable strings, `StringBuffer` and `StringBuilder`. Both are just like any other ordinary object which are stored in the heap and not shared and therefore can be modified without causing any effects to other objects.

Tips:

`StringBuffer` is an ordinary object. You need to use a constructor to create a `StringBuffer` and '+' operator cannot be applied to object including `StringBuffer`.

Slides 45 to 51

Let us understand working with `String` class.

Working with String Class 1-7

- Some of the frequently used methods of `String` class are as follows:
 - length(String str)**
 - The `length()` method is used to find the length of a string. For example,
 - `String str = "Hello";`
 - `System.out.println(str.length()); // output: 5`
 - charAt(int index)**
 - The `charAt()` method is used to retrieve the character value at a specific index.
 - The index ranges from zero to `length() - 1`.
 - The index of the first character starts at zero. For example,
 - `System.out.println(str.charAt(2)); // output: 'l'`
 - concat(String str)**
 - The `concat()` method is used to concatenate a string specified as argument to the end of another string.
 - If the length of the string is zero, the original `String` object is returned, otherwise a new `String` object is returned.
 - `System.out.println(str.concat("World")); // output: "HelloWorld"`

© Aptech Ltd. Arrays and Strings/Session 8 45

Working with String Class 2-7

- compareTo(String str)**
 - The `compareTo()` method is used to compare two `String` objects.
 - The comparison returns an integer value as the result.
 - The comparison is based on the Unicode value of each character in the strings.
 - The result will return a negative value, if the argument string is alphabetically greater than the original string.
 - The result will return a positive value, if argument string is alphabetically lesser than the original string and the result will return a value of zero, if both the strings are equal.
 - For example,
 - `System.out.println(str.compareTo("World")); // output: -15`
 - The output is 15 because, the second string "World" begins with 'W' which is alphabetically greater than the first character 'H' of the original string, `str`.
 - The difference between the position of 'H' and 'W' is 15.
 - Since 'H' is smaller than 'W', the result will be -15.
- indexOf(String str)**
 - The `indexOf()` method returns the index of the first occurrence of the specified character or string within a string.
 - If the character or string is not found, the method returns -1. For example,
 - `System.out.println(str.indexOf("e")); // output: 1`

© Aptech Ltd. Arrays and Strings/Session 8 46

Working with String Class 3-7



lastIndexOf(String str)

- The `lastIndexOf()` method returns the index of the last occurrence of a specified character or string from within a string.
- The specified character or string is searched backwards that is the search begins from the last character. For example,

```
System.out.println(str.lastIndexOf("l")); // output: 3
```

replace(char old, char new)

- The `replace()` method is used to replace all the occurrences of a specified character in the current string with a given new character.
- If the specified character does not exist, the reference of original string is returned. For example,

```
System.out.println(str.replace('e','a'));
// output: 'Hello'
```

substring(int beginIndex, int endIndex)

- The `substring()` method is used to retrieve a part of a string, that is, substring from the given string.
- One can specify the start index and the end index for the substring.
- If end index is not specified, all characters from the start index to the end of the string will be returned. For example,

```
System.out.println(str.substring(2,5)); // output: 'llo'
```

© Aptech Ltd. Arrays and Strings/Session 8 47

Working with String Class 4-7



toString()

- The `toString()` method is used to return a `String` object.
- It is used to convert values of other data types into strings. For example,

```
Integer length = 5;
System.out.println(length.toString()); // output: 5
```

- Notice that the output is still 5. However, now it is represented as a string instead of an integer.

trim()

- The `trim()` method returns a new string by trimming the leading and trailing whitespace from the current string. For example,

```
String str1 = " Hello ";
System.out.println(str1.trim()); // output: 'Hello'
```

- The `trim()` method will return 'Hello' after removing the spaces.

© Aptech Ltd. Arrays and Strings/Session 8 48

Working with String Class 5-7



- Following code snippet demonstrates the use of `String` class methods:

```
public class Strings {
    String str = "Hello"; // Initialize a String variable
    Integer strLength = 5; // Use the Integer wrapper class

    /**
     * Displays strings using various String class methods
     */
    public void displayStrings(){
        // using various String class methods
        System.out.println("String length is:"+ str.length());
        System.out.println("Character at index 2 is:"+ str.charAt(2));
        System.out.println("Concatenated string is:"+ str.concat("World"));
        System.out.println("String comparison is:"+ str.compareTo("World"));
        System.out.println("Index of o is:"+ str.indexOf("o"));
        System.out.println("Last index of l is:"+ str.lastIndexOf("l"));
        System.out.println("Replaced string is:"+ str.replace('e','a'));
        System.out.println("Substring is:"+ str.substring(2, 5));
    }
}
```

© Aptech Ltd. Arrays and Strings/Session 8 49

Working with String Class 6-7

```

System.out.println("Integer to String is:"+ strLength.toString() );
String str1=" Hello ";
System.out.println("Untrimmed string is:"+ str1);
System.out.println("Trimmed string is:"+ str1.trim());
}

/**
 * @param args the command line arguments
 */
public static void main(String[] args) {

    //Instantiate class, Strings
    Strings objString = new Strings(); // line 1

    //Invoke the displayStrings() method
    objString.displayStrings();
}

```

Working with String Class 7-7

- Following figure shows the output of the Strings.java class:

```

Output - Session8 (run)
run:
String length is:5
Character at index 2 is:1
Concatenated string is:HelloWorld
String comparison is:-15
Index of o is:4
Last index of l is:3
Replaced string is:Hallo
Substring is:lo
Integer to String is:5
Untrimmed string is: Hello
Trimmed string is:Hello

```

Using slides 45 to 51, explain working with String class.

Explain some of the frequently used methods of String class mentioned on slides 45 to 48.

Explain the code snippet demonstrates the use of String class methods mentioned in slides 49 to 51.

The class `String` includes methods for examining individual characters of the sequence, for comparing strings, for searching strings, for extracting substrings, and for creating a copy of a string. `String` class provides methods for dealing with characters and `char` values.

Slides 52 and 53

Let us understand working with `StringBuilder` class.

Working with `StringBuilder` Class 1-2



- `StringBuilder` objects are similar to `String` objects, except that they are mutable and flexible.
- Internally, the system treats these objects as a variable-length array containing a sequence of characters.
- The length and content of the sequence of characters can be changed through methods available in the `StringBuilder` class.
- For concatenating a large number of strings, using a `StringBuilder` object is more efficient.
- The `StringBuilder` class also provides a `length()` method that returns the length of the character sequence in the class.
- Unlike strings a `StringBuilder` object also has a property `capacity` that specifies the number of character spaces that have been allocated.
- The capacity is returned by the `capacity()` method and is always greater than or equal to the length.
- The capacity will automatically expand to accommodate the new strings when added to the string builder.
- `StringBuilder` object allows insertion of characters and strings as well as appending characters and strings at the end.

© Aptech Ltd. Arrays and Strings/Session 8 52

Working with `StringBuilder` Class 2-2



- The constructors of the `StringBuilder` class are as follows:

<code>StringBuilder()</code>	• Default constructor that provides space for 16 characters.
<code>StringBuilder(int capacity)</code>	• Constructs an object without any characters in it. • However, it reserves space for the number of characters specified in the argument, capacity.
<code>StringBuilder(String str)</code>	• Constructs an object that is initialized with the contents of the specified string, str.

© Aptech Ltd. Arrays and Strings/Session 8 53

Using slides 52 and 53, explain working with `StringBuilder` class.

Explain the features of `StringBuilder` class listed on slide 52. The constructors of the `StringBuilder` class are as follows:

- `StringBuilder()` - Default constructor that provides space for 16 characters. It creates an empty `StringBuilder`.
- `StringBuilder(int capacity)` - Constructs an object without any characters in it. However, it reserves space for the number of characters specified in the argument, capacity.
- `StringBuilder (String str)` - Constructs an object that is initialized with the contents of the specified string, str.

`StringBuilder` objects are like `String` objects, except they can be modified. This class provides an API compatible with `StringBuffer`, but no guarantee of synchronization. This class is faster under most of the implementations.

The principal operations on a `StringBuilder` class are `insert()` and `append()` methods which are overloaded so as to accept data of any type.

Slides 54 to 58

Let us understand methods of `StringBuilder` class.

Methods of `StringBuilder` Class 1-5

- The `StringBuilder` class provides several methods for appending, inserting, deleting, and reversing strings as follows:

append()

- The `append()` method is used to append values at the end of the `StringBuilder` object.
- This method accepts different types of arguments, including `char`, `int`, `float`, `double`, `boolean`, and so on, but the most common argument is of type `String`.
- For each `append()` method, `String.valueOf()` method is invoked to convert the parameter into a corresponding string representation value and then the new string is appended to the `StringBuilder` object.
- For example,

```
StringBuilder str = new StringBuilder("JAVA ");
str.append("SE ");
System.out.println(str);
// output: JAVA SE
```

© Aptech Ltd. Arrays and Strings/Session 8 54

Methods of `StringBuilder` Class 2-5

insert()

- The `insert()` method is used to insert one string into another.
- It calls the `String.valueOf()` method to obtain the string representation of the value.
- The new string is inserted into the invoking `StringBuilder` object.
- The `insert()` method has several versions as follows:
 - `StringBuilder insert(int insertPosition, String str)`
 - `StringBuilder insert(int insertPosition, char ch)`
 - `StringBuilder insert(int insertPosition, float f)`
- For example,

```
StringBuilder str = new StringBuilder("JAVA 7 ");
str.insert(5, "SE");
System.out.println(str);
// output: JAVA SE 7
```

© Aptech Ltd. Arrays and Strings/Session 8 55

Methods of `StringBuilder` Class 3-5

delete()

- The `delete()` method deletes the specified number of characters from the invoking `StringBuilder` object.
- For example,

```
StringBuilder str = new StringBuilder("JAVA SE 7");
str.delete(4, 7);
System.out.println(str);
// output: JAVA
```

reverse()

- The `reverse()` method is used to reverse the characters within a `StringBuilder` object.
- For example,

```
StringBuilder str = new StringBuilder("JAVA SE 7");
str.reverse();
System.out.println(str);
// output: 7 ES AVAJ
```

© Aptech Ltd. Arrays and Strings/Session 8 56

Methods of StringBuilder Class 4-5

- Following code snippet demonstrates the use of methods of the `StringBuilder` class:

```
package session8;
public class StringBuilders {
    // Instantiate a StringBuilder object
    StringBuilder str = new StringBuilder("JAVA ");

    /**
     * Displays strings using various StringBuilder methods
     * @return void
     */
    public void displayStrings(){
        // Use the various methods of the StringBuilder class
        System.out.println("Appended String is "+ str.append("7"));
        System.out.println("Inserted String is "+ str.insert(5, "SE "));
        System.out.println("Deleted String is "+ str.delete(4,7));
        System.out.println("Reverse String is "+ str.reverse());
    }
}
```

© Aptech Ltd.

Arrays and Strings/Session 8

57

Methods of StringBuilder Class 5-5

```
/*
 * @param args the command line arguments
 */
public static void main(String[] args) {
    //Instantiate the StringBuilders class
    StringBuilders objStrBuild = new StringBuilders(); // line 1

    //Invoke the displayStrings() method
    objStrBuild.displayStrings();
}
}
```

- Following figure shows the output of the `StringBuilders.java` class:

Output - Session8 (run)

run:
Appended String is JAVA 7
Inserted String is JAVA SE 7
Deleted String is JAVA 7
Reverse String is 7 AVAJ

© Aptech Ltd.

Arrays and Strings/Session 8

58

Using slides 54 to 58, explain methods of `StringBuilder` class.

Explain the `StringBuilder` class provides several methods for appending, inserting, deleting, and reversing strings as follows mentioned on slides 54 to 56.

Explain the code snippet demonstrates the use of methods of the `StringBuilder` class mentioned in slides 57 and 58.

Tips:

You can use any `String` method on a `StringBuilder` object by first converting the string builder to a string with the `toString()` method of `StringBuilder` class. Then convert the string back into a string builder using `StringBuilder(String str)` constructor.

Slides 59 to 61

Let us understand string arrays.

String Arrays 1-3

- Sometimes there is a need to store a collection of strings.
- String arrays can be created in Java in the same manner as arrays of primitive data types.
- For example, `String[] empNames = new String[10];`
- This statement will allocate memory to store references of 10 strings.
- However, no memory is allocated to store the characters that make up the individual strings.
- Loops can be used to initialize as well as display the values of a String array.

Following code snippet demonstrates the creation of a String array:

```
package session8;
public class StringArray {
    // Instantiate a String array
    String[] empID = new String[5];
```

© Aptech Ltd.

Arrays and Strings/Session 8

59

String Arrays 2-3

```
/*
 * Creates a String array
 * @return void
 */
public void createArray() {
    System.out.println("Creating Array. Please wait...");
    // Use a for loop to initialize the array
    for(int count = 1; count < empID.length; count++){
        empID[count] = "E00"+count; // storing values in the array
    }
}
/*
 * Displays the elements of a String array
 * @return void
 */
public void printArray() {
    System.out.println("The Array is:");
    // Use a for loop to print the array
    for(int count = 1; count < empID.length; count++){
        System.out.println("Employee ID is: "+ empID[count]);
    }
}
```

© Aptech Ltd.

Arrays and Strings/Session 8

60

String Arrays 3-3

```
/*
 * @param args the command line arguments
 */
public static void main(String[] args) {
    //Instantiate class Strings
    StringArray objStrArray = new StringArray(); // line 1
    //Invoke createArray() method
    objStrArray.createArray();
    //Invoke printArray() method
    objStrArray.printArray();
}
```

Following figure shows the output of the `StringArray.java` class:

Output - Session8 (run)

```
run:
Creating Array. Please wait...
The Array is:
Employee ID is: E001
Employee ID is: E002
Employee ID is: E003
Employee ID is: E004
```

© Aptech Ltd.

Arrays and Strings/Session 8

61

Using slides 59 to 61, explain string arrays.

Sometimes there is a need to store a collection of strings. String arrays can be created in Java in the same manner as arrays of primitive data types. For example, `String [] empNames = new String[10];` the statement will allocate memory to store references of 10 strings.

However, no memory is allocated to store the characters that make up the individual strings. Loops can be used to initialize as well as display the values of a string array.

Explain the code snippet demonstrates the creation of a string array mentioned on slides 59 to 61.

String array is just like creating and using any other Java object array. It is nothing more than a sequence of strings. The total number of strings must be fixed but each string can be of any length. String arrays are relatively straightforward, and Java provides a moderate amount of built-in functionality to make them easy to work with.

The two basic aspects of string array are, initialization and iteration. Inline initialization is well suited where number of strings is relatively small and you happen to know at compile-time which strings should be in the array.

The iteration can be basic and advanced. The basic iteration is well suited where you need to write to an element of an array and to analyze its existing contents. Advanced iteration is well suited where you do not need to write to an array.

In-Class Question:

After you finish explaining the string arrays, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



What is the role of loops in string array?

Answer:

They can be used to initialize and display the values of a string array.

Slides 62 to 68

Let us understand command line arguments.

Command Line Arguments 1-7



- A user can pass any number of arguments to a Java application at runtime from the OS command line.
- The `main()` method declares a parameter named `args[]` which is a `String` array that accepts arguments from the command line.
- These arguments are placed on the command line and follow the class name when it is executed.
 - For example,
`java EmployeeDetail Roger Smith Manager`
 - Here, `EmployeeDetail` is the name of a class.
 - `Roger, Smith, and Manager` are command line arguments which are stored in the array in the order that they are specified.
- When the application is launched, the runtime system passes the command line arguments to the application's `main()` method using a `String` array, `args[]`.
- The array of strings can be given any other name.
- The `args[]` array accepts the arguments and stores them at appropriate locations in the array.

© Aptech Ltd. Arrays and Strings/Session 8 62

Command Line Arguments 2-7



- The length of the array is determined from the number of arguments passed at runtime.
- The arguments are separated by a space.
- The basic purpose of command line arguments is to specify the configuration information for the application.
- The `main()` method is the entry point of a Java program, where objects are created and methods are invoked.
- The `static main()` method accepts a `String` array as an argument as depicted in the following code snippet:

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

- The parameter of the `main()` method is a `String` array that represents the command line arguments.
- The size of the array is set to the number of arguments specified at runtime.
- All command line arguments are passed as strings.

© Aptech Ltd. Arrays and Strings/Session 8 63

Command Line Arguments 3-7



- Following code snippet demonstrates an example of command line arguments:

```
package session8;
public class CommandLine {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {

        // Check the number of command line arguments
        if(args.length==3) {

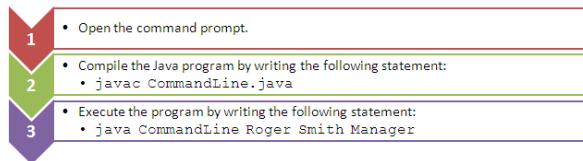
            // Display the values of individual arguments
            System.out.println("First Name is "+args[0]);
            System.out.println("Last Name is "+args[1]);
            System.out.println("Designation is "+args[2]);
        }
        else {
            System.out.println("Specify the First Name, Last Name, and Designation");
        }
    }
}
```

© Aptech Ltd. Arrays and Strings/Session 8 64

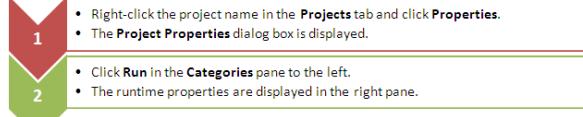
Command Line Arguments 4-7



- To run the program with command line arguments at command prompt, do the following:



- To run the program with command line arguments using NetBeans IDE, perform the following steps:



© Aptech Ltd.

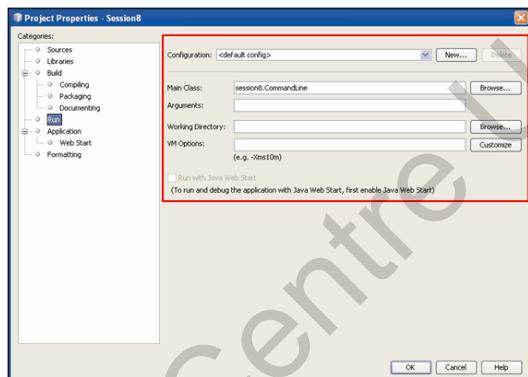
Arrays and Strings/Session 8

65

Command Line Arguments 5-7



- The runtime properties are shown in the following figure:



© Aptech Ltd.

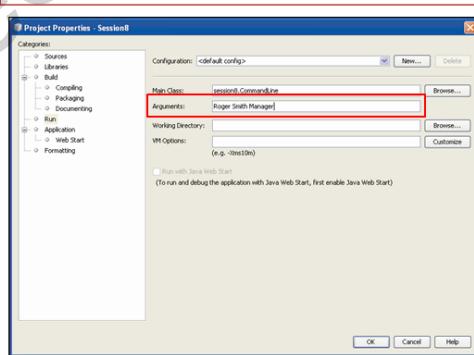
Arrays and Strings/Session 8

66

Command Line Arguments 6-7



- Type the arguments **Roger**, **Smith**, and **Manager** in the **Arguments** box as shown in the following figure:



© Aptech Ltd.

Arrays and Strings/Session 8

67

Command Line Arguments 7-7

4 • Click **OK** to close the **Project Properties** dialog box.

5 • Click **Run** on the toolbar or press **F6**.
• The command line arguments are supplied to the `main()` method and printed as shown in the following figure:

Output - Session8 (run)

```
run:
First Name is Roger
Last Name is Smith
Designation is Manager
```

Using slides 62 to 68, explain command line arguments.

A user can pass any number of arguments to a Java application at runtime from the OS command line. The `main()` method declares a parameter named `args []` which is a String array that accepts arguments from the command line. These arguments are placed on the command line and follow the class name when it is executed.

When the application is launched, the runtime system passes the command line arguments to the application's `main()` method using a String array, `args[]`. The array of strings can be given any other name. The `args[]` array accepts the arguments and stores them at appropriate locations in the array.

The length of the array is determined from the number of arguments passed at runtime. The arguments are separated by a space. The basic purpose of command line arguments is to specify the configuration information for the application. The `main()` method is the entry point of a Java program, where objects are created and methods are invoked.

The static `main()` method accepts a String array as an argument as depicted in the following code snippet:

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

The parameter of the `main()` method is a String array that represents the command line arguments. The size of the array is set to the number of arguments specified at runtime. All command line arguments are passed as strings.

Explain the code snippet demonstrates an example of command line arguments mentioned in slide 64.

Explain the steps to run the program with command line arguments at command prompt mentioned in slide 65.

Explain the steps to run the program with command line arguments using NetBeans IDE mentioned in slides 66 to 68.

A Java application can accept any number of arguments from the command line. This allows the user to specify configuration information when the application is launched.

The user enters command-line arguments when invoking the application and specifies them after the name of the class to be run. Many Java applications started from the command line which take arguments to control their behavior.

Slides 69 to 79

Let us understand wrapper classes.

Wrapper Classes 1-11

Java provides a set of classes known as wrapper classes for each of its primitive data type that 'wraps' the primitive type into an object of that class.

In other words, the wrapper classes allow accessing primitive data types as objects.

The wrapper classes for the primitive data types are: Byte, Character, Integer, Long, Short, Float, Double, and Boolean.

The wrapper classes are part of the `java.lang` package.

- ◆ The primitive types and the corresponding wrapper types are listed in the following table:

Primitive type	Wrapper class
byte	Byte
char	Character
float	Float
double	Double
int	Integer
long	Long
short	Short
boolean	Boolean

© Aptech Ltd. Arrays and Strings/Session 8 69

Wrapper Classes 2-11

What is the need for wrapper classes?

The use of primitive types as objects can simplify tasks at times.

For example, most of the collections store objects and not primitive data types.

Many of the activities reserved for objects will not be available to primitive data types.

Also, many utility methods are provided by the wrapper classes that help to manipulate data.

Wrapper classes convert primitive data types to objects, so that they can be stored in any type of collection and also passed as parameters to methods.

Wrapper classes can convert numeric strings to numeric values.

© Aptech Ltd. Arrays and Strings/Session 8 70

Wrapper Classes 3-11

valueOf()

- The `valueOf()` method is available with all the wrapper classes to convert a type into another type.
- The `valueOf()` method of the `Character` class accepts only `char` as an argument.
- The `valueOf()` method of any other wrapper class accepts either the corresponding primitive type or `String` as an argument.

typeValue()

- The `typeValue()` method can also be used to return the value of an object as its primitive type.

- ◆ Some of the wrapper classes and their methods are listed in the following table:

Wrapper Class	Methods	Example
Byte	<code>byteValue()</code> – returns a byte value of the invoking object. <code>parseByte()</code> – returns the byte value from a string storing a byte value.	<code>byte byteVal = Byte.byteValue();</code> <code>byte byteVal = Byte.parseByte("45");</code>

© Aptech Ltd. Arrays and Strings/Session 8 71

Wrapper Classes 4-11



Wrapper Class	Methods	Example
Character	isDigit() – checks if a character is a digit. isLowerCase() – checks if a character is a lower case alphabet. isLetter() – checks if a character is an alphabet.	<pre>if(Character.isDigit('4')) System.out.println("Digit"); if(Character.isLetter('L') System.out.println("Letter");</pre>
Integer	intValue() – returns the Integer value as a primitive type int. parseInt() – returns the int value from a string storing an integer value.	<pre>int intValue = Integer.intValue(); int intVal=Integer.parseInt("45");</pre>

© Aptech Ltd.

Arrays and Strings/Session 8

72

Wrapper Classes 5-11

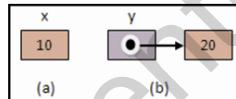


- The difference between creation of a primitive type and a wrapper type is as follows:

Primitive type • `int x = 10;`

Wrappertype • `Integer y = new Integer(20);`

- The first statement declares and initializes the `x` variable **x** to 10.
- The second statement instantiates an `Integer` object **y** and initializes it with the value 20.
- In this case, the reference of the object is assigned to the object variable **y**.
- The memory assignment for the two statements is shown in the following figure:



- It is clear from the figure that **x** is a variable that holds a value whereas **y** is an object variable that holds a reference to an object.

© Aptech Ltd.

Arrays and Strings/Session 8

73

Wrapper Classes 6-11



The six methods of type `parseXxx()` available for each numeric wrapper type are in close relation to the `valueOf()` methods of all the numeric wrapper classes including Boolean.

The two type of methods, that is, `parseXxx()` and `valueOf()`, take a `String` as an argument.

If the `String` argument is not properly formed, both the methods throw a `NumberFormatException`.

These methods can convert `String` objects of different bases if the underlying primitive type is any of the four integer types.

The `parseXxx()` method returns a named primitive whereas the `valueOf()` method returns a new wrapped object of the type that invoked the method.

© Aptech Ltd.

Arrays and Strings/Session 8

74

Wrapper Classes 7-11



- Following code snippet demonstrates the use of `Integer` wrapper class to convert the numbers passed by user as strings at command line into integer types to perform the calculation based on the selected operation:

```
package session8;
public class Wrappers {
    /**
     * Performs calculation based on user input
     *
     * @return void
     */
    public void calcResult(int num1, int num2, String choice){
        // Switch case to evaluate the choice
        switch(choice) {
            case "+": System.out.println("Result after addition is: "+
                (num1+num2));
            break;
            case "-": System.out.println("Result after subtraction is: "+
                (num1-num2));
            break;
            case "*": System.out.println("Result after multiplication is: "+
                (num1*num2));
            break;
        }
    }
}
```

© Aptech Ltd.

Arrays and Strings/Session 8

75

Wrapper Classes 8-11



```
case "/": System.out.println("Result after division is: " +
    (num1/num2));
break;
}

/**
 * @param args the command line arguments
 */
public static void main(String[] args) {

    // Check the number of command line arguments
    if(args.length==3){

        // Use the Integer wrapper to convert String argument to int type
        int num1 = Integer.parseInt(args[0]);
        int num2 = Integer.parseInt(args[1]);

        // Instantiate the Wrappers class
        Wrappers objWrap = new Wrappers();
    }
}
}
```

© Aptech Ltd.

Arrays and Strings/Session 8

76

Wrapper Classes 9-11



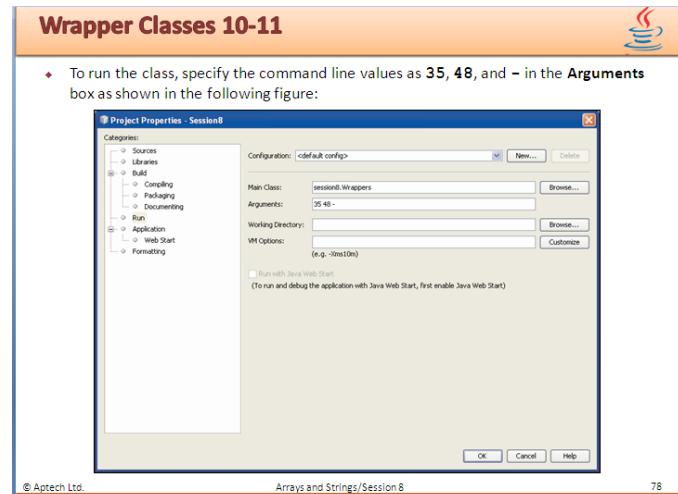
```
// Invoke the calcResult() method
objWrap.calcResult(num1, num2, args[2]);
}
else{
    System.out.println("Usage: num1 num2 operator");
}
}
}
```

- The class `Wrappers` consists of the `calcResult()` method that accepts two numbers and an operator as the parameter.
- The `main()` method is used to convert the `String` arguments to `int` type by using the `Integer` wrapper class.
- Next, the object, `objWrap` of `Wrappers` class is created to invoke the `calcResult()` method with three arguments namely, `num1`, `num2`, and `args[2]` which is the operator specified by the user as the third argument.

© Aptech Ltd.

Arrays and Strings/Session 8

77



Using slides 69 to 79, explain wrapper classes.

Java provides a set of classes known as wrapper classes for each of its primitive data type that ‘wraps’ the primitive type into an object of that class. In other words, the wrapper classes allow accessing primitive data types as objects.

The wrapper classes for the primitive data types are: Byte, Character, Integer, Long, Short, Float, Double, and Boolean. The wrapper classes are part of the `java.lang` package.

The primitive types and the corresponding wrapper types are listed in the table in slide 69.

What is the need for wrapper classes?

The use of primitive types as objects can simplify tasks at times. For example, most of the collections store objects and not primitive data types. Many of the activities reserved for objects will not be available to primitive data types.

Also, many utility methods are provided by the wrapper classes that help to manipulate data. Wrapper classes convert primitive data types to objects, so that they can be stored in any type of collection and also passed as parameters to methods. Wrapper classes can convert numeric strings to numeric values.

valueOf()

- The `valueOf()` method is available with all the wrapper classes to convert a type into another type.
- The `valueOf()` method of the Character class accepts only `char` as an argument.
- The `valueOf()` method of any other wrapper class accepts either the corresponding primitive type or `String` as an argument.

typeValue()

The `typeValue()` method can also be used to return the value of an object as its primitive type.

Explain some of the wrapper classes and their methods are listed in the table in slides 71 and 72.

Explain the difference between creation of a primitive type and a wrapper type mentioned in slide 73.

The six methods of type `parseXxx()` available for each numeric wrapper type are in close relation to the `valueOf()` methods of all the numeric wrapper classes including Boolean. The two type of methods, that is, `parseXxx()` and `valueOf()`, take a `String` as an argument. If the `String` argument is not properly formed, both the methods throw a `NumberFormatException`.

These methods can convert `String` objects of different bases if the underlying primitive type is any of the four integer types. The `parseXxx()` method returns a named primitive whereas the `valueOf()` method returns a new wrapped object of the type that invoked the method.

Explain the code snippet demonstrates the use of Integer wrapper class to convert the numbers passed by user as strings at command line into integer types to perform the calculation based on the selected operation mentioned in slides 75 to 79.

A wrapper class encloses around a data type and gives it an object appearance. Whenever, the data type is required as an object, this object can be used.

Wrapper classes include methods to unwrap the object and give back the data type.

The two main uses with Wrapper classes are:

- To convert simple data types into objects, that is, to give object form to a data type.
- To convert strings into data types known as parsing operations, here methods of type `parse()` are used.

In-Class Question:

After you finish explaining the wrapper classes, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



Use of wrapper classes utility methods?

Answer:

They help to manipulate data.

Slides 80 to 82

Let us understand autoboxing and unboxing.

Autoboxing and Unboxing 1-3

Autoboxing

- The automatic conversion of primitive data types such as `int`, `float`, and so on to their corresponding object types such as `Integer`, `Float`, and so on during assignments and invocation of methods and constructors is known as **autoboxing**.
 - For example,
- ```
ArrayList<Integer> intList = new ArrayList<Integer>();
intList.add(10); // autoboxing
Integer y = 20; // autoboxing
```

#### Unboxing

- The automatic conversion of object types to primitive data types is known as **unboxing**.
  - For example,
- ```
int z = y; // unboxing
```

Autoboxing and unboxing helps a developer to write a cleaner code.

Using autoboxing and unboxing, one can make use of the methods of wrapper classes as and when required.

© Aptech Ltd.

Arrays and Strings/Session 8

80

Autoboxing and Unboxing 2-3

- Following code snippet demonstrates an example of **autoboxing** and **unboxing**:

```
package session8;
public class AutoUnbox {

    /**
     * @param args the command line arguments
     */
    public static void main(String args[]) {
        Character chBox = 'A'; // Autoboxing a character
        char chUnbox = chBox; // Unboxing a character

        // Print the values
        System.out.println("Character after autoboxing is:" + chBox);
        System.out.println("Character after unboxing is:" + chUnbox);
    }
}
```

- The class `AutoUnbox` consists of two variable declarations `chBox` and `chUnbox`.
- `chBox` is an object type and `chUnbox` is a primitive type of character variable.

© Aptech Ltd.

Arrays and Strings/Session 8

81

Autoboxing and Unboxing 3-3

- Following figure shows the output of the code:

Output - Session8 (run)

```
run:
Character after autoboxing is:A
Character after unboxing is:A
```

- The figure shows that both primitive as well as object type variable give the same output.
- However, the variable of type object, that is `chBox`, can take advantage of the methods available with the wrapper class `Character` which are not available with the primitive type `char`.

© Aptech Ltd.

Arrays and Strings/Session 8

82

Using slides 80 to 82, explain autoboxing and unboxing.

Autoboxing

The automatic conversion of primitive data types such as int, float, and so on to their corresponding object types such as Integer, Float, and so on during assignments and invocation of methods and constructors is known as autoboxing.

For example,

```
ArrayList<Integer> intList = new ArrayList<Integer>();  
intList.add(10); // autoboxing  
Integer y = 20; // autoboxing
```

Converting a primitive value into an object of the corresponding wrapper class is called autoboxing. The Java class applies autoboxing when a primitive value is passed as a parameter to a method that expects an object of the corresponding wrapper class or assigned to a variable for the corresponding wrapper class.

Unboxing

The automatic conversion of object types to primitive data types is known as unboxing.

For example,

```
int z = y; // unboxing
```

Autoboxing and unboxing helps a developer to write a cleaner code. Using autoboxing and unboxing, one can make use of the methods of wrapper classes as and when required.

Explain the code snippet demonstrates an example of autoboxing and unboxing mentioned in slides 81 and 82.

Converting an object of a wrapper type to its corresponding primitive value is called unboxing. The Java compiler applies unboxing when an object of a wrapper class is passed as a parameter to a method that expects a value of the corresponding primitive type or assigned to a variable of the corresponding primitive type.

Slide 83

Let us summarize the session.

Summary



- ◆ An array is a special data store that can hold a fixed number of values of a single type in contiguous memory locations.
- ◆ A single-dimensional array has only one dimension and is visually represented as having a single column with several rows of data.
- ◆ A multi-dimensional array in Java is an array whose elements are also arrays.
- ◆ A collection is an object that groups multiple elements into a single unit.
- ◆ Strings are constant and immutable, that is, their values cannot be changed once they are created.
- ◆ `StringBuilder` objects are similar to String objects, except that they are mutable.
- ◆ Java provides a set of classes known as Wrapper classes for each of its primitive data type that 'wrap' the primitive type into an object of that class.
- ◆ The automatic conversion of primitive types to object types is known as `autoboxing` and conversion of object types to primitive types is known as `unboxing`.

© Aptech Ltd. Arrays and Strings/Session 8 88

In slide 83, you will summarize the session. End the session with a brief summary of what has been taught in the session.

8.3 Post Class Activities for Faculty

You should familiarize yourself with the topics of the next session which is based on modifiers and packages.

Tips:

You can also check the Articles/Blogs/Expert Videos uploaded on the Online Varsity site to gain additional information related to the topics covered in the next session. You can also connect to online tutors on the Online Varsity site to ask queries related to the sessions.

Session 9 – Modifiers and Packages

9.1 Pre-Class Activities

Before you commence the session, you should familiarize yourself with the topics of this session in-depth.

Here, you can discuss the key points with the students that were covered in the previous session. Prepare a question or two which will help you to relate the current session objectives.

9.1.1 Objectives

By the end of this session, the learners will be able to:

- Describe field and method modifiers
- Explain the different types of modifiers
- Explain the rules and best practices for using field modifiers
- Describe class variables
- Explain the creation of static variables and methods
- Describe package and its advantages
- Explain the creation of user-defined package
- Explain the creation of .jar files for deployment

9.1.2 Teaching Skills

To teach this session, you should be well-versed with different fields and method modifiers in Java with the rules for using field modifiers. Also, familiarize yourself with creation of packages and .jar files.

You should teach the concepts in the theory class using the images provided. For teaching in the class, you are expected to use slides and LCD projectors.

Tips:

It is recommended that you test the understanding of the students by asking questions in between the class.

In-Class Activities:

Follow the order given here during In-Class activities.

Overview of the Session:

Give the students the overview of the current session in the form of session objectives. Show the students slide 2 of the presentation.

Objectives	Cup of coffee icon
<ul style="list-style-type: none"> ◆ Describe field and method modifiers ◆ Explain the different types of modifiers ◆ Explain the rules and best practices for using field modifiers ◆ Describe class variables ◆ Explain the creation of static variables and methods ◆ Describe package and its advantages ◆ Explain the creation of user-defined package ◆ Explain the creation of .jar files for deployment 	
<small>© Aptech Ltd. Modifiers and Packages/Session 9 2</small>	

Tell the students that this session introduces different fields and method modifiers in Java with the rules for using field modifiers. Also, the session will explain in detail about the packages, creation of packages, and creation of JAR files.

9.2 In-Class Explanation**Slide 3**

Let us understand the package and package specifier in Java.

Introduction	Cup of coffee icon
<ul style="list-style-type: none"> ◆ Java is a tightly encapsulated language. ◆ Java provides a set of access specifiers such as <code>public</code>, <code>private</code>, <code>protected</code>, and <code>default</code> that help to restrict access to class and class members. ◆ Java provides additional field and method modifiers to further restrict access to the members of a class to prevent modification by unauthorized code. ◆ Java provides the concept of class variables and methods to create a common data member that can be shared by all objects of a class as well as other classes. ◆ Java provides packages that can be used to group related classes that share common attributes and behavior. ◆ The entire set of packages can be combined into a single file called the <code>.jar</code> file for deployment on the target system. 	
<small>© Aptech Ltd. Modifiers and Packages/Session 9 3</small>	

Using slide 3, explain package and package specifier in Java.

Java is a tightly encapsulated language. Java provides a set of access specifiers such as `public`, `private`, `protected`, and `default` that help to restrict access to class and class members.

Explain them that Java provides additional field and method modifiers to further restrict access to the members of a class to prevent modification by unauthorized code.

Java provides the concept of class variables and methods to create a common data member that can be shared by all objects of a class as well as other classes. Java provides packages that can be used to group related classes that share common attributes and behavior. The entire set of packages can be combined into a single file called the .jar file for deployment on the target system.

Slide 4

Let us understand field and method modifiers.

Field and Method Modifiers

Field and method modifiers are keywords used to identify fields and methods that provide controlled access to users.

Some of these can be used in conjunction with access specifiers such as public and protected.

- The different field modifiers that can be used are as follows:
 - **volatile**
 - **native**
 - **transient**
 - **final**

© Aptech Ltd. Modifiers and Packages/Session 9 4

Using slide 4, explain field and method modifiers.

Field and method modifiers are keywords used to identify fields and methods that provide controlled access to users. Some of these can be used in conjunction with access specifiers such as public and protected.

Explain them the different field modifiers that can be used in Java. Tell them that **volatile** modifier can be used to inform the compiler that it should not attempt to perform optimization on the field which could cause unpredicted results when the fields are accessed by multiple threads.

Native methods are specified in the class as method prototype with the keyword native, no method body is defined in the Java class.

The **final** variable is constant, its value cannot be changed after initialization. It can be applied to local, instance, and static variables including parameters that are declared as final.

Transient fields are declared with keyword **Transient** in their class declaration if their value should not be saved when object of class written to persistent storage.

Slides 5 and 6

Let us understand ‘volatile’ modifier.

‘volatile’ Modifier 1-2



The `volatile` modifier allows the content of a variable to be synchronized across all running threads.

A thread is an independent path of execution of code within a program. Many threads can run concurrently within a program.

- The `volatile` modifier is applied only to fields.
- Constructors, methods, classes, and interfaces cannot use this modifier.
- The `volatile` modifier is not frequently used.
- While working with a multithreaded program, the `volatile` keyword is used.

© Aptech Ltd. Modifiers and Packages/Session 9 5

‘volatile’ Modifier 2-2



When multiple threads of a program are using the same variable, in general, each thread has its own copy of that variable in the local cache.

In such a case, if the value of the variable is updated, it updates the copy in the local cache and not the main variable present in the memory.

The other thread using the same variable does not get the updated value.

- To avoid this problem, a variable is declared as `volatile` to indicate that it will not be stored in the local cache.
- Whenever a thread updates the values of the variable, it updates the variable present in the main memory.
- This helps other threads to access the updated value.
- For example,

```
private volatile int testValue; // volatile variable
```

© Aptech Ltd. Modifiers and Packages/Session 9 6

Using slides 5 and 6, explain volatile modifier.

The `volatile` modifier allows the content of a variable to be synchronized across all running threads. A thread is an independent path of execution of code within a program.

The features of volatile modifier are as follows:

- The `volatile` modifier is applied only to fields.
- Constructors, methods, classes, and interfaces cannot use this modifier.

When multiple threads of a program are using the same variable, in general, each thread has its own copy of that variable in the local cache. In such a case, if the value of the variable is updated, it updates the copy in the local cache and not the main variable present in the memory. The other thread using the same variable does not get the updated value.

To avoid this problem, a variable is declared as `volatile` to indicate that it will not be stored in the local cache. Whenever a thread updates the values of the variable, it updates the variable present in the main memory. This helps other threads to access the updated value.

For example, `private volatile int testValue; // volatile variable`

Declaring a `volatile` Java variable means the value of the variable will never cache thread-locally that means all read and write will go to main memory and access to the variable acts as synchronized on itself. Every thread accessing a `volatile` field will read its current value before continuing, instead of using a cached value. A primitive variable may be declared as `volatile`.

Slides 7 to 9

Let us understand ‘native’ modifier.

‘native’ Modifier 1-3



- The `native` modifier is used only with methods.
- It indicates that the implementation of the method is in a language other than Java such as C or C++.
- Constructors, fields, classes, and interfaces cannot use this modifier.
- The methods declared using the `native` modifier are called native methods.
- The Java source file typically contains only the declaration of the native method and not its implementation.
- The implementation of the method exists in a library outside the JVM.

- ◆ Before invoking a native method, the library that contains the method implementation must be loaded by making the following system call:
`System.loadLibrary("libraryName");`
- ◆ To declare a native method, the method is preceded with the `native` modifier.
- ◆ The implementation is not provided for the method. For example,
`public native void nativeMethod();`

© Aptech Ltd. Modifiers and Packages/Session 9 7

‘native’ Modifier 2-3



- After declaring a native method, a complex series of steps are used to link it with the Java code.
- Following code snippet demonstrates an example of loading a library named `NativeMethodDefinition` containing a native method named `nativeMethod()`:

```
class NativeModifier {
    native void nativeMethod(); // declaration of a native method
    /**
     * static code block to load the library
     */
    static {
        System.loadLibrary("NativeMethodDefinition");
    }
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        NativeModifier objNative = new NativeModifier(); // line1
        objNative.nativeMethod(); // line2
    }
}
```

© Aptech Ltd. Modifiers and Packages/Session 9 8

'native' Modifier 3-3



- ◆ Notice that a `static` code block is used to load the library.
- ◆ The `static` keyword indicates that the library is loaded as soon as the class is loaded.
- ◆ This ensures that the library is available when the call to the native method is made. The native method can be used in the same way as a non-native method.
- ◆ Native methods allow access to existing library routines created outside the JVM.
- ◆ However, the use of native methods introduces two major problems.

Impending security risk

- The native method executes actual machine code, and therefore, it can gain access to any part of the host system.
- That is, the native code is not restricted to the JVM execution environment.
- This may lead to a virus infection on the target system.

Loss of portability

- The native code is bundled in a DLL, so that it can be loaded on the machine on which the Java program is executing.
- Each native method is dependent on the CPU and the OS.
- This makes the DLL inherently non-portable.
- This means, that a Java application using native methods will run only on a machine in which a compatible DLL has been installed.

© Aptech Ltd. Modifiers and Packages/Session 9 9

Using slides 7 to 9, explain 'native' modifier.

The native modifier is used only with methods. It indicates that the implementation of the method is in a language other than Java such as C or C++. Constructors, fields, classes, and interfaces cannot use this modifier.

The methods declared using the `native` modifier are called native methods. The Java source file typically contains only the declaration of the native method and not its implementation. The implementation of the method exists in a library outside the JVM.

Before invoking a native method, the library that contains the method implementation must be loaded by making the following system call:

```
System.loadLibrary("libraryName");
```

To declare a native method, the method is preceded with the native modifier. The implementation is not provided for the method. For example, `public native void nativeMethod();`

After declaring a native method, a complex series of steps are used to link it with the Java code.

Explain the code snippet demonstrates an example of loading a library named `NativeMethodDefinition` containing a native method named `nativeMethod()` mentioned in slide 8.

Notice that a static code block is used to load the library.

The `static` keyword indicates that the library is loaded as soon as the class is loaded. This ensures that the library is available when the call to the native method is made. The native method can be used in the same way as a non-native method. Native methods allow access to existing library routines created outside the JVM.

However, the use of native methods introduces two major problems, discuss with them referring slide 9.

Slides 10 and 11

Let us understand ‘transient’ modifier.

‘transient’ Modifier 1-2



When a Java application is executed, the objects are loaded in the Random Access Memory (RAM).

Objects can also be stored in a persistent storage outside the JVM so that it can be used later.

This determines the scope and life span of an object.

The process of storing an object in a persistent storage is called serialization.

For any object to be serialized, the class must implement the `Serializable` interface.

If `transient` modifier is used with a variable, it will not be stored and will not become part of the object’s persistent state.

The `transient` modifier is useful to prevent security sensitive data from being copied to a source in which no security mechanism has been implemented.

The `transient` modifier reduces the amount of data being serialized, improves performance, and reduces costs.

© Aptech Ltd. Modifiers and Packages/Session 9 10

‘transient’ Modifier 2-2



- ◆ The `transient` modifier can only be used with instance variables.
- ◆ It informs the JVM not to store the variable when the object, in which it is declared, is serialized.
- ◆ Thus, when the object is stored in persistent storage, the instance variable declared as `transient` is not persisted.
- ◆ Following code snippet depicts the creation of a `transient` variable:

```
class Circle {
    transient float PI; // transient variable that will not persist
    float area; // instance variable that will persist
}
```

© Aptech Ltd. Modifiers and Packages/Session 9 11

Using slides 10 and 11, explain transient modifier.

When a Java application is executed, the objects are loaded in the Random Access Memory (RAM). Objects can also be stored in a persistent storage outside the JVM so that it can be used later. This determines the scope and life span of an object. The process of storing an object in a persistent storage is called serialization.

For any object to be serialized, the class must implement the `Serializable` interface. If `transient` modifier is used with a variable, it will not be stored and will not become part of the object’s persistent state.

The `transient` modifier is useful to prevent security sensitive data from being copied to a source in which no security mechanism has been implemented. The `transient` modifier reduces the amount of data being serialized, improves performance, and reduces costs.

The transient modifier can only be used with instance variables. It informs the JVM not to store the variable when the object, in which it is declared, is serialized. Thus, when the object is stored in persistent storage, the instance variable declared as transient is not persisted.

Explain the code snippet that depicts the creation of a transient variable mentioned in slide 11.

An instance variable is marked transient to indicate the JVM to skip the particular variable when serializing the object containing it. This modifier is included in the statement that creates the variable, preceding the class, and data type of the variable.

In-Class Question:

After you finish explaining the transient modifier, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



What is the use of transient modifier?

Answer:

It is useful to prevent security sensitive data from being copied to a source in which no security mechanism has been implemented.

Slides 12 to 16

Let us understand 'final' modifier.

'final' Modifier 1-5

The final modifier is used when modification of a class or data member is to be restricted.

The final modifier can be used with a variable, method, and class.

Final Variable

A variable declared as final is a constant whose value cannot be modified.

A final variable is assigned a value at the time of declaration.

A compile time error is raised if a final variable is reassigned a value in a program after its declaration.

- Following code snippet shows the creation of a final variable:
`final float PI = 3.14;`
- The variable PI is declared final so that its value cannot be changed later.

© Aptech Ltd. Modifiers and Packages/Session 9 12

'final' Modifier 2-5



Final Method

A method declared `final` cannot be overridden or hidden in a Java subclass.

The reason for using a `final` method is to prevent subclasses from changing the meaning of the method.

A `final` method is commonly used to generate a random constant in a mathematical application.

- Following code snippet depicts the creation of a `final` method:

```
final float getCommission(float sales){  
    System.out.println("A final method. . .");  
}
```

- The method `getCommission()` can be used to calculate commission based on monthly sales.
- The implementation of the method cannot be modified by other classes as it is declared as `final`.
- A `final` method cannot be declared abstract as it cannot be overridden.

'final' Modifier 3-5



Final Class

A class declared `final` cannot be inherited or subclassed.

Such a class becomes a standard and must be used as it is.

The variables and methods of a class declared `final` are also implicitly `final`.

- The reason for declaring a class as `final` is to limit extensibility and to prevent the modification of the class definition.

- Following code snippet shows the creation of a `final` class:

```
public final class Stock {  
    ...  
}
```

- The class `Stock` is declared `final`.
- All data members within this class are implicitly `final` and cannot be modified by other classes.

'final' Modifier 4-5



- Following code snippet demonstrates an example of creation of a `final` class:

```
package session9;  
public class Final {  
  
    // Declare and initialize a final variable  
    final float PI = 3.14F; // variable to store value of PI  
  
    /**  
     * Displays the value of PI  
     *  
     * @param pi a float variable storing the value of PI  
     * @return void  
     */  
    public void display(float pi) {  
        PI = pi; // generates compilation error  
        System.out.println("The value of PI is:"+PI);  
    }  
  
    /**  
     * @param args the command line arguments  
     */  
    public static void main(String[] args) {
```

'final' Modifier 5-5

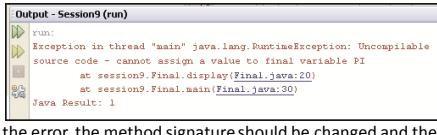


```
// Instantiate the Final class
final Final objFinal = new Final();

// Invoke the display() method
objFinal.display(22.7F);
}

◆ The class Final consists of a final float variable PI set to 3.14.
◆ The method display() is used to set a new value passed by the user to PI.
◆ This leads to compilation error 'cannot assign a value to final
variable PI'.
◆ If the user chooses to run the program anyway, the following runtime error is issued as
shown in the following figure:
```

Output - Session9 (run)



- To remove the error, the method signature should be changed and the statement, **PI = pi;** should be removed.

© Aptech Ltd. Modifiers and Packages/Session 9 16

Using slides 12 to 16, explain final modifier.

The **final** modifier is used when modification of a class or data member is to be restricted. The **final** modifier can be used with a variable, method, and class.

Final Variable

A variable declared as **final** is a constant whose value cannot be modified. A **final** variable is assigned a value at the time of declaration. A compile time error is raised if a **final** variable is reassigned a value in a program after its declaration.

Following code snippet shows the creation of a **final** variable: `final float PI = 3.14;` Explain them that the variable **PI** is declared **final** so that its value cannot be changed later.

A **final** variable can only be initialized once, either through an initialize or an assignment statement. It does not need to be initialized at the point of declaration. This is called blank **final** variable.

Final Method

A method declared **final** cannot be overridden or hidden in a Java subclass. The reason for using a **final** method is to prevent subclasses from changing the meaning of the method. A **final** method is commonly used to generate a random constant in a mathematical application.

Explain the code snippet that depicts the creation of a **final** method mentioned in slide 13.

Final Class

A class declared **final** cannot be inherited or subclassed. Such a class becomes a standard and must be used as it is. The variables and methods of a class declared **final** are also implicitly **final**.

The reason for declaring a class as **final** is to limit extensibility and to prevent the modification of the class definition.

Explain the code snippet that shows the creation of a **final** class mentioned in slide 14.

Explain the code snippet demonstrates an example of creation of a final class mentioned in slides 15 and 16.

As the final class cannot be subclassed, this provides security and efficiency benefits. Java local classes can only reference local variables and parameters that are declared as final. A visible advantage of declaring Java variable as static final is, the compiled java class results in faster performance.

Tips:

If the variable is the reference, this means that the variable cannot be re-bound to reference another object.

Slide 17

Let us understand rules and best practices for using field modifiers.

Rules and Best Practices for Using Field Modifiers

- ◆ Some of the rules for using field modifiers are as follows:

- Final fields cannot be volatile.
- Native methods in Java cannot have a body.
- Declaring a transient field as static or final should be avoided as far as possible.
- Native methods violate Java's platform independence characteristic. Therefore, they should not be used frequently.
- A transient variable may not be declared as final or static.

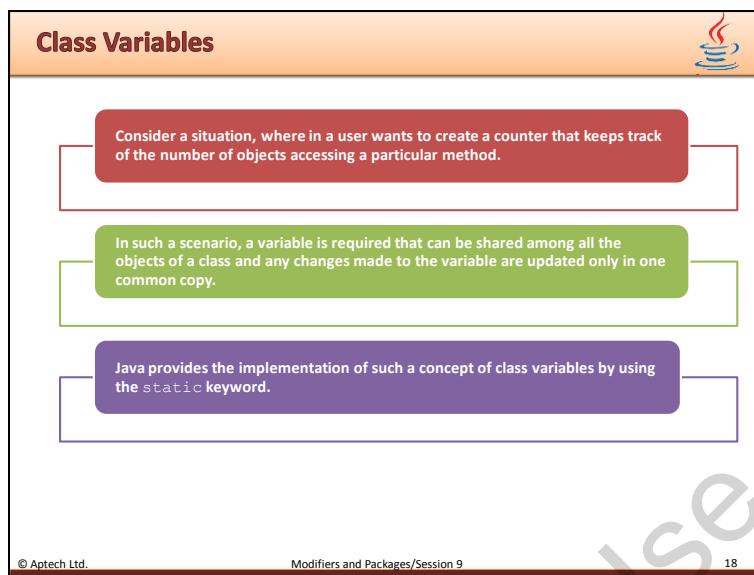
© Aptech Ltd. Modifiers and Packages/Session 9 17

Using slide 17, explain rules and best practices for using field modifiers.

Explain the rules listed on the slide 17 for using field modifiers.

Slide 18

Let us understand class variables.



The slide has a title 'Class Variables' at the top left. At the top right is a small orange icon of a flame or swirl. The main content area contains three colored callout boxes: a red one at the top with the text 'Consider a situation, where in a user wants to create a counter that keeps track of the number of objects accessing a particular method.', a green one in the middle with the text 'In such a scenario, a variable is required that can be shared among all the objects of a class and any changes made to the variable are updated only in one common copy.', and a purple one at the bottom with the text 'Java provides the implementation of such a concept of class variables by using the `static` keyword.' At the bottom left is the copyright notice '© Aptech Ltd.', in the center is 'Modifiers and Packages/Session 9', and at the bottom right is the page number '18'.

Using slide 18, explain class variables.

Consider a situation, where in a user wants to create a counter that keeps track of the number of objects accessing a particular method. In such a scenario, a variable is required that can be shared among all the objects of a class and any changes made to the variable are updated only in one common copy.

Java provides the implementation of such a concept of class variables by using the `static` keyword.

When a number of objects are created from the same class, they each have their own distinct copies of instance variables. Sometimes, you want to have variables that are common to all objects. This is accomplished with the static modifier. Fields that have static modifier in their declaration are called static fields or class variables.

Every instance of class shares a class variable, which is in one fixed location in memory. Any object can change the value of a class variable, but class variable can also be manipulated without creating an instance of the class.

Slide 19

Let us understand declaring class variables.

The slide has a title 'Declaring Class Variables' at the top left. On the right side, there is a small orange icon of a coffee cup with steam. Below the title, there is a bulleted list of seven points, each enclosed in a horizontal bar of a different color (purple, dark blue, medium blue, light blue, and white). At the bottom of the slide, there is a section with a bullet point followed by code examples.

- Class variables are also known as `static` variables.
- Note that the `static` variables are not constants.
- Such variables are associated with the class rather than with any object.
- All instances of the class share the same value of the class variable.
- The value of a `static` variable can be modified using class methods or instance methods.
- Unlike instance variable, there exists only one copy of a class variable for all objects in one fixed location in memory.
- A `static` variable declared as `final` becomes a constant whose value cannot be modified.

◆ For example,

```
static int PI=3.14; // static variable-can be modified  
static final int PI=3.14; // static constant-cannot be modified
```

© Aptech Ltd. Modifiers and Packages/Session 9 19

Using slide 19, explain declaring class variables.

Class variables are also known as static variables. Note that the static variables are not constants. Such variables are associated with the class rather than with any object. All instances of the class share the same value of the class variable.

The value of a static variable can be modified using class methods or instance methods. Unlike instance variable, there exists only one copy of a class variable for all objects in one fixed location in memory. A static variable declared as final becomes a constant whose value cannot be modified.

The class body contains all the code that provides for the life cycle of the object created from the class: constructors for initializing new object, declaration for the fields that provide the state of the class and its objects and methods to implement the behavior of the class and its objects.

Slides 20 to 25

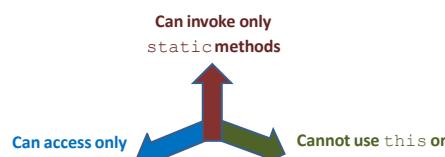
Let us understand creating static variables, static methods, and static blocks.

Creating Static Variables, Static Methods, and Static Blocks 1-6



- One can also create static methods and static initializer blocks along with static variables.
- Static variables and methods can be manipulated without creating an instance of the class.
- This is because there is only one copy of a static data member that is shared by all objects.
- A static method can only access static variables and not instance variables.

- Methods declared as static have the following restrictions:



Can invoke only static methods

Can access only static data

Cannot use this or super keywords

© Aptech Ltd. Modifiers and Packages/Session 9 20

Creating Static Variables, Static Methods, and Static Blocks 2-6



- Generally, a constructor is used to initialize variables.
- However, a static block can also be used to initialize static variables because static block is executed even before the main() method is executed.
- It is used when a block of code needs to be executed during loading of the class by JVM.
- Execution of Java code starts from static blocks and not from main() method. It is enclosed within {} braces.
- There can be more than one static block in a program. They can be placed anywhere in the class.
- A static initialization block can reference only those class variables that have been declared before it.

© Aptech Ltd. Modifiers and Packages/Session 9 21

Creating Static Variables, Static Methods, and Static Blocks 3-6



- Following code snippet demonstrates an example of static variables, static method, and static block:

```
package session9;
public class StaticMembers {
    // Declare and initialize static variable
    public static int staticCounter = 0;

    // Declare and initialize instance variable
    int instanceCounter = 0;
    /**
     * static block
     */
    static{
        System.out.println("I am a static block");
    }

    /**
     * Static method
     */
    @return void
```

© Aptech Ltd. Modifiers and Packages/Session 9 22

Creating Static Variables, Static Methods, and Static Blocks**4-6**

```
/*
public static void staticMethod(){
    System.out.println("I am a static method");
}

/**
 * Displays the value of static and instance counters
 *
 * @return void
 */
public void displayCount(){

    //Increment the static and instance variable
    staticCounter++;
    instanceCounter++;

    // Print the value of static and instance variable
    System.out.println("Static counter is:"+ staticCounter);
    System.out.println("Instance counter is:"+ instanceCounter);
}
```

© Aptech Ltd.

Modifiers and Packages/Session 9

23

Creating Static Variables, Static Methods, and Static Blocks**5-6**

```
/*
 * @param args the command line arguments
 */
public static void main(String[] args) {

    System.out.println("I am the main method");
    // Invoke the static method using the class name
    StaticMembers.staticMethod();

    // Create first instance of the class
    StaticMembers objStatic1 = new StaticMembers();
    objStatic1.displayCount();

    // Create second instance of the class
    StaticMembers objStatic2 = new StaticMembers();
    objStatic2.displayCount();

    // Create third instance of the class
    StaticMembers objStatic3 = new StaticMembers();
    objStatic3.displayCount();
}
```

© Aptech Ltd.

Modifiers and Packages/Session 9

24

Creating Static Variables, Static Methods, and Static Blocks**6-6**

- ◆ Following figure shows the output of the program:

Output - Session9 (run)

```
run:
I am a static block
I am the main method
I am a static method
Static counter is:1
Instance counter is:1
Static counter is:2
Instance counter is:1
Static counter is:3
Instance counter is:1
```

- ◆ From the figure it is clear that the **static block** is executed even before the **main()** method.
- ◆ Also, the value of **static counter** is incremented to 1, 2, 3, ..., and so on whereas the value of **instance counter** remains 1 for all the objects.
- ◆ This is because a separate copy of the **instance counter** exists for each object.
- ◆ However, for the **static variable**, only one copy exists per class.
- ◆ Every object increments the same copy of the variable, **staticCounter**.
- ◆ Thus, by using a **static counter**, a user can keep track of the number of instances of a class.

© Aptech Ltd.

Modifiers and Packages/Session 9

25

Using slides 20 to 25, explain creating static variables, static methods, and static blocks.

One can also create `static` methods and `static` initializer blocks along with static variables. `Static` variables and methods can be manipulated without creating an instance of the class. This is because there is only one copy of a static data member that is shared by all objects. A static method can only access static variables and not instance variables.

Methods declared as static have the following restrictions:

- Can access only static data
- Can invoke only static methods
- Cannot use `this` or `super` keywords

Generally, a constructor is used to initialize variables. However, a static block can also be used to initialize static variables because static block is executed even before the `main()` method is executed. It is used when a block of code needs to be executed during loading of the class by JVM.

Execution of Java code starts from static blocks and not from `main()` method. It is enclosed within {} braces. There can be more than one static block in a program. They can be placed anywhere in the class. A static initialization block can reference only those class variables that have been declared before it.

Explain the code snippet that demonstrates an example of static variables, static method, and static block mentioned in slides 22 to 25.

Static variables are initialized only once, at the start of the execution. A static variable can be accessed directly by the class name and does not need any object.

Static method belongs to the class and not to the object. Static method can access only static data and can call only other static methods. Static method cannot refer to 'this' keyword.

Tips:

Static variables will be initialized first, before the initialization of static variables.

Slides 26 and 27

Let us understand packages.

Packages 1-2



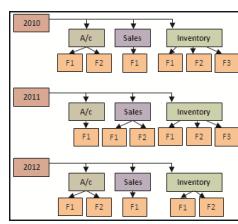
Consider a situation where a user has about fifty files of which some are related to sales, others are related to accounts, and some are related to inventory.

Also, the files belong to different years. All these files are kept in one section of a cupboard.

Now, when a particular file is required, the user has to search the entire cupboard. This is very time consuming and difficult.

For this purpose, the user creates separate folders and divides the files according to the years and further groups them according to the content.

- This is depicted in the following figure:



© Aptech Ltd. Modifiers and Packages/Session 9 26



Packages 2-2

- ◆ Similarly, in Java, one can organize the files using packages.

A package is a namespace that groups related classes and interfaces and organizes them as a unit.

- ◆ For example, one can keep source files in one folder, images in another, and executables in yet another folder. Packages have the following features:

- A package can have sub packages.
- A package cannot have two members with the same name.
- If a class or interface is bundled inside a package, it must be referenced using its fully qualified name, which is the name of the Java class including its package name.
- If multiple classes and interfaces are defined within a package in a single Java source file, then only one of them can be public.
- Package names are written in lowercase.
- Standard packages in the Java language begin with `java` or `javax`.

© Aptech Ltd. Modifiers and Packages/Session 9 27

Using slides 26 and 27, explain packages.

Consider a situation where in a user has about fifty files of which some are related to sales, others are related to accounts, and some are related to inventory. Also, the files belong to different years. All these files are kept in one section of a cupboard. Now, when a particular file is required, the user has to search the entire cupboard. This is very time consuming and difficult.

For this purpose, the user creates separate folders and divides the files according to the years and further groups them according to the content. Similarly, in Java, one can organize the files using packages.

A package is a namespace that groups related classes and interfaces and organizes them as a unit.

For example, one can keep source files in one folder, images in another, and executable in yet another folder. Then, explain the features of the packages listed on slide 27.

A Java package is a mechanism for organizing Java classes into namespace. Java packages can be stored in compressed files called JAR files. It allows classes to download faster as a group rather than one at a time. Package provides a unique namespace for the types it contain. Classes in the package can access each other package-access members.

Programmers also used packages to organize classes belonging to the same category or providing similar functionality.

In-Class Question:

After you finish explaining the packages, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



How the class or interface is referenced inside a package?

Answer:

It can be referenced using the name of the Java class including its package name.

Slide 28

Let us understand advantages of using packages.

Advantages of Using Packages



- One can easily determine that these classes are related.
- One can know where to find the required type that can provide the required functions.
- The names of classes of one package would not conflict with the class names in other packages as the package creates a new namespace.
- For example, `myPackage1.Sample` and `myPackage2.Sample`.
- One can allow classes within one package to have unrestricted access to one another while restricting access to classes outside the package.
- Packages can also store hidden classes that can be used within the package, but are not visible or accessible outside the package.
- Packages can also have classes with data members that are visible to other classes, but not accessible outside the package.
- When a program from a package is called for the first time, the entire package gets loaded into the memory.
- Due to this, subsequent calls to related subprograms of the same package do not require any further disk Input/Output (I/O).

© Aptech Ltd. Modifiers and Packages/Session 9 28

Using slide 28, explain advantages of using packages.

Explain them the reasons to group classes and interfaces in Java.

Slides 29 to 39

Let us understand type of packages.

Type of Packages 1-11



- ◆ The Java platform comes with a huge class library which is a set of packages.
- ◆ These classes can be used in applications by including the packages in a class.
- ◆ This library is known as the Application Programming Interface (API).
- ◆ Every Java application or applet has access to the core package in the API, the `java.lang` package.
- ◆ For example,

 The `String` class stores the state and behavior related to character strings.

 The `File` class allows the developer to create, delete, compare, inspect, or modify a file on the file system.

 The `Socket` class allows the developer to create and use network sockets.

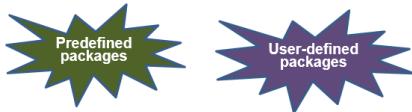
 The various Graphical User Interface (GUI) control classes such as `Button`, `Checkbox`, and so on provide ready to use GUI controls.

© Aptech Ltd. Modifiers and Packages/Session 9 29

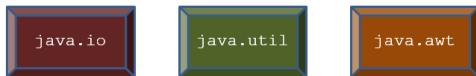
Type of Packages 2-11



- The different types of Java packages are as follows:



- The predefined packages are part of the Java API. Predefined packages that are commonly used are as follows:



Type of Packages 3-11



- To create user-defined packages, perform the following steps:

1

- Select an appropriate name for the package by using the following naming conventions:
 - Package names are usually written in all lower case to avoid conflict with the names of classes or interfaces.
 - Companies usually attach their reversed Internet domain name as a prefix to their package names.
 - For example, `com.sample.mypkg` for a package named `mypkg` created by a programmer at sample.com.
 - Naming conflicts occurring in the projects of a single company are handled according to the naming conventions specific to that company.
 - This is done usually by including the region name or the project name after the company name.
 - For example, `com.sample.myregion.mypkg`.
 - Package names should not begin with `java` or `javax` as they are used for packages that are part of Java API.

Type of Packages 4-11



- In certain cases, the Internet domain name may not be a valid package name.
- For example, if the domain name contains special characters such as hyphen, if the package name consists of a reserved Java keyword such as `char`, or if the package name begins with a digit or some other character that is illegal to use as the beginning of a Java package name.
- In such a case, it is advisable to use an underscore as shown in the following table:

Domain Name	Suggested Package Name
sample-name.sample.org	org.sample.sample_name
sample.int	int_sample
007name.sample.com	com.sample._007name

2

- Create a folder with the same name as the package.

3

- Place the source files in the folder created for the package.

Type of Packages 5-11



- 4** • Add the package statement as the first line in all the source files under that package as depicted in the following code snippet:

```
package session9;
class StaticMembers{
    public static void main(String[] args)
    {}
}
```

- Note that there can only be one package statement in a source file.

- 5** • Save the source file **StaticMembers.java** in the package **session9**.

- 6** • Compile the code as follows:

javac StaticMembers.java

OR

- Compile the code with **-d** option as follows:
javac -d . StaticMembers.java
- where, **-d** stands for directory and **'.'** stands for current directory.
- The command will create a sub-folder named **session9** and store the compiled class file inside it.

Type of Packages 6-11



- 7** • From the parent folder of the source file, execute it using the fully qualified name as follows:

java session9.StaticMembers

- Java allows the user to import the classes from predefined as well as user-defined packages using an import statement.
- However, the access specifiers associated with the class members will determine if the class members can be accessed by a class of another package.
- A member of a **public** class can be accessed outside the package by doing any of the following:

- ↳ Referring to the member class by its fully qualified name, that is,
package-name.class-name.
- ↳ Importing the package member, that is,
import package-name.class-name.
- ↳ Importing the entire package, that is,
import package-name.*.

Type of Packages 7-11

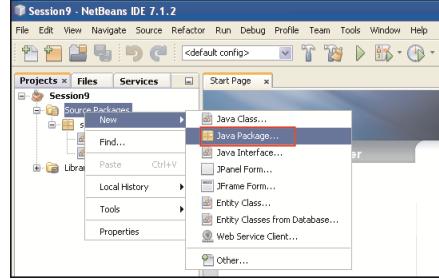


- To create a new package using NetBeans IDE, perform the following steps:

- 1** • Open the project in which the package is to be created.
• In this case **Session9** project has been chosen.

- 2** • Right-click **Source Packages** → **New** → **Java Package** to display the **New Java Package** dialog box.

- For example, in the following figure, the project **Session9** is opened and the **Java Package** option is selected:



Type of Packages 8-11

3 • Type **userpkg** in the Package Name box of the New Java Package dialog box that is displayed.

4 • Click **Finish**. The **userpkg** package is created as shown in the following figure:

5 • Right-click **userpkg** and select **New → Java Class** to add a new class to the package.

6 • Type **UserClass** as the Class Name box of the New Java Class dialog box and click **Finish**.

© Aptech Ltd. Modifiers and Packages/Session 9 36

Type of Packages 9-11

7 • Type the code in the class as depicted in the following code snippet:

```
package userpkg;
// Import the predefined and user-defined packages
import java.util.ArrayList;
import session9.StaticMembers;

public class UserClass {

    // Instantiate ArrayList class of java.util package
    ArrayList myCart = new ArrayList(); // line 1

    /**
     * Initializes an ArrayList
     * @return void
     */
    public void createList() {

        // Add values to the list
        myCart.add("Doll");
    }
}
```

© Aptech Ltd. Modifiers and Packages/Session 9 37

Type of Packages 10-11

```
myCart.add("Bus");
myCart.add("Teddy");
// Print the list
System.out.println("Cart contents are:" + myCart);
}

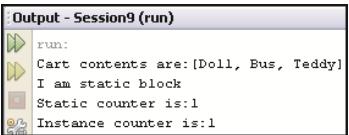
/**
 * @param args the command line arguments
 */
public static void main(String[] args) {

    // Instantiate the UserClass class
    UserClass objUser = new UserClass();
    objUser.createList(); // Invoke the createList() method

    // Instantiate the StaticMembers class
    StaticMembers objStatic = new StaticMembers();
    objStatic.displayCount(); // Invoke the displayCount() method
}
}
```

© Aptech Ltd. Modifiers and Packages/Session 9 38

Type of Packages 11-11



- The two import statements namely, `java.util.ArrayList` and `session9.StaticMembers` are used to import the packages `java.util` and `session9` into the `UserClass` class.
- Following figure shows the output of the program:

© Aptech Ltd. Modifiers and Packages/Session 9 39

Using slides 29 to 39, explain type of packages.

Tell them that every Java application or applet has access to the core package in the API, the `java.lang` package.

For example, the `String` class stores the state and behavior related to character strings. The `File` class allows the developer to create, delete, compare, inspect, or modify a file on the file system. The `Socket` class allows the developer to create and use network sockets. The various Graphical User Interface (GUI) control classes such as `Button`, `Checkbox`, and so on provide ready to use GUI controls.

Tell them that Java packages are of two types namely, predefined packages and user-defined packages.

The predefined packages are part of the Java API. Some of the predefined packages that are commonly used are `java.io`, `java.util`, `java.awt`, and so on.

The predefined packages are the type of Java packages that contain wide range of classes and methods to perform different functionalities. They are also termed as Java API packages.

User-defined packages are Java package type created by user. The user can group user defined classes into user defined packages.

Explain the steps to create user-defined packages mentioned in slides 31 to 34.

Explain the steps to create a new package using NetBeans IDE mentioned in slides 35 to 39.

Tips:

Package creates a new namespace hence, there will not be any name conflicts with names in other packages.

Slides 40 to 55

Let us understand creating .jar files for deployment.

Creating .jar Files for Deployment 1-16

- ◆ All the source files of a Java application are bundled into a single archive file called the Java Archive (JAR).
- ◆ The .jar file contains the class files and additional resources associated with the application.
- ◆ The .jar file format provides several advantages as follows:

Security	• The .jar file can be digitally signed so that only those users who recognize your signature can optionally grant the software security privileges that the software might not otherwise have.
Decrease in Download Time	• The source files bundled in a .jar file can be downloaded to a browser in a single HTTP transaction without having to open a new connection for each file.
File Compression	• The .jar format compresses the files for efficient storage.

© Aptech Ltd. Modifiers and Packages/Session 9 40

Creating .jar Files for Deployment 2-16

Packaging for Extensions	• The extension framework in Java allows adding additional functionality to the Java core platform. • The .jar file format defines the packaging for extensions. For example, Java 3D and Java Mail extensions developed by Sun Microsystems.
Package Sealing	• Java provides an option to seal the packages stored in the .jar files so that the packages can enforce version consistency. • When a package is sealed within a .jar file, it implies that all classes defined in that package must be available in the same .jar file.
Package Versioning	• A .jar file can also store additional information about the files, such as vendor and version information.
Portability	• The .jar files are packaged in a ZIP file format. • This enables the user to use them for tasks such as lossless data compression, decompression, archiving, and archive unpacking.

© Aptech Ltd. Modifiers and Packages/Session 9 41

Creating .jar Files for Deployment 3-16

- ◆ To perform basic tasks with .jar files, one can use the Java Archive Tool.
- ◆ This tool is provided with the JDK.
- ◆ The Java Archive Tool is invoked by using the `jar` command.
- ◆ The basic syntax for creating a .jar file is as follows:

Syntax

```
jar cf jar-file-name input-file-name(s)
```

where,

- c: indicates that the user wants to CREATE a .jar file.
- f: indicates that the output should go to a file instead of stdout.
- jar-file-name: represents the name of the resulting .jar file. Any name can be used for a .jar file. The .jar extension is provided with the file name, though it is not required.
- input-file-name (s): represents a list of one or more files to be included in the .jar separated by a space. This argument can contain the wildcard symbol '*' as well. If any of the input files specified is a directory, the contents of that directory are added to the .jar recursively.

© Aptech Ltd. Modifiers and Packages/Session 9 42

Creating .jar Files for Deployment 4-16



- The options `c` and `f` can be used in any order, but without any space in between.
- The `jar` command generates a compressed `.jar` file and places it by default in the current directory.
- Also, it will generate a default manifest file for the `.jar` file.
- The metadata in the `.jar` file such as entry names, contents of the manifest, and comments must be encoded in UTF8.
- Some of the other options, apart from `cf`, available with the `jar` command are listed in the following table:

Option	Description
<code>v</code>	Produces VERBOSE output on stdout while the <code>.jar</code> is being built. The output displays the name of each of the files that are included in the <code>.jar</code> file.
<code>0 (zero)</code>	Indicates that the <code>.jar</code> file must not be compressed.
<code>M</code>	Indicates that the default manifest file must not be created.
<code>m</code>	Allows inclusion of manifest information from an existing manifest file. <code>jar cmf existing-manifest-name jar-file-name input-file-name(s)</code>
<code>-c</code>	Used to change directories during execution of the command.

© Aptech Ltd.

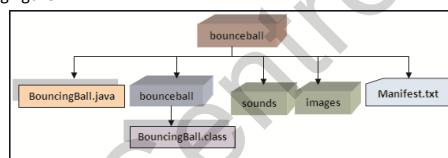
Modifiers and Packages/Session 9

43

Creating .jar Files for Deployment 5-16



- When a `.jar` file is created, the time of creation is stored in the `.jar` file.
- Therefore, even if the contents of the `.jar` file are not changed, if the `.jar` file is created multiple times, the resulting files will not be exactly identical.
- For this reason, it is advisable to use versioning information in the manifest file instead of creation time, to control versions of a `.jar` file.
- Consider the following files of a simple **BouncingBall** game application as shown in the following figure:



- The figure shows the **BouncingBall** application with the source file `BouncingBall.java`, class file `BouncingBall.class`, `sounds` directory, `images` directory, and `Manifest.txt` file.
- `sounds` and `images` are subdirectories that contain the sound files and animated `.gif` images used in the application.

© Aptech Ltd.

Modifiers and Packages/Session 9

44

Creating .jar Files for Deployment 6-16



- To create a `.jar` of the application using command line, perform the following steps:

1

- Create the directory structure as shown in the earlier figure.

2

- Create a text file with the code depicted in the following code snippet:

```

package bounceball;
public class BouncingBall {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        System.out.println("This is the bouncing ball game");
    }
}
  
```

3

- Save the file as `BouncingBall.java` in the source package `bounceball`.

© Aptech Ltd.

Modifiers and Packages/Session 9

45

Creating .jar Files for Deployment 7-16



- 4 • Compile the .java file at command prompt by writing the following command:

```
javac -d . BouncingBall.java
```

- The command will create a subfolder with the same name **bounceball** and store the class file **BouncingBall.java** in that directory.

- 5 • Create a text file with the Main-class attribute as shown in the following figure:

```
Manifest - Notepad
File Edit Format View Help
Manifest-Version: 1.0
Main-Class: bounceball.BouncingBall
```

© Aptech Ltd.

Modifiers and Packages/Session 9

46

Creating .jar Files for Deployment 8-16



- 6 • Save the file as **Manifest.txt** in the source **bounceball** folder.

- The **Manifest.txt** file will be referred by the Jar tool for the Main class during .jar creation.
- This will inform the Jar tool about the starting point of the application.

- 7 • To package the application in a single .jar named **BouncingBall.jar**, write the following command:

```
jar cvmf Manifest.txt bounceball.jar
bounceball/BouncingBall.class sounds images
```

- This will create a **bounceball.jar** file in the source folder as shown in the following figure:



© Aptech Ltd.

Modifiers and Packages/Session 9

47

Creating .jar Files for Deployment 9-16



- The command options **cvmf** indicate that the user wants to create a .jar file with verbose output using the existing manifest file, **Manifest.txt**.
- The name of the output file is specified as **bounceball.jar** instead of **stdout**.
- Further, the name of the class file is provided with its location as **bounceball/BouncingBall.class** followed by the directory names **sounds** and **images** so that the respective files under these directories are also included in the .jar file.

- 8 • To execute the .jar file at command prompt, type the following command:

```
java -jar bounceball.jar
```

- The command will execute the **main()** method of the class of the .jar file and print the following output:
This is the bouncing ball game.

© Aptech Ltd.

Modifiers and Packages/Session 9

48

Creating .jar Files for Deployment 10-16

- Following figure shows the entire series of steps for creation of .jar file with the verbose output and the final output after .jar file execution:

```

E:\bounceball>set path="C:\Program Files\Java\jdk1.7.0\bin"
E:\bounceball>javac -d . BouncingBall.java
E:\bounceball>jar cvf Manifest.txt bounceball.jar BouncingBall.class
E:\bounceball>java -jar bounceball.jar
This is the bouncing ball game
  
```

© Aptech Ltd.

Modifiers and Packages/Session 9

49

Creating .jar Files for Deployment 11-16

Since sounds and images are directories, the Jar tool will recursively place the contents in the .jar file.

The resulting .jar file **BouncingBall.jar** will be placed in the current directory.

The use of the option 'v' will show the verbose output of all the files that are included in the .jar file.

In the example, the files and directories in the archive retained their relative path names and directory structure.

One can use the -c option to create a .jar file in which the relative paths of the archived files will not be preserved.

- For example, suppose one wants to put sound files and images used by the **BouncingBall** program into a .jar file, and that all the files should be on the top level, with no directory hierarchy.

- One can accomplish this by executing the following command from the parent directory of the sounds and images directories:

```
jar cf SoundImages.jar -c sounds . -c images .
```

© Aptech Ltd.

Modifiers and Packages/Session 9

50

Creating .jar Files for Deployment 12-16

- Here, '-c sounds' directs the Jar tool to the sounds directory and the '.' following '-c sounds' directs the Jar tool to archive all the contents of that directory.

- Similarly, '-c images .' performs the same task with the images directory.

- The resulting .jar file would consist of all the sound and image files of the sounds and images folder as follows:

```
META-INF/MANIFEST.MF
beep.au
failure.au
ping.au
success.au
greenball.gif
redball.gif
table.gif
```

- However, if the following command is used without the -c option:

```
jar cf SoundImages.jar sounds images
```

© Aptech Ltd.

Modifiers and Packages/Session 9

51

Creating .jar Files for Deployment 13-16

- The resulting .jar file would have the following contents:

```
META-INF/MANIFEST.MF
sounds/beep.au
sounds/failure.au
sounds/ping.au
sounds/success.au
images/greenball.gif
images/redball.gif
images/table.gif
```

- Following table shows a list of frequently used jar command options:

Task	Command
To create a .jar file	jar cf jar-file-name input-file-name(s)
To view contents of a .jar file	jar tf jar-file-name
To extract contents of a .jar file	jar xf jar-file-name
To run the application packaged into the .jar file, Manifest.txt file is required with Main-class header attribute.	java -jar jar-file-name

Creating .jar Files for Deployment 14-16

- To create a .jar file of the application using NetBeans IDE, perform the following steps:

- 1 • Create a new package **bounceball** in the **Session9** application.
- 2 • Create a new java class named **BouncingBall.java** within the **bounceball** package.
- 3 • Type the code depicted in the following code snippet in the **BouncingBall** class:

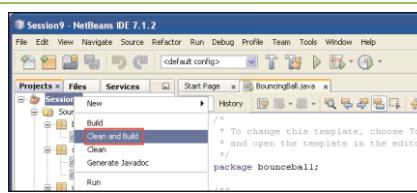
```
package bounceball;
public class BouncingBall {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args)
    {
        System.out.println("This is the bouncing ball game");
    }
}
```

- 4 • Set **BouncingBall.java** as the main class in the **Run** properties of the application.

Creating .jar Files for Deployment 15-16

- 5 • Run the application by clicking the **Run** icon on the toolbar.
- 6 • The output will be shown in the **Output** window.

- To create the .jar file, right-click the **Session9** application and select **Clean and Build** option as shown in the following figure:



- The IDE will build the application and generate the .jar file.

- A message as shown in the following figure will be displayed to the user in the status bar, once .jar file generation is finished:



Creating .jar Files for Deployment 16-16

- The Clean and Build command creates a new dist folder into the application folder and places the .jar file into it as shown in the following figure:

User can load this .jar file in any device that has a JVM and execute it by double-clicking the file or run it at command prompt by writing the following command:
java -jar Session9.jar

© Aptech Ltd. Modifiers and Packages/Session 9 55

Using slides 40 to 55, explain creating .jar files for deployment.

All the source files of a Java application are bundled into a single archive file called the Java Archive (JAR). The .jar file contains the class files and additional resources associated with the application.

Explain the .jar file format provides several advantages mentioned in slides 40 and 41.

To perform basic tasks with .jar files, one can use the Java Archive Tool. This tool is provided with the JDK. The Java Archive Tool is invoked by using the jar command.

Then, explain the basic syntax for creating a .jar file. Tell them that the options c and f can be used in any order, but without any space in between. The jar command generates a compressed .jar file and places it by default in the current directory. Also, it will generate a default manifest file for the .jar file.

The metadata in the .jar file such as entry names, contents of the manifest, and comments must be encoded in UTF8.

Explain some of the other options available with the jar command are listed in the slide 43.

When a .jar file is created, the time of creation is stored in the .jar file. Therefore, even if the contents of the .jar file are not changed, if the .jar file is created multiple times, the resulting files will not be exactly identical. For this reason, it is advisable to use versioning information in the manifest file instead of creation time, to control versions of a .jar file.

Consider the following files of a simple **BouncingBall** game application as shown in the slide 44.

Explain the steps to create a .jar of the application using command line mentioned in slides 45 to 49.

Tell them that since sounds and images are directories, the Jar tool will recursively place the contents in the .jar file. The resulting .jar file **BouncingBall.jar** will be placed in the current directory. The use of the option 'v' will show the verbose output of all the files that are included in the .jar file.

In the example, the files and directories in the archive retained their relative path names and directory structure. One can use the `-c` option to create a .jar file in which the relative paths of the archived files will not be preserved.

Explain the example, suppose one wants to put sound files and images used by the **BouncingBall** program into a .jar file, and that all the files should be on the top level, with no directory hierarchy mentioned in slides 50 to 52.

Explain the steps to create a .jar file of the application using NetBeans IDE mentioned in slides 53 to 55.

JAR file packaged with the ZIP file format, so you can use them for task such as data compression, archiving, decompression, and archive unpacking. These tasks are among the most common uses of JAR files.

Explain the benefits of JAR file format:

- Security- You can digitally sign the content of JAR file.
- Compression- the JAR format allows compressing your file for efficient storage.
- Package Versioning- A JAR files can hold data about the files it contains such as version information.
- Portability- The mechanism for handling JAR files is a standard part of Java platform core API.

In-Class Question:

After you finish explaining the creating .jar files for deployment, you will ask the students an In-class question. This will help you in reviewing their understanding of the topic.



Why the resulting files will not be identical?

Answer:

Because when a .jar file is created, the time of creation is stored in the .jar file.

Slide 56

Let us summarize the session.

Summary



- ◆ Field and method modifiers are used to identify fields and methods that have controlled access to users.
- ◆ The volatile modifier allows the content of a variable to be synchronized across all running threads.
- ◆ A thread is an independent path of execution within a program.
- ◆ The native modifier indicates that the implementation of the method is in a language other than Java such as C or C++.
- ◆ The transient modifier can only be used with instance variables. It informs the JVM not to store the variable when the object, in which it is declared, is serialized.
- ◆ The final modifier is used when modification of a class or data member is to be restricted.
- ◆ Class variables are also known as static variables and there exists only one copy of that variable for all objects.
- ◆ A package is a namespace that groups related classes and interfaces and organizes them as a unit.
- ◆ All the source files of a Java application are bundled into a single archive file called the Java Archive (JAR).

© Aptech Ltd. Modifiers and Packages/Session 9 56

In slide 56, you will summarize the session. End the session with a brief summary of what has been taught in the session.

9.3 Post Class Activities for Faculty

You should familiarize yourself with the topics of the next session which is based on inheritance and polymorphism.

Tips:

You can also check the Articles/Blogs/Expert Videos uploaded on the Online Varsity site to gain additional information related to the topics covered in the next session. You can also connect to online tutors on the Online Varsity site to ask queries related to the sessions.

Session 10 – Inheritance and Polymorphism

10.1 Pre-Class Activities

Before you commence the session, you should familiarize yourself with the topics of this session in-depth.

Here, you can discuss the key points with the students that were covered in the previous session. Prepare a question or two which will help you to relate the current session objectives.

10.1.1 Objectives

By the end of this session, the learners will be able to:

- Describe inheritance
- Explain the types of inheritance
- Explain super class and subclass
- Explain the use of super keyword
- Explain method overriding
- Describe Polymorphism
- Differentiate type of reference and type of objects
- Explain static and dynamic binding
- Explain virtual method invocation
- Explain the use of abstract keyword

10.1.2 Teaching Skills

To teach this session, you should be well-versed with concept of inheritance and its types, the subclass and superclass. You should be familiar with the method overriding and polymorphism, static, and dynamic binding and virtual method invocation.

You should teach the concepts in the theory class using the images provided. For teaching in the class, you are expected to use slides and LCD projectors.

Tips:

It is recommended that you test the understanding of the students by asking questions in between the class.

In-Class Activities:

Follow the order given here during In-Class activities.

Overview of the Session:

Give the students the overview of the current session in the form of session objectives. Show the students slide 2 of the presentation.

Objectives	
<ul style="list-style-type: none"> ◆ Describe inheritance ◆ Explain the types of inheritance ◆ Explain super class and subclass ◆ Explain the use of super keyword ◆ Explain method overriding ◆ Describe Polymorphism ◆ Differentiate type of reference and type of objects ◆ Explain static and dynamic binding ◆ Explain virtual method invocation ◆ Explain the use of abstract keyword 	

© Aptech Ltd. Inheritance and Polymorphism/Session 10 2

Tell the students that this session introduces the basic concept of inheritance and its types. The session explains the concept of subclass and superclass in inheritance. Also the session will introduce them with the concept of polymorphism and method overriding. The session explains the concept of static and dynamic binding in polymorphism. Finally, the session will explain them about virtual method invocation and abstract classes.

10.2 In-Class Explanations

Slides 3 to 5

Let us understand the concept of inheritance.

Introduction	
<ul style="list-style-type: none"> ◆ In the world around us, there are many animals and birds that eat the same type of food and have similar characteristics. ◆ Therefore, all the animals that eat plants can be categorized as herbivores, those that eat animals as carnivores, and those that eat both plants and animals as omnivores. ◆ This kind of grouping or classification of things is called <u>subcategorization</u> and the child groups are known as subclasses. ◆ Similarly, Java provides the concept of inheritance for creating subclasses of a particular class. ◆ Also, animals such as chameleon change their color based on the environment. ◆ Human beings also play different roles in their daily life such as father, son, husband, and so on. ◆ This means, that they behave differently in different situations. ◆ Similarly, Java provides a feature called polymorphism in which objects behave differently based on the context in which they are used. 	

© Aptech Ltd. Inheritance and Polymorphism/Session 10 3

Inheritance 1-2

In daily life, one often comes across objects that share a kind-of or is-a relationship with each other.

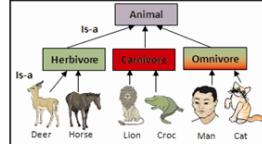
For example, Car is-a four-wheeler, a four-wheeler is-a vehicle, and a vehicle is-a machine.

Similarly, many other objects can be identified having such relationship. All such objects have properties that are common.

For example, all four wheelers have wipers and a rear view mirror.

All vehicles have a vehicle number, wheels, and engine irrespective of a four-wheeler or two-wheeler.

- ◆ Following figure shows some examples of is-a relationship:



© Aptech Ltd.

Inheritance and Polymorphism/Session 10

4

Inheritance 2-2

- ◆ The figure shows is-a relationship between different objects.
- ◆ For example, Deer is-a herbivore and a herbivore is-a animal.
- ◆ The common properties of all herbivores can be stored in class herbivore.
- ◆ Similarly, common properties of all types of animals such as herbivore, carnivore, and omnivore can be stored in the Animal class.
- ◆ Thus, the class Animal becomes the top-level class from which the other classes such as Herbivore, Carnivore, and Omnivore inherit properties and behavior.
- ◆ The classes Deer, Horse, Lion, and so on inherit properties from the classes Herbivore, Carnivore, and so on.
- ◆ This is called inheritance.
- ◆ Thus, inheritance in Java is a feature through which classes can be derived from other classes and inherit fields and methods from those classes.

© Aptech Ltd.

Inheritance and Polymorphism/Session 10

5

Using slides 3 to 5, explain the concept of inheritance.

Discuss with the students the example on animals falling in similar categories. For example, the animals that eat plants can be categorized as herbivores, those that eat animals as carnivores, and those that eat both plants and animals as omnivores. Now, discuss that in each category, the animals and birds normally eat the same type of food and have similar characteristics.

Based on the discussed example, explain that Java also provides the concept of inheritance where the classes can extend and reuse the features of existing class and implement the behavior.

Similarly, you can discuss one more example of inheritance seen in human beings. Inform them that in a family, traits of parents are often inherited by their children. In addition to having unique traits and behavior of their own, children inherit those of their parents.

Introduce them to the concept of polymorphism in Java. Tell them that polymorphism in Java allows an objects can take many forms, which means it can behave differently based on the context in which it has been used.

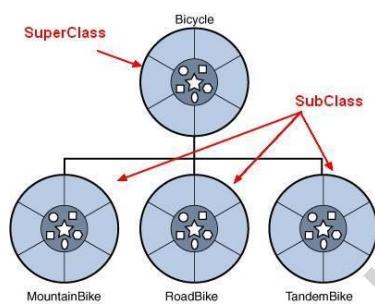
Explain them the kind of relationship that exists in inheritance. In daily life, one often comes across objects that share a kind-of or is-a relationship with each other. For example, Car is-a four-wheeler, a four-wheeler is-a vehicle, and a vehicle is-a machine. Similarly, many other objects can be identified having such relationship. All such objects have properties that are common.

For example, all four wheelers have wipers and a rear view mirror. All vehicles have a vehicle number, wheels, and engine irrespective of a four-wheeler or two-wheeler.

Explain the figure that shows some examples of is-a relationship mentioned in slides 4 and 5.

Tell them that inheritance is based on is-a relationship where a class that is derived from another class is called a *subclass* (also a *derived class*, *extended class*, or *child class*). The class from which the subclass is derived is called a *superclass* (also a *base class* or a *parent class*).

Explain them that in Java every class will have one and only one direct superclass. The following figure depicts the superclass and subclass relationship:



For example, in the absence of any other explicit superclass, every class is implicitly a subclass of object.

In-Class Question:

After you finish explaining concept of inheritance, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



What is the advantage of using inheritance in Java applications?

Answer:

Inheritance encourages the code reusability and allows the developers to extend the functionalities of the existing classes in the application.

Slides 6 and 7

Let us understand features and types of inheritance.

Features and Terminologies 1-2



- The class that is derived from another class is called a subclass, derived class, child class, or extended class.
- The class from which the subclass is derived is called a super class, base class, or parent class.
- The derived class can reuse the fields and methods of the existing class without having to re-write or debug the code again.
- A subclass inherits all the members such as fields, nested classes, and methods from its super class except those with `private` access specifier.
- Constructors of a class are not considered as members of a class and are not inherited by subclasses.
- The child class can invoke the constructor of the super class from its own constructor.
- Members having default accessibility in the super class are not inherited by subclasses of other packages.
- The subclass will have its own specific characteristics along with those inherited from the super class.

© Aptech Ltd.

Inheritance and Polymorphism/Session 10

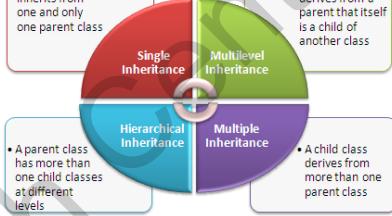
6

Features and Terminologies 2-2



- There are several types of inheritance as shown in the following figure:

Single	Multilevel	Hierarchical	Multiple



- A child class inherits from one and only one parent class
- A child class derives from a parent that itself is a child of another class
- A parent class has more than one child classes at different levels
- A child class derives from more than one parent class

© Aptech Ltd.

Inheritance and Polymorphism/Session 10

7

Using slides 6 and 7, explain the features and types of inheritance.

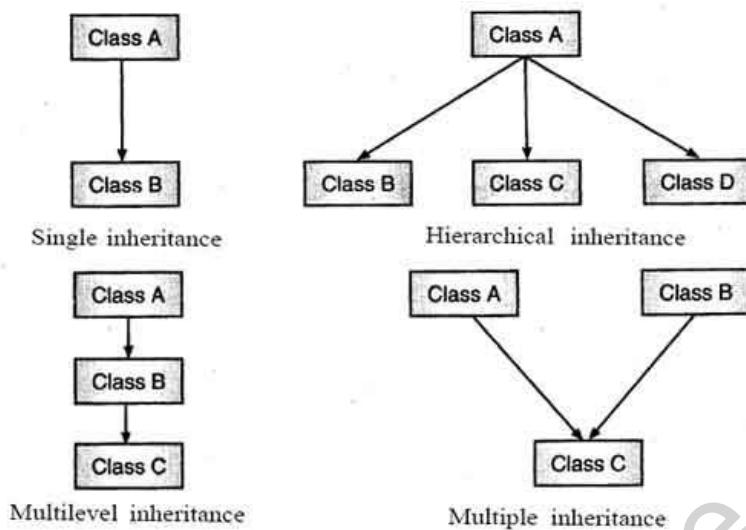
Discuss with them the advantages of inheritance. Tell them that it saves work and encourages code reuse and customization, because the specialized class inherits all the properties and methods of the general class, only new properties and methods need to be implemented. In other words, a user can simply derive the new class from the existing class without having to write or debug the new code. The subclass will have its own specific characteristics along with those inherited from the super class.

A subclass inherits all the members such as fields, nested classes, and methods from its super class except those with `private` access specifier.

Constructors

Constructors of a superclass are not inherited by subclasses. Thus, the child class can invoke the constructor of the super class from its own constructor. Also, the members having default modifier in the super class are not inherited by subclasses of other packages.

Then, explain the different types of inheritance supported by an object-oriented language. The following figure shows the different types of inheritance:



Single Inheritance - A child class inherits from one and only one parent class.

Multilevel Inheritance - A child class derives from a parent that itself is a child of another class. Here, one class extends only one. The lower most subclass can make use of all of its super classes members.

Multiple Inheritance - A child class derives from more than one parent class. It means one class extending more than one base class.

Hierarchical Inheritance - A parent class has more than one child classes at different levels. It means one class derived from many subclasses.

In-Class Question:

After you finish explaining features and terminologies, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



Which type of inheritance have more than one parent classes?

Answer:
Multiple

Tips:

Java language does not support multiple inheritance.

Slides 8 to 13

Let us understand working with super class and subclass in inheritance.

Working with Super class and Subclass 1-6

- Within a subclass, one can use the inherited members as is, hide them, replace them, or enhance them with new members as follows:

The inherited members, including fields and methods, can be directly used just like any other fields.

One can declare a field with the same name in the subclass as the one in the super class. This will lead to hiding of super class field which is not advisable.

One can declare new fields in the subclass that are not present in the super class. These members will be specific to the subclass.

One can write a new instance method with the same signature in the subclass as the one in the super class. This is called method overriding.

A new static method can be created in the subclass with the same signature as the one in the super class. This will lead to hiding of the super class method.

One can declare new methods in the subclass that are not present in the super class.

A subclass constructor can be used to invoke the constructor of the super class, either implicitly or by using the keyword super.

The extends keyword is used to create a subclass. A class can be directly derived from only one class.

© Aptech Ltd.

Inheritance and Polymorphism/Session 10

8

Working with Super class and Subclass 2-6

- If a class does not have any super class, it is implicitly derived from `Object` class.
- The syntax for creating a subclass is as follows:

Syntax

```
public class <class1-name> extends <class2-name>
{
...
}
```

where,

- `class1-name`: Specifies the name of the child class.
- `class2-name`: Specifies the name of the parent class.

- Following code snippet demonstrates the creation of super class `Vehicle`:

```
package session10;
public class Vehicle {
    // Declare common attributes of a vehicle
    protected String vehicleNo; // Variable to store vehicle number
    protected String vehicleName; // Variable to store vehicle name
    protected int wheel; // Variable to store number of wheels
}
```

© Aptech Ltd.

Inheritance and Polymorphism/Session 10

9

Working with Super class and Subclass 3-6

```
/*
 * Accelerates the vehicle
 *
 * @return void
 */
public void accelerate(int speed) {
    System.out.println("Accelerating at:" + speed + " kmph");
}
```

- The parent class `Vehicle` consists of common attributes of a vehicle such as `vehicleNo`, `vehicleName`, and `wheels`.
- Also, it consists of a common behavior of a vehicle, that is, `accelerate()` that prints the speed at which the vehicle is accelerating.
- Following code snippet demonstrates the creation of subclass `FourWheeler`:

```
package session10;
class FourWheeler extends Vehicle {
    // Declare a field specific to child class
    private boolean powerSteer; // Variable to store steering information
}
```

© Aptech Ltd.

Inheritance and Polymorphism/Session 10

10

Working with Super class and Subclass 4-6

```
/*
 * Parameterized constructor to initialize values based on user input
 *
 * @param vId a String variable storing vehicle ID
 * @param vName a String variable storing vehicle name
 * @param numWheels an integer variable storing number of wheels
 * @param pSteer a String variable storing steering information
 */
public FourWheeler(String vId, String vName, int numWheels, boolean
pSteer) {
    // Attributes inherited from parent class
    vehicleNo = vId;
    vehicleName = vName;
    wheels = numWheels;

    // Child class' own attribute
    powerSteer = pSteer;
}
```

© Aptech Ltd.

Inheritance and Polymorphism/Session 10

11

Working with Super class and Subclass 5-6

```
/*
 * Displays vehicle details
 *
 * @return void
 */
public void showDetails() {
    System.out.println("Vehicle no:" + vehicleNo);
    System.out.println("Vehicle Name:" + vehicleName);
    System.out.println("Number of Wheels:" + wheels);

    if(powerSteer == true)
        System.out.println("Power Steering:Yes");
    else
        System.out.println("Power Steering:No");
}

/**
 * Define the TestVehicle.java class
 */
public class TestVehicle {
```

© Aptech Ltd.

Inheritance and Polymorphism/Session 10

12

Working with Super class and Subclass 6-6

```
/*
 * @param args the command line arguments
 */
public static void main(String[] args) {
    // Create an object of child class and specify the values
    FourWheeler objFour = new FourWheeler("LA-09 CS-1406", "Volkswagen",
4, true);
    objFour.showDetails(); // Invoke the child class method
    objFour.accelerate(200); // Invoke the inherited method
}
```

- Following figure shows the output of the program:

Output - Session10 (run)

```
run:
Vehicle no:LA-09 CS-1406
Vehicle Name:Volkswagen
Number of Wheels:4
Power Steering:Yes
Accelerating at:200 kmph
```

© Aptech Ltd.

Inheritance and Polymorphism/Session 10

13

Using slides 8 to 13, explain working with super class and subclass in inheritance.

Explain the features of a subclass that one can use for the inherited members as listed in slide 8.

Then, explain the syntax of the subclass where <class1-name> is the name of the subclass and <class2-name> is the name of the superclass.

Explain the code snippet that demonstrates the creation of super class **Vehicle** mentioned in slides 9 and 10. The parent class **Vehicle** consists of common attributes of a vehicle such as **vehicleNo**, **vehicleName**, and **wheels**. Also, it consists of a common behavior of a vehicle, that is, **accelerate()** that prints the speed at which the vehicle is accelerating.

Explain the code snippet that demonstrates the creation of subclass **FourWheeler** mentioned in slides 11 to 13. The code depicts the child class **FourWheeler** with its own attribute **powerSteer**. The values for the inherited attributes and its own attribute are specified in the constructor. The **showDetails()** method is used to display all the details. The class **TestVehicle** consists of the **main()** method. In the **main()** method, the object **objFour** of child class is created and the parameterized constructor is invoked by passing appropriate arguments. Next, the **showDetails()** method is invoked to print the details. Also, the child class object is used to invoke the **accelerate()** method inherited from parent class and the value of speed is passed as argument.

Slides 14 to 18

Let us understand overriding methods.

Overriding Methods 1-5

- ◆ Java allows creation of an instance method in a subclass having the same signature and return type as an instance method of the super class.
- ◆ This is called method overriding.
- ◆ Method overriding allows a class to inherit behavior from a super class and then, to modify the behavior as needed.
- ◆ Rules to remember when overriding:
 - The overriding method must have the same name, type, and number of arguments as well as return type as the super class method.
 - An overriding method cannot have a weaker access specifier than the access specifier of the super class method.
- ◆ The **accelerate()** method can be overridden in the subclass as shown in the following code snippet:

```
package session10;
class FourWheeler extends Vehicle {

    // Declare a field specific to child class
    private boolean powerSteer; // Variable to store steering information
```

© Aptech Ltd. Inheritance and Polymorphism/Session10 14

Overriding Methods 2-5

```
/*
 * Parameterized constructor to initialize values based on user input
 *
 * @param vId a String variable storing vehicle ID
 * @param vName a String variable storing vehicle name
 * @param numWheels an integer variable storing number of wheels
 * @param pSteer a String variable storing steering information
 */
public FourWheeler(String vId, String vName, int numWheels, boolean
pSteer) {

    // attributes inherited from parent class
    vehicleNo=vId;
    vehicleName=vName;
    wheels=numWheels;

    // child class' own attribute
    powerSteer=pSteer;
}
```

© Aptech Ltd. Inheritance and Polymorphism/Session10 15

Overriding Methods 3-5

```
/*
 * Displays vehicle details
 *
 * @return void
 */
public void showDetails() {
    System.out.println("Vehicle no:" + vehicleNo);
    System.out.println("Vehicle Name:" + vehicleName);
    System.out.println("Number of Wheels:" + wheels);

    if(powerSteer==true){
        System.out.println("Power Steering:Yes");
    } else {
        System.out.println("Power Steering:No");
    }
}

/**
 * Overridden method
 * Accelerates the vehicle
 *
 * @return void
 */

```

© Aptech Ltd.

Inheritance and Polymorphism/Session10

16

Overriding Methods 4-5

```
/*
@override
public void accelerate(int speed) {
    System.out.println("Maximum acceleration:" + speed + " kmph");
}

/**
 * Define the TestVehicle.java class
 */
public class TestVehicle {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // Create an object of child class and specify the values
        FourWheeler objFour = new FourWheeler("LA-09 CS-1406", "Volkswagen",
        4, true);
        objFour.showDetails(); // Invoke child class method
        objFour.accelerate(200); // Invoke inherited method
    }
}
```

© Aptech Ltd.

Inheritance and Polymorphism/Session10

17

Overriding Methods 5-5

- The `accelerate()` method is overridden in the child class with the same signature and return type but with a modified message.
- Notice the use of `@Override` annotation on top of the method.

Annotations provide additional information about a program. Annotations have no direct effect on the functioning of the code they annotate.

- Following figure shows the output of the code:

```
Output - Session10 (run)
run:
Vehicle no:LA-09 CS-1406
Vehicle Name:Volkswagen
Number of Wheels:4
Power Steering:Yes
Maximum acceleration:200 kmph
```

- The `accelerate()` method now prints the message specified in the subclass.
- Since the `accelerate()` method is overridden in the subclass, the subclass version of the `accelerate()` method is invoked and not the super class `accelerate()` method.

© Aptech Ltd.

Inheritance and Polymorphism/Session10

18

Using slides 14 to 18, explain overriding methods.

Java allows creation of an instance method in a subclass having the same signature and return type as an instance method of the super class. This is called method overriding. Method overriding allows a class to inherit behavior from a super class and then, to modify the behavior as needed.

The purpose of overriding is to define new or different behavior for the objects of the subclass. For instance, taking a real-world example, a 1970's model of a Toyota car also had a basic action of driving and a latest model of a Toyota car also has the same basic action but with greater and better performance. So, though the basic action still remains the same, how the models implement the action is different in each case.

Discuss with the students the rules that one should remember when overriding methods. They are as follows:

- The overriding method must match in name, type, number of arguments, and return type with the overridden method.
- An overridden method cannot have a weaker access specifier than the access specifier of the method it overrides. For example, a `protected` method in a superclass can be overridden by a `public` method having same signature, but a `public` method in a superclass cannot be overridden by a `protected` method in the subclass.

Tips:

The ability of a subclass to override a method allows a class to inherit from a superclass whose behavior is 'close enough' and then to modify behavior as needed. The overriding method has the same name, number and type of parameters, and return type as the method that it overrides. An overriding method can also return a subtype of the type returned by the overridden method.

Explain the `accelerate()` method that can be overridden in the subclass as in the code snippet mentioned in slides 14 to 18.

The `accelerate()` method is overridden in the child class with the same signature and return type but with a modified message. Notice the use of `@Override` on top of the method. This is an annotation that instructs the compiler that the method that follows is overridden from the parent class. If the compiler detects that such a method does not exist in the super class, it will generate compilation error.

Slides 19 to 24

Let us understand accessing super class constructor and methods.

Accessing Super class Constructor and Methods 1-6



In the code, the subclass constructor is used to initialize the values of the common attributes inherited from the super class `Vehicle`.

This is not the correct approach because all the subclasses of the `Vehicle` class will have to initialize the values of common attributes every time in their constructors.

Java allows the subclass to invoke the super class constructor and methods using the keyword `super`.

Also, when the `accelerate()` method is overridden in the subclass, the statement(s) written in the `accelerate()` method of the super class, `Vehicle`, are not printed.

- To address these issues, one can use:
 - the `super` keyword to invoke the super class method using `super.method-name()`.
 - the `super()` method to invoke the super class constructors from the subclass constructors.

© Aptech Ltd. Inheritance and Polymorphism/Session 10 19

Accessing Super class Constructor and Methods 2-6

- Following code snippet demonstrates the modified super class **Vehicle.java** using a parameterized constructor:

```
package session10;
class Vehicle {

    protected String vehicleNo; // Variable to store vehicle number
    protected String vehicleName; // Variable to store vehicle name
    protected int wheels; // Variable to store number of wheels

    /**
     * Parameterized constructor to initialize values based on user input
     *
     * @param vId a String variable storing vehicle ID
     * @param vName a String variable storing vehicle name
     * @param numWheels an integer variable storing number of wheels
     */
    public Vehicle(String vId, String vName, int numWheels){
        vehicleNo=vId;
        vehicleName=vName;
        wheels=numWheels;
    }
}
```

© Aptech Ltd.

Inheritance and Polymorphism/Session10

20



Accessing Super class Constructor and Methods 3-6

- Following code snippet depicts the modified subclass **FourWheeler.java** using the super keyword to invoke super class constructor and methods:

```
/*
 * Accelerates the vehicle
 *
 * @return void
 */
public void accelerate(int speed){
    System.out.println("Accelerating at:" + speed + " kmph");
}
```

© Aptech Ltd.

Inheritance and Polymorphism/Session10

21



Accessing Super class Constructor and Methods 4-6

```
* @param pSteer a String variable storing steering information
*/
public FourWheeler(String vId, String vName, int numWheels, boolean
pSteer) {
    // Invoke the super class constructor
    super(vId,vName,numWheels);
    powerSteer=pSteer;
}

/**
 * Displays vehicle details
 *
 * @return void
 */
public void showDetails() {
    System.out.println("Vehicle no:" + vehicleNo);
    System.out.println("Vehicle Name:" + vehicleName);
    System.out.println("Number of Wheels:" + wheels);

    if(powerSteer==true){
        System.out.println("Power Steering:Yes");
    }
}
```

© Aptech Ltd.

Inheritance and Polymorphism/Session10

22



For Aptechnical Use Only

Accessing Super class Constructor and Methods 5-6

```

        else{
            System.out.println("Power Steering:No");
        }
    /**
     * Overridden method
     * Displays the acceleration details of the vehicle
     *
     * @return void
     */
    @Override
    public void accelerate(int speed){
        // Invoke the super class accelerate() method
        super.accelerate(speed);
        System.out.println("Maximum acceleration:"+ speed + " kmph");
    }
}
*/
public class TestVehicle {
    /**

```

© Aptech Ltd. Inheritance and Polymorphism/Session10

23

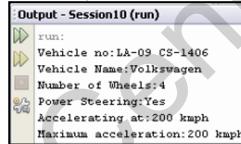
Accessing Super class Constructor and Methods 6-6

```

    * @param args the command line arguments
    */
    public static void main(String[] args){
        FourWheeler objFour = new FourWheeler("LA-09 CS-1406", "Volkswagen", 4,
                true);
        objFour.showDetails();
        objFour.accelerate(200);
    }
}

```

- The `super()` method is used to call the super class constructor from the child class constructor.
- Similarly, the `super.accelerate()` statement is used to invoke the super class `accelerate()` method from the child class.
- Following figure shows the output of the code:



© Aptech Ltd.

Inheritance and Polymorphism/Session10

24

Using slides 19 to 24, explain accessing super class constructor and methods.

In the code explained on the earlier slides, the subclass constructor is used to initialize the values of the common attributes inherited from the super class **Vehicle**. This is not the correct approach because all the subclasses of the **Vehicle** class will have to initialize the values of common attributes every time in their constructors.

Java allows the subclass to invoke the super class constructor and methods using the keyword `super`. Also, when the `accelerate()` method is overridden in the subclass, the statement(s) written in the `accelerate()` method of the super class, **Vehicle**, are not printed.

The basic purpose of `super` keyword is to enable access to those members of the superclass that are presently hidden by members in the subclass. To invoke the super class method, use the syntax: `super.method-name()` and to invoke super class constructor, use the syntax: `super()`.

Explain the code snippet demonstrates the modified super class **Vehicle.java** using a parameterized constructor mentioned in slides 20 and 21.

Explain the code snippet depicts the modified subclass **FourWheeler.java** using the `super` keyword to invoke super class constructor and methods mentioned in slides 21 to 24.

In the code, the use of `super()` to call the super class constructor from the child class constructor. Similarly, the `super.accelerate()` statement is used to invoke the super class `accelerate()` method from the child class.

In-Class Question:

After you finish explaining accessing super class constructor and methods, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



How to work with the `super()` keyword?

Answer:

To invoke the super class method, use the syntax: `super.method-name()` and to invoke super class constructor, use the syntax: `super()`.

Slide 25

Let us understand the concept of polymorphism.

Polymorphism

The word polymorph is a combination of two words namely, 'poly' which means 'many' and 'morph' which means 'forms'.

Thus, polymorph refers to an object that can have many different forms.

This principle can also be applied to subclasses of a class that can define their own specific behaviors as well as derive some of the similar functionality of the super class.

The concept of method overriding is an example of polymorphism in object-oriented programming in which the same method behaves in a different manner in super class and in subclass.

© Aptech Ltd. Inheritance and Polymorphism/Session 10 25

Using slide 25, explain the concept of polymorphism.

The word polymorph is a combination of two words namely, 'poly' which means 'many' and 'morph' which means 'forms'. Thus, polymorph refers to an object that can have many different forms. This principle can also be applied to subclasses of a class that can define their own specific behaviors as well as derive some of the similar functionality of the super class.

The concept of method overriding is an example of polymorphism in object-oriented programming in which the same method behaves in a different manner in super class and in subclass.

The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object. It is the ability to create a variable, a function, or an object that has more than one form.

Tips:

Any Java object that can pass more than one is-a test is considered to be polymorphic.

Slides 26 to 33

Let us understand static and dynamic binding.

Understanding Static and Dynamic Binding 1-8

- Some important differences between static and dynamic binding are listed in the following table:

Static Binding	Dynamic Binding
Static binding occurs at compile time.	Dynamic binding occurs at runtime.
Private, static, and final methods and variables use static binding and are bounded by compiler.	Virtual methods are bounded at runtime based upon the runtime object.
Static binding uses object type information for binding. That is, the type of class.	Dynamic binding uses reference type to resolve binding.
Overloaded methods are bounded using static binding.	Overridden methods are bounded using dynamic binding.

- Following code snippet demonstrates an example of static binding:

```
package session10;
class Employee {

    String empId; // Variable to store employee ID
    String empName; // Variable to store employee name
    int salary; // Variable to store salary
    float commission; // Variable to store commission
```

© Aptech Ltd.

Inheritance and Polymorphism/Session10

26

Understanding Static and Dynamic Binding 2-8

```
/*
 * Parameterized constructor to initialize the variables
 *
 * @param id a String variable storing employee id
 * @param name a String variable storing employee name
 * @param sal an integer variable storing salary
 *
 */
public Employee(String id, String name, int sal) {
    empId=id;
    empName=name;
    salary=sal;
}
/*
 * Calculates commission based on sales value
 * @param sales a float variable storing sales value
 *
 * @return void
 */
public void calcCommission(float sales){
    if(sales > 10000)
        commission = salary * 20 / 100;
```

© Aptech Ltd.

Inheritance and Polymorphism/Session10

27

Understanding Static and Dynamic Binding 3-8

```
else
    commission=0;
}

/*
 * Overloaded method. Calculates commission based on overtime
 * @param overtime an integer variable storing overtime hours
 *
 * @return void
 */
public void calcCommission(int overtime){
    if(overtime > 0)
        commission = salary/30;
    else
        commission = 0;
}

/*
 * Displays employee details
 *
 * @return void
 */

```

© Aptech Ltd.

Inheritance and Polymorphism/Session10

28

Understanding Static and Dynamic Binding 4-8



```
public void displayDetails(){
    System.out.println("Employee ID:"+empId);
    System.out.println("Employee Name:"+empName);
    System.out.println("Salary:"+salary);
    System.out.println("Commission:"+commission);
}
/**
 * Define the EmployeeDetails.java class
 */
public class EmployeeDetails {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args)
    {
        // Instantiate the Employee class object
        Employee objEmp = new Employee("E001","Maria Nemeth", 40000);
        // Invoke the calcCommission() with float argument
        objEmp.calcCommission(20000F);
        objEmp.displayDetails(); // Print the employee details
    }
}
```

© Aptech Ltd.

Inheritance and Polymorphism/Session 10

29

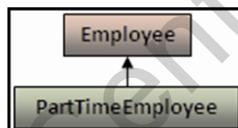
Understanding Static and Dynamic Binding 5-8



- In the example, when the `calcCommission()` method is executed, the method with float argument gets invoked because it was bounded during compile time based on the type of variable, that is, float.
- Following figure shows the output of the code:

```
Output - Session10 (run)
run:
Employee ID:E001
Employee Name:Maria Nemeth
Salary:40000
Commission:8000.0
```

- Now, consider the class hierarchy shown in the following figure:



© Aptech Ltd.

Inheritance and Polymorphism/Session 10

30

Understanding Static and Dynamic Binding 6-8



- Following code snippet demonstrates an example of dynamic binding:

```
package session10;
class PartTimeEmployee extends Employee{
    // Subclass specific variable
    String shift; // Variable to store shift information

    /**
     * Parameterized constructor to initialize values based on user input
     *
     * @param id a String variable storing employee ID
     * @param name a String variable storing employee name
     * @param sal an integer variable storing salary
     * @param shift a String variable storing shift information
     */
    public PartTimeEmployee(String id, String name, int sal, String shift)
    {
        // Invoke the super class constructor
        super(id, name, sal);
        this.shift=shift;
    }
}
```

© Aptech Ltd.

Inheritance and Polymorphism/Session 10

31

Understanding Static and Dynamic Binding 7-8



```


    /**
     * Overridden method to display employee details
     */
    @Override
    public void displayDetails(){
        calcCommission(12); // Invoke the inherited method
        super.displayDetails(); // Invoke the super class display method
        System.out.println("Working shift:"+shift);
    }
}

/*
 * Modified EmployeeDetails.java
 */
public class EmployeeDetails {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args)
{
}


```

© Aptech Ltd.

Inheritance and Polymorphism/Session 10

32

Understanding Static and Dynamic Binding 8-8



```


// Instantiate the Employee class object
Employee objEmp = new Employee("E001", "Maria Nemeth", 40000);
objEmp.calcCommission(20000F); // Calculate commission
objEmp.displayDetails(); // Print the details
System.out.println("-----");

// Instantiate the Employee object as PartTimeEmployee
Employee objEmpl = new PartTimeEmployee("E002", "Rob Smith", 30000,
"Day");
objEmpl.displayDetails(); // Print the details
}
}


```

- Following figure shows the output of the code:

Output - Session10 (run)

Employee ID: E001
Employee Name: Maria Nemeth
Salary: 40000
Commission: 8000.0

Employee ID: E002
Employee Name: Rob Smith
Salary: 30000
Commission: 1000.0
Working shift: Day

© Aptech Ltd.

Inheritance and Polymorphism/Session 10

33

Using slides 26 to 33, explain understanding static and dynamic binding.

When the compiler resolves the binding of methods and method calls at compile time, it is called static binding or early binding. If the compiler resolves the method calls and the binding at runtime, it is called dynamic binding or late binding. All `static` method calls are resolved at compile time and therefore, static binding is done for all static method calls. The instance method calls are always resolved at runtime.

Explain some important differences between static and dynamic binding are listed in table in slide 26.

Then, explain the code snippet demonstrates an example of static binding mentioned in slides 26 to 30. The code snippet depicts a class `Employee` with four member variables. The constructor is used to initialize the member variables with the received values. The class consists of two `calcCommission()` methods. The first method calculates commission based on the sales done by the employee and the other based on the number of hours the employee worked overtime. The `displayDetails()` method is used to print the details of the employee.

Another class `EmployeeDetails` is created with the `main()` method. Inside the `main()` method, object `objEmp` of class `Employee` is created and the parameterized constructor is invoked with appropriate arguments. Next, the `calcCommission()` method is invoked with a `float` argument `20000F`.

Explain the code snippet demonstrates an example of dynamic binding mentioned in slides 31 to 33. The code snippet displays the **PartTimeEmployee** class that is inherited from the **Employee** class created earlier. The class has its own variable called **shift** which indicates the day or night shift for an employee. The constructor of **PartTimeEmployee** class calls the super class constructor using the **super** keyword to initialize the common attributes of an employee. Also, it initializes the **shift** variable.

The subclass overrides the **displayDetails()** method. Within the overridden method, the **calcCommission()** method is invoked with an integer argument. This will calculate commission based on overtime. Next, the super class **displayDetails()** method is invoked to display the basic details of the employee as well as the shift details.

The **EmployeeDetails** class is modified to create another object **objEmp1** of class **Employee**. However, the object is assigned the reference of class **PartTimeEmployee** and the constructor is invoked with four arguments. Next, the **displayDetails()** method is invoked to print the employee details.

In-Class Question:

After you finish explaining static and dynamic binding, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



Which type of binding method bounds the invocation of method call to the object at runtime?

Answer:

Dynamic

Slides 34 and 35

Let us understand to differentiate type of reference and type of object.

Differentiate Type of Reference and Type of Object 1-2



Upcasting

- ◆ In the earlier code, type of object of **objEmp1** is **Employee**.
- ◆ This means that the object will have all characteristics of an **Employee**.
- ◆ However, the reference assigned to the object was of **PartTimeEmployee**.
- ◆ This means that the object will bind with the members of **PartTimeEmployee** class during runtime.
- ◆ That is, object type is **Employee** and reference type is **PartTimeEmployee**.
- ◆ This is possible only when the classes are related with a parent-child relationship.
- ◆ Java allows casting an instance of a subclass to its parent class.
- ◆ This is known as **upcasting**.
- ◆ For example,

```
PartTimeEmployee objPT = new PartTimeEmployee();
Employee objEmp = objPT; // upcasting
```

- ◆ While **upcasting** a child object, the child object **objPT** is directly assigned to the parent class object **objEmp**.
- ◆ However, the parent object cannot access the members that are specific to the child class and not available in the parent class.

© Aptech Ltd. Inheritance and Polymorphism/Session 10 34

Differentiate Type of Reference and Type of Object 2-2



Downcasting

- ◆ Java also allows casting the parent reference back to the child type.
- ◆ This is because parent references an object of type child.
- ◆ Casting a parent object to child type is called downcasting because an object is being casted to a class lower down in the inheritance hierarchy.
- ◆ However, downcasting requires explicit type casting by specifying the child class name in brackets.
- ◆ For example,

```
PartTimeEmployee objPT1 = (PartTimeEmployee) objEmp;
// downcasting
```

© Aptech Ltd. Inheritance and Polymorphism/Session 10 35

Using slides 34 and 35, explain to differentiate type of reference and type of object.

Upcasting

Java allows casting an instance of a subclass to its parent class. This is known as upcasting. For example,

```
PartTimeEmployee objPT = new PartTimeEmployee();
Employee objEmp = objPT; // upcasting
```

While upcasting a child object, the child object `objPT` is directly assigned to the parent class object `objEmp`. However, the parent object cannot access the members that are specific to the child class and not available in the parent class.

In upcasting, if two types are compatible and destination type is larger than source type, then Java will perform the conversion automatically. For example, `int` type is always large enough to hold all valid byte values, so no explicit cast statement is required.

Downcasting

Java also allows casting the parent reference back to the child type. This is because parent references an object of type child.

Casting a parent object to child type is called downcasting because an object is being casted to a class lower down in the inheritance hierarchy. However, downcasting requires explicit type casting by specifying the child class name in brackets.

For example,

```
PartTimeEmployee objPT1 = (PartTimeEmployee) objEmp;
// downcasting
```

Upcasting and downcasting are important parts of Java which allow us to build complex programs using simple syntax. Upcasting can be done automatically, but downcasting must be manually done by the programmer.

Slide 36

Let us understand invocation of virtual method.

Invocation of Virtual Method

In the earlier code, during execution of the statement `Employee objEmp1= new PartTimeEmployee(...)`, the runtime type of the `Employee` object is determined.

The compiler does not generate error because the `Employee` class has a `displayDetails()` method.

At runtime, the method executed is referenced from the `PartTimeEmployee` object. This aspect of polymorphism is called virtual method invocation.

- ◆ The difference here is between the compiler and the runtime behavior.
 - The compiler checks the accessibility of each method and field based on the class definition whereas the behavior associated with an object is determined at runtime.
 - Since the object created was of `PartTimeEmployee`, the `displayDetails()` method of `PartTimeEmployee` is invoked even though the object type is `Employee`.
 - This is referred to as virtual method invocation and the method is referred to as virtual method.
 - In other languages such as C++, the same can be achieved by using the keyword `virtual`.

© Aptech Ltd. Inheritance and Polymorphism/Session 10 36

Using slide 36, explain invocation of virtual method.

In the earlier code, during execution of the statement `Employee objEmp1= new PartTimeEmployee(...)`, the runtime type of the `Employee` object is determined. The compiler does not generate error because the `Employee` class has a `displayDetails()` method.

At runtime, the method executed is referenced from the `PartTimeEmployee` object. This aspect of polymorphism is called virtual method invocation. In other languages such as C++, the same can be achieved by using the keyword `virtual`. In object-oriented programming, a virtual method is a method whose behavior can be overridden within an inheriting class by a function with the same signature to provide polymorphic behavior. The method which cannot be inherited for polymorphic behavior is not a virtual method.

Static method is not a virtual method in Java as they are bound to the class itself, so calling static method from class name or object does not provide polymorphic behavior to the static method.

Explain the difference between the compiler and the runtime behavior listed in the slide.

Tips:

Every non-static method in Java is by default virtual method except `final` and `private` methods.

Slides 37 to 45

Let us understand using the 'abstract' keyword.

Using the 'abstract' Keyword 1-9

Abstract method

An abstract method is one that is declared with the `abstract` keyword and is without an implementation, that is, without any body.

- The abstract method does not contain any '{}' brackets and ends with a semicolon.
- The syntax for declaring an abstract method is as follows:

Syntax

```
abstract <return-type> <method-name> (<parameter-list>);
```

where,
`abstract`: Indicates that the method is an abstract method.

- For example,
`public abstract void calculate();`

© Aptech Ltd. Inheritance and Polymorphism/Session 10 37

Using the 'abstract' Keyword 2-9

Abstract class

An abstract class is one that consists of abstract methods.

- Abstract class serves as a framework that provides certain behavior for other classes.
- The subclass provides the requirement-specific behavior of the existing framework.
- Abstract classes cannot be instantiated and they must be subclassed to use the class members.
- The subclass provides implementations for the abstract methods in its parent class.
- The syntax for declaring an abstract class is as follows:

Syntax

```
abstract class <class-name>
{
    // declare fields
    // define concrete methods
    [abstract <return-type> <method-name>(<parameter-list>);]
}
```

© Aptech Ltd. Inheritance and Polymorphism/Session 10 38

Using the 'abstract' Keyword 3-9

where,
`abstract`: Indicates that the method is an abstract method.

- For example,
`public abstract Calculator`
`{`
 `public float getPI() { // Define a concrete method`
 `return 3.14F;`
 `}`
 `abstract void Calculate(); // Declare an abstract method`
`}`
- Consider the class hierarchy as shown in following figure:

© Aptech Ltd. Inheritance and Polymorphism/Session 10 39

Using the 'abstract' Keyword 4-9



- Following code snippet demonstrates creation of abstract class and abstract method:

```
package session10;
abstract class Shape {
    private final float PI = 3.14F; // Variable to store value of PI
    /**
     * Returns the value of PI
     *
     * @return float
     */
    public float getPI(){
        return PI;
    }

    /**
     * Abstract method
     * @param val a float variable storing the value specified by user
     *
     * @return float
     */
    abstract void calculate(float val);
}
```

© Aptech Ltd.

Inheritance and Polymorphism/Session10

40

Using the 'abstract' Keyword 5-9



- Following code snippet demonstrates two subclasses **Circle** and **Rectangle** inheriting the abstract class **Shape**:

```
package session10;
/**
 * Define the child class Circle.java
 */
class Circle extends Shape{
    float area; // Variable to store area of a circle

    /**
     * Implement the abstract method to calculate area of circle
     *
     * @param rad a float variable storing value of radius
     * @return void
     */
    @Override
    void calculate(float rad){
        area = getPI() * rad * rad;
        System.out.println("Area of circle is:" + area);
    }
}
```

© Aptech Ltd.

Inheritance and Polymorphism/Session10

41

Using the 'abstract' Keyword 6-9



```
/**
 * Define the child class Rectangle.java
 */
class Rectangle extends Shape{

    float perimeter; // Variable to store perimeter value
    float length=10; // Variable to store length

    /**
     * Implement the abstract method to calculate the perimeter
     *
     * @param width a float variable storing width
     * @return void
     */
    @Override
    void calculate(float width){

        perimeter = 2 * (length+width);
        System.out.println("Perimeter of the Rectangle is:" + perimeter);
    }
}
```

© Aptech Ltd.

Inheritance and Polymorphism/Session10

42

Using the 'abstract' Keyword 7-9



- Following code snippet depicts the code for **Calculator** class that uses the subclasses based on user input:

```
package session10;
public class Calculator {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        Shape objShape; // Declare the Shape object
        String shape; // Variable to store the type of shape

        if(args.length==2) { // Check the number of command line arguments
            //Retrieve the value of shape from args[0]
            shape = args[0].toLowerCase(); // converting to lower case
            switch(shape){

                // Assign reference to Shape object as per user input
                case "circle": objShape = new Circle();
                objShape.calculate(Float.parseFloat(args[1]));
                break;

                case "rectangle": objShape = new Rectangle();
                objShape.calculate(Float.parseFloat(args[1]));
                break;

                case "triangle": objShape = new Triangle();
                objShape.calculate(Float.parseFloat(args[1]));
                break;

                default:
                    System.out.println("Usage: java Calculator <shape-name> <value>");
                    break;
            }
        }
    }
}
```

© Aptech Ltd.

Inheritance and Polymorphism/Session 10

43

Using the 'abstract' Keyword 8-9



```
case "rectangle": objShape = new Rectangle();
objShape.calculate(Float.parseFloat(args[1]));
break;
}
}
else{
// Error message to be displayed when arguments are not supplied
System.out.println("Usage: java Calculator <shape-name> <value>");
}
}
}
```

- To execute the example at command line, write the following command:
java Calculator Rectangle 12
- Note that the word **Circle** can be in any case.
- Within the code, it will be converted to lowercase.

© Aptech Ltd.

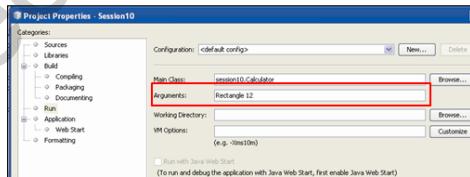
Inheritance and Polymorphism/Session 10

44

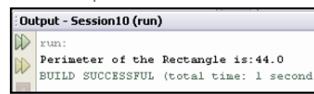
Using the 'abstract' Keyword 9-9



- To execute the example in **NetBeans IDE**, type the arguments in the **Arguments** box of **Run** property as shown in the following figure:



- Following figure shows the output of the code after execution:



- The output shows the perimeter of a rectangle.
- This is because the first command line argument was **Rectangle**.
- Therefore, within the `main()` method, the `switch` case for rectangle got executed.

© Aptech Ltd.

Inheritance and Polymorphism/Session 10

45

Using slides 37 to 45, explain Using the 'abstract' Keyword.

Java provides the abstract keyword to create a super class that serves as a generalized form that will be inherited by all of its subclasses. The methods of the super class serve as a contract or a standard that the subclass can implement in its own way.

Abstract method

An abstract method is one that is declared with the `abstract` keyword and is without an implementation, that is, without any body. The abstract method does not contain any '{ }' brackets and ends with a semicolon. An `abstract` method is only a contract that the subclass will provide its implementation.

Explain the syntax for declaring an abstract method in the class.

Tell them that declaring a method as abstract has two outcomes:

- The class must also be declared abstract. If a class contains an abstract method, the class must be abstract as well.
- Any child class must either override the abstract method or declare itself abstract.

Abstract class

An abstract class is one that consists of abstract methods. Abstract class serves as a framework that provides certain behavior for other classes. The subclass provides the requirement-specific behavior of the existing framework.

Abstract classes cannot be instantiated and they must be subclassed to use the class members. The subclass provides implementations for the abstract methods in its parent class.

The syntax for declaring an abstract class is as follows:

The following code shows the implementation of abstract class:

```
public abstract Calculator
{
    public float getPI(){ // Define a concrete method
        return 3.14F;
}
abstract void Calculate(); // Declare an abstract method
}
```

Explain the code snippet demonstrates creation of abstract class and abstract method mentioned in slide 40.

Explain the code snippet demonstrates two subclasses **Circle** and **Rectangle** inheriting the abstract class **Shape** mentioned in slides 41 and 42. The class **Circle** implements the abstract method **calculate()** to calculate the area of a circle. Similarly, the class **Rectangle** implements the abstract method **calculate()** to calculate the perimeter of a rectangle.

Explain the code snippet depicts the code for **Calculator** class that uses the subclasses based on user input mentioned 43 and 45. The class **Calculator** takes two arguments from the user at command line namely, the shape and the value for the shape. Notice the **Shape** class object **objShape**. It was mentioned earlier that an abstract class cannot be instantiated. That is, one cannot write **Shape objShape = new Shape()**. However, an abstract class can be assigned a reference of its subclasses.

The statement `args.length` checks the number of arguments supplied by the user. If the length is 2, the value for shape is extracted from the first argument `args[0]` and stored in the `shape` variable. Next, the switch statement evaluates the value of `shape` variable and accordingly assigns the reference of the appropriate shape to the `objShape` object. For example, if shape is circle, it assigns `new Circle()` as the reference. Then, using the object `objShape`, the `calculate()` method is invoked for the referenced subclass.

To execute the example at command line, write the following command:
`java Calculator Rectangle 12`

Slide 46

Let us summarize the session.

Summary



- ◆ Inheritance is a feature in Java through which classes can be derived from other classes and inherit fields and methods from classes it is inheriting.
- ◆ The class that is derived from another class is called a subclass, derived class, child class, or extended class. The class from which the subclass is derived is called a super class.
- ◆ Creation of an instance method in a subclass having the same signature and return type as an instance method of the super class is called method overriding.
- ◆ Polymorphism refers to an object that can have many different forms.
- ◆ When the compiler resolves the binding of methods and method calls at compile time, it is called static binding or early binding. If the compiler resolves the method calls and the binding at runtime, it is called dynamic binding or late binding.
- ◆ An abstract method is one that is declared with the abstract keyword without an implementation, that is, without any body.
- ◆ An abstract class is one that consists of abstract methods.
- ◆ An abstract class serves as a framework that provides certain pre-defined behavior for other classes that can be modified later as per the requirement of the inheriting class.

© Aptech Ltd.

Inheritance and Polymorphism/Session 10

46

In slide 46, you will summarize the session. End the session with a brief summary of what has been taught in the session.

10.3 Post Class Activities for Faculty

You should familiarize yourself with the topics of the next session which is based on interfaces and nested classes.

Tips:

You can also check the Articles/Blogs/Expert Videos uploaded on the Online Varsity site to gain additional information related to the topics covered in the next session. You can also connect to online tutors on the Online Varsity site to ask queries related to the sessions.

Session 11 – Interfaces and Nested Classes

11.1 Pre-Class Activities

Before you commence the session, you should familiarize yourself with the topics of this session in-depth.

Here, you can discuss the key points with the students that were covered in the previous session. Prepare a question or two which will help you to relate the current session objectives.

11.1.1 Objectives

By the end of this session, the learners will be able to:

- Describe Interface
- Explain the purpose of interfaces
- Explain implementation of multiple interfaces
- Describe Abstraction
- Explain Nested class
- Explain Member class
- Explain Local class
- Explain Anonymous class
- Describe Static nested class

11.1.2 Teaching Skills

To teach this session, you should be well-versed with concept and use of interface and implementation of multiple interfaces. You should be familiar with the abstraction, multiple class, nested class, and local class.

You should teach the concepts in the theory class using the images provided. For teaching in the class, you are expected to use slides and LCD projectors.

Tips:

It is recommended that you test the understanding of the students by asking questions in between the class.

In-Class Activities:

Follow the order given here during In-Class activities.

Overview of the Session:

Give the students the overview of the current session in the form of session objectives. Show the students slide 2 of the presentation.

Objectives	
<ul style="list-style-type: none"> ◆ Describe Interface ◆ Explain the purpose of interfaces ◆ Explain implementation of multiple interfaces ◆ Describe Abstraction ◆ Explain Nested class ◆ Explain Member class ◆ Explain Local class ◆ Explain Anonymous class ◆ Describe Static nested class 	
<small>© Aptech Ltd. Interfaces and Nested Classes/Session 11 2</small>	

Tell the students that this session explains the concept of interfaces in Java. The session explains how to declare the interfaces and implement them in Java class. The session also explains the concept of multiple interfaces. The session explains the concept of abstraction, multiple class, nested class, and local class. The session also explains the concept of anonymous class and static nested class in Java.

11.2 In-Class Explanations**Slide 3**

Let us understand the concept of interfaces and nested classes.

Introduction	
<ul style="list-style-type: none"> ◆ Java does not support multiple inheritance. ◆ However, there are several cases when it becomes mandatory for an object to inherit properties from multiple classes to avoid redundancy and complexity in code. ◆ For this purpose, Java provides a workaround in the form of interfaces. ◆ Also, Java provides the concept of nested classes to make certain types of programs easy to manage, more secure, and less complex. 	
<small>© Aptech Ltd. Interfaces and Nested Classes/Session 11 3</small>	

Using slide 3, explain the concept of interfaces and nested classes.

Java does not support multiple inheritance. However, there are several cases when it becomes mandatory for an object to inherit properties from multiple classes to avoid redundancy and complexity in code.

For this purpose, Java provides a workaround in the form of interfaces. Also, Java provides the concept of nested classes to make certain types of programs easy to manage, more secure, and less complex.

Slides 4 and 5

Let us understand the concept of interfaces in Java.

Interfaces 1-2



An interface in Java is a contract that specifies the standards to be followed by the types that implement it.

- ◆ The classes that accept the contract must abide by it.
- ◆ An interface and a class are similar in the following ways:

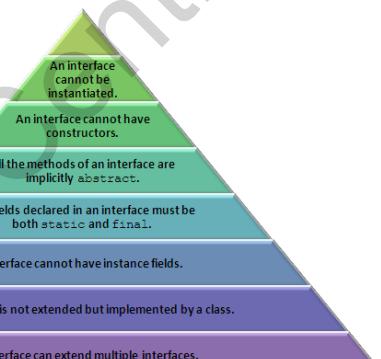
- An interface can contain multiple methods.
- An interface is saved with a .java extension and the name of the file must match with the name of the interface just as a Java class.
- The ~~bytecode~~ of an interface is also saved in a .class file.
- Interfaces are stored in packages and the ~~bytecode~~ file is stored in a directory structure that matches the package name.

© Aptech Ltd. Interfaces and Nested Classes/Session 11 4

Interfaces 2-2



◆ An interface and a class differ in several ways as follows:



- An interface cannot be instantiated.
- An interface cannot have constructors.
- All the methods of an interface are implicitly abstract.
- The fields declared in an interface must be both static and final.
- Interface cannot have instance fields.
- An interface is not extended but implemented by a class.
- An interface can extend multiple interfaces.

© Aptech Ltd. Interfaces and Nested Classes/Session 11 5

Using slides 4 and 5, explain the concept of Interfaces in Java.

Tell the students that there are a number of situations in software engineering when it is important for disparate groups of programmers to agree to a 'contract' that specifies how their software interacts. Each group should be able to write their code without any knowledge of how the other group's code is written. Generally speaking, *interfaces* are such contracts.

Consider a new employee who joins an organization to work with them usually signs a contract which states that he/she will abide by the rules laid down by the organization and comply with their guidelines. In the programming world too, a contract can be defined for classes, such that once they accept the contract they will abide by it. Such a contract is an interface.

An interface in Java is a contract that specifies the standards to be followed by the types that implement it. The classes that accept the contract must abide by it.

Then, explain the differences between an interface and a class in Java as presented in the figure in slide 5.

Writing an interface is similar to writing a class, however the difference is that the class describes the attributes and behaviors of an object. An interface contains the method declarations that a class can implement by signing the interface.

Slides 6 to 16

Let us understand purpose of interfaces.

Purpose of Interfaces 1-11

- In several situations, it becomes necessary for different groups of developers to agree to a 'contract' that specifies how their software interacts.
- Each group should have the liberty to write their code in their desired manner without having the knowledge of how the other groups are writing their code.
- Java interfaces can be used for defining such contracts.
- Interfaces do not belong to any class hierarchy, even though they work in conjunction with classes.
- Java does not permit multiple inheritance for which interfaces provide an alternative.
- In Java, a class can inherit from only one class but it can implement multiple interfaces.
- Therefore, objects of a class can have multiple types such as the type of their own class as well as the types of the interfaces that the class implements.

© Aptech Ltd. Interfaces and Nested Classes/Session 11 6

Purpose of Interfaces 2-11

- The syntax for declaring an interface is as follows:

Syntax

```
<visibility> interface <interface-name> extends <other-interfaces, ... >
{
    // declare constants
    // declare abstract methods
}
```

where,

- <visibility>: Indicates the access rights to the interface. Visibility of an interface is always public.
- <interface-name>: Indicates the name of the interface.
- <other-interfaces>: List of interfaces that the current interface inherits from.

- For example,

```
public interface Sample extends Interface1{
    static final int someInteger;
    public void someMethod();
}
```

© Aptech Ltd. Interfaces and Nested Classes/Session 11 7

Purpose of Interfaces 3-11



- ◆ In Java, interface names are written in CamelCase, that is, first letter of each word is capitalized.
- ◆ Also, the name of the interface describes an operation that a class can perform. For example,
interface Enumerable
interface Comparable
- ◆ Some programmers prefix the letter 'I' with the interface name to distinguish interfaces from classes. For example,
interface IEnumerable
interface IComparable
- ◆ Notice that the method declaration does not have any braces and is terminated with a semicolon.
- ◆ Also, the body of the interface contains only abstract methods and no concrete method.
- ◆ Since all methods in an interface are implicitly abstract, the abstract keyword is not explicitly specified with the method signature.

© Aptech Ltd.

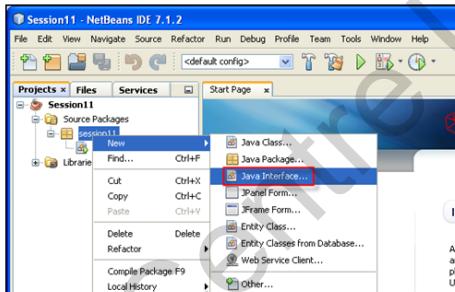
Interfaces and Nested Classes/Session 11

8

Purpose of Interfaces 4-11



- ◆ Consider the hierarchy of vehicles where IVehicle is the interface that declares the methods which the implementing classes such as TwoWheeler, FourWheeler, and so on can define.
- ◆ To create a new interface in NetBeans IDE, right-click the package name and select **New → Java Interface** as shown in the following figure:



© Aptech Ltd.

Interfaces and Nested Classes/Session 11

9

Purpose of Interfaces 5-11



- ◆ A dialog box appears where the user must provide a name for the interface and then, click **OK**. This will create an interface with the specified name.
- ◆ Following code snippet defines the interface, IVehicle:

```
package session11;
public interface IVehicle {
    // Declare and initialize constant
    static final String STATEID="LA-09"; // variable to store state ID

    /**
     * Abstract method to start a vehicle
     * @return void
     */
    public void start();

    /**
     * Abstract method to accelerate a vehicle
     * @param speed an integer variable storing speed
     * @return void
     */
    public void accelerate(int speed);
}
```

© Aptech Ltd.

Interfaces and Nested Classes/Session 11

10

Purpose of Interfaces 6-11



```
/*
 * Abstract method to apply a brake
 * @return void
 */
public void brake();

/*
 * Abstract method to stop a vehicle
 * @return void
 */
public void stop();
}
```

- The syntax to implement an interface is as follows:

Syntax

```
class <class-name> implements <Interface1>,...  
{  
    // class members  
    // overridden abstract methods of the interface(s)  
}
```

© Aptech Ltd.

Interfaces and Nested Classes/Session 11

11

Purpose of Interfaces 7-11



- Following code snippet defines the class **TwoWheeler** that implements the **IVehicle** interface:

```
package session11;  
class TwoWheeler implements IVehicle {  
  
    String ID; // variable to store vehicle ID  
    String type; // variable to store vehicle type  
  
    /**
     * Parameterized constructor to initialize values based on user input
     *  

     * @param ID a String variable storing vehicle ID
     * @param type a String variable storing vehicle type
     */  
    public TwoWheeler(String ID, String type){  
        this.ID = ID;  
        this.type = type;  
    }  
  
    /**
     * Overridden method, starts a vehicle
     *  

     * @return void
     */
}
```

© Aptech Ltd.

Interfaces and Nested Classes/Session 11

12

Purpose of Interfaces 8-11



```
* @return void
*/
@Override
public void start() {
    System.out.println("Starting the "+ type);
}

/**
 * Overridden method, accelerates a vehicle
 * @param speed an integer storing the speed
 * @return void
 */
@Override
public void accelerate(int speed) {
    System.out.println("Accelerating at speed:"+speed+ " kmph");
}

/**
 * Overridden method, applies brake to a vehicle
 * @return void
*/

```

© Aptech Ltd.

Interfaces and Nested Classes/Session 11

13

Purpose of Interfaces 9-11

```

/*
@Override
public void brake() {
    System.out.println("Applying brakes");
}

/**
 * Overridden method, stops a vehicle
 *
 * @return void
 */
@Override
public void stop() {
    System.out.println("Stopping the "+ type);
}

/**
 * Displays vehicle details
 *
 * @return void
 */
public void displayDetails(){
}

```

Interfaces and Nested Classes/Session 11

14

Purpose of Interfaces 10-11

```

        System.out.println("Vehicle No.: "+ STATEID+ " "+ ID);
        System.out.println("Vehicle Type.: "+ type);
    }

    /**
     * Define the class TestVehicle.java
     *
     */
    public class TestVehicle {

        /**
         * @param args the command line arguments
         */
        public static void main(String[] args){

            // Verify the number of command line arguments
            if(args.length==3) {

                // Instantiate the TwoWheeler class
                TwoWheeler objBike = new TwoWheeler(args[0], args[1]);

```

Interfaces and Nested Classes/Session 11

15

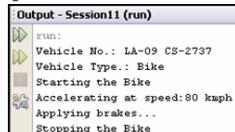
Purpose of Interfaces 11-11

```

        // Invoke the class methods
        objBike.displayDetails();
        objBike.start();
        objBike.accelerate(Integer.parseInt(args[2]));
        objBike.brake();
        objBike.stop();
    }
    else {
        System.out.println("Usage: java TwoWheeler <ID> <Type> <Speed>");
    }
}

```

- Following figure shows the output of the code when the user passes **CS-2723 Bike 80** as command line arguments:



Interfaces and Nested Classes/Session 11

16

Using slides 6 to 16, explain purpose of interfaces.

The two main benefits that are derived from using interfaces:

- An interface provides a means of setting a standard. It defines a contract that promotes reuse. If an object implements an interface then that object is promising to conform to a standard.
- An interface also provides a level of abstraction that makes programs easier to understand. Interfaces allow developers to start talking about the general way that code behaves without having to get into a lot of detailed specifics.

In Java, an interface is a collection of related methods without any body. These methods form the contract that the implementing class must agree with. When a class implements an interface, it becomes more formal about the behavior it promises to provide. This contract is enforced at build time by the compiler. If a class implements an interface, all the methods declared by that interface must appear in the implementing class for the class to compile successfully.

Declaring an interface

Explain the syntax for declaring an interface. Tell them that an interface in Java is defined as a reference type and is similar to a class except that it has only `final` and `static` variables, whereas the methods are `abstract`.

The following code snippet shows the declaration of the interface:

```
public interface Sample extends Interface1{
    static final int someInteger;
    public void someMethod();
}
```

Tell them that the method declaration does not have any braces and is terminated with a semicolon. Also, the body of the interface contains only abstract methods and no concrete method. Since all methods in an interface are implicitly abstract, the `abstract` keyword is not explicitly specified with the method signature.

Tell them the convention followed by Java for the interface names. It is written in CamelCase, that is, first letter of each word is capitalized. Also, the name of the interface describes an operation that a class can perform.

For example,

```
interface Enumerable
interface Comparable
```

Some programmers prefix the letter 'I' with the interface name to distinguish interfaces from classes. For example, `interface IComparable` and `interface Comparable`.

Consider the hierarchy of vehicles where `IVehicle` is the interface that declares the methods which the implementing classes such as `TwoWheeler`, `FourWheeler`, and so on can define.

Explain the figure to create a new interface in NetBeans IDE, right-click the package name and select **New → Java Interface** as mentioned in slide 9.

A dialog box appears where the user must provide a name for the interface and then, click **OK**. This will create an interface with the specified name.

Explain the code snippet that defines the interface, **IVehicle** as mentioned in slides 10 and 11.

Implementing the Interface

To declare a class that implements an interface, you include an `implements` clause in the class declaration. Your class can implement more than one interface, so the `implements` keyword is followed by a comma-separated list of the interfaces implemented by the class.

Explain the syntax to implement an interface in the Java class:

```
class <class-name> implements <Interface1>,...  
{  
    // class members  
    // overridden abstract methods of the interface(s)  
}
```

Explain the code snippet defines the class **TwoWheeler** that implements the **IVehicle** interface mentioned in slides 12 to 16. The code snippet defines the class **TwoWheeler** that implements the **IVehicle** interface. The class consists of some instance variables and a constructor to initialize the variables. Notice, that the class implements all the methods of the interface **IVehicle**. The `displayDetails()` method is used to display the details of the specified vehicle.

The `main()` method is defined in another class **TestVehicle**. Within the `main()` method, the number of command line arguments specified is verified and accordingly the object of class **TwoWheeler** is created. Next, the object is used to invoke the various methods of the class.

In-Class Question:

After you finish explaining purpose of interfaces, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



How the interface names are written in Java?

Answer:

The first letter of each word in the interface name is capitalized.

Tips:

1. Interfaces are unaffected by changes to specific classes or the class hierarchy as a whole, because they are declared independently of classes.
2. Java also supports the creation of empty interfaces. Empty interface does not have declared methods. Empty interfaces can be used as types and to mark classes without requiring any particular method implementations. For seeing an example of a useful empty interface, see `java.io.Serializable`.

Slides 17 to 25

Let us understand implementing multiple interfaces.

Implementing Multiple Interfaces 1-9



- ◆ Java does not support multiple inheritance of classes but allows implementing multiple interfaces to simulate multiple inheritance.
 - ◆ To implement multiple interfaces, write the interface names after the implements keyword separated by a comma.
 - ◆ For example,
- ```
public class Sample implements Interface1, Interface2{}
```

- ◆ Following code snippet defines the interface **IManufacturer**:

```
package session11;
public interface IManufacturer {
 /**
 * Abstract method to add contact details
 * @param detail a String variable storing manufacturer detail
 * @return void
 */
 public void addContact(String detail);

 /**
 * Abstract method to call the manufacturer
 */
}
```

© Aptech Ltd.

Interfaces and Nested Classes/Session 11

17

## Implementing Multiple Interfaces 2-9



```
* @param phone a String variable storing phone number
* @return void
*/
public void callManufacturer(String phone);

/**
 * Abstract method to make payment
 * @param amount a float variable storing amount
 * @return void
*/
public void makePayment(float amount);
```

- ◆ The modified class, **TwoWheeler** implementing both the **IVehicle** and **IManufacturer** interfaces is displayed in the following code snippet:

```
package session11;
class TwoWheeler implements IVehicle, IManufacturer {
 String ID; // variable to store vehicle ID
 String type; // variable to store vehicle type
```

© Aptech Ltd.

Interfaces and Nested Classes/Session 11

18

## Implementing Multiple Interfaces 3-9



```
/*
 * Parameterized constructor to initialize values based on user input
 *
 * @param ID a String variable storing vehicle ID
 * @param type a String variable storing vehicle type
 */
public TwoWheeler(String ID, String type) {
 this.ID = ID;
 this.type = type;
}

/**
 * Overridden method, starts a vehicle
 *
 * @return void
 */
@Override
public void start() {
 System.out.println("Starting the "+ type);
}
```

© Aptech Ltd.

Interfaces and Nested Classes/Session 11

19

**Implementing Multiple Interfaces 4-9**

```
/*
 * Overridden method, accelerates a vehicle
 * @param speed an integer storing the speed
 * @return void
 */
@Override
public void accelerate(int speed) {
 System.out.println("Accelerating at speed:" + speed+ " kmph");
}

/*
 * Overridden method, applies brake to a vehicle
 *
 * @return void
 */
@Override
public void brake() {
 System.out.println("Applying brakes...");
}
```

© Aptech Ltd.

Interfaces and Nested Classes/Session 11

20

**Implementing Multiple Interfaces 5-9**

```
/*
 * Overridden method, stops a vehicle
 *
 * @return void
 */
@Override
public void stop() {
 System.out.println("Stopping the "+ type);
}

/*
 * Displays vehicle details
 *
 * @return void
 */
public void displayDetails()
{
 System.out.println("Vehicle No.: "+ STATEID+ " "+ ID);
 System.out.println("Vehicle Type.: "+ type);
}
```

© Aptech Ltd.

Interfaces and Nested Classes/Session 11

21

**Implementing Multiple Interfaces 6-9**

```
// Implement the IManufacturer interface methods

/*
 * Overridden method, adds manufacturer details
 * @param detail a String variable storing manufacturer detail
 * @return void
 */
@Override
public void addContact(String detail) {
 System.out.println("Manufacturer: "+detail);
}

/*
 * Overridden method, calls the manufacturer
 * @param phone a String variable storing phone number
 * @return void
 */
@Override
public void callManufacturer(String phone) {
 System.out.println("Calling Manufacturer @: "+phone);
}
```

© Aptech Ltd.

Interfaces and Nested Classes/Session 11

22

### Implementing Multiple Interfaces 7-9



```
/*
 * Overridden method, makes payment
 * @param amount a String variable storing the amount
 * @return void
 */
@Override
public void makePayment(float amount) {
 System.out.println("Payable Amount: $" + amount);
}

/**
 * Define the class TestVehicle.java
 *
 */
public class TestVehicle {
 /**
 * @param args the command line arguments
 */
 public static void main(String[] args) {
 // Verify number of command line arguments
 if(args.length==6) {
 ...
 }
 }
}
```

© Aptech Ltd.

Interfaces and Nested Classes/Session 11

23

### Implementing Multiple Interfaces 8-9



```
// Instantiate the class
TwoWheeler objBike = new TwoWheeler(args[0], args[1]);
objBike.displayDetails();
objBike.start();
objBike.accelerate(Integer.parseInt(args[2]));
objBike.brake();
objBike.stop();
objBike.addContact(args[3]);
objBike.callManufacturer(args[4]);
objBike.makePayment(Float.parseFloat(args[5]));
}
else{
 // Display an error message
 System.out.println("Usage: java TwoWheeler <ID> <Type> <Speed>
<Manufacturer> <Phone> <Amount>");
}
}
```

© Aptech Ltd.

Interfaces and Nested Classes/Session 11

24

### Implementing Multiple Interfaces 9-9



- ◆ The class **TwoWheeler** now implements both the interfaces; **IVehicle** and **IManufacturer** as well as all the methods of both the interfaces.
- ◆ Following figure shows the output of the modified code.
- ◆ The user passes **CS-2737 Bike 80 BN-Bikes 808-283-2828 300** as command line arguments.

**Output - Session11 (run)**

```
run:
Vehicle No.: LA-09 CS-2737
Vehicle Type.: Bike
Starting the Bike
Accelerating at speed:80 kmph
Applying brakes...
Stopping the Bike
Manufacturer: BN-Bikes
Calling Manufacturer #: 080-283-2828
Payable Amount: $300.0
```

- ◆ The interface **IManufacturer** can also be implemented by other classes such as **FourWheeler**, **Furniture**, **Jewelry**, and so on, that require manufacturer information.

© Aptech Ltd.

Interfaces and Nested Classes/Session 11

25

Using slides 17 to 25, explain implementing multiple interfaces.

Java does not support multiple inheritance of classes but allows implementing multiple interfaces to simulate multiple inheritance. To implement multiple interfaces, write the interface names after the **implements** keyword separated by a comma.

For example,

```
public class Sample implements Interface1, Interface2{
 . . .
}
```

Explain the code snippet that defines the interface **IManufacturer** mentioned in slides 17 and 18. Tell them that the interface **IManufacturer** declares three abstract methods namely, **addContact(String)**, **callManufacturer(String)**, and **makePayment(float)** that must be defined by the implementing classes.

Then tell them that the class class, **TwoWheeler** is modified for implementing both the **IVehicle** and **IManufacturer** interfaces is displayed in the code snippet mentioned in slides 18 to 25. The class **TwoWheeler** now implements all the methods of both the interfaces; **IVehicle** and **IManufacturer**. Otherwise, the class has to be declared as abstract.

### Slides 26 to 28

Let us understand the concept of abstraction.

#### Understanding the Concept of Abstraction 1-3



Abstraction is defined as the process of hiding the unnecessary details and revealing only the essential features of an object to the user.

Abstraction is a concept that is used by classes that consist of attributes and methods that perform operations on these attributes.

Abstraction can also be achieved through composition. For example, a class Vehicle is composed of an engine, tyres, ignition key, and many other components.

In Java, abstract classes and interfaces are used to implement the concept of abstraction.

To use an abstract class or interface one needs to extend or implement abstract methods with concrete behavior.

Abstraction is used to define an object based on its attributes, functionality, and interface.

© Aptech Ltd.      Interfaces and Nested Classes/Session 11      26

#### Understanding the Concept of Abstraction 2-3



- The differences between an abstract class and an interface are listed in the following table:

| Abstract Class                                                                                                                         | Interface                                                              |
|----------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------|
| An abstract class can have abstract as well as concrete methods that are methods with a body.                                          | An interface can have only abstract methods.                           |
| An abstract class may have non-final variables.                                                                                        | Variables declared in an interface are implicitly final.               |
| An abstract class may have members with different access specifiers such as <code>private</code> , <code>protected</code> , and so on. | Members of an interface are <code>public</code> by default.            |
| An abstract class is inherited using the <code>extends</code> keyword.                                                                 | An interface is implemented using the <code>implements</code> keyword. |
| An abstract class can inherit from another class and implement multiple interfaces.                                                    | An interface can extend from one or more interfaces.                   |

© Aptech Ltd.      Interfaces and Nested Classes/Session 11      27

**Understanding the Concept of Abstraction 3-3**

- ◆ Some of the differences between Abstraction and Encapsulation are as follows:
  - Abstraction refers to bringing out the behavior from 'How exactly' it is implemented whereas Encapsulation refers to hiding details of implementation from the outside world so as to ensure that any change to a class does not affect the dependent classes.
  - Abstraction is implemented using an interface in an abstract class whereas Encapsulation is implemented using private, default or package-private, and protected access modifier.
  - Encapsulation is also referred to as data hiding.
  - The basis of the design principle 'programming for interface than implementation' is abstraction and that of 'encapsulate whatever changes' is encapsulation.

© Aptech Ltd.      Interfaces and Nested Classes/Session 11      28

Using slides 26 to 28, explain the concept of abstraction.

Abstraction is defined as the process of hiding the unnecessary details and revealing only the essential features of an object to the user. Abstraction is a concept that is used by classes that consist of attributes and methods that perform operations on these attributes.

Abstraction can also be achieved through composition. For example, a class **Vehicle** is composed of an engine, tyres, ignition key, and many other components.

To use an abstract class or interface one needs to extend or implement abstract methods with concrete behavior.

Explain the differences between an abstract class and an interface are listed in the table mentioned in slide 27. Tell them that an abstract class is just a superclass which enables other classes to inherit from it so that each subclass can share a common design.

#### Tips:

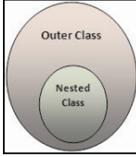
Programmers like to use abstract classes to reduce dependency on a range of specific subclass types. To include abstract method in a class, you also have to declare the class as abstract.

**Slides 29 and 30**

Let us understand the concept of nested class.

### Nested Class 1-2

- Java allows defining a class within another class.
- Such a class is called a nested class as shown in the following figure:



- Following code snippet defines a nested class:

```
class Outer{
 ...
 class Nested{
 ...
 }
}
```

- The class **Outer** is the external enclosing class and the class **Nested** is the class defined within the class **Outer**.

© Aptech Ltd. Interfaces and Nested Classes/Session 11 29

### Nested Class 2-2

- Nested classes are classified as static and non-static.
- Nested classes that are declared **static** are simply termed as **static nested classes** whereas non-static nested classes are termed as **inner classes**.
- This has been demonstrated in the following code snippet:

```
class Outer{
 ...
 static class StaticNested{
 ...
 }
 class Inner{
 ...
 }
}
```

- The class **StaticNested** is a nested class that has been declared as **static** whereas the non-static nested class, **Inner**, is declared without the keyword **static**.
- A nested class is a member of its enclosing class.
- Non-static nested classes or inner classes can access the members of the enclosing class even when they are declared as private.
- Static nested classes cannot access any other member of the enclosing class.

© Aptech Ltd. Interfaces and Nested Classes/Session 11 30

Using slides 29 and 30, explain the concept of nested class.

Java allows defining a class within another class. Such a class is called a nested class.

Tell them that the nested classes are classified as static and non-static. Nested classes that are declared static are simply termed as static nested classes whereas non-static nested classes are termed as inner classes. This has been demonstrated in the following code snippet mentioned in slide 30.

Non-static nested classes have access to other members of the enclosing class, even if they are declared private. Static nested classes do not have access to other members of the enclosing class.

**Tips:**

A static nested class interacts with the instance members of its outer class just like any other top-level class. In effect, a static nested class is behaviorally a top-level class that has been nested in another top-level class for packaging convenience.

**Slide 31**

Let us understand benefits of using nested class.

**Benefits of Using Nested Class**

- The reasons for introducing this advantageous feature of defining nested class in Java are as follows:

**Creates logical grouping of classes**

- If a class is of use to only one class, then it can be embedded within that class and the two classes can be kept together.
- In other words, it helps in grouping the related functionality together.
- Nesting of such 'helper classes' helps to make the package more efficient and streamlined.

**Increases encapsulation**

- In case of two top level classes such as class A and B, when B wants access to members of A that are `private`, class B can be nested within class A so that B can access the members declared as `private`.
- Also, this will hide class B from the outside world.
- Thus, it helps to access all the members of the top-level enclosing class even if they are declared as `private`.

**Increased readability and maintainability of code**

- Nesting of small classes within larger top-level classes helps to place the code closer to where it will be used.

© Aptech Ltd. | Interfaces and Nested Classes/Session 11 | 31

Using slide 31, explain benefits of using nested class.

Explain that nested classes can be used for the following reasons:

➤ **Allows logical grouping of classes**

If a class works as a helper class to another class, then it is logical to embed the second class within the first and keep them together.

➤ **Increases encapsulation**

If class **A** needs access to private members of class **B**, then class **A** can be declared as nested class of class **B**. As a result, class **A** can access all the private members of Class **B** and at the same time class **A** will be hidden from the outside world.

➤ **More maintainable code**

Nesting small classes within top-level classes places the code closer to where it is used. This makes it easier to maintain the code.

**Tips:**

A nested class is a member of its enclosing class.

**Slide 32**

Let us understand types of nested classes.

**Types of Nested Classes**

- The different types of nested classes are as follows:

|                                                        |
|--------------------------------------------------------|
| <b>Member classes or<br/>non-static nested classes</b> |
| <b>Local classes</b>                                   |
| <b>Anonymous classes</b>                               |
| <b>Static Nested classes</b>                           |

© Aptech Ltd.      Interfaces and Nested Classes/Session 11      32

Using slide 32, explain types of nested classes.

The different types of nested classes are as follows:

- Member classes or non-static nested classes
- Local classes
- Anonymous classes
- Static Nested classes

**Slides 33 to 37**

Let us understand member classes.

**Member Classes 1-5**

- A member class is a non-static inner class.
- It is declared as a member of the outer or enclosing class.
- The member class cannot have static modifier since it is associated with instances of the outer class.
- An inner class can directly access all members that is, fields and methods of the outer class including the private ones.
- However, the outer class cannot access the members of the inner class directly even if they are declared as public.
- This is because members of an inner class are declared within the scope of inner class.
- An inner class can be declared as public, private, protected, abstract, or final.
- Instances of an inner class exist within an instance of the outer class.
- To instantiate an inner class, one must create an instance of the outer class.

© Aptech Ltd.      Interfaces and Nested Classes/Session 11      33

**Member Classes 2-5**

- One can access the inner class object within the outer class object using the statement defined in the following code snippet:

```
// accessing inner class using outer class object
Outer.Inner objInner = objOuter.new Inner();
```

- Following code snippet describes an example of non-static inner class:

```
package session1;
class Server {
 String port; // variable to store port number

 /**
 * Connects to specified server
 *
 * @param IP a String variable storing IP address of server
 * @param port a String variable storing port number of server
 * @return void
 */
 public void connectServer(String IP, String port) {
 System.out.println("Connecting to Server at:" + IP + ":" + port);
 }
}
```

© Aptech Ltd.

Interfaces and Nested Classes/Session 11

34

**Member Classes 3-5**

```
/*
 * Define an inner class
 *
 */
class IPAddress
{
 /**
 * Returns the IP address of a server
 *
 * @return String
 */
 String getIP() {
 return "101.232.20.12";
 }
}

/*
 * Define the class TestConnection.java
 *
 */
public class TestConnection {
```

© Aptech Ltd.

Interfaces and Nested Classes/Session 11

35

**Member Classes 4-5**

```
/*
 * @param args the command line arguments
 */
public static void main(String[] args){
 /**
 * @param args the command line arguments
 */
 public static void main(String[] args) {
 // Check the number of command line arguments
 if(args.length==1) {
 // Instantiate the outer class
 Server objServer1 = new Server();
 // Instantiate the inner class using outer class object
 Server.IAddress objIP = objServer1.new IPAddress();
 // Invoke the connectServer() method with the IP returned from
 // the getIP() method of the inner class
 objServer1.connectServer(objIP.getIP(),args[0]);
 }
 else {
 System.out.println("Usage: java Server <port-no>"); }
 }
}
```

© Aptech Ltd.

Interfaces and Nested Classes/Session 11

36

**Member Classes 5-5**

- ◆ The class `Server` is an outer class that consists of a variable port that represents the port at which the server will be connected.
- ◆ Also, the `connectServer(String, String)` method accepts the IP address and port number as a parameter.
- ◆ The inner class `IPAddress` consists of the `getIP()` method that returns the IP address of the server.
- ◆ Following figure shows the output of the code when user provides '8080' as the port number at command line:

```
Output - Session11 (run)
run:
Connecting to Server at:101.232.28.12:8080
BUILD SUCCESSFUL (total time: 1 second)
```

© Aptech Ltd.

Interfaces and Nested Classes/Session 11

37

Using slides 33 to 37, explain member classes.

Tell them that as with instance methods and variables, an inner class is associated with an instance of its enclosing class and has direct access to that object's methods and fields. Also, because an inner class is associated with an instance, it cannot define any static members itself.

A member class is a non-static inner class and is declared as a member of an outer or an enclosing class. It cannot have static modifiers because it is associated with instances. It can access all fields and methods of the outer class, but the reverse is not true. An outer class cannot access a member of an inner class, even if it is declared as `public` because members of an inner class are declared within the scope of inner class. A member class can be declared as `public`, `protected`, `private`, `abstract`, `final`, or `static`.

One can access the inner class object within the outer class object using the statement defined in the code snippet mentioned in slide 34.

Explain the code snippet that describes an example of non-static inner class mentioned in slides 34 to 37.

Member class are declared outside a function hence it is a member and not declared static. A public member class behaves similar to a nested top-level class with the difference being that member classes have access to the specific instance of the enclosing class.

**In-Class Question:**

After you finish explaining member classes, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



What is the condition to instantiate an inner class?

**Answer:**

To instantiate an inner class, one must create an instance of the outer class.

**Slides 38 to 43**

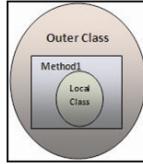
Let us understand local class.

### Local Class 1-6



- An inner class defined within a code block such as the body of a method, constructor, or an initializer, is termed as a local inner class.
- The scope of a local inner class is only within that particular block.
- Unlike an inner class, a local inner class is not a member of the outer class and therefore, it cannot have any access specifier.
- That is, it cannot use modifiers such as public, protected, private, or static.
- However, it can access all members of the outer class as well as final variables declared within the scope in which it is defined.

Following figure displays a local inner class:



© Aptech Ltd. Interfaces and Nested Classes/Session 11 38

### Local Class 2-6



- Local inner class has the following features:
  - It is associated with an instance of the enclosing class.
  - It can access any members, including private members, of the enclosing class.
  - It can access any local variables, method parameters, or exception parameters that are in the scope of the local method definition, provided that these are declared as final.
- Following code snippet demonstrates an example of local inner class.

```
package session11;
class Employee {

 /**
 * Evaluates employee status
 *
 * @param empID a String variable storing employee ID
 * @param empAge an integer variable storing employee age
 * @return void
 */
 public void evaluateStatus(String empID, int empAge){
 // local final variable
 final int age=40;
```

© Aptech Ltd. Interfaces and Nested Classes/Session 11 39

### Local Class 3-6



```
/**
 * Local inner class Rank
 *
 */
class Rank{

 /**
 * Returns the rank of an employee
 *
 * @param empID a String variable that stores the employee ID
 * @return char
 */
 public char getRank(String empID){
 System.out.println("Getting Rank of employee: "+ empID);
 // assuming that rank 'A' was returned from server
 return 'A';
 }

 // Check the specified age
 if(empAge>=age){
```

© Aptech Ltd. Interfaces and Nested Classes/Session 11 40

### Local Class 4-6

```
// Instantiate the Rank class
Rank objRank = new Rank();

// Retrieve the employee's rank
char rank = objRank.getRank(emplID);

// Verify the rank value
if(rank == 'A') {
 System.out.println("Employee rank is:" + rank);
 System.out.println("Status: Eligible for upgrade");
}
else{
 System.out.println("Status: Not Eligible for upgrade");
}
else{
 System.out.println("Status: Not Eligible for upgrade");
}
}
```

© Aptech Ltd.

Interfaces and Nested Classes/Session 11

41

### Local Class 5-6

```
/*
 * Define the class TestEmployee.java
 *
 */
public class TestEmployee {

 /**
 * @param args the command line arguments
 */
 public static void main(String[] args)
 {
 if(args.length==2) {
 // Object of outer class
 Employee objEmpl = new Employee();
 // Invoke the evaluateStatus() method
 objEmpl.evaluateStatus(args[0], Integer.parseInt(args[1]));
 }
 else{
 System.out.println("Usage: java Employee <Emp-Id> <Age>");
 }
 }
}
```

© Aptech Ltd.

Interfaces and Nested Classes/Session 11

42

### Local Class 6-6

- ◆ The class **Employee** is the outer class with a method named **evaluateStatus (String, int)**.
- ◆ The class **Rank** is a local inner class within the method.
- ◆ The **Rank** class consists of one method **getRank ()** that returns the rank of the specified employee id.
- ◆ If the age of the employee is greater than 40, the object of **Rank** class is created and the rank is retrieved.
- ◆ If the rank is equal to 'A' then, the employee is eligible for upgrade otherwise the employee is not eligible.
- ◆ Following figure shows the output of the code when user passes 'E001' as employee id and 50 for age:

Output - Session11 (run)  
run:  
Getting Rank of employee: E001  
Employee rank is:A  
Status: Eligible for upgrade

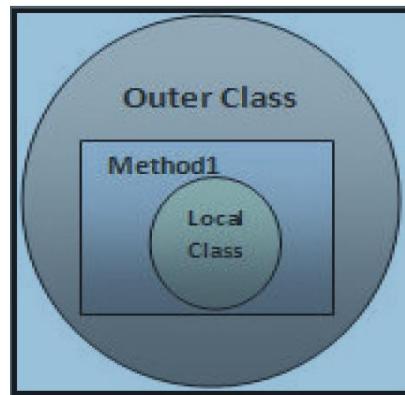
© Aptech Ltd.

Interfaces and Nested Classes/Session 11

43

Using slides 38 to 43, explain local class.

The following figure shows the local class:



Tell them that this is a special kind of inner class. Local classes are classes that are defined in a *block*, which is a group of zero or more statements between balanced braces. You typically find local classes defined in the body of a method.

A local class is declared within a method, constructor, or an initializer. In other words, a local class is declared within a block of code and is visible only within that particular block. It cannot have a `static` modifier. It has the ability to refer to local variables in the scope that defines them. Modifiers, such as `public`, `protected`, `private`, or `static` cannot be used in local classes.

Next, tell them that local classes have the following features:

- Local classes are associated with an instance of containing class, and can access any members, including `private` members, of the containing class.
- Local classes can access any local variables, method parameters, or exception parameters that are in the scope of the local method definition, provided that these are declared as `final`.

Explain the code snippet that demonstrates an example of local inner class mentioned in slides 39 to 43.

**Tips:**

1. You can define a local class not only in a method body, but also in a `for` loop, or an `if` clause.
2. Local classes are like local variables, specific to a block of code and their scope is only within the block of their declaration. The modifiers cannot be used with them as local classes are not members.
3. You cannot declare static initializers or member interfaces in a local class.

**Slides 44 to 49**

Let us understand anonymous class.

### Anonymous Class 1-6

An inner class declared without a name within a code block such as the body of a method is called an anonymous inner class.

- An anonymous class does not have a name associated, so it can be accessed only at the point where it is defined.
- It cannot use the `extends` and `implements` keywords nor can specify any access modifiers, such as `public`, `private`, `protected`, and `static`.
- It cannot define a constructor, `static` fields, methods, or classes.
- It cannot implement anonymous interfaces because an interface cannot be implemented without a name.
- Since, an anonymous class does not have a name, it cannot have a named constructor but it can have an instance initializer.
- Rules for accessing an anonymous class are the same as that of local inner class.

© Aptech Ltd.      Interfaces and Nested Classes/Session 11      44

### Anonymous Class 2-6

- Usually, anonymous class is an implementation of its super class or interface and contains the implementation of the methods.
- Anonymous inner classes have a scope limited to the outer class.
- They can access the internal or `private` members and methods of the outer class.
- Anonymous class is useful for controlled access to the internal details of another class.
- Also, it is useful when a user wants only one instance of a special class.
- Following figure displays an anonymous class:

© Aptech Ltd.      Interfaces and Nested Classes/Session 11      45

### Anonymous Class 3-6

- Following code snippet describes an example of anonymous class:

```

package session11;
class Authenticate {
 /**
 * Define an anonymous class
 */
 Account objAcc = new Account() {
 /**
 * Displays balance
 *
 * @param accNo a String variable storing balance
 * @return void
 */
 @Override
 public void displayBalance(String accNo) {
 System.out.println("Retrieving balance. Please wait...");
 }
 // Assume that the server returns 40000
 System.out.println("Balance of account number " + accNo.toUpperCase()
 + " is $40000");
 }
}

```

© Aptech Ltd.      Interfaces and Nested Classes/Session 11      46

**Anonymous Class 4-6**

```

); // End of anonymous class
}
/**
 * Define the Account class
 *
 */
class Account {
 /**
 * Displays balance
 *
 * @param accNo a String variable storing balance
 * @return void
 */
 public void displayBalance(String accNo) {
 }
}

/**
 * Define the TestAuthentication class
 *
 */
public class TestAuthentication {
}

```

© Aptech Ltd. Interfaces and Nested Classes/Session 11

47

**Anonymous Class 5-6**

```

/**
 * @param args the command line arguments
 */
public static void main(String[] args) {
 // Instantiate the Authenticate class
 Authenticate objUser = new Authenticate()
 // Check the number of command line arguments
 if (args.length == 3) {
 if (args[0].equals("admin") && args[1].equals("abc@123")) {
 // Invoke the displayBalance() method
 objUser.objAcc.displayBalance(args[2]);
 }
 else{
 System.out.println("Unauthorized user");
 }
 }
 else {
 System.out.println("Usage: java Authenticate <user-name> <password>
<account-no>");
 }
}

```

© Aptech Ltd. Interfaces and Nested Classes/Session 11

48

**Anonymous Class 6-6**

- ◆ The class **Authenticate** consists of an anonymous object of type **Account**.
- ◆ The **displayBalance(String)** method is used to retrieve and display the balance of the specified account number.
- ◆ Following figure shows the output of the code when user passes '**admin**', '**abc@123**', and '**AKDLE26152**', as arguments:

```

Output - Session11 (run)
run:
Retrieving balance. Please wait...
Balance of account number AKDLE26152 is ₹40000

```

© Aptech Ltd. Interfaces and Nested Classes/Session 11

49

Using slides 44 to 49, explain anonymous class.

Tell them that anonymous classes enable you to make your code more concise. They enable you to declare and instantiate a class at the same time. They are like local classes except that they do not have a name. Use them if you need to use a local class only once.

An anonymous class does not have a name. It is a type of local class. It also does not allow the use of extends, implements clauses, and access modifiers, such as public, private, protected, and static. Since, it does not have a name, it cannot define a constructor. If constructor declaration is needed, then a local class can be used. An anonymous class cannot define any static fields, methods, or classes.

Anonymous interfaces are not possible to implement because an interface cannot be implemented without a name.

Explain the code snippet that describes an example of anonymous class mentioned in slides 46 to 49. The class **Authenticate** consists of an anonymous object of type **Account**. The **displayBalance(String)** method is used to retrieve and display the balance of the specified account number.

The **main()** method is defined in the **TestAuthentication** class. Within **main()** method, an object of class **Authenticate** is created. The number of command line arguments is verified. If the number of arguments are correct, the username and password is verified and accordingly the object of anonymous class is used to invoke the **displayBalance()** method with account number as the argument.

#### Tips:

1. While local classes are class declarations, anonymous classes are an expression, which means that you define the class in another expression. The syntax of an anonymous class expression is like the invocation of a constructor, except that there is a class definition contained in a block of code.
2. Anonymous classes have no name. No constructor can be provided for them. Anonymous inner class is used when an object creation, one time single method invocation to be done and releasing the object at once. They can specify arguments to the constructor of the super class, but cannot have a constructor.
3. Some of the features of anonymous class are as follows:
  - An anonymous class has access to the members of its enclosing class.
  - An anonymous class cannot access local variables in its enclosing scope that are not declared as final or effectively final.
  - Like a nested class, a declaration of a type (such as a variable) in an anonymous class shadows any other declarations in the enclosing scope that have the same name.
  - Anonymous classes also have the same restrictions as local classes with respect to their members.
  - You cannot declare static initializers or member interfaces in an anonymous class.
  - An anonymous class can have static members provided that they are constant variables.
  - Note that you can declare the following in anonymous classes:
    - Fields
    - Extra methods (even if they do not implement any methods of the supertype)
    - Instance initializers
    - Local classes
4. However, you cannot declare constructors in an anonymous class.

**Slides 50 to 56**

Let us understand static nested class.

**Static Nested Class 1-7**

A static nested class is associated with the outer class just like variables and methods.

A static nested class cannot directly refer to instance variables or methods of the outer class just like static methods but can access only through an object reference.

A static nested class, by behavior, is a top-level class that has been nested in another top-level class for packaging convenience.

Static nested classes are accessed using the fully qualified class name, that is, `OuterClass.StaticNestedClass`.

A static nested class can have public, protected, private, default or packageprivate, final, and abstract access specifiers.

- Following code snippet demonstrates the use of static nested class:

```
package session11;
import java.util.Calendar;
class AtmMachine {
 /**
 * Define the static nested class
 */
}
```

© Aptech Ltd.

Interfaces and Nested Classes/Session 11

50

**Static Nested Class 2-7**

```
/*
static class BankDetails
{
 // Instantiate the Calendar class of java.util package
 static Calendar objNow = Calendar.getInstance();

 /**
 * Displays the bank and transaction details
 *
 * @return void
 */
 public static void printDetails(){
 System.out.println("State Bank of America");
 System.out.println("Branch: New York");
 System.out.println("Code: K3983LKSIE");

 // retrieving current date and time using Calendar object
 System.out.println("Date-Time: " + objNow.getTime());
 }
}
```

© Aptech Ltd.

Interfaces and Nested Classes/Session 11

51

**Static Nested Class 3-7**

```
/*
 * Displays balance
 * @param accNo a String variable that stores the account number
 * @return void
 */
public void displayBalance(String accNo) {
 // Assume that the server returns 20000
 System.out.println("Balance of account number " + accNo.toUpperCase()
 " is $200000");
}

/**
 * Define the TestATM class
 */
public class TestATM {
 /**
 * @param args the command line arguments
 */
 public static void main(String[] args) {
```

© Aptech Ltd.

Interfaces and Nested Classes/Session 11

52

### Static Nested Class 4-7



```

if(args.length ==1) { // verifying number of command line arguments
 // Instantiate the outer class
 AtmMachine objAtm = new AtmMachine();
 // Invoke the static nested class method using outer class object
 AtmMachine.BankDetails.printDetails();
 // Invoke the instance method of outer class
 objAtm.displayBalance(args[0]);
}
else{
 System.out.println("Usage: java AtmMachine <account-no>");
}
}

```

- Following figure shows the output of the code when user passes 'akd1e26152' as account number:

**Output - Session11 (run)**

```

run:
State Bank of America
Branch: New York
Code: K3983LKSIE
Date-Time:Thu Dec 27 11:50:38 GMT+05:30 2012
Balance of account number AKD1E26152 is ₹200000

```

© Aptech Ltd.

Interfaces and Nested Classes/Session11

53

### Static Nested Class 5-7



- Notice that the output of date and time shows the default format as specified in the implementation of the `getTime()` method.
- The format can be modified according to the user requirement using the `SimpleDateFormat` class of `java.text` package.
- `SimpleDateFormat` is a concrete class used to format and parse dates in a locale-specific manner.
- `SimpleDateFormat` class allows specifying user-defined patterns for date-time formatting.
- The modified `BankDetails` class using `SimpleDateFormat` class is displayed in the following code snippet:

```

import java.text.SimpleDateFormat;
import java.util.Calendar;
class AtmMachine {
 /**
 * Define the static nested class
 */
 static class BankDetails {
 // Instantiate the Calendar class of java.util package
 static Calendar objNow = Calendar.getInstance();
 }
}

```

© Aptech Ltd.

Interfaces and Nested Classes/Session11

54

### Static Nested Class 6-7



```

/**
 * Displays the bank and transaction details
 */
public static void printDetails(){
 System.out.println("State Bank of America");
 System.out.println("Branch: New York");
 System.out.println("Code: K3983LKSIE");

 // Format the output of date-time using SimpleDateFormat class
 SimpleDateFormat objFormat = new SimpleDateFormat("dd/MM/yyyy
 HH:mm:ss");

 // Retrieve the current date and time using Calendar object
 System.out.println("Date-Time: " +
 objFormat.format(objNow.getTime()));
}
...
...
}

```

© Aptech Ltd.

Interfaces and Nested Classes/Session11

55

### Static Nested Class 7-7



- ◆ The `SimpleDateFormat` class constructor takes the date pattern as a `String`.
- ◆ In the code, the pattern `dd/MM/yyyy HH:mm:ss` uses several symbols that are pattern letters recognized by the `SimpleDateFormat` class.
- ◆ Following table lists the pattern letters used in the code with their description:

| Pattern Letter | Description          |
|----------------|----------------------|
| d              | Day of the month     |
| M              | Month of the year    |
| y              | Year                 |
| H              | Hour of a day (0-23) |
| m              | Minute of an hour    |
| s              | Second of a minute   |

- ◆ Following figure shows the output of the modified code:

```

Output - Session11 (run)
run:
State Bank of America
Branch: New York
Code: K3983LKSIE
Date-Time:27/12/2012 11:51:27
Balance of account number ARDLE26152 is $200000

```

- ◆ The date and time are now displayed in the specified format that is more understandable to the user.

© Aptech Ltd.

Interfaces and Nested Classes/Session 11

56

Using slides 50 to 56, explain static nested class.

A static nested class is associated with the outer class just like variables and methods. A static nested class cannot directly refer to instance variables or methods of the outer class just like static methods but can access only through an object reference.

A static nested class, by behavior, is a top-level class that has been nested in another top-level class for packaging convenience. Static nested classes are accessed using the fully qualified class name, that is, `OuterClass.StaticNestedClass`.

Static nested classes can have `public`, `private`, `protected`, `package`, `final`, and `abstract` access specifiers. Private and protected static nested classes are used to implement the internal features of a class or a class hierarchy. On the other hand, public static nested classes are often used to group classes together or give access to private static variables and methods to a group of classes.

Explain the code snippet demonstrates the use of static nested class mentioned in slides 50 to 53. The class `AtmMachine` consists of a static nested class named `BankDetails`. The static nested class creates an object of the `Calendar` class of `java.util` package. The `Calendar` class consists of built-in methods to set or retrieve the system date and time. The `printDetails()` method is used to print the bank details along with the current date and time using the `getTime()` method of `Calendar` class.

The `main()` method is defined in the `TestATM` class. Within `main()` method, the number of command line arguments is verified and accordingly an object of class `AtmMachine` is created. Next, the static method `printDetails()` of the nested class `BankDetails` is invoked and accessed directly using the class name of the outer class `AtmMachine`. Lastly, the object `objAtm` of outer class is used to invoke `displayBalance()` method of the outer class with account number as the argument.

The modified `BankDetails` class using `SimpleDateFormat` class is displayed in the code snippet mentioned in slides 54 to 56.

**Tips:**

A nested class is a member of its enclosing class. Non-static nested classes (inner classes) have access to other members of the enclosing class, even if they are declared `private`. A static nested class interacts with the instance members of its outer class (and other classes) just like any other top-level class. In effect, a static nested class is behaviorally a top-level class that has been nested in another top-level class for packaging convenience.

**In-Class Question:**

After you finish explaining Static Nested Class, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



How the static nested class can be accessed?

**Answer:**

Static nested classes are accessed using the fully qualified class name.

**Slide 57**

Let us summarize the session.

| Summary                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |  |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--|
| <ul style="list-style-type: none"> <li>◆ An interface in Java is a contract that specifies the standards to be followed by the types that implement it.</li> <li>◆ To implement multiple interfaces, write the interfaces after the <code>implements</code> keyword separated by a comma.</li> <li>◆ Abstraction, in Java, is defined as the process of hiding the unnecessary details and revealing only the necessary details of an object.</li> <li>◆ Java allows defining a class within another class. Such a class is called a nested class.</li> <li>◆ A member class is a non-static inner class. It is declared as a member of the outer or enclosing class.</li> <li>◆ An inner class defined within a code block such as the body of a method, constructor, or an <code>initializer</code>, is termed as a local inner class.</li> <li>◆ An inner class declared without a name within a code block such as the body of a method is called an anonymous inner class.</li> <li>◆ A static nested class cannot directly refer to instance variables or methods of the outer class just like static methods but only through an object reference.</li> </ul> |  |

© Aptech Ltd.      Interfaces and Nested Classes/Session 11      57

In slide 57, you will summarize the session. End the session with a brief summary of what has been taught in the session.

**11.3 Post Class Activities for Faculty**

You should familiarize yourself with the topics of the next session which is based on error handling in Java.

**Tips:**

You can also check the Articles/Blogs/Expert Videos uploaded on the Online Varsity site to gain additional information related to the topics covered in the next session. You can also connect to online tutors on the Online Varsity site to ask queries related to the sessions.

# Session 12 – Exceptions

## 12.1 Pre-Class Activities

Before you commence the session, you should familiarize yourself with the topics of this session in-depth.

Here, you can discuss the key points with the students that were covered in the previous session. Prepare a question or two which will help you to relate the current session objectives.

### 12.1.1 Objectives

By the end of this session, the learners will be able to:

- Describe exceptions
- Explain types of errors and exceptions
- Describe the Exception class
- Describe exception handling
- Explain try-catch block
- Explain finally block
- Explain execution flow of exceptions
- Explain guidelines to exception handling

### 12.1.2 Teaching Skills

To teach this session, you should be well-versed with concept of exception handling in Java. You should be aware with the types of error that occur during the program execution. You must be aware with the exception classes hierarchy supported by Java, exception handling mechanism such as try-catch and finally blocks.

You should teach the concepts in the theory class using the images provided. For teaching in the class, you are expected to use slides and LCD projectors.

#### Tips:

It is recommended that you test the understanding of the students by asking questions in between the class.

#### In-Class Activities:

Follow the order given here during In-Class activities.

### Overview of the Session:

Give the students the overview of the current session in the form of session objectives. Show the students slide 2 of the presentation.

| Objectives                                                                                                                                                                                                                                                                                                                                                                       |  |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"> <li>◆ Describe exceptions</li> <li>◆ Explain types of errors and exceptions</li> <li>◆ Describe the Exception class</li> <li>◆ Describe exception handling</li> <li>◆ Explain try-catch block</li> <li>◆ Explain finally block</li> <li>◆ Explain execution flow of exceptions</li> <li>◆ Explain guidelines to exception handling</li> </ul> |                                                                                     |
| <small>© Aptech Ltd.</small>                                                                                                                                                                                                                                                                                                                                                     | <small>Exceptions / Session 12</small>                                              |
| <small>2</small>                                                                                                                                                                                                                                                                                                                                                                 |                                                                                     |

Tell the students that this session explains them the concept of error handling in Java. The session explains the type of errors that can occur at runtime. The session also introduce them with the Exception class and its hierarchy in Java. The session also explain the mechanism to handle exceptions using try-catch and finally blocks.

Tell them that this session focuses on defining, creating, and handling of exceptions in Java programs.

They will also learn the execution flow of exceptions and guidelines to exception handling.

### 12.2 In-Class Explanations

#### Slide 3

Let us understand the causes of getting errors in application.

| Introduction                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |  |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"> <li>◆ Java is a very robust and efficient programming language.</li> <li>◆ Features such as classes, objects, inheritance, and so on make Java a strong, versatile, and secure language.</li> <li>◆ However, no matter how well a code is written, it is prone to failure or behaves erroneously in certain conditions.</li> <li>◆ These situations may be expected or unexpected.</li> <li>◆ In either case, the user would be nonplussed or confused with such unexpected behavior of code.</li> <li>◆ To avoid such a situation, Java provides the concept of exception handling using which, a programmer can display appropriate message to the user in case such unexpected behavior of code occurs.</li> </ul> |                                                                                       |
| <small>© Aptech Ltd.</small>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             | <small>Exceptions / Session 12</small>                                                |
| <small>3</small>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |                                                                                       |

Using slide 3, explain the causes of getting errors in application.

Java is a very robust and efficient programming language. Features such as classes, objects, inheritance, and so on make Java a strong, versatile, and secure language. However, no matter how well a code is written, it is prone to failure or behaves erroneously in certain conditions. This results in raising of expected or unexpected errors in the application. In either case, the user would be nonplussed or confused with such unexpected behavior of code. Tell them that the errors raised by the environment during application execution are known as exceptions.

To avoid such a situation, Java provides the concept of exception handling using which, a programmer can display appropriate message to the user in case such unexpected behavior of code occurs.

#### Tips:

In programs, exceptions can occur due to any of the following reasons:

##### ➤ **Programming errors**

These exceptions arise due to errors present in APIs, such as `NullPointerException`. The program using these APIs cannot do anything about these errors.

##### ➤ **Client code errors**

The client code attempts operations, such as reading content from a file without opening it. This will raise an exception. The exception will provide information about the cause of the error and is handled by the client code.

##### ➤ **Errors beyond the control of a program**

There are certain exceptions that are not expected to be caught by the client code, such as memory error or network connection failure error. These are errors which have been raised by the run-time environment.

## Slides 4 to 7

Let us understand exceptions.

The slide has a title bar 'Exceptions 1-4' and a coffee cup icon. A purple callout box defines an exception as an event or abnormal condition in a program occurring during execution of a program that leads to disruption of normal flow of program instructions. Below the callout, a bulleted list details reasons for exceptions:

- An exception can occur for different reasons such as:
  - when the user enters invalid data
  - a file that needs to be opened cannot be found
  - a network connection has been lost in the middle of communications
  - the JVM has run out of memory
- When an error occurs inside a method, it creates an exception object and passes it to the runtime system.
- This object holds information about the type of error and state of the program when the error occurred.

At the bottom, there are footer links: © Aptech Ltd., Exceptions/Session 12, and a page number 4.

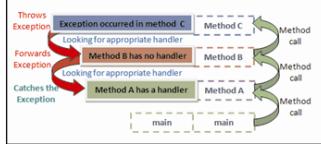
## Exceptions 2-4



The process of creating an exception object and passing it to the runtime system is termed as throwing an exception.

- After an exception is thrown by a method, the runtime system tries to find some code block to handle it.
  - The possible code blocks where the exception can be handled are a series of methods that were invoked in order to reach the method where the error has actually occurred.
- This list or series of methods is called the call stack. The stack trace shows the sequence of method invocations that led up to the exception.

- Following figure shows an example of method call stack:



- The figure shows the method call from main → Method A → Method B → Method C.

## Exceptions 3-4



When an exception occurs in method C, it throws the exception object to the runtime environment.

The runtime environment then searches the entire call stack for a method that consists of a code block that can handle the exception.

This block of code is called an exception handler.

The runtime environment first searches the method in which the error occurred.

If a handler is not found, it proceeds through the call stack in the reverse order in which the methods were invoked.

When an appropriate handler is found, the runtime environment passes the exception to the handler.

An appropriate exception handler is one that handles the same type of exception as the one thrown by the method.

In this case, the exception handler is said to 'catch' the exception.

- If while searching the call stack, the runtime environment fails to find an appropriate exception handler, the runtime environment will consequently terminate the program.

© Aptech Ltd.

Exceptions/Session 12

6

## Exceptions 4-4



- An exception is thrown for the following reasons:

→ A throw statement within a method was executed.

→ An abnormality in execution was detected by the JVM, such as:

- Violation of normal semantics of Java while evaluating an expression such as an integer divided by zero.
- Error occurring while linking, loading, or initializing part of the program that will throw an instance of a subclass of `LinkageError`.
- The JVM is prevented from executing the code due to an internal error or resource limitation that will throw an instance of a subclass of `VirtualMachineError`.

→ An asynchronous exception occurred.

- The use of exceptions to handle errors offers some advantages as follows:

Separate Error-Handling Code from Normal Code

Propagate Errors Higher Up in the Call Stack



Group the Similar Error Types

© Aptech Ltd.

Exceptions/Session 12

7

Using slides 4 to 7, explain Exceptions.

Tell the students that an exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions. When an error occurs within a method, the method creates an object and hands it off to the runtime system. The object, called an *exception object*, contains information about the error, including its type and the state of the program when the error occurred. The runtime system in Java is JVM which provides a default exception handler for handling

the runtime exceptions raised in an exception. Creating an exception object and handing it to the runtime system is called *throwing an exception*.

An exception can occur for different reasons such as:

- when the user enters invalid data
- a file that needs to be opened cannot be found
- a network connection has been lost in the middle of communications
- the JVM has run out of memory

The process of creating an exception object and passing it to the runtime system is termed as throwing an exception. After an exception is thrown by a method, the runtime system tries to find some code block to handle it.

The possible code blocks where the exception can be handled are a series of methods that were invoked in order to reach the method where the error has actually occurred.

This list or series of methods is called the call stack. The stack trace shows the sequence of method invocations that led up to the exception. Explain the figure shows an example of method call stack mentioned in slides 5 and 6. If while searching the call stack, the runtime environment fails to find an appropriate exception handler, the runtime environment will consequently terminate the program.

An abnormality in execution detected by the JVM, such as:

- Violation of normal semantics of Java while evaluating an expression such as an integer divided by zero.
- Error occurring while linking, loading, or initializing part of the program that will throw an instance of a subclass of `LinkageError`.
- The JVM is prevented from executing the code due to an internal error or resource limitation that will throw an instance of a subclass of `VirtualMachineError`.

Then explain the use of exceptions to handle errors that offer some advantages as follows:

- Separate Error-Handling Code from Normal Code
- Propagate Errors Higher Up in the Call Stack
- Group the Similar Error Types

#### Tips:

The components of the runtime system involved in handling the exception are as follows:

➤ **Call stack:**

When a method throws an exception, the runtime system tries to find an appropriate method to handle it starting from the method that caused the exception. In this process, the system invokes a set of methods. The list of invoked methods that is called by the runtime system to reach to the method where the exception occurred is known as the call stack.

➤ **Exception handler:**

The runtime system searches the call stack for a method that contains a piece of code that is able to handle the exception. This piece of code is called an exception handler.

➤ **Catching an exception:**

When an appropriate exception handler matching an exception object is found, then it is known as catching an exception.

## Slides 8 to 12

Let us understand types of errors and exceptions.

**Types of Errors and Exceptions 1-5**

- Java provides the following two types of exceptions:

**Checked Exceptions**

- These are exceptions that a well-written application must anticipate and provide methods to recover from.
- For example, suppose an application prompts the user to specify the name of a file to be opened and the user specifies the name of a nonexistent file.
- In such a case, the `java.io.FileNotFoundException` is thrown.
- However, a well-written program will have the code block to catch this exception and inform the user of the mistake by displaying an appropriate message.
- In Java, all exceptions are checked exceptions, except those indicated by `Error`, `RuntimeException`, and their subclasses.

© Aptech Ltd. Exceptions/Session 12 8

**Types of Errors and Exceptions 2-5**

**Unchecked Exceptions**

- The unchecked exceptions are as follows:

**Error**

- These are exceptions that are external to the application.
- The application usually cannot anticipate or recover from errors.
- For example, suppose the user specified correct file name for the file to be opened and the file exists on the system.
- However, the runtime fails to read the file due to some hardware or system malfunction.
- Such a condition of unsuccessful read throws the `java.io.IOException` exception.
- In this case, the application may catch this exception and display an appropriate message to the user or leave it to the program to print a stack trace and exit.
- Errors are exceptions generated by `Error` class and its subclasses.

© Aptech Ltd. Exceptions/Session 12 9

**Types of Errors and Exceptions 3-5**

**Runtime Exception**

- These exceptions are internal to the application and usually the application cannot anticipate or recover from such exceptions.
- These exceptions usually indicate programming errors, such as logical errors or improper use of an API.
- For example, suppose a user specified the file name of the file to be opened.
- However, due to some logical error a `null` is passed to the application, then the application will throw a `NullPointerException`.
- The application can choose to catch this exception and display appropriate message to the user or eliminate the error that caused the exception to occur.
- Runtime exceptions are indicated by `RuntimeException` class and its subclasses.

Errors and runtime exceptions are collectively known as unchecked exceptions.

In Java, `Object` class is the base class of the entire class hierarchy.

`Throwable` class is the base class of all the exception classes.

`Object` class is the base class of `Throwable`.

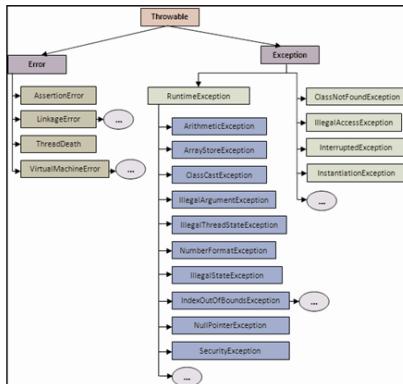
`Throwable` class has two direct subclasses namely, `Exception` and `Error`.

© Aptech Ltd. Exceptions/Session 12 10

### Types of Errors and Exceptions 4-5



- The `Throwable` class hierarchy is shown in the following figure:



© Aptech Ltd.

Exceptions/Session 12

11

### Types of Errors and Exceptions 5-5



- Following table lists some of the checked exceptions:

| Exception              | Description                                                      |
|------------------------|------------------------------------------------------------------|
| InstantiationException | Occurs upon an attempt to create instance of an abstract class.  |
| InterruptedException   | Occurs when a thread is interrupted.                             |
| NoSuchMethodException  | Occurs when JVM is unable to resolve which method to be invoked. |

- Following table lists some of the commonly observed unchecked exceptions:

| Exception                       | Description                                                                              |
|---------------------------------|------------------------------------------------------------------------------------------|
| ArithmaticException             | Indicates an arithmetic error condition.                                                 |
| ArrayIndexOutOfBoundsException  | Occurs if an array index is less than zero or greater than the actual size of the array. |
| IllegalArgumentException        | Occurs if method receives an illegal argument.                                           |
| NegativeArraySizeException      | Occurs if array size is less than zero.                                                  |
| NullPointerException            | Occurs on access to a null object member.                                                |
| NumberFormatException           | Occurs if unable to convert the string to a number.                                      |
| StringIndexOutOfBoundsException | Occurs if index is negative or greater than the size of the string.                      |

© Aptech Ltd.

Exceptions/Session 12

12

Using slides 8 to 12, explain types of errors and exceptions.

Java provides the following two types of exceptions:

#### Checked Exceptions

Checked exceptions are generated in situations that occur during the normal execution of a program. Some common examples of checked exceptions are: requesting for missing files, invalid user input, and network failures. These exceptions should be handled to avoid compile-time errors. If an exception occurs during the execution of a method, the method can handle the exception itself or throw the exception back to the calling method to denote that a problem has occurred. The calling method can again handle the exception itself or throw it to its calling method. This process can continue until the exception reaches the top of a thread and the thread is terminated. This is known as the **Call-Stack** mechanism. The main advantage of this practice is that a developer has the flexibility of putting the error handling code wherever to choose.

These types of exceptions are derived from the `Exception` class. The program either handles the checked exception or moves the exception further up the stack. It is checked during compilation.

### Unchecked Exceptions

Unchecked exceptions are generated in situations that are considered non-recoverable for a program. Common examples of these situations are attempting to access an element beyond the maximum length of an array. An application is not required to check for these kinds of exceptions. Runtime exceptions are also examples of unchecked exceptions. They generally arise from logical bugs. Unchecked exceptions that arise on account of environmental problems or errors are impossible to recover from and are known as Errors. Exhausting a program's allocated memory is a common example of an error.

These exceptions are derived from the `RuntimeException` class, which in turn is derived from the `Exception` class. The program need not handle unchecked exception. It is generated during the execution of the program.

The unchecked exceptions are of two types:

**Error:** These are exceptions that are external to the application. The application usually cannot anticipate or recover from errors. For example, suppose the user specified correct file name for the file to be opened and the file exists on the system. However, the runtime fails to read the file due to some hardware or system malfunction. Such a condition of unsuccessful read throws the `java.io.IOException` exception.

**Runtime Exception:** These exceptions are internal to the application and usually the application cannot anticipate or recover from such exceptions. These exceptions usually indicate programming errors, such as logical errors or improper use of an API. For example, suppose a user specified the file name of the file to be opened. However, due to some logical error a `null` is passed to the application, then the application will throw a `NullPointerException`. A runtime exception is an exception that occurs probably and can be avoided by the programmers. As opposed to checked exceptions, runtime exceptions are ignored at the time of compilation.

### Exception hierarchy

Explain the `Throwable` class hierarchy shown in the figure mentioned in slide 11. `Throwable` class is the base class of all the exception classes. `Throwable` class has two direct subclasses namely, `Exception`, and `Error`.

Explain the mentioned table that lists some of the checked exceptions mentioned in slide 12. Explain the mentioned table that lists some of the commonly observed unchecked exceptions mentioned in slide 12.

#### **In-Class Question:**

After you finish explaining types of errors and exceptions, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



What type of exceptions are those which are beyond the control of the user or the programmer and are external to the application?

#### **Answer:**

Error.

**Tips:**

JVM does not force you to handle unchecked exceptions as they are mostly generated at runtime due to programmatic errors.

**Slides 13 to 15**

Let us understand Exception class.

### Exception Class 1-3

The class `Exception` and its subclasses indicate conditions that an application might attempt to handle.

The `Exception` class and all its subclasses except `RuntimeException` and its subclasses, are checked exceptions.

- ◆ The checked exceptions must be declared in a method or constructor's `throws` clause if the method or constructor is liable to throw the exception during its execution and propagate it further in the call stack.
- ◆ Following code snippet displays the structure of the `Exception` class:

```
public class Exception extends Throwable
{
 ...
}
```

© Aptech Ltd. Exceptions/Session 12 13

### Exception Class 2-3

- ◆ Following table lists the constructors of `Exception` class:

| Exception Class Constructor                             | Description                                                                                                                                                                                                                                                                                                                                                           |
|---------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>Exception()</code>                                | Constructs a new exception with error message set to null.                                                                                                                                                                                                                                                                                                            |
| <code>Exception(String message)</code>                  | Constructs a new exception with error message set to the specified string <code>message</code> .                                                                                                                                                                                                                                                                      |
| <code>Exception(String message, Throwable cause)</code> | Constructs a new exception with error message set to the specified strings <code>message</code> and <code>cause</code> .                                                                                                                                                                                                                                              |
| <code>Exception(Throwable cause)</code>                 | Constructs a new exception with the specified cause. The error message is set as per the evaluation of <code>cause == null?null:cause.toString()</code> . That is, if <code>cause</code> is null, it will return null, else it will return the String representation of the message. The message is usually the class name and detail message of <code>cause</code> . |

© Aptech Ltd. Exceptions/Session 12 14

### Exception Class 3-3

- ◆ `Exception` class provides several methods to get the details of an exception.
- ◆ Following table lists some of the methods of `Exception` class:

| Exception Class Method                                   | Description                                                                                                                                                                                                                                                                                                                                  |
|----------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>public String getMessage()</code>                  | Returns the details about the exception that has occurred.                                                                                                                                                                                                                                                                                   |
| <code>public Throwable getCause()</code>                 | Returns the cause of the exception that is represented by a <code>Throwable</code> object.                                                                                                                                                                                                                                                   |
| <code>public String toString()</code>                    | If the <code>Throwable</code> object is created with a message string that is not null, it returns the result of <code>getMessage()</code> along with the name of the exception class concatenated to it. If the <code>Throwable</code> object is created with a null message string, it returns the name of the actual class of the object. |
| <code>public void printStackTrace()</code>               | Prints the result of the method, <code>toString()</code> and the stack trace to <code>System.out</code> , that is, the error output stream.                                                                                                                                                                                                  |
| <code>public StackTraceElement [] getStackTrace()</code> | Returns an array where each element contains a frame of the stack trace. The index 0 represents the method at the top of the call stack and the last element represents the method at the bottom of the call stack.                                                                                                                          |
| <code>public Throwable fillInStackTrace()</code>         | Fills the stack trace of this <code>Throwable</code> object with the current stack trace, adding to any previous information in the stack trace.                                                                                                                                                                                             |

© Aptech Ltd. Exceptions/Session 12 15

Using slides 13 to 15, explain Exception class.

The class `Exception` and its subclasses indicate conditions that an application might attempt to handle. The `Exception` class and all its subclasses except `RuntimeException` and its subclasses, are checked exceptions.

The checked exceptions must be declared in a method or constructor's throws clause if the method or constructor is liable to throw the exception during its execution and propagate it further in the call stack.

Explain the code snippet displays the structure of the `Exception` class mentioned in slide 13.

Explain the table that lists the constructors of `Exception` class mentioned in slide 14.

The `Exception` class provides several methods to get the details of an exception. Explain the table lists some of the methods of `Exception` class mentioned in slide 15.

In Java, exceptions are objects. Java defines several exception classes inside the standard package `java.lang`. The most general of these exceptions are subclasses of the standard type `RuntimeException`. Since `java.lang` is implicitly imported into all Java programs, most exceptions derived from `RuntimeException` are automatically available.

## Slides 16 and 17

Let us understand handling exceptions in Java.



### Handling Exceptions in Java 1-2

Any exception that a method is liable to throw is considered to be as much a part of that method's programming interface as its parameters and return value.

The code that calls a method must be aware about the exceptions that a method may throw.

This helps the caller to decide how to handle them if and when they occur.

More than one runtime exception can occur anywhere in a program.

Having to add code to handle runtime exceptions in every method declaration may reduce a program's clarity.

Thus, the compiler does not require that a user must catch or specify runtime exceptions, although it does not object to it either.

© Aptech Ltd. Exceptions/Session 12 15

**Handling Exceptions in Java 2-2**



- ◆ A common situation where a user can throw a `RuntimeException` is when the user calls a method incorrectly.
- ◆ For example, a method can check beforehand if one of its arguments is incorrectly specified as `null`.
- ◆ In that case, the method may throw a `NullPointerException`, which is an unchecked exception.
- ◆ Thus, if a client is capable of reasonably recovering from an exception, make it a checked exception.
- ◆ If a client cannot do anything to recover from the exception, make it an unchecked exception.

© Aptech Ltd. Exceptions/Session 12 17

Using slides 16 and 17, explain handling exceptions in java.

Any exception that a method is liable to throw is considered to be as much a part of that method's programming interface as its parameters and return value. The code that calls a method must be aware about the exceptions that a method may throw. This helps the caller to decide how to handle them if and when they occur.

A common situation where a user can throw a `RuntimeException` is when the user calls a method incorrectly. For example, a method can check beforehand if one of its arguments is incorrectly specified as `null`. In that case, the method may throw a `NullPointerException`, which is an unchecked exception.

Thus, if a client is capable of reasonably recovering from an exception, make it a checked exception. If a client cannot do anything to recover from the exception, make it an unchecked exception.

**Tips:**

An exception is handled by saving the current state of execution and switching the execution to a specific subroutine is known as exception handler.

**In-Class Question:**

After you finish explaining handling exceptions in java, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



When a runtime exception is thrown?

**Answer:**

When a user calls a method incorrectly.

## Slides 18 to 22

Let us understand try-catch block.

### try-catch Block 1-5



- The first step in creating an exception handler is to identify the code that may throw an exception and enclose it within the `try` block.
- The syntax for declaring a `try` block is as follows:

#### Syntax

```
try{
 // statement 1
 // statement 2
}
```

- The statements within the `try` block may throw an exception.
- Now, when the exception occurs, it is trapped by the `try` block and the runtime looks for a suitable handler to handle the exception.
- To handle the exception, the user must specify a `catch` block within the method that raised the exception or somewhere higher in the method call stack.

© Aptech Ltd.

Exceptions/Session12

18

### try-catch Block 2-5



- The syntax for declaring a `try-catch` block is as follows:

#### Syntax

```
try{
 // statements that may raise exception
 // statement 1
 // statement 2
}
catch(<exception-type> <object-name>){
 // handling exception
 // error message
}
```

where,

`exception-type`: Indicates the type of exception that can be handled.

`object-name`: Object representing the type of exception.

- The `catch` block handles exceptions derived from `Throwable` class.

© Aptech Ltd.

Exceptions/Session12

19

### try-catch Block 3-5



- Following code snippet demonstrates an example of `try` with a single `catch` block:

```
package session12;
class Mathematics {
 /**
 * Divides two integers
 * @param num1 an integer variable storing value of first number
 * @param num2 an integer variable storing value of second number
 * @return void
 */
 public void divide(int num1, int num2) {
 // Create the try block
 try {
 // Statement that can cause exception
 System.out.println("Division is: " + (num1/num2));
 }
 catch(ArithmaticException e){ //catch block for ArithmaticException
 // Display an error message to the user
 System.out.println("Error: " + e.getMessage());
 }
 // Rest of the method
 System.out.println("Method execution completed");
 }
}
```

© Aptech Ltd.

Exceptions/Session12

20

**try-catch Block 4-5**

```
/*
 * Define the TestMath.java class
 */
public class TestMath {
 /**
 * @param args the command line arguments
 */
 public static void main(String[] args)
 {
 // Check the number of command line arguments
 if(args.length==2) {
 // Instantiate the Mathematics class
 Mathematics objMath = new Mathematics();
 // Invoke the divide(int,int) method
 objMath.divide(Integer.parseInt(args[0]),
 Integer.parseInt(args[1]));
 }
 else {
 System.out.println("Usage: java Mathematics <number1> <number2>");
 }
 }
}
```

© Aptech Ltd. Exceptions/Session 12 21

**try-catch Block 5-5**

- It is clear that the statement `num1/num2` might raise an error if the user specifies zero for the denominator `num2`.
- Therefore, the statement is enclosed within the `try` block.
- Division being an arithmetic operation, the user can create an appropriate `catch` block with `ArithmaticException` class object.
- Within the `catch` block, the `ArithmaticException` class object `e` is used to invoke the `getMessage()` method that will print the detail about the error.
- Following figure shows the output of the program when user specifies 12 as numerator and 0 as denominator:

Output - Session12 (run)

run:  
Error: / by zero  
Method execution completed  
Back to Main

© Aptech Ltd. Exceptions/Session 12 22

Using slides 18 to 22, explain try-catch block.

The first step in creating an exception handler is to identify the code that may throw an exception and enclose it within the `try` block.

Explain the syntax for declaring a `try` block in Java. The statements within the `try` block may throw an exception. Now, when the exception occurs, it is trapped by the `try` block and the runtime looks for a suitable handler to handle the exception.

To handle the exception, the user must specify a `catch` block within the method that raised the exception or somewhere higher in the method call stack.

Explain the syntax for declaring a try-catch block in Java. The `catch` block handles exceptions derived from `Throwable` class.

Explain the code snippet demonstrates an example of `try` with a single `catch` block mentioned in slides 20 to 22. The class `Mathematics` consists of one method named `divide()` that accepts two integers as parameters and prints the value after division on the screen. However, it is clear that the statement `num1/num2` might raise an error if the user specifies zero for the denominator `num2`.

Therefore, the statement is enclosed within the `try` block. Division being an arithmetic operation, the user can create an appropriate `catch` block with `ArithmeticException` class object.

Within the `catch` block, the `ArithmeticException` class object `e` is used to invoke the `getMessage()` method that will print the detail about the error. The `main()` method is defined in the `TestMath` class. Within the `main()` method, the object of `Mathematics` class is created to invoke the `divide()` method with the parameters specified by the user at command line.

#### Tips:

A try and its catch statement form a unit. To handle a runtime error, simply enclose the code that you want to monitor inside a `try` block. Immediately following the `try` block, includes a `catch` clause that specifies the exception type that you wish to catch.

The scope of the `catch` clause is restricted to those statements specified by the immediately preceding `try` statement. The statements that are protected by `try` must be surrounded by curly braces. The goal of `catch` clauses should be to resolve the exceptional condition and then continue on as if the error had never happened.

#### Slides 23 and 24

Let us understand execution flow of exceptions.

**Execution Flow of Exceptions 1-2**

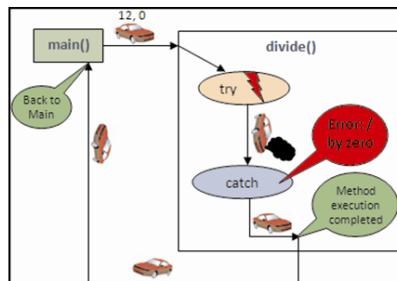
In the code, divide-by-zero exception occurs on execution of the statement `num1/num2`.  
 If `try-catch` block is not provided, any code after this statement is not executed as an exception object is automatically created.  
 Since, no `try-catch` block is present, JVM handles the exception, prints the stack trace, and the program is terminated.  
 Following figure shows the execution of the code when `try-catch` block is not provided:

© Aptech Ltd. Exceptions/Session 12 23

### Execution Flow of Exceptions 2-2



- ◆ When the try-catch block is provided, the divide-by-zero exception occurring in the code is handled by the try-catch block and an exception message is displayed.
- ◆ Also, the rest of the code gets executed normally.
- ◆ Following figure shows the execution of the code when try-catch block is provided:



© Aptech Ltd.

Exceptions/Session 12

24

Using slides 23 and 24, explain execution flow of exceptions.

In the code, divide-by-zero exception occurs on execution of the statement `num1/num2`. If try-catch block is not provided, any code after this statement is not executed as an exception object is automatically created. Since, no try-catch block is present, JVM handles the exception, prints the stack trace, and the program is terminated.

Explain the figure shows the execution of the code when try-catch block is not provided mentioned in slide 23.

When the try-catch block is provided, the divide-by-zero exception occurring in the code is handled by the try-catch block and an exception message is displayed. Also, the rest of the code gets executed normally.

Explain the figure that shows the execution of the code when try-catch block is provided mentioned in slide 24.

When the code present in the `try` block body does not throw any exceptions then, first the body of `try` block executes and then the code after `catch` block. Here, `catch` blocks never runs because no exceptions occur. However, when the exception occurs in `try` block the control is immediately transferred to the `catch` block.

## Slides 25 to 29

Let us understand 'throw' and 'throws' keywords.

### 'throw' and 'throws' Keywords 1-5

- ◆ Java provides the `throw` and `throws` keywords to explicitly raise an exception in the `main()` method.
- ◆ The `throw` keyword throws the exception in a method.
- ◆ The `throws` keyword indicates the exception that a method may throw.
- ◆ The `throw` clause requires an argument of `Throwable` instance and raises checked or unchecked exceptions in a method.
- ◆ Following code snippet demonstrates the modified class `Mathematics` now using `throw` and `throws` keywords for handling exceptions:

```
package session12;
class Mathematics {

 /**
 * Divides two integers, throws ArithmeticException
 * @param num1 an integer variable storing value of first number
 * @param num2 an integer variable storing value of second number
 * @return void
 */
 public void divide(int num1, int num2) throws ArithmeticException {
 if(num2==0) {
 // Throw the exception
 throw new ArithmeticException("/ by zero");
 }
 else {
 System.out.println("Division is: " + (num1/num2));
 }

 // Rest of the method
 System.out.println("Method execution completed");
 }
}
```

© Aptech Ltd.

Exceptions/Session12

25

### 'throw' and 'throws' Keywords 2-5

```
// Check the value of num2
if(num2==0) {
 // Throw the exception
 throw new ArithmeticException("/ by zero");
}
else {
 System.out.println("Division is: " + (num1/num2));
}

// Rest of the method
System.out.println("Method execution completed");
}

/**
 * Define the TestMath.java class
 */
public class TestMath {
 /**
 * @param args the command line arguments
 */
 public static void main(String[] args) {
 // Check the number of command line arguments
 if(args.length==2) {
 // Instantiate the Mathematics class
 Mathematics objMath = new Mathematics();
 try {
 // Invoke the divide(int,int) method
 objMath.divide(Integer.parseInt(args[0]),
 Integer.parseInt(args[1]));
 }
 catch(Arithmeti
```

© Aptech Ltd.

Exceptions/Session12

26

### 'throw' and 'throws' Keywords 3-5

```
// Check the number of command line arguments
if(args.length==2) {
 // Instantiate the Mathematics class
 Mathematics objMath = new Mathematics();
 try {
 // Invoke the divide(int,int) method
 objMath.divide(Integer.parseInt(args[0]),
 Integer.parseInt(args[1]));
 }
 catch(Arithmeti
```

© Aptech Ltd.

Exceptions/Session12

27

**'throw' and 'throws' Keywords 4-5**

- Within `divide (int, int)`, the code checks for the value of `num2`.
- If it is equal to zero, it creates an instance of `ArithmaticException` using the new keyword with the error message as an argument.
- The `throw` keyword throws the instance to the caller.
- Within the `main ()` method, an instance, `objMath`, is used to invoke the `divide (int, int)` method.
- However, this time, the code is written within the `try` block since `divide (int, int)` may throw an `ArithmaticException` that the `main ()` method will have to handle within its `catch` block.
- Following figure shows the output of the program when user specifies 12 as numerator and 0 as denominator:

The figure shows that upon execution of the code, the `if` condition becomes true and it throws the `ArithmaticException`.

© Aptech Ltd. Exceptions/Session 12 28

**'throw' and 'throws' Keywords 5-5**

- The control returns back to the caller, that is, the `main ()` method where it is finally handled.
- The `catch` block was executed and the result of `getMessage ()` is displayed to the user.
- Notice, that the remaining statement of the `divide (int, int)` method is not executed in this case.
- Following figure shows the execution of the code when `throw` and `throws` clauses are used:

© Aptech Ltd. Exceptions/Session 12 29

Using slides 25 to 29, explain 'throw' and 'throws' Keywords.

Tell the students that if a method does not handle a checked exception, the method must declare it using the `throws` keyword. The `throws` keyword appears at the end of a method's signature. Also, mention that one can throw an exception, either a newly instantiated one or an exception that you just caught, by using the `throw` keyword.

The `throws` keyword indicates that a method might throw the declared exception during its execution. On the other hand, the `throw` keyword is used to manually throw an exception after performing some validation in a program. After the `throw` keyword is encountered, the flow of execution is altered and the subsequent statements are not executed. The exception generated by the `throw` statement is then propagated to the previous calling method on the call stack.

The `throw` statement requires a single argument: a `Throwable` object that is an instance of the `Throwable` class.

Sometimes, a method can throw more than one exception. A comma-separated list of all exceptions thrown by a method is given with the method declaration. The `throws` keyword is used by the method to raise any checked or unchecked exception that it does not handle and enables the caller of the method to guard themselves against exception.

More than one exception can be listed with the `throws` clause and are separated by a comma. Except for `Error` or `RuntimeException` and their subclasses, the `throws` clause is necessary for all exceptions.

Explain the code snippet demonstrates the modified class **Mathematics** now using `throw` and `throws` keywords for handling exceptions mentioned in slides 25 to 28.

The control returns back to the caller, that is, the `main()` method where it is finally handled. The `catch` block was executed and the result of `getMessage()` is displayed to the user. Notice, that the remaining statement of the `divide(int, int)` method is not executed in this case.

Explain the figure shows the execution of the code when `throw` and `throws` clauses are used mentioned in slide 29.

#### Tips:

Never catch `Throwable` class. Because Java errors are also subclasses of the `Throwable`. Errors are irreversible conditions that cannot be handled by JVM alone.

#### Slides 30 to 33

Let us understand multiple 'catch' blocks.

### Multiple 'catch' Blocks 1-4

- The user can associate multiple exception handlers with a `try` block by providing more than one `catch` blocks directly after the `try` block.
- The syntax for declaring a `try` block with multiple `catch` blocks is as follows:

**Syntax**

```
try
{...}
catch (<exception-type> <object-name>)
{...}
catch (<exception-type> <object-name>)
{...}
```

- In this case, each `catch` block is an exception handler that handles a specific type of exception indicated by its argument `exception-type`.
- The runtime system invokes the handler in the call stack whose `exception-type` matches the type of the exception thrown.

© Aptech Ltd. Exceptions/Session 12 30

### Multiple 'catch' Blocks 2-4

- Following code snippet demonstrates the use of multiple `catch` blocks:

```
package session12;
public class Calculate {
 /**
 * @param args the command line arguments
 */
 public static void main(String[] args) {
 // Check the number of command line arguments
 if (args.length == 2){
 try {
 // Perform the division operation
 int num3 = Integer.parseInt(args[0]) / Integer.parseInt(args[1]);
 System.out.println("Division is: "+num3);
 }
 catch (ArithmaticException e) { // Catch the ArithmaticException
 System.out.println("Error: " + e.getMessage());
 }
 catch (NumberFormatException e){ // Catch the NumberFormatException
 System.out.println("Error: Required Integer found String:" +
 e.getMessage());
 }
 }
 }
}
```

© Aptech Ltd. Exceptions/Session 12 31

### Multiple 'catch' Blocks 3-4

```

 catch (Exception e) {
 System.out.println("Error: " + e.getMessage());
 }
 } else {
 System.out.println("Usage: java Calculate <number1> <number2>");
 }
}

```

- Following figure shows the output of the code when user specifies 12 and 0 as arguments:

Output - Session12 (run)  
run:  
Error: / by zero

- Following figure shows the output of the code when user specifies 12 and 'zero' as arguments:

Output - Session12 (run)  
run:  
Error: Required Integer found String:For input string: "zero"

© Aptech Ltd.

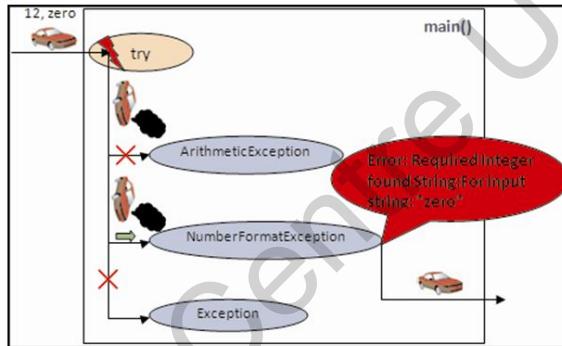
Exceptions/Session 12

32

### Multiple 'catch' Blocks 4-4



- Following figure shows the execution of the code when multiple catch blocks are used:



© Aptech Ltd.

Exceptions/Session 12

33

Using slides 30 to 33, explain multiple 'catch' blocks.

The user can associate multiple exception handlers with a try block by providing more than one catch blocks directly after the try block.

Explain the syntax for declaring a try block with multiple catch blocks. In this case, each catch block is an exception handler that handles a specific type of exception indicated by its argument exception-type. The runtime system invokes the handler in the call stack whose exception-type matches the type of the exception thrown.

Explain the code snippet demonstrates the use of multiple catch blocks mentioned in slides 31 and 32.

Then, explain the figure that shows the execution of the code when multiple catch blocks are used as mentioned in slide 33.

## Slides 34 to 38

Let us understand 'finally' block.

### 'finally' Block 1-5

Java provides the `finally` block to ensure execution of certain statements even when an exception occurs.

The `finally` block is always executed irrespective of whether or not an exception occurs in the `try` block.

This ensures that the cleanup code is not accidentally bypassed by a `return`, `break`, or `continue` statement.

The `finally` block is mainly used as a tool to prevent resource leaks.

- ◆ Tasks such as closing a file and network connection, closing input-output streams, or recovering resources, must be done in a `finally` block to ensure that a resource is recovered even if an exception occurs.
- ◆ However, if due to some reason, the JVM exits while executing the `try` or `catch` block, then the `finally` block may not execute.
- ◆ Similarly, if a thread executing the `try` or `catch` block gets interrupted or killed, the `finally` block may not execute even though the application continues to execute.

© Aptech Ltd.

Exceptions/Session 12

34

### 'finally' Block 2-5

- ◆ The syntax for declaring `try-catch` blocks with a `finally` block is as follows:

#### Syntax

```
try
{
 // statements that may raise exception
 // statement 1
 // statement 2
}
catch(<exception-type> <object-name>)
{
 // handling exception
 // error message
}
finally
{
 // clean-up code
 // statement 1
 // statement 2
}
```

© Aptech Ltd.

Exceptions/Session 12

35

### 'finally' Block 3-5

- ◆ Following code snippet demonstrates the modified class `Calculate` using the `finally` block:

```
package session12;
public class Calculate {
 /**
 * @param args the command line arguments
 */
 public static void main(String[] args) {
 // Check the number of command line arguments
 if (args.length == 2) {
 try {
 int num3 = Integer.parseInt(args[0]) / Integer.parseInt(args[1]);
 System.out.println("Division is: " + num3);
 }
 catch (ArithmaticException e) {
 System.out.println("Error: " + e.getMessage());
 }
 catch (NumberFormatException e) {
 System.out.println("Error: Required Integer found String:" +
 e.getMessage());
 }
 }
 }
}
```

© Aptech Ltd.

Exceptions/Session 12

36

**'finally' Block 4-5**

```

 catch (Exception e) {
 System.out.println("Error: " + e.getMessage());
 }
 finally {
 // Write the clean-up code for closing files, streams, and
 // network connections
 System.out.println("Executing Cleanup Code. Please Wait...");
 System.out.println("All resources closed.");
 }
 }
 else {
 System.out.println("Usage: java Calculate <number1> <number2>");
 }
}

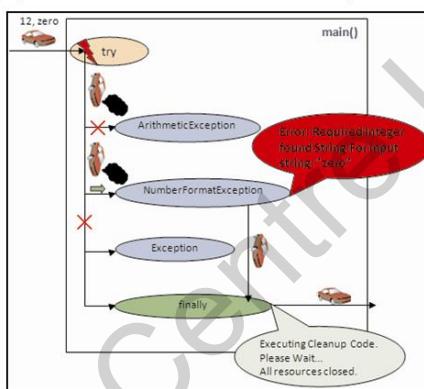
```

- Following figure shows the output of the code after using `finally` block when user passes `12` and '`zero`' as command line arguments:

**Output - Session12 (run)**

**'finally' Block 5-5**

- Following figure shows the execution of code when `finally` block is used:



Using slides 34 to 38, explain 'finally' block.

Java provides the `finally` block to ensure execution of certain statements even when an exception occurs. The `finally` block is always executed irrespective of whether or not an exception occurs in the `try` block. The runtime system always executes the statements within the `finally` block regardless of what happens within the `try` block. So it's the perfect place to perform cleanup.

Using a `finally` block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code. A `finally` block appears at the end of the `catch` blocks.

**Tips:**

The `finally` block is a key tool for preventing resource leaks. When closing a file or otherwise recovering resources, place the code in a `finally` block to ensure that resource is *always* recovered. Similarly, if a thread executing the `try` or `catch` block gets interrupted or killed, the `finally` block may not execute even though the application continues to execute.

Explain the syntax for declaring try-catch blocks with a `finally` block mentioned in slide 35.

Explain the code snippet demonstrates the modified class **Calculate** using the `finally` block mentioned in slide 36 and 37. Within the class **Calculate**, `finally` block is included after the last `catch` block. In this case, even if an exception occurs in the code, the `finally` block statements will be executed.

Explain the figure shows the execution of code when finally block is used as mentioned in slide 38.

### In-Class Question:

After you finish explaining finally block, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



What will happen when thread executing the try or catch block gets interrupted?

### Answer:

The `finally` block may not execute even though the application continues to execute.

### Tips:

For each `try` block there can be zero or more `catch` blocks, but only one `finally` block.

### Slides 39 to 41

Let us understand guidelines for handling exceptions.

**Guidelines for Handling Exceptions 1-3**

- The `try` statement must be followed by at least one `catch` or a `finally` block.
- Use the `throw` statement to throw an exception that a method does not handle by itself along with the `throws` clause in the method declaration.
- The `finally` block must be used to write clean up code.
- The `Exception` subclasses should be used when the caller of the method is expected to handle the exception.
  - The compiler will raise an error message if the caller does not handle the exception.
- Subclasses of `RuntimeException` class can be used to indicate programming errors such as `IllegalArgumentException`, `UnsupportedOperationException`, and so on.

© Aptech Ltd.      Exceptions/Session 12      39

### Guidelines for Handling Exceptions 2-3



**Avoid using the `java.lang.Exception` or `java.lang.Throwable` class to catch exceptions that cannot be handled.**

- Since, `Error` and `Exception` class can catch all exception of its subclasses including `RuntimeException`, the runtime behavior of such a code often becomes vague when global exception classes are caught.
- For example, one would not want to catch the `OutOfMemoryError`.
- How can one possible handle such an exception?

**Provide appropriate message along with the default message when an exception occurs.**

- All necessary data must be passed to the constructor of the exception class which can be helpful to understand and solve the problem.

**Try to handle the exception as near to the source code as possible.**

- If the caller can perform the corrective action, the condition must be rectified there itself.
- Propagating the exception further away from the source leads to difficulty in tracing the source of the exception.

© Aptech Ltd.

Exceptions/Session 12

40

### Guidelines for Handling Exceptions 3-3



**Exceptions should not be used to indicate normal branching conditions that may alter the flow of code invocation.**

- For example, a method that is designed to return a zero, one, or an object can be modified to return `null` instead of raising an exception when it does not return any of the specified values.
- However, a disconnected database is a critical situation for which no alternative can be provided.
- In such a case, exception must be raised.

**Repeated re-throwing of the same exception must be avoided as it may slow down programs that are known for frequently raising exceptions.**

**Avoid writing an empty catch block as it will not inform anything to the user and it gives the impression that the program failed for unknown reasons.**

© Aptech Ltd.

Exceptions/Session 12

41

Using slides 39 to 41, explain guidelines for handling exceptions.

- The `try` statement must be followed by at least one `catch` or a `finally` block.
- Use the `throw` statement to throw an exception that a method does not handle by itself along with the `throws` clause in the method declaration.
- The `finally` block must be used to write clean up code.
- The `Exception` subclasses should be used when the caller of the method is expected to handle the exception.
- The compiler will raise an error message if the caller does not handle the exception.
- Subclasses of `RuntimeException` class can be used to indicate programming errors such as `IllegalArgumentException`, `UnsupportedOperationException`, and so on.
- Avoid using the `java.lang.Exception` or `java.lang.Throwable` class to catch exceptions that cannot be handled. Since, `Error` and `Exception` class can catch all exception of its subclasses including `RuntimeException`, the runtime behavior of such a code often becomes vague when global exception classes are caught. For example, one would not want to catch the `OutOfMemoryError`. How can one possible handle such an exception?
- Provide appropriate message along with the default message when an exception occurs. All necessary data must be passed to the constructor of the exception class which can be helpful to understand and solve the problem.

- Try to handle the exception as near to the source code as possible. If the caller can perform the corrective action, the condition must be rectified there itself. Propagating the exception further away from the source leads to difficulty in tracing the source of the exception.
- Exceptions should not be used to indicate normal branching conditions that may alter the flow of code invocation. For example, a method that is designed to return a zero, one, or an object can be modified to return `null` instead of raising an exception when it does not return any of the specified values. However, a disconnected database is a critical situation for which no alternative can be provided. In such a case, exception must be raised.
- Repeated re-throwing of the same exception must be avoided as it may slow down programs that are known for frequently raising exceptions.
- Avoid writing an empty `catch` block as it will not inform anything to the user and it gives the impression that the program failed for unknown reasons.

### Slides 42 and 43

Let us summarize the session.

**Summary 1-2**



- An exception is an event or an abnormal condition in a program occurring during execution of a program that leads to disruption of the normal flow of the program instructions.
- The process of creating an exception object and passing it to the runtime system is termed as throwing an exception.
- An appropriate exception handler is one that handles the same type of exception as the one thrown by the method.
- Checked exceptions are exceptions that a well-written application must anticipate and provide methods to recover from.
- Errors are exceptions that are external to the application and the application usually cannot anticipate or recover from errors.
- Runtime Exceptions are exceptions that are internal to the application from which the application usually cannot anticipate or recover from.

© Aptech Ltd. Exceptions/Session 12 42

**Summary 2-2**



- `Throwable` class is the base class of all the exception classes and has two direct subclasses namely, `Exception` and `Error`.
- The try block is a block of code which might raise an exception and catch block is a block of code used to handle a particular type of exception.
- The user can associate multiple exception handlers with a try block by providing more than one catch blocks directly after the try block.
- Java provides the `throw` and `throws` keywords to explicitly raise an exception in the `main()` method.
- Java provides the finally block to ensure execution of cleanup code even when an exception occurs.

© Aptech Ltd. Exceptions/Session 12 43

In slides 42 and 43, you will summarize the session. End the session with a brief summary of what has been taught in the session.

### **12.3 Post Class Activities for Faculty**

The session ends the Fundamentals of Java course. Ask the students some questions related from all the topics which will help you to know the learnings taken by the students. You can solve the queries related to the sessions taught in the course.

**Tips:**

You can also check the Articles/Blogs/Expert Videos uploaded on the Online Varsity site to gain additional information related to the topics covered in the next session. You can also connect to online tutors on the Online Varsity site to ask queries related to the sessions.

# Session 13 – New Date and Time API

## 13.1 Pre-Class Activities

Before you commence the session, you should familiarize yourself with the topics of this session in-depth. Prepare a question or two that will be a key point to relate the current session objectives.

### 13.1.1 Objectives

By the end of this session, learners will be able to:

- Explain new classes of the Date and Time API in Java 8
- Explain Enum and Clock types
- Describe the role of time-zones in Java 8
- Explain support for backward compatibility in the new API

### 13.1.2 Teaching Skills

To teach this session, you should be well versed with the new Date and Time API introduced in Java 8.

You should teach the concepts in the theory class using the images provided. For teaching in the class, you are expected to use slides and LCD projectors.

#### Tips:

It is recommended that you test the understanding of the students by asking questions in between the class.

#### In-Class Activities

Follow the order given here during In-Class activities.

#### Overview of the Session

Give the students an overview of the current session in the form of session objectives. Read out the objectives on slide 2.

**Slide 2**

Objectives

- ❖ Explain new classes of the Date and Time API in Java 8
- ❖ Explain `Enum` and `clock` types
- ❖ Describe the role of time-zones in Java 8
- ❖ Explain support for backward compatibility in the new API

© Aptech Ltd.

Additional Programming in Java Session 1 / Slide 2

Show the slide 2 and give the students a brief overview of the current session in the form of session objectives. Tell the students that they will be introduced to the new Date-Time API in Java 8. Tell them that the session will cover enum and clock types and describe the role of time-zones in Java 8. Finally, the session will explain how the new API supports backward compatibility.

**13.2 In-Class Explanations****Slide 3**

Let us understand about the new Date-Time API in Java 8.

Introduction

New Date-Time API overcomes issues faced by earlier version of date and time library such as:

- Thread-safe issue
- Poor design
- Time-zone handling issue

© Aptech Ltd.

Additional Programming in Java Session 1 / Slide 3

Explain about the new Date-Time API in Java 8.

Introduce the new Date-Time API by saying that earlier versions of Java had drawbacks related to date and time such as `java.util.Date` class is not thread-safe and can lead to potential concurrency issues. The object of the `Date` class stores both time and date leading to confusion on the name of the class. Also, it does not store time zone information.

#### **Additional Information:**

Refer the following links for more information:

[http://www.javaworld.com/article/2078757/java-se/java-101-the-next-generation-it-s-time-for-a-change.html](http://www.javaworld.com/article/2078757/java-se/java-se-java-101-the-next-generation-it-s-time-for-a-change.html)

<http://stackoverflow.com/questions/24631909/differences-between-java-8-date-time-api-java-time-and-joda-time>

#### **Slide 4**

Let us look at the overview of the classes in the new Date-Time API.



Explain that:

`java.time` package contains the classes and types of the new Java 8 Date-Time API. All the classes are immutable and thread-safe and are based on ISO calendar system.

#### **Additional Information:**

Refer the following link for more information:

<https://dzone.com/articles/deeper-look-java-8-date-and>

## Slides 5 to 7

Let us understand the Clock class.

**Classes in New Date and Time API 2/29**

**Clock**

Clock class in Java 8 version is used to get the date and time using current time-zone

Clock can be used in the place of System.currentTimeMillis() and TimeZone.getDefault().

© Aptech Ltd. Additional Programming in Java - Session 1 / Slide 5

**Classes in New Date and Time API 3/29**

**Clock**

Following Code Snippet displays the instance of using clock. It represents how a clock can provide access to the current date and time using a time-zone

```
import java.time.*;
...
// Creates a new Clock instance based on UTC.
...
Clock defaultClock = Clock.systemUTC();
System.out.println("Clock : " + defaultClock);
...
// Creates a clock instance based on system clock zone
Clock defaultClock2 = Clock.systemDefaultZone();
System.out.println("Clock : " + defaultClock2);
```

© Aptech Ltd. Additional Programming in Java - Session 1 / Slide 6

**Classes in New Date and Time API 4/29**

**Clock**

Following Code Snippet displays how the given date can be verified against the clock object:

```
public class MyClass {
 private Clock clock;
 ...
 public void process(LocalDate eventDate) {
 if (eventDate.isBefore(LocalDate.now(clock))) {
 // logic
 }
 }
}
```

© Aptech Ltd. Additional Programming in Java - Session 1 / Slide 7

Explain about the `Clock` class in the Date-Time API.

Explain that:

Current date-time depends on the time-zone. For globalized applications, it is important to create date-time with proper time-zone. `java.time` package consists of two classes: `Zoned` class and `Local` class. `Zoned` class of `java.time` package deals with various time-zones. You can use the `Local` class of `java.time` package without any time-zone specification.

Explain about the code on slide 6 that shows example of using `Clock`.

Explain that:

`Clock.systemUTC()` method returns a clock for the current instant converting date and time using the Greenwich/UTC time zone.

`Clock.systemDefaultZone()` method returns the current instant of clock. It converts date and time using the default time-zone.

Explain about the code on slide 7 that shows how to verify a given date against the `Clock` object.

Explain that:

The code is based on dependency injection framework for `Clock` object.

`LocalDate` class has two methods: `isBefore()` and `isAfter()` to compare dates. In the given code, `isBefore()` method returns true if the given date comes before the date on which the method is called.

To check whether today's date is either the start or end date, you can write

```
if (!todayDate.isBefore(startDate) && !todayDate.isAfter(endDate))
```

#### **Additional Information:**

Refer the following links for more information:

<http://farenda.com/java/java-time-clock-fixed/>

<http://stackoverflow.com/questions/308683/how-can-i-get-the-current-date-and-time-in-utc-or-gmt-in-java>

## Slides 8 and 9

Let us learn about the Duration class.



**Duration Class**

Following Code Snippet shows the usage of `plusDays()` and `minusDays()` methods:

```
Duration present = ... // assume code is written to
// get a present duration
Duration samplePlusA = present.plusDays(3);
Duration sampleMinusA = present.minusDays(3);
```

Here, first line of code produces a `Duration` variable, `present` that will be used as the base of calculations. It is assumed that code to create the `Duration` object is added.

Code Snippet then produces two new `Duration` objects based on the `present` object. The second line generates a `Duration`, which is equivalent to `present` plus three days. The third line builds `Duration` that is equivalent to `present` minus three days.

© Aptech Ltd. Java™ Concurrency Programming In Java® 8 Edition © 2014

Explain using slides 8 and 9 that:

A `Duration` object represents the time interval between two instant objects. A `Duration` object is immutable therefore, once it is created, its values cannot be changed.

Following code shows how to find the duration between two instants:

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

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

```
Duration dur = Duration.between(one, two)
```

`Duration` object consists of two parts, nanoseconds part and seconds part. You can access nanoseconds and seconds part of duration using the `getNano()` and `getSeconds()` method respectively.

Explain about the code that shows the usage of `plusDays()` and `minusDays()` methods.

Explain that:

The syntax for `plusDays()` method is:

```
public Duration plusDays(long daysToAdd)
```

The code will return 3 days from the current date and 3 days before the current date. To get 10 days from current date, you can state:

```
LocalDate today = LocalDate.now();
System.out.println("10 days after today will be
"+today.plusDays(10));
```

Give some more examples to illustrate the methods.

## Slide 10

Let us look at the `Instant` class.

**Instant  
(java.time.Instant)**

**Instant class helps in time stamp creation.**

**Generating an Instant:**  
An instance of an `Instant` can be generated using one of the `Instant` class factory methods.

**Code Snippet shows an `Instant` object representing the exact moment of now, using method `Instant.now()`:**

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

**Instant Calculations:**  
Code Snippet displays the use of `Instant` in nanoseconds and milliseconds.

```
Instant sampleFuture = sampleNow.plusNanos(4);
// four nanoseconds in the future
Instant samplePast = sampleNow.minusNanos(4);
//four nanoseconds in the past
```

Using this slide, explain about `Instant` class and how to generate an instant. Explain the code that shows an `Instant` object representing the exact moment of now, using `Instant.now()` method. Then, explain the code that displays the use of `Instant` in nanoseconds and milliseconds.

Explain that:

`Instant` class denotes a specific moment in time. A value of the `Instant` class calculates time starting from the first second of January 1, 1970 in the format 1970-01-01T00:00:00Z. If you run the code on 14<sup>th</sup> October, 2016 you will get the output as 2016-10-14T10:11:26.838Z.

## Slides 11 to 15

Let us look at the LocalDate class.

**Classes in New Date and Time API 8/29**

LocalDate class is bundled with the `java.time` package.

**Creating a LocalDate:**  
LocalDate objects can be created using several approaches. The first approach is to get a LocalDate equivalent to the local date of today.

Code Snippet shows creating a LocalDate object using `now()`.

```
LocalDate sampleLocData =
LocalDate.now();
```

**Obtain a LocalDate:**  
To obtain a LocalDate, you can also create it from a specific year, month, and day information.

Code Snippet shows creating LocalDate using `of()`.

```
LocalDate sampleLocData =
LocalDate.of(2016, 07, 04);
```

© Aptech

Aptech / Programming Java - Day 10 / Slide 11

**Classes in New Date and Time API 9/29**

**LocalDate** Date information of a LocalDate object can be accessed using following methods:

```

graph TD
 LocalDate[LocalDate] --> getYear[getYear()]
 LocalDate --> getDayOfYear[getDayOfYear()]
 LocalDate --> getMonth[getMonth()]
 LocalDate --> getDayOfMonth[getDayOfMonth()]
 LocalDate --> getDayOfWeek[getDayOfWeek()]

```

© Aptech

Aptech / Programming Java - Day 10 / Slide 12

**Classes in New Date and Time API 10/29**

Following Code Snippet illustrates date information of a LocalDate.

```
int year = localDate.getYear();
int dayOfMonth = localDate.getDayOfMonth();
Month month = localDate.getMonth();
int dayOfYear = localDate.getDayOfYear();
DayOfWeek dayOfWeek = localDate.getDayOfWeek();
int monthValue = month.getValue();
```

Notice how `getMonth()` and `getDayOfWeek()` methods return an enum instead of an int. These enums can provide their data as int values by calling their `getValue()` methods.

© Aptech

Aptech / Programming Java - Day 10 / Slide 13

**LocalDate Calculations:**  
A set of date calculations can be achieved with the `LocalDate` class using one or more of following methods:

- `plusDays ()`
- `minusDays ()`
- `plusWeeks ()`
- `minusWeeks ()`
- `plusMonths ()`
- `minusMonths ()`
- `plusYears ()`
- `minusYears ()`

**LocalDate Calculation Method:**  
Code Snippet displays how a `LocalDate` calculation methods works.

```
LocalDate sampleLocDa = LocalDate.of(2016, 04, 30);
LocalDate sampleLocDaA = sampleLocDa.plusYears(4);
LocalDate sampleLocDaB = sampleLocDa.minusYears(4);
```

In the code, `sampleLocDa`, a new instance of `LocalDate`, is created using `of()` method. Then, the code builds a new `LocalDate` instance that represents the date four years later from the specified date. Finally, the code generates a new `LocalDate` instance that denotes the date four years earlier from the specified date.

Using slide 11, explain about `LocalDate` class and the method to create a `LocalDate` and obtain a `LocalDate`.

Explain that:

`LocalDate` class represents date in the format `yyyy-mm-dd`. Its `now()` method helps to get the current date. For example, use the given code to get the current date:

```
LocalDate curr = LocalDate.now();
System.out.println("Today is"+ curr);
}
```

List the methods as given on slide 12 that are used to access date information of a `LocalDate` object and explain about `LocalDate`.

Explain that:

`LocalDate` class is used to represent only date without the time. This class uses the clock of the default time-zone to show the date. You can obtain a particular date using the static method, `now()`. You can also create a date from another date using the `LocalDate.of()` method.

Explain the code on slide 13 that illustrates the date information of a `LocalDate`.

Explain that:

When you use `int` in the `LocalDate` object, you must remember that month, week, and so on are zero based. That is, the month January is 0 and December is 11. Similarly, Sunday is 0 and Saturday is 6.

Explain the different methods available in `LocalDate` class for date calculations given on slide 14.

Explain that:

You can obtain the date information using the different methods available in `LocalDate` class. `plus()` methods adds days, weeks, months, and years and `minus()` methods subtracts days, weeks, months, and years.

Explain the code on slide 15 that shows how a `LocalDate` calculation method works.

Explain that:

In the code, a new instance of `LocalDate` is created using `of()` method. Then, a new `LocalDate` instance shows a date four years later from the specified date. Finally, a new `LocalDate` instance is created to denote the date four years before the specified date.

## Slides 16 and 17

Let us understand the `LocalDateTime` class.

**Classes in New Date and Time API 13/29**

**LocalDateTime**  
Represents a local date and time without any time-zone data.  
Date-Time information of a `LocalDateTime` object can be accessed using `getValue()` method.  
Various date and time calculations can be performed on `LocalDateTime` object with plus or minus methods.

**Creating a `LocalDateTime` object based on a specific year, month, and day:**

```
LocalDateTime sampleLocDaTiB = LocalDateTime.of(2016, 05, 07, 12, 06, 16, 054);
```

Parameters passed to `of()` are `year`, `month`, `day (of month)`, `hours`, `minutes`, `seconds`, and `nanoseconds` respectively.

**Classes in New Date and Time API 14/29**

**Code Snippet illustrates how `LocalDateTime` calculation methods work:**

```
LocalDateTime sampleLocDaTi = LocalDateTime.now();
LocalDateTime sampleLocDaTiA = sampleLocDaTi.plusYears(4);
LocalDateTime sampleLocDaTiB = sampleLocDaTi.minusYears(4);
```

The code first creates a `LocalDateTime` instance `sampleLocDaTi` signifying the current moment.  
Then, the code creates a `LocalDateTime` object that denotes a date and time four years later.  
Finally, the code builds a `LocalDateTime` object that denotes a date and time four years prior.

Using slide 16, explain about `LocalDateTime` class.

Explain that:

`LocalDateTime` class is a combination of `LocalDate` and `LocalTime` classes of Date-Time API.

Code shows creating a `LocalDateTime` object based on a specific year, month, and day. Various parameters passed to `of()` are `year`, `month`, `day (of month)`, `hours`, `minutes`, `seconds`, and `nanoseconds` respectively.

Using slide 17, explain the code that shows how `LocalDateTime` calculation methods work.

Explain that:

In the code, a new instance of `LocalDateTime` is created signifying the current moment. Then, the code creates a `LocalDateTime` object that denotes a date and time four years later. Finally, the code builds a `LocalDateTime` object that denotes a date and time four years prior.

In the code, a new instance of `LocalDate` is created using `of()` method. Then, a new `LocalDate` instance shows a date four years later from the specified date. Finally, a new `LocalDate` instance is created to denote the date four years before the specified date.

### Slides 18 and 19

Let us explore the `LocalTime` class.

**Classes in New Date and Time API 15/29**

**LocalTime Class** LocalTime class in Date-Time API signifies exact time of day without any time-zone data.

**Creating a LocalTime Class:**

A LocalTime instance can be generated using several approaches. The foremost approach is to create a LocalTime instance that denotes the exact time of now. Code Snippet shows the `now()` method.

**Code Snippet:**

```
LocalTime sampleLocTiA = LocalTime.now();
```

Another approach to produce a LocalTime object is to create it from specific hours, minutes, seconds, and nanoseconds. Code Snippet displays the `of()` method.

**Code Snippet:**

```
LocalTime sampleLocTiB = LocalTime.of(12, 24, 33, 001235);
```

**Classes in New Date and Time API 16/29**

**LocalTime Calculation**

- LocalTime class consists of a set of methods that can perform local time calculations
- For example, `plusMinutes()` method adds minutes and `minusMinutes()` subtracts minutes from a given value in a calculation
- Plus or minus methods are in `LocalDateTime` object
- Code Snippet explains LocalTime calculations.

```
LocalTime sampleLocTi = LocalTime.of(12, 24, 33, 001235);
// current local time
LocalTime sampleLocTiFuture = sampleLocTi.plusHours(4) // future
LocalTime sampleLocTiPast = sampleLocTi.minusHours(4) // past
```

Using slide 18, explain about the `LocalTime` class and the method to create a `LocalTime` class. Explain the code that shows the `now()` method and then explain the code that shows how to use `of()` method.

**Explain that:**

LocalTime class in Date-Time API signifies exact time of day without any time-zone data. The foremost approach to generate LocalTime instance is to create a LocalTime instance that denotes the exact time of now. Another approach to produce a LocalTime object is to create it from specific hours, minutes, seconds, and nanoseconds.

### **Additional Information:**

Refer the following link for more information:

<http://tutorials.jenkov.com/java-date-time/localtime.html>

Using slide 19, explain about the LocalTime calculations and explain the code that explains LocalTime calculations.

**Explain that:**

LocalTime class consists of a set of methods that can perform local time calculations. For example, plusMinutes () method adds minutes and minusMinutes () subtracts minutes from a given value in a calculation. LocalDateTime object consists of the plus or minus methods.

### **Slide 20**

Let us understand the MonthDay class.

**MonthDay Class**

- ❖ MonthDay is an immutable Date-Time object that represents month as well as day-of-month
- ❖ Code Snippet depicts how MonthDay class can be used for checking recurring date-time events.

```
// Code to display Birthday wishes
LocalDate dateOfBirth = LocalDate.of(2006, 02, 24);
MonthDay bday = MonthDay.of(dateOfBirth.getMonth(), dateOfBirth.getDayOfMonth());
MonthDay currentMonthDay = MonthDay.from(today); // assume today is defined
if(currentMonthDay.equals(bday)){
 System.out.println("Colorful Joyful Birthday Buddy");
}
else{
 System.out.println("Nope, today is not your B'day");
}
```

© Aptech Limited Business Application Programming Java v-10 Edition / Slide 20

Explain about the MonthDay class and then explain the code that depicts how MonthDay class can be used for checking the recurring date-time events.

**Explain that:**

MonthDay class represents month as well as day-of-month. You can use this class to derive a birthday or anniversary day from a month and day object.

**Additional Information:**

Refer the following link for more information:

<http://docs.oracle.com/javase/8/docs/api/java/time/MonthDay.html>

**Slide 21**

Let us look at the OffsetDateTime class.

The slide has a light blue header bar with the text 'Classes in New Date and Time API 18/29'. Below this, there is a section titled 'OffsetDateTime Class' with three bullet points:
 

- OffsetDateTime is an immutable illustration of date and time with an offset.
- Code Snippet displays an example stating California is GMT or UTC–07:00 and to get a similar time-zone, static method ZoneOffset.of() can be used.
- After fetching the offset value, OffsetDateTime can be shaped by passing a LocalDateTime and an offset to it.

 A code snippet window contains the following Java code:
 

```
LocalDateTime datetime = LocalDateTime.of(2016, Month.FEBRUARY, 15, 18, 20);
// to display the result using Offset
ZoneOffset sampleoffset = ZoneOffset.of("-07:00");
OffsetDateTime date = OffsetDateTime.of(datetime, sampleoffset);
System.out.println("Sample display of Date and Time using time-zone offset : " + date);
```

 At the bottom left is the text 'ib Aptech' and at the bottom right is 'Business Programming Java 8 Edition / Slide 21'.

Using this slide, explain about the OffsetDateTime class.

Explain that:

OffsetDateTime is an immutable illustration of date and time with an offset. This class stores all date and time fields, to an accuracy of nanoseconds, as well as the offset from UTC/Greenwich.

Explain the code that displays an example stating California is GMT or UTC–07:00 and to get a similar time-zone, static method ZoneOffset.of() can be used.

Explain that:

After fetching the offset value, OffsetDateTime can be shaped by passing a LocalDateTime and an offset to it.

**Additional Information:**

Refer to the link <https://blog.tompawlak.org/java-8-conversion-new-date-time-api> to explain how to use toInstant() method with ZoneOffset information.

## Slide 22

Let us look at the OffsetTime class.

**OffsetTime Class**

- ◆ `OffsetTime` class is an immutable Date-Time object that denotes a time, frequently observed as hour-minute-second-offset
- ◆ Following Code Snippet shows the complete program to fetch the seconds using the `OffsetTime` class:

```
import java.time.OffsetTime; // Class to show the result by using
// OffsetTime class method
public class MinuteOffset {
 public static void main(String[] args) {
 OffsetTime d = OffsetTime.now();
 int e = d.getMinute();
 System.out.println("Minutes: " + e);
 }
}
```

**Output:**

Minutes: 49

Author: Sanjeev Singh, Author - Head Content / File: 22

Using this slide, explain about the `OffsetTime` class and the code that shows the complete program to fetch the seconds using the `OffsetTime` class.

Explain that:

`OffsetTime` class is an immutable Date-Time object. This class denotes a time with an offset from UTC/Greenwich time. Generally, the output is shown in the format hour-minute-second-offset. The output of the code will be Minutes: 49.

### Additional Information:

Refer the following link to view more code samples of `OffsetTime` class:

<http://www.programcreek.com/java-api-examples/index.php?api=java.time.OffsetTime>

## Slides 23 to 25

Let us learn about the Period class.

**Classes in New Date and Time API 20/29**

- ❖ Period (java.time.Period) represents an amount of time in terms of days, months, and years.
- ❖ Duration and Period are somewhat similar; however, the difference between the two can be seen in their approach towards Daylight Savings Time (DST) when they are added to ZonedDateTime

© Aptech Limited. Authorised Programming Java - Session 1 / Slide 20

**Classes in New Date and Time API 21/29**

Following Code Snippet displays an example to calculate the span of time from today until a birthday, assuming the birthday is on May 22<sup>nd</sup>:

```

import java.time.LocalDate; // Class to get the present day
import java.time.Month; // Class to get month related calculations
import java.time.Period; // Class to calculate the time period between two
// time instances
import java.time.temporal.ChronoUnit;
public class NextBDay {
 public static void main(String[] args) {
 LocalDate presentday = LocalDate.now();
 LocalDate bday = LocalDate.of(1983, Month.MAY, 22);
 LocalDate comingBday = bday.withYear(presentday.getYear());
 // To address the belated b'day celebration.
 if (comingBday.isBefore(presentday) || comingBday.isEqual(presentday))
 {
 comingBday = comingBday.plusYears(1);
 }
 Period waitA = Period.between(presentday, comingBday);
 long waitB = ChronoUnit.DAYS.between(presentday, comingBday);
 }
}

```

© Aptech Limited. Authorised Programming Java - Session 1 / Slide 21

**Classes in New Date and Time API 22/29**

```

System.out.println("Totally, I need to wait for " + waitA.getMonths() + " months, and " +
waitA.getDays() + " days to celebrate my next b'day. (" + waitA + " days in total); // to display the waiting time for b'day Bash
}
}

```

**Output:**  
Totally, I need to wait for 0 months and 22 days to celebrate my next b'day. (22 days in total)

© Aptech Limited. Authorised Programming Java - Session 1 / Slide 22

Using slide 23, explain about the `Period` class.

Explain that:

`Period` class allows you to show an amount of time in the format days, months, and years. The difference between `Duration` and `Period` is duration adds an accurate number of seconds. We know that duration of one day is always exactly 24 hours. However, a `Period` adds a conceptual day. `Period` tries to balance with the local time.

Using slide 24, explain the code that displays an example to calculate the span of time from today until a birthday, assuming the birthday is on May 22<sup>nd</sup>. The code shows the usage of `Period` class. Note that the line `java.time.temporal.ChronoUnit` shows that the code uses `ChronoUnit` enumeration.

Using slide 25, explain the output of the code. Note that the number of days will differ based on the current date that is, the calculated days will be current date minus May 22, 1983.

## Slide 26

Let us learn about the `Year` class.

**Year Class**

- ❖ A `Year` (`java.time.Year`) object is an immutable Date-Time object that denotes a year.
- ❖ Following Code Snippet displays the calculations using `Year` class.

```
import java.time.Year; // Class to use Year values in calculations
public class SampleYear {
 public static void main(String[] args) {
 System.out.println("The Present Year(): " + Year.now());
 System.out.println("The year 2002 is a leap year: " +
 Year.isLeap(2002)); // to display whether the year 2002 is a leap
 // year or not
 System.out.println("The year 2012 is a leap year: " +
 Year.isLeap(2012)); // to display whether the year 2012 is a leap year or not
 }
}
```

**Output:**

```
The Present Year(): 2016
The year 2002 is a leap year: false
The year 2012 is a leap year: true
```

Explain the `Year` class and the code that displays the calculations using `Year` class.

A `Year` object is an immutable Date-Time object. It is used to denote a year. This class does not store or represent a specific month, day, time, or time-zone.

The `Year.isLeap()` method checks whether the specified year is leap or not. It returns true if the year represented is a leap year.

**Additional Information:**

Refer the following links to get more information on Year class:

<https://docs.oracle.com/javase/8/docs/api/java/time/Year.html>

<http://www.concretewebpage.com/java/jdk-8/java-8-time-api-example-period-year-yearmonth-zoneddatetime>

**Slides 27 and 28**

Let us explore the YearMonth class.

**Classes in New Date and Time API 24/29**

**YearMonth**

**YearMonth** (`java.time.YearMonth`) is a stable Date-Time object that denotes the combination of year and month. This class does not store or denote a day, time, or time-zone. For example, the value 'November 2011' can be stored in a `YearMonth`.

**YearMonth** can be used to denote things such as credit card expiry, Fixed Deposit maturity date, Stock Futures, Stock options expiry dates, or determining if the year is a leap year or not.

© Aptech Limited. Business Application Programming Java - Day 10 / Slide 27

**Classes in New Date and Time API 25/29**

**YearMonth**

Following Code Snippet shows the `YearMonth` calculations:

```
import java.time.YearMonth; // to use the Year and Month info
public class YearMonth {
 public static void main(String[] args) {
 System.out.println("The Present Year Month: " + YearMonth.now());
 // To display present year and month
 System.out.println("Month alone: " + YearMonth.parse("2016-04").getMonthValue()); // To display only the month value
 System.out.println("Year alone: " + YearMonth.parse("2016-04").getYear()); // To display the year value alone
 System.out.println("This year is a Leap year: " + YearMonth.parse("2016-04").isLeapYear()); // leap year check
 }
}
```

**Output:**

The Present Year Month: 2016-05  
Month alone: 4  
Year alone: 2016  
This year is a Leap year: true

© Aptech Limited. Business Application Programming Java - Day 10 / Slide 28

Explain the YearMonth class with the help of slides 27 and 28.

YearMonth class combines and displays year and month. You can use this class to calculate the number of days in the current month. `lengthOfMonth()` method returns the number

of days in the current `YearMonth` object. `YearMonth` class can also be used to check the number of days in the month of February that is, 28 or 29 days.

Using slide 28, explain the code that shows the `YearMonth` calculations.

Explain that:

The method `YearMonth.now()` displays the current year and month. `getMonthValue()` method displays only the month suppressing the year and `getYear()` method displays only the year suppressing the month. `isLeapYear()` method returns true if the given year is a leap year.

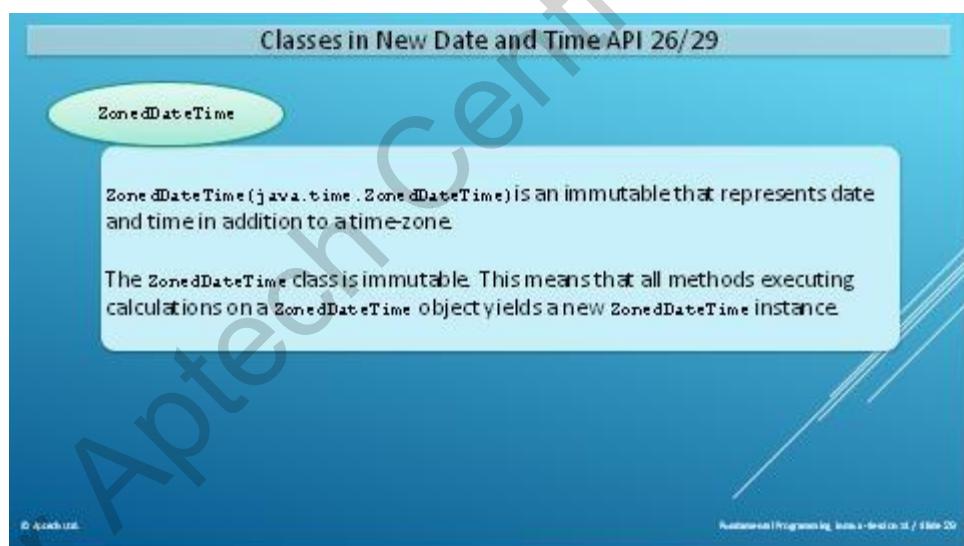
#### Additional Information:

Refer the following link for more information:

<http://www.concretewebpage.com/java/jdk-8/java-8-time-api-example-period-year-yearmonth-zoneddatetime>

#### Slides 29 and 30

Let us see the `ZonedDateTime` class.



**ZonedDateTime**

Following example depicts the usage of methods to get year, month, day, hour, minute, seconds, and zone offset:

```
import java.time.ZonedDateTime; // to access ZonedDateTime
public class ZoneDT { //Class ZoneDT refers to ZonedDateTime
public static void main(String[] args) {
 System.out.println(ZonedDateTime.now());
 ZonedDateTime sampleZDT = ZonedDateTime.parse("2016-04-03T01:15:30+08:00[Asia/Singapore]");
 System.out.println("Present day of the year:" +sampleZDT.getDayOfYear());
 System.out.println("Present year:" +sampleZDT.getYear());
}
}
```

**Output:**  
2016-05-06T06:03:51.787+08:00[Etc/UTC]  
Present day of the year: 94  
Present year: 2016

Aptech | Programming Java - Day 10 / 10

Using slide 29, explain about the `ZonedDateTime` class.

Explain that:

`ZonedDateTime` class represents date and time with zone information. This class stores the date and time fields to the precision of nanosecond values along with time-zone information.

Explain the code given on slide 30 that depicts the usage of methods to get year, month, day, hour, minute, seconds, and zone offset.

Explain that:

`ZonedDateTime.now()` method displays the current date and time with zone information. `Parse()` method converts the string to Date. `ZonedDateTime` class manages the conversion from the local time-line to the instant time-line. The difference between the two time-lines is the offset from UTC/Greenwich, represented by a `ZoneOffset`.

#### Additional Information:

Refer the following link for more information:

<http://www.concretepage.com/java/jdk-8/java-8-time-api-example-period-year-month-zoneddatetime>

## Slides 31 and 32

Let us understand ZoneId and ZoneOffset.

**Classes in New Date and Time API 28/29**

**ZoneId** is used to recognize rules used to convert between an Instant and a LocalDateTime.

The two different ID types are as follows:

- Fixed offsets
- Geographical regions

**Classes in New Date and Time API 29/29**

A time-zone offset is the quantity of time that a time-zone differs from Greenwich/UTC.

For example, Berlin is two hours ahead of Greenwich/UTC in Spring and four hours ahead during Autumn.

The ZoneId instance for Berlin will reference two ZoneOffset instances - a +02:00 instance for Spring and a +04:00 instance for Autumn.

Code Snippet illustrates the usage of this class

```
ZoneOffset sampleOffset = ZoneOffset.of("+05:00");
```

Explain the ZoneID class using slides 31 and 32.

Explain that:

ZoneId contains the zone rules that defines the time zone of a location. You can use the class when you have to obtain the date and time of specific time zone. ZoneOffset represents the period of time that denotes the difference between Greenwich time and a time-zone.

Using slide 32, explain the ZoneOffset class and explain the code that shows the usage of the ZoneOffset class.

Explain that:

A time-zone offset is the quantity of time that a time-zone differs from Greenwich/UTC. The time zone ID is unique. An offset ID begins with UTC, GMT, + or -, is a single letter.

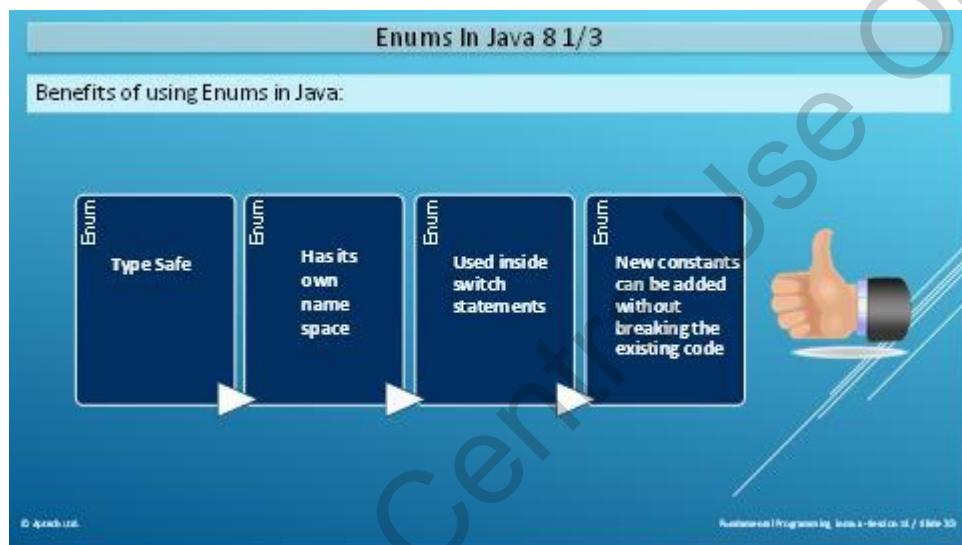
For example, 'Z', '+03:00', '-06:00', 'UTC+04' and 'GMT-2' are all valid offset IDs. However, +ABC and 'D' are invalid ID's even if they meet the criteria.

### **Additional Information:**

You can use the following link to compare time of different zones:  
<http://www.java2s.com/Tutorials/Java/java.time/ZonId/index.htm>

### **Slide 33**

Let us see the benefits of using enums.



Using this slide, explain about Enums in Java. An enumeration or enum helps to denote the fixed number of well-known values in Java. It can contain constants, methods. For example, to define days of week, one can use:

```
public enum week{
 Sunday,
 Monday,
 Tuesday,
 Wednesday,
 Thursday,
 Friday,
 Saturday}
```

### **Additional Information:**

Refer the following links for more information on Enums:

<https://docs.oracle.com/javase/8/docs/api/java/lang/Enum.html>  
<http://tutorials.jenkov.com/java/enums.html>

## Slides 34 and 35

Let us understand ChronoUnit.

**Enums In Java 8 2/3**

ChronoUnit enumeration is used in following Code Snippet:

```

import java.time.LocalDate;
import java.time.temporal.ChronoUnit;
public class EnumsInCalculation {
 public static void main(String args){
 EnumsInCalculation.java8main = new EnumsInCalculation();
 java8main.mainChromoUnits();
 }
 public void mainChromoUnits(){
 // To display the current date
 LocalDate today = LocalDate.now();
 System.out.println("Current date: " + today);
 // To display the result 2 weeks addition to the current
 date
 LocalDate nextWeek = today.plus(2, ChronoUnit.WEEKS);
 System.out.println("After 2 weeks: " + nextWeek);
 // To display the result 2 months addition to the current
 date
 LocalDate nextMonth = today.plus(2, ChronoUnit.MONTHS);
 System.out.println("After 2 months: " + nextMonth);
 // To display the result 2 years addition to the current
 date
 }
}

```

Output:

```

Current date: 2016-04-07
After 2 weeks: 2016-04-21
After 2 months: 2016-06-07
After 2 years: 2018-04-07
Date after twenty year: 2036-04-07

```

**Enums In Java 8 3/3**

```

date
LocalDate nextYear = today.plus(2, ChronoUnit.YEARS);
System.out.println("After 2 years: " + nextYear);
// To display the result 20 years addition to the current
date
LocalDate nextDecade = today.plus(2, ChronoUnit.DECADES);
System.out.println("Date after twenty years: " + nextDecade);
}

```

Explain the code that shows the usage of ChronoUnit enumeration.

Explain that:

The Date-Time API in Java SE 8 supports several new and useful enumerations such as ChronoUnit enumeration to represent the day, month. These units are especially helpful while designing multiple calendar systems.

Continue explaining the code on slide 35 that shows the usage of ChronoUnit enumeration.

Explain that:

`LocalDate.now()` method helps to obtain the current date. `today.plus(2, ChronoUnit.WEEKS)` method gives the date 2 weeks from the current date. `today.plus(2, ChronoUnit.MONTHS)` method calculates the date 2 months from current date, `today.plus(2, ChronoUnit.YEARS)` method calculates 2 years from

current date and finally, `today.plus(2, ChronoUnit.DECADES)` method calculates 2 decades from current date.

### Additional Information:

Refer the following links to get more information:

<https://docs.oracle.com/javase/8/docs/api/java/time/temporal/ChronoUnit.html>  
[https://www.tutorialspoint.com/java8/java8\\_chronounits.htm](https://www.tutorialspoint.com/java8/java8_chronounits.htm)

### Slides 36 to 38

Let us explore temporal adjusters.

**Temporal Adjusters 1/3**

Temporal Adjuster acts as a key tool in modifying the Temporal Object.

TemporalAdjuster is a functional interface that uses `adjustInto(Temporal)` method to return a copy of Temporal object with unchanged field value.

A Temporal Adjuster can be used to perform complicated date math that is popular in business applications.

for example, it can be used to find first Thursday of the month or next Tuesday.

`java.time.temporal`      Date-Time objects      `FirstDayOfMonth()`

**Temporal Adjusters 2/3**

Following Code Snippet shows how to find the first day of a month using a specified date:

```

import java.time.LocalDate;
import java.time.temporal.TemporalAdjusters;
import java.time.DayOfWeek;
public class TemporalAdj {
 public static void main(String args[]) {
 TemporalAdj TemporalAdj = new TemporalAdj();
 TemporalAdj.sampleAdj();
 }
 public void sampleAdj() {
 // To display the current date
 LocalDate sampledateA = LocalDate.now();
 System.out.println("Current date: " + sampledateA);
 // To display the next Wednesday from current date
 LocalDate nextWednesday =
 sampledateA.with(TemporalAdjusters.next(DayOfWeek.WEDNESDAY));
 }
}

```

```

System.out.println("Next Wednesday on : " + nextWednesday);
// To display the second Sunday of next month
LocalDate firstInYear =
LocalDate.of(sampledateA.getYear(), sampledateA.getMonth(), 1);
LocalDate secondSunday = firstInYear.with(TemporalAdjusters.nextOrSame(DayOfWeek.SUNDAY)).with(TemporalAdjusters.next(DayOfWeek.SUNDAY));
System.out.println("Second Sunday on : " + secondSunday);
}
}

```

Output:  
Current date: 2016-04-07  
Next Wednesday on: 2016-04-13  
Second Sunday on: 2016-04-10

Explain about TemporalAdjuster class using slides 36 to 38.

Explain that:

You can use TemporalAdjuster to modify a temporal object, especially date and time.  
You must input the temporal value and it outputs the adjusted value.

Using slides 37 and 38, explain the code that shows how to find the first day of a month using a specified date.

Explain that:

To invoke temporal adjuster, use ‘with’ method of the temporal object to be adjusted.

`LocalDate.now()` method returns the current date.

`sampledateA.with(TemporalAdjusters.next(DayOfWeek.WEDNESDAY))`  
method returns the next Wednesday from current date.

`firstInYear.with(TemporalAdjusters.nextOrSame(DayOfWeek.SUNDAY)).with(TemporalAdjusters.next(DayOfWeek.SUNDAY))` method returns the second Sunday of the month. Note the usage of with method with the temporal object.

#### Additional Information:

Refer the following links for more information:

<https://docs.oracle.com/javase/8/docs/api/java/time/TemporalAdjusters.html>

<http://javapapers.com/java/java-8-date-and-time-temporal-adjuster/>

## Slides 39 to 41

Let us understand backward compatibility support in Java 8 for legacy classes.

**Backward Compatibility with Older Versions 1/3**

`toInstant()`

```
Date sampleDate = new Date();
Instant sampleNow = sampleDate.toInstant();
LocalDateTime dateTime =
LocalDateTime.ofInstant(sampleNow, myZone);
ZonedDateTime zdt =
ZonedDateTime.ofInstant(sampleNow, myZone);
```

`Instant, ZoneId`

In the given code, `toInstant()` method is being added to the original `Date` and `Calendar` objects to convert them into new Date-Time API.

**Backward Compatibility with Older Versions 2/3**

`ofInstant(Instant, ZoneId)` Method is used to get a `LocalDateTime` or `ZonedDateTime` object.

```
import java.time.LocalDateTime; // to initiate local date and time
import java.time.ZonedDateTime; // to initiate zoned time
import java.util.Date;
import java.time.Instant;
import java.time.ZoneId;

public class BWCompatibility {
 public static void main(String args[]){
 BWCompatibility bwcompatibility = new BWCompatibility();
 bwcompatibility.sampleBW();
 }
 public void sampleBW(){
 // To display the current date
 Date sampleCurDay = new Date();
 System.out.println(" Desired Current date= " + sampleCurDay);
 // to display result
 // To display the instant of current date
 Instant sampleNow = sampleCurDay.toInstant();
 ZoneId sampleCurZone = ZoneId.systemDefault();
 }
}
```

**Backward Compatibility with Older Versions 3/3**

```
// To display the current local date
LocalDateTime sampleLoDaTi = LocalDateTime.ofInstant(sampleNow,
sampleCurZone);
System.out.println(" Desired Current Local date= " + sampleLoDaTi);

// To display result
// To display the desired current zoned date
ZonedDateTime sample2eDaTi = ZonedDateTime.ofInstant(sampleNow,
sampleCurZone);
System.out.println(" Desired Current Zoned date= " + sample2eDaTi);
// To display result
}
```

**Output:**

```
Desired Current date= Fri May 06 07:32:58 EDT 2016
Desired Current Local date= 2016-05-06T07:32:58.769
Desired Current Zoned date= 2016-05-06T07:32:58.769-04:00[America/New_York]
```

Explain about backward compatibility with older versions using slides 39 to 41.

Explain that:

Java 8 helps to convert the date and time class represented by `java.util.Date` class to `java.time.LocalDate` or `LocalDateTime` or `LocalTime` using the `toInstant()` method. This method converts a date object to an instant.

Using slides 40 and 41, explain the code to get a `LocalDateTime` or `ZonedDateTime` object.

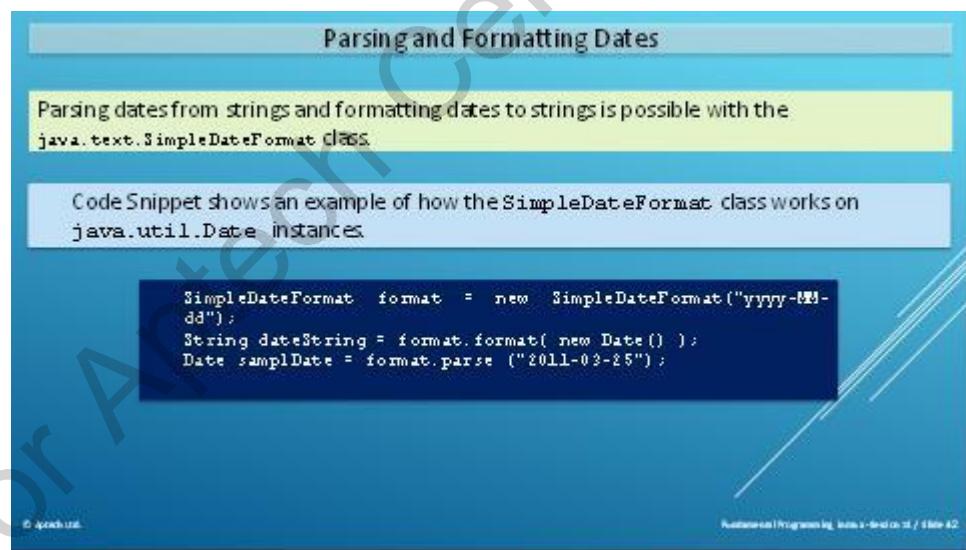
Explain that:

The code uses `ofInstant(Instant, ZoneId)` method to get a `LocalDateTime` or `ZonedDateTime` object. The new `Date()` method is used to get the current date. The method `sampleCurDay.toInstant()` is used to get the instant of current date.

`LocalDateTime.ofInstant(sampenow, samplecurZone)` method is used to display the current local date and the method `ZonedDateTime.ofInstant(sampenow, samplecurZone)` is used to display the desired current zoned date.

## Slide 42

Let us examine how to parse and format dates.



Explain about parsing and formatting dates.

Explain that:

`java.text.SimpleDateFormat` class helps to parse dates from strings (text to date) and format dates to string (date to text).

Explain the code that shows how the `SimpleDateFormat` class works on `java.util.Date` instances.

Explain that:

yyyy/MM/dd shows the date in the form of year/month/day. The year will be shown in full form. For example, 2016 of the year will be used. In the code, the dateString instance passed to the format() method is a java.util.Date instance.

When you execute the code, the sampleDate variable points to a Date instance that denotes March 25th, 2011.

### Additional Information:

Refer the following links for more information:

<https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html>

<https://dzone.com/articles/datetime-formattingparsing>

### Slides 43 and 44

Let us explore TimeZone.

**TimeZone (java.util.TimeZone) 1/2**

Time Zone (java.util.TimeZone) is used in time-zone bound calculations.

Code Snippet displays a simple example of how to get the time-zone from a Calendar.

```
Calendar cal = new GregorianCalendar();
TimeZone tizo = cal.getTimeZone();
```

Code Snippet displays a simple example of how to set time-zone.

```
cal.setTimeZone(tizo);
```

Code Snippet shows two ways to obtain a TimeZone instance.

```
TimeZone tizo = TimeZone.getDefault();
OR
TimeZone tizo =
TimeZone.getTimeZone("Europe/Paris");
```

B Aptech Limited Business Programming Java - Day 01 / Slide 43

**TimeZone (java.util.TimeZone) 2/2**

Following Code Snippet shows a sample of time-zone:

```
import java.time.ZonedDateTime;
import java.time.ZoneId;

public class Java8CurTimeZone {
 public static void main(String args[]){
 Java8CurTimeZone = new Java8CurTimeZone();
 java8curtime.sampleZDTime();
 }
 public void sampleZDTime(){
 // To display the current date and time
 ZonedDateTime dateA = ZonedDateTime.parse("2016-04-05T01:53+08:00[Asia/Singapore]");
 System.out.println("dateA: " + dateA);
 // To display the zoneId
 ZoneId sampleIdA = ZoneId.of("Asia/Singapore");
 System.out.println("ZoneId: " + sampleIdA);
 // To display the current Zone
 ZoneId sampleCurrentZoneA = ZoneId.systemDefault();
 System.out.println("CurrentZone: " +
sampleCurrentZoneA);
 }
}
```

Output:

dateA: 2016-04-05  
T10:15:30+08:00[Asia/Singapore]  
ZoneId: Asia/Singapore  
CurrentZone: Etc/UTC

B Aptech Limited Business Programming Java - Day 01 / Slide 44

Explain about `TimeZone` class and the codes to display a simple example of how to get the time-zone from a `Calendar`, how to set time-zone and the two ways to obtain a `TimeZone` instance.

**Explain that:**

`TimeZone` class in Java represents the time-zones and is used in time-zone bound calculations. You can get the time-zone using the `getTimeZone` method along with the time zone ID.

Using slide 44, explain the code that shows a sample of time-zone.

**Explain that:**

The `TimeZone.getDefault()` method displays the default time-zone of the system on which the program is executed. `TimeZone.getTimeZone("Europe/Paris")` method returns the `Time-Zone` instance corresponding to the given `TimeZoneID`, in this example Europe/Paris.

#### **Additional Information:**

Refer the following link for more information:

<http://www.oracle.com/technetwork/articles/java/jf14-date-time-2125367.html>

#### **In-Class Question:**

After you finish explaining slide 44, you will ask the students some In-Class questions. This will help you in reviewing their understanding of the topic.



What would be the output for this program?

```
import java.time.LocalDate;
import java.time.Month;
import java.time.format.DateTimeFormatter;
import java.time.Clock;

public class HelloWorld {
 public static void main(String[] args)
 {
 LocalDate dateFromString = LocalDate.parse("2014-
05-05");
 System.out.println("LocalDate from a String: " +
dateFromString);

 LocalDate date1 =
LocalDate.now(Clock.systemDefaultZone());
 }
}
```

```

 System.out.println("LocalDate from default time-
zone clock: " + date1);
 }
}

```

**Answer:**

LocalDate from a String: 2014-05-05

LocalDate from default time-zone clock: 2016-11-11



What would be the output for the following program?

```

import java.time.LocalDate;
import java.time.Month;
import java.time.format.DateTimeFormatter;
import java.time.Clock;

public class HelloWorld {
 public static void main(String[] args)
 {
 LocalDate localDate = LocalDate.now();
 System.out.println(localDate.toString());

 System.out.println(localDate.getDayOfWeek().toString());
 System.out.println(localDate.getDayOfMonth());
 System.out.println(localDate.getDayOfYear());
 System.out.println(localDate.isLeapYear());

 System.out.println(localDate.plusDays(12).toString());
 }
}

```

**Answer:**

Suppose today's date is: 11-11-2016, the output would be:

2016-11-11  
FRIDAY  
11  
316  
true  
2016-11-23



What would be the output for the following program?

```
import java.time.Duration;
import java.time.LocalTime;

public class HelloWorld {
 public static void main(String[] args) {
 Duration duration =
Duration.between(LocalTime.MIDNIGHT, LocalTime.NOON);
 duration = duration.plusNanos(1000);
 System.out.println(duration.getNano());
 }
}
```

**Answer:** 1000



What would be the output for the following program?

```
import java.time.LocalDate;

public class HelloWorld {
 public static void main(String[] args) {
 LocalDate a = LocalDate.of(2014, 6, 30);
 LocalDate b = a.plusMonths(8);
 System.out.println(b);
 }
}
```

**Answer:**

2015-02-28

## Slide 45

Let us summarize the session.

**SUMMARY**

- ❖ The new Date-Time API introduced in Java 8 is a solution for many unaddressed drawbacks of the previous API.
- ❖ Date-Time API contains many classes to reduce coding complexity and provides various additional features to work on date and time.
- ❖ Enum in Java denotes fixed number of well-known values.
- ❖ TemporalAdjuster is a functional interface and a key tool for altering a temporal object.
- ❖ Java TimeZone class is a class that denotes time-zones and is helpful when doing calendar arithmetic across time-zones.
- ❖ A time-zone offset is the quantity of time that a time-zone differs from Greenwich/UTC.

© Aptech Ltd.      Fundamental Programming in Java -Session 13 / Slide 45

Use this slide to summarize the session. You will end the session, with a brief summary of what has been taught in the session. Tell the students pointers of the session. This will be a revision of the current session.

Explain that:

The new Date-Time API introduced in Java 8 is a solution for many unaddressed drawbacks of the previous API. Date-Time API contains many classes to reduce coding complexity and provides various additional features to work on date and time. Enum in Java denotes fixed number of well-known values. TemporalAdjuster is a functional interface and a key tool for altering a temporal object. Java Time-Zone class is a class that denotes time-zones and is helpful when doing calendar arithmetic across time-zones. A time-zone offset is the quantity of time that a time-zone differs from Greenwich/UTC.

### 13.3 Post Class Activities for Faculty

You should familiarize yourself with the topics of the next session.

#### Tips:

You can also check the Articles/Blogs/Expert Videos uploaded on the Online Varsity site to gain additional information related to the topics covered in the next session. You can also connect to online tutors on the Online Varsity site to ask queries related to the sessions.

# Session 14 - Annotations and Base64 Encoding

## 14.1 Pre-Class Activities

Before you commence the session, you should familiarize yourself with the topics of this session in-depth. Prepare a question or two that will be a key point to relate the current session objectives.

### 14.1.1 Objectives

By the end of this session, learners will be able to:

- Explain declaring an annotation type in Java
- Describe predefined annotation types
- Explain Type annotations
- Explain Repeating annotations
- Describe Base64 encoding

### 14.1.2 Teaching Skills

To teach this session, you should be well versed with the concepts of Java Programming. You should be familiar with the concept of annotations, various types of annotations, and how to create and use them.

You should teach the concepts in the theory class using the images provided. For teaching in the class, you are expected to use slides and LCD projectors.

#### Tips:

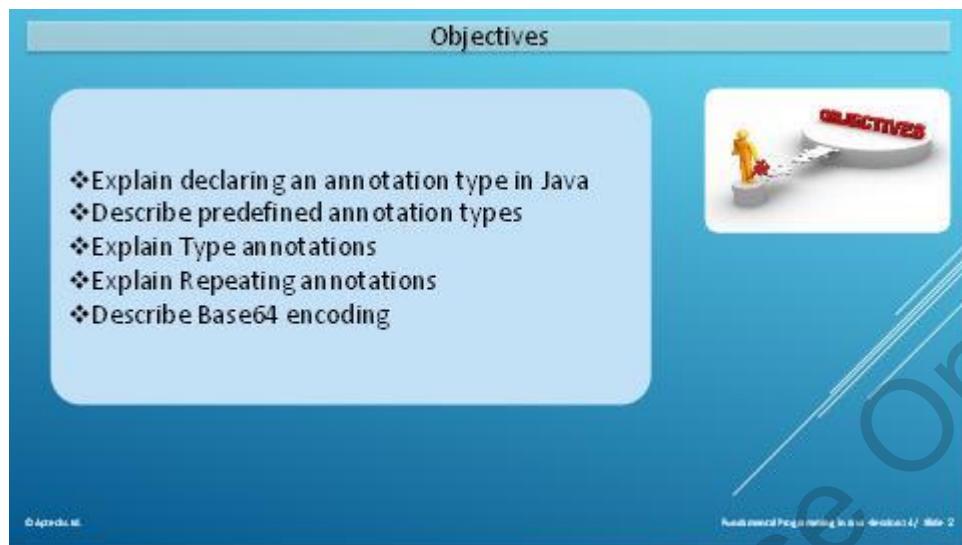
It is recommended that you test the understanding of the students by asking questions in between the class.

#### In-Class Activities

Follow the order given here during In-Class activities.

#### Overview of the Session

Give the students an overview of the current session in the form of session objectives. Read out the objectives on slide 2.

**Slide 2**

The slide has a blue header bar with the word "Objectives" in white. Below it is a white rounded rectangle containing a bulleted list of objectives. To the right is a small graphic of a person climbing a ladder labeled "OBJECTIVES". The footer contains the copyright notice "© Aptech Ltd." and the slide number "Annotations Programming in Java / Session 4 / Slide 2".

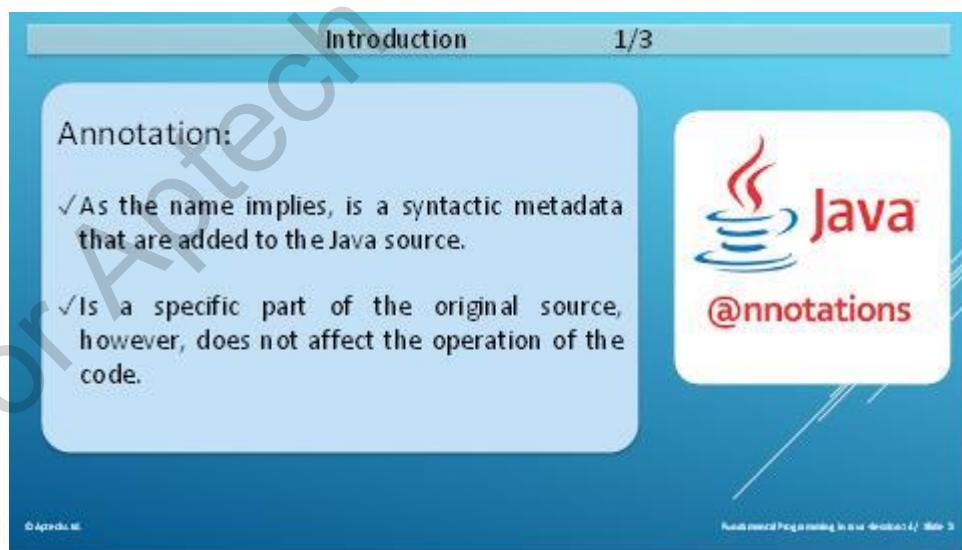
- ❖ Explain declaring an annotation type in Java
- ❖ Describe predefined annotation types
- ❖ Explain Type annotations
- ❖ Explain Repeating annotations
- ❖ Describe Base64 encoding

Show the slide and give the students a brief overview of the current session in the form of session objectives. Tell the students that they will get to know how to declare an annotation type in Java. Describe about predefined annotation types. Explain type annotations and repeating annotations. Finally, describe Base 64 encoding.

## 14.2 In-Class Explanations

### Slides 3 to 5

Let us understand annotations.



The slide has a blue header bar with "Introduction" and "1/3". Below is a white rounded rectangle containing text and a list. To the right is a graphic with the Java logo and the word "@nnotations". The footer contains the copyright notice "© Aptech Ltd." and the slide number "Annotations Programming in Java / Session 4 / Slide 3".

Annotation:

- ✓ As the name implies, is a syntactic metadata that are added to the Java source.
- ✓ Is a specific part of the original source, however, does not affect the operation of the code.

Introduction 2/3

Following elements are annotated in Java:

|            |         |
|------------|---------|
| Tables     | Methods |
| Variables  | Classes |
| Parameters |         |

© Aptech

Annotated Programming in Java 5/Java 6 / Page 4

Introduction 3/3

An enhanced feature is introduced in Java 8 called Pluggable Annotation Processing API.

API helps to

- Write a customized Annotation Processor
- Code dynamically to operate on the set of annotations

© Aptech

Annotated Programming in Java 5/Java 6 / Page 5

Using slide 3, give an introduction about annotations.

Explain that annotations are comments, notes, remarks, or explanations. They are used to link metadata to the elements. Annotations are declared using @ symbol followed by the interface keyword and annotation name. They can be declared at class level, field level, and method level.

Using slide 4, state the elements that are annotated in Java.

Elements that are annotated in Java are namely, tables, methods, variables, classes, and parameters. Java 5 allowed annotation processing during compilation of code as well as execution. When Java source code is compiled, annotations are processed by compiler plug-ins called annotation processor.

Using slide 5, state the benefits of Pluggable Annotation Processing API in Java 8.

Java SE 6 consists of Pluggable Annotation Processing API that helps application developers to plug a Customized Annotation Processor into the code to operate on the set of annotations existing in a source file.

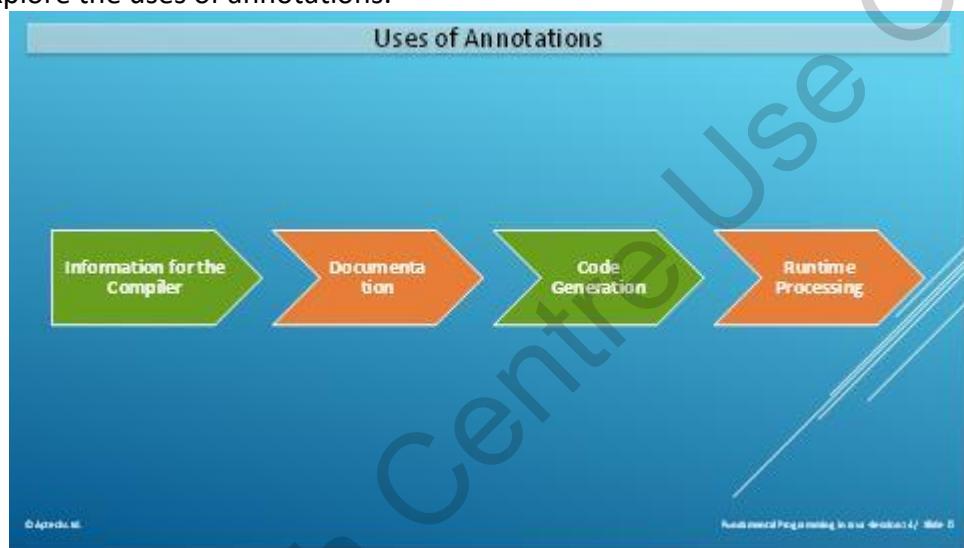
### **Additional Information:**

Refer the following link for more information:

<http://howtodoinjava.com/core-java/annotations/complete-java-annotations-tutorial/>

### **Slide 6**

Let us explore the uses of annotations.



Explain the uses of annotations.

Say that some of the uses of annotations are:

- **Information for the compiler:** Helps compiler to produce notifications or errors based on different rules. For example, the `@FunctionalInterface` annotation helps the compiler to validate the annotated class and check whether it is a functional annotation or not. When a type is annotated as `@FunctionalInterface`, compilers generate an error message if:
  - The type is an interface type and not an annotation type, enum, or class.
  - The annotated type satisfies the requirements of a functional interface.
- **Documentation:** Decides the quality of the code or generate automatic reports.
- **Code generation:** Creates code or XML files automatically using metadata information available in the code.
- **Runtime processing:** Annotations can be processed at runtime or compile time or both. Annotation is available at runtime if it has the runtime `RetentionPolicy`. Helps applications in purposes such as unit testing, dependency injection, validation, logging, data access, and so on.

**Additional Information:**

Refer the following link for more information:

<http://beginnersbook.com/2014/09/java-annotations/>

**Slide 7**

Let us see how to declare annotations.

The slide has a light blue header bar with the title "Declaring Annotations". Below it is a white content area. At the top left of the content area, there is a box containing the text "To declare an annotation, prefix with @ as follows:". Inside this box, under the heading "Syntax:", is the code "@<name>". Below this, under the heading "Example:", is the code "@Item". A callout box points to the "@Item" code with the text "Here, the annotation name is Item". At the bottom left of the slide, there is a small copyright notice: "© Aptech Ltd.". At the bottom right, there is a footer note: "Fundamental Programming in Java -Session 14 / Slide 7".

Explain the procedure to declare an annotation.

Explain that to write custom annotation, you must prefix the annotation name with @ symbol. For example, @Author where author is the name of the annotation.

**Additional Information:**

Refer the following link for more information:

<https://www.mkyong.com/java/java-custom-annotations-example/>

## Slides 8 to 11

Let us explore predefined annotations.

Predefined Annotations 1/4

Following are three built-in annotations:

- @Deprecated
- @Override
- @SuppressWarnings

© Aptech Ltd. Java™ Programming In easy steps 2/e - Page 5

Predefined Annotations 2/4

- @Deprecate
  - Annotation deprecates a class, method, or field
- @Override
  - Creates a compile time check
- @SuppressWarnings
  - Can compress compiler warnings

© Aptech Ltd. Java™ Programming In easy steps 2/e - Page 5

**Predefined Annotations** 3/4

Predefined annotations are demonstrated through following codes:

```

@Deprecated
@Deprecated
public class SampleContent {
}

public class Test {
 public static void main(String args[]) {
 SampleContent = new SampleContent();
 }
}

@Override
public class ClassOne {
 public void show (String text) {
 System.out.println(text);
 }

 public static void main(String args[]) {
 SubClass obj = new SubClass();
 obj.show ("Good day!!!");
 }
}

class SubClass extends ClassOne {
 @Override
 public void show (String text) {
 System.out.println("I want to say: " + text);
 }
}

@SuppressWarnings
@SuppressWarnings ("unchecked")
@SuppressWarnings ("rawtypes")
public void abcMethod() {
}

```

Apache © Apache

**Predefined Annotations** 4/4

There are two advantages of using `@Override` annotation:

- If a programmer makes any unintentional error while overriding, then it results in a compile time error. Using `@Override` instructs compiler that it is overriding this method.
- It helps to enhance code readability.

Apache © Apache

Using slide 8, explain about predefined annotations.

Predefined annotations are the built-in annotations available in the compiler that can be readily used. Following are the three built-in annotations:

- `@deprecated`
- `@override`
- `@suppresswarnings`

Using slide 9, explain each of the predefined annotation.

`@deprecated` annotation: Indicates parts of the code that are not required and hence need not be executed. When a code consisting deprecated class, method, or field is compiled, the compiler generates a warning message.

`@override`: Indicates that the annotated method is an overridden method.

`@suppresswarnings`: Suppresses the compiler warning. For example, the compiler generates a warning message when a method calls a deprecated method.

Using slide 10, explain the code snippets, describing each of the predefined annotation.

Explain that:

@deprecated annotation: In the code, using @deprecated annotation before the class declaration turns the entire class as deprecated. When you try to generate an instance of the class, you will get a compiler warning.

@override: Code Snippet shows class ClassOne has a method named show() and its subclass also has a method show() whereas, the child class is not overriding the parent class method, as it has a different parameter type String and int respectively.

@suppresswarnings: Code Snippet shows that @SuppressWarnings annotation type can be used with one or more warnings in the form of arguments. These warnings are predefined by the compiler.

Explain using slide 11 the advantages of using @override annotation.

Explain that:

@override annotation:

- Helps to identify any unintentional error such as wrong method, parameter type in the code. Once the error is identified, the annotation can be used to instruct the compiler to override the erroneous method.
- Makes the code easy to read and understand. You can understand the behavior of the code and the impact of renaming methods and parameter type in the code.

#### Additional Information:

Refer the following links for more information:

<http://tutorials.jenkov.com/java/annotations.html>

<http://stackoverflow.com/questions/212614/should-we-override-an-interfaces-method-implementation>

## Slides 12 and 13

Let us see how to create custom annotations.

Creating Custom Annotations
1/2

Custom Annotations created in Java are defined in their own files.

```
@interface SampleAnnotate{
 String samp1Value();
 String name();
 int age();
 String[] addNames();
}
```

© Aptech Ltd.
Fundamental Programming in Java -Session 14 / Slide 12

Creating Custom Annotations
2/2

Custom Annotations uses following syntax:

| @Retention                                                                           | @Target                                                                        | @Inherited                                                                                                                                                                          | @Documented                                                                                                                                |
|--------------------------------------------------------------------------------------|--------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>@Retention(RetentionPolicy.RUNTIME) @interface SampleAnnotate {     ... }</pre> | <pre>@Target({ElementType.METHOD}) public @interface SampleAnnotate{...}</pre> | <pre>import java.lang.annotation.Inherited; @Inherited public @interface SampleAnnotate { } @SampleAnnotate class Person { ... } public class Employee extends Person { ... }</pre> | <pre>import java.lang.annotation.Documented; @Documented @interface TestAnnotate { ... } @TestAnnotate public class Employee { ... }</pre> |

© Aptech Ltd.
Fundamental Programming in Java -Session 14 / Slide 13

Using slide 12, explain the code to create custom annotations.

Explain that:

To create custom annotation, it is mandatory to use `@interface` keyword. The code defines an annotation called `SampleAnnotate`, having four elements: `samp1Value`, `name`, `age`, and `addNames`.

Explain the syntax shown on slide 13.

Each element in an annotation is defined similar to a method in an interface. It consists of a data type and a name.

`@retention` annotation specifies the way the annotation, `SampleAnnotate` must be stored. `RetentionPolicy` determines the stopping point of annotation.

`@target` annotation restricts the usage of the custom annotation on specific Java elements such as interface, class, or methods. `@Target` restricts the usage of `SampleAnnotate` on Methods.

`@inherited` annotation when applied to indicates that all the subclasses must inherit the annotated class. Sub class `Employee` inherits from the class `Person`.

`@documented` annotation indicates the JavaDoc tool that custom annotations, `@TestAnnotate` have to be visible in the JavaDoc for the `Employee` class using custom annotation.

### Slide 14

Let us explore type annotations.

The slide has a blue header bar with the title "Type Annotations". Below the header, there is a bulleted list of five points:

- Are formed to maintain better analysis of Java programs, ensuring better type checking.
- For example, to ensure that a variable in program is never assigned to null or to avoid triggering a `NullPointerException`, a custom plugin can be written.
- Then, modify code to annotate variable specifying that it is never assigned to null.
- Following syntax shows the variable declaration:

```
ONNonNull String str;
```

Here,  
`str` is a String variable in the annotation.

Compiler shows a warning if it notices a potential problem, allowing you to modify the code to avoid the error while calculating the NonNull module at the command line during code compilation. After editing the code to remove all warnings, this specific error will not occur when the program runs.

© Aptech Limited. *Annotated Programming - Java 8 - Day 1 / Slide 14*

Using slide 14, explain about type annotations also explain the Code Snippet that shows usage of type annotations.

Explain that:

Java 8 does not consists built-in type checking annotation. Type annotations helps in type checking in a program. You can apply annotations to types in instances such as creating class instance, implementing methods, type casting.

#### Additional Information:

Refer the following link for more information:

<https://dzone.com/articles/java-8-type-annotations>

**Slides 15 to 18**

Let us understand repeating annotations.

**Repeating Annotations** 1/4

Helps to apply the same annotations at multiple times for the same declaration.

```
@ScoreSchedule(dayOfMonth="last")
@ScoreSchedule(dayOfWeek="Wed", hour="21")
public void scorePapers() { ... }
```

Here, the annotation `@ScoreSchedule` has been applied twice, thus, it is a repeating annotation. Repeating annotations can be applied not only to methods also to any item that can be annotated.

© Aptech Ltd. Fundamental Programming in Java - Session 14 / Slide 15

**Repeating Annotations** 2/4

For compatibility, repeating annotations are loaded in a container annotation generated by Java compiler.

For the compiler to perform this, two declarations are necessary in the code:

- 1) Declare a Repeatable Annotation type.

```
/*
 *Aptech Java8
 *Java tester
 */
import java.lang.annotation.Repeatable;
@Repeatable(ScoreSchedules.class)
public @interface ScoreSchedule {
 String monthDay() default "1st";
 String weekDay() default "Monday";
 int hour() default 12;
}
```

© Aptech Ltd. Fundamental Programming in Java - Session 14 / Slide 15

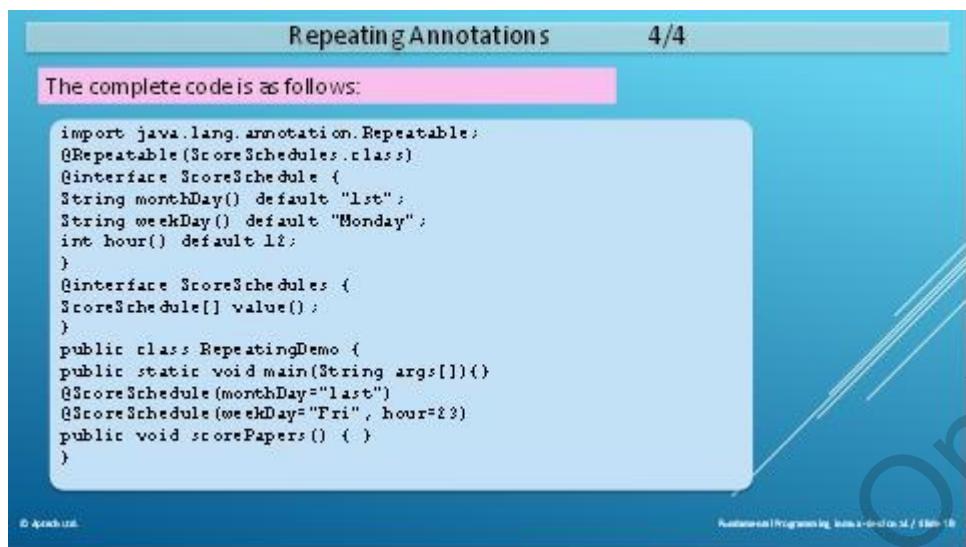
**Repeating Annotations** 3/4

- 2) Declare containing annotation type.

```
public @interface ScoreSchedules {
 ScoreSchedule[] value();
}
```

Here, `ScoreSchedules` is the class for `@interface` annotation.

© Aptech Ltd. Fundamental Programming in Java - Session 14 / Slide 15



Using slide 15, explain about repeating annotations and also explain the code snippet that shows usage of repeating annotations.

Explain that:

Repeating annotations help to repeat the same annotations more than once in a program. They can be applied not only to methods but also to any item that can be annotated.

```

public @interface Coaching {
 String subject();
}

@ Coaching(subject= "Maths")
@ Coaching(subject="Science")
@ Coaching(subject="English")
public class school {
}

```

Here, @Coaching is the repeating annotation.

Using slide 16, explain about the two necessary declarations in the code to use repeating annotations using the Code Snippet.

Explain that:

To define repeating annotations:

- Annotate repeating annotation using @Repeatable for a annotation to be repeatable.
- Create a container annotation to store repeating annotations.

In the Code Snippet, annotating repeating annotation ScoreSchedules using @Repeatable.

Using slide 17, explain the second necessary declarations in the code to use repeating annotations from the Code Snippet.

The second necessary declaration is to declare containing annotation type. The containing annotation type contains an array type element value. The array type value must be the repeatable annotation type. In the Code Snippet that shows the containing annotation type declaration, `ScoreSchedules` is the class for `@interface` annotation.

Using slide 18, explain the complete Code Snippet that shows using the second necessary declarations in the code to use repeating annotations.

In the Code Snippet `@Repeatable (ScoreSchedules.class)`, `@interface ScoreSchedule` enables to run a method `scorePapers ()` at a given time on the last day of a month and on every Wednesday at 09:00 p.m.

### Slides 19 to 22

Let us examine processing of annotations through reflection.

**Processing Annotations Using Reflection** 1/4

- Reflection API of Java can be used to access annotations on any type such as class or interface or methods.
- Several methods in Reflection API help to retrieve annotations.
- Methods that generate a single annotation, such as `AnnotatedElement.getAnnotationByType (Class <T>)` remain unchanged.
- They return only a single annotation when one annotation of the requisite type is available.
- If one or more annotation of the required type is available, it can be acquired by getting the container annotation.

© Aptech

Annotations Programming Java - slide 19 / slide 19

**Processing Annotations Using Reflection** 2/4

```

import java.lang.reflect.Method;
import java.lang.reflect.ParameterizedType;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.IntStream;

public class SampleClass {
 public static void main(String[] args) {
 RuntimeException exception = new RuntimeException();
 exception.printStackTrace();
 ParameterizedType sampleType = (ParameterizedType) exception.getActualTypeArguments()[0]; // here List<Type>
 // defined as SampleList
 ParameterizedType sampleCity = (ParameterizedType) sampleType.getActualTypeArguments()[0]; // here Cities<Type>
 // defined as SampleCity
 WildcardType sampleCityType = (WildcardType) sampleCity.getActualTypeArguments()[0]; // here generic Type
 // defined as SampleCityType
 Class<?> sampleGenC = (Class<?>) sampleCityType.getActualTypeArguments()[0]; // here generic Class defined as sampleGenC
 boolean exception = exception instanceof SampleGenC; // to display whether the statement is true or false
 System.out.println("This Class extends RuntimeException " + exception);
 boolean RuntimeException = RuntimeException instanceof instanceofSampleGenC; // to display whether the statement is true or false
 System.out.println("This Class extends RuntimeException " + instanceof RuntimeException);
 }
}

```

© Aptech

Annotations Programming Java - slide 20 / slide 20

The image contains two screenshots of a Java IDE interface, likely Eclipse or IntelliJ IDEA, demonstrating the use of annotations and reflection.

**Screenshot 1 (Top): Processing Annotations Using Reflection**

- Output:** A yellow box displays the output of a program. It shows two lines of text: "This Class extends RuntimeException: true" and "This Class extends RuntimeException: false".
- Description:** Below the output, a message states: "Here, SampleGenC1 class is reflected through sampMeth() method."
- Code Snippet:** A yellow box contains the code for the `@FunctionalInterface` annotation.

**Screenshot 2 (Bottom): Processing Annotations Using Reflection**

- Description:** A message states: "@Functional Interface: The annotated element will operate as a functional interface and compiler generates an error if the element does not comply with the requirements."
- Code Snippet:** A yellow box contains the code for the `@FunctionalInterface` annotation, identical to the one in Screenshot 1.

Using slide 19, explain about processing annotations using Reflection API.

Explain that:

In Java, Reflection API helps to access annotations on any type such as class, or interface, or methods. The API contains several methods to retrieve annotations.

For example, `AnnotateElement.getAnnotationByType(Class <T>)` remains unchanged and returns only a single annotation when one annotation of the requisite type is available. If more than one annotation is available, then it can be acquired by using the container annotation.

Using slide 20, explain the Code Snippet that displays the usage of Reflection annotation.

Explain that:

Code Snippets shows the usage of Reflection annotation. In the code, `SampleGenC1` class is reflected through `sampMeth()` method.

Using slide 21, discuss the output of the Code Snippet that displays the usage of Reflection annotation.

Using slide 22, explain @Functional interface using the Code Snippet that shows usage of this annotation.

Explain that:

A functional interface has one abstract method (not default). When an annotation having functional interface is compiled, the compiler will operate the annotated element as a functional interface and generate an error if the element does not comply with the requirements.

#### Additional Information:

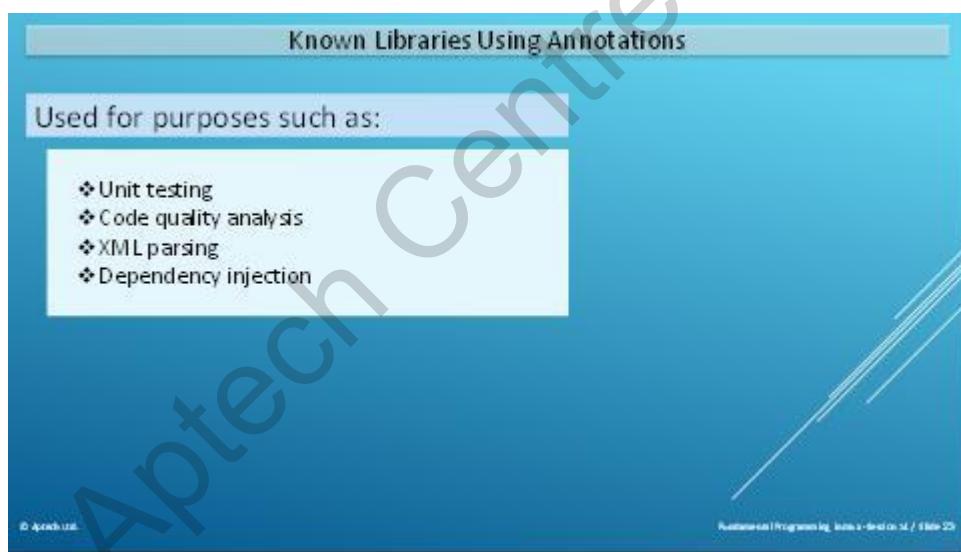
Refer the following links for more information:

<http://tutorials.jenkov.com/java-reflection/annotations.html>

[https://www.tutorialspoint.com/java8/java8\\_functional\\_interfaces.htm](https://www.tutorialspoint.com/java8/java8_functional_interfaces.htm)

#### Slide 23

Let us look at well-known Java libraries that use annotations.



Using slide 23, explain about the well-known libraries that are using annotations.

Explain that:

Libraries use annotations for analysing code quality analysis, unit testing, XML parsing, dependency injection. Some of these libraries include:

- Java Architecture for XML Binding (JAXB) which helps easier access to XML documents from applications written in the Java programming language
- JUnit, which is a unit testing framework for Java programming language
- FindBugs, which is an open source static code analyzer which detects possible bugs in Java programs

**Additional Information:**

Refer the following link for more information:

<https://www.javacodegeeks.com/2014/11/java-annotations-tutorial.html#Known%20libraries%20using%20annotations>

**Slides 24 to 28**

Let us explore Base64 encoding.

**Base64 (java.util.Base64) Encoding** 1/5

Java 8 encloses an inbuilt encoder and decoder for Base64 encoding.

Three types of Base64 encoding:

- Simple** •Output is limited to a set of characters between A-Z, a-z, 0-9, and +.
- URL** •The final output is safe from filename and URL and is limited to a set of characters between A-Z, a-z, 0-9, and +\_.
- MIME** •Output is limited to MIME friendly format.

© Aptech Ltd. Fundamental Programming in Java -Session 14 / Slide 24

**Base64 (java.util.Base64) Encoding** 2/5

Two nested classes in Base64:

- static class Base64.Encoder
- static class Base64.Decoder

© Aptech Ltd. Fundamental Programming in Java -Session 14 / Slide 25

Base64 (java.util.Base64) Encoding

3/5

Methods:

```
static Base64.Decoder getDecoder()
```

```
static Base64.Encoder getEncoder()
```

```
getMimeDecoder()
```

```
getMimeEncoder()
```

```
getMimeEncoder(int lineLength, byte[] lineSeparator)
```

```
getUrlDecoder()
```

```
getUrlEncoder()
```

## Base64 (java.util.Base64) Encoding

4/5

Following code encodes a string to base64, then decode the same String back to a base64 encoded output stream:

```
import java.util.Base64;
import java.util.Scanner;
import java.io.UnsupportedEncodingException;
/*Special Java
 *license example to detail
 */
public class Base64 {
 public static void main(String args[]) {
 // Encoding a string using Base64
 String argBase64Encode = Base64.getEncoder().encodeToString("HelloWorld".getBytes("UTF-8"));
 System.out.println("Encoded String (Base64) looks like this: " + argBase64Encode);
 // Decoding a string using Base64
 byte[] base64DecodeBytes = Base64.getDecoder().decode(argBase64Encode);
 System.out.println("Decoded string is: " + new String(base64DecodeBytes, "UTF-8"));
 String argBase64Decode = Base64.getDecoder().decode("MjAxLjIwMjQgMS4z");
 System.out.println("Decoded string looks like this: " + argBase64Decode);
 String resultString = new String(argBase64Decode);
 for (int i = 0; i < 30; i++) {
 resultString.append("A");
 }
 System.out.println(resultString);
 }
}
```

## Base64 (java.util.Base64) Encoding

5/5

Output:

Encoded String (Basic) looks like this: QXB0ZU0No3mF2YTg=

Decoded String is: AptechJava8

Encoded String (URL) looks like this: QXB0ZU0o3mF2YTg=

Encoded String (MIME) look like this:

```
YzNlNmYyNjctY2JkN000OTfILThiMzQt0T2Tjk2jk2Tu2YT1jM2F1N2YtZTJ1Bi00YmV1L0i2
NDQteRjFjMGziYmI4MDc5MWFm0T&3YjkjtYTc5Mi00MDEBnLTg2ZjIt0T11OGRIiMDJk0DE4YmZm2mYx
YzQtNjA1Yz00M2E2L0JkNtM+MzEo00I5MaYmGtY4NGE0YTZh2jUe200Q5Z100N0Y0LUE3YTAt7mZi
MDU5M0RjMG820Wj0TjYtDE+2GULYi00WFiLT11MmYtM2MyYjg3MjY4ZTYzNDg0MGMsYzctZjEy
M3000Tj1LThjM2M+2T0+MDE+MmQm#2R1MDdYjMyMTYtMjI2Mj00NjWmLTgy&It0W506Q3OTdH
ZmJ1Mzj1YyNrbN0E+2mHlMi00Zm1llTg5MDatZjdj26Jj00Fh20WjNTg4YYX0DktYjk2NC00YTJk
LT1hM02tYmNmNGRm0TQ5MTI4
```

Explain about Base64 encoding.

Explain that:

The three types of Base64 encoding are:

**Simple:** The encoder do not add any line feed in output and the decoder declines any input other than A-Z, a-z, 0-9, and +/-.

`Base64.getEncoder().encodeToString()` method is used for encoding strings.

`Base64.getDecoder().decode()` method is used for decoding strings.

**URL:** This type of encoding helps to encode and decode filename or Url in an input. They return the output without the filename and URL. However, the output is limited to a set of characters between A-Z, a-z, 0-9, and +\_.

`Base64.getUrlEncoder().encodeToString()` method is used for encoding filename and Url content in an input.

**MIME:** The output is MIME friendly and is denoted in lines with maximum 76 characters each. `Base64.getMimeEncoder().encodeToString()` method is used to encode using the MIME type base 64 encoding scheme.

Using slide 25, explain about the two nested classes in Base64 encoding.

Explain that:

Base64 encoding has two nested classes, static class `Base64.Encoder` and static class `Base64.Decoder`. Both the classes use the Base64 encoding scheme. These class implements an encoder for encoding byte data and a decoder for decoding byte data respectively.

Using slide 26, explain the methods used for Base64 encoding.

Explain that:

`getEncoder()` and `getDecoder()` methods process basic strings.

`getMIMEEncoder()` and `getMIMEdecoder()` methods processes the MIME type input. `getUrlEncoder()` and `getUrlDecoder()` method process URLs and filename.

Using slides 27 and 28, explain the Code Snippet that shows how to encode a string to Base64, then decode the same string back to a Base64 encoded output stream.

Explain that:

Base64 class inherits few methods from the `java.lang.Object` class. The recommended encoding scheme to use is UTF-8.

When encoding a string, the alphanumeric characters “a” through “z”, “A” through “Z” and “0” through “9” remains unchanged, the special characters “.”, “\_”, “\*”, and “\_” remains

unchanged, the space character “ ” is converted into a plus sign “+”, and all other characters are unsafe and are first converted into one or more bytes using some encoding scheme.

In the Code Snippet, the string “AptechJava8” is encoded and decoded back to a Base64 encoded output.

Using slide 28, explain the output of the Code Snippet.

#### **Additional Information:**

Refer the following links for more information:

<https://dzone.com/articles/base64-encoding-java-8>

[https://www.tutorialspoint.com/java8/java8\\_base64.htm](https://www.tutorialspoint.com/java8/java8_base64.htm)

<http://howtodoinjava.com/java-8/base64-encoding-and-decoding-example-in-java-8/>

#### **In-Class Question:**

After you finish explaining slide 28, you will ask the students some In-Class questions. This will help you in reviewing their understanding of the topic.



If a class method is implementing an interface method, which annotation will be used?

**Answer:** @Override annotation



Will the following code compile without error? Why or why not?

```
public @interface Food { ... }
@Food("breakfast", item="porridge")
@Food("lunch", item=" spaghetti")
@Food("dinner", item="pizza")
public void evaluateDiet() { ... }
```

**Answer:** The code will not compile. Since JDK 8, annotations should be defined as repeatable. In the code, the Food annotation type is not defined as repeatable. Hence, it can be fixed by adding the @Repeatable meta-annotation and defining a container annotation type as follows:

```
@java.lang.annotation.Repeatable(FoodContainer.class)
public @interface Food { ... }

public @interface FoodContainer {
 Food[] value();
```



While compiling the following code, the program throws a warning, what is the warning and how will you resolve the warning?

```
public interface Door {
 @Deprecated
 public void open();
 public void openLeftDoor();
 public void openRightDoor();
}

public class Restaurant implements Door {
 public void open() {}
 public void openLeftDoor() {}
 public void openRightDoor() {}
}
```

**Answer:** Through the following ways, we can resolve warning message:

1. By deprecating the implementation of open() method.

```
public class Restaurant implements Door {
 // The documentation is
 // inherited from the interface.
 @Deprecated
 public void open() {}
 public void openLeftDoor() {}
 public void openRightDoor() {}
}
```

2. The warning can also be suppressed as follows:

```
public class Restaurant implements Door {
 @SuppressWarnings("deprecation")
 public void open() {}
 public void openLeftDoor() {}
 public void openRightDoor() {}
}
```

**Slide 29**

Let us summarize the session.

**Summary**

- ❖ Annotations are comments, notes, remarks, or explanations. In Java, annotations help to associate these additional information (also called metadata) to the program elements. Annotations can be determined from source files or class files at runtime.
- ❖ The @Deprecated annotation is used for deprecating or marking a class, method or field as deprecated, signifying that the part of code no longer will no longer be used.
- ❖ The @SuppressWarnings annotation can suppress the compiler warnings in any available method.
- ❖ Since, annotation types are piled up and stored in byte code files such as classes, the annotations reverted by these methods can be enquired as any systematic Java object.
- ❖ Base64 encoding has an in-built encoder and a decoder. In Java 8, there are three types of Base64 encoding namely, Simple, URL, and MIME.

© Aptech Limited. Page 14 of 14 | Session 14 | Annotations and Base64 Encoding / 14 Slide 29

Use slide 29 to summarize the session. You will end the session, with a brief summary of what has been taught in the session. Tell the students the key pointers of the session. This will be a revision of the current session.

Explain that:

Annotations are comments, notes, remarks, or explanations. In Java, annotations help to associate these additional information (also called metadata) to the program elements. Annotations can be determined from source files or class files at runtime. The @Deprecated annotation is used for deprecating or marking a class, method or field as deprecated, signifying that the part of code no longer will be used. The @SuppressWarnings annotation can suppress the compiler warnings in any available method. Since, annotation types are piled up and stored in byte code files such as classes, the annotations reverted by these methods can be enquired as any systematic Java object. Base64 encoding has an in-built encoder and a decoder. In Java 8, there are three types of Base64 encoding namely, Simple, URL, and MIME.

### 14.3 Post Class Activities for Faculty

You should familiarize yourself with the topics of the next session.

#### Tips:

You can also check the Articles/Blogs/Expert Videos uploaded on the Online Varsity site to gain additional information related to the topics covered in the next session. You can also connect to online tutors on the Online Varsity site to ask queries related to the sessions.

# Session 15 – Functional Programming in Java

## 15.1 Pre-Class Activities

Before you commence the session, you should familiarize yourself with the topics of this session in-depth. Prepare a question or two that will be a key point to relate the current session objectives.

### 15.1.1 Objectives

By the end of this session, learners will be able to:

- Explain lambda expressions
- Describe method references
- Explain functional interfaces
- Explain default methods

### 15.1.2 Teaching Skills

To teach this session, you should be well versed with functional programming in Java. You should be familiar with lambda expressions, creating and using them, method references, and various functional interfaces present in `java.util.function` package. You must also be familiar with default methods. You should teach the concepts in the theory class using the images provided. For teaching in the class, you are expected to use slides and LCD projectors.

#### Tips:

It is recommended that you test the understanding of the students by asking questions in between the class.

#### In-Class Activities

Follow the order given here during In-Class activities.

#### Overview of the Session

Give the students an overview of the current session in the form of session objectives. Read out the objectives on slide 2.

**Slide 2**

The slide has a light blue header bar with the word "Objectives". Below it is a white rounded rectangle containing a bulleted list of four items:

- ❖ Explain lambda expressions
- ❖ Describe method references
- ❖ Explain functional interfaces
- ❖ Explain default methods

On the right side of the slide is a 3D-style illustration of a yellow stick figure standing on a white ladder. The ladder is leaning against a white circular platform. On top of the platform, the word "OBJECTIVES" is written in red, blocky letters. The background of the slide is a gradient from light blue at the top to white at the bottom.

Show the slide and give the students a brief overview of the current session in the form of session objectives. Tell the students that this session defines Functional Programming in Java. They will be introduced to lambda expressions. The session will describe method references and functional interfaces. Finally, the session will explain about default methods and static methods in interfaces.

### 15.2 In-Class Explanations

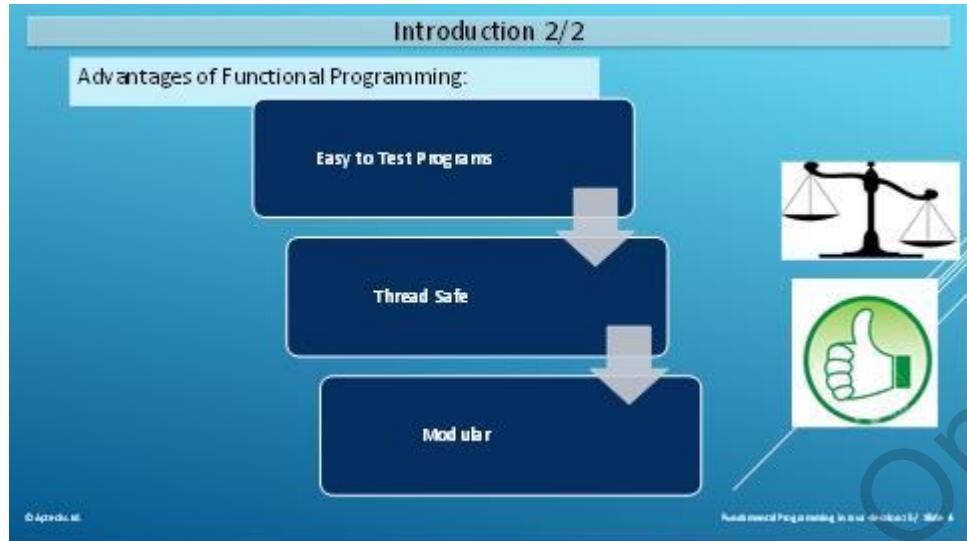
#### Slides 3 and 4

Let us learn about lambda expressions.

The slide has a light blue header bar with the text "Introduction 1/2". Below it is a white rounded rectangle containing the following text:

Lambda expressions are newly introduced in Java 8 to facilitate functional programming.

On the right side of the slide is a large black lambda symbol ( $\lambda$ ) next to the word "Expressions" in orange. The background of the slide is a gradient from light blue at the top to white at the bottom.



Using slide 3, give an introduction about lambda expressions. Explain that functional programming emphasizes on evaluation of expressions instead of executing commands.

Functional programming makes a program modular, thread-safe, and more readable. A program is considered to be thread-safe if it manages shared data structures in a manner that ensures secure implementation by multiple threads simultaneously. Modular denotes dividing the functionality of a program into independent and compatible modules.

Following code shows an example that can be used to explain to the students:

```
var str = function(text) {
 return function() {
 alert(text);
 }
};
setTimer(str('Say Hi after 10 seconds'), 10000);
```

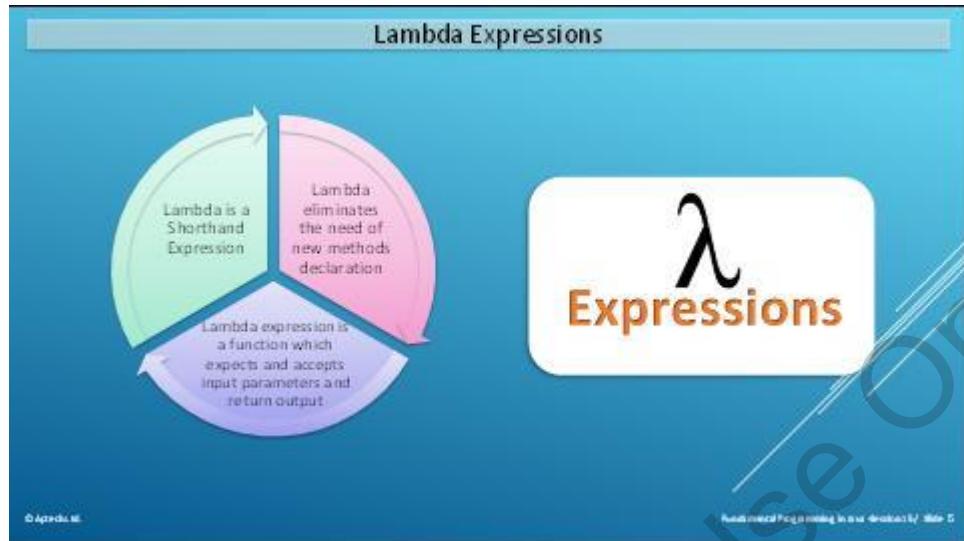
On compilation, `str()` will be created. `str()` returns an anonymous function. The anonymous function alerts the argument `text` that is passed to `str()` function.

Using slide 4, state the advantages of functional programming.

Say that lambda expressions are newly introduced in Java 8 to facilitate functional programming.

## Slides 5 to 7

Let us explore lambda expressions in detail.



### Syntax of Lambda Expressions

**Syntax:**  
parameters → body

where,  
parameters are variables  
→ is the lambda operator  
body is parameter values

### Rules for Lambda Expressions

**Rules** →

- Type declarations are optional
- Parentheses around parameters can be omitted
- Curly braces may or may not be present
- Return keyword may or may not be present
- Varied number of statements

Using slide 5, describe lambda expressions.

Explain that:

Lambda expression facilitates functional programming. It is a function that does not belong to any class. It is similar to shorthand which allows user to write a method at the same place where it is going to be used.

Using slide 6, explain the syntax of lambda expressions.

Explain that:

Examples of declaration of lambda expressions are:

```
(int a, int b) → { return a + b; }
() → System.out.println("My Black Cat");
(String str) → { System.out.println(str); }
() → 102
() → { return 3.1415 };
```

Using slide 7, explain the rules of lambda expressions.

Explain that:

The data type of the parameters in a lambda expression is optional. The body may consist of a single expression such as `() → System.out.println("Hi")`; or a statement block such as:

```
() → {
 one(() → { System.out.println("I am one."); });
 two(() → { System.out.println("I am two."); });
}
```

For single expression, curly braces are optional and for single parameters, parentheses is optional.

#### Additional Information:

Refer the link for more information:

[https://www.tutorialspoint.com/java8/java8\\_lambda\\_expressions.htm](https://www.tutorialspoint.com/java8/java8_lambda_expressions.htm)

## Slides 8 to 10

Let us learn about single method interfaces and lambdas.

**Single Method Interface and Lambdas 1/3**

- Functional programming creates event listeners.
- Event listeners are defined as Java interfaces.

A single method interface Code Snippet where State is already declared will be:

```
interface StateChangeListener {
 public void onStateChange(State previousState,
 State presentState);
}
```

**Single Method Interface and Lambdas 2/3**

Code Snippet shows adding an event listener using an anonymous method implementation.

```
public class StateTest {
 public static void main(String args[]) {
 StateTest objStateTest = new StateTest();
 objStateTest.addStateListener(new StateChangeListener() {
 public void onStateChange(State previousState, State presentState) {
 // action statements.
 System.out.println("State change event occurred");
 }
 });
 }
}
```

**Single Method Interface and Lambdas 3/3**

Code Snippet shows adding an event listener using a lambda expression.

```
public class StateTest {
 public static void main(String args[]) {
 StateTest objStateTest = new StateTest();
 objStateTest.addStateListener((previousState,
 presentState) ->
 System.out.println("State change event occurred"));
 }
}
```

Explain about single method interface and lambdas. Explain the code that shows an example of a single method interface.

Explain that:

The method `onStateChange` is called if a state change occurs. `previousState` and `presentState` are parameters of the event.

Using slide 9, explain the code that shows how to add an event listener using an anonymous method implementation.

Explain that:

In the code, `objStateTest.addStateListener(new StateChangeListener()` is the event listener that is added to capture each state change event, `previousState` and `presentState`.

Using slide 10, explain the code that shows how to add an event listener using a lambda expression.

Explain that:

In the code, `objStateTest.addStateListener((previousState, presentState) → System.out.println("State change event occurred"));` is the event listener that is added using a lambda expression → to capture each state change event.

`previousState` and `presentState` are parameters of the event. The lambda expression and the parameter type of `addStateListener()` method have to match each other. If the type and expression is matched, lambda expression turns into a function which creates the same interface as a parameter. This interface contains a single method. Thus, the lambda expression is matched successfully.

## Slides 11 to 13

Let us understand lambda parameters and how to use them.

Lambda Parameters 1/3

| Types of Lambda Parameters | Description                  |
|----------------------------|------------------------------|
| ★ Zero Parameters          | Parentheses with no comments |
| ★ One Parameter            | Parentheses contains a value |

© Aptech Limited

Functional Programming Java - Day 05 / Slide 11

**Lambda Parameters 2/3**

| Types of Lambda Parameters                                                   | Description                                                                       |
|------------------------------------------------------------------------------|-----------------------------------------------------------------------------------|
| ★ Multiple Parameters<br><br>( <i>param1</i> , <i>param2</i> ) → <i>body</i> | Added within parentheses<br><br>Compiler is inconclusive about the parameter type |

© Aptech  
Functional Programming Java - Session 12 / Slide 12

**Lambda Parameters 3/3**

Following Code Snippets show different types of parameters for lambdas.

```

Zero Parameter

() → System.out.println("Zero parameter lambda");

One Parameter

(param) → System.out.println("One parameter: " + param); //with parentheses

Multiple Parameters

(pA, pB) → System.out.println("Multiple parameters defined: " + pA + ", " + pB);

Parameter Types

(Phone smartphone) → System.out.println("The smartphone is: " + smartphone.getName());

```

© Aptech  
Functional Programming Java - Session 12 / Slide 13

Using slide 11, explain about lambda parameters.

Explain that:

Lambda expressions can handle parameters similar to methods. Here, the method takes one parameter, *Seat s* and returns True if seat is available.

CheckSeat interface:

```
interface CheckSeat{
 boolean test(Seat s);
}
```

Using slide 12, explain about types of lambda parameters.

Explain that:

On the single method interface, it is must that the lambda expression parameters match the parameters of the method. At least the number of parameters in the lambda expression and the method must match.

Using slide 13, explain the codes that shows usage of different types of parameters for lambda.

Explain that: Multiple line lambda expressions must be enclosed within brackets { }, while single parameter does not needs a bracket.

#### **Additional Information:**

Refer the link for more information:

<https://dzone.com/articles/java-lambda-expressions-basics>

#### **Slide 14**

Let us look at how to return values from a lambda expression.

The slide has a blue header bar with the title "Returning a Value". The main content area is light blue with a dark blue callout box containing Java code. The code defines a lambda expression that prints a message and returns a value. The footer contains copyright and session information.

For returning values from lambda expression, a `return` statement can be added for specific calculations.

```
(pA) → {
 System.out.println("The output will be: " + pA);
 return "result value"; // return statement
}
```

© Aptech Ltd.      Fundamental Programming in Java -Session 15 / Slide 14

Using this slide, explain the procedure for returning values from Java lambda expressions.

Explain that:

For returning values from lambda expression, a return statement can be added for specific calculations. You add a return statement to the body of the lambda function. The return statement is enclosed within the braces ({ }). In the code, the compiler evaluates the expression `pA` and then returns its value.

#### **Additional Information:**

Refer the link for more information:

<http://tutorials.jenkov.com/java/lambda-expressions.html>

## Slide 15

Let us understand use of lambdas as objects.

**Lambdas as Objects**

**Java lambda is an object and it can be assigned as a regular object to a variable.**

```
public interface SampleComparator {
 public boolean compare(int iA, int iB);
}
```

**Implementation of lambda where the lambda object is assigned to a variable and passed as an object.**

```
SampleComparator sampleComparator = (iA, iB) ->
 return iA>iB;
boolean result = sampleComparator.compare(3, 6);
```

**Code Snippet shows a lambda used to sort strings by length.**

```
Arrays.sort(sampleStrArr,
 (String strA, String strB) -> strB.length() -
 strA.length());
```

For Aptech Centre Only

Functional Programming Java - 100% OOPS / 40 hrs - 15

Using this slide, explain about using lambdas as objects.

Explain that:

A Java lambda expression is basically an object. The code shows how a Java lambda expression can be assigned as a regular object to a variable. Here, `iA` and `iB` are the objects to be compared. The second code shows implementation of lambda where the lambda object is assigned to a variable and passed as an object. The values 3 and 6 are assigned to `iA` and `iB` and then compared. In the last code, a lambda is used to sort strings by length. Here, `Arrays.sort` method is used to sort the strings `strA` and `strB` by its length.

### Additional Information:

Refer the link for more information:

<http://tutorials.jenkov.com/java/lambda-expressions.html>

**Slides 16 to 18**

Let us explore advantages and uses of lambda expressions.

### Advantages and Uses of Lambda Expressions 1/3

**More readable code**

**Rapid fast coding specifically in Collections**

**Much easier parallel processing**

© Aptech Limited

Functional Programming, Java 8 & beyond / Slide 16

### Advantages and Uses of Lambda Expressions 2/3

**Following Code Snippet shows a complete program utilizing lambda:**

```

public class SampleLambda {
 public static void main(String args[])
 {
 SampleLambda perform = new SampleLambda();
 //to receive results with type declaration
 MathOperation add = (int ab, int xy) -> ab + xy;
 //to receive results without type declaration
 MathOperation subtr = (ab, xy) -> ab - xy;
 // to receive results with return statement along with curly braces
 MathOperation multi = (int ab, int xy) -> { return ab * xy; };
 // to receive results without return statement and curly braces
 MathOperation div = (int ab, int xy) -> ab / xy;
 System.out.println("Addition operation with Type declaration : 20 + 5 = " + perform.operate(20, 10, add));
 System.out.println("Subtraction operation without Type declaration: 20 - 5 = " + perform.operate(20, 10, subtr));
 System.out.println("Multiplication with return statement : 20 x 5 = " + perform.operate(20, 10, multi));
 System.out.println("Division operation without return statement : 20 / 5 = " + perform.operate(20, 10, div));
 }
 interface MathOperation {
 int operate(int ab, int xy);
 }
 private int operate(int ab, int xy, MathOperation mathOperation)
 {
 return mathOperation.operate(ab, xy);
 }
}

```

© Aptech Limited

Functional Programming, Java 8 & beyond / Slide 16

### Advantages and Uses of Lambda Expressions 3/3

**Following is the output for complete program utilizing lambda:**

**Output:**

```

Addition operation with Type declaration : 20 + 5 = 20
Subtraction operation without Type declaration: 20 - 5 = 10
Multiplication with return statement : 20 x 5 = 200
Division operation without return statement : 20 / 5 = 2

Here, the code performs basic math operations using lambda expressions.

```

© Aptech Limited

Functional Programming, Java 8 & beyond / Slide 16

Using slide 16, explain the advantages and uses of Java lambda expressions.

Explain that:

Lambda expressions limits lengthy codes thus, assists in faster execution of code. It also makes parallel processing easier.

Using slide 17, explain the code that shows a complete program utilizing lambda.

Explain that:

In the code, various types of lambda expressions are used to define the operation method of MathOperation interface. They are:

- Receive results with type declaration (addition)
- Receive results without type declaration (subtraction)
- Receive results with return statement along with curly braces (multiplication)
- Receive results without return statement and curly braces (division)

Using slide 18, explain the output of the code that shows a complete program utilizing lambda.

Explain that:

The code performs basic math operations using lambda expressions. The input values are taken as ab=20 and xy=5.

#### **Additional Information:**

Refer the link for more information:

<http://www.nagarro.com/at/de/perspectives/post/26/Lambda-Expressions-in-Java-8-Why-and-How-to-Use-Them>

## Slide 19

Let us understand scope for lambda expressions.

Scope for Lambda Expressions

Code Snippet uses lambda expressions with Runnable Interface:

```
import static java.lang.System.out;
/**Aptech Java8
 Scope of Lambda example/
public class MyWishes {
 Runnable dA = () -> out.println(this);
 Runnable dB = () -> out.println(toString());
 public String toString() { return "Happy New Year!"; }
 public static void main(String args[]) {
 new MyWishes().dA.run(); //Happy New Year
 new MyWishes().dB.run(); //Happy New Year
 }
}
```

Both the `dA` and `dB` lambdas call the `toString()` method of the `MyWishes` class.  
This shows the scope available to the lambda.

Using this slide, explain the code that shows usage of lambda expressions with Runnable interface.

Explain that:

In the code, both the `dA` and `dB` lambdas call the `toString()` method of the `MyWishes` class. Both lambdas use the `run` method and return Happy New Year as output.

**Additional Information:**

Refer the link for more information:

<http://www.codejava.net/java-core/the-java-language/java-8-lambda Runnable-example>

**Slides 20 to 23**

Let us examine method references.

**Method References 1/4**

Refers to constructors or methods.

Following are six types of method references:

|                                       |                                                                              |
|---------------------------------------|------------------------------------------------------------------------------|
| <code>TypeName::static</code>         | Refers to a static method of a class, an <code>enum</code> , or an interface |
| <code>TypeName.super::instance</code> | Refers to an instance method from the supertype of an object                 |
| <code>ObjectRef::instance</code>      | Refers to an instance method                                                 |
| <code>ClassName::instance</code>      | Refers to an instance method of a class                                      |
| <code>ClassName::new</code>           | Refers to the constructor from a class                                       |
| <code>ArrayTypeName::new</code>       | Refers to the constructor of the specified array type                        |

© Aptech

**Method References 2/4**

Methods to determine a file type

Frequent filter of a list of files based on file types can be made by determining a file type.

```
public class FileFilters { // to filter files
 public static boolean fileIsJpeg(File file)
 /*sample code*/
 public static boolean fileIsTiff(File file)
 /*sample code*/
 public static boolean fileIsPng(File file)
 /*sample code*/
}
```

© Aptech

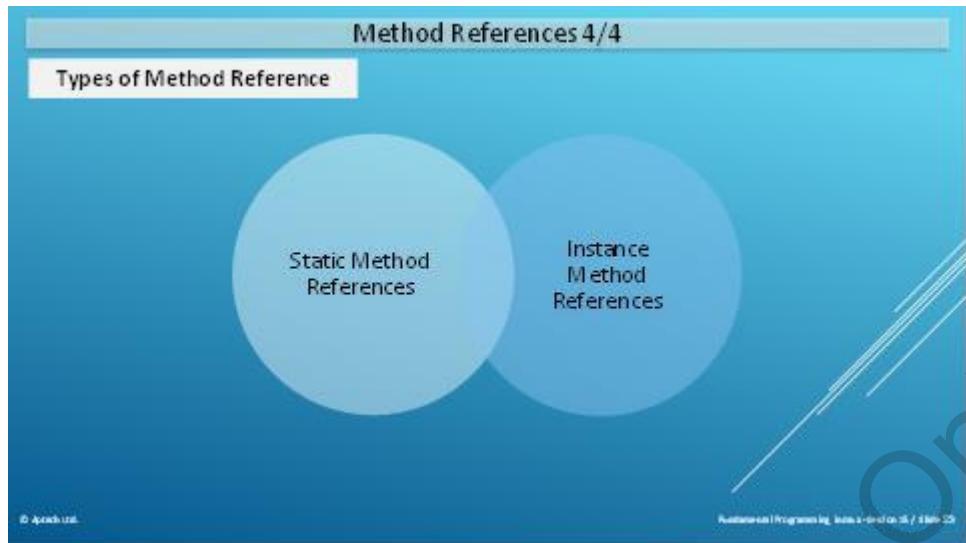
**Method References 3/4**

File Filtering Case

Method reference can be useful in file filtering cases.  
In the following CodeSnippet, a method is predefined as `getFiles()` that returns a Stream:

```
Stream<File> Jpegs =
 getFiles().filter(FileFilters::fileIsJpeg);
Stream<File> Tiffs =
 getFiles().filter(FileFilters::fileIsTiff);
Stream<File> Pngs =
 getFiles().filter(FileFilters::fileIsPng);
```

© Aptech



Using slide 20, explain about method references.

Explain that:

In Java 8, method references are used to refer to constructors or methods without performing an execution. There are six types of method references as shown on slide 20.

Then explain:

**Reference to static methods:** In the syntax, `ReferenceStatic::demoMethod` `ReferenceStatic` is the class in which static method `demoMethod` is defined.

**Reference to instance from the supertype of an object:** In the syntax, `demoInstance::demoMethod`

Here, `demoInstance` is an object reference for `ReferenceStatic` class and `demoMethod` is instance method defined in class.

**Reference to a constructor `ClassName::new`:** In the syntax, `String:: new` is a reference to the `String` class constructor.

Using slide 21, explain the code that shows a set of methods to determine a file type.

Explain that:

In the code, the class `File` implements the interface `FileFilters` and to determine the filenames with extensions such as jpeg, tiff, and png.

Using slide 22, explain the code that shows how to use method reference for file filtering.

Explain that:

In the code, a method is predefined as `getFiles()` that returns a `Stream`. The method reference `FileFilters::fileIsJpeg`, `FileFilters::fileIsTiff`, and `FileFilters::fileIsPng` is a reference to a static method.

Explain about the various types of method references using slide 23.

Explain that:

Method references simply call a method which is already defined. They refer to the methods by the method name. In a lambda expression, double colon(:) operator is used for method reference.

A static method reference refers to a static method of the specified class.

```
public class Author {
 static String DemoField1;
}
Author.DemoField1 = "value";
System.out.println(Author.DemoField1);
```

Static fields belong to the class `Author`, hence, an instance of the class is not required to access static fields.

## Slide 24

Let us understand static method references.

**Static Method References**

Static methods can be defined in an enum, a class, or an interface.

```
import java.util.function.Function;
public class MainTest {
 public static void main(String[] args) {
 // To retrieve result with a lambda expression
 Function<Integer, String> funcA = x -> Integer.toBinaryString(x);
 System.out.println(funcA.apply(11));
 // To retrieve result with a method reference
 Function<Integer, String> funcB =
 Integer::toBinaryString;
 System.out.println(funcB.apply(11));
 }
}
```

**Output:**

```
1011
1011
```

The first lambda expression `funcA` is created by defining an input value `x` and providing a lambda expression body. This is the normal way of creating a lambda expression.  
The second lambda expression `funcB` is created by referencing a static method from `Integer` class.

Explain the code on the slide that demonstrates static method reference.

Explain that:

Static methods can be defined in an enum, a class, or an interface. The first lambda expression `funcA` is created by defining an input value `x` and providing a lambda expression body. This is the normal way of creating a lambda expression.

The second lambda expression `funcB` is created by referencing a static method from `Integer` class.

**Additional Information:**

Refer the link for more information:

<http://java8.in/java-8-method-references/>

**Slide 25**

Let us understand instant method references.

The screenshot shows a Java code editor window titled "Instance Method References". The code is as follows:

```
import java.util.function.Supplier;
public class MainTest{
public static void main(String[] args){
Supplier<Integer>sampleSupA = () ->
"Aptech".length();
System.out.println(sampleSupA.get());
// display result
Supplier<Integer>sampleSupB=
"Aptech)::length;
System.out.println(sampleSupB.get());
// display result
}
}
```

To the right of the code, there is a yellow box labeled "Output" containing the text "6" and "6".

At the bottom left of the slide, it says "© Aptech Ltd." and at the bottom right, it says "Fundamental Programming in Java -Session 15 / Slide 25".

Using this slide, explain the code that demonstrates an instance method reference.

Explain that:

In the code, lambda expressions sampleSupA and SampleSupB gets the length of the input value Aptech. Both produce the output as 6.

**Additional Information:**

Refer the link for more information:

<http://baddotrobot.com/blog/2014/02/18/method-references-in-java8/>

## Slides 26 and 27

Let us understand functional interfaces.

**Functional Interface 1/2**

A functional interface is an interface with one method and is used as the type of a lambda expression.

New functional interfaces included in the Java 8 package, `java.util.function`, are:

- `Predicate<T>` - Returns a Boolean value based on input of type T.
- `Supplier<T>` - Returns an object of type T.
- `Consumer<T>` - Performs an action with given object of type T.
- `Function<T, R>` - Gets an object of type T and returns R.
- `BiFunction` - Similar to Function but with two parameters.
- `BiConsumer` - Similar to Consumer but with two parameters.

© Aptech Ltd. Fundamental Programming in Java -Session 15 / Slide 26

**Functional Interface 2/2**

The second line defines a function that represents '@' symbol to a String.

```
//Functional Interface sample use case
Function<String, Integer>sampleLengthA = (name) ->
 name.length(); //as int
Function<String, String>atr = (name) -> {return "@" + name;}; //as string
Function<String, Integer>sampleLengthB = String::length;
//as int
```

© Aptech Ltd. Fundamental Programming in Java -Session 15 / Slide 27

Explain about functional interface and the new functional interfaces included in `java.util.function`.

Explain that:

A functional interface is with one method and is used as a type of a lambda expression. A functional interface type with an abstract method takes a String as a parameter and returns an integer value.

Predicate interface creates lambda expressions that returns a boolean value based on the input argument.

Supplier<T> returns a supplier of results.

Consumer<T> accepts a single input argument and returns no result.

Function<T, R> represents a function that accepts one argument of type T and produces a result of type R.

**Additional Information:**

Refer the link for more information

<https://dzone.com/articles/function-interface-functional>

Using slide 27, explain the code that shows sample functional interface use case.

Explain that:

Functional interface can process any type of parameter and return another type of parameter. For example, String to Integer or String to String. In the code, the line `Function<String, String>atr = (name) → {return "@" + name;};` defines a function that represents @ symbol to a String.

Both the lines,

`Function<String, Integer>sampleLengthA = (name) → name.length();`

and

`Function<String, Integer>sampleLengthB = String::length;`  
define the same process of calculating a length of a string, but using different approaches.

**Slide 28**

Let us look at default methods.

**Default Methods**

Default methods are Virtual Extensions that contains default implementations of `forEach()` method.

```

class Test{
 public static void main(String args[]){
 Book book = new Novel();
 book.print();
 }
}
interface Book {
 default void print(){
 System.out.println("This is a book");
 }
 static void turnPages(){
 System.out.println("Turning pages.");
 }
}
interface Journal {
 default void print(){
 System.out.println("This is a journal");
 }
}
class Novel implements Book, Journal {
 public void print(){
 Book.super.print();
 Journal.super.print();
 Book.turnPages();
 System.out.println("This is a novel");
 }
}

```

© Aptech Ltd.

Fundamental Programming in Java -Session 15 / Slide 28

Explain the code that shows usage of default methods.

Explain that:

Default methods allow implementation for methods in an interface. Default methods are also known as Virtual Extensions. Classes implementing an interface automatically inherits the default implementation. The class Novel implementing Book and Journal interface contains default implementations of `print()` method.

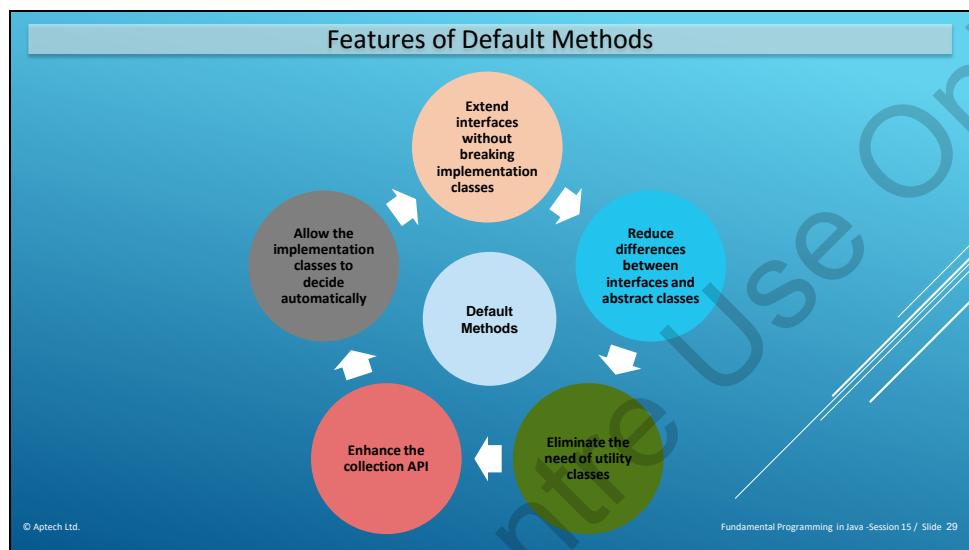
**Additional Information:**

Refer the link for more information

<https://dzone.com/articles/interface-default-methods-java>

**Slide 29**

Let us explore the features of default methods.



Explain the features of default methods.

Explain that:

Default methods help to include new functionalities to interfaces without breaking the implementation classes that implements that interface. A default method cannot override a method from `java.lang.Object` as the `Object` is the base class for all the other Java classes.

```
public interface TestInterface {
 void defaultmethod1(String str);
 default void show(String str){
 System.out.println("This is the string"+str);
 }
}
```

`show(String str)` is the default method in the `TestInterface`. When a class implements `TestInterface`, it is not required to provide implementation for default methods of interface.

**Slide 30**

Let us view the difference between default methods and regular methods.

The screenshot shows a Java code editor with a title bar "Default Method and Regular Method". Below the title, a note states: "Default method contains default modifier - that is the main difference between a regular method and default method." The code is as follows:

```

public class Java8Tester {
 public static void main(String args[]){
 Gadget gadget = new SmartGadget();
 gadget.print();
 }
}
interface Gadget {
 default void print(){
 System.out.println("This is a Gadget!");
 }
 static void call(){
 System.out.println("With Calling feature!");
 }
}
interface TextMessage {
 default void print(){
 System.out.println("With Text Messaging feature!");
 }
}
class SmartGadget implements Gadget, TextMessage {
 public void print(){
 Gadget.super.print();
 TextMessage.super.print();
 Gadget.call();
 System.out.println("It is a Smartphone!");
 }
}

```

**Output:**

```

This is a Gadget!
With calling feature!
With Text Messaging
feature!
It is a Smartphone!

```

© Aptech Ltd.

Fundamental Programming in Java -Session 15 / Slide 30

Using this slide, explain the code that shows the difference between default methods and regular methods.

Explain that:

The main difference between a regular method and default method is default method contains default modifier. Methods in classes can access and modify method arguments and fields of their class.

In the code sample, the same `print()` method gives different output based on the modified method arguments for each of their class.

**Additional Information:**

Refer the link for more information:

<http://stackoverflow.com/questions/27833168/difference-between-static-and-default-methods-in-interface>

## Slides 31 to 33

Let us understand the concept of multiple interfaces.

### Multiple Defaults 1/3

Multiple Defaults are multiple interfaces contained within Java class. Java throws a compilation error if two or more interfaces defining the same default method.

```
public interface Green {
 default void defaultMethod() {
 System.out.println("Green default method");
 }
}
public interface Red {
 default void defaultMethod() {
 System.out.println("Red default method");
 }
}
public class Impl implements Green, Red{}
```

© Aptech Ltd. Fundamental Programming in Java -Session 15 / Slide 31

### Multiple Defaults 2/3

Compiling the previous Code Snippet will result in an error. In order to fix this, provide explicit default method implementation.

```
public class Impl implements Green, Red{
 public void defaultMethod() {
 ...
 }
}
```

© Aptech Ltd. Fundamental Programming in Java -Session 15 / Slide 32

### Multiple Defaults 3/3

Further, to invoke default implementation provided by any of super interfaces, the code can be as follows:

```
public class Impl implements Green, Red {
 public void defaultMethod() {
 // remaining code...
 Green.super.defaultMethod();
 }
}
```

© Aptech Ltd. Fundamental Programming in Java -Session 15 / Slide 33

Explain about multiple interfaces and the code that shows implementation of multiple methods.

Explain that:

Multiple Defaults are multiple interfaces contained within Java class. The compiler will throw a compilation error if two or more interfaces define the same default method. In the code sample, the inherited methods, Green and Red conflicts with one another. On execution, the compiler shows error as ‘java: class Impl inherits unrelated defaults for defaultMethod() from types Green and Red.’

Using slide 32, explain how the error in the previously shown code can be rectified.

Explain that:

To fix the error, provide explicit default method implementation. The default method is defined within public class `impl`.

Using slide 33, explain how to invoke default implementation.

Explain that:

The code is revised as given to invoke default implementation by revising the default method Green as super interface.

#### **Additional Information:**

Refer the link for more information:

[https://www.tutorialspoint.com/java8/java8\\_default\\_methods.htm](https://www.tutorialspoint.com/java8/java8_default_methods.htm)

#### **Slides 34 and 35**

Let us understand static methods on interfaces.

**Static Methods on Interfaces 1/2**

| <b>Static Methods on Interfaces</b>                                                                                                                           | <b>Description</b>                                                                                            |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"> <li>★ Easy to organize and access helper method in libraries</li> <li>★ Eliminates the need of a separate class</li> </ul> | <p>For example, the new Stream interface contains many static methods</p> <p>Parentheses contains a value</p> |

© Aptech Ltd.

Fundamental Programming in Java -Session 15 / Slide 34

Static Methods on Interfaces 2/2

All method declarations in an interface, including static methods, are implicitly `public`.

```
public interface ProductInfo {
 ...
 static ProductId getProductId (String
 ProductString) {
 ...
 }
 ...
}
```

© Aptech Ltd.

Fundamental Programming in Java -Session 15 / Slide 35

Explain about static methods on interfaces.

Explain using slide 34 that:

A static method is a method that associates itself with the class in which it is defined. Each instance of the class shares the static method of the class. In Java 8, Static methods can be defined in interfaces to help default methods. Thus, static methods help to organize helper methods in libraries. Static methods can be kept specific to an interface in the same interface instead of a separate class.

Using slide 35, explain the code that shows example of implementation of static methods on interfaces.

Explain that:

`static` keyword is used to mark a method as static. The keyword is mentioned at the beginning of the method signature. As static methods are implicitly `public`, it is not required to mention the `public` modifier explicitly in the declaration.

#### In-Class Question:

After you finish explaining slide 34, you will ask the students some In-Class questions. This will help you in reviewing their understanding of the topic.



Curly braces may or may not be present when declaring lambda expression. Is it true or false?

**Answer:** True



Returning the values from lambda expression return statement added to the body of lambda function and the statement is enclosed within which brackets?

**Answer:** The return statement added to the body of lambda function and the statement is enclosed within curly braces ({}).



What will be the result of operation, if the operation() method of MathOperation interface in lambda expression receives results without return statement and curly braces?

**Answer:** The result of operation will be division.



What will be the output for the following program, if the program runs without error?

```
import java.util.function.Supplier;
public class MainTest{
public static void main(String[] argv) {
Supplier<Integer>sampleSupA= ()-> "Employee".length();
System.out.println(sampleSupA.get());
Supplier<Integer>sampleSupB="Employee":length;
System.out.println(sampleSupB.get());
}
}
```

**Answer:**

8

8

**Slide 36**

Let us summarize the session.

**Summary**

- ❖ Functional programming emphasizes that utilization of functions and writing code that does not change state.
- ❖ Using functional programming, you can pass functions as parameters to other functions and return them as values.
- ❖ A lambda expression is a compact expression that does not require a separate class/function definition. It facilitates functional programming.
- ❖ Depending on the parameters being passed to the lambda expression, you will use/omit parentheses.
- ❖ Default method is a new feature in Java 8 that allows default implementation for methods in an interface.
- ❖ In addition to default methods, static methods can be defined in interfaces that makes it easy to organize and access helper methods in libraries.



© Aptech Ltd.

Fundamental Programming In Java -Session 15 / Slide 36

Use this slide to summarize the session. You will end the session, with a brief summary of what has been taught in the session. Tell the students pointers of the session. This will be a revision of the current session.

Explain that:

Functional programming emphasizes that utilization of functions and writing code that does not change state. Using functional programming, you can pass functions as parameters to other functions and return them as values. A lambda expression is a compact expression that does not require a separate class/function definition. It facilitates functional programming. Depending on the parameters being passed to the lambda expression, you will use/omit parentheses. Default method is a new feature in Java 8 that allows default implementation for methods in an interface. In addition to default methods, static methods can be defined in interfaces that makes it easy to organize and access helper methods in libraries.

### 15.3 Post Class Activities for Faculty

You should familiarize yourself with the topics of the next session.

#### Tips:

You can also check the Articles/Blogs/Expert Videos uploaded on the Online Varsity site to gain additional information related to the topics covered in the next session. You can also connect to online tutors on the Online Varsity site to ask queries related to the sessions.

# Session 16 – Stream API

## 16.1 Pre-Class Activities

Before you commence the session, you should familiarize yourself with the topics of this session in-depth. Prepare a question or two that will be a key point to relate the current session objectives.

### 16.1.1 Objectives

By the end of this session, learners will be able to:

- Describe the Stream API
- Outline the differences between collections and streams
- Explain the classes and interfaces in Stream API
- Describe how to use functional interfaces with Stream API
- Describe the Optional class and Spliterator interface
- Explain stream operations
- Discuss the limitations of Stream API

### 16.1.2 Teaching Skills

To teach this session, you should be well versed with the new Stream API in Java 8. You should be aware of the difference between collections and streams. You must be familiar with functional interfaces in the Stream API and operations performed by them.

You should teach the concepts in the theory class using the images provided. For teaching in the class, you are expected to use slides and LCD projectors.

#### Tips:

It is recommended that you test the understanding of the students by asking questions in between the class.

#### In-Class Activities

Follow the order given here during In-Class activities.

#### Overview of the Session

Give the students an overview of the current session in the form of session objectives. Read out the objectives on slide 2.

**Slide 2**

Objectives

- ❖ Describe the Stream API
- ❖ Outline the differences between collections and streams
- ❖ Explain the classes and interfaces in Stream API
- ❖ Describe how to use functional interfaces with Stream API
- ❖ Describe the Optional class and Spliterator interface
- ❖ Explain stream operations
- ❖ Discuss the limitations of Stream API



Fundamental Programming in Java 4 module 6 / Slide 2

Show the slide and give the students a brief overview of the current session in the form of session objectives. Tell the students that this session explains about the new Stream API in Java 8. The session will describe the differences between collections and streams and the classes and interfaces in Stream API. Then, it describes how to use functional interfaces with Stream API and concentrates on various operations supported by the Stream. Finally, the session will discuss the limitations of Stream API.

### 16.2 In-Class Explanations

#### Slides 3 and 4

Let us look at an overview of the Stream API.

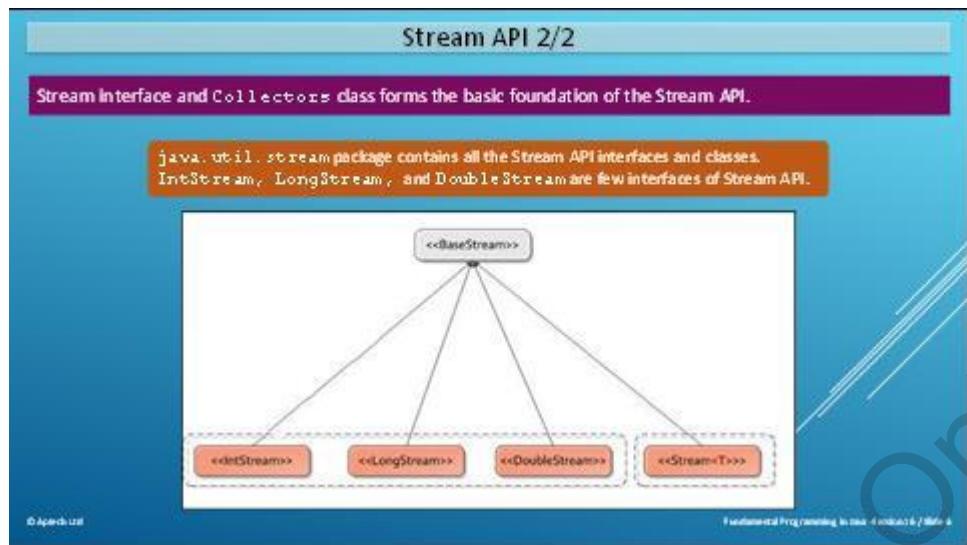


Stream API 1/2

Stream API is a notable Java 8 inclusion that allows parallel processing and helps to express efficient, SQL-like queries and manipulations on data.



Fundamental Programming in Java 4 module 6 / Slide 3



Explain about Stream API.

Using slide 3, explain that:

Stream API is a new abstract layer introduced in Java 8 that supports parallel processing and helps to express efficient, SQL-like queries and manipulations on data. Also, Stream API supports sequential and parallel aggregate operations for data processing. Sequential operations must be performed in a specific order. Aggregate operations process elements from a stream and allows you to customize the behavior of a particular aggregate operation. That means, dividing a process into sub processes, executing the sub processes simultaneously, and then finally combining the results into one solution.

Stream API can also use lambda expressions.

Using slide 4, explain about Stream interface and Collectors class.

Stream interface and Collectors class form the basic foundation of the Stream API.

`java.util.stream` package contains all the Stream API interfaces and classes. Few interfaces of Stream API are `IntStream`, `LongStream`, `DoubleStream`, `Stream<T>`, and so on.

## Slide 5

Let us understand streams and collections and the differences between them.

| Collections and Streams                                                                                  |                                                           |
|----------------------------------------------------------------------------------------------------------|-----------------------------------------------------------|
| <b>A stream is a series or set of elements that support sequential and parallel aggregate operations</b> |                                                           |
| <b>A collection is a set of data in the form of objects or elements</b>                                  |                                                           |
| Major differences between Streams and Collections are:                                                   |                                                           |
| Streams                                                                                                  | Collections                                               |
| Fixed structures computed on-demand                                                                      | In-memory data structure to store values                  |
| Operates on user demand basis                                                                            | Focus on holding data                                     |
| Data storage not available                                                                               | Collections are actual data structures                    |
| Supports pipelining                                                                                      | May not support pipelining                                |
| Do not iterate                                                                                           | Iterate explicitly                                        |
| Functional interface friendly with slow processing time                                                  | Functional Interface friendly with faster processing time |

Describe collections and using the content on this slide and differentiate between streams and collections.

Explain that:

A stream is a series or set of elements that supports sequential and parallel aggregate operations and a collection is an in-memory data structure. A collection is a set of data in the form of objects or elements which holds data and before we start using collection. That is, all the values should have been populated.

Stream is a data structure that is operated on user demand. It does not store data. It operates on the source data structure (collection and array) and produces ordered data. You can create a stream from a list and filter it based on a condition.

The similarity between collections and streams is that both allow you to traverse them only once. If you use an iterator to traverse a list, you will have to create a new iterator to loop through it again.

## Slides 6 to 11

Let us explore the process of generating a stream.

**Generating Streams 1/6**

There are many options available to generate a Stream in Java 8.

`stream(): Is used to get a sequential Stream with the collection as its source.`

**Code Snippet shows the usage of stream().**

```
Stream<String> str = list.stream();
```

`parallelStream(): Is used to get a possibly-parallel Stream lateral to the collection given as its source.`

**Code Snippet shows an example.**

```
Stream parStr = list.parallelStream();
```

© Aptech Fundamental Programming in Java 4 module 6 / Slide 6

**Generating Streams 2/6**

BufferedReader class of java.io package includes the lines() method that returns a Stream, as shown in Code Snippet.

```
try (FileReader sampleFR = new FileReader ("D:\\random_file.txt");
BufferedReader sampleBR = new BufferedReader(sampleFR))
{
 sampleBR.lines().forEach(System.out::println);
}
```

Here, SampleBR is created as a BufferedReader instance that uses lines() method for a simple stream operation- reading and displaying data from a text file.

© Aptech Fundamental Programming in Java 4 module 6 / Slide 7

**Generating Streams 3/6**

**Code Snippet shows how to read a file as a java.util.stream.Stream object using Files.lines(Path filePath).**

```
try (Stream sampleST = Files.lines(Paths.get("D:\\random_file.txt")))
{
 sampleST.forEach(System.out::println);
}
```

© Aptech Fundamental Programming in Java 4 module 6 / Slide 8

### Generating Streams 4/6

Static methods on the `Files` class assist in navigating file trees using a Stream. Some of these are listed in the table.

| Method                                                                                       | Explanation                                                                                                                                                                          |
|----------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>static Stream&lt;Path&gt; list(Path dir)</code>                                        | Retrieves a Stream whose elements include files in the specific directory.                                                                                                           |
| <code>static Stream&lt;Path&gt; walk(Path dir, FileVisitOption options)</code>               | Retrieves a Stream that is created by traversing the file tree starting at a specific file. <code>FileVisitOption</code> is an enumeration that defines file tree traversal options. |
| <code>static Stream&lt;Path&gt; walk(Path dir, int maxDepth, FileVisitOption options)</code> | Retrieves a Stream that is created by traversing the file tree depth-first starting at a specific file.                                                                              |

© Aptech      Fundamental Programming in Java - 4th Edition | Chapter 10

### Generating Streams 5/6

Text patterns can be streamed using the `Pattern` class that contains a method, `splitAsStream(CharSequence)` to generate a stream.

```
import java.util.regex.Pattern // to use Pattern class
public class TextPatterns
{
 public static void main(String args[])
 {
 // Creating a pattern
 Pattern createPatt = Pattern.compile(",") // adding a comma
 // to pass a set of names
 createPatt.splitAsStream("Nathan, Ethan, Hank, Dennis, Sarah")
 .forEach(System.out::println) //
 }
} // result as stream
```

© Aptech      Fundamental Programming in Java - 4th Edition | Chapter 10

### Generating Streams 6/6

- The example in previous slide generates a Stream from a simple text pattern that contains a comma as separator and separates the text into a Stream by using the `splitAsStream()` method.
- Then, each element in the Stream is printed out using a `forEach` loop. In practical scenarios, a similar code to match and display huge collections of strings can be used.

**Output:**

```
Nathan
Ethan
Hank
Dennis
Sarah
```

© Aptech      Fundamental Programming in Java - 4th Edition | Chapter 10

Explain how to generate streams.

Explain that:

`java.util.Collection` must be inherited to generate a stream from a data source.

Here, in the code snippet shown on slide 6, a list of strings is created.

The statement

```
Stream <String> stream = Stream.of("Apple", "Mango", "Orange",
"Guava");
```

shows creating a stream with four objects.

Streams can be sequential or parallel. Operations on sequential stream are processed in serial manner by a single thread and operations on a parallel stream are processed in parallel using multiple threads.

The `Collection` interface provides `stream()` and `parallelStream()` methods, which are inherited by all implementing classes and sub-interfaces.

Using slide 7, explain about `BufferedReader` class and the code snippet demonstrating its use.

Explain that:

`BufferedReader` class that is present in the `java.io` package contains a method `lines()` that returns a `Stream`. In the code snippet shown on slide 7, a `BufferedReader` instance is used with `FileReader` to read a text file, `random_file.txt`.

The statement

```
BufferedReader sampleBR = new BufferedReader(sampleFR) indicates a
BufferedReader class being wrapped around FileReader instance named
sampleFR. BufferedReader takes FileReader as input.
```

`SampleBR` is created as a `BufferedReader` instance. It makes use of `lines()` method to read and display data from the text file.

`BufferedReader` helps to read text from a character-input stream, buffering characters so that characters, arrays, and lines are read efficiently.

Using slide 8, explain the code snippet that shows how to read a file as a `java.util.stream.Stream` object using `Files.lines(Path filePath)`.

Explain that:

We can read text from a file as a stream of strings. Each element in the stream represents one line of text. The code reads the text file, `random_file.txt` using `lines()` and `forEach()` methods. The `Files.lines()` method returns `Stream` of all lines in the text file.

Using slide 8, explain about static methods in the `Files` class.

Explain that:

Static methods in the `Files` class assist in navigating file trees using a Stream. Files are used to store data and are organized in directories in a file system. Files can be accessed using streams, as streams present data in a sequential form.

`File` instance is created by passing a path string to the new `File()` constructor. The `File` class consists of various methods to retrieve information about files and directories. The table on slide 9 shows some of these methods.

Explain about `Pattern` class.

Explain that:

Text patterns can be streamed using the `Pattern` class that contains a method, `splitAsStream(CharSequence)` to generate a stream. The `splitAsStream(CharSequence)` splits a `CharSequence` by a regular expression (a special sequence of characters that helps you match or find other strings or sets of strings, using a specialized syntax held in a pattern).

In the code snippet shown on slide 10, in the class `TextPatterns`, a pattern object is created as `Pattern createPatt = Pattern.compile(",")` to add a comma to a set of names. The names are split as sequence of characters using `splitAsStream(CharSequence)`.

Using slide 11, explain the output of the code snippet that was shown on slide 10.

Explain that:

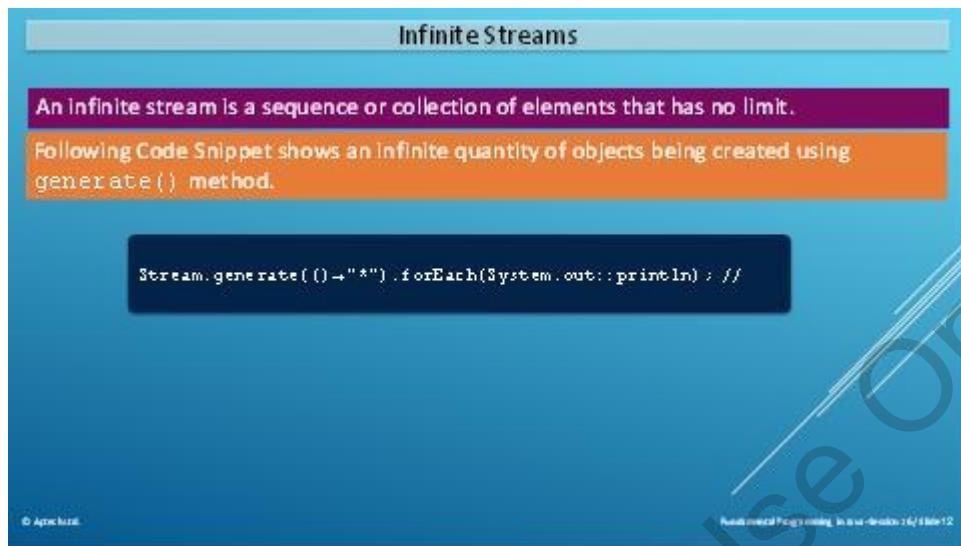
The example on slide 11 generates a Stream from a simple text pattern that contains a comma as separator and separates the text into a Stream by using the `splitAsStream()` method.

Then, each element in the Stream is printed out using a `forEach` loop. In practical scenarios, a similar code to match and display large collections of strings can be used.

Explain the output.

## Slide 12

Let us look at Infinite Streams.



Explain about infinite streams and also explain the code snippet that shows creation of infinite quantity of objects.

Explain that:

An infinite stream is a sequence or collection of elements that has no limit.

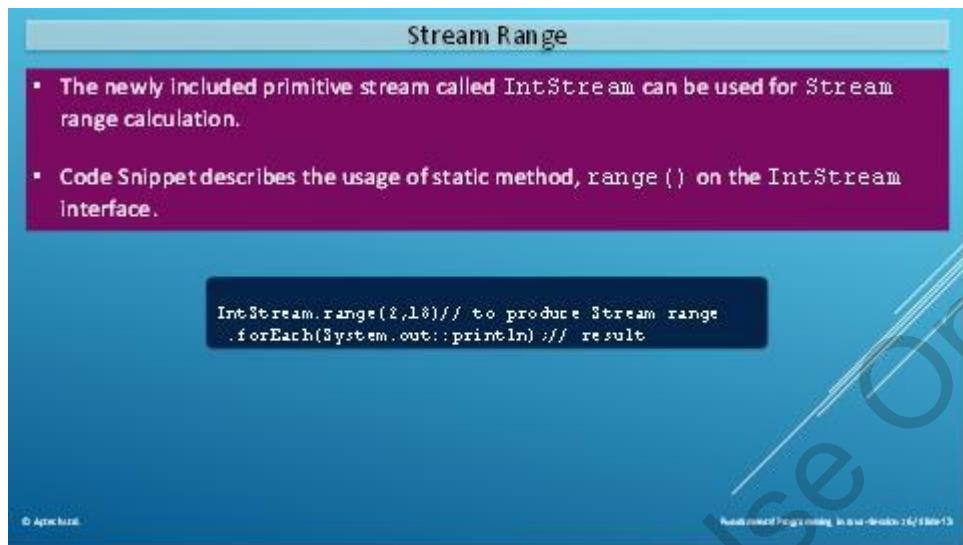
The code shown on this slide shows an infinite quantity of objects being created using generate() method. The generate() method invokes Supplier functional interface instance passed to it as input parameter.

You can also use the iterate() method to create infinite streams. For example,  
IntStream.iterate(0, i -> i + 2).limit(3);

Using iterator methods, you can define an initial value and a method modifies the value.

### Slide 13

Let us understand Stream range.



Explain about stream range and also explain the code snippet that describes the usage of range () on the IntStream interface.

Explain that:

The newly included primitive stream called IntStream can be used for Stream range calculations. This can be useful in practical scenarios involving large ranges of numbers that will be cumbersome to generate manually.

IntStream is a part of `java.util.stream` package.

Code snippet on slide 13 describes the usage of static method `range ()` in the `IntStream` interface. In the code snippet, the integers within the range of 2 to 18 are displayed.

You can define a range of integers to create a stream using `range ()` and `rangeClosed ()` methods.

The `rangeClosed ()` method includes the ending integer value while the `range` method excludes it.

You can also use the `of ()` method. For example, `IntStream.of (1, 2, 3);` creates a `IntStream` based on the values 1, 2, and 3.

Besides `IntStream`, other primitive Stream interfaces such as `DoubleStream` and `LongStream` also contain corresponding `range ()` methods.

## Slides 14 to 16

Let us explore operations that can be performed on streams.

Operations on Streams 1/3

**Intermediate Operations**

- In intermediate operations, operators (intermediate operators) apply logic thus, the inbound Stream generates another stream.
- A Stream can contain 'n' number of intermediate operators, which has no limitations.
- Intermediate operators can start a pipeline of Stream elements to execute the process further.

© Aptech Ltd. Fundamental Programming in Java -Session 16 / Slide 14

Operations on Streams 2/3

**Terminal Operations**

Following are commonly used terminal methods:

```

graph TD
 forEach[forEach] --- toArray[toArray]
 toArray --- min[min]
 min --- max[max]
 max --- findFirst[findFirst]
 findFirst --- findAny[findAny]
 findAny --- anyMatch[anyMatch]
 anyMatch --- allMatch[allMatch]
 allMatch --- noneMatch[noneMatch]

```

© Aptech Ltd. Fundamental Programming in Java -Session 16 / Slide 15

Operations on Streams 3/3

**Short-Circuiting Operations**

**Not Standalone Operations**

**Operation generating finite Stream from infinite Stream is defined as Short-circuiting**

© Aptech Ltd. Fundamental Programming in Java -Session 16 / Slide 16

Explain about intermediate operations.

Explain that:

Stream operations that can be connected are called intermediate operations. It is possible to connect stream operations as their return type is also Stream. In intermediate operations, operators (intermediate operators) apply logic, thus, the inbound Stream generates another stream. For example, filter and map operations can be connected to form a pipeline of stream.

A Stream can contain 'n' number of intermediate operators, which has no limitations.

Read out the commonly used terminal operations.

Explain that:

Terminal operations help to close a pipeline of stream. Intermediate operations do not perform any action until the terminal operation is invoked. Use the collect operation to collect the result of filter and map operations and return a result. Thus, collect operation serves as a terminal operation.

Explain about short-circuiting operations.

Explain that:

Short-circuiting operations are not standalone operations. Short-circuiting operations are defined as operations generating finite Stream from infinite Stream. For example, when you use the boolean operators || in an expression, if the left-side expression returns true, then the right-side expression is not checked.

### Slide 17

Let us understand map/filter/reduce operations with streams.

**Map/Filter/Reduce with Streams**

**Map/Filter/Reduce methods implementations are allowed in lambda expressions.**

**Map:**  
This method is applied for mapping all the elements to its output.

**Filter:**  
Choosing a set of element and eliminating other elements based on the instructions is the basic feature of Filter.

**Reduce:**  
Reduce method is applied to reduce the elements based on the given instructions.

```
String outcome = scores.stream()
 .reduce((acc, score) → acc + " " + score)
 .get();
```

© Aptech Limited

Java Advanced Programming - Java Stream API | Page 17

Explain about map/filter/reduce methods implementation in lambda expressions.

Explain that:

Stream supports operations from functional programming languages similar to SQL operations. For example, filter, map, reduce, find, and so on.

For example, you are checking your expenditure of credit card.

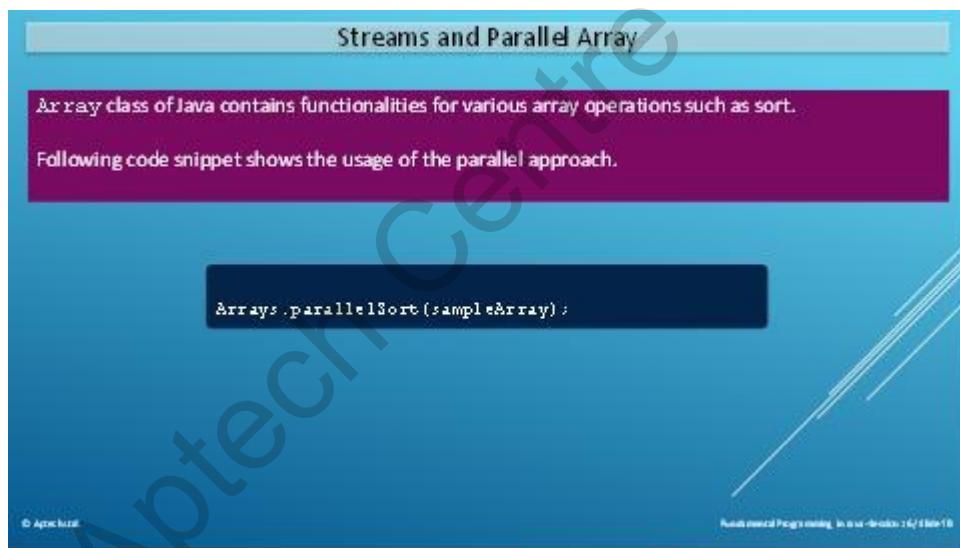
Map method is applied for mapping all the elements to its output. You extract information from the bill.

Filter helps in choosing a set of element and eliminating other elements based on the instructions. You apply filter to the mapping by giving a condition for example, filter by expenditure more than 500 USD.

Reduce method is applied to reduce the elements based on the given instructions. You further reduce the findings by calculating the sum of all transactions more than 500 USD.

### Slide 18

Let us understand streams and parallel arrays.



Explain about streams and parallel array. Also, explain the code snippet that shows usage of the parallel approach.

Explain that:

Java 8 consists of new functionality of arrays which is parallelSort. For example, `parallelSort(int[] a, int fromIndex, int toIndex)` helps to sort and arrange integers. `parallelSort()` method assigns the process of sorting to various threads existing in a thread pool.

**Slide 19**

Let us understand the limit() method of Stream.

**Limit**

To limit a Stream to a specified number of elements, limit() method can be applied.

```
Random sampleRand = new Random();
sampleRand.ints().limit(12)
 .forEach(System.out::println) // to display the results
```

Here, sampleRand is used to return random integer values and limit() method is applied to limit the numbers. The code is limited to display only 12 random numbers.

© Aptech Limited. All Rights Reserved. Programming In Java - Session 2 of 10 | Page 20

Explain about limit and explain the code snippet that shows usage of limit() method.

Explain that:

limit() method can be applied to limit a Stream to a specified number of elements. Here, sampleRand is used to return random integer values and limit() method is applied to limit the numbers to display only 12 random numbers.

**Slide 20**

Let us understand how to perform sorting using sorted() method.

**Sort**

sorted() method is another method within Stream API that helps to sort the Stream.

**'Lazy' execution**

- No process is started until a terminal operation (such as reduce or foreach) is called.
- A limiting operation must be called before the sorting operation on an infinite Stream.

```
sampleRand.ints().limit(12).sorted().limit before sort
 .forEach(System.out::println) // to display output
```

© Aptech Limited. All Rights Reserved. Programming In Java - Session 2 of 10 | Page 20

Explain about sorted() method and explain the code snippet that shows usage of sorted() method.

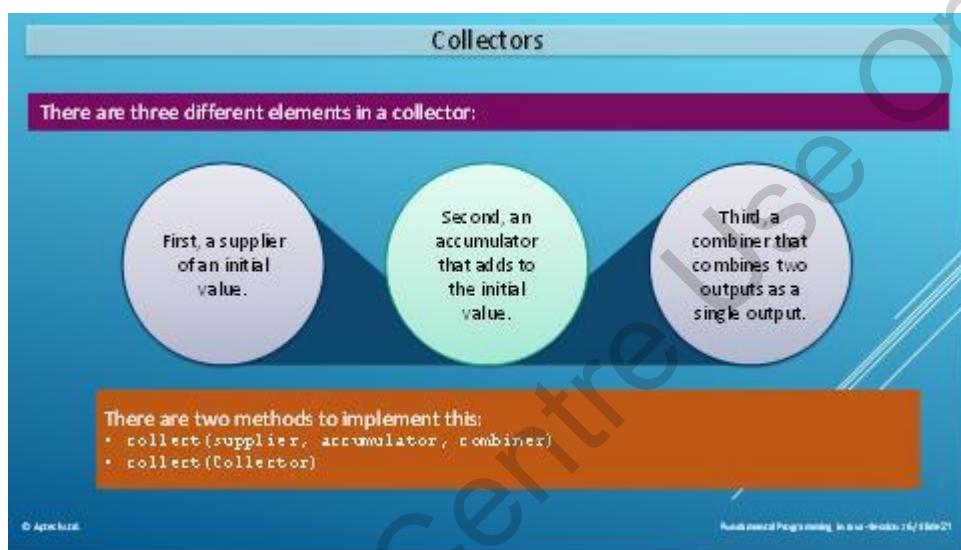
Explain that:

`sorted()` method is another method within Stream API that helps to sort the Stream. It is known as ‘Lazy’ execution because no process is started until a terminal operation (such as `reduce` or `foreach`) is called.

Note that a limiting operation must be called before the sorting operation on an infinite Stream, a Stream that does not have a fixed size.

### Slide 21

Let us understand collectors.



Explain about collectors and implementation of collectors.

Explain that:

The three elements of collector are:

- Supplier: A supplier of an initial value.
- Accumulator: An accumulator that adds to the initial value.
- Combiner: A combiner that combines two outputs as a single output.

For example, from a Student list, Supplier, you collect the marks of two semesters (Accumulator) and combine the result to find whether the student qualifies (Combiner).

The two methods to implement this are:

- `collect(supplier, accumulator, combiner)`
- `collect(Collector)`

**Slides 22 to 24**

Let us understand grouping and partitioning.

### Grouping and Partitioning 1/3

#### Grouping

**Grouping (groupingBy) collector groups elements based on a given function.**

```
// Grouping using first letter
List<Tiger>tigers = getTigers();
Map<Character, List<Tiger>> map = tigers.stream()
.collect(groupingBy(tiger ->tiger.getName().charAt(0)));
// first letter
```

Names from the tigers list is grouped based on the first letter of each name.

© Aptech

Java Stream Programming In Java 8 (slide 22 of 24)

### Grouping and Partitioning 2/3

#### Partitioning

**Partitioning (partitioningBy) method is parallel to Grouping method that creates a map with a boolean key.**

```
Map<Boolean, List<Tiger>> map = tigers.stream()
.collect(partitioningBy(Tiger::isWhite)) // white or not
```

Groups the elements based on whether the Tiger is white or not

© Aptech

Java Stream Programming In Java 8 (slide 23 of 24)

### Grouping and Partitioning 3/3

#### Parallel Grouping

**Parallel Grouping (groupingByConcurrent) executes grouping in parallel (without ordering).**

```
tigers.parallelStream().unordered()
.collect(groupingByConcurrent(Tiger::getColor)) //parallel grouping
```

© Aptech

Java Stream Programming In Java 8 (slide 24 of 24)

Explain about grouping. Also, explain the code snippet that groups strings based on first letter.

Explain that:

Grouping (`groupingBy`) collector groups elements based on a given function.

The `groupingBy` operation returns a map of values that are obtained by applying the lambda expression specified as its parameter. For example,

```
Map<City, List<People>> byCity
 = peoples.stream()
 .collect(Collectors.groupingBy(People::getCity));
```

Here, the parameter for grouping is `city`.

Explain about partitioning. Also, explain the code snippet that partitions the tiger list based on whether it is white or not.

Explain that:

Partitioning (`partitioningBy`) method is parallel to Grouping method that creates a map with a boolean key, True or False.

In the City example, of previous slide, you can mention the partition condition of more than 1 lakh population.

```
Map<Boolean, List<People>> partitioned =
 peoples.stream()
 .collect(partitioningBy(p -> p.getCity() >=
100000));
```

Similarly, a Students map can be obtained for the boolean key, Pass or Fail.

Thus, in partitioning the resultant is a map that contains two groups, Pass/Fail, Yes/No, and so on.

In the code snippet, the tiger list is partitioned by the boolean key, white or not. The code line `Map<Boolean, List<Tiger>> map = tigers.stream().collect(partitioningBy(Tiger::isWhite))` counts the number of tigers that are white.

Explain about parallel grouping. Also, explain the code snippet that displays parallel grouping.

Explain that:

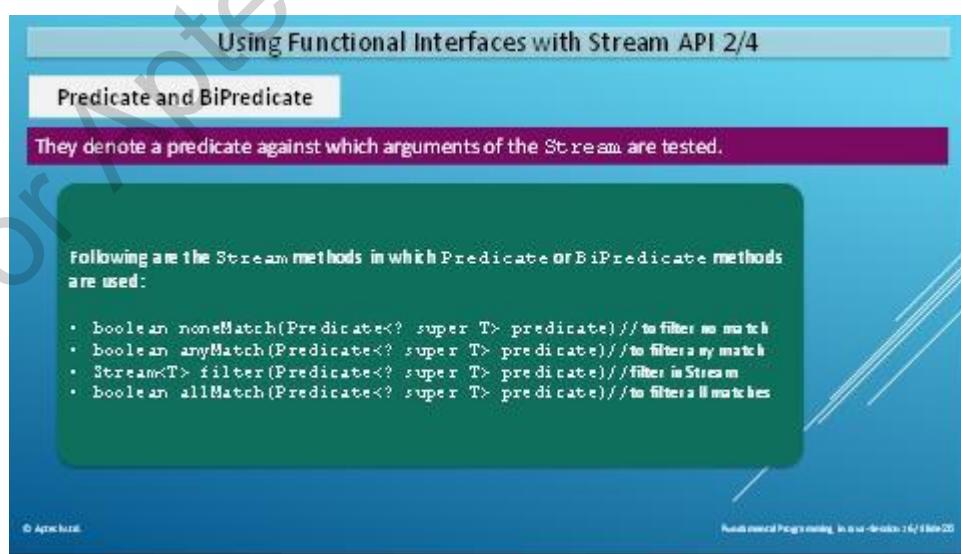
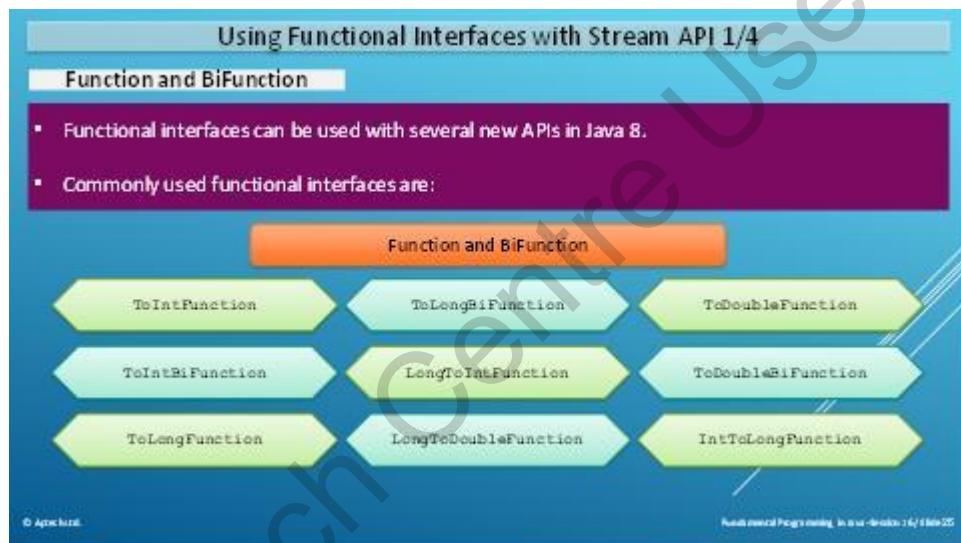
Parallel Grouping (`groupingByConcurrent`) executes grouping in parallel (without ordering).

`groupingByConcurrent` returns a concurrent Collector implementing a ‘group by’ operation on input elements of type T, grouping elements according to a classification function.

In the code snippet, from an unordered stream of list of Tigers, the input element ‘tigers’ are grouped based on their color (`getColor`).

### Slides 25 to 28

Let us explore functional interfaces in Java 8.



**Using Functional Interfaces with Stream API 3/4**

**Consumer and Biconsumer**

They denote operations that accept a single input element and produce no output.

Example demonstrates Consumer and Biconsumer functions.

```

import java.util.Arrays;
import java.util.List;
import java.util.function.Consumer;
public class SampleDemo {
 public static void main(String[] args) {
 List<Employee> employees = Arrays.asList(
 new Employee("John Simmens", 350000),
 new Employee("Mark Smith", 413000),
 new Employee("Jane Weston", 344000),
 new Employee("Gillian Bush", 690000)
);
 displayAllEmployees(employees, e → {
 e.salary *= 1.5;
 });
 System.out.println("Salaries after increment:");
 displayAllEmployees(employees, e →
 System.out.println(e.empname + ": " + e.salary));
 }
}

```

© Aptech

Functional Programming In Java - Chapter 26 / Slide 23

**Using Functional Interfaces with Stream API 4/4**

```

public static void displayAllEmployee(List<Employee> emp,
 Consumer<Employee> printer) {
 for (Employee e : emp) {
 printer.accept(e);
 }
}
public static void display(List<Employee> emp,
 Consumer<Employee> printer) {
}
class Employee {
 public String empname;
 public long salary;
 Employee(String name, long sal) {
 this.empname = name;
 this.salary = sal;
 }
}

```

© Aptech

Functional Programming In Java - Chapter 26 / Slide 23

Explain about Function and BiFunction interfaces.

Using slide 25, explain that:

Functional interfaces can be used with several new APIs in Java 8.

BiFunction interface produces output from two argument.

```

BiFunction<String, String, String>
bi = (a, b) -> {
 return a + b;
};

```

The two input arguments are: a and b.

Using slide 26, explain about **Predicate** and **BiPredicate** and their respective methods.

Explain that:

**Predicate** and **BiPredicate** denote a predicate or boolean-valued function against which arguments of the Stream are tested.

Consider the code:

```
BiPredicate<Integer, Integer> SamplebiPred = (t, u) -> t > u;
System.out.println(SamplebiPred .test(20, 10));
```

The `test()` method evaluates the arguments 20 and 10 to find whether 20 is greater than 10. The output is true.

Using slides 27 and 28, explain about **Consumer** and **BiConsumer**. Explain the code snippet that demonstrates **Consumer** methods.

Explain that:

**Consumer** `<T>` denote operations that accept a single input element and produce no output and **Biconsumer** `<T, U>` operation accepts two input arguments and returns no result.

Continue to explain the code snippet that demonstrates **Consumer** and **Biconsumer** methods.

Explain that:

The `displayAllEmployee()` method iterates through the list of employees and invokes the `accept()` method of the consumer object for every employee in the list. The `accept()` method shown in `printer.accept(e)` is the primary abstract method of the **Consumer** functional interface that displays the name and salary of every employee.

Stream methods in which **Consumer**, **BiConsumer**, or other primitive interfaces are used:

```
void forEach(Consumer<? super T> action)
Stream<T> peek(Consumer<? super T> action) // peek action
void forEachOrdered(Consumer<? super T> action)
```

SampleDemo is a class with three methods `main()`, `displayAllEmployee()`, and `display`. In `main()` method, an array of employees are taken. Then, the `main` method displays all employee name after multiplying salary of each employee by 1.5 that is, `displayAllEmployee(employees, e -> { e.salary *= 1.5});`

The salaries after increment `displayAllEmployee(employees, e -> System.out.println(e.empname + ": " + e.salary));`. Here two arguments are displayed `e.empname` and `e.salary` showing **BiConsumer** operation.

### **Additional Information:**

Use the following links for additional reference:

<http://www.java2s.com/Tutorials/Java/java.util.function/BiFunction/index.htm>  
<https://www.safaribooksonline.com/library/view/java-8-lambda/9780133750867/part23.html>  
<http://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/>  
<http://www.java2s.com/Tutorials/Java/java.util.function/BiPredicate/index.htm>  
<http://data-structure-learning.blogspot.in/2015/07/java-lambda-bipredicate-functional.html>  
<http://www.concretепage.com/java/jdk-8/java-8-biconsumer-bifunction-bipredicate-example>

Slide 29

Let us understand about Optional class and Spliterator interface.

## Optional and Spliterator API

Optional is a container object that optionally contains a value (non-null).

If it contains a value, isPresent () shows true and get () returns the value.

Following are Stream terminal operations that return an Optional object:

- `Optional<T> min(Comparator<? super T> comparator)`  
    // minimum
- `Optional<T> max(Comparator<? super T> comparator)`  
    // maximum
- `Optional<T> reduce(BinaryOperator<T> accumulator)`  
    // to reduce
- `Optional<T>findFirst()`  
    // to find first
- `Optional<T>findAny()`  
    // to find any

Spliterator interface is used to support the parallel execution. Spliterator(trySplit) method which produces a new Spliterator that manages a subset of the elements of the original Spliterator.

© Aptech Ltd.

Fundamental Programming in Java -Session 16 / Slide 29

Explain that:

The `Optional` class and `Spliterator` interface defined in `java.util` package are used with Stream API.

`Optional` is a container object that optionally contains a value (non-null). Stream operators use optional, when it is not sure about the result. `isPresent()` function is called to check for any contents in the stream. The `get()` method is used to extract the output, if `Option` has values.

`Spliterator` interface is used to support parallel execution of processes. The `trySplit()` method produces a new `Spliterator` that manages a subset of the elements of the original `Spliterator`.

Using a Splitterator, the input array or collection is divided into a numerous sub-arrays or collections. For each of these sub-arrays or collections, a parallel thread is used to execute the stream pipeline.

## Slide 30

Let us understand about parallelism.

**Parallelism**

- Parallelism is splitting a task into its sub-tasks and then simultaneously running these tasks to merge their outputs.
- Adding a `parallel()` method to the Stream instructs the library to deal with the complexities of threading. Thus, the library controls the process of forking.

Fundamental Programming in Java -Session 16 / Slide 30

Explain about parallelism.

Explain that:

Parallelism is splitting a task into its sub-tasks and then simultaneously running these tasks to merge their outputs. Parallelism executes with parallel streams. It divides streams into multiple subtasks and merges them. Thus, `parallel()` method helps in dividing a work. It is similar to an intermediate operation.

Adding a `parallel()` method to the Stream instructs the library to deal with the complexities of threading. Java 8 consists of fork/join framework that aids in implementation of parallel computing by splitting the work into subtasks. After splitting, the task waits until all the subtask finish executing. Then, the task joins all the results into one single result.

## Slides 31 to 34

Let us understand how to execute streams in parallel.

### Executing Streams in Parallel 1/4

- ❖ Aggregate operations are implemented to combine the results.
- ❖ This process is known as concurrent reduction.
- ❖ Following conditions must be true for performing a collect operation in the process:
  - The Stream must be parallel.
  - The parameter of the collect operation, the collector, contains the characteristic Collector.Characteristics.CONCURRENT.
  - Stream must be unordered or the collector must contain the Collector.Characteristics.UNORDERED.

© Aptech Ltd. Fundamental Programming in Java -Session 16 / Slide 31

### Executing Streams in Parallel 2/4

Following example shows a complete program with various Stream API operations:

```

import java.util.Arrays;
import java.util.IntSummaryStatistics;
import java.util.List;
import java.util.stream.Collectors;
public class AptechJavaStreamAPI {
 public static void main(String args[]) {
 List<String> clientList = Arrays.asList("Flipkart",
 "Snapdeal", "PayTm", "King", "", "", "MaBeats", "Miniclip");
 System.out.println("^The new Client List: " + clientList);
 System.out.println("Result2:no. of clients with name length > 5: " + lengthCount);
 //To receive the client name starts with letter 'A' and display count
 long startCount = clientList.stream().filter(x -> x.startsWith("M")).count();
 System.out.println("Result3:no. of clients which name starts with letter M: " +
 startCount);
 // To eliminate all empty Strings from List
 List<String>removeEmptyStrings =
 clientList.stream().filter(x ->!x.isEmpty()).collect(Collectors.toList());
 System.out.println("Result4:no. New Client List without empty list" +
 removeEmptyStrings);
 // To display the client names with > 8 characters
 }
}

```

© Aptech Ltd. Fundamental Programming in Java -Session 16 / Slide 32

### Executing Streams in Parallel 3/4

```

List<String> newList = clientList.stream().filter(x ->x.length() >
8).collect(Collectors.toList());
System.out.println("Result5: New client list with letter count > 8: " + newList + "\n");
List<Integer> aptechInt = Arrays.asList(77,66,888, 22, 33,7, 121, 89,55);
IntSummaryStatistics aptechStats = aptechInt.stream().mapToInt((x) ->
x).summaryStatistics();
System.out.println("^ A list of Random numbers: " + aptechInt);
System.out.println("Highest number in the lot -" + aptechStats.getMax());
System.out.println("Lowest number in the lot -" + aptechStats.getMin());
System.out.println("Combined value of All: " + aptechStats.getSum());
System.out.println("Average value of all numbers: " + aptechStats.getAverage() + "\n");
// To convert a Message in UPPERCASE and join them using space
List<String> aptechTips = Arrays.asList("java8", "has", "some", "great", "features");
String joinList = aptechTips.stream().map(x ->
x.toUpperCase()).collect(Collectors.joining(" "));
System.out.println("- To Join and Display the message with UPPERCASE: " + joinList);
// To display the cube value of the numbers
List<Integer> numbers = Arrays.asList(5,10,15,20,25);
List<Integer> cubes = numbers.stream().map(myInt ->myInt *
myInt * myInt).distinct().collect(Collectors.toList());
System.out.println("- Display the cube value of the numbers : " + cubes + "\n");
}

```

© Aptech Ltd. Fundamental Programming in Java -Session 16 / Slide 33

Executing Streams in Parallel 4/4

**Output:**

```
$javac AptechJavaStreamAPI.java 2>&1
^The new Client List: [Flipkart, Snapdeal, PayTm, King, , , MaBeats, Miniclip]
Result1:no. of Empty Strings: 2
Result2:no. of clients with name length > 5: 4
Result3:no. of clients which name starts with letter M: 2
Result4:no. New Client List without empty list[Flipkart, Snapdeal, PayTm, King,
MaBeats, Miniclip]
Result5: New client list with letter count > 8: []
^ A list of Random numbers: [77, 66, 888, 22, 33, 7, 121, 89, 55]
Highest number in the lot -888
Lowest number in the lot -7
Combined value of All: 1358
Average value of all numbers: 150.88888888888889
- To Join and Display the message with UPPERCASE: JAVA8 HAS SOME GREAT FEATURES
- Display the cube value of the numbers : [125, 1000, 3375, 8000, 15625]
```

© Aptech Ltd.

Fundamental Programming in Java -Session 16 / Slide 34

Explain about aggregate operations.

Explain that:

Aggregate operations are implemented to combine the results. This process is known as concurrent reduction. Aggregate operations process elements from a stream. A sequence of aggregate operations is known as pipeline. You can customize the behavior of an aggregate operator by using lambda expression as parameter. Aggregate operations do not contain a method such as `next` to process the next element within a collection.

Explain the code snippet that shows a complete program with various Stream API operations.

Explain that:

In the code snippet, a list is created for strings Flipkart, Snapdeal, PayTm, King, MaBeats, and Miniclip, including two empty lists.

The line `System.out.println("The new Client List: " + clientList)` is used to count the number of empty strings.

Then, the line `System.out.println("Result2:no. of clients with name length > 5: " + lengthCount);` shows name of clients with length of more than five characters.

The line `long startCount = clientList.stream().filter(x -> x.startsWith("M")).count();` shows the client name starts with letter 'M' and displays the count.

The line `List<String>removeEmptyStrings = clientList.stream().filter(x -> !x.isEmpty()).collect(Collectors.toList());` eliminates all empty strings from lists and lists the client names without the empty strings.

Continue the explanation of the code snippet that shows a complete program with various Stream API operations.

Explain that:

Finally, the client names with more than eight characters are displayed, in the code snippet

```
List<String> newList = clientList.stream().filter(x ->x.length() >8).collect(Collectors.toList());
```

An array of numbers are taken 77, 66, 888, 22, 33, 7, 121, 89, 55 then, aptechStats.getMax() method picks the highest number in the list, aptechStats.getMin() method picks the lowest number in the lot, aptechStats.getSum() method gets the sum of all the values and aptechStats.getAverage() method is used to calculate the average value of all numbers.

A list of strings, ("java8", "has", "some", "great", "features") is created and they are converted into uppercase, aptechTips.stream().map(x ->x.toUpperCase()).collect(Collectors.joining(" "));

Finally, an array of numbers is created with 5, 10, 15, 20, and 25 and its cubic value is calculated, numbers.stream().map(myInt ->myInt \*myInt \* myInt).distinct().collect(Collectors.toList());

Using slide 34, explain the output of the code snippet that shows a complete program with various Stream API operations.

### Slide 35

Let us look at the limitations of Java Stream API.

The slide has a light blue background with a dark blue header bar. The title 'Limitations of Java Stream API' is centered in the header. Below the header, there are two green rectangular callout boxes containing white text. The first box contains the text: 'Once a Stream is consumed, it cannot be used later.' The second box contains the text: 'Learning is time-consuming and cumbersome due to overloaded Stream APIs.' At the bottom left of the slide, there is a small copyright notice: '© Aptech Ltd.' and at the bottom right: 'Fundamental Programming in Java -Session 16 / Slide 35'.

- Once a Stream is consumed, it cannot be used later.
- Learning is time-consuming and cumbersome due to overloaded Stream APIs.

Explain the limitations of Java Stream API.

Explain that:

There are many advantages of the Java Stream API, however, there are a few drawbacks too.

Some of these are:

- Once a Stream is consumed, it cannot be used later.
- There are vast number of methods in Stream APIs. Therefore, learning is time-consuming and cumbersome due to overloaded Stream APIs.
- Parallel processing is not always beneficial as it may speed up or slow down an application based on the nature of the application.
- Parallel streams are unpredictable and are complex to use.

#### **Additional Information:**

Use the following links for additional reference:

<http://www.javaworld.com/article/2095503/java-se/uses-and-limitations-of-the-stream-api-in-java-8.html>

<https://blog.jooq.org/2014/06/13/java-8-friday-10-subtle-mistakes-when-using-the-streams-api/>

#### **Slide 36**

Let us summarize the session.

**Summary**

- ❖ The new Stream API in Java 8 supports many sequential and parallel aggregate operations
- ❖ Stream API interfaces and classes are contained within `java.util.stream` package
- ❖ The foundation of the Stream API is `Stream` interface and `Collectors` class
- ❖ Some of the interfaces in the API include `IntStream`, `LongStream`, and `DoubleStream`
- ❖ Streams are lazily implemented and support parallel operation
- ❖ Function denotes a function that gets one type of element and produces another type of element
- ❖ The `Optional` class and `Spliterator` interface defined in `java.util` package can be used with Stream API
- ❖ Commonly used functional interfaces with Stream API include `Function` and `BiFunction`, `Predicate` and `BiPredicate`, `Consumer` and `BiConsumer`, and `Supplier`



© Aptech Ltd.      Fundamental Programming in Java - Session 16 / Slide 36

Use the slide to summarize the session. You will end the session, with a brief summary of what has been taught in the session. Tell the students pointers of the session. This will be a revision of the current session.

Explain that:

The new Stream API in Java 8 supports many sequential and parallel aggregate operations. Stream API interfaces and classes are contained within `java.util.stream` package. The foundation of the Stream API is `Stream` interface and `Collectors` class. Some of the interfaces in the API include `IntStream`, `LongStream`, and `DoubleStream`. Streams are lazily

implemented and support parallel operation. Function denotes a function that gets one type of element and produces another type of element. The Optional class and Spliterator interface defined in java.util package can be used with Stream API. Commonly used functional interfaces with Stream API include Function and BiFunction, Predicate and BiPredicate, Consumer and BiConsumer, and Supplier.

### **16.3 Post Class Activities for Faculty**

You should familiarize yourself with the topics of the next session.

**Tips:**

You can also check the Articles/Blogs/Expert Videos uploaded on the OnlineVarsity site to gain additional information related to the topics covered in the next session. You can also connect to online tutors on the OnlineVarsity site to ask queries related to the sessions.

# Session 17 – More on Functional Programming

## 17.1 Pre-Class Activities

Before you commence the session, you should familiarize yourself with the topics of this session in-depth. Prepare a question or two that will be a key point to relate the current session objectives.

### 17.1.1 Objectives

By the end of this session, learners will be able to:

- Explain functional interfaces
- Describe immutability in Java
- Define and explain concurrency in Java
- Explain Recursion in Java

### 17.1.2 Teaching Skills

To teach this session, you should be well versed with the concepts of advanced functional programming in Java. You should be familiar with various functional interfaces, immutability, reinforcing immutability, and implementing concurrency. You must also be familiar with recursion techniques in Java.

You should teach the concepts in the theory class using the images provided. For teaching in the class, you are expected to use slides and LCD projectors.

#### Tips:

It is recommended that you test the understanding of the students by asking questions in between the class.

#### In-Class Activities

Follow the order given here during In-Class activities.

#### Overview of the Session

Give the students an overview of the current session in the form of session objectives. Read out the objectives on slide 2.

**Slide 2**


Functional Programming in Java - Recursion / Slide 2

### Objectives

- ❖ Explain functional interfaces
- ❖ Describe immutability in Java
- ❖ Define and explain concurrency in Java
- ❖ Explain Recursion in Java



Functional Programming in Java - Recursion / Slide 2

Show the slide and give the students a brief overview of the current session in the form of session objectives. Tell the students that this session explains about functional interfaces. The session will cover immutability in Java and explains about concurrency in Java. Finally, it explains about recursion in Java.

**17.2 In-Class Explanations****Slide 3**

Let us see an introduction to functional interfaces.



Functional Programming in Java - Recursion / Slide 3

### Introduction to Functional Interfaces

|                                                                                                                                                   |                                                                                                                                           |                                                                                                                                   |
|---------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"> <li>➢ Functional interfaces are key elements that helps to implement functional programming in Java</li> </ul> | <ul style="list-style-type: none"> <li>➢ It supplies target type of elements such as lambda expressions and method references.</li> </ul> | <ul style="list-style-type: none"> <li>➢ Functional interface has one abstract method and zero or more default methods</li> </ul> |
|---------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------|

Functional Programming in Java - Recursion / Slide 3

Tell the students that they learnt about functional programming in an earlier session. Explain that Java is basically an object-oriented programming language. With the introduction of lambda expressions and method references, it is possible to implement functional programming in Java 8.

An interface with only one abstract method is known as a functional interface. Some examples of functional interfaces are Runnable, Comparable, and Cloneable. Comparable interface has a method called `compareTo()` used for comparison purposes.

### **Additional Information:**

Refer the following link for more information:

[https://www.tutorialspoint.com/java8/java8\\_functional\\_interfaces.htm](https://www.tutorialspoint.com/java8/java8_functional_interfaces.htm)

### **Slide 4**

Let us understand Function <T, R> interface.

The built-in functional interfaces are included in the `java.util.function` package. `Function<T, R>` is one of them.

`Function<T,R>` can be used:

- As assignment target for a lambda expression.
- To map scenarios such as streams wherein `map()` function of a stream takes an instance of `Function` to convert the stream of one type to a stream of different type.

© Aptech Limited Java Functional Programming In Java 8 Version 17.06.18

Explain about Function <T, R> interface.

Explain that:

In Java, `java.util.Function` package defines a list of functional interfaces to be used in lambda expressions.

`Function<T, R>` is a function that accepts one argument as input to produce the output.

An argument of type T is given as input to the lambda expression. It produces an object of type R as output. This function is mainly used when an object of specific type is given as input and is converted into another type.

For example, the following lambda expression can be used to define an interface to check for new products:

```
IPrduct newProductChecker = (Product p) ->
p.getStatus().equals("New");
```

Here, the argument  $p$  returns an output of type boolean by checking the type of the product whether new or old.

## Slides 5 and 6

Let us explore the key points under functional interfaces.

**Function 2/8**

$T \rightarrow R$  is the functional descriptor for Function< $T, R$ >

Key Points under Functional Interfaces are:

- 1) Method `apply()` is the prime abstract functional method of Function interface
- 2) Function < $T, R$ > contains two default methods: `compose()` and `andThen()`.

© Aptech Limited Functional Programming in Java - Chapter 17 / Slide 5

**Function 3/8**

`compose()`

- > Combines the function on which it is activated
- > Combines with another function that is named `before()`
- > Input type will change from type  $V$  to type  $T$  if the before function is applied after the combined function

`andThen()`

- > Combines the function on which it is activated
- > Combines with another function called `after()`
- > `andThen()` combines with `after()`, so when combined function is called, at that point the current function is activated which changes the initial value of type  $T$  to type  $V$

© Aptech Limited Functional Programming in Java - Chapter 17 / Slide 6

Using slide 5, explain about the parameters of Function < $T, R$ > interface and about `apply()`, `compose()` and `andThen()` methods.

Explain that:

The `apply()` method is the primary abstract method of Functional interface. It is generally used to convert a list to map or transform one object to another. In this example, a student is mapped to a course.

```
Function<Student, Course> mapStudenttoCourse = new
Function<Student, Course>() {
```

```

public Course apply(Student student) {
 Course crse = new Course(Student.getId(),
student.getName());
 return crse;
}

```

Using slide 6, explain the difference between compose () and andThen () methods.

Consider the following code:

```

Function<Integer, Integer> prod2 = x -> x * 2;
Function<Integer, Integer> squaredprod = x -> x * x;
prod2.compose(squared).apply(3);
prod2.andThen(squared).apply(3);

```

The compose () method returns 18, while andThen () method returns 36.

## Slides 7 and 8

Let us understand the andThen () and compose () methods.

Function 4/8

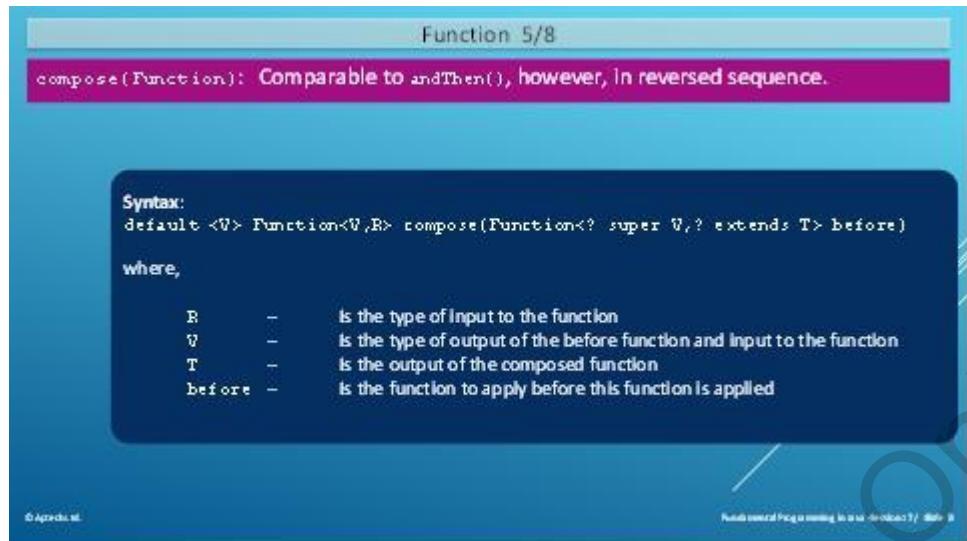
**andThen(Function): Returns a composed function that initially applies this function to its input and later applies the specified function to the output.**

**Syntax:**  
`default <V> Function<T,V> andThen(Function<? super R,>? extends V> after)`

**where,**

|       |   |                                                                          |
|-------|---|--------------------------------------------------------------------------|
| T     | - | is the type of input to the function                                     |
| R     | - | is the type of result of the function                                    |
| V     | - | is the type of output of the after function and of the composed function |
| after | - | is the function to apply after this function is applied                  |

Aptech Limited  
Functional Programming in Java - Session 7 / Slide 7



Using slide 7, explain the syntax of andThen() method.

Explain that:

The following code converts a given input into lowercase:

```
import java.util.function.Consumer;
```

```
public class Main {
 public static void main(String[] args) {
 Consumer<String> change = (a) ->
 System.out.println(a.toLowerCase());
 change.andThen(change).accept("YOU WILL SEE THIS LINE IN
 LOWER CASE");
 }
}
```

Using slide 8, explain the syntax of compose() method.

**Additional Information:**

Refer to the following link to explain how to achieve functional composition using andThen() and compose() methods:

<http://www.deadcoderising.com/2015-09-07-java-8-functional-composition-using-compose-and-andthen>

## Slides 9 and 10

Let us understand the `identity()` function and see an example how to use the method.

**Function 6/8**

**identity(): Is a function which returns its input argument as is.**

**Syntax:**  
`static <T> Function<T,T> identity()`

**where,**  
`T` – Is the type of input and output to the function

**Function 7/8**

**The methods can be used in a chain for creating a function as shown in Code Snippet.**

```
Function<integer, String> sampleF = Function.<integer>identity()
 .andThen(i → i * i).andThen(i → "sampleStr" + i);
```

**Resultant function will acquire an integer, multiply it by 2 and finally, prepend "sampleStr" to it.**

Using slide 9, explain about `identity()` method.

Explain that:

Identity method always return its input argument. For example, the following code produces an output of 9.

```
Function<Integer, Integer> id = Function.identity();
System.out.println(id.apply(9));
```

Using slide 10, explain the method that can be used in a chain for creating a method shown in the code.

Explain that:

Resultant method will acquire an integer, multiply it by 2 and finally, append ‘sampleStr’ to it.

### **Additional Information:**

Refer the following link for more information:

[http://www.java2s.com/Tutorials/Java/java.util.function/Function/1080\\_Function.identity.htm](http://www.java2s.com/Tutorials/Java/java.util.function/Function/1080_Function.identity.htm)

### **Slide 11**

Let us see a complete program demonstrating the methods.

The screenshot shows a Java code editor with the following code:

```

import java.util.*;
import java.util.stream.*;
import static java.util.stream.Collectors.*;
import java.time.*;

public class FunctionDemo {
 public static void main(String[] args) {
 Function<LocalDate, LocalDateTime> plusTwoM =
 Function.<LocalDate> identity();
 .andThen(displayDateTime(d -> d.plusMonths(2)));
 System.out.println(Stream.iterate(LocalDate.now(), d ->
 d.plusDays(1))
 .limit(10)
 .map(plusTwoM)
 .map(Object::toString)
 .collect(joining(", ")));
 }
}

public static Function<LocalDate, LocalDateTime> displayDateTime() {
 final Function<LocalDate, LocalDateTime> test; {
 return test.andThen(d -> d.atTime(2, 2));
 }
}

```

The code uses functional interfaces like `Function` and `Collector` along with streams to generate a sequence of dates two months from today, formatted as strings separated by commas.

Using this slide, explain the code how to implement `displayDateTime()` method.

Explain that:

The code iterates through days and displays date and time two months from today's date using functional interfaces and use the `limit()` method to limit the stream to a size of 10, `map()` method converts Integer stream to String stream, and `collect()` method joins the results.

When you execute the code, the output will be:

2016-12-31T02:02, 2017-01-01T02:02, 2017-01-02T02:02, 2017-01-03T02:02, 2017-01-04T02:02, 2017-01-05T02:02, 2017-01-06T02:02, 2017-01-07T02:02, 2017-01-08T02:02, 2017-01-09T02:02

## Slides 12 to 17

Let us understand currying.

**Currying 1/6**

Currying process transforms a function having multiple arguments into a function with a single argument.

`f(a, b) = (g(a))(b)`

Here,  $f$  is a function,  $a$  and  $b$  are arguments.

© Aptech Limited

Functional Programming, Java 8-Module 12 / Slide 12

**Currying 2/6**

Following is the basic pattern of all unit conversions:

- Multiply by the conversion factor
- Adjust the baseline if relevant

```
static double converter(double a, double e, double y) {
 return a * e + y;
}
```

© Aptech Limited

Functional Programming, Java 8-Module 12 / Slide 13

**Currying 3/6**

**Curry-Converter Usage:**

- Following code is more flexible and it reuses the existing conversion logic.
- Instead of passing the arguments  $a$ ,  $e$ , and  $b$  all at once to the converter method, the arguments  $e$  and  $b$  are called to return another function, which when given an argument returns  $a * e + b$ .
- The method enables to reuse the conversion logic and create different functions with different conversion factors.

```
static DUOcurryConverter(double e, double b){
 return (double a) → a * e + b;
}
```

© Aptech Limited

Functional Programming, Java 8-Module 12 / Slide 14

### Currying 4/6

Apply converters as follows:

Conversion factor and baseline (e and b) is passed to the code which returns a function (of a) to do exactly what is expected.

For example, the converters can be used as follows:

```
DoubleUnaryOperator convert_kgtolbs = curriedConverter(0, 2.2046);
DoubleUnaryOperator convert_GBPtoEUR = curriedConverter(1.268, 0);
DoubleUnaryOperator convert_CtoF = curriedConverter(9.0/5, 32);
```

Code Snippet shows an example `DoubleUnaryOperator` defines a method `applyAsDouble()` which is used here.

```
double gbp =
convertUSDtoGBP.applyAsDouble(1000);
```

© Aptech Limited. Functional Programming, Java 8-Module 12 / Slide-15

### Currying 5/6

Example demonstrates `compose()` in detail.

```
import java.util.function.BiFunction;
import java.util.function.Function;
public class JavaCurry {
 public void curryFunction() {
 // Create a Function that adds 2 integers
 BiFunction<Integer, Integer, Integer> adder = (x, y) -> x + y;
 Function<Integer, Function<Integer, Integer>> curried = x -> y
 -> adder.apply(x, y);
 Function<Integer, Integer> curried = curried.apply(5);
 // To display Results
 System.out.printf("Curry : %d\n", curried.apply(2));
 }
 public void compose() {
 //Function to display the result with * 4
 Function<Integer, Integer> addFour = (x) -> x * 4;
 // Function to display the result with * 5
 Function<Integer, Integer> timesFive = (x) -> x * 5;
 // to display the result with n number of times using compose
 Function<Integer, Integer> compose1 = addFour.compose(timesFive);
 // to display the result with add
 Function<Integer, Integer> compose2 = timesFive.compose(addFour);
 // To display the end Result
 System.out.printf("7 times then add: %d\n", compose2.apply(7));
 // (7 * 4) + 5
 }
}
```

© Aptech Limited. Functional Programming, Java 8-Module 12 / Slide-15

### Currying 6/6

```
System.out.printf("Add then times: %d\n", compose2.apply(7));
// (7 * 5) * 4
}
public static void main(String[] args) {
 new JavaCurry().curryFunction();
 new JavaCurry().compose();
}
```

**Output:**

Curry:7  
Result as Times then add:39  
Result as Add then times:55

© Aptech Limited. Functional Programming, Java 8-Module 12 / Slide-15

Using slide 12, explain about currying process.

Explain that:

The currying process works by subdividing the task and working on the in-between results.

Consider in the given example, we use the method for calculating sum of two numbers say  $f(a,b) = a+b$ .

The currying process subdivides the task as:

$$f(2,b) = 2+b$$

$$g(b) = 2+b$$

$$f(2,b) = g(b) = 2+b$$

$$f(2,3) = g(3) = 2+3=5$$

Using slide 13, explain the basic pattern of all unit conversions. Using slide 14, explain about the usage of curry converter method. Then, using slide 15, explain the pattern of all unit conversions.

Explain that:

Conversion factor and baseline ( $e$  and  $b$ ) is passed to the code which returns a function to do the expected calculation. DoubleUnaryOperator defines a method applyAsDouble().

Using slide 16, explain the code that demonstrates the usage of the compose() method.

Explain that:

In the curryfunction() method, adder.apply(x, y) will add the two integers 2, 7 and display the result. In the method addFour.compose(timesFive), values 7 and 4 are added and the result is multiplied with 5. In the method timesFive.compose(addFour), 7 and 5 are multiplied and the result is added with 4.

Using slide 17, explain the output of the code.

#### **Additional Information:**

Refer the following links for more information:

<http://www.beyondjava.net/blog/java-8-functional-programming-language/>

<https://gist.github.com/timyates/7674005>

## Slides 18 to 21

Let us explore immutability.

### Immutability 1/4

- ❖ If the state of an object cannot change after it is constructed, it is considered immutable.
- ❖ For example, String is an immutable type in Java.
- ❖ Immutable objects are specifically valuable in concurrent applications.
- ❖ Example shows the concept of immutability.

```
public class Main {
 public static void main(String[] args) {
 String sample = "immutable";
 System.out.println(sample); // immutable
 change(sample);
 System.out.println(sample); // immutable
 }
 public static void change(String str) {
 str = "mutable";
 }
}
```

**Output:**

immutable  
immutable

Aptech Limited

Functional Programming, Java 8 - Day 03 / Slide 10

### Immutability 2/4

#### Immutable Objects

Objectives to use immutable objects:

- ❖ If it is known that an object's state cannot be changed by another method, then it is easier to reason about how your program works.
- ❖ Immutable objects are thread-safe by default.
- ❖ Immutable objects can be used as keys in a **HashMap** (or similar), as they have the same hash code forever.

Aptech Limited

Functional Programming, Java 8 - Day 03 / Slide 10

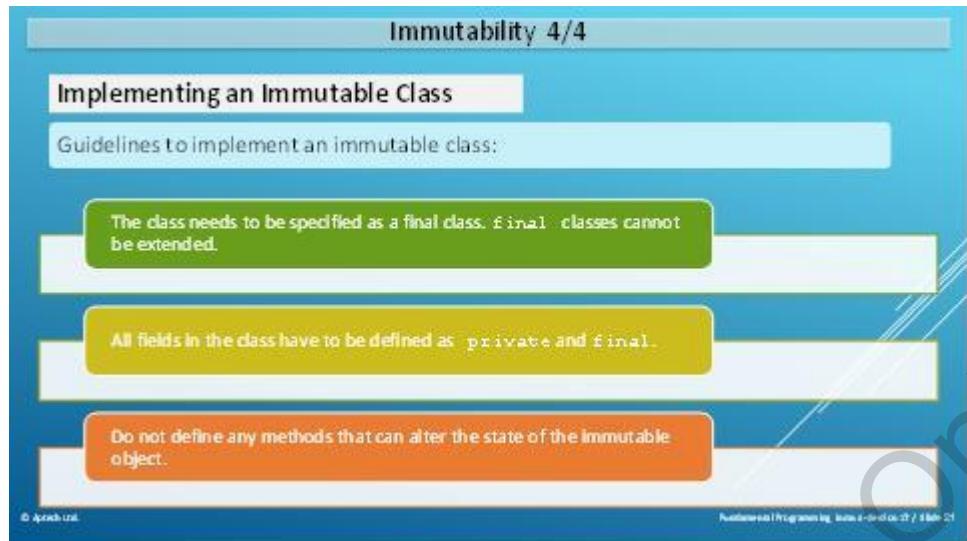
### Immutability 3/4

#### Immutable Class

- ❖ Immutable class is a class in which the state of its instances does not change while it is constructed.
- ❖ Following are some of immutable classes in Java: `java.lang.String`, `java.lang.Integer`, `java.lang.Float`, and `java.math.BigDecimal`.

Aptech Limited

Functional Programming, Java 8 - Day 03 / Slide 20



Explain that:

Immutability remains unchanged. If a class has only static members, then the objects of the class are immutable. Immutability helps to ensure thread safety.

Using slide 19, explain about immutability objects.

Explain that:

Immutable objects are consistent, thread safe, and hence, can be shared safely among threads. All its members must be final and private.

Using slide 20, explain about immutable class.

Explain that:

An immutable class is stateless. Its properties does not change once it is initialized. All class members of immutable class are declared final. The child objects of immutable class are also immutable.

Using slide 21, explain the steps to implement immutable class.

#### Additional Information:

Refer the link

<https://docs.oracle.com/javase/tutorial/essential/concurrency/immutable.html> to explain about immutable objects and the strategy to define immutable objects.

Refer the link <http://stackoverflow.com/questions/6305752/how-to-create-immutable-objects-in-java> to explain the features of immutable class.

Refer the following link for more information:

<https://docs.oracle.com/javase/tutorial/essential/concurrency/imstrat.html>

## Slides 22 and 23

Let us understand concurrency.

**Concurrency 1/2**

- ❖ Concurrency is performing multiple processes that can start, run, and finish in an overlapping time period.
- ❖ An alternate way in which Java 8 supports concurrency is through the new `CompletableFuture` class.
- ❖ One of its methods is the `supplyAsync()` static method that accepts an instance of the functional interface `Supplier`.



© Aptech Ltd.  
Fundamental Programming In Java - Session 17 / Slide 22

**Concurrency 2/2**

- ❖ Concurrency also has the method `thenAccept()` which accepts a `Consumer` that manages completion of the task.
- ❖ The `CompletableFuture` class calls on the specified supplier in a diverse thread and executes the consumer when it is complete.

© Aptech Ltd.  
Fundamental Programming In Java - Session 17 / Slide 23

Using slide 22, explain about concurrency. Mention that concurrency was supported in Java in earlier versions too (since Java 5) but is made better in Java 8. Give some examples to illustrate the concept of concurrency. Say that end-users assume that their systems can perform multiple actions at a time. Often, even a single program is expected to perform more than one task at a time. Software that can do such tasks is known as concurrent software. A well-written program uses concurrency to ensure that the program is always responsive to user interaction, even when it is already doing something else.

Explain that:

`CompletableFuture` class in Java 8 implements the `Future` interface and `CompletionStage` interface. It supports lambda expressions, parallelism, and is event driven. `supplyAsync()` method helps to run tasks asynchronously.

Using slide 23, explain about thenAccept method and CompleteableFuture class.

thenAccept () method takes a Consumer as an argument and applies Consumer to handle the result of the preceding computation and returns CompleteableFuture.

### Additional Information:

Refer the following links for more information:

<http://docs.oracle.com/javase/tutorial/essential/concurrency/procthread.html>

<https://examples.javacodegeeks.com/core-java/util/concurrent/java-8-concurrency-tutorial/>

<http://www.deadcoderising.com/java8-writing-asynchronous-code-with-completablefuture/>

### Slides 24 to 28

Let us understand recursion.

**Recursion 1/5**

Recursion is a programming language feature supported by various languages including Java. Following example clearly shows how to produce Fibonacci numbers with recursive lambda.

```

import java.lang.Math;
import java.util.Locale;
import java.text.NumberFormat;
import java.text.DecimalFormat;
import java.util.stream.IntStream;
import java.util.stream.DoubleStream;
import java.util.function.IntUnaryOperator;
//Aptech Java ITC
/*Recursion Sample*/
public class FibonacciRecursion {
 static IntUnaryOperator fibonacci;
 public static void main(String[] args) {
 System.out.println("Fibonacci Number Sequence");
 IntStream.range(0,15).// to display how many Fibonacci numbers
 .map(FibonacciRecursion::f);
 return f == 0 || f == 1
 ? 1
 : FibonacciRecursion.f.applyAsInt(f - 2) + FibonacciRecursion.f.applyAsInt(f - 1);
 }
 .parallel();
 .ForEachOrdered(g → System.out.printf("%s ", g));
}

```

**Output:**

Fibonacci Number Sequence:  
1 1 2 3 5 8 13 21 34  
55 89 144 233 377 610

**Recursion 2/5**

Recursion vs. Iteration

- Typical functional programming languages are not bundled with iterative constructs such as while and for loops
- Following Code Snippet illustrates the usage of while loop with an iterator.

```

mangoes { } pass into the Iterator shown here:
Iterator<Mango> th = mangoes.iterator();
while (th.hasNext()) {
 Mango = th.next();
 // ...
}

```

Here, the mutations (both changing the state of the Iterator with the next() method and assigning to the variable mango inside the while body) are not visible to the caller of the method where the mutations occur.

**Recursion 3/5**

**Recursion vs. Iteration**

Using `for-each` loop, such as a search algorithm, is problematic since the loop body is updating a data structure that is shared with the caller as shown in Code Snippet.

```
public void searchForPlatinum(List<String> l, Stats bunch){
 for(String p: l) {
 if("platinum".equals(p)){
 bunch.incrementFor("platinum");
 }
 }
}
```

The body of the loop has a side effect that cannot be neglected as functional style: it mutates the state of the `bunch` object, which is shared with other parts of the program.

© Aptech Limited Functional Programming, Java 8-Module 27 / 48 Min: 20

**Recursion 4/5**

**Iterative Factorial**

Following Code Snippet demonstrates a standard loop-based form; the variables `k` and `h` are updated and iterated.

```
static int factIter(int j) //iterative approach
int k = 1;
for (int h = 1; h<= j; h++) {
k *= h;
}
return k;
}//result
```

© Aptech Limited Functional Programming, Java 8-Module 27 / 48 Min: 20

**Recursion 5/5**

**Recursive Factorial**

Following Code Snippet demonstrates a recursive definition.

```
static long factRecur(long i) //recursive approach
return i == 1 ? 1 : i * factRecur(i-1);
}// result
```

© Aptech Limited Functional Programming, Java 8-Module 27 / 48 Min: 20

Using slide 24, explain about recursion.

Explain that:

A method that calls itself is said to be recursive.

Using slides 25 to 28, explain the difference between recursion and factorial constructs.

Recursion is faster when there are lesser elements. When the value of the nth element increases recursion becomes slower and iteration becomes faster.

#### **Additional Information:**

Refer the following links for more information:

<https://dzone.com/articles/do-it-java-8-recursive-and>

<http://www.programcreek.com/2012/10/iteration-vs-recursion-in-java/>

<https://howtoprogramwithjava.com/java-recursion/>

#### **In-Class Question:**

After you finish explaining slide 28, you will ask the students some In-Class questions. This will help you in reviewing their understanding of the topic.



What is the output for the following method?

```
Function<Integer, Integer> times2 = e -> e * 2;
Function<Integer, Integer> squared = e -> e * e;
times2.compose(squared).apply(4);
times2.andThen(squared).apply(4);
```

#### **Answer:**

32

64



What would be the output for the following program?

```
public static void main(String[] args)
{
 IntBinaryOperator simpleAdd = (a, b) -> a + b;
 IntFunction<IntUnaryOperator> curriedAdd = a -> b -> a + b;
 out.println(simpleAdd.applyAsInt(4, 5));
 out.println(curriedAdd.apply(4).applyAsInt(5));
 IntUnaryOperator adder5 = curriedAdd.apply(5);
 out.println(adder5.applyAsInt(4));
 out.println(adder5.applyAsInt(6));
}
```

**Answer:**

9  
9  
9  
11



Will the program compile without error, if yes, what will be the output?

```
public static void main(String[] args) {
 Function<Integer, Integer> id = Function.identity();
 System.out.println(id.apply(3));
}
```

**Answer:** Yes, the program compiles without any error.

The output will be: 3



What will be the output for the following code? Also, explain the output.

```
class Testimmutablestring{
 public static void main(String args[]){
 String a="Anthony";
 a.concat(" Mathews");
 System.out.println(a);
 }
}
```

**Answer:** The output is Anthony

It will print Anthony since strings are immutable objects.

## Slide 29

Let us summarize the session.

**Summary**

- ❖ Functional interfaces acts as a key element to implement functional programming in Java
- ❖ Functional interfaces are defined in `java.util.function` package, which is new to Java 8
- ❖ A functional interface has only one abstract method
- ❖ Currying is a process that transforms a function having multiple arguments into a function with a single entity that returns a function
- ❖ Immutability is the capability of an object to resist or prevent change
- ❖ In Java programming, a feature that permits a method to call itself is called recursion

© Aptech Limited

Use slide 29 to summarize the session. You will end the session, with a brief summary of what has been taught in the session. Tell the students pointers of the session. This will be a revision of the current session.

Explain that:

Functional interfaces acts as a key element to implement functional programming in Java. Functional interfaces are defined in `java.util.function` package, which is new to Java 8. A functional interface has only one abstract method. Currying is a process that transforms a method having multiple arguments into a method with a single entity that returns a method. Immutability is the capability of an object to resist or prevent change. In Java programming, a feature that permits a method to call itself is called recursion.

### 17.3 Post Class Activities for Faculty

You should familiarize yourself with the topics of the next session.

#### Tips:

You can also check the Articles/Blogs/Expert Videos uploaded on the OnlineVarsity site to gain additional information related to the topics covered in the next session. You can also connect to online tutors on the OnlineVarsity site to ask queries related to the sessions.

# Session 18 – Additional Features of Java 8

## 18.1 Pre-Class Activities

Before you commence the session, you should familiarize yourself with the topics of this session in-depth. Prepare a question or two that will be a key point to relate the current session objectives.

### 18.1.1 Objectives

By the end of this session, learners will be able to:

- Explain the Nashorn Engine
- Describe the jjs tool and its use for scripting
- Explain the new mathematical functions in Java 8

### 18.1.2 Teaching Skills

To teach this session, you should be well versed with JavaScript and Nashorn. You should be aware of how to invoke JavaScript code from within Java programs through Nashorn.

You should teach the concepts in the theory class using the images provided. For teaching in the class, you are expected to use slides and LCD projectors.

#### Tips:

It is recommended that you test the understanding of the students by asking questions in between the class.

## In-Class Activities

Follow the order given here during In-Class activities.

### Overview of the Session

Give the students an overview of the current session in the form of session objectives. Read out the objectives on slide 2.

**Slide 2**

**Objectives**

- ❖ Explain the Nashorn Engine
- ❖ Describe the jjs tool and its use for scripting
- ❖ Explain the new mathematical functions in Java 8

© Aptech Ltd.

Mathematical Programming in Java - Week 2 / Slide 2

Show the slide 2 and give the students a brief overview of the current session in the form of session objectives. Tell the students that they will be introduced to the Nashorn Engine. Then, describe the jjs tool and explain how it is used for scripting. Finally, explain the new mathematical functions in Java 8.

## 18.2 In-Class Explanations

### Slides 3 and 4

Let us see an introduction to Nashorn.

**Introduction**      1/2

- ✓ Nashorn is a German term which means Rhinoceros.
- ✓ In Java, it refers to the newly included JavaScript engine.
- ✓ Java Nashorn is the replacement for existing JavaScript engine in versions up to Java 7 called the Rhino.
- ✓ Nashorn and Rhino implement JavaScript language for JVM.
- ✓ Slower functioning of Rhino caused the need for a new engine, Nashorn.
- ✓ JavaScript on JVM is friendly with Nashorn that helps in faster implementation of Java language.

© Aptech Ltd.

Mathematical Programming in Java - Week 2 / Slide 3

The screenshot shows a presentation slide with the following content:

- Main Goal of Nashorn:**
  - ✓ Provide a lightweight and high-performance JavaScript runtime in Java.
- Nashorn:**
  - ✓ Compiles JavaScript into Java bytecode
  - ✓ Uses InvokeDynamic API of JVM specification that makes it faster than its predecessor.
  - ✓ Also has a command line tool called jjs.
  - ✓ Helps developers embed JavaScript in applications and also invoke Java methods and classes from JavaScript code
  - ✓ Helps developers extend functionality of applications and make them more versatile

At the bottom left is the copyright notice: © Aptech Limited. At the bottom right is the page number: Additional Programming in Java Version 8 / Page 8.

Explain about Nashorn Engine. Explain the reason for Nashorn replacing the Rhino engine that is the JavaScript engine in versions up to Java 7. Also, list the advantages of using Nashorn.

Explain that:

Nashorn is a new JavaScript engine of Java 8. A German tank destroyer named Rhinoceros was used in World War II. Nashorn is a German term, which means Rhinoceros. Jim Laskey of Oracle was the main developer of Nashorn.

Using slide 4, explain the main goal of Nashorn.

Explain that:

Nashorn is made available to Java applications using `javax.script` API and also using a new command-line tool called `jjs`.

#### **Additional Information:**

Refer to the following links for more information on Nashorn:

<http://openjdk.java.net/jeps/174>

<https://docs.oracle.com/javase/8/docs/technotes/tools/unix/jjs.html>

<http://www.oracle.com/technetwork/articles/java/jf14-nashorn-2126515.html>

<http://winterbe.com/posts/2014/04/05/java8-nashorn-tutorial/>

## Slide 5

Let us explore the goals for which Nashorn was designed.

The slide has a light blue header bar with the title "Primary Goals of Nashorn". Below the title is a white rectangular box containing a bulleted list of 12 goals. To the right of the list is a graphic of a clipboard with several checkmarks and a pencil. At the bottom left is the copyright notice "© Aptech Limited." and at the bottom right is the slide number "Notes on Java Programming for Java Developers / Slide 5".

- ✓ Should pass ECMAScript-262 compliance tests
- ✓ Will be based on ECMAScript-262 Edition 5.1 language specification
- ✓ Applications should perform better than Rhino and memory usage capabilities should also be better than Rhino
- ✓ Mutual support must be provided for Java and JavaScript code access between both the languages
- ✓ A new command line tool called `jjs` should validate JavaScript codes
- ✓ Should have support for JSR 223 (`javax.script` API)
- ✓ Must be safe from additional security risks
- ✓ Libraries provided with Nashorn should behave correctly under localization
- ✓ Error messages and documentation should be mapped to meet international standards

Using this slide, explain the primary goals of Nashorn.

Explain that:

Nashorn follows the ECMAScript-262 Edition 5.1 language specification and complies with ECMAScript-262 compliance tests.

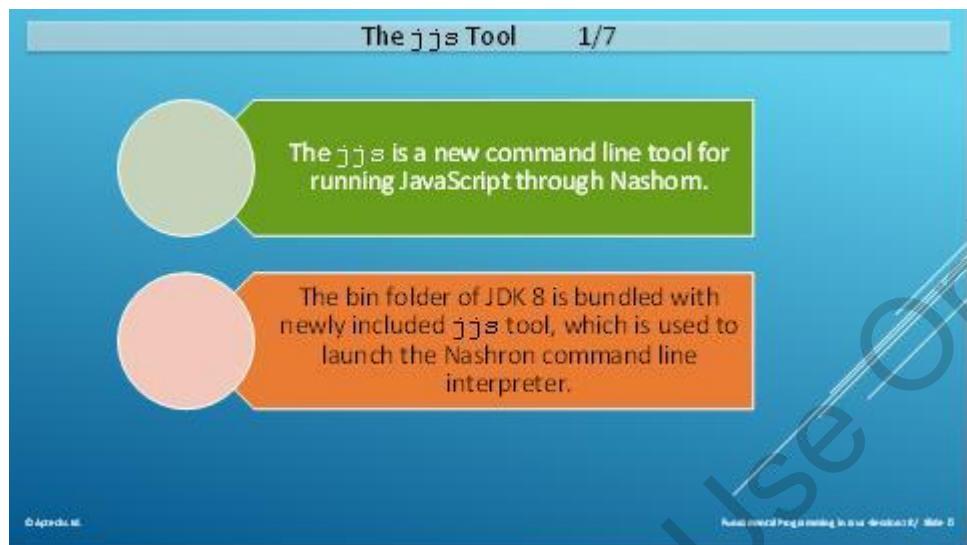
### Additional Information:

Refer to the following links to know more about Nashorn:

- <https://docs.oracle.com/javase/8/docs/technotes/tools/unix/jjs.html>
- <http://www.oracle.com/technetwork/articles/java/jf14-nashorn-2126515.html>
- <http://winterbe.com/posts/2014/04/05/java8-nashorn-tutorial/>
- <http://www.javaworld.com/article/2144908/scripting-jvm-languages/nashorn--javascript-made-great-in-java-8.html>

## Slide 6

Let us look at jjs tool.



Using slide 6, explain about jjs tool.

Explain that:

jjs is the command line tool used for running JavaScript through Nashorn. It is located in `JDK_HOME\bin` directory. You can use the command to execute shell scripts and script files.

### Additional Information:

Refer to the following link to know more about jjs:

<https://docs.oracle.com/javase/8/docs/technotes/tools/unix/jjs.html>

## Slides 7 and 8

Let us view some code demonstrating use of jjs tool.

The jjs Tool 2/7

Following Code Snippet shows how to execute a JavaScript program using Nashorn.

```
$jjs newfile.js
```

Following Code Snippet shows how to launch the jjs interpreter in interactive mode.

```
C:\>jjs
jjs> print ("hi everyone this is
Nashornjjs")
Hi everyone this is Nashornjjs
jjs>
```

To exit from the interactive mode of jjs, just type quit().

© Aptech Limited Additional Programming in Java - Session 07 / Slide 7

The jjs Tool 3/7

Consider an example where you want to quickly find the sum of some numbers.  
Enter the code shown in Code Snippet and save it as test.js

```
var data = [1,3,5,7,11]
var sum = data.reduce(function(x, y){ return x + y},0)
print(sum)
```

Then, give jjs test.js at the command prompt  
The output of the code when executed with jjs tool is 27.

© Aptech Limited Additional Programming in Java - Session 07 / Slide 8

Using slide 7, explain the code to execute a JavaScript program using Nashorn, launch the jjs interpreter in interactive mode and to exit from the interactive mode of jjs.

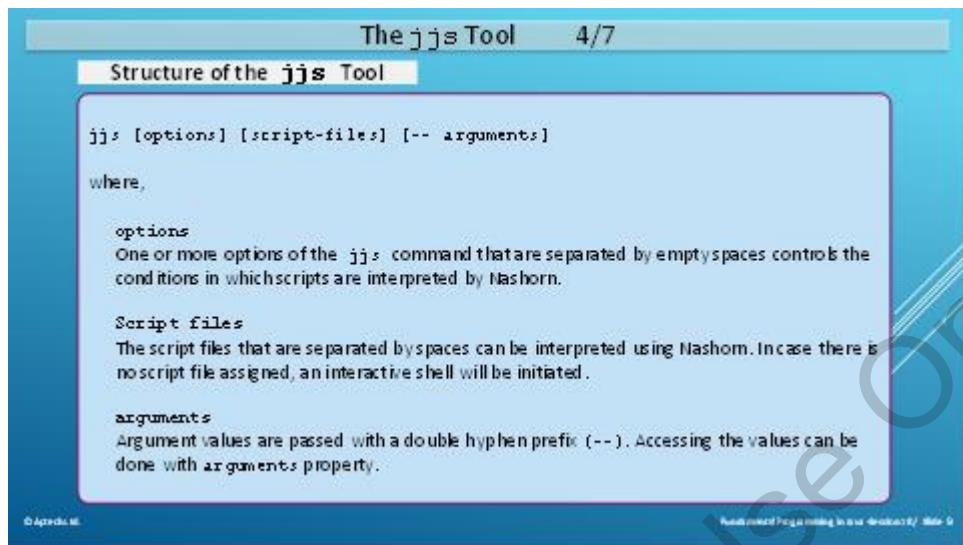
Using slide 8, explain the code to find the sum of some numbers.

Explain that:

In the code, the function returns sum of 1 and 3. Then, in the subsequent iterations, the function adds the result to the next number in the array. Finally, the answer obtained is 27.

## Slide 9

Let us understand the structure of jjs tool.



Explain the structure of jjs tool.

You can use the following syntax to print a list of options:

```
jjs -xhelp
```

The options `-classpath <path>` and `-cp <path>` both are used to set CLASSPATH.

`-fv` option prints the full version of the Nashorn engine. Some options may start with double hyphens.

If you do not mention any option, jjs is run in interactive mode. In interactive mode, the script is interpreted as you enter it.

## Slide 10

Let us view some code demonstrating interactive mode using jjs.

The screenshot shows a terminal window titled "The jjs Tool 5/7". It displays the following code snippet:

```
jjs -D sampleKey=sampleValue
jjs>java.lang.System.getProperty("sampleKey")
sampleValue
jjs>
```

A callout box above the code states: "Code Snippet shows how to invoke Nashorn in interactive mode and assign `sampleValue` to the property named `sampleKey` and then retrieve it using `getProperty()`". Below the code, a message says: "The code can be repeated with appropriate modifications to set multiple properties". The bottom right corner of the window shows "ib Aptech".

Using this slide, explain the code that shows how to invoke Nashorn in interactive mode and assign `sampleValue` to the property named `sampleKey` and then retrieve it using `getProperty()` method.

Explain that:

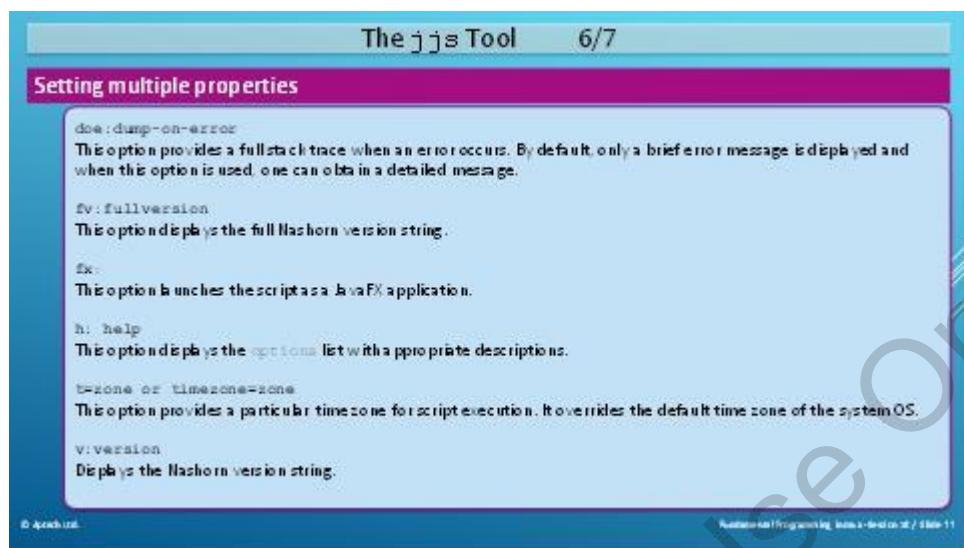
The syntax for `-D` option is  
`-D<name>=<value>`

You can use the option to set system properties for Java runtime.

option `-Dname = value` can be repeated with appropriate modifications to set multiple runtime properties.

**Slide 11**

Let us understand how to set multiple properties with jjs.



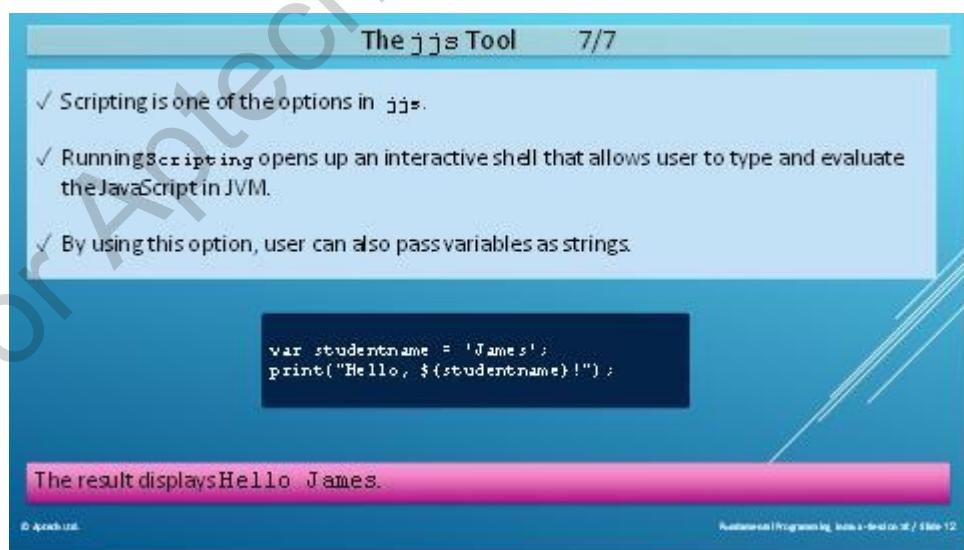
Using this slide, explain the method to set multiple properties in the jjs tool.

Explain that:

jjs tool automatically enables the scripting mode if a script file run by the jjs tool begins with # sign. Then, the entire script is executed in scripting mode.

**Slide 12**

Let us understand scripting in jjs.



Using this slide, explain about the scripting option of jjs tool.

Explain that:

Using the `-scripting` option, jjs is run in scripting mode. The option allows you to use any operating system-specific shell commands.

When a script file begins with the `#` sign, executing the script file automatically enables scripting mode.

### Slides 13 to 17

Let us explore the ScriptEngine in Nashorn.

**ScriptEngine 1/5**

- ✓ The `ScriptEngine` interface and `ScriptEngineManager` class are used to run JavaScript in Java
- ✓ These are defined in public API `javax.script` and must be imported into your application.
- ✓ When Oracle `Nashorn` is available, it can be accessed using the identifier, `nashorn`.
- ✓ Code Snippet here shows the process of importing `ScriptEngine` in a Java program.
- ✓ `ScriptException` class is imported to handle any script-related exceptions

```
import javax.script.ScriptEngine; // to use ScriptEngine
import javax.script.ScriptEngineManager; // to use ScriptEngineManager
import javax.script.ScriptException; // to handle exceptions
```

© Aptech Limited. Autumnal | Programming Java - Detailed / Slide 13

**ScriptEngine 2/5**

`ScriptEngineManager` is used to initiate Nashorn engine

Following Code Snippet shows how to use `ScriptEngineManager` in Java

```
ScriptEngineManager newEngManager = new ScriptEngineManager();
ScriptEngine newEng = newEngManager.getEngineByName("nashorn");
```

© Aptech Limited. Autumnal | Programming Java - Detailed / Slide 14

**ScriptEngine 3/5**

- After the import of `ScriptEngine` and `ScriptEngineManager`, evaluation of JavaScript can be done anytime.
- Code Snippet shows the evaluation of JavaScript.
- Note that it is mandatory to enclose the code in a try-catch block that will handle the `ScriptException`, failing which, there will be compiler errors.

```
try{
 newEng.eval("function f(g) { print(g) }"); //evaluation of JavaScript
 newEng.eval("f(' Hi this is through JavaScript being executed from within
Java')"); //displays result
} catch(ScriptException e){...}
```

© Aptech      Aptech Programming Java - 13 of 10 / 100-10

**ScriptEngine 4/5**

In the code shown earlier:

- A JavaScript function `f` is defined that takes one argument and prints its value.
- This function is then called in the next statement and a string argument is passed.
- As a result, when the code is executed, the string is displayed in the output.

A `FileReader` can also be passed as an input in the evaluation method (`eval`).

Following Code Snippet demonstrates this. Ensure that appropriate exception handling is done while using this or similar code dealing with files.

```
newEng.eval(new FileReader('newfile.js'));//as input
```

© Aptech      Aptech Programming Java - 13 of 10 / 100-10

**ScriptEngine 5/5**

In this code, `newManager` is an instance of `ScriptEngineManager`. It will be used to create an instance of `ScriptEngine` using the `getEngineByName()` method. Through `newEng`, the JavaScript `eval` method is applied to evaluate the variables `x`, `y`, and `z`.

```
import javax.script.ScriptEngineManager; //to use ScriptEngineManager
import javax.script.ScriptEngine; //to use ScriptEngine
import javax.script.*;
public class ScriptEngineUsage {
 public static void main(String args[]){ throw ScriptException
 {
 ScriptEngineManager newEngManager = new ScriptEngineManager();
 ScriptEngine newEng = newEngManager.getEngineByName("javascript");
 newEng.eval("var x = 10;");
 newEng.eval("var y = 20;");
 newEng.eval("var z = x + y;");
 newEng.eval("print(z);");//displays result
 }
}
```

Output:  
30

© Aptech      Aptech Programming Java - 13 of 10 / 100-10

Using slides 13 and 14, explain about the `ScriptEngine` interface and `ScriptEngineManager` class.

**Explain that:**

The instances of `ScriptEngine` interface executes scripts written in a scripting language.

Using slide 14, explain about the `ScriptEngineManager` class. Explain the code that shows how to use `ScriptEngineManager` class in Java.

**Explain that:**

The method in the following code returns the final value as an object:

```
import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
import javax.script.ScriptException;

public class mymath {
 public static void main(String[] args) throws ScriptException {
 ScriptEngineManager manager = new ScriptEngineManager();
 ScriptEngine engine = manager.getEngineByName("Nashorn");

 Object findings= null;
 findings = engine.eval("5 * 6; 2 * 4; 3 * 4; 2 * 9; 2 * 3;");
 System.out.println(findings);

 }
}
```

On compilation, the answer obtained will be 6.

Explain how evaluation of JavaScript is done.

**Explain that:**

`ScriptException` is a checked Exception, so the compiler will force you to handle this. The try-catch block is necessary in a code to handle the `ScriptException` properly.

**Additional Information:**

Refer the following link of sample codes to learn on `ScriptException`.

<https://github.com/shekhargulati/java8-the-missing-tutorial/blob/master/10-nashorn.md>

Using slide 16, continue to explain the code and then, describe `FileReader` instance and explain the code that demonstrates the usage of `FileReader` instance.

**Explain that:**

A js file can be parsed by passing a `FileReader` object pointing to the file. For example,

```
ScriptEngine myengine = new
ScriptEngineManager().getEngineByName("nashorn");
```

```
myengine.eval(new FileReader("newfile.js"));
You can also pass a string to the eval method of the engine object.
```

```
ScriptEngine myengine = new
 ScriptEngineManager().getEngineByName("nashorn");
myengine.eval("Learning Nashorn");
```

Using slide 17, explain the code that demonstrates the usage of newEngManager, an instance of ScriptEngineManager.

Explain that:

You can use Nashorn engine names such as "nashorn", "javascript", and "js".

### Slides 18 and 19

Let us explore how to import Java classes and packages into JavaScript.

**Importing Classes and Packages 1/2**

Code Snippet shows importing of `java.util` and other packages. This code needs to be saved as a `.js` file and then, executed with `jjs` tool.

```
var imports = new JavaImporter(java.util,java.io,java.nio.file);
with(imports){// importing packages
 var samplePaths = new LinkedList();
 println(samplePaths instanceof LinkedList); //true
 samplePaths.add("newDoc1");
 samplePaths.add("newDoc2");
 samplePaths.add("newDoc3");
 println(samplePaths); // [newDoc1, newDoc2, newDoc3]
}//display result.
```

In this code:

- Variable `samplePaths` is created as an instance of `LinkedList`, and using `add()` method, a list of nodes is created and displayed.
- Code assumes there are three files, `newDoc1`, `newDoc2`, and `newDoc3` created in current folder.

**Importing Classes and Packages 2/2**

Following snippet shows the usage of `.write` method:

```
for(var x=0; x<samplePaths.size();x++)
 FileSystems.getDefault().getPath(samplePaths.get(x))
 .write("test\n".getBytes())
```

Note that Nashorn allows importing existing Java classes, but creating new classes is also possible with this new JavaScript engine.

Using slides 18 and 19, explain the code to import `java.util` package and other packages into a JavaScript program so that Java classes, interfaces, and methods can be used in JavaScript.

**Explain that:**

`JavaImporter` is a function object that accepts a list of Java packages and classes.

To import all classes from the `java.lang` package, you can specify:

```
var importlangpackage = new JavaImporter(Packages.java.lang);
```

Similarly, you can import all classes from various packages. For example, to import the `java.lang` and `java.util` packages, you can specify:

```
var importlangpackagenew = JavaImporter(java.lang, java.util);
```

Note that, in this code, the `new` operator is not used, because here, `JavaImporter` is being used as a function.

Using slide 19, explain the code that shows usage of `.write()` method.

**Explain that:**

To write a string to a stream, it must be converted to bytes.

The `.write()` method helps to write character data to a stream. `getBytes()` method returns the byte array of the string.

#### **Additional Information:**

Refer to the following link for more practical examples:

<https://github.com/gAmUssA/java-scripting-experiments>

## Slide 20

Let us look at how to extend Java classes and interfaces in JavaScript.

The slide has a blue header bar with the title "Extending Classes and Interfaces". Below it is a light blue section containing the text: "Java.type and Java.extend functions are used to extend Java classes". A red box highlights the following text: "Following Code Snippet shows an example with Callable interface and call method implementation. This code needs to be saved as a .js file and executed with jjs." The main content area contains a block of JavaScript code:

```

var newConcur = new JavaImporter(java.util,java.util.concurrent);
//extending using Java.type
var newCall = Java.type("java.util.concurrent.Callable");
with(newConcur){
 var newExecutor = new java.util.concurrent.ThreadPoolExecutor();
 var newTask = new java.util.LinkedHashMap();
 for(var x=1;x<200;x++){
 var SampleTask = Java.extend(newCall, {call:Function});
 SampleTask.prototype = new java.util.concurrent.Callable();
 SampleTask.prototype.call = function(){
 print("Result displayed as "+x);
 }
 var newTaskA = new SampleTask();
 newTask.add(newTaskA);
 newExecutor.submit(newTaskA);
 }
}
//display result

```

A pink box at the bottom left contains the note: "Here, newConcur is defined as a JavaImporter instance. newCall will represent the Callable type. A new LinkedHashMap is created and sub tasks are added to it. Using a cached thread pool a list of numbers are then displayed as output." At the bottom right, there is a small note: "Additional Programming Java 8 - Thread / Slide 20".

Using this slide, describe the functions that are used to extend Java classes. Explain the code that shows an example with Callable interface and call method implementation.

Explain that:

Callable interface binds a task and passes it to a thread pool for executing it asynchronously. Callable interface returns a value. The interface consists a call() method that holds the code that is to be executed asynchronously.

### Additional Information:

Refer to the following link for more information:

<http://winterbe.com/posts/2015/04/07/java8-concurrency-tutorial-thread-executor-examples/>

## Slides 21 and 22

Let us examine the Invocable interface.

The image shows two slides from a presentation. Both slides have a blue header bar with the text "Invocable Interface" and a page number (1/2 for the first slide, 2/2 for the second). A large watermark reading "Aptech © All rights reserved" is diagonally across both slides.

**Slide 1/2 Content:**

An **Invocable Interface** initiation helps in invoking JavaScript directly from Java. For this, Java objects are passed as function arguments and the resultant data can be returned back to the Java method.

```
Invocable tryInvoke = (Invocable) engine;//to invoke engine
```

**Slide 2/2 Content:**

**invokeFunction()** method can be applied to invoke any user-defined function.

Code Snippet shows `invokeFunction()` method.

```
engine.eval("function f(g) { print(g) }");
tryInvoke.invokeFunction("f","HI");// invoke function usage
```

Using slide 21, describe Invocable Interface.

Explain that:

Invocable interface helps to invoke procedures, functions, and methods. A script engine class invoking procedure must implement the Invocable interface. The `invokeMethod()` method helps to invoke a method of an object. The `invokeFunction()` method helps to invoke the top level functions in the script. The `getInterface()` method is used to get the implementation of a Java interface.

Using slide 22, describe the `invokeFunction()` method.

**Additional Information:**

Refer to the following link for more examples on Invocable interface:

<http://www.programcreek.com/java-api-examples/index.php?api=javax.script.Invocable>

**Slides 23 to 29**

Let us explore mathematical methods in Java 8.

The image contains two vertically stacked screenshots of a presentation slide titled "Mathematical Methods in Java".

**Top Screenshot (Slide 1/7):** The title is "Mathematical Methods in Java". Below it, a text box states: "Basic mathematical operations such as logarithms, exponential square root, numeric calculations, and trigonometric methods are made easier with `java.lang` package". A section titled "Basic Math Methods:" lists three methods: `Math.abs()`, `Math.ceil()`, and `Math.floor()`.

**Bottom Screenshot (Slide 2/7):** The title is "Mathematical Methods in Java". It shows the `Math.abs()` method definition: 

```
public class BasicMathDemo { public static void main(String args[]) { int abs1 = Math.abs(10); // abs1 = 10 int abs2 = Math.abs(-20); // abs2 = 20 System.out.println("Result A: " +abs1); System.out.println("Result B: " +abs2); } // displays result}
```

 A note below the code says: "Here, the code produces output as the same value as the input. However, the negative value (-20) is returned as positive value (20)."

**Mathematical Methods in Java**      3/7

**Math.abs() method can be overloaded by using one of these:**

int      double      long      float

Usage of the overloaded methods may depend on the respective parameters and operations performed.

© Aptech

Aptech Programming Java 8-Test 01 / 8 Min 25

**Mathematical Methods in Java**      4/7

**Math.ceil()**

is used to round up a floating-point value into integer value.

```
public class BasicMathDemo {
 public static void main(String args[]) {
 double objCeil = Math.ceil(6.454);
 System.out.println("Result as " + objCeil);
 }
} // displays result
```

After executing this code, objCeil contains value 7.0.

© Aptech

Aptech Programming Java 8-Test 01 / 8 Min 25

**Mathematical Methods in Java**      5/7

**Math.floor()**

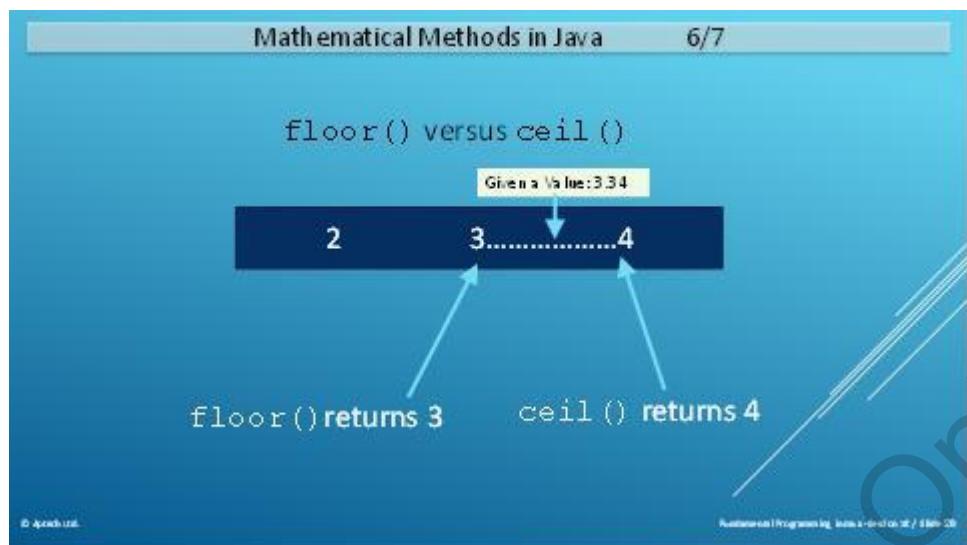
- ◆ This method rounds down a floating-point value down to the closest integer value.
- ◆ The rounded value is displayed as a double.
- ◆ Example given here demonstrates the Math.floor() method.

```
public class BasicMathFunctions {
 public static void main(String args[]) {
 double objFloor = Math.floor(6.454);
 // will result in objFloor = 6.0
 System.out.println("Result as " + objFloor);
 }
} // displays result
```

After executing this code, objFloor contains value 6.0.

© Aptech

Aptech Programming Java 8-Test 01 / 8 Min 25



| Mathematical Methods in Java               |                                                                                                                                                                          |                                                                                                                                                                                    |
|--------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 7/7                                        |                                                                                                                                                                          |                                                                                                                                                                                    |
| Table lists a few more basic Math methods: |                                                                                                                                                                          |                                                                                                                                                                                    |
| Method Name                                | Description                                                                                                                                                              | Example                                                                                                                                                                            |
| Math.floorDiv()                            | Similar to floor(), but is combined with division. Math.floorDiv() divides an integer or long value by another one, and rounds resultant value to closest integer value. | double result = Math.floorDiv(100, 9);<br>Output: 11.0                                                                                                                             |
| Math.min()                                 | Produces the smallest of given values passed as inputs.                                                                                                                  | int minVal = Math.min(5, 12);<br>Here, minVal variable produces output as 5                                                                                                        |
| Math.max()                                 | Similar to Math.min() with one difference that it produces the biggest value of the given inputs.                                                                        | int maxVal = Math.max(5, 12);<br>Here, maxVal variable produces output as 12.                                                                                                      |
| Math.round()                               | Rounds a float or double to the closest integer. Math.round() method applies common mathematical rules for rounding the values.                                          | double roundDown = Math.round(14.524);<br>double roundUp = Math.round(14.654);<br><br>Here, roundDown variable produces output as 14.0 and roundUp variable produces 15 as output. |
| Math.random()                              | Returns a random floating point value between 0 and 1 by default. However, Math.random can be applied to get a random number between 0 and n (any number within 100).    | double randomVal1 = Math.random();<br>double randomVal2 = Math.random() * 100;<br><br>Output:<br>randomVal1=0.746356203131188<br>randomVal2=26.36046906579073                      |

Using slide 23, state various mathematical methods available in `java.lang` package. Tell the students that they may need to use these methods in complex calculations or in calculations related to everyday tasks too. For example, while calculating price of products or weight of some item.

Using slide 24, explain `Math.abs()` method and using slide 25, explain various parameters that are used to overload `Math.abs()` method.

Using slide 26, explain the `Math.ceil()` method and using slide 27, explain the `Math.floor()` method.

Using slide 28, explain the difference between the `Math.floor()` and `Math.ceil()` method, tell them the `Math.floor()` and `Math.ceil()` methods works fine in itself, but they will return same value only when the input value is an integer. Using slide 29, explain the remaining Math methods.

**Additional Information:**

Refer the following links for more information:

[http://www.jchq.net/tutorial/09\\_01Tut.htm](http://www.jchq.net/tutorial/09_01Tut.htm)  
<http://www.cafeaulait.org/course/week4/40.html>

**In-Class Question:**

After you finish explaining slide 29, you will ask the students some In-Class questions. This will help you in reviewing their understanding of the topic.



What will be output of the statement `System.out.println(Math.abs(-52.1))`?

**Answer:** 52.1

Code:

```
import java.lang.Math;
public class BasicMathDemo {
 public static void main(String args[]) {
 System.out.println(Math.abs(-52.1));
 }
}
```



What will be output of the statement `System.out.println(Math.ceil(-52.1));`?

**Answer:** -52.0

Code:

```
import java.lang.Math;
public class BasicMathDemo {
 public static void main(String args[]) {
 System.out.println(Math.ceil(-52.1));
 }
}
```



What will be output of the statement  
`System.out.println(Math.floor(-52.1));`?

**Answer:** -53.0

Code:

```
import java.lang.Math;
public class BasicMathDemo {
 public static void main(String args[]) {
```

```

System.out.println(Math.floor(-52.1));
}
}

```

### Slides 30 to 34

Let us understand the usage of exponential and logarithmic methods in Java 8.

**Exponential and Logarithmic Math Methods** 1/5

Math class contains methods for exponential and logarithmic calculations such as:

The diagram consists of five hexagonal icons arranged in a cluster. The top-left icon is red and contains "Math.exp()". The top-right icon is orange and contains "Math.sqrt()". The bottom-left icon is green and contains "Math.log()". The bottom-middle icon is red and contains "Math.pow()". The bottom-right icon is teal and contains "Math.log10()".

© Aptech Limited. Author(s) / Program(s) used in this document / Slide 30

**Exponential and Logarithmic Math Methods** 2/5

Math. exp () produces e (Euler's number) increased to the power of the value given as a parameter.

Following example shows Math. exp () in use.

```

public class ExpandoLogMathFunctions {
 public static void main(String args[]) {
 double newExpA = Math.exp(4);
 System.out.println("OutputA = " + newExpA);
 double newExpB = Math.exp(5);
 System.out.println("OutputB = " + newExpB);
 }
} // displays result

```

Here, newExpA and newExpB variables produce the following output:

OutputA = 54.598150033144236  
OutputB = 148.4131591025766

© Aptech Limited. Author(s) / Program(s) used in this document / Slide 31

**Exponential and Logarithmic Math Methods**      3/5

Logarithm can be obtained using `Math.log()`. It also uses Euler's number e as the base. This method also performs the reverse function of `Math.exp()`.

Following example demonstrates `Math.log()` method.

```
public class ExpoandLogMathFunctions {
 public static void main(String args[]) {
 double newLogA = Math.log(2);
 System.out.println("OutputA = " + newLogA);
 double newLogB = Math.log(100);
 System.out.println("OutputB = " + newLogB);
 }
} // displays result
```

Here, `newLogA` and `newLogB` are variables that produce following result:

```
OutputA = 0.6931471805599453
OutputB = 4.605170185988092
```

Aptech Ltd.      Business Application Programming, Java - A Detailed / 100% 12

**Exponential and Logarithmic Math Methods**      4/5

- ◆ `Math.pow()` method takes two parameters and produces the value of the first parameter raised to the power of the second parameter.
- ◆ Following example demonstrates the `Math.pow()` method.

```
public class ExpoandLogMathFunctions {
 public static void main(String args[]){
 double newPowerA = Math.pow(2, 4);
 System.out.println("OutputA as = " + newPowerA);
 double newPowerB = Math.pow(2, 5);
 System.out.println("OutputB as = " + newPowerB);
 }
}
```

Here, `newPowerA` and `newPowerB` produce following results:

```
OutputA as = 16.0
OutputB as = 32.0
```

Aptech Ltd.      Business Application Programming, Java - A Detailed / 100% 35

**Exponential and Logarithmic Math Methods**      5/5

- ◆ `Math.sqrt()`
- ◆ This method performs the square root operation for the given parameter.
- ◆ Example here demonstrates `Math.sqrt()` method.

```
public class ExpoandLogMathFunctions {
 public static void main(String args[]){
 double newRootA = Math.sqrt(8);
 System.out.println("OutputA = " + newRootA);
 double newRootB = Math.sqrt(25);
 System.out.println("OutputB = " + newRootB);
 }
}
```

Here, the `newRootA` and `newRootB` produce the following results:

```
OutputA = 2.8284271247461903
OutputB = 5.0
```

Aptech Ltd.      Business Application Programming, Java - A Detailed / 100% 36

Using slides 30 to 34, explain various exponential and logarithmic Math methods in Math class. These methods include Math.exp(), Math.log(), Math.pow(), and Math.sqrt().

**Additional Information:**

Refer the following links for more information:

[http://www.jchq.net/tutorial/09\\_01Tut.htm](http://www.jchq.net/tutorial/09_01Tut.htm)

<http://www.cafeaulait.org/course/week4/40.html>

**In-Class Question**

After you finish explaining slide 34, you will ask the students some In-Class questions. This will help you in reviewing their understanding of the topic.



What will be the output of the expression Math.exp(0)?

**Answer:** 1.0

Code demonstrating the output:

```
import java.lang.Math;
public class BasicMathDemo {
 public static void main(String args[]) {
 System.out.println(Math.exp(0));
 }
}
```



What will be the output of the expression Math.log(1)?

**Answer:** 0.0

Code demonstrating the output:

```
import java.lang.Math;
public class BasicMathDemo {
 public static void main(String args[]) {
 System.out.println(Math.log(1));
 }
}
```



What will be the output of the expression `Math.sqrt(7)`?

**Answer:** 2.6457513110645907

Code demonstrating the output:

```
import java.lang.Math;
public class BasicMathDemo {
 public static void main(String args[]) {
 System.out.println(Math.sqrt(7));
 }
}
```



What will be the output of the expression `Math.pow(7, 2)`?

**Answer:** 49.0

Code demonstrating the output:

```
import java.lang.Math;
public class BasicMathDemo {
 public static void main(String args[]) {
 System.out.println(Math.pow(7, 2));
 }
}
```

## Slides 35 to 45

Let us explore the various trigonometric methods available in Math class.

**Trigonometric Math Methods 1/11**

- ◆ The **Math** class also includes a set of trigonometric methods that can calculate values used in trigonometry such as sine, cosine, tan, and so on.
- ◆ **Math.PI** is a constant double that contains a value closest to PI.
- ◆ **Math.PI** is frequently used in trigonometric operations/calculations.

**Trigonometric Math Methods 2/11**

| Method Name       | Description                                                                                    | Example                                                                                                                                                                                   |
|-------------------|------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Math.sin()</b> | Performs the sine operation. It calculates the sine value of the given angle value in radians. | <pre>double newSin =<br/>Math.sin(Math.PI);<br/><br/>System.out.println("The<br/>value of sin = " +<br/>newSin);<br/><br/>Output:<br/>The value of sin =<br/>1.2246467991473532E-16</pre> |

**Trigonometric Math Methods 3/11**

| Method Name       | Description                                                                                     | Example                                                                                                                                                                                   |
|-------------------|-------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Math.cos()</b> | Performs the cos operation. It calculates the cosine value of the given angle value in radians. | <pre>double newTan =<br/>Math.tan(Math.PI);<br/><br/>System.out.println("The<br/>value of tan = " +<br/>newTan);<br/><br/>Output:<br/>The value of tan =<br/>1.2246467991473532E-16</pre> |

**Trigonometric Math Methods**      4/11

| Method Name | Description                                                          | Example                                                                                                                                                                 |
|-------------|----------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Math.asin() | Performs the arc sine value calculation of a value between 1 and -1. | <pre>double newAsin =<br/>Math.asin(Math.PI);<br/><br/>System.out.println("The<br/>value of Asin = " +<br/>newAsin);<br/><br/>Output:<br/>The value of Asin = NaN</pre> |

**Trigonometric Math Methods**      5/11

| Method Name | Description                                                         | Example                                                                                                                                                             |
|-------------|---------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Math.acos() | Performs the arc cos value calculation of a value between 1 and -1. | <pre>double newAcos =<br/>Math.acos(1.0);<br/><br/>System.out.println("The<br/>value of acos = " +<br/>newAcos);<br/><br/>Output:<br/>The value of acos = 0.0</pre> |

**Trigonometric Math Methods**      6/11

| Method Name | Description                                                             | Example                                                                                                                                                                                |
|-------------|-------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Math.atan() | Performs the arc tangent value calculation of a value between 1 and -1. | <pre>double newAtan =<br/>Math.atan(1.0);<br/><br/>System.out.println("The<br/>value of Atan = " +<br/>newAtan);<br/><br/>Output:<br/>The value of Atan =<br/>0.7853981633974483</pre> |

Trigonometric Math Methods      7/11

| Method Name              | Description                                                                       | Example                                                                                                                                            |
|--------------------------|-----------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>Math.sinh()</code> | Performs the hyperbolic sine value calculation of a given value between 1 and -1. | <pre>double newSinh = Math.sinh(1.0);  System.out.println("The value of sinh = " + newSinh);  Output: The value of sinh = 1.1752011936428014</pre> |

© Aptech Limited  
Autumnal Programming Java 8-Part 01 / Slide 41

Trigonometric Math Methods      8/11

| Method Name              | Description                                                                         | Example                                                                                                                                           |
|--------------------------|-------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>Math.cosh()</code> | Performs the hyperbolic cosine value calculation of a given value between 1 and -1. | <pre>double newCosh = Math.cosh(1.0);  System.out.println("The value of cosh = " + newCosh);  Output: The value of cosh = 1.542080634815244</pre> |

© Aptech Limited  
Autumnal Programming Java 8-Part 01 / Slide 42

Trigonometric Math Methods      9/11

| Method Name              | Description                                                                          | Example                                                                                                                                            |
|--------------------------|--------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>Math.tanh()</code> | Performs the hyperbolic tangent value calculation of a given value between 1 and -1. | <pre>double newTanh = Math.tanh(1.0);  System.out.println("The value of tanh = " + newTanh);  Output: The value of tanh = 0.7615941559557649</pre> |

© Aptech Limited  
Autumnal Programming Java 8-Part 01 / Slide 43

**Trigonometric Math Methods 10/11**

| Method Name                   | Description                                                       | Example                                                                                                                                      |
|-------------------------------|-------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------|
| <code>Math.toDegrees()</code> | Performs the convert operation of an angle in radians to degrees. | <pre>double newDegrees =<br/>Math.toDegrees(Math.PI)<br/><br/>System.out.println("Output<br/>= " + newDegrees);<br/><br/>Output= 180.0</pre> |

**Trigonometric Math Methods 11/11**

| Method Name                   | Description                                                                                                                                | Example                                                                                                                                               |
|-------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>Math.toRadians()</code> | Performs an reverse operation of <code>Math.toDegrees()</code> method; it performs the convert operation of an angle indegrees to radians. | <pre>double newRadians =<br/>Math.toRadians(180);<br/><br/>System.out.println("Output<br/>= " + newRadians);<br/><br/>Output= 3.141592653589793</pre> |

Using slides 35 to 45, explain about the various trigonometric methods available in Math class.

#### Additional Information:

Refer to the following links for more information:

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Math/sin](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/sin)  
<http://www.cafeaulait.org/course/week4/40.html>

**In-Class Questions:**

After you finish explaining slide 34, you will ask the students some In-Class questions. This will help you in reviewing their understanding of the topic.



What will be the output of

```
System.out.println(Math.sin(Math.PI/2));?
```

**Answer:** 1.0

**Code:**

```
import java.lang.Math;
public class BasicMathDemo {
 public static void main(String args[]) {
 System.out.println(Math.sin(Math.PI/2));
 }
}
```



What will be the output of System.out.println(

```
Math.sinh(0));?
```

**Answer:** 0.0

**Code:**

```
import java.lang.Math;
public class BasicMathDemo {
 public static void main(String args[]) {
 System.out.println(Math.sinh(0));
 }
}
```



What will be output of System.out.println(Math.cosh(0));?

**Answer:** 1.0

**Code:**

```
import java.lang.Math;
public class BasicMathDemo {
 public static void main(String args[]) {
 System.out.println(Math.cosh(0));
 }
}
```



What will be the output of `System.out.println(Math.cos(2 * Math.PI));?`

**Answer:** 1.0

**Code:**

```
import java.lang.Math;
public class BasicMathDemo {
 public static void main(String args[]) {
 System.out.println(Math.cos(2 * Math.PI));
 }
}
```



What will be output of `System.out.println(Math.tanh(0));?`

**Answer:** 0.0

**Code:**

```
import java.lang.Math;
public class BasicMathDemo {
 public static void main(String args[]) {
 System.out.println(Math.tanh(0));
 }
}
```

**Slides 46 and 47**

Let us examine exact numeric operations.

Exact Numeric Operations 1/2

Following methods are used to perform exact numeric operations:

- addExact
- subtractExact
- multiplyExact
- incrementExact
- decrementExact
- negateExact
- toIntExact

Exact Numeric Operations 2/2

Example given here demonstrates the usage of `addExact()` method.

```
public class ExactMethodDemo {
 public static void main(String args[]) {
 int ex1 = 90000000; //
 int ex2 = 125000000; //
 System.out.println(Math.addExact(ex1, ex2));
 }
} // displays result
```

Ideally, if `ex1` and `ex2` were added using the `+` sign, the program will not show any errors instead, it will produce an inaccurate result as it exceeds the maximum limit for integers.

Hence, here using `addExact()` is recommended so that it would throw an exception and alert the user instead of displaying an incorrect output.

Using slide 46, explain various methods that are used to perform exact numeric operations. Mention that this category of methods is new in Java 8. They did not exist in Java 7.

Explain that:

`addExact()` : Returns the sum of its argument.

`subtractExact()` : Returns the difference of its arguments.

`multiplyExact()` : Returns the product of its arguments.

`incrementExact()` : Returns the argument incremented by one.

`decrementExact()` : Returns the argument decremented by one.

`negateExact()` : Returns the negation of the argument.

`toIntExact()` : Returns the value of the long argument.

**Additional Information:**

Refer to the following link for more information:

<http://javatutorialhq.com/java/lang/math-class-tutorial/>

Using slide 47, demonstrate the usage of `addExact()` method using the code shown on the slide.

Explain that:

In the code, `addExact()` method returns the sum of `ex1` and `ex2`. `ex1` and `ex2` must be of datatype `long` and the result is also of datatype `long`. The method throws an exception if the result overflows a `long`.

**Slides 48 and 49**

Let us explore next numeric operations.

Next Numeric Operations 1/2

Methods that perform numeric operations where there is a need to display the closest value of a given number:

© Aptech Limited

Aptech Programming Java 8-Module 2 / Slide 48

Next Numeric Operations 2/2

Example given here demonstrates the `nextDown()` method.

```
public class ExactMethodDemo {
 public static void main(String args[]) {
 int nextA = 1000; // ...
 System.out.println(Math.nextDown(nextA));
 }
} // displays result
```

Output:  
999.99994

© Aptech Limited

Aptech Programming Java 8-Module 2 / Slide 48

Using slide 48, explain various methods that are used to perform next numeric operations. Mention that this category of methods is new in Java 8. They did not exist in Java 7.

Explain that:

The syntax of `nextAfter()` method is as follows:

```
public static double nextAfter(double start, double direction)
```

`nextAfter()` method returns the floating-point number next to the first argument in the path of the second argument.

Following code shows usage of `nextAfter` method:

```
double x=12345.678;
double y=125.6589;
System.out.println("You will see value after x: " +
Math.nextAfter(x, y));
```

If you execute the code, the answer will be:

You will see value after x: 12345.677999999998

#### In-Class Question:

Ask the students what would be the answer if they interchange x and y.

Hint:

```
double x=12345.678;
double y=125.6589;
System.out.println("Now You will see value after y: " +
Math.nextAfter(y, x));
```

If you execute the code, the answer will be:

Now, you will see value after y: 125.65890000000002

Using slide 49, explain about the `nextDown()` method with the help of code on the slide.

**Slide 50**

Let us summarize the session.

**Summary**

- Nashorn and Rhino implements a JavaScript engine to enable its use with JVM. Slower functioning of Rhino caused the need for Nashorn
- jjs is a command line tool to launch Nashorn
- Various jjs command options control the conditions in which scripts are interpreted by Nashorn
- Nashorn enables JavaScript functions to be invoked directly from Java. Also, Java objects can be passed as function arguments, and the resultant data can be returned back to the Java method
- Advanced mathematical operations can be performed with new methods in Math class

© Aptech Limited. All rights reserved. This program is designed for educational purposes only.

Use this slide to summarize the session. You will end the session, with a brief summary of what has been taught in the session. Tell the students pointers of the session. This will be a revision of the current session.

Explain that:

Nashorn and Rhino implements a JavaScript engine to enable its use with JVM. Slower functioning of Rhino caused the need for Nashorn. jjs is a command line tool to launch Nashorn. Various jjs command options control the conditions in which scripts are interpreted by Nashorn. Nashorn enables JavaScript functions to be invoked directly from Java. Also, Java objects can be passed as function arguments, and the resultant data can be returned back to the Java method. Advanced mathematical operations can be performed with new methods in Math class.

### 18.3 Post Class Activities for Faculty

**Tips:**

You can also check the Articles/Blogs/Expert Videos uploaded on the OnlineVarsity site to gain additional information related to the topics covered in the session. You can also connect to online tutors on the OnlineVarsity site to ask queries related to the sessions.