`

“Beyond Obvious” Icons

In the student book, we have included special icons (Beyond Obvious Icons) in the form of footnotes that are interspersed in the study material to give you the precise context of the concept you are learning. As you get accustomed to this way of learning, you will enjoy the fun of it which will make this learning highly productive!

This icon indicates a particular best practice which is followed while developing the applications, in design, in coding etc. Knowledge of best practices makes one a good developer or designer.

This icon indicates the points which are important from your technical interview. Before interview, you may visit these small tips as quick revision pointers.

This icon indicates the features which are new and available in new version JDK 7.

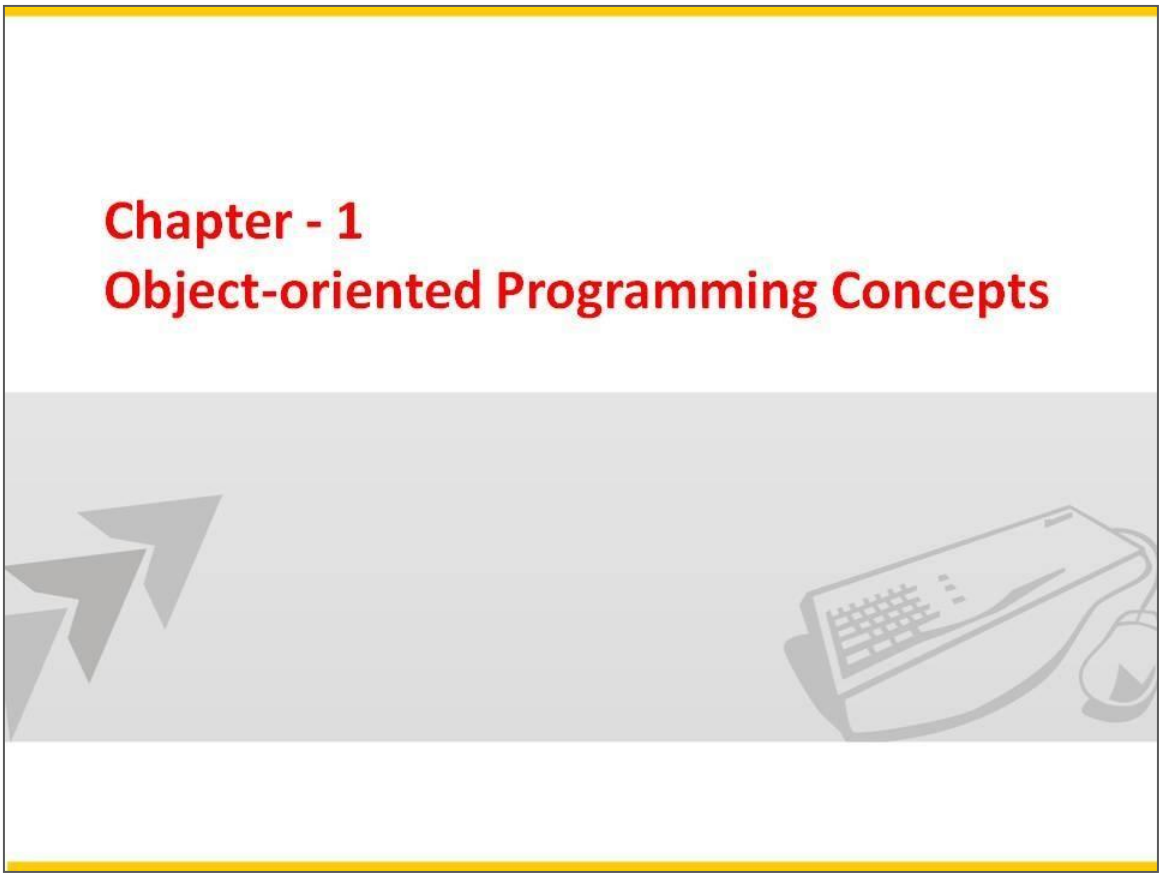
This icon suggests that it is good for the student to go through this additional reading material to bring more clarity to the concepts.

This icon indicates that this is a group discussion or group exercise. This is added for collaborative learning and problem solving. In the job environment one needs to take part in such discussions to arrive at the solution.

This icon indicates that more details are provided with respect to application of the technology/tool/concept which you are learning. This is application of technology learned to solve the real world problem.

This icon indicates a important note or tip from the application design perspective.

This icon indicates quiz to be solved in the classroom. This is for reinforcement of what you have learnt by challengin you through the questions.

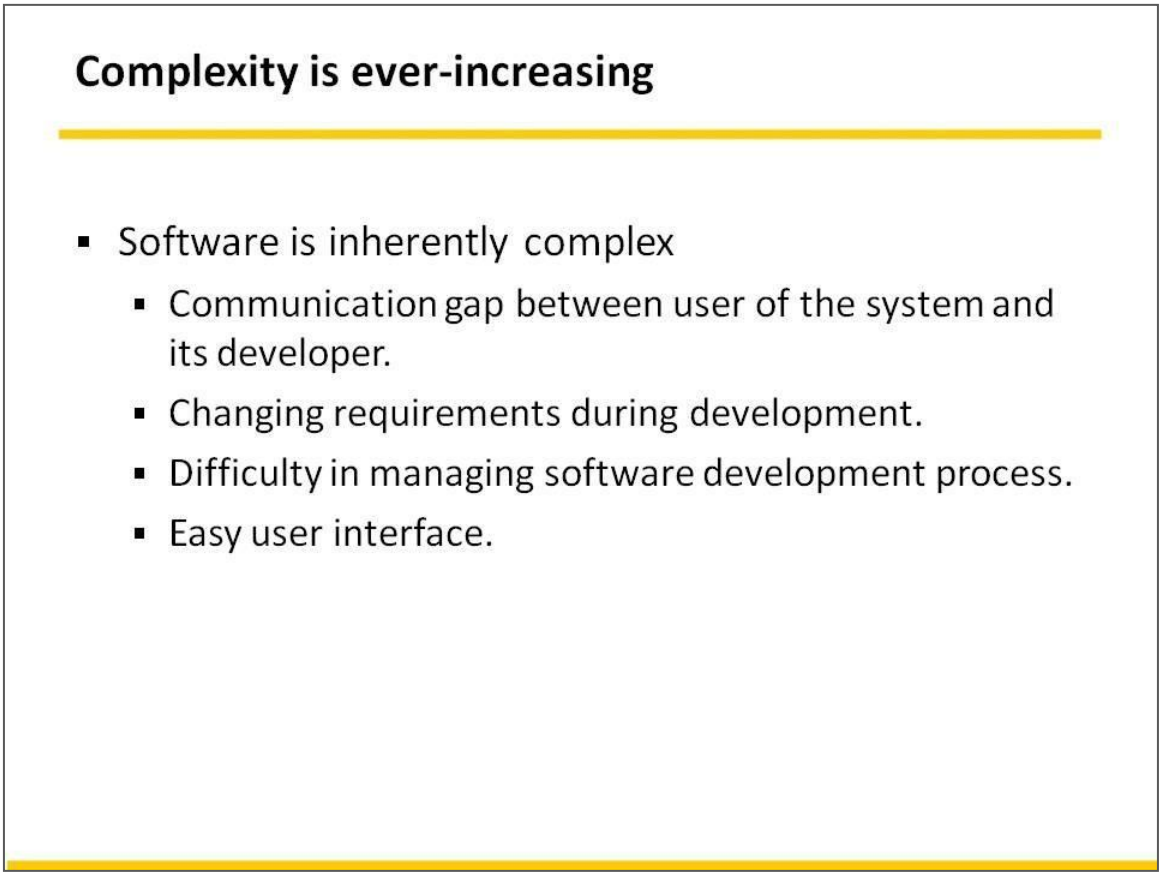


This chapter describes various complexities involved in developing software. Two different approaches are discussed, namely, the traditional approach and the object-oriented programming approach. Key concepts of object-oriented approach are also discussed.

Objectives

At the end of this chapter, you will be able to:

- State the limitations of procedural approach to develop applications.
- Define object-oriented approach and state its advantages.
- Define an object and state its characteristics.
- State the key concepts of object-oriented approach - abstraction, encapsulation, inheritance, and polymorphism.
- Define abstraction.
- Define encapsulation.
- Define inheritance.
- Define polymorphism.
- Define relationship between two or more objects using containment.



Software is a collection of programs that, when executed, does the task in a prescribed manner. During software development, the complexity keeps on increasing due to various factors:

Communication gap between user of the system and its developer

The understanding of the system to be developed by the developer and communicating his viewpoint to the user can be a difficult process. It is mainly because both user and developer belong to different working domains and the user may find it difficult to describe his/her requirements and vice versa.

Changing requirements during development

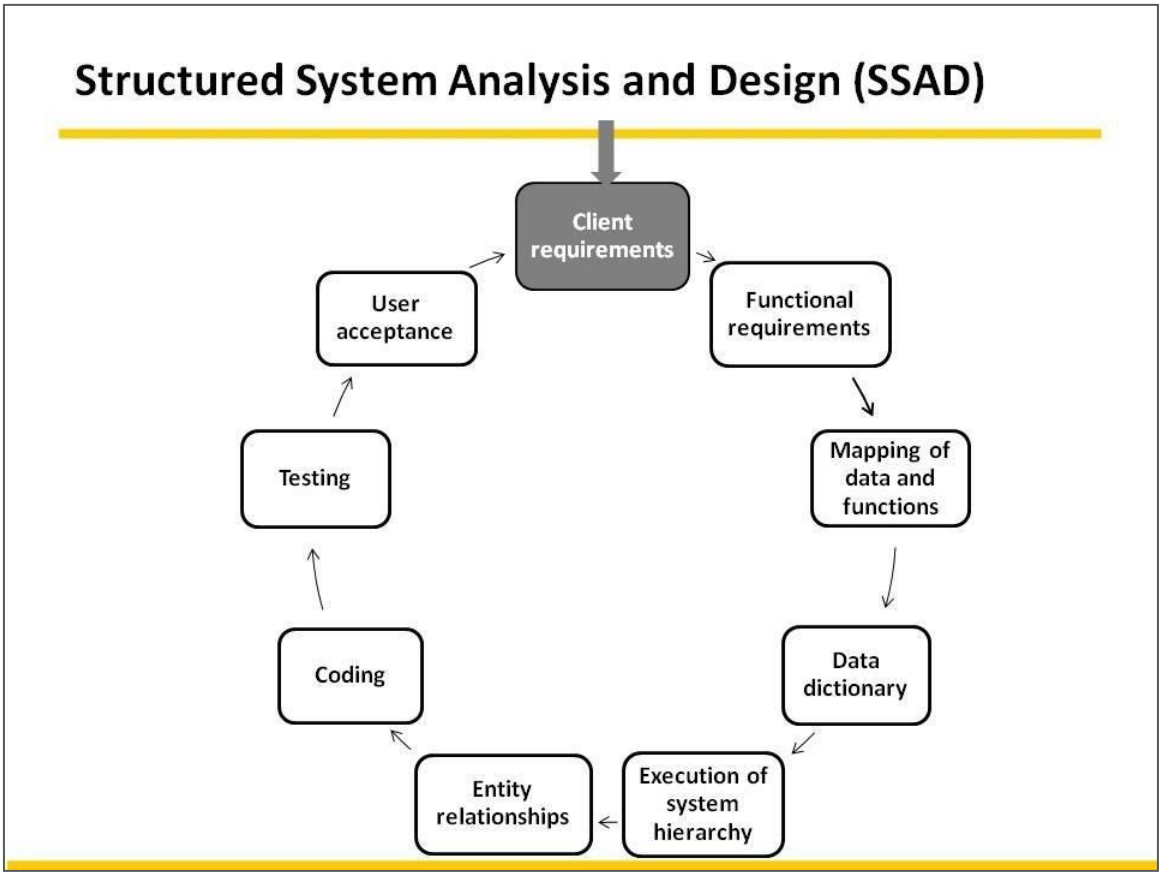
User requirements are very dynamic in nature i.e. they tend to vary during the course of development.

Difficulty in managing the development process

It may be possible that the whole system is divided into hundreds of modules and the modules are divided among the team members. If the number of team members is more then coordination among the team members becomes very difficult. In such cases, the key management challenge is to maintain unity and integrity of the design.

Easy user interface

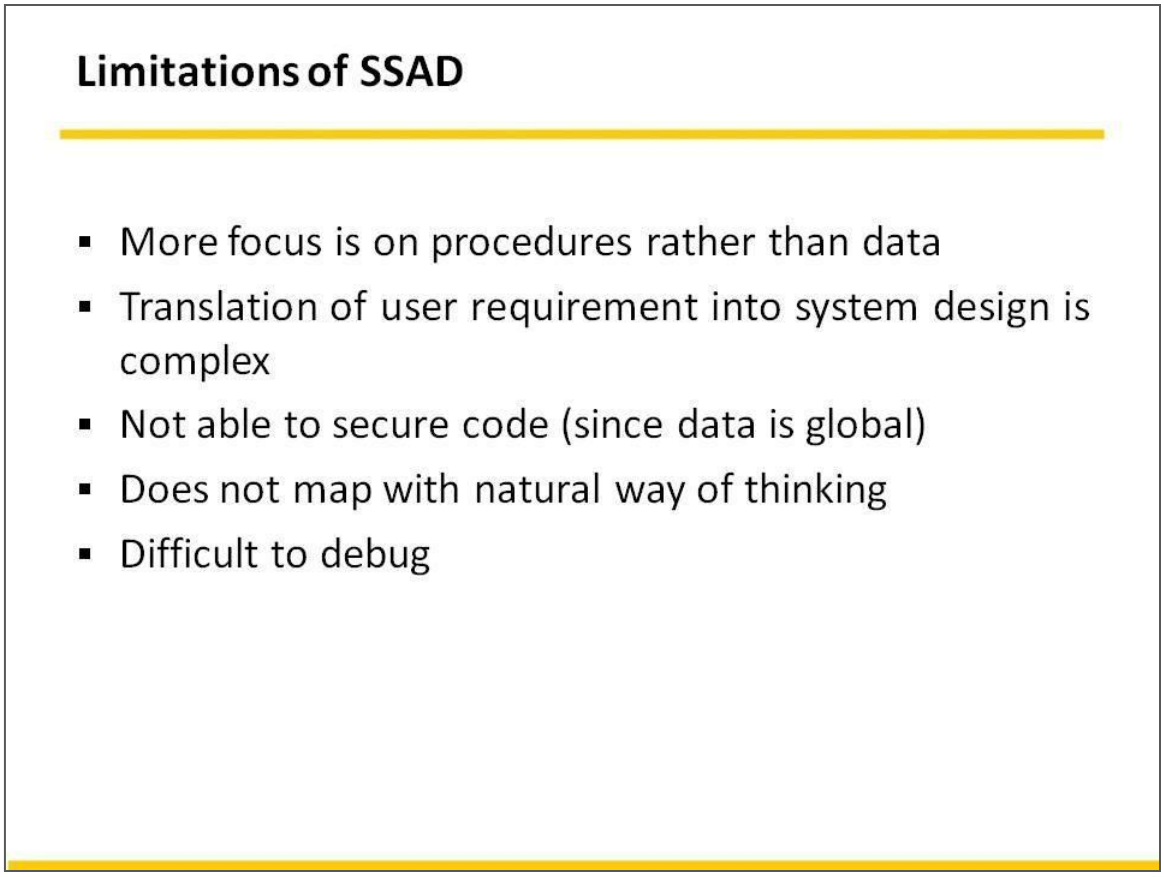
The main aim of any software engineer while developing any software system is to provide as much user friendliness as possible. While designing software, this has to be the ultimate aim in mind. If the user does not find the interface easy, it might have to be redesigned or additional training will have to be provided to the end users, which will familiarize them to the interface designed.



To simplify software development process, the Structured System Analysis and Design (SSAD) methodology was devised. Its process is as follows:

- The client's requirements are collected using various formats such as a questionnaire, interviewing all the users and so on. These are documented. An abstract view of the system is then prepared using Context level Data Flow Diagrams.
- The requirements are then studied and divided into various functional requirements. The requirements are represented as processes.
- The data associations between functionalities are mapped and data dictionary is prepared.
- Each process is then further broken down to the level of smallest individual processes.
- Execution hierarchy of the system is prepared. This is known as the structure chart.

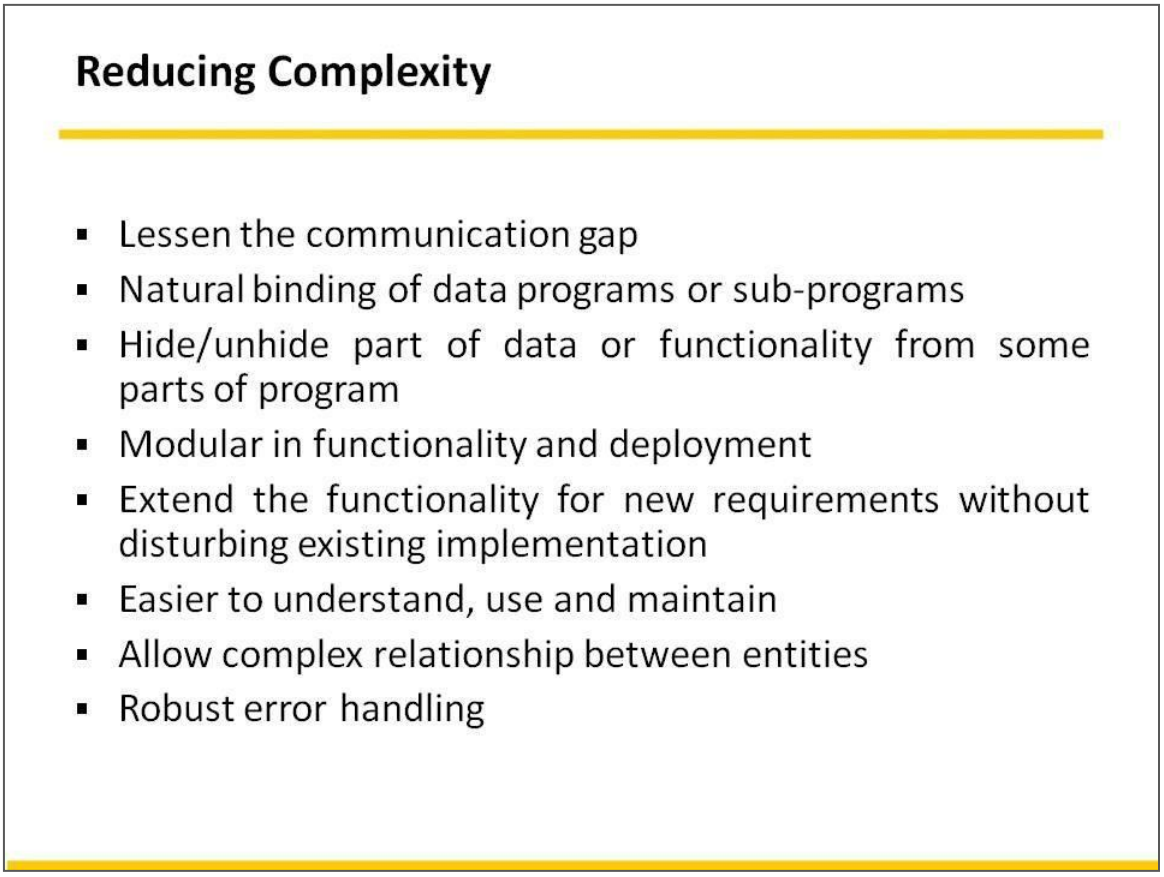
- This helps to establish relationships between various entities.
- Coding then begins using the technology decided for development.
- When the code for an individual process is completed, it is tested. All such small pieces of code that are tested are then integrated.
- Now the integrated system is tested for performance.
- Finally, the user's acceptance is tested.



The SSAD methodology has been in use for long time. It has some limitations. They are:

- The conversion of user requirements into system design is quite a complex and time consuming phase.
- This methodology does not map easily to the natural way of thinking. It is difficult to map real world entities into the system design. Suppose a cycle is to be programmed. It is difficult to bind the data to the functionality of the cycle because its functionality is considered separately.
- To perform any particular task, the focus is on procedure rather than the data. More importance is given to the method of performing the task. Data is considered to be secondary.
- Many procedures share data; data is global. This poses a risk of data security. More than one procedure may work on the same data at the same time, some procedure may inadvertently modify data that it is not supposed to or some procedure may work with the old values of data.

- Debugging a structured program is a difficult and time consuming process. Since data is shared over the entire program, different functions may be accessing it. To exactly find which function is causing the error is not easy.

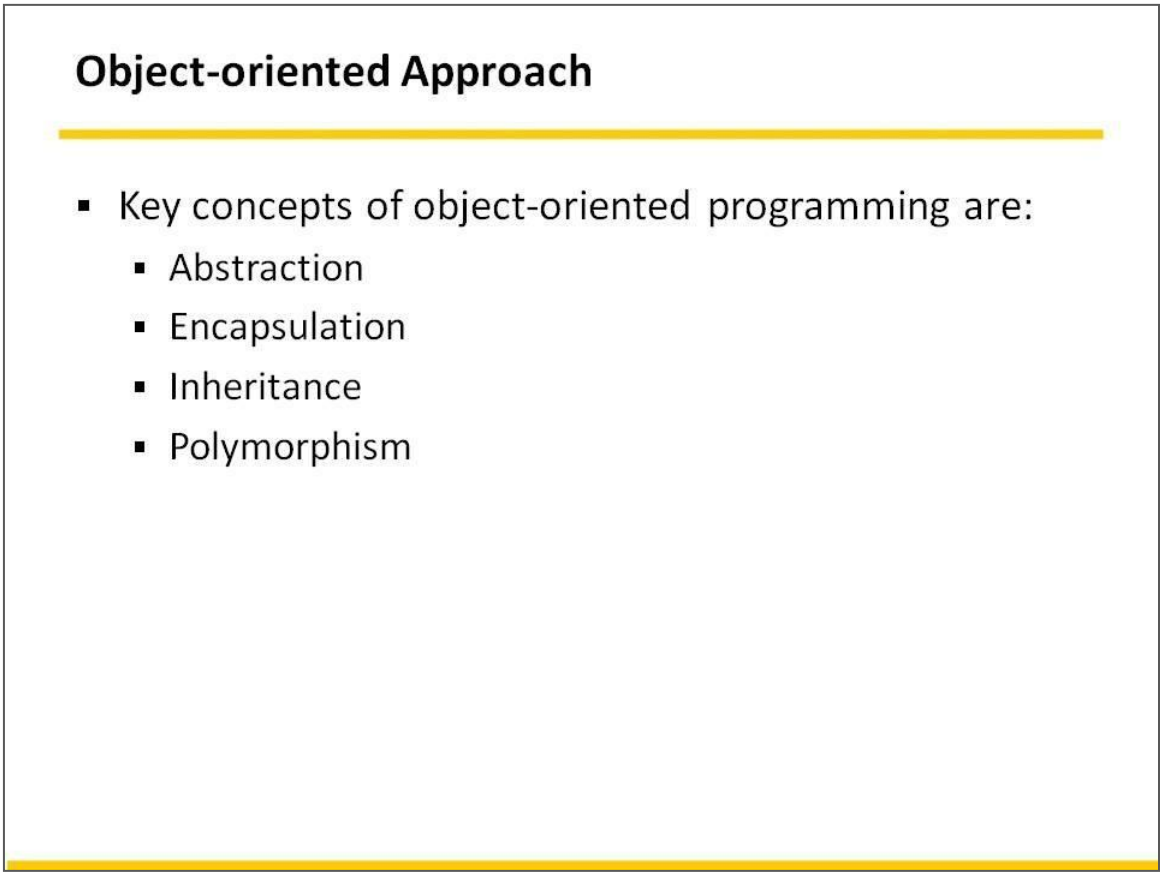


Developing software is a complex process. To reduce this complexity, a methodology such as SSAD can be used; but it has some limitations which have been discussed earlier. On studying this process, a methodology is desired that has the following features:

1. There is less communication gap between user and developer.
2. Natural mapping and association of data and programs or sub-programs which operate on that data.
3. Hide or unhide part of the data or functionality from some components.
4. Modular functionality and deployment.
5. Extend the functionality for new requirements without disturbing the existing implementation.
6. Easier to understand, use and maintain.
7. Allow to create complex relationships between entities.

8. Robust error handling.

A methodology that provides these features is the object-oriented approach. It is simpler to create applications that easily map to real-world entities or objects using this methodology.



The key aspects of objects-oriented programming that are used to develop applications can be seen above.

Feature	Concept
Natural mapping and association of data and programs or sub-programs which operate on that data.	Abstraction
Hide or unhide part of the data or functionality from some components	Encapsulation
Modular functionality and deployment.	Object
Extend the functionality for new requirements without disturbing the existing implementation.	Inheritance
Allow to create complex relationships between entities.	Containment or aggregation

Object-oriented languages

All programming languages that support abstraction, encapsulation, inheritance and polymorphism are known as object-oriented languages. Smalltalk, Eiffel, Java, C#, VB.NET, etc., are object-oriented.

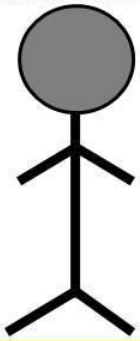
Object based languages

All programming languages that support abstraction and encapsulation but do not support inheritance; therefore cannot support polymorphism are known as object based languages. VB and Ada are object-based languages.

What is the difference between object - based and object-oriented languages? Give examples.

Abstraction

- Abstraction is the process of identifying the key aspects of an entity and ignoring the rest.
- Only those aspects are selected that are important to the current problem scenario.
- Example : Abstraction of a person object
  - Enumerate attributes of a "person object" that needs to be created for developing a database
    - useful for social survey
    - useful for health care industry
    - useful for payroll system

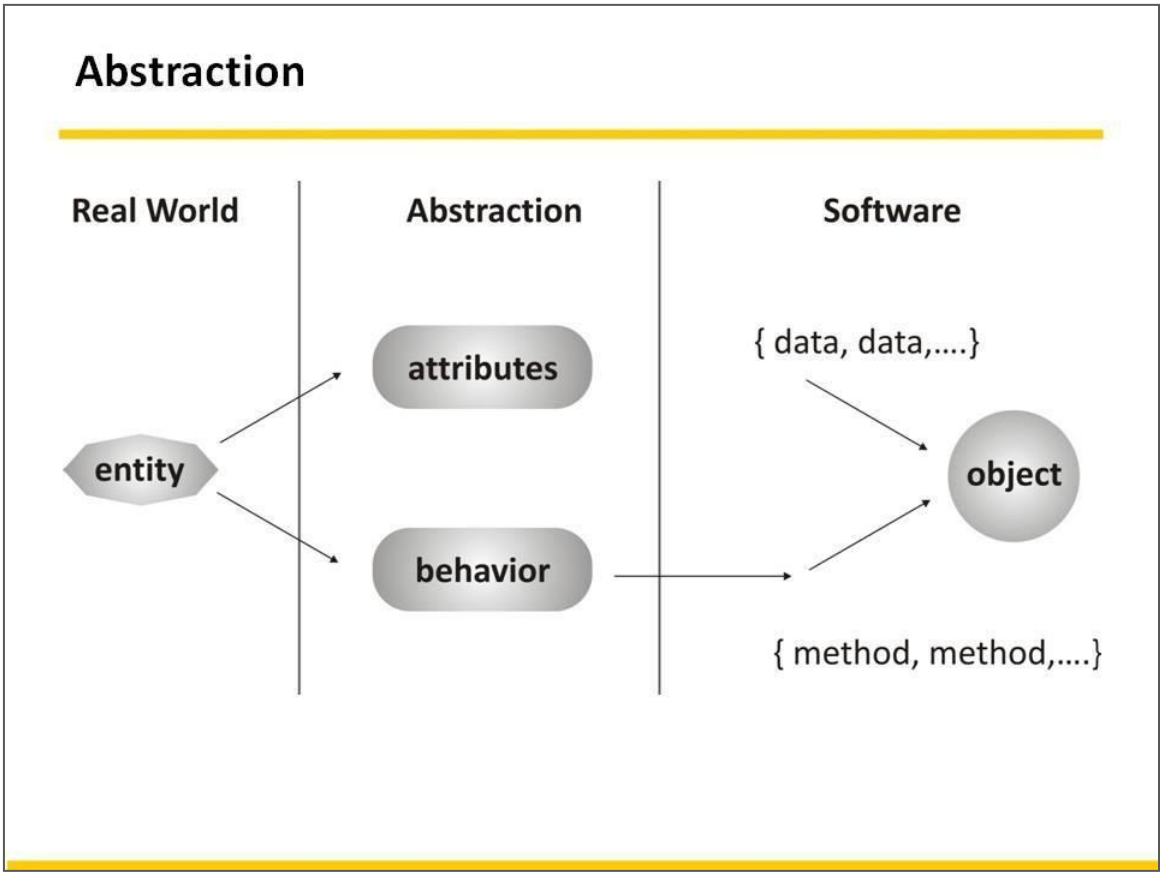


Every object has certain attributes and functions associated with it. Abstraction is extracting those attributes and functions of the object, which are essential in the problem domain and ignoring the rest. Consider a person as an object; the abstraction of this object would be different in different problem domains.

For example as shown in the table below, a person would be abstracted in different ways in three different problem domains.

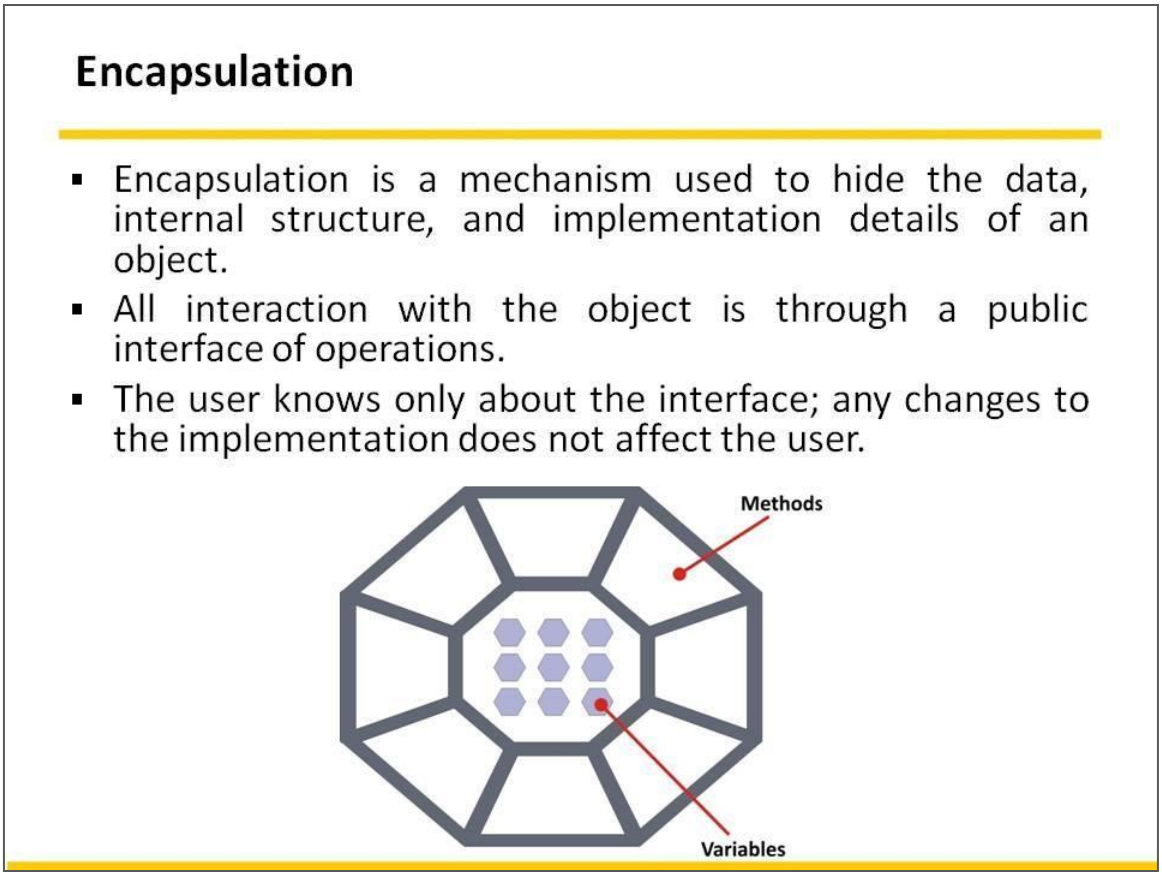
Social Survey	Health Care	Employment
name	name	name
age	age	age
marital status	---	---
religion	---	---
income group	---	---

address	address	address
occupation	occupation	occupation
---	blood group	---
---	weight	---
---	previous record	---
---	---	qualification
---	---	department
---	---	job responsibility



The above diagram shows the meaning of abstraction in real world and how it is implemented in software.

- Abstraction is extracting the essential / relevant details of an entity from the real world.
- Abstraction refers to attributes (data) as well as behavioural (functional) abstraction.
- In the software domain, an entity comprises of the abstracted data as well as the functions.



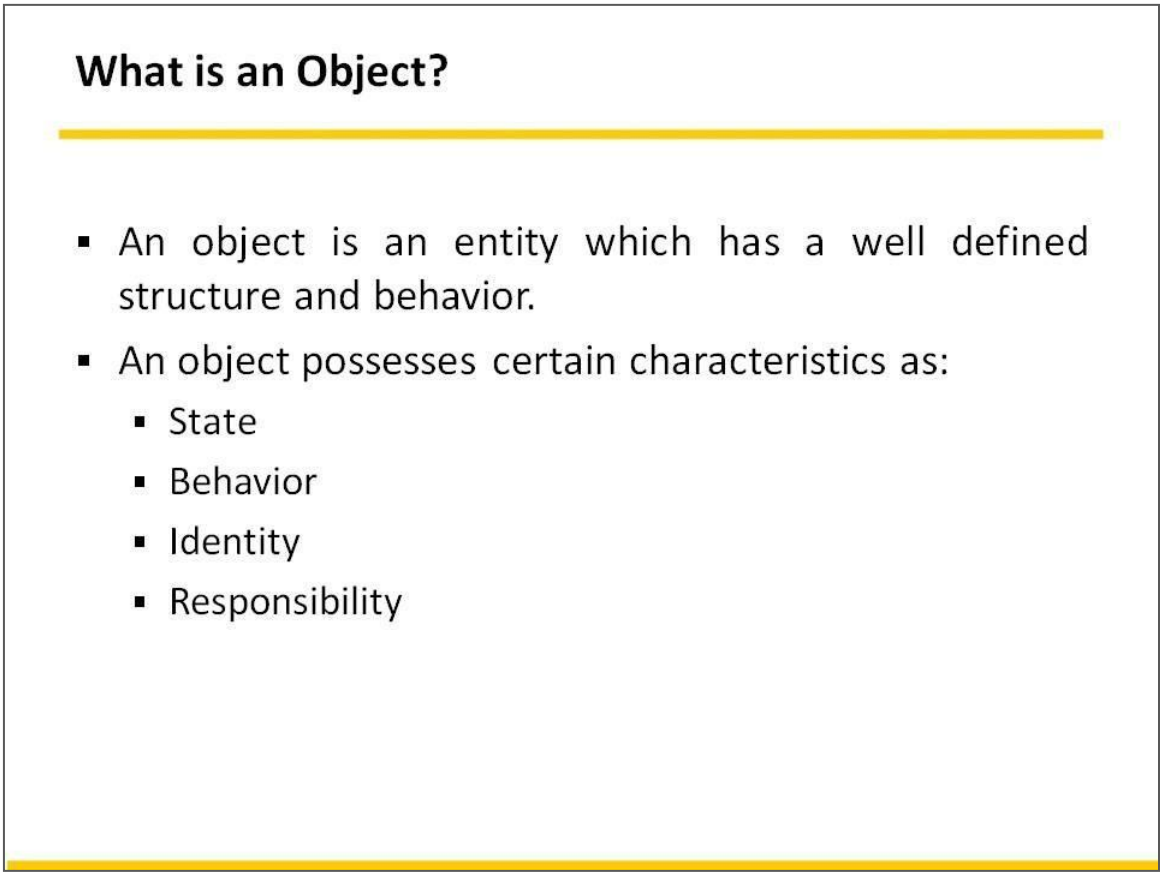
Encapsulation means data hiding; the hidden data can be accessed only through a public interface provided to the user. This interface consists of all the functions or the methods of that object.

When an object hides what it does from other objects and from the outside world is known as encapsulation. It is also information hiding. But an object has to provide an 'interface' to the outside world to initiate the operations on the data. If this is not done then the object cannot be used.

For example: The internal components of a television and their working are hidden from the end user. TV has a set of buttons either on the television itself or on a remote. The TV buttons or remote is an interface to operate the TV. So the behaviour of the object is given to the end user along with the object.

Fan can be operated with the 'On' or 'Off' interface. Its speed can be regulated using a speed regulator. These are the public interfaces provided to the end user. There can be other hidden functionalities that help to regulate the speed.

Data hiding ensures security of data. Encapsulation separates interface of an abstraction and its implementation. One advantage of encapsulation is that even if the method / function implementation is changed, the user need not know about it as long as the interface (calling mechanism) remains the same.



Everything in this world is an object, from a nail to a hammer, from a lever to a rocket. From human perspective, an object is a tangible or visible thing or something that exists in time and space.

Object can be defined as:

“An object represents an individual, identifiable item, unit or entity, either real or abstract, with a well-defined role in the problem domain i.e. object has Identity, State, Behaviour, and Responsibility.”

For example, a car, a person, etc. are different objects.

An object can be tangible, intangible or conceptual. Some more examples of objects are:

Air, hard disk, pen, person, printer, sound, instrument, point, bank account, contract, signal, industrial process, medical investigation, transaction, training course.

An object possesses certain characteristics - state, behaviour, identity, responsibility.

1. Make a list of objects that you know.
2. Take one of the objects listed earlier and perform abstraction. For example, abstraction can be performed on Person object as a patient in Hospital Management System.
3. Perform encapsulation for any object abstracted earlier.

State of an Object

- The state of an object includes the current values of all its attributes.
  - An attribute can be static or dynamic.

Employee Attributes

empId, name, gender } static

age, address, phone, basicSal, education, experience } dynamic

Employee

The state of an object includes the current values of all its attributes. An attribute can be static or dynamic.

Considering the example of an employee, the attributes such as the empid, name and gender of an employee remain constant throughout the tenure of an employee. Such non-changing attributes are known as static attributes. The dynamic attributes, such as age, address, phone, basic salary, education and experience can change.

Other examples are:

- Example of a window as an object - The height, width, the background color and the co-ordinates of a window can be its attributes. The current values in those attributes will describe the state of that window (e.g. minimized, maximized, resized etc.).
- Example of a fan – Color, make, product number and speed of a fan are attributes of a fan. The current value of its attributes describes the state of the fan.

- Example of a mobile phone – Make, product number, model number, SIM card, category, charge of the battery etc. are the attributes of the mobile phone. The current value of charge of the battery is the state of the mobile phone at that point of time.

Behavior of the Object

- Behavior is how an object acts and reacts, in terms of its state changes and message passing.

Employee as an object

Employee

calculateSalary

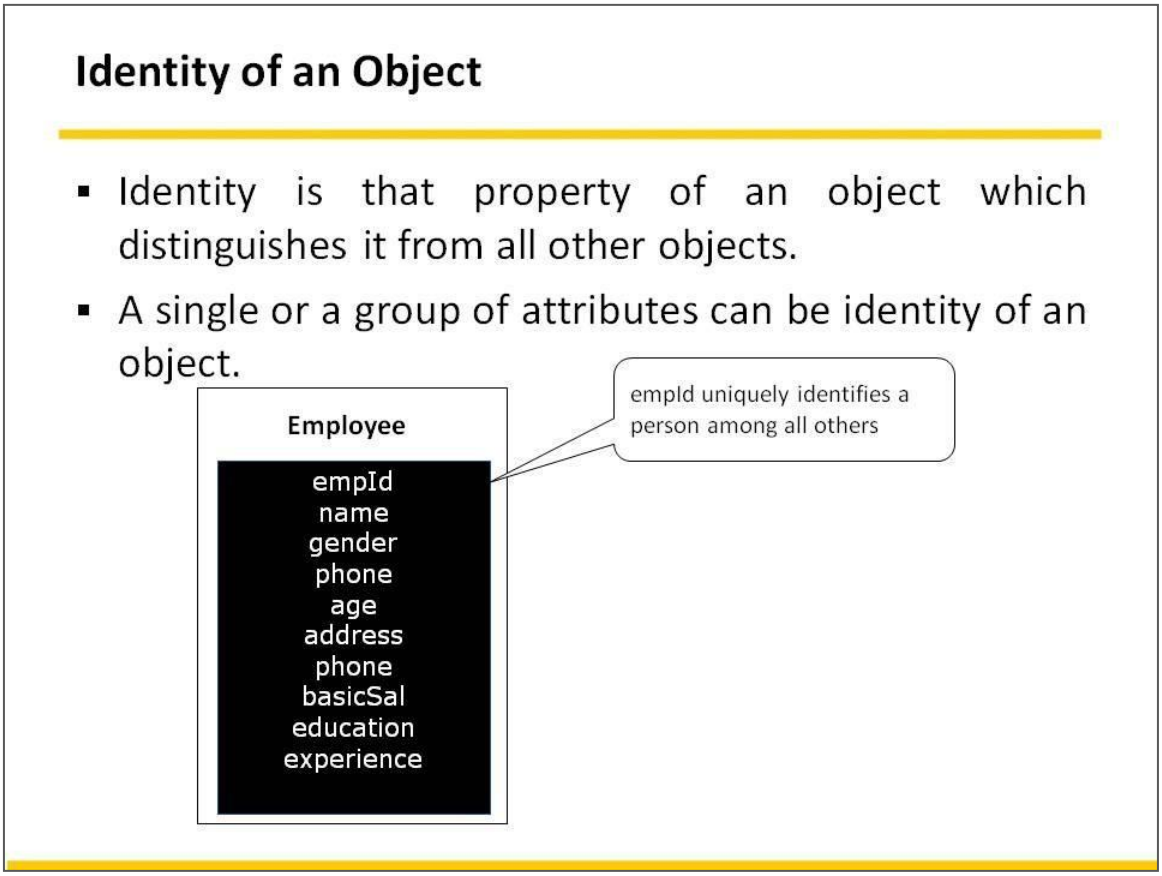
printDetails

Totality of operations – how an employee behaves or responds to, is the behavior of a person. Behavior can help in retrieval of information or changes in the attributes of an object

The behaviour of an object corresponds to its functions. The change in state of an object reflects in its behaviour.

In the example shown, 'Calculate Salary' and 'Print Details' are the behaviours of the employee. There can be other behaviours too. They can either help to retrieve the information or change the state of the attributes.

The values of the attributes of a window object change whenever a window is resized, minimized or maximized. The state of a fan changes whenever the fan is switched off, or switched on or its speed is regulated. 'Off', 'On' and speed regulation is the behaviour of the fan. Charging the battery of a mobile phone would change the state of the battery at a given point of time.

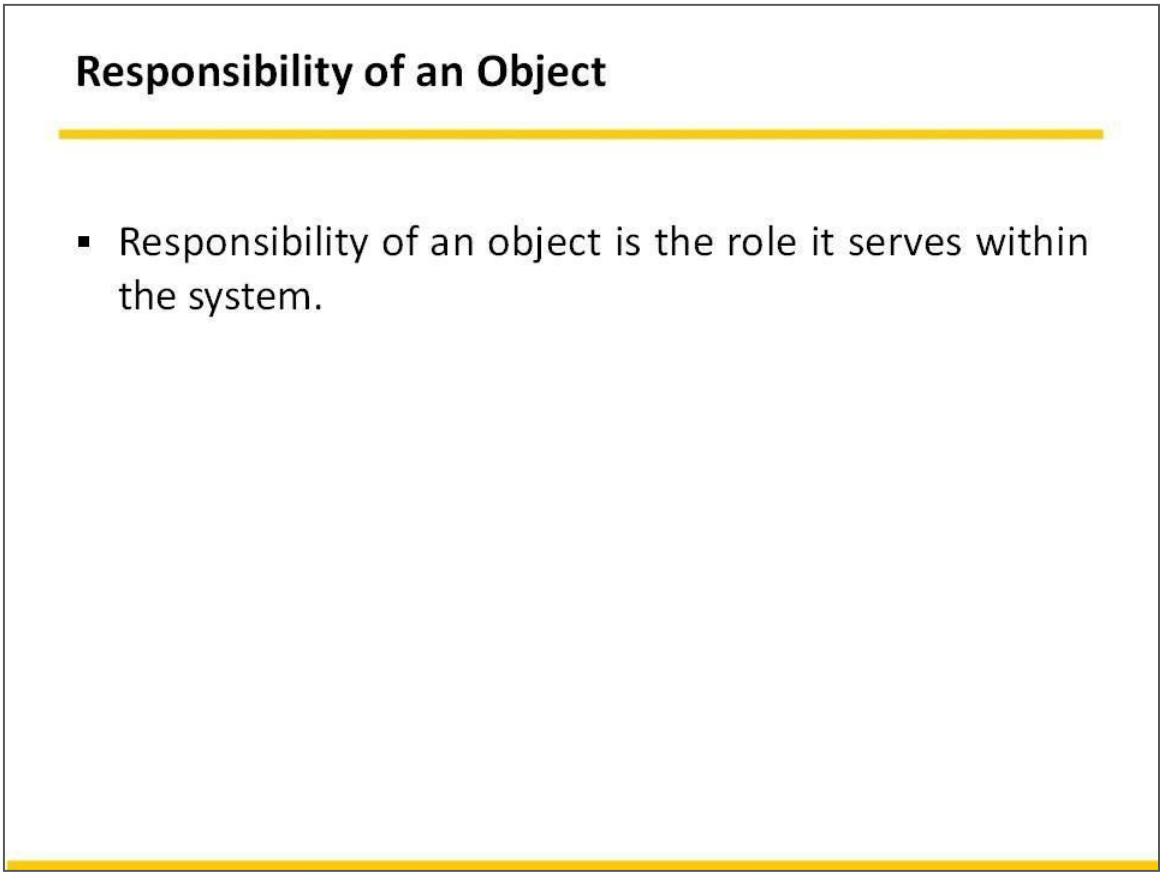


Every object existing in this world can be distinguished from the other. In other words, every object has its own identity.

Having an identity is absolutely essential for an object because it distinguishes a particular object from the other objects in the system. It is easier to interact with an object that has identity.

As mentioned above, `empid` is an identity of an employee. No two employees will have same employee number.

For example, while using the Windows Operating System; there can be more than one window on the screen. Each window is uniquely identified by the unique number assigned to it by the OS, called a 'handle'. This number ensures that user interaction with each window remains unique. Fan and mobile number would have a unique product number as an identity.



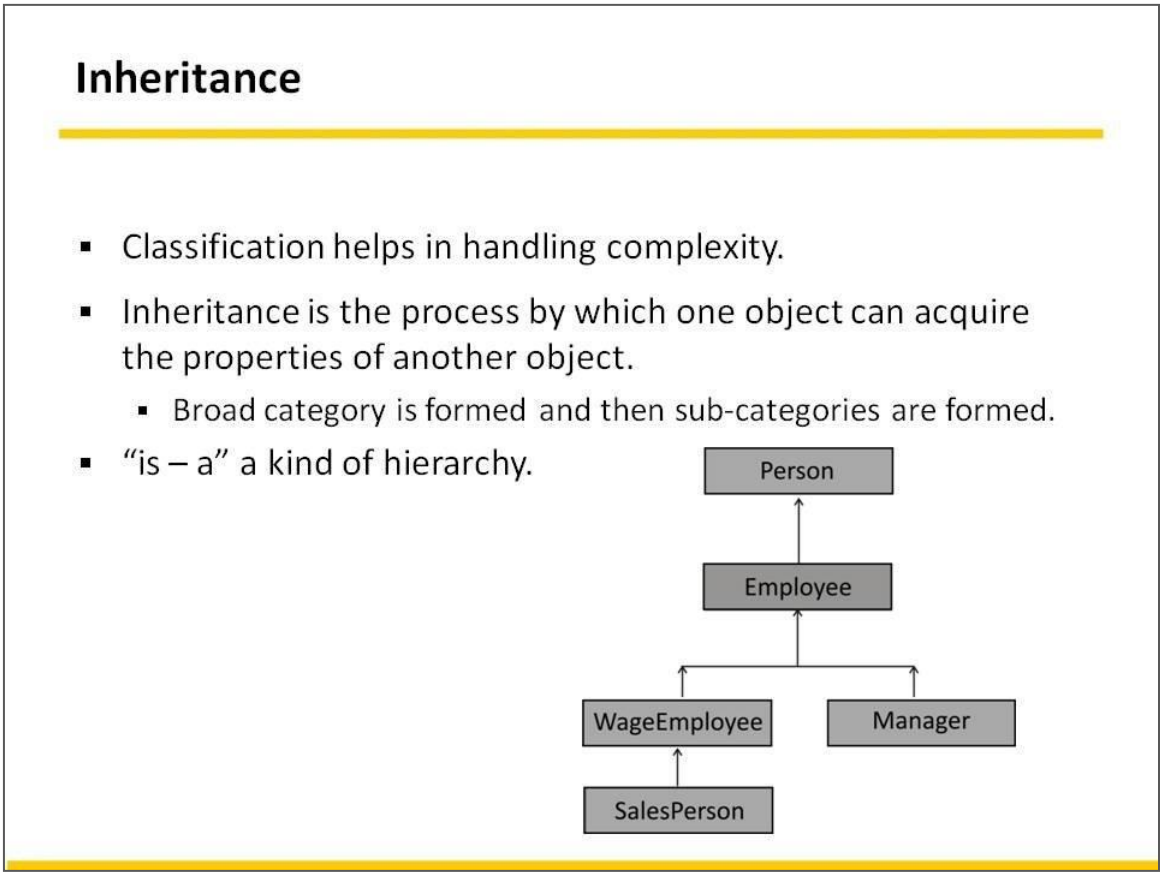
The responsibility of an object is the role it serves within the system. For example, the responsibility of an employee is to carry out the work given and get the salary. The responsibility of window object is to provide an easy and consistent user interface.

Some points to remember:

- Every object has a state, behaviour, responsibility and a unique identity.
- A system comprises of multiple objects existing together in a meaningful interaction.
- This meaningful interaction in the given system decides the responsibility of that object in that system.
- The responsibility of an object decides the behaviour of that object.
- The behaviour of an object is governed by the change in state of that object.
- For changing the state of a particular object, it is essential to identify it separately from all the other objects.

Attributes	State	Identity	Behaviour	Responsibility
empid	empid =10	empid =10	Swipe Card	To be present in the office on time
name	name= “Dhruv”		Fill Timesheet	Do allotted work
gender	gender =“male”		Wear ICard	Identity with organization
age	age=27		Calculate salary	Get salary
address	address = “Pune”		Print details	Provide personal information
phone	phone = 9823013451			
basic salary	basic Salary = 9000			
education				
experience				

State the characteristics of any three objects listed earlier.



Object-oriented terminology helps to map real world scenarios to solution domain. In real world, objects interact with each other based on their responsibilities. These responsibilities are associated with entities based on their classification. In other words, the behaviour of an entity is dependent on its place within the hierarchy in which it operates.

This hierarchy provides a basic framework for an entity's behaviour and characteristics. The specific entity is supposed to fulfil these interpretations in its own way. This is called inheritance.

An example of inheritance is children acquiring qualities like color of eyes or hair from parents.

In object-oriented approach, inheritance is a ranking or ordering of abstractions, where one abstraction inherits the properties of another abstraction. Any information (data), when categorized, leads to simplicity of management of that data. Classification is a technique used for handling this categorization.

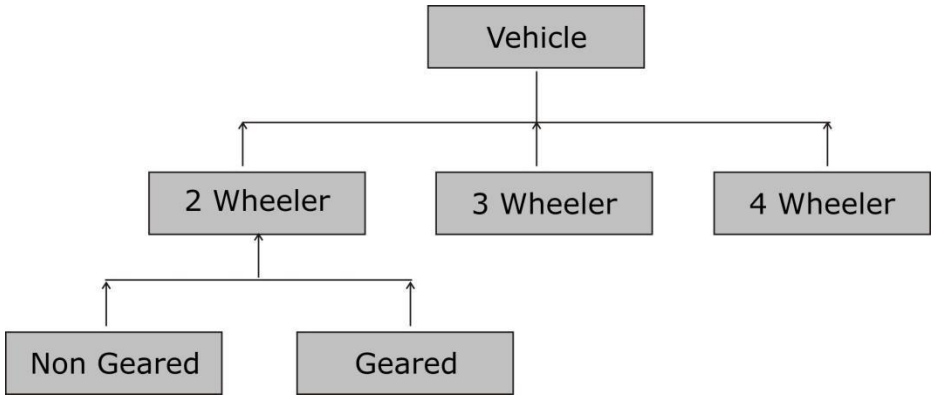
Hierarchical classification / Inheritance

While classifying the information, first a broad category / parent is formed and then its sub-categories / children are formed.

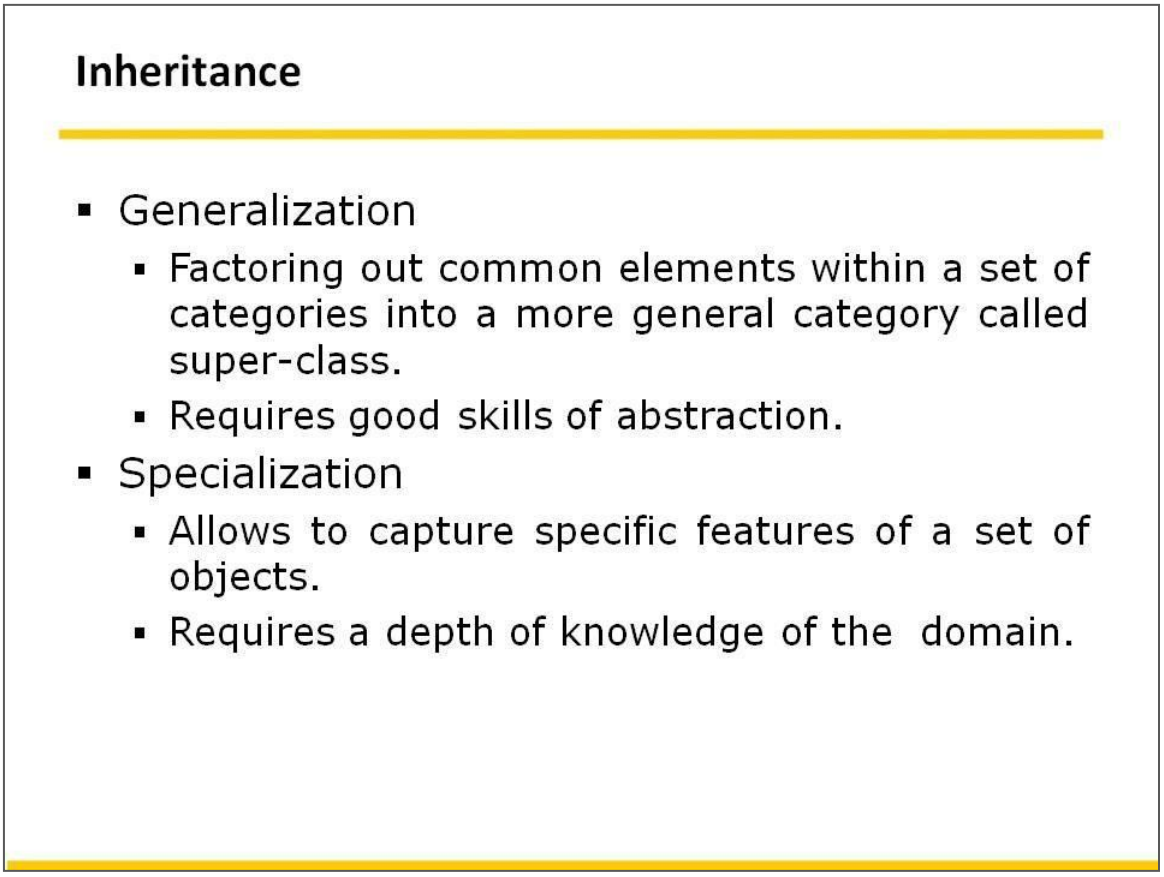
These sub-categories are called children, and they inherit all the attributes and the behaviors from their parent. Therefore, while forming the subcategories, only those attributes and behaviors are abstracted, which are specific to that subcategory, since, the common attributes and the methods belonging to its parent class are already inherited.

In the example shown, employee is a person. It is a broad category or a parent. `Wage Employee` and `Manager` are the sub categories. `Wage Employee` and `Manager` inherit all the features of `Employee` and have their own properties. `Employee` inherits its features from `Person`.

In the diagram shown below, `Vehicle` is the parent or broad category and `2-wheeler`, `3-wheeler` and the `4-wheeler` are the sub-categories / children. There can be a further categorization of the 2-wheeler into `'non-geared'` and `'geared'`.



This type of relation leads towards building 'is-a' kind of hierarchy and is also called inheritance in object-oriented terminology.



Generalization and specialization are two points of view that are based on class hierarchies. They express the direction in which the hierarchy is extended.

Generalization

Moving up in the hierarchy is said to be generalization. In other words, it is a very general / broad level of abstraction. The topmost class in the hierarchy is the most generalized class.

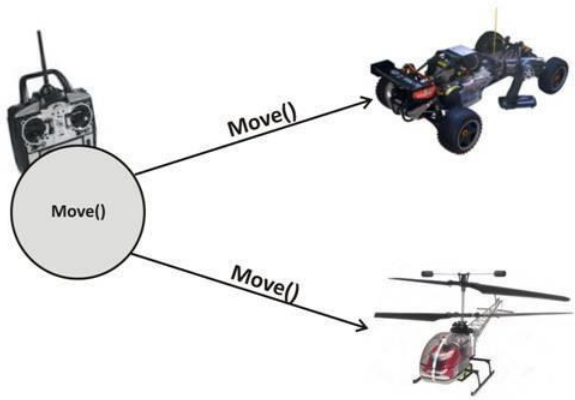
Specialization

While moving down the hierarchy more and more specific features are added in each sub-category that is formed. This is said to be specialization.

Draw a hierarchy for an object in given a problem domain. For example, in Hospital Management System, persons will be of different categories. A person can be a doctor, patient, nurse etc. Doctor can be further categorized as senior doctor, junior doctor and trainee doctor.

Polymorphism

- The ability of different types of related objects to respond to the same message in their own ways is called polymorphism.
- Polymorphism helps us to :
  - Design extensible software; as new objects can be added to the design without rewriting existing procedures.



The word 'polymorphism' is combined from two words; 'poly' means many and 'morph' is a form. Thus, polymorphism is the ability to take more than one form. This means that one command may invoke different implementation for different related objects.

Polymorphism plays an important role in allowing different objects to share the same external interface although the implementations may be different.

Here, the same command '`Move`' is used to move remote controlled objects like a car and a plane. Both the toys move but the way in which all these objects respond to the command is different.

If another object is added to the above list e.g. a window, it will also respond to the same command `Move`, but in its own way.

Apply polymorphism concept for the hierarchy created earlier. For example, each of the categories of doctor will provide different treatment.

Containment

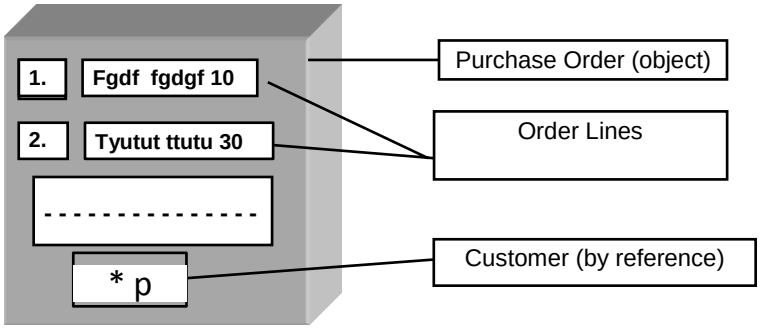
- One object may contain another as a part of its attribute
  - Document contains sentences which contain words.
  - Computer System has a hard disk, processor, RAM, mouse, monitor, etc.
- The containment need not be physical
  - E.g. Computer System has a warranty

Every object in an object-oriented system has to collaborate with other objects in some way or the other. Every object has some attributes. Some of these attributes can be another object. This concept of an object being a part of another object is called containment or aggregation. Containment need not be physical.

A typical computer system is an example of an composition. It consists of a hard disk, processor, RAM, a keyboard, a mouse, a monitor, a CD-ROM drive, one or more hard drives, a modem, a printer and possibly a web cam.

An object that is made up of a combination of a number of different types of objects is another example of aggregation.

Composition is usually implemented such that an object contains another object. In aggregation, the object may only contain a reference or pointer to the object (and not have lifetime responsibility for it).



Consider another example of Purchase Order object. The order lines are the contained objects in the Purchase Order. This containment is of a physical nature. But the reference to the customer object is a conceptual containment since the customer object exists independently of the Purchase Order object. So, if the inner object's lifetime is not dependent on the outer object, then it is aggregation.

The Difference . . .

- Containment is used:
  - When the features of an existing class are wanted inside your new class, but not its interface.
    - Computer system has a hard disk.
    - Car has an engine, chassis, steering wheel.
- Inheritance is used:
  - When you want to force the new type to be the same type as the base class.
  - Computer system is an electronic device.
  - Car is a vehicle.

Hard disk is a part of computer system or computer system has a hard disk. In the example shown above, the engine is a part of the car. The chassis is a part of the car and so is the steering wheel. This kind of relationship illustrates 'containment'.

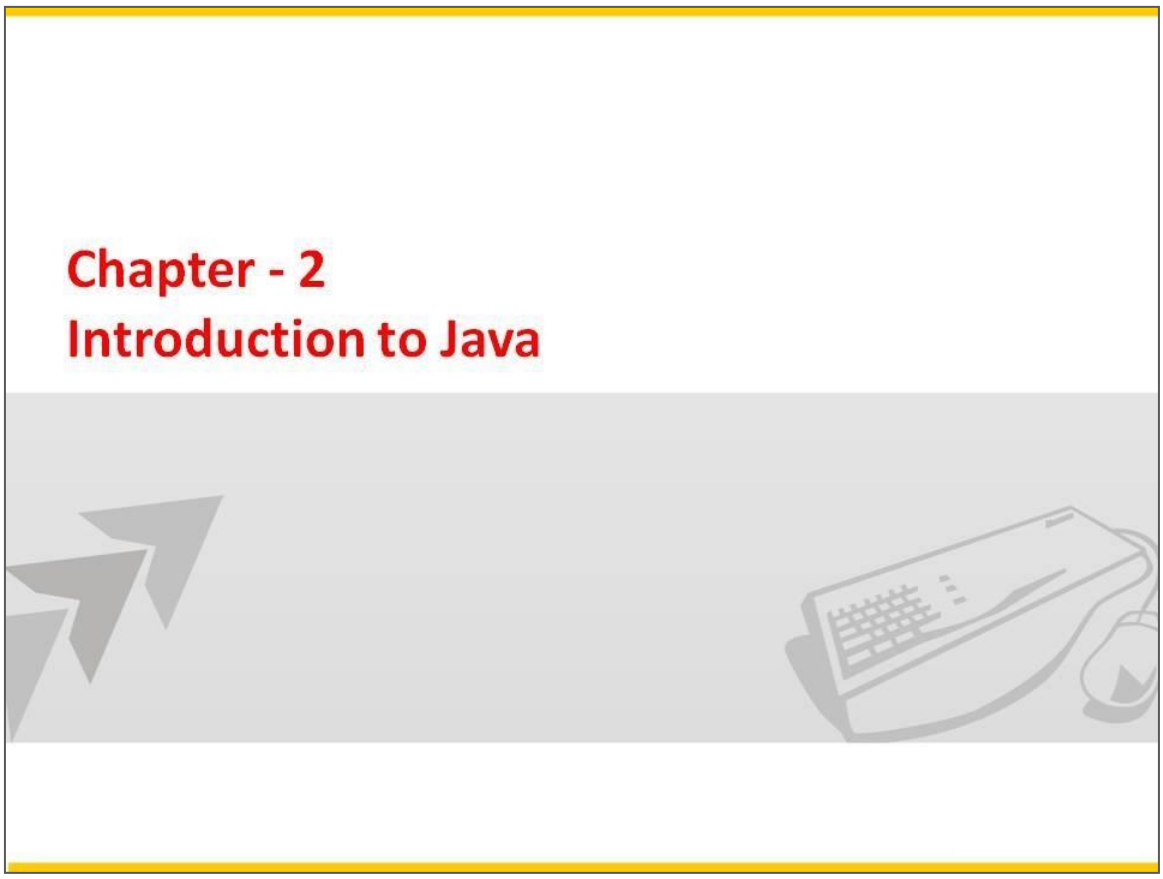
A computer system is an electronic device. A car is a vehicle. This type of relationship illustrates 'inheritance'.

'Containment' helps to reuse existing types as a part of the underlying implementation of the new type; 'inheritance' is used when the new type should have an identity extended from its parent type.

Make a list of any three objects that have contained objects.

Explain the difference between containment and aggregation.
Explain the difference between containment and inheritance.

Explain concept of each object-oriented concept with real life examples and their benefits.

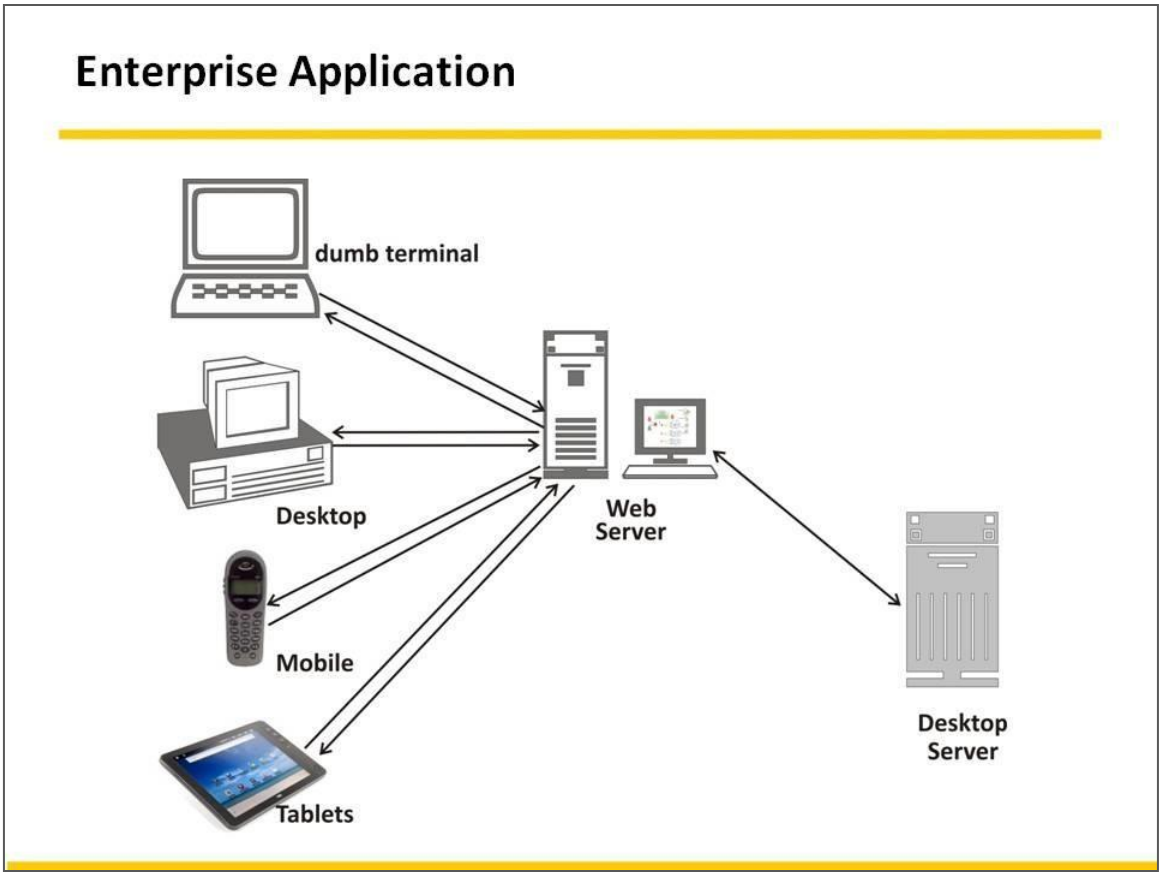


This chapter describes the origin and features of Java. Different types of Java applications and their components are also discussed. It also explains the sequence of steps in compilation and execution of a Java program.

Objectives

After completion of this chapter you will be able to

- List components of an enterprise application.
- Describe the origin of Java.
- Describe features of Java.
- Describe concept of platform independence and virtual machine.
- State different types of Java applications and their components.
- List various JDK libraries and their usage.
- Describe the sequence of steps in compilation and execution of a Java program.
- State usage of JDK, JVM and JRE.
- Construct the simple Java class: `MyDate`



Information at your fingertips is the motto of today's information world. A client requests for some application that is hosted on one web server; the server sends a response to the client. The application sometimes needs to fetch data from a database. The database may be hosted on some other server. The end result in either case is that a response is generated and sent to the client.

The client machine can be a desktop, a mobile, a palmtop, and so on. Majority of the applications built today are internet-aware distributed applications where multiple devices interact with each other. Online shopping, online reservation systems are some examples.

In such a scenario, the developer has a few questions in his mind.

1. Would the application be able to support a set of thousand more users?
2. Would the service be available for 24 x 7 x 365?
3. Would the response time suffer if user load increases?
4. Would the data integrity be maintained?
5. How easy would it be to fiddle around with the components of a running system?

6. Would the hardware resource requirements increase day by day with increasing user load?
7. Would the application be able to integrate with external applications smoothly?
8. Would anybody be able to hack, steal or corrupt confidential data?
9. Would the application run on any device?

And many more …

So, ultimately scalability, availability, performance, flexibility, reusability and security are the main features required for any distributed or enterprise application.

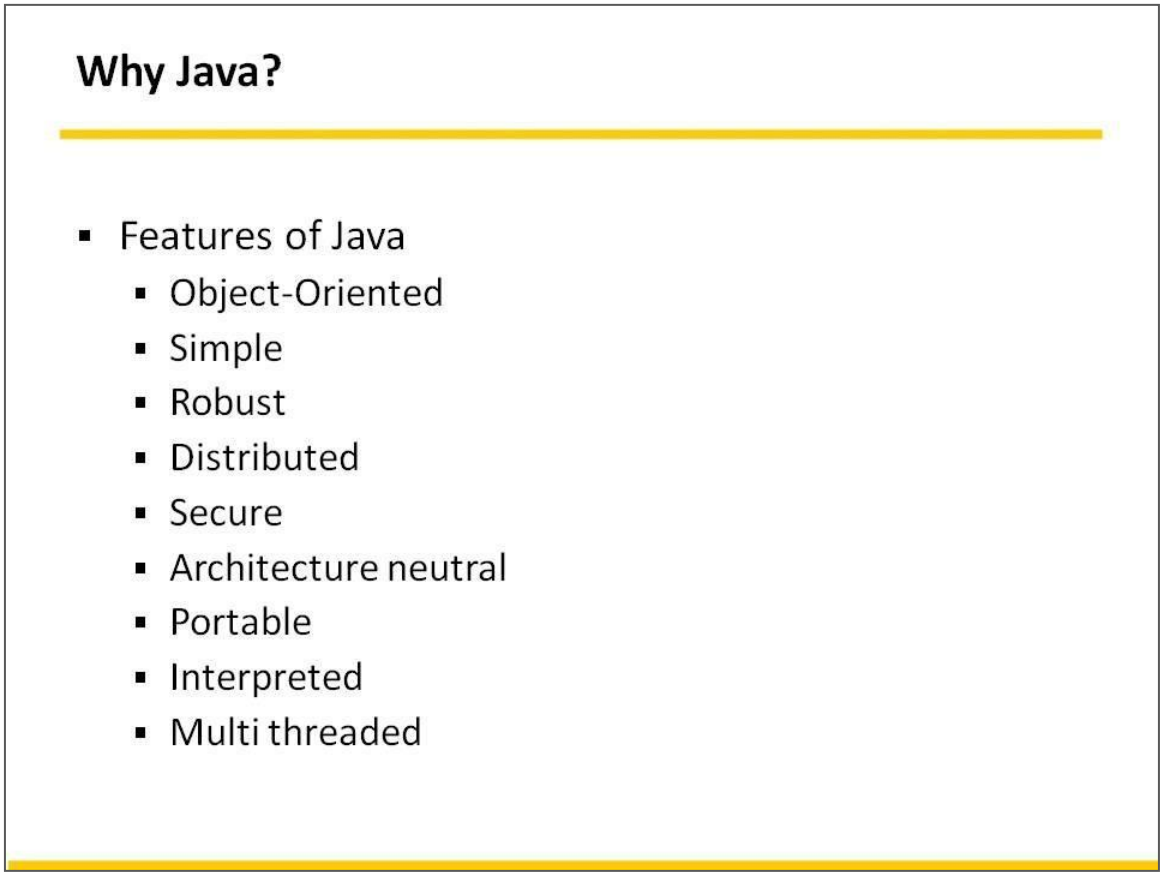
Concept of enterprise application should be known. Difference between desktop and enterprise application should be understood.

Brief History of Java

- Background: Electronic Consumer devices.
- Sun commissioned 'Project Green'.
- Developed language 'Oak-Later renamed to Java'.
- Was dismissed as just another OO programming language.
- Became popular with the rising popularity of 'www'.

Electronic consumer goods manufacturers required to use different processors for their products. Hence the same logic had to be coded multiple times in a language understandable by those processors. The development of Java was projected at providing a platform independent language that could be used to write the logic once and execute on any processor. The project was commissioned as 'Project Green' by Sun Microsystems. The initial name suggested for the language was 'Oak' after an oak tree that was there in the Sun's campus. But they soon realized that there was already a language by that name hence the name was changed to Java which is a popular coffee brand and which the language developers consumed.

Java initially failed to make its mark since it was discarded as another Object-oriented language similar to C++. In those days, C++ was a very popular language and there was a large community of C++ programmers who were reluctant to turn to Java. It was after the advent of internet or the World Wide Web that Java started becoming popular due to its platform independent feature.



Features of Java

Java is an Object-oriented, simple, robust, secure, portable, distributed, interpreted, dynamic, multi threaded, architecture neutral language.

Object-Oriented

Java is Object-oriented language.

Simple

Java language is simple in the sense that any programmer who is familiar with C or C++ can easily migrate to Java since it uses the same syntax that C++ uses. Apart from that Java removes some of the annoying features of C++ that have been troubling programmers for a long time. For example there are no header files to be included which increased the code unnecessarily and were required because the library consisted of functions as API. In Java the API consists of classes which are instantiated on demand. There are no structures, unions, pointer arithmetic, operator overloading or virtual base classes.

Robust

A language is said to be robust if it provides a mechanism for early detection of potential problems. Java provides a way in which the developer does not have to worry about chronic problems such as bad pointers, memory allocation errors and memory leakages among others. This is made possible because Java Virtual Machine (JVM) does not rely on the Operating System (OS) for memory management.

Distributed

Java is a language for developing distributed applications. Distributed applications are those applications which have modules that run in different runtimes and try to interact with each other to achieve functionality. The runtimes could be available locally or remotely i.e. on the same machine or on different machines or in different geographical areas. For such modules to communicate they have to use the underlying network services. Java provides an API which allows seamless communication between such disparate modules without exposing the network complexities to the developer. For example, Socket Programming which will be covered at a later stage in the course.

Secure

The JVM needs a bytecode file as its executable. Every executable has header that gives details about the file and also helps identify the file as a valid executable file for that machine. If the file gets corrupted then a bytecode verifier checks it out with the information in the header such as parity check etc. and discards the file if it is corrupted. This is a means of security that the Java language provides. On the other hand if the application is distributed in nature then the modules share data in the form of objects. These objects can be accessed for information only if Security Manager gives a green signal that the accessing module has reading rights on the object. The object also might want to access some data on the machine which is allowed only if Security Manager allows it.

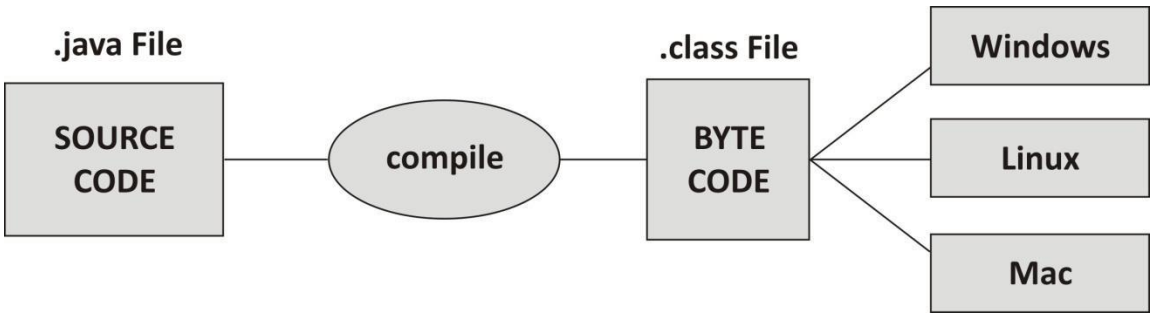
Architectural Neutral

Java claims itself to be Platform Independent. Independence means one does not have to rely on other for a particular service. In Java the JVM does most of the things for which other languages have to rely on the OS such as memory

management, process management etc. Platform means the combination of the OS and the underlying processor.

Portable

The size of the basic data types changes according to the platform. For example the size of integer on 16 bit architecture is 2 bytes whereas on 32 bit architecture it is 4 bytes. This causes the code compiled on 32 bit architecture to fail at times on 16 bit. To resolve such issues Java provides the behavior of its basic data types such as its size and operations possible which apply to the JVM than to the actual platform. This makes the same code to run on all platforms irrespective of what the architecture is.

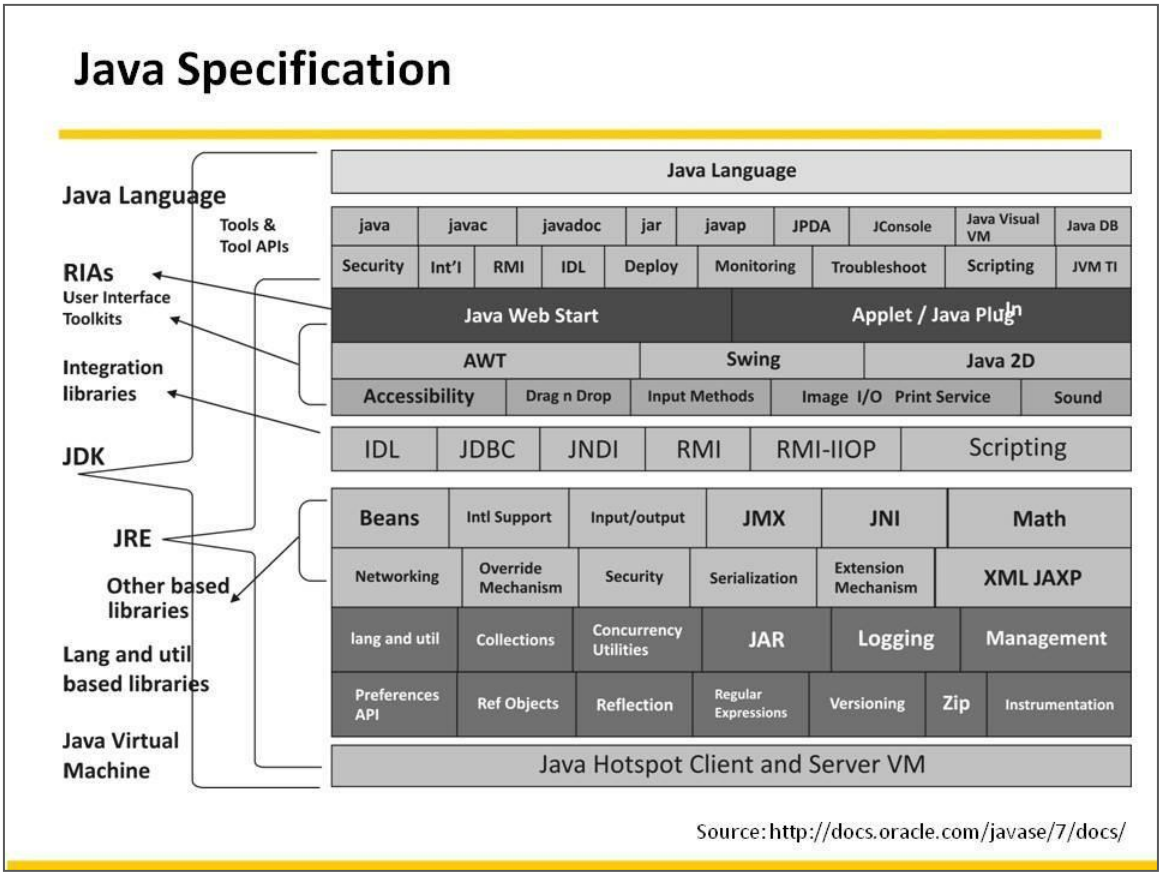


Interpreted

Java is compiled as well as interpreted. The source code is compiled to bytecode and then the bytecode is interpreted at every execution to the native code.

Multithreaded

Multithreading is a service provided by the OS. The OS provides some API for programs that wish to incorporate multithreading within their code. Programs written in languages like C and C++ have to use these OS provided API to achieve multithreading, since these languages do not provide abstraction over those API. The Java language provides an abstraction over these OS provided API by providing its own API for achieving multithreading. Hence, It is said Java is multithreaded. We will be discussing multithreading API during the course.



Sun provides a platform or an environment to develop, host, deploy, maintain and execute applications. This is something called Java Specification. Application can have be used of type desktop based, User Interface (GUI) based, web based,client-server, components, a web service that can be used by any other application in a language neutral and platform independent manner. Java Specification is a platform introduced by Sun to develop, host, deploy, maintain and execute applications.

Java specification has number of layers like:

Lang and Util Based libraries

- Math
- Monitoring and Management:
- Reference Objects
- Reflection
- Collection Framework
- Concurrency Utilities
- Java Archives (JAR) files

- Logging
- Preferences

Other Libraries

- Input Output
- Networking
- Security
- Internationalization
- Beans
- Java Management Extensions (JMX)
- XML (JAXP)
- Java Native Interfaces (JNI)

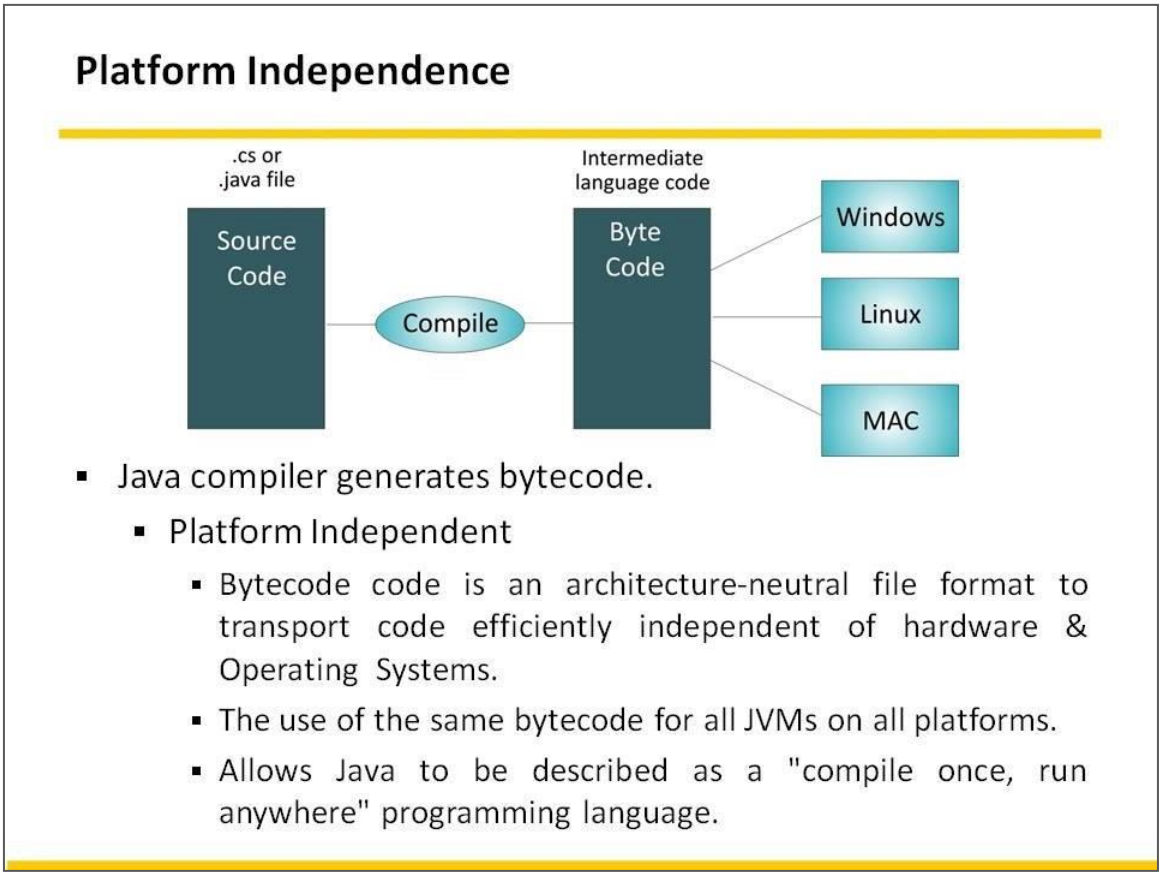
Integration Libraries

- Java Database Connective API (JDBC)
- Remote Method Invocation (RMI)
- Java IDL (CORBA)
- RMI-IIOP
- Java Naming and Directory Interface (JNDI)
- Input Method Framework
- Accessibility
- Print Service
- Sound
- Java 2D
- AWT
- Swing

Rich Internet Applications Development and Distribution

- Debugger Architecture
- VM Tool Interface
- Javadoc Tool
- Annotation Processing Tool (APT)
- JConsole API

To get complete information about Java Specification read Appendix.

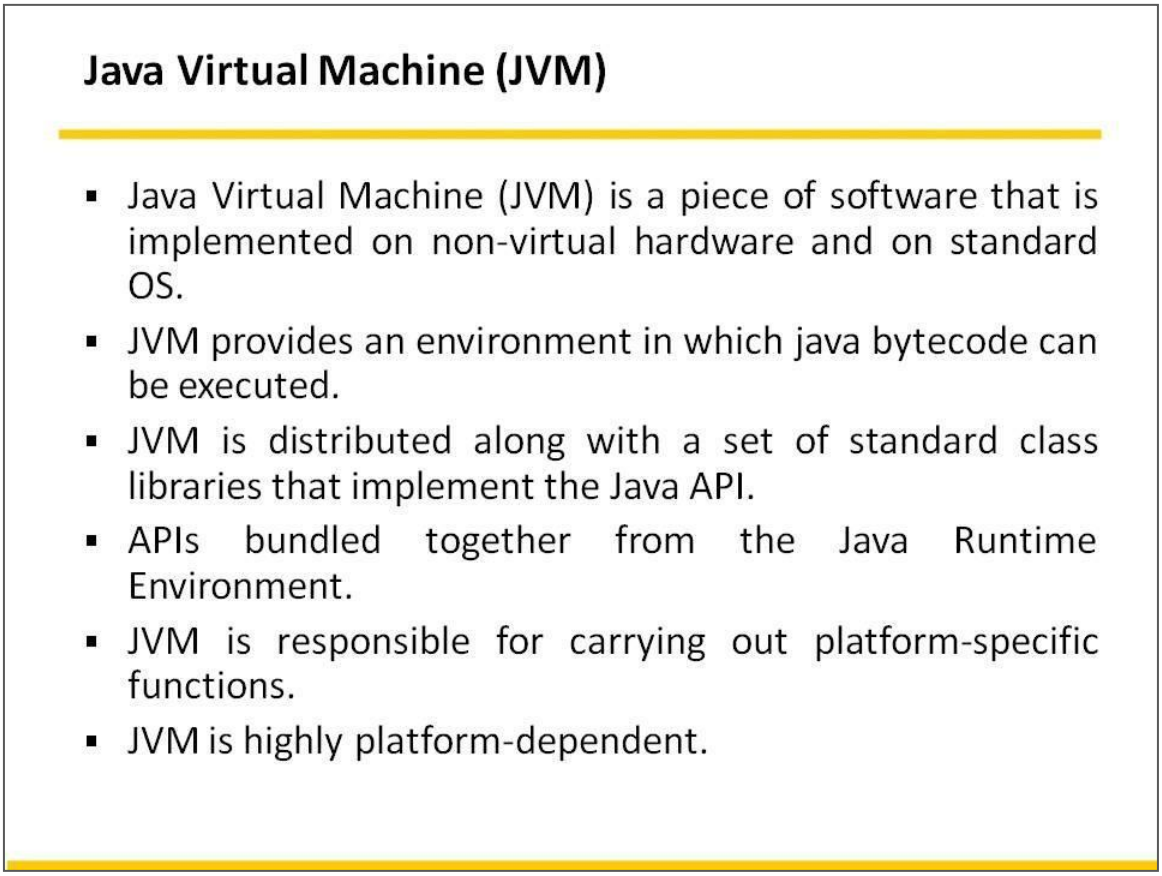


Over a period of time, software applications were created in many different languages based on availability and suitability of a particular language. Compilers were created to convert these high level language source codes in machine understandable 'machine codes' (sometimes called as native code). Every computer with its own specific machine language made it very difficult to port the application from one type of processor or platform to another. This activity involved creating platform specific programs or versions of applications which made their maintenance a nightmare.

To solve this problem, Java and .NET came up with the solution of isolating the platform level dependencies from the developers. The first component i.e. the language specific compiler translates the source code in some specific pattern independent of any platform. This translated code is platform independent code. It is also non executable code. It is then translated by second component into executable instruction set specific to the platform.

In Java, the executable is a bytecode file which is to be executed on the JVM. The JVM acts like a processor as well as the Operating System (collectively as Platform). Hence the executable is independent of the underlying platform for its execution. The same Java executable can be executed on any platform as long as there is an appropriate JVM available there.

Why platform independence is needed in real world?



Virtual machine is a combination of three things:

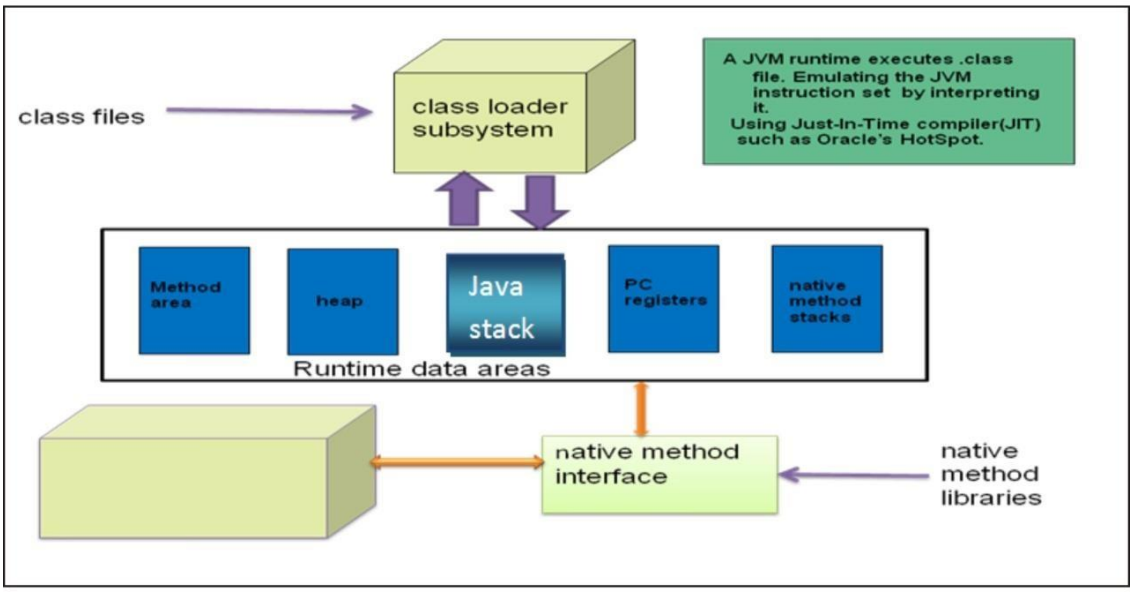
- The abstract specification
- A concrete implementation
- A runtime instance.

The Java virtual machine is called “virtual” because it is an abstract processor defined by specification. To run a Java program, concrete implementation of abstract specification is required.

When any Java application is executed at that time a runtime instance is born and when application completes the instance dies.

Java Virtual Machine (JVM) is a piece of software that sits on the top of the actual platform (OS + processor). Simply viewed JVM is just an old interpreter. Java achieves its independence from the platform. JVM does many house-keeping tasks such as class loading, memory management, process management and at times threading, as well, for which Java would have had to rely on the Operating System otherwise. JVM defines its own format of the executable, due to which the

executable becomes independent of the platform, since it has to run on the virtual machine and not the actual machine. It takes the responsibility of converting the bytecode in the executable to the native code understandable by the processor. As a result the JVM is itself highly dependent on the platform and there are different flavors of the JVM for different platforms. All these flavors of the JVM accept the same byte code but produce different native codes suitable to the processor.



Java Virtual Machine has a class loader subsystem which is a mechanism for loading classes. Each Java Virtual Machine also has execution engine which is responsible for executing the instructions enclosed in the methods of loaded classes. When JVM executes the program it requires some memory space to store many things like bytecodes, objects the program instances, return values, local variables etc. for that purpose it needs to execute a program into several runtime data area.

JVM has a stack-based architecture that facilitates code optimization by utilizing Just-In Time (JIT) compiler. Runtime data is shared between method area and heap

Explain services of JVM.

JVM- Java 7

- Support for dynamically-typed languages
  - It is also called as InvokeDynamic.
  - The existing set of JVM instructions are
    - statically typed.
    - only works in a debugging environment.
  - Dynamic typed is often more flexible than statically typed.
    - Allows programs to generate or configure types based on runtime data.
    - Dynamic typed languages have more sensible type matching rules.
    - Can perform many type conversions automatically.

JVM's basic functionality is to run the compiled Java programs. The JVM has no built-in support to run dynamically typed language. The current JVM set of instructions are statically typed.

With Java 7 the JVM now has the capability to execute a dynamic language interpreter other than that of Java. As of now support is provided for

- Ruby
- Python
- Groovy
- Java script

Types of Applications

- Java Technology divided into 3 parts:
  - JSE(Java Standard Edition)
    - Console Applications
      - Java Fundamental
    - GUI Applications
      - AWT
      - Swing
    - Applets
  - JEE(Java Enterprise Edition)
    - Client-Server Application
      - Socket
      - RMI

Java technology is divided into 3 parts:

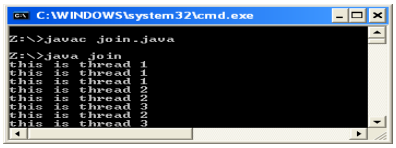
- JSE (Java Standard Edition)
- JEE(Java Enterprise Edition)
- JME(Java Micro Edition)

JSE

JSE is a collection of Java Programming Language API (Application Programming Interface). It is a basic edition commonly known as Java Standard Edition. Console-based and GUI application can be developed using JSE.

Console Application

Java application using simple command prompt or console can be developed. Generally; basic and simple applications are console based applications.



GUI Applications

Console based applications are very basic applications such applications cannot communicate with user properly so, for that purpose a user interactive screen is required. Java supports writing or developing such GUI based applications under J2SE environment. To develop such applications, following techniques are used:

- **AWT**: Abstract Window Toolkit to develop GUI based applications. AWT has number of components like Button, TextField, List etc. inside a package called `java.awt`. AWT also employs different layouts to arrange the UI components properly and increase application usability.AWT components are OS-bound components.
- **Swing:** Swing is similar to AWT but major difference between Swing and AWT is Swing components are Java based components and have number of new components which are not available in AWT. All the components are available in `javax.swing` package.

- **Applets:** If small internet based programs are to be executed in Java, then JSE provides a technique called as Applets. Applets are small internet based programs written in Java and executed inside a browser. Applet based programs can be developed using `java.applet` package.

JEE

The Java Enterprise Edition (JEE) defines the standard for developing multi-tier enterprise based applications. The application development is a thin client environment. Applications are more modular, scalable and secure. The JEE platform has numbers of features of JSE i.e. Standard edition like portability and all. Just like JSE application these applications are also “write once run anywhere”. It can also be said “write once deploy anywhere”. The applications are distributed applications. The types of application are:-

Client-Server Application

Client-server architecture is called two-tier architecture. A two-tier application ideally provides multiple workstations with uniform presentation layer that communicates with a centralized data storage layer. The presentation layer is generally the client. The data storage layer is server. Most of the applications like– email, TelNet, FTP, gopher and even the web –are simple two-tier applications. To develop such applications, following technique are used techniques are used:

- **Socket:** A socket is a combination of port and address (host) which is a communication channel between two programs running on a network. The `java.net` package provides two classes `Socket` and `ServerSocket` to perform client-server applications.
- **RMI:** A RMI is Java technology to develop client-server applications just like socket programming. It is a wrapper over Socket programming to avoid the complexity of socket programming.

Web Applications

Web application is an application that is accessed via a web browser over a network such as the internet or an intranet. It is an application which is resides in side a web.A web application is any application that uses a web browser as a client. Web applications require web servers.

- **Servlet:** Servlet is a server side technology. They are server side components that extend the capabilities of server and provide a powerful mechanism for developing server side programs.
- **JSP:** Java Server Pages is a Java technology that generates dynamic web pages based on HTML.
  - Java code embedded in HTML code.
  - More HTML code and less Java code.
  - Simplify the dynamic presentation layer in a multi-tiered architecture.
  - Separate presentation and business logic

- Template based content generation-JSP.
- Programmatic content generation- Servlet.

Component Applications

The J2EE specifications provide standard multi-tier applications to develop enterprise level applications. To provide a service that service must be a scalable, flexible, and performance based. That service must be secure and re-usable. Apart from this that service must be available to 24*7 for each and every type of client. J2EE specification gives Component Applications using Technology like:

- **EJB:** The Enterprise JavaBeans architecture is component architecture for the development and deployment of component-based distributed business applications. Applications written using the Enterprise JavaBeans architecture is scalable, transactional and multi-user secure. These applications may be written once, and then deployed on any server platform that supports the Enterprise JavaBeans specification.

JME

Java Micro Edition (JME) provides standard environment to develop handheld devices like mobile phones, PDAS, iPhones etc. The JME applications are very light weight. Whenever mobile or smart device application is created then it is tested or debugged on a virtual mobile which is known as emulator.

Connected Limited Device Configuration

The Connected Limited Device Configuration (CLDC) defines the basic set of application programming interfaces and a virtual machine for resource-

constrained devices like mobile phones, pagers, and mainstream personal digital assistants. It provides a solid Java platform for developing applications to run on devices with limited memory, processing power, and graphical capabilities.

- **PDA:** A Personal Digital Assistance (PDA) also called PalmTop; it is a mobile device which connects to internet. A PDA has electronic visual display. A web browser is included in it.

- **Cell Phone:** JME can be used to develop all types of services on all types of cell phones.

Connected Device Configuration

The Connected Device Configuration is a fully compliant Hotspot implementation of Java virtual machine that is highly optimized for resource-constrained devices, such as consumer products and embedded devices. These applications have excellent performance and reliability but here a low memory.

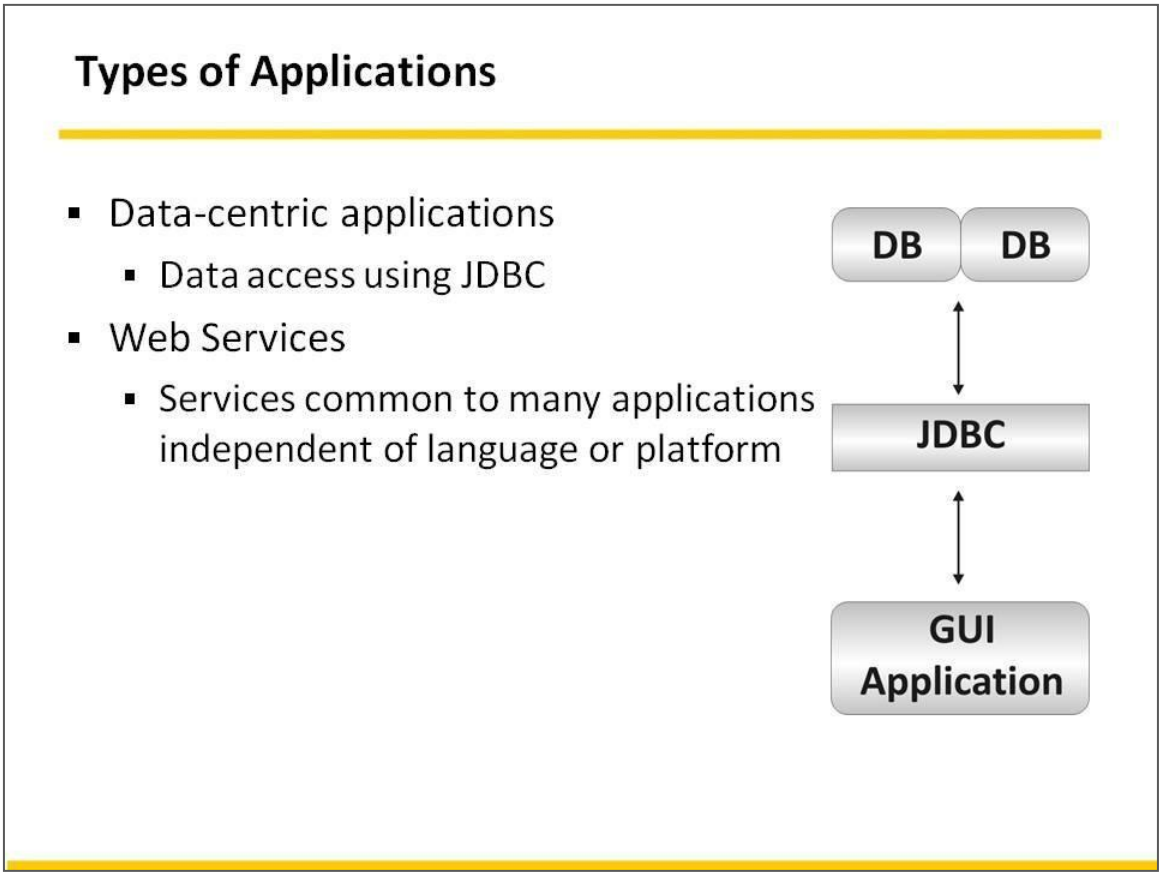
- **Set top-boxes:** A set-top box (STB) or set-top unit (STU) is an information appliance device that generally a tuner. It connects to a television set and is an external source of signal. It turns the signal into content which is then displayed on the television screen or other display device.

Java-Card

Java Card Technology provides a secure environment for applications that run smart cards and other devices with a limited memory and processing capabilities. Number of applications can performed using this single card.

Android

Android is a software stack for mobile devices that includes an operating system, middleware and key applications. An application can be developed on an Android platform using Java programmin language.

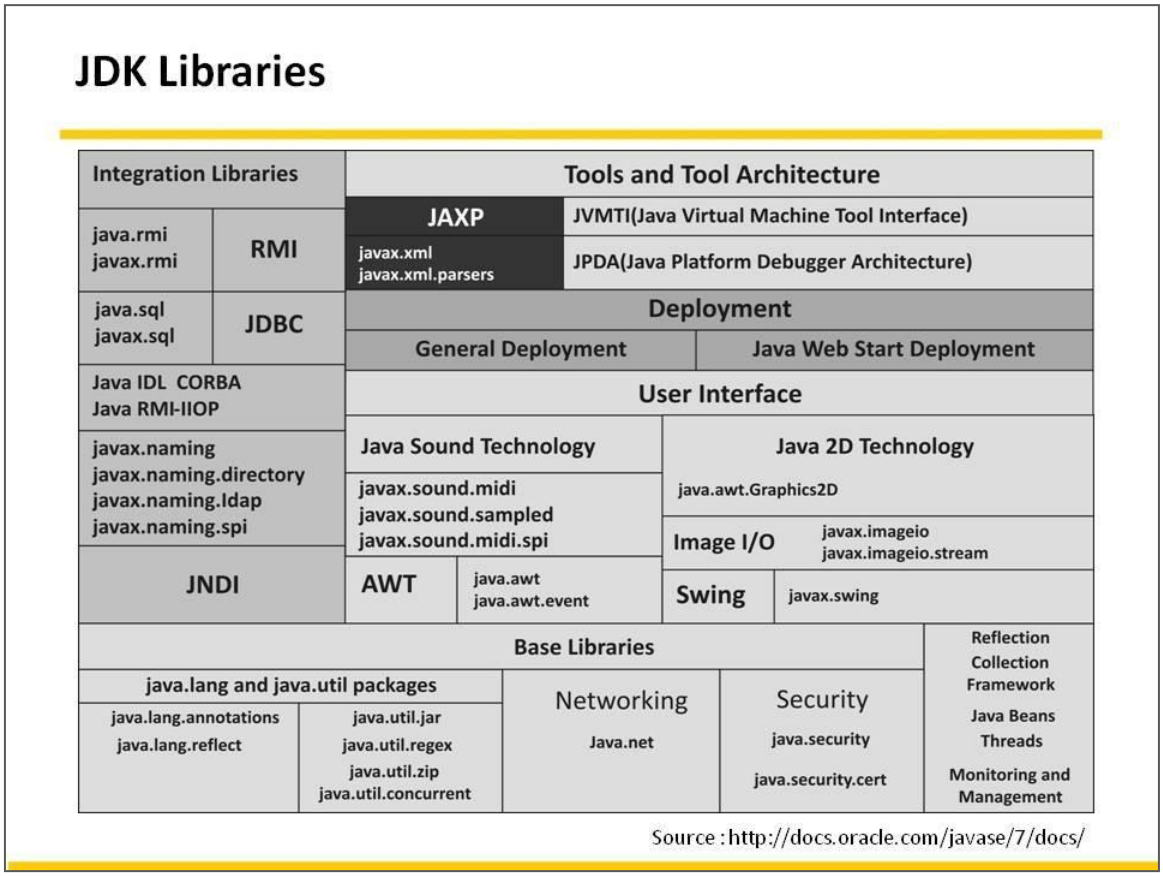


Web Services

Web service is a programmable application component accessible via standard web protocols. In other words, it is a component that can be called remotely from a client application, over the Internet. Web Services allow people, companies, customers, suppliers, doctors, patients, etc. to interact using different computers, different operating systems, and different applications. Web services are the functionality which is defined by web service developer in some programming language and can be used by anyone whenever required so as to avoid re-generation or re-implementation of commonly used functionalities. Web services are platform independent as well as language independent.

Data-centric Applications

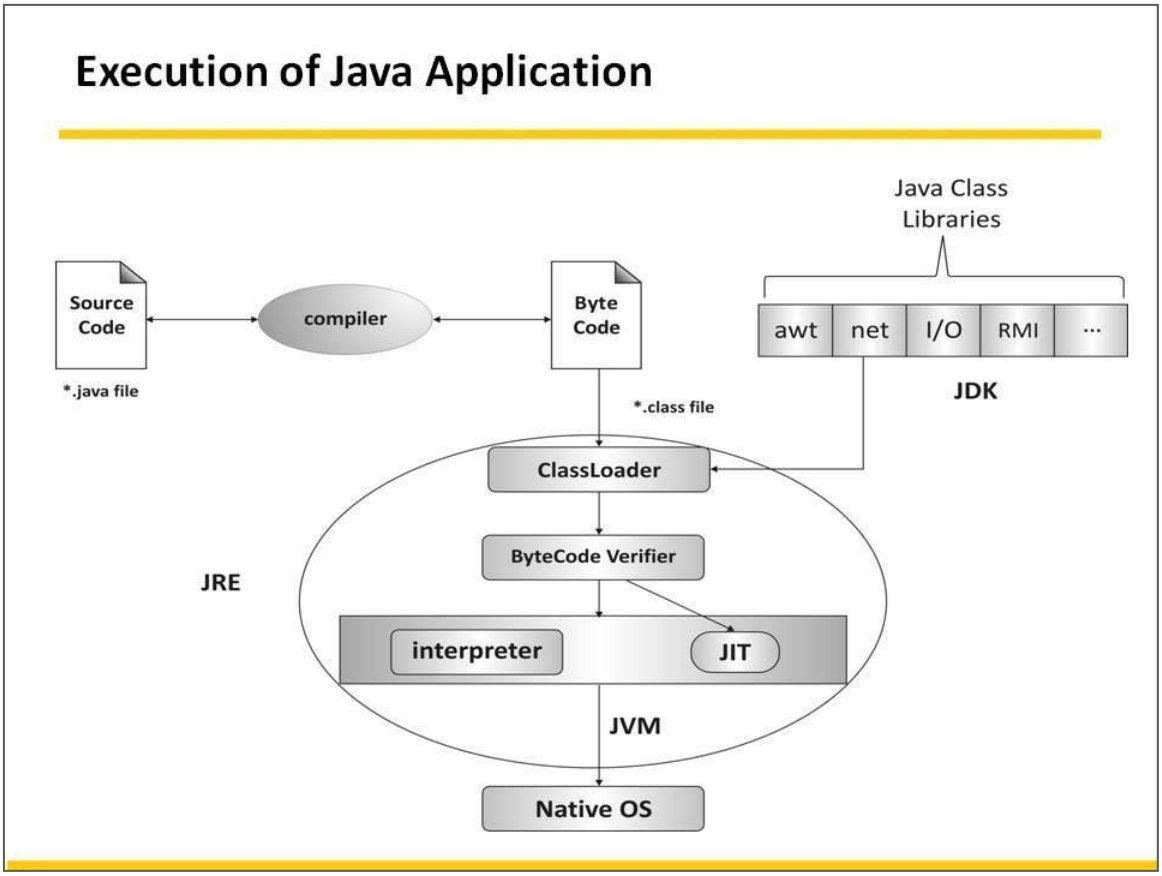
This is one more environment where a Java application finally persists the data inside a database. The application may be J2SE based simple console or GUI based application or JEE based servlet or EJB based application or JEE based PDA or Android based application it always store the information and retrieves the information from database only. To do this kind of applications a technology like JDBC (Java Database Connectivity) is required.



Jdk libraries	packages
java.lang	▪ java.lang.annotation ▪ java.lang.reflect
java.util	▪ java.util.jar ▪ java.util.regex ▪ java.util.zip ▪ java.util.concurrent
Networking	▪ java.net
Security	▪ java.security ▪ java.security.cert
Reflection	java.lang.reflect
Beans	java.beans

Monitoring and Management	java.lang.management
JNDI	▪ javax.naming ▪ javax.naming.directory ▪ javax.naming.Idap ▪ javax.naming.spi
Java IDL CORBA	org.omg.CORBA
RMI-IIOP	▪ javax.rmi.CORBA
JDBC	▪ java.sql ▪ javax.sql
RMI	▪ java.rmi ▪ javax.rmi
JAXP	▪ javax.xml ▪ javax.xml.parsers
Java Sound Technology	▪ javax.sound.midi ▪ javax.sound.sampled ▪ javax.sound.midi.spi
Java 2D Technology	▪ java.awt.Graphics2D
Image I/O	▪ javax.imageio ▪ javax.imageio.stream
AWT	▪ java.awt ▪ java.awt.event
Swing	▪ javax.swing

To get complete information about JDK libraries read Appendix.



A typical Java environment has been depicted in the above diagram. The Java source file is stored as a '.Java' file and is compiled to get an executable which is the '.class' file. The executable is loaded on demand whenever referred to and this is done by a Class Loader. The Class Loader will be discussed a little later. The loaded class is then interpreted by an interpreter and executed.

The interpreter converts the bytecode in the class file to the native code understood by the underlying processor. This native code is never stored and the class file is interpreted every time it is required. This makes the execution of the application slower than what would have been if the application were written in a language such as C. Java people realized this problem and hence they provided a Just-in-time compiler along with the interpreter.

All those class files which are frequently required are compiled into their native code only once by this compiler instead of interpreting them every time. This increases the execution speed to acceptable standards. In this case also, the native code generated is not stored permanently but exists only for the lifetime of the

application. Along with the Class Loader that loads the class there is a bytecode verifier that verifies every file that is loaded for its correctness. The JVM also has a constant access to the API that the language provides. The whole thing minus the compiler that converts the '.Java' file to '.class' file is called the JRE (Java Runtime Environment).

Class Loader

The Class Loader is a utility which closely resembles the loader of the Operating System. As discussed above, Java has tried to become independent of the Operating System by providing some functionalities from its side for which it would have had to depend on the Operating System. The loader locates the executable file on the drive and loads its instance as a process on the primary memory so that the processor can execute it. The Class Loader's task is similar but instead of the actual processor it is the VM that executes the class file. The class files can be on the secondary storage, over the local network or on the internet. Whenever the class is referred to or is instantiated, the Class Loader tries to locate the file on the system and create its instance on the primary memory.

Bytecode Verifier

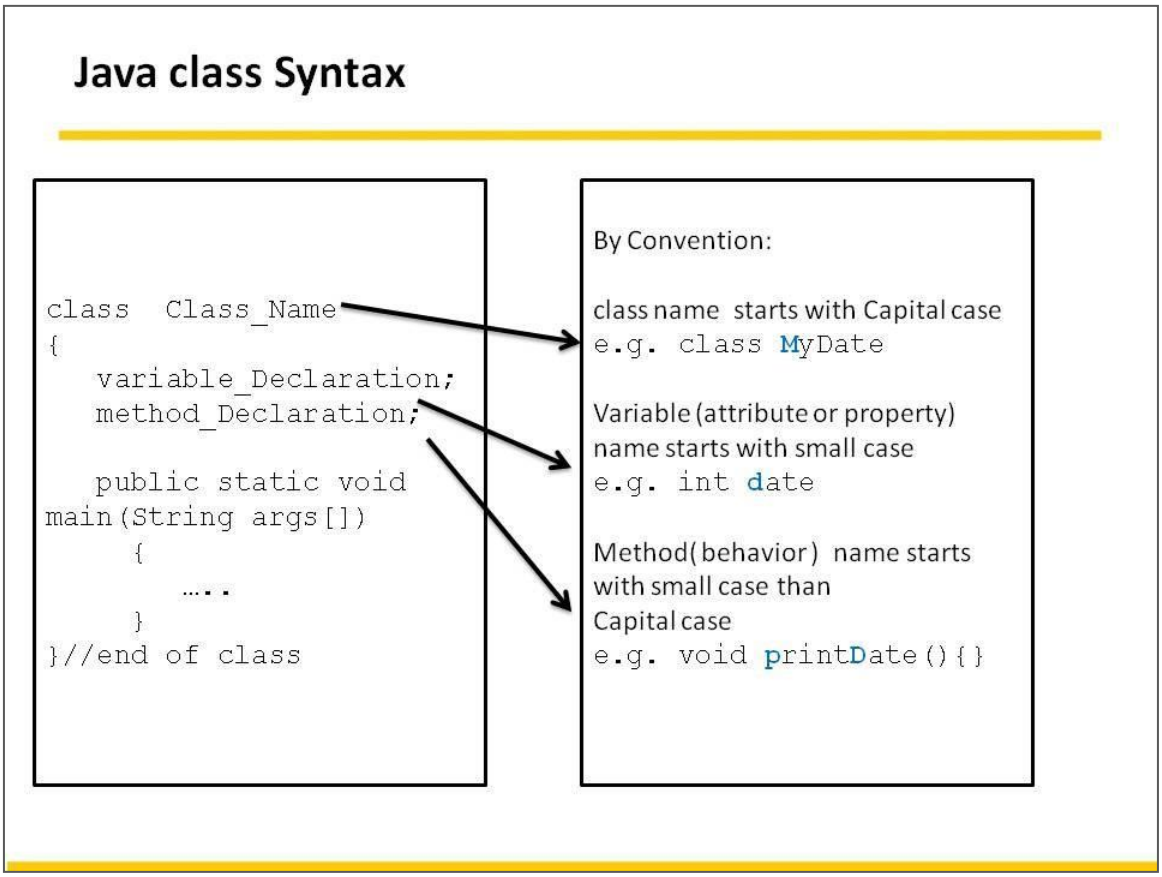
Java supports modular approach of application development. Hence in a typical scenario two classes could be developed by two different teams of programmers. If one class refers to the other class then there is a high possibility that the other class might undergo some modification after the user class was compiled. To resolve such problems Java adopts late binding of code. All the references to the external objects are resolved at runtime. One of the tasks of the Bytecode verifier is to see that all the references are valid and allowed. Moreover it is also the responsibility of the Bytecode Verifier to see that the bytecode that is in the class file has been generated by an authentic Java compiler.

Explain execution of Java Application

Anatomy of JAVA class

- The term 'class' comes from the word called 'Classification'.
- A `class` is a template for creation of like objects.
- An `Object` is an instance of `class`.
- Classes are used to map real world entities into data members and member functions.
- By writing a class and creating objects of the class one can map two key concepts of major pillars of object model i.e. abstraction and encapsulation into software domain.

Class is a way of representing a real world entity in the software domain. It is a template and hence just describes the entity. There is no memory associated with the class and hence it cannot hold any values. One instance of a class can hold values for a specific entity. There can be many instances of same class type each with different values.



Java class syntax is as follows:

Java Class Syntax

```
//Java Class Syntax
 class class-name
{
  variable declaration ;
  method declaration;
  public static void main(String args[ ])
   {
    . . .
   }
} // end of class
```

Writing Java Class

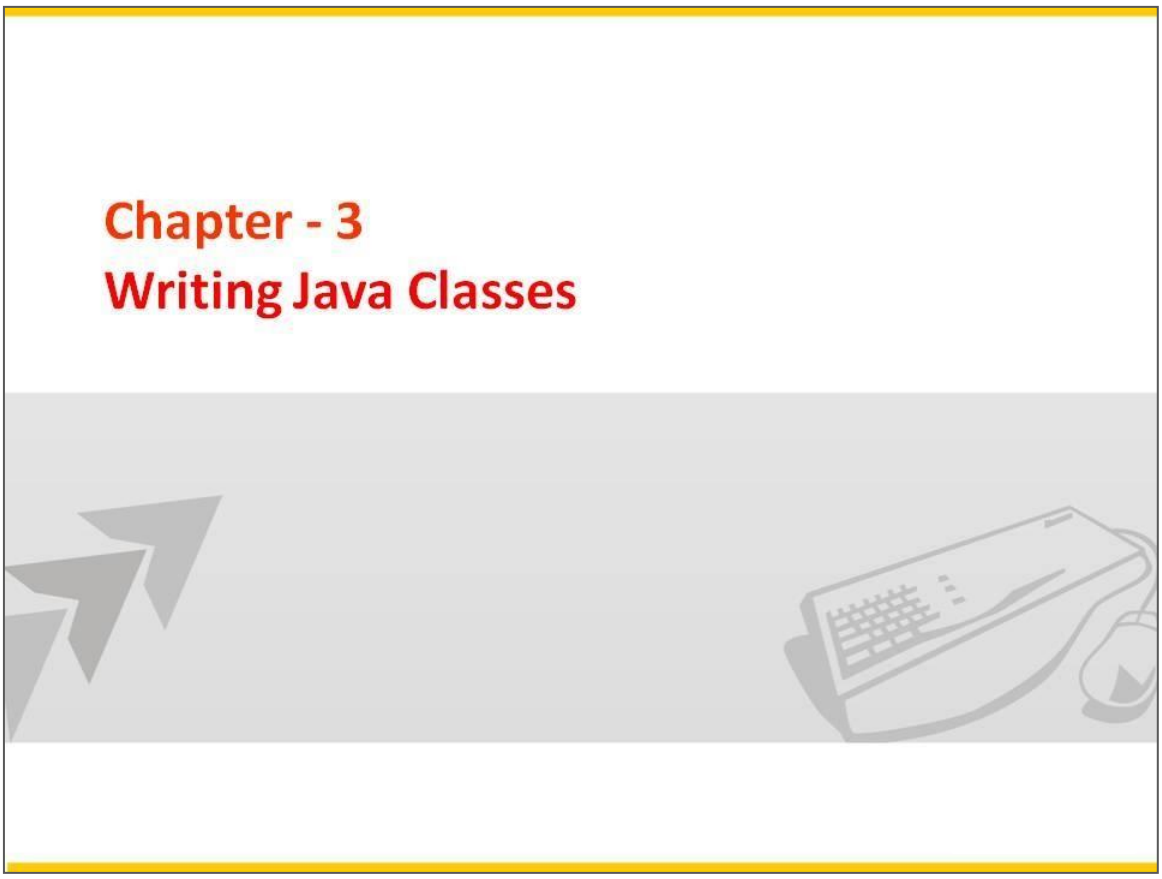
Code Example - 1

```
/* a comment */
// another comment
class Greeting
{
 public static void main(String args[ ])
 {
  System.out.println ("Hello world");
 } // end of main
}   //end of class
```

Code Example - 2

```
class MyDate
 {
   int dd, mm, yy;
   public void initDate ()
     {
     dd = mm = yy = 0;
     }
   public void setDate(int d, int m, int y)
     {
        dd = d;
        mm = m;
        yy = y;
        }
    public void dispDate()
```

```
    {
        System.out.println ("Date is : " + dd +"-"+ mm
+"-"+ yy);
    }
    public static void main(String args[])
     {
     MyDate d1;
       d1 = new MyDate ();
     d1.initDate ();
     d1.dispDate ();
     d1.setDate (3, 7,90);
     d1.dispDate ();
    }
}
```



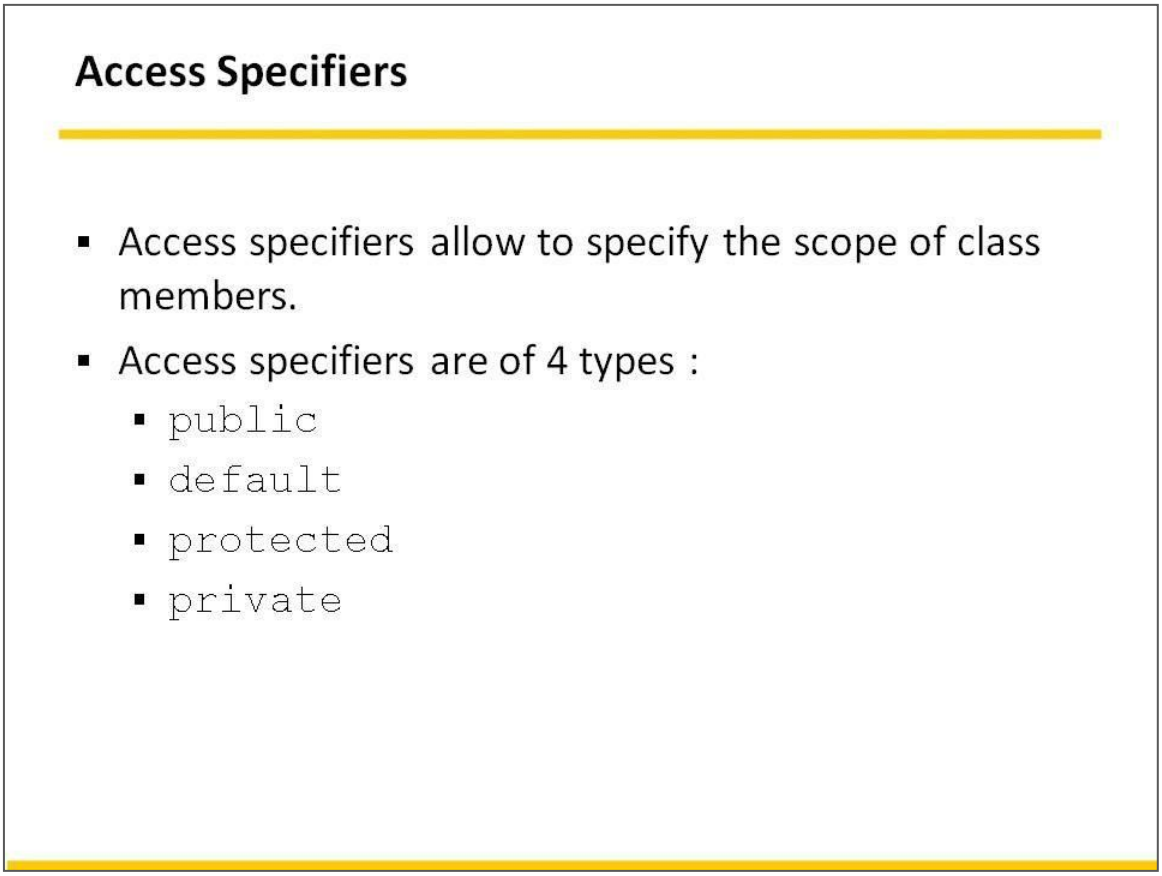
This chapter covers basic fundamentals of writing Java classes. It includes concepts of access specifiers, accessor-mutator, and constructors. It covers `this` and `static` keywords and contains concept of packages.

Objectives

After completion of this chapter you will be able to:

- Use access specifiers: public, private, protected and package.
- Create accessor and mutator methods.
- Create a consturctor and overload it.
- Create an object and invoke methods of MyDate class.
- Create and use finalizer.
- Use primitive data types.
- Identify the need of `this` keyword.
- Differentiate between static and non-static variables and methods
- Implement `toString` method and demonstrate its use.

- Create user defined package and use it.
- Configure CLASSPATH variable.
- Use package access specifiers.
- Use static import shortcut for importing static variables and methods from a package.



In Java, encapsulation is implemented using access specifiers. There are four access specifiers that give different visibilities to the data and methods inside a class.

The public access allows anyone to access the data or method specified with this access. It is the maximum visibility that a class can provide.

Data and methods declared without any access specifier are said to have default access. The default access allows all the classes within the same package to access the data and methods.

Protected variables and methods allow access to all the classes in the hierarchy, regardless of the package structure.

The private access is the most restrictive of all. It allows only the methods of that class to access the data and methods declared with this access. It hides the data from the rest of the world only making it accessible through the public methods of the class. This is exactly what encapsulation talks about. Hence it is recommended that data should be kept private within a class as far as possible.

```
class MyDate
{
  int day, month, year;
  public void setDay(int dd)
  {
   day=dd;
  }
  public int getDay()
  {
    return day;
  }
...
}
```

Accessor and Mutator Methods

- Accessor methods merely access instance fields/variables.
- e.g. getXXX() methods
- Mutator methods actually change contents of instance fields.
- e.g. setXXX(...) methods

```
class MyDate
{
   private int day,month,year;
   public void setDay (int d)
   {
      day=d;
    }

   public int getDay()
   {
      return day;
   }
}
```

Accessors and Mutators are setter and getter methods. The way to force the data encapsulation is through the use of accessors and mutators. An accessor method is used to return the value of a private field. A mutator method is used to set a value of a private field.

Practice to solve the questions based on the access specifier usages and errors related them.

Writing private data members and public methods is a good programming practice since it can facilitate validation.

Create an Object

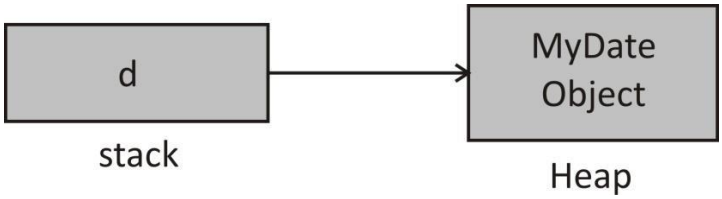
```
public static void main(String args[])
{
      MyDate d;
      d=new MyDate();

            // OR

      MyDate d=new MyDate();
}
```

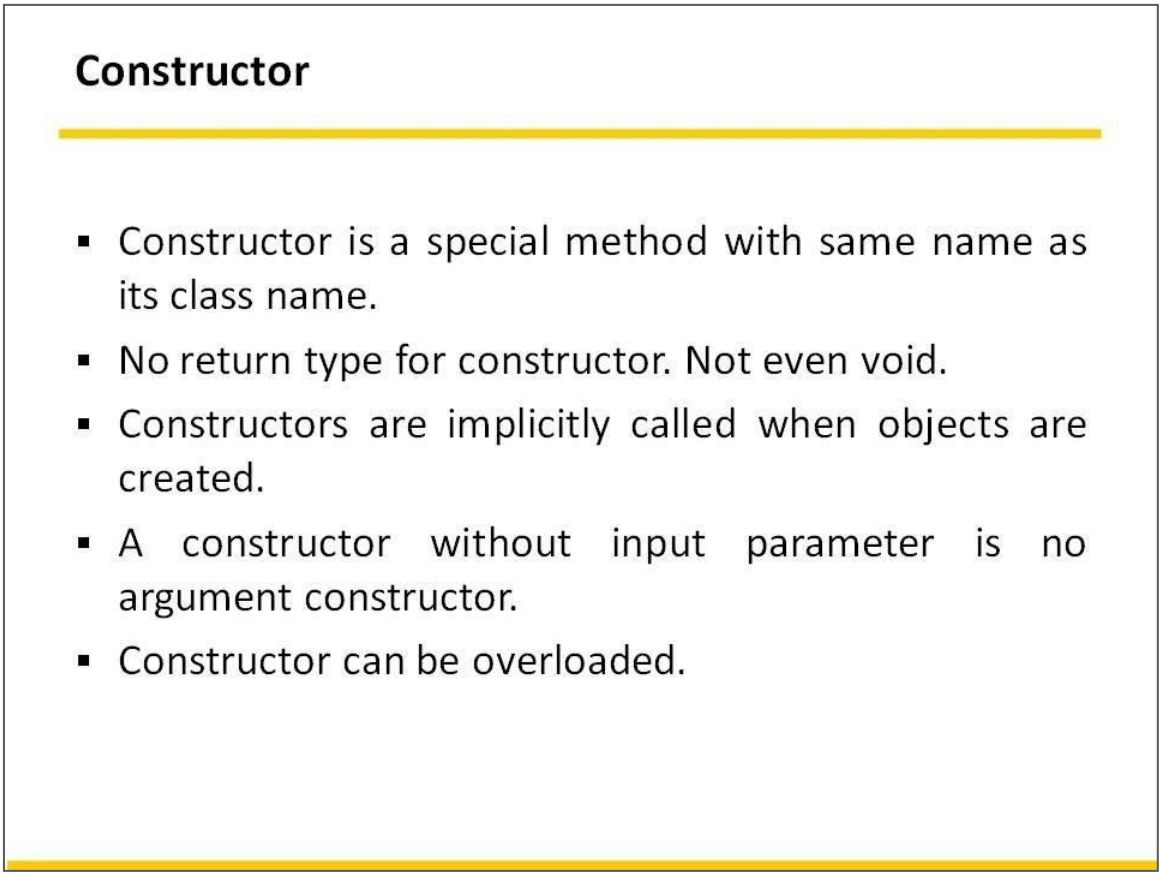
Object is an instance of a class. It is created using `new` keyword in Java. Objects are always created on heap, whereas, references are stored on stack. For example, consider the following snippet of code.

Objects always created on heap.

```
public static void main(String  args[])
{
  Date d1;
  d1 =new Date();
}
```

In the code snippet above, `d1` is just a reference to the actual object of `Date` class, yet to be created. The actual object is created using `new` keyword in the second statement where memory is allocated. The reference of this memory is assigned to `d1`.





A constructor is a special member function used to initialize the values of the attributes of an object. This function is implicitly called when an object is created. It is not mandatory to define a constructor. In this case the compiler provides a default constructor but the attributes are initialized to default values.

When an object is created, the data members should be initialized. For example, the data members of `d1` object are initialized to some default values and that of `d2` are initialized to the values passed in the parenthesis.

Following are the rules to create constructor

- Constructor name is same as class name.
- If a constructor is not defined for a class, a default parameterless constructor is automatically created by the compiler. This is called default constructor. It calls the default parent constructor (super ()) and initializes all instance variables to default value (zero for numeric types, null for object references, and false for booleans).

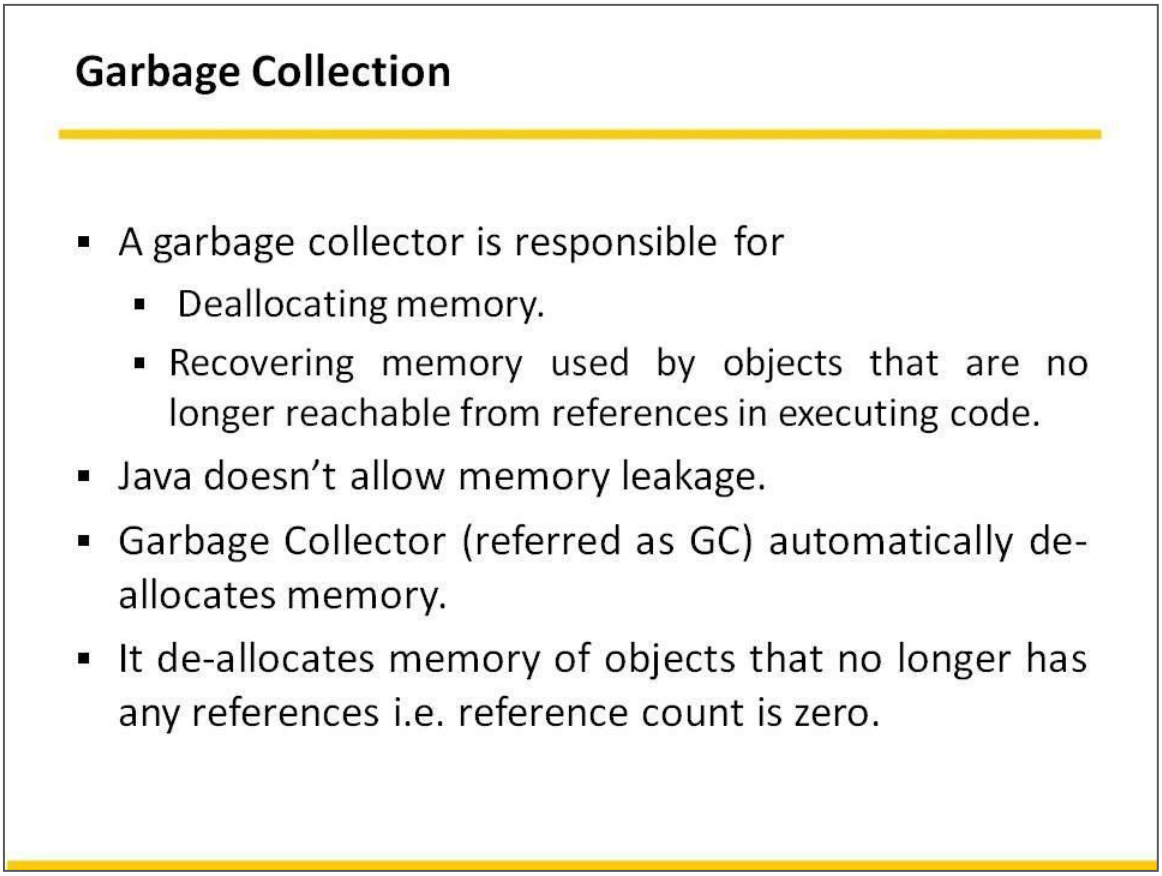
- There is no return type given in a constructor signature (header). The value is this object itself so there is no need to indicate a return value.
- There is no return statement in the body of the constructor.
- Constructors can be overloaded.

Note: Concept of overloaded discussed later in chapter.

Code Example

```
class Date
{
  //data members
  private int day;
  private int month;
  private int year;
  //Default Constructor

   public Date ()
   {
    day = month = year = 1;
   }
    //Parameterized Constructor
    public Date (int d, int m, int y)
     {
    day = d;
      month = m;
        year = y;
     }
   public static void main(String args[])
     {
```

```
      Date d1=new Date();// no argument Constructor
will be called.
      Date d2=new Date(1,1,2004); //Parameterized
Constructor


    }
}
```



Unlike C++, in Java memory is allocated and freed automatically. The technique, which implements this approach, is called as Garbage Collection.

Garbage Collector is a program that runs frequently, steadily and on regular basis. If there are any dangling references then Garbage Collector reclaims that memory. The actual implementation of Garbage Collector depends on vendor.

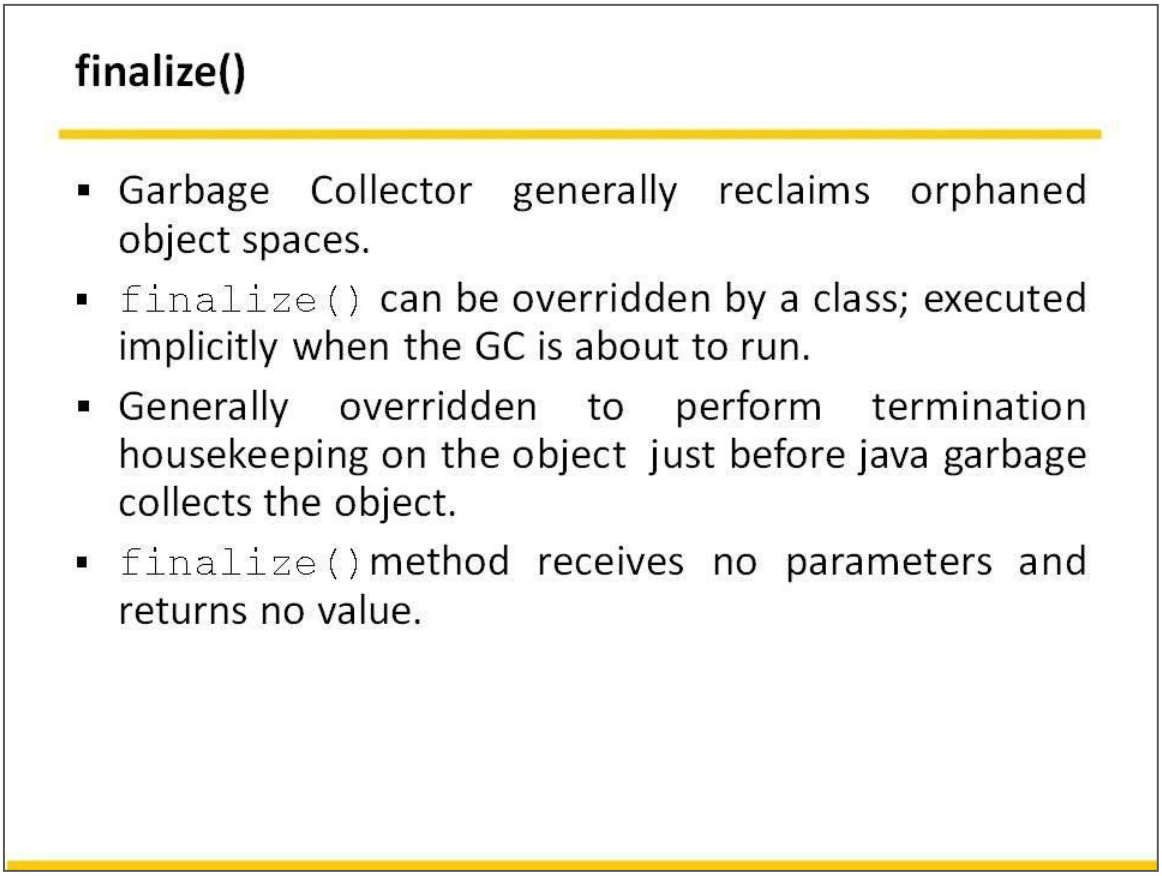
Java performs garbage collection and eliminates the need to free objects explicitly. When an object is 'no longer referenced', when there is no reference to the object in any static data, nor in any variable of any currently executing method, nor can a reference to the object be found by starting with static data and method variables etc., it is automatically garbage collected. Object can be created using `new`, but there is no corresponding delete.

When an object is no longer required or referred to or control returns from a method so its local variables no longer exists; when an object has no references to it anywhere, except in other objects that are also unreferenced, it can be garbage collected.

Garbage collection means never having to worry about 'dangling references'. Suppose object1 has a reference to object2.systems in which there is a direct control when to delete objects,object1 can be deleted.Now,object1 points to a space that the system considers free. This would create all manner of havoc when it uses the values in that space as if they were part of something they are not. Java solves this problem because an object that is still referenced somewhere will never be garbage collected.

```
System.gc() Or Runtime.getRuntime().gc()
```

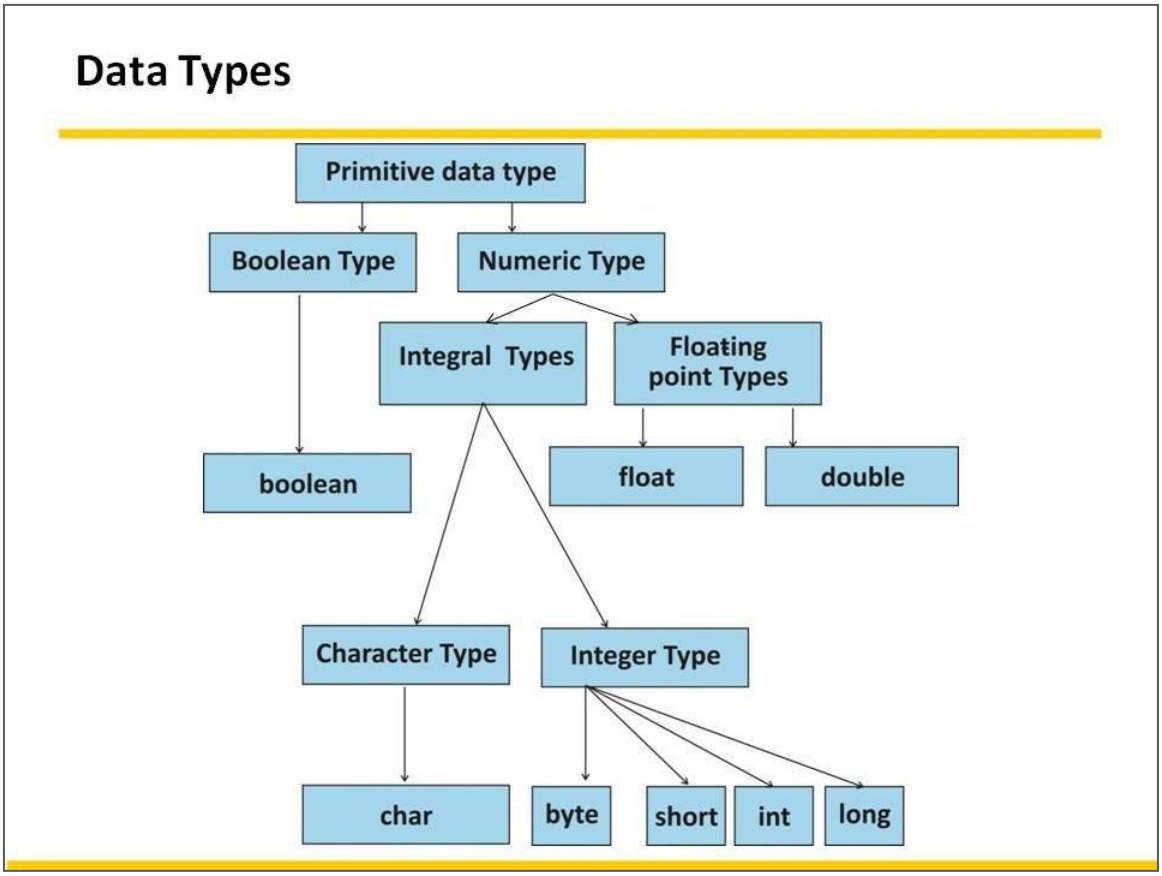
The above method runs the garbage collector. Calling the GC method suggests that the Java Virtual Machine expend effort toward recycling unused objects in order to make the memory they currently occupy available for quick reuse. When control returns from the method call, the JVM has made best effort to reclaim space from all discarded objects.



It is not normally noticed when an orphaned object's space is reclaimed. A class can implement a `finalize()` method that is executed before an object's space is reclaimed. Such a method gives a chance to reclaim other non-Java resources.

Using `finalize()` method is important when dealing with non-Java resources that are too valuable to wait until garbage collection. For example, open files cannot wait until finalize phase to be reclaimed. There is no guarantee that the object holding the open file will be garbage collected before all the open file resources are used up.

Objects that allocate external resources should provide a finalize method that cleans them up or class may create a resource leak.



Java Primitive Data Types

Java supports most of the primitive data types that languages such as `C` and `C++` provide and more. It also includes a `boolean` data type.

Type	Storage Requirement
`int`	4 bytes
`short`	2 bytes
`long`	8 bytes
`byte`	1 byte
`boolean`	1 bit
`float`	4 bytes

`double`	8 bytes
`char`	2 bytes

The above hierarchy gives a brief account of all the primitive data types supported by Java and their relation with each other.

Method Overloading

- Reusing the same name for method.
- The method calls are resolved at compile time using method signature.
- Compile time error occurs if compiler cannot match the arguments or if more than one match is possible.
- Method signature consists of
  - Number of arguments passed to a function.
  - Data types of arguments.
  - Sequence in which they are passed.

When the same task needs to be done but using varying number of parameters, different methods need to be written. Each of these methods has to be given a different name. So, in spite of doing the same task, the methods will have different names. The user has to use these methods as per his requirement.

```
public static void main(String args[])
{
  MathEngine me=new MathEngine();
  int res = me.add1(12,13,14);
  int num = me.add2(12,14);
  double sum = me.add3(12.5,13.5);
}
```

To solve this problem Java comes with concept called method overloading. Methods with same name but different parameters and sequence in which they

are passed to the method is overloading. Compiler decides the correct method to be invoked depending on the call of the method.

Example

```
int add (int a, int b)
float add (float a, float b)
float add (int a, float b)
float add (float a, int b)
```

Following are the rules of method overloading

1. The two methods should have different number of parameters or arguments passed.
2. If the two methods have same number of parameters than sequence of parameters must be different.
3. It is not sufficient for two methods to differ only in their return type.

Code Example

Refer the following code. When the call is given as `add(12, 13)`, the method with two paramters will be called. If the call is given as `add(12, 13, 14)`, the method with three paramters will be called.

```
public class MathEngine
{  public int add(int num1, int num2)
  {
     . . .
  }
 public int add(int num1, int num2, int num3)
 {
  . . .
 }
}
class Program
```

```
{
  public static void main(String [] args)
  {
     MathEngine me=new MathEngine();
     int res = me.add(12,13);
     int num = me.add(12,13,14);
  }
}
```

Name mangling is the mechanism of Java compiler to link functions and variables of same names to resolve them to some unique symbol names by altering the names with some extra information line index, argument size etc

To get complete information use the following link

http://en.wikipedia.org/wiki/Name_mangling

varArgs

- varArgs means variable arguments.
- Restricting number and type of input parameters while declaring the method.
- Solution:make the signature dynamic.
- To represent variable argument use operator called as ellipses (…) 3 dots after data type.

```
static void addTest (String message, int... numbers)
 {
     int result = 0;
     for(int num : numbers)
          {
               result = result + num;
          }    return result; }
}
```

Consider the following code snippet showing the addition of two numbers being done in the `add()` method.

```
class AddTest
{
 void add(int x,int y)
  {
   int result=x+y;
   System.out.println("Addition is "+" "+result);
  }
 public static void main(String args[])
 {
  AddTest at=new AddTest();
  at.add(10,20);
```

```
 }
}
```

Consider now `add()` method wants to add 3 numbers, so a third parameter will have to be introduced in the specific method. Some changes will have to be made in the program. likes this:

```
void add (int x,int y,int z)
{
 int  result=x+y+z;
 System.out.println("Addition is "+" "+result);
}
```

If number of parameter values are not fixed or they are going to increase then every time will have to be made changes in the program or that many numbers of overloaded methods will have to be written.

To solve this problem Java 5 comes with the feature called **'Variable Arguments'** (**Var-Args**)

The declaration of method with variable number of arguments is

```
void add(int… x){}
```

Inside the parentheses, three dots are given, after the data type '`int`' followed by the variable name. These three dots are called as ellipsis.

To avoid the versions of overloaded methods generally varArgs are used. Internally, a variable argument is maintained as an array that can hold zero or one or more arguments of the same type. Method that accepts zero to many arguments is called as varArgs method.

Following are the rules of varArgs:

1. More than one variable argument types are not accepted.

```
void add(String…s,int… x){}
```

This is an invalid declaration of method.

2. Variable argument type must be the last parameter.

```
void add(int…x,String s) {}
```

This method declaration is also invalid.

Code Example

Consider the following code snippet for declaration of varArg method.

```
class AddTest
 {
   int result=0;
   void add(int… x)
     {
       for(int i:x)
        {
          result=result+i;
        }
       System.out.println ("Addition is "+"
"+result);
     }
  public static void main(String args[])
    {
      AddTest at=new AddTest ();
      at.add (10, 20);
      at.add (10, 20, 30);
      at.add (10, 20, 30, 40);
    }
}
```

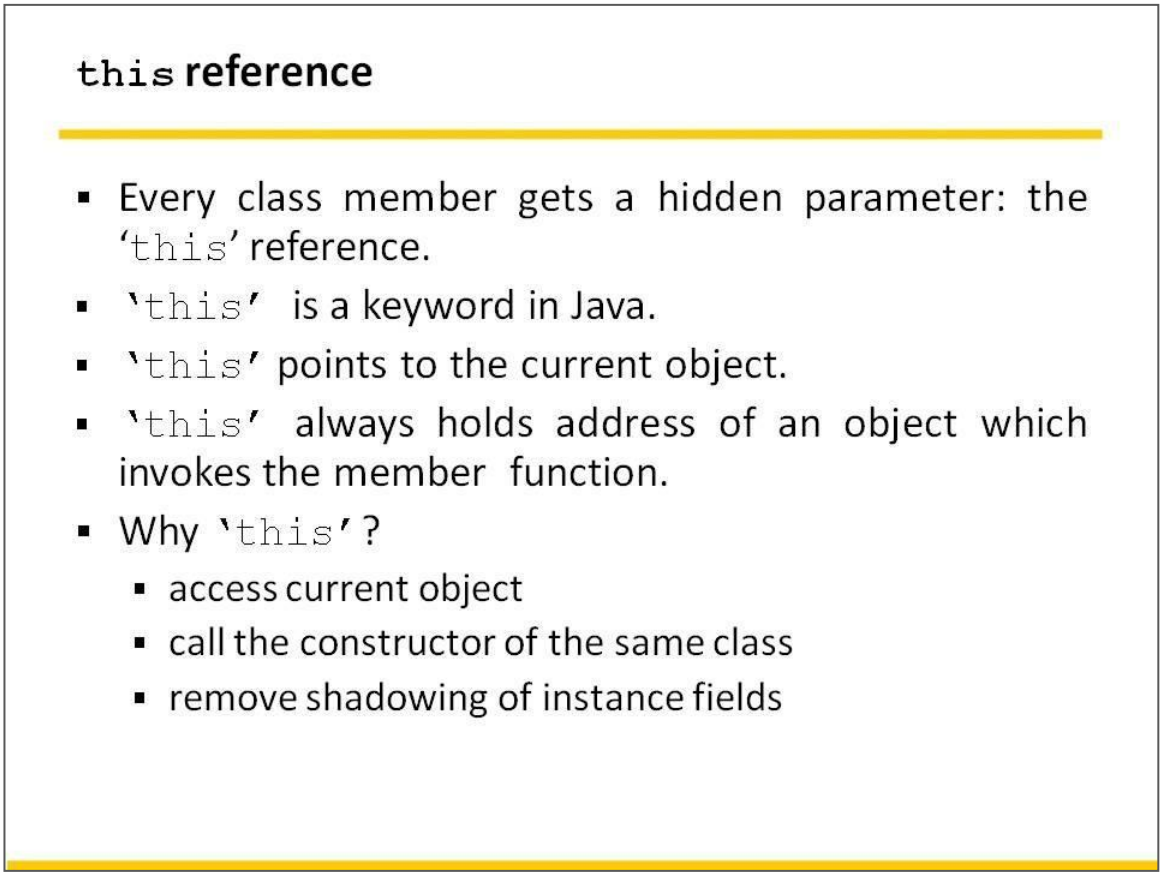
Simplified varArgs Method Invocation

- Mix varArgs with generics.
- VarArgs method with non- reifiable varArgs type.
- Compiler generates "unsafe operation" warning message .
- Java 7 removes this warning message.
- VarArgs method annotated with safeVarArg annotation.

Code Example

```
import java.util.*;
public class DemoSafeVarArgs{
@safeVarargs
static <T> List<T> asList(T… tt){
  System.out.println(tt);
  return null;
}
@SuppressWarnings({"unchecked"});
void foo(){
  asList(new ArrayList<String>());
}
}
```

Java 7 comes with a new feature called simplified varArgs. It provides a way to remove a compiler warning about generics varArgs invocation. Using this varArg methods can be annotated with safeVararg annotation.



`this` reference is used to

- Access the attributes of current object.
- Remove the shadowing of instance members when the parameter name and the attribute name are same.
- Call the constructor of the same class.

Any method or property of a class is invoked by using an object. Such methods are called instance methods. Whenever an object invokes a property or a method, `this` reference is passed implicitly to them. It holds the reference of current object that invokes the member. `'this'` is a keyword in Java.

- `this` keyword must be the first statement in the constructor block.

Following code snippet shows how the shadowing of instance members can be removed when the parameter name and the attribute name are same.

Code Example

```
public MyDate(int dd,int mm,int yy){
```

```
    this.dd=dd;
    this.mm=mm;
    this.yy=yy;
}
```

Following code snippet shows how to call the constructor of the same class. This is also called as Constructor Chaining.

Code Example

```
public MyDate(){
this(20);
}
public MyDate(int dd){
this(4,2010);
this.dd=dd;
}
// Assume that two-argument constructor is present in
the class


//In main
MyDate m=new MyDate();
```

Static Variable

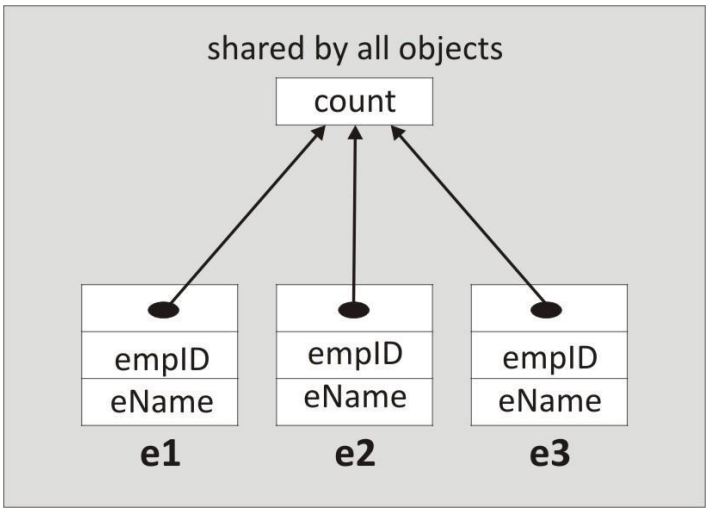
- Some characteristics or behaviors belong to the class rather than a specific instance
  - `interestRate`, `calculateInterest` method for a `savingsAccount` class
  - `count` variable in `Employee` to count the number of objects
- Such data members are static for all instances
  - Change in static variable value affects all instances

Normally, every object of a class has its own copy of its data members. But there may be a certain attribute or characteristic which has the same value for all the objects of the class. In such a case, this characteristic is said to belong to the class rather than a specific instance or object. For example, `InterestRate` in `Account` class, `count` variable in `Employee` class.

Such data members are declared as `static`. Since only one copy of the data is maintained for all objects of the class, it is also called `class` variable.

```
class Employee
{
   int empId;
   int eName;
   static int count;
   public Employee(string name)
   {
```

```
        count++;
        empId = count;
        eName = name;
    }
}
```



`e1`, `e2` and `e3` are objects of class Employee. `count` is a static member of the class that is used to generate an id for every employee. It is shared by all the objects of the class.

List two scenarios where static data members can be used.

If user wants to create single instance of class. The best practice is singleton design pattern.

To get complete information use the following link

http://www.javabeginner.com/learn-java/java-singleton-design-pattern

Static Method

- Static method can access static members only.
- Static method is invoked using class name
- <class name>.<method name>()
- Reference 'this' is never passed to a static method.

```
public class Employee
{
 static int count;
 static int
  showCount()
  {
   return count;
  }
}
```

```
public static void main(String args[])
{
   int numberOfEmployees=
   Employee.showCount();
System.out.println(numberOfEmployees);
}
```

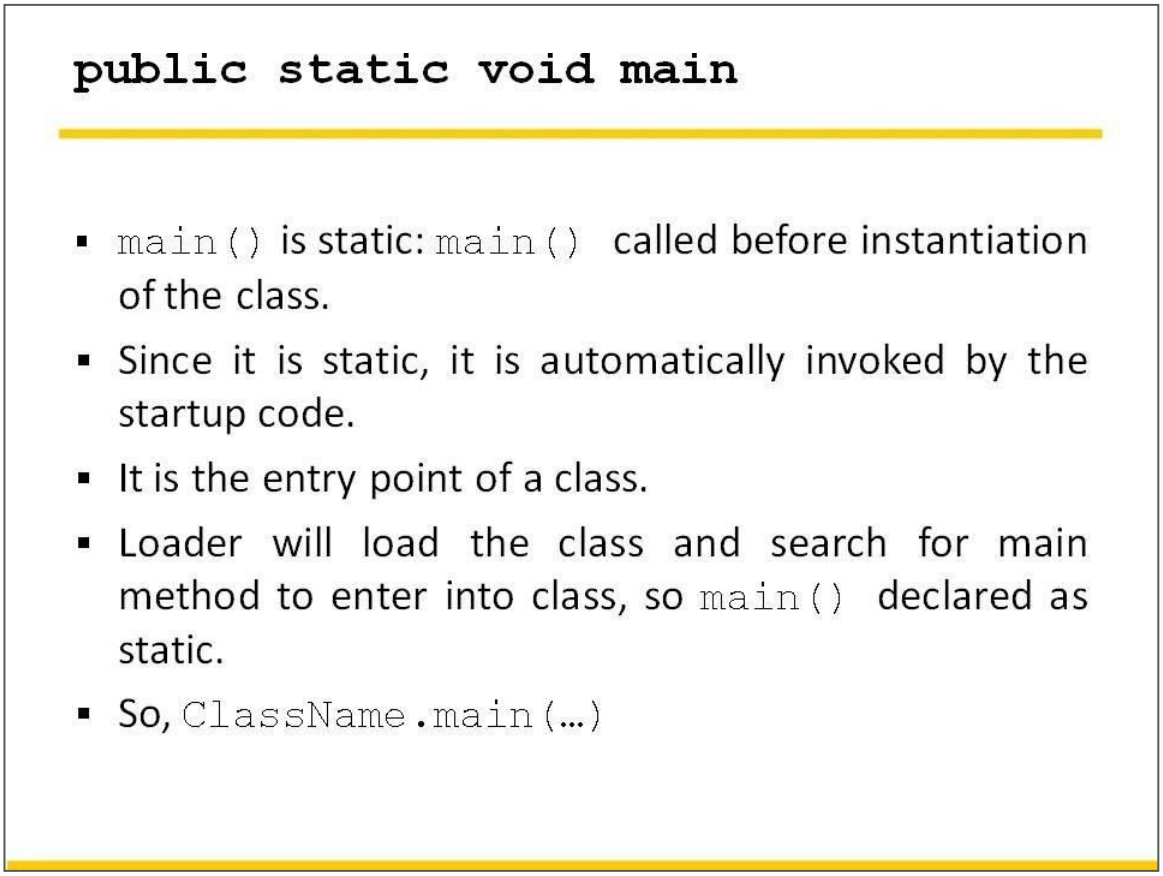
Each object does not have a copy of static members. They are class variables. A static method is needed to access these members. Rather static methods can access only static members. Though they are accessible from instance methods, it does not make sense to use them there.

A static method is invoked using the class name as shown above. This is because reference `'this'` is never passed to a static method.

In the example mentioned on the slide, `count` is a static variable that is used to create an id for every employee. A method `printId()` is used to print the id. It is a static method and so it is called using class name. As `count` is shared by all objects, this method will always print the last id generated.

`main()` method is a static method. It is an entry point method that is called automatically. It is called before instantiation of the class that contains it. Actually, class loader checks the class that contains `main()` method and loads it.

Distinguish between static method and instance method.



Details of invoking a Java application vary from system to system, but whatever the details, the name of a Java class that drives that application must be provided. When a Java program is executed, the system locates and runs the `main()` method for that class.

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

An application can have any number of main methods, since each class can have one. Only one main is used for any given program. Being able to have multiple main methods, each class can have a main that tests its own code, providing a facility for unit-testing a class. The `main()` method is static and public because class loader invokes this method before class instance is created. This method has array of `String` as an argument called as command line argument. Any initial information can be passed to a class through this array.

static initialization blocks

- Arbitrary blocks of code.
- Executed before `main()` when class is loaded.
- Used for initializing static variables.
- A class can have more than one static blocks.
- If more than one static blocks exists in a program then called in the order they appear in the source code.

```
static{

//manipulation of static variables
}
```

Sometimes initialization is required before main application starts, like database connections. Here connection field has to be declared before main application starts.

So, to fulfill this requirement a class can have static initialization blocks to set up static fields. A static initialization block is used whenever simple initialization statements for static members are either not possible or are too clumsy. It is a block of code executed before `main()`.

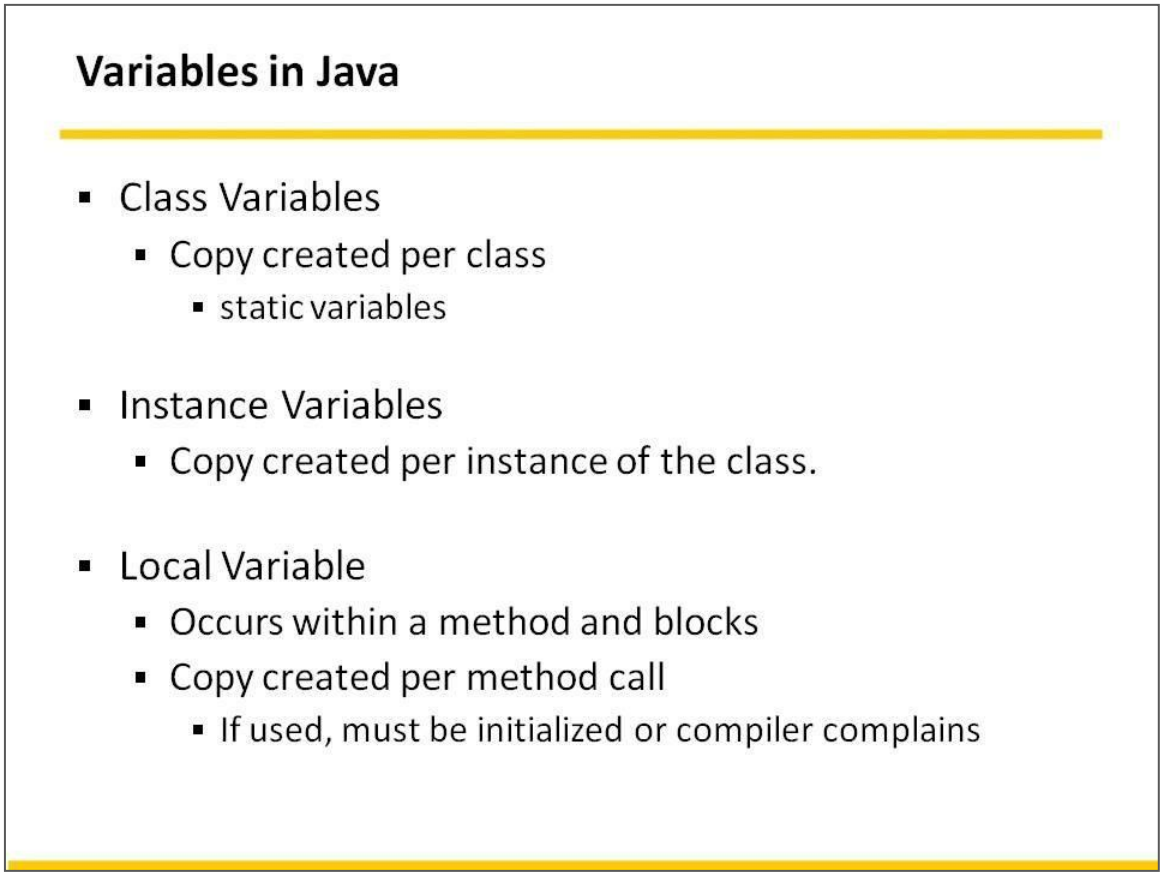
In general, Java executes the statements in following manner:

- Java <class name>
- static blocks executed
- main method invoked
- Constructor(s) invoked
- Method Invocation(s) according to order in main function
- Following are the rules of the static initialization block:

1. Static block always executes before `main()`.
2. A class can have more than one static block.
3. Execution of these blocks is in order as they appear in source code.

Code Example

```
class MyClass {
 static Connection con;

static{
    con=DriverManager.getConnection (…);
     }
…
}
```



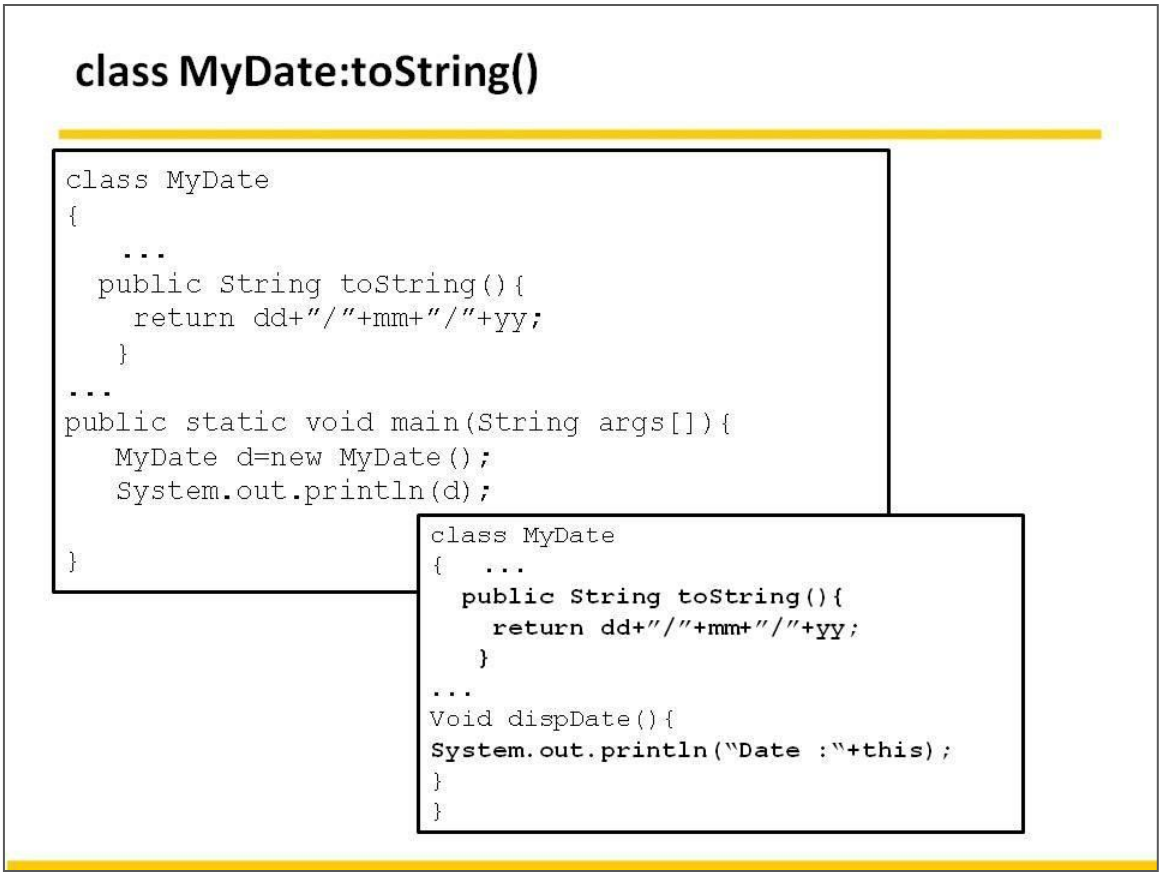
There are three types of variables in Java. They are:

1. Class variables: These are static variables and copy is created per class.
2. Instance variables: Copy is created for each object or per instance of a class.
3. Local variables: These variables occur within methods or blocks and for these variables copy is created per method call. If local variables are used they must be initialized otherwise compiler throws error.

Terminology for Class Members

`Instance Members`	These are instance variables and instance methods of an object. They can only be accessed or invoked through an object reference.
`Instance Variable`	A variable, which is allocated when the class is instantiated. i.e. when an object of the class is created.
`Instance Method`	A method, which belongs to an instance of the class.

	Objects of the same class share its implementation.
Static Members	These are static variables and static methods of a class. They can be accessed or invoked either using the class name or through an object reference.
Static Variable	A variable which is allocated when the class is loaded. It belongs to the class and not to any object of the class.
Static Method	A method which belongs to the class and not to any object of the class.



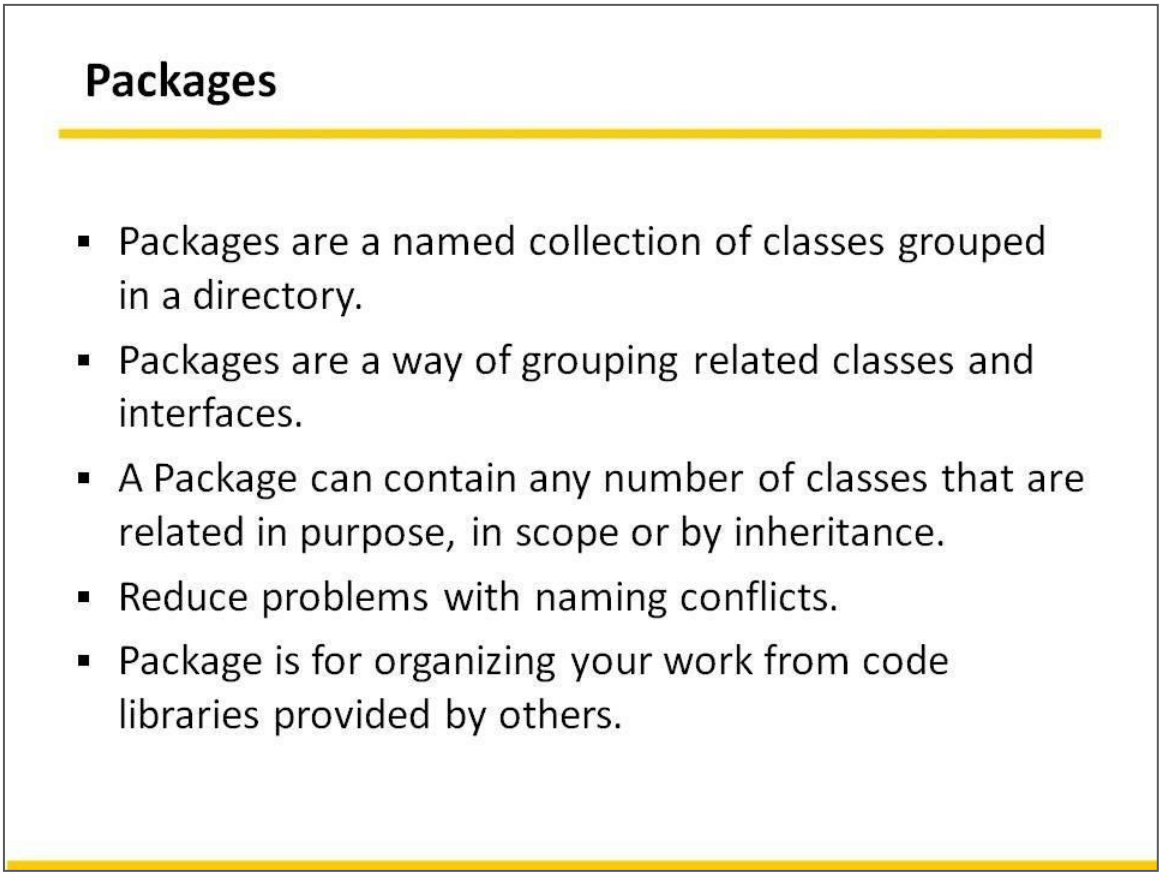
`toString()` returns a string representation of the object. In general, the `toString()` method returns a string that "textually represents" this object. The result is informative representation that is easy for a person to read.

Following snippet of code shows the use of `toString()` method.

```
class MyDate
{
    ……
  public String toString ()
  {
   return dd + "/" + mm + "/" + yy;
  }
    ……
  public static void main (String args[])
```

```
  {
   MyDate d1 = new MyDate ();
   System.out.println (d1);
  }
}
```

```
class MyDate
{
    ……
    public String toString (){
        return  dd + "/" + mm + "/" + yy;
    }
    void dispDate (){
        System.out.println ("Date : "+this);
    }
}
```



Packages in Java are named collection of classes grouped in a directory. A package is a set of types grouped together under a common package name. Each type has a *simple name,* and each package has a *package name*.

They are required for following purposes:

Avoiding Name Conflicts

When a Java program is designed, the problem domain is modeled by identifying and defining types and assigning a name to each type. Each type name must be unique. If a large application is designed, there may be name conflicts as many developers are working on the same application. To address the problem of name conflicts, packages are used. Thus logically related classes can be a part of one package.

Packages make names of types more distinct. In Java, every class belongs to some package.

Libraries

Any Java application that is written makes use of libraries developed by others and made available to a program as packages.

Hiding the Implementation

Special access privileges can be granted between types within the same package, and can be declared to be accessible only to other types within the same package. The full details of how to do this will be given later in this chapter as part of a discussion of Java's access levels.

A package is a collection of related classes and interfaces. It is a convenient way for organizing work and separating it from code libraries provided by others.

- Packages create a grouping for related interfaces and classes.
- Packages reduce the problems of name conflict.

Packages can have types and members that are available only within the package (they control visibility of classes. Multiple classes of larger programs are grouped together into a package. Packages correspond to directories in the file system, and may be nested just as directories are nested. Small, single-class programs typically do not use packages.

Steps for Creating a Package

PATH variable is used for locating executables on actual machine. It is used by the OS.

CLASSPATH variable is used for locating executables on virtual machine. It is used by JVM.

- Use keyword `package` at the beginning of the file.
- Compile the file using –d flag which creates a directory and keep the `dot (.)class` files inside the directory.
- Set the `class path` from the root up to subdirectory created above.
- Use import keyword whenever the class in a particular `package` has to be used.

Defining a Package

The general form of a package statement is:

```
package package_name;
```

- Only one package statement is allowed in a program file.
- Package name must be the first non commented statement in the file.
- If the package statement is omitted from the file, the classes generated will be put under an unnamed default package that is always imported.

Accessing classes from a package

- Any method or data member can be referred in the program by using fully qualified name. For example,

```
package_name.class_name.method_name ();
```

- The `import` statement allows classes and interfaces defined in packages to be referred to solely by class names instead of the fully qualified names. For example,

```
import package_name.*;
```

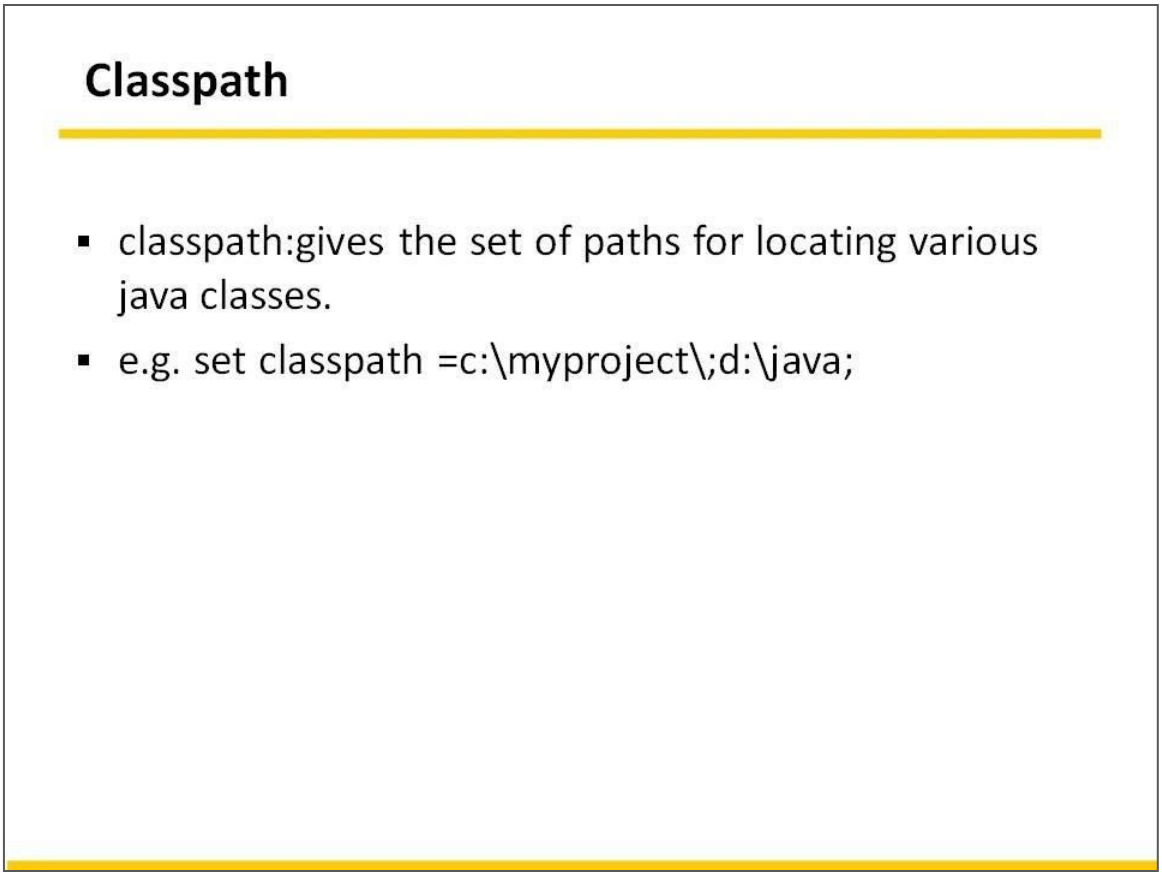
- `import` statement imports all the public classes within the package; it does not import sub packages.

Java Source files Structure

```
//part 1: (optional)
//package name
 package com.seedinfotech.project.employee

//part 2: (ZERO OR MORE)
//package used

import java.util.*;
import java.io.*;
import com.seedinfotech.project.employee.Date;

//Part 3: (ZERO OR MORE)
//Defination of classes and interfaces (in any order)

public class Employee{...}
class C1{...}
interface I1{...}
```

A typical Java file may start with keyword '`package`'. A Java source file contains only one package statement; however, it may contain any number of `import` statements.

After declaration of `package and import` statements, follows the actual class definition and implementation.



Classpath's specifies the path `javac` uses to look up classes needed to run `javac` or being referenced by other classes that are being compiled. It overrides the default or the CLASSPATH environment variable if it is set. Directories are separated by semi-colons. It is often useful for the directory containing the source files to be on the class path. Always include the system classes at the end of the path.

For example: `javac -classpath .;C:\users\dac\classes;C:\tools\java\classes`

Packages and Directories

Every package must be mapped to a subdirectory of the same name in the file system. Nested packages will be reflected as a hierarchy of directories in the file system. For example, the class files of a package `java.awt.image` must be stored under directory `java\awt\image`.

All these directories need not branch off the root directory; they can branch off from any directory named in the CLASSPATH variable.

`package_myPackage` is created under directory `c:\Files`

Then classpath is set using following command in DOS

```
set classpath=.;c:\Files\;
```

The `import` statement allows classes and interfaces defined in packages to be referred to solely by class names instead of the fully qualified names.

```
import packagename.*;
```

-d directory

This specifies the root directory of the class file hierarchy. In other words, this is essentially a destination directory for the compiled classes. For example, doing:

```
javac -d C:\users\dac\classes MyProgram.java
```

causes the class files for the classes in the MyProgram.java source file to be created in the directory `C:\users\dac\classes.`

Note that the `-d` and `-classpath` options have independent effects. The compiler reads only from the class path, and writes only to the destination directory. It is often useful for the destination directory to be on the class path. If the `-d` option is not specified, the source files should be stored in a directory hierarchy which reflects the package structure, so that the resulting class files can be easily located.

How Compiler Locates a File

Compiler locates a file in a following sequence:

1. First it checks in the current directory.
2. Then, compiler looks through all directories specified in the class path for the actual class file.

   OR

- It checks in the subdirectory that has the name of the imported package.

3. Then, looks for the file in one of the imported packages.
4. Finally, looks for file in `java.lang` package.
5. If compiler still does not locate the file, it gives an error.

Contents for batch file to set class path:

set JAVA_HOME=C:\Program Files\Java\jdk1.6.0_02<current jdk folder>

set path=%JAVA_HOME%\bin

set classpath=.;

Package Scope

Access Specified In class	Accessibility in subclass		Accessibility in other class	
	Same package	Different Package	Same Package	Different Package
private	χ	χ	χ	χ
public	√	√	√	√
protected	√	√	√	χ
default	√	χ	√	χ

In general, in any class that is designed with object oriented principles, the implementation is hidden. Given that packages can (and should) be used to group related types, however, some fields and methods may be required to be exposed to other classes in the same package while keeping them hidden from classes outside the package. Java provides access control modifiers to support this intermediate level of implementation hiding. By applying proper modifiers on a class fields and methods, the class's implementation can be hidden from classes outside the package while exposing the implementation to classes inside the package.

Access Specifiers

`private`

The private modifier restricts access of members of a class, so that no other classes can call member functions or directly access member variables.

`protected`

A protected member gives preferred access for subclasses in other packages. Member functions are sometimes created within a class for utility purposes (that is, to be used only within the class itself and not for general consumption). Declaring utility functions as protected allows them to be used not only by the class itself, but by its subclass as well,regardless of the package.

`public`

When public keyword is used it means that member declaration that immediately follows public is available to everyone. A public class can be accessed by any other class. Methods and member variables when declared public can be accessed by code from other classes.

`default`

Package access (denoted by no access control modifier keywords) - a field or method accessible to any type in the same package

`private` and `protected`

The `private` keyword grants exclusive access not to an object, but to a class. An object can access its private members, but so can any other object of the same class. For example, if a `MyDate` object has a reference to another `MyDate` object, the first `MyDate` can access the second `MyDate's` private members through that reference. This is true of both private variables and private methods, whether they are static or not.

Inside a package, the true meaning of the `protected` keyword is quite simple. To classes in the same package, protected access looks just like default access. Any class can access any protected member of another class declared in the same package.

static import

- Static import is a feature introduced in the Java programming language.
- This feature was introduced into the language in version 5.0.
- static members(fields and methods) defined in a class as public static .
- Allow to use static members and fields without specifying the class in which the field is defined.

When an application is developed, sometimes it is required to use static attributes or methods. For example, it is frequently required to print System.out.println `().out` is a static attribute of `System` class. To print any information or data `System.out.println()` has to be used. There is no other alternative.

So, the requirement is whenever some data is to be printed by `System.out.println()`, it should be printed by just saying `out.println()`.

Java 5 comes with a feature so that the static attributes as well as methods can be imported using `static` keyword after `import` keyword. Using static imports it is possible to refer static members directly by their names without qualifying them.

The syntax is:

```
import static packageName.ClassName.staticMethodName;
```

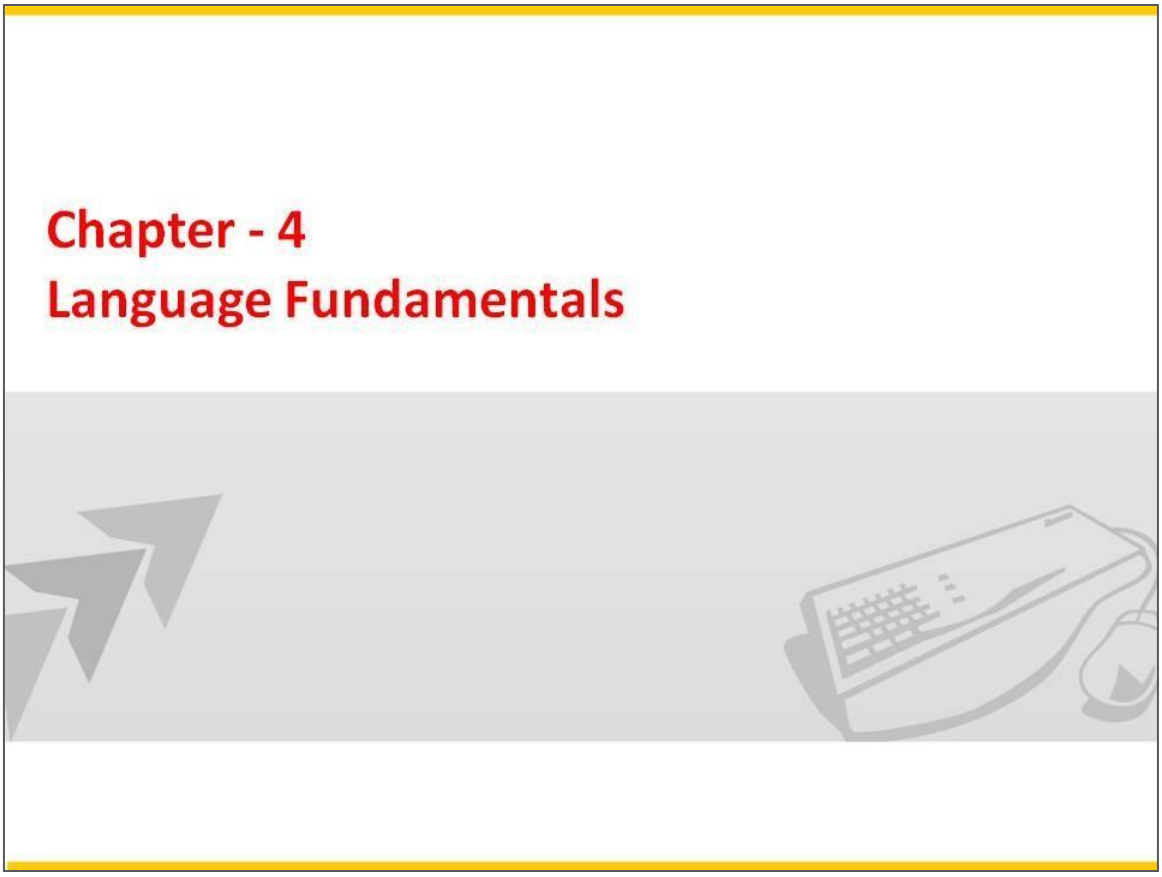
```
OR
import static package.ClassName.staticAttributeName;
OR
import static package.ClassName.*;
```

- Following is a rule of static import:
- If two or more classes are holding same static fields then static imports should not be used. For example, `Integer` class and `Long` class both hold `'Max-value'` field. So if both the classes are imported statically it will create naming conflicts for `'Max-value'` field.

Code Example

```
import static java.lang.System.out;
import static java.lang.Math.pow;
//OR
import static java.lang.Math.*;
public class StaticImportDemo
{
 public static void main(String args[]){
 out.println((PI*5)+" cm");
}
```

So, the requirement is whenever some data to be printed by `System.out.println`(),it should be printed by just saying `out.println()`.



Arrays, parameter passing, enumerated data type, structure are the fundamental topics of any language.

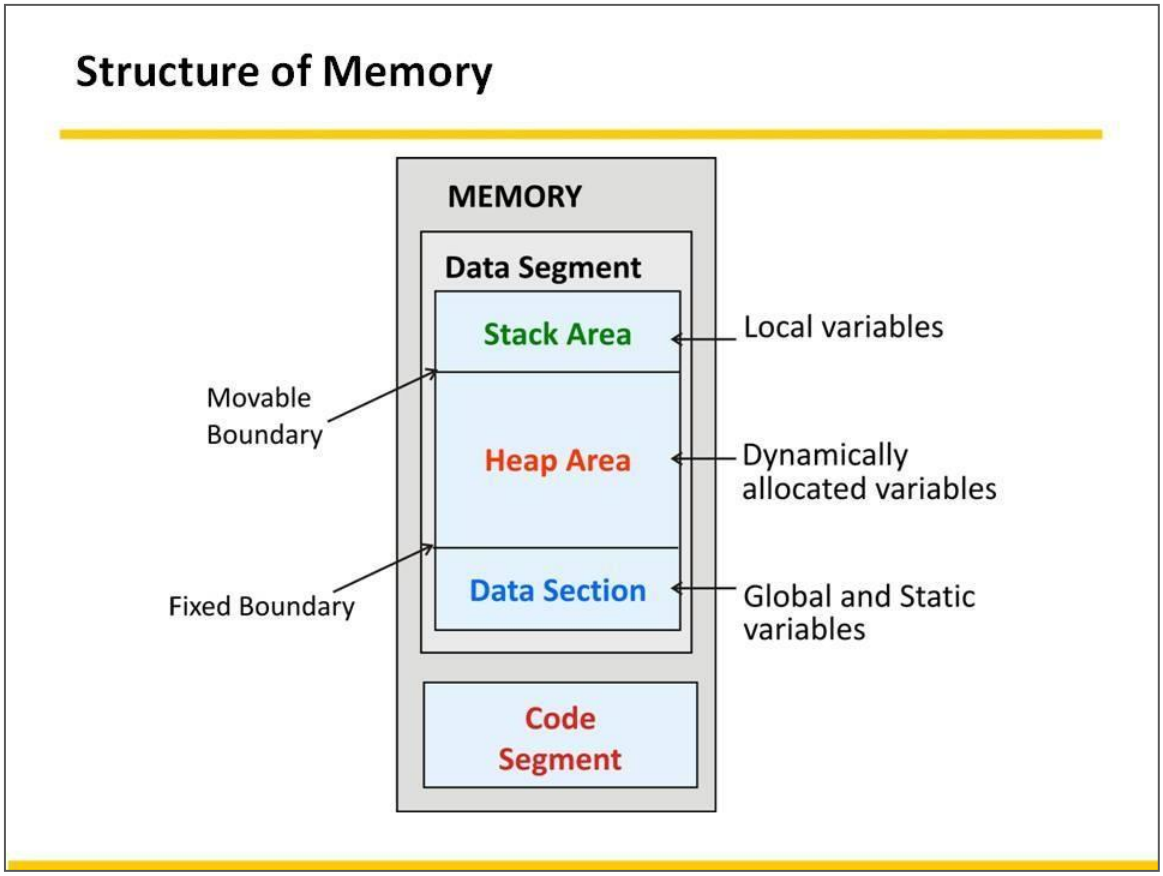
This chapter covers arrays (1D, 2D), passing parameters to methods, enhanced `for` loop, `enum` and containment.

Objectives

At the end of this chapter you will be able to

- Describe the process of object creation in context with memory structure.
- Describe how garbage collector works.
- Use and elaborate parameter passing mechanism in Java.
- Use 1D and 2D arrays in Java.
- Iterate a collection using enhanced for loop.
- State concept of containment and aggregation.

- Construct "has-a" relationship between classes e.g. `Employee` and `MyDate` class
- Define constants using final keyword.
- Use `enum`.



Before going ahead, it is essential to understand the structure of an exe file in the memory. This would help in understanding the part of memory where the variables are stored.

When the exe is loaded into memory, it is organized into two areas - code segment and data segment. The code segment is where the compiled code of the program resides. The data segment contains the data part of the program.

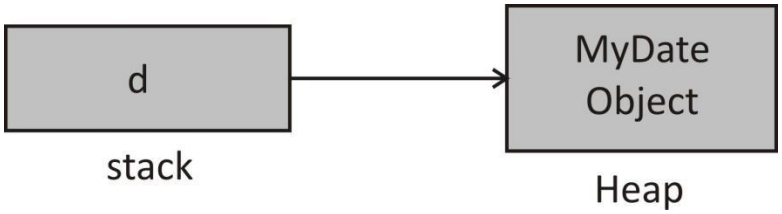
The stack section is where memory is allocated for local (automatic) variables within functions. Memory allocated for variables in this area contains garbage values.

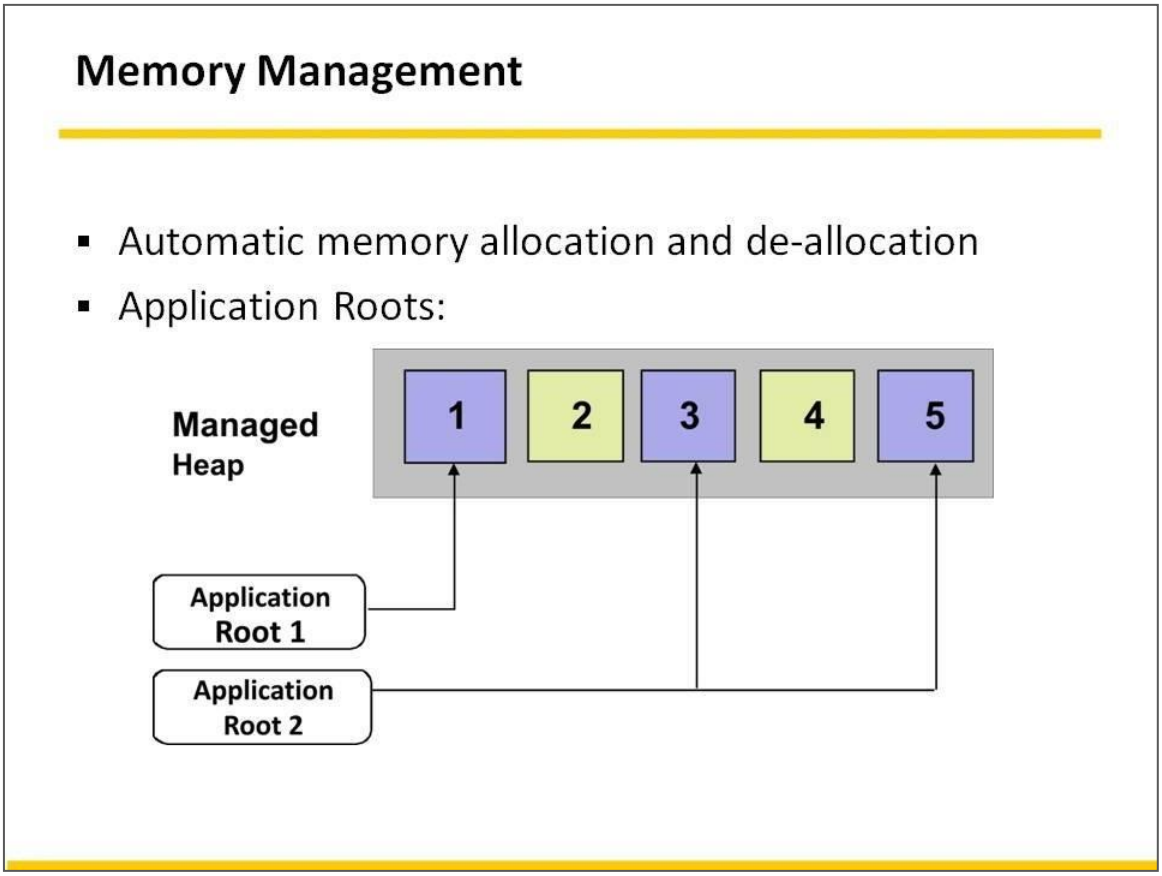
The heap section provides storage for variables that are dynamically allocated memory.

The data section is where the static variables are allocated memory.

Create an Object

```
public static void main(String args[])
  {
     MyDate  d;
     d=new MyDate();

          // OR
     MyDate d=new MyDate();
    }
```





In Java unlike C++, memory is allocated and freed automatically. The technique, which implements this approach, is called as Garbage Collection.

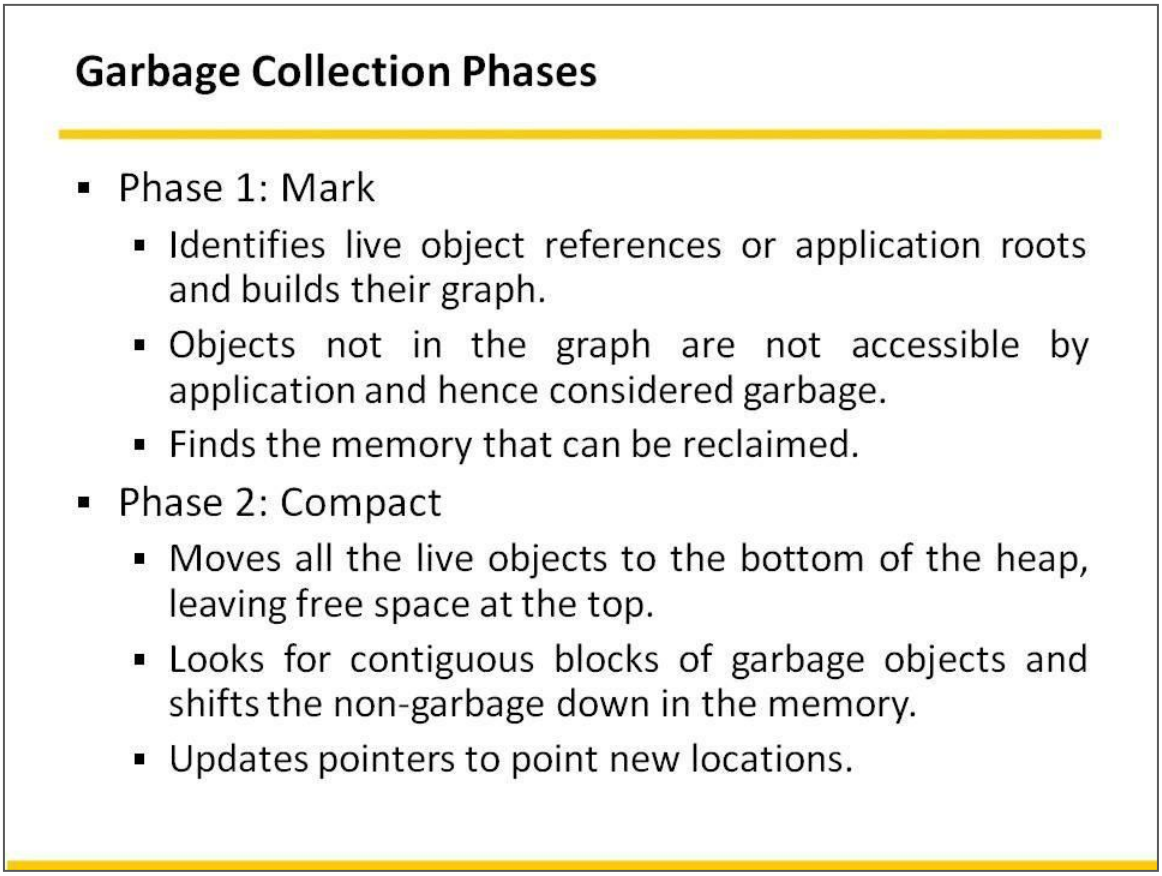
Garbage Collector is a program that runs frequently, steadily and on regular basis. If there are any dangling references then Garbage Collector reclaims that memory. Automatic memory management in Java is done by Garbage Collector (GC). GC is a component of JVM. It manages the allocation of memory and its release when not in use for any Java application.

Data is stored on stack and heap. Value types are stored on stack. Allocation and deallocation on the stack is automatic and safe. Reference types are stored on heap, known as managed heap. It is managed and controlled by JVM.

Memory management in Java is performed using reference tracing. Unused objects in memory are treated as garbage. Garbage collection task runs in the background (as a background thread) and reclaimed by Garbage Collector (GC). When an object is created, it is allocated on managed heap. Memory on heap is

not infinite. Allocation of memory for object continues as long as space on the heap is available. GC should collect unused objects in order to free memory.

The algorithm used by GC for memory management is highly optimized. The garbage collection algorithm runs periodically. The exact time is not known (as it is not revealed in specification by Sun). The memory is not freed immediately after a variable goes out of scope. It checks for the objects not used by the application and reclaims their memory. Such memory management is called as non-deterministic finalization.



Mark and Compact (Mark and Sweep) method is used by GC for garbage collection. It consists of two phases:

- Mark
- Sweep / Compact

Phase 1: Mark

In this phase, objects that are not reachable are marked for collection. While application running, with the help of `new` keyword, an object is created on the heap. Consider a scenario where there is a need to create an object but enough address space is not available on heap. Runtime throws an `OutOfMemoryException` in such case.

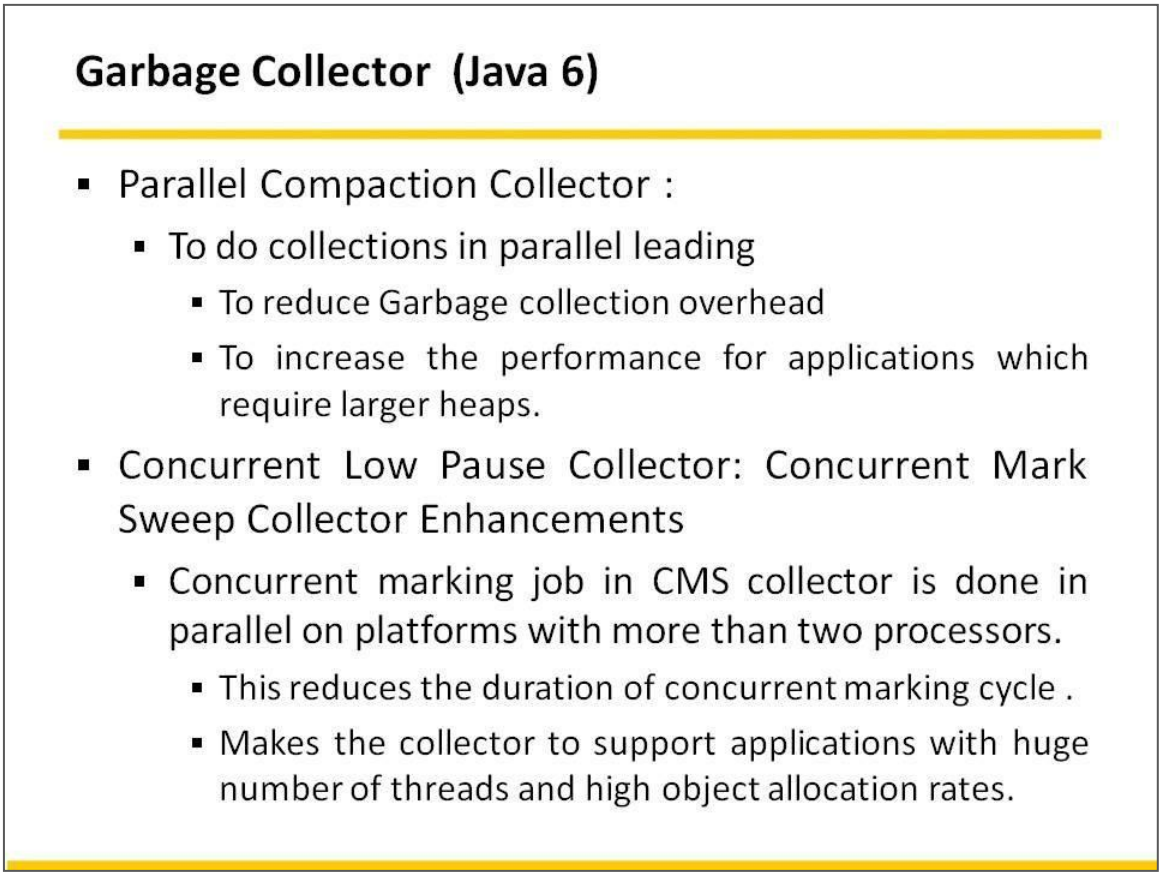
- GC identifies the live object references or active roots. The list of active roots is maintained by JIT compiler and JVM.
- At the start GC assumes that all objects in the heap are garbage. It walks through the roots and builds a graph of all objects that are reachable from the roots.

- Objects that are not in the graph mean they are not accessible by the application. Such objects are considered as garbage. These objects are collected by GC.

Phase 2: Compact

In this phase Non-garbage objects are compacted. The objects are allocated on heap contiguously. The compaction of object is performed by moving all live objects to the bottom of heap. This makes free space available at the top of heap.

- Linearly the garbage collector walks through the heap. It looks for contiguous blocks of garbage objects, which is free space.
- The non-garbage objects are shifted to bottom of memory. In this way, GC removes all of the gaps in the heap.
- As the objects are moved, the pointer that points to them becomes invalid. The roots of the application are modified by GC so that the pointers point to the new locations of the objects.
- The heap maintains a pointer. This pointer indicates where the next object is to be allocated within the heap. This pointer is positioned after the last non-garbage object to indicate the position where the next object can be added.

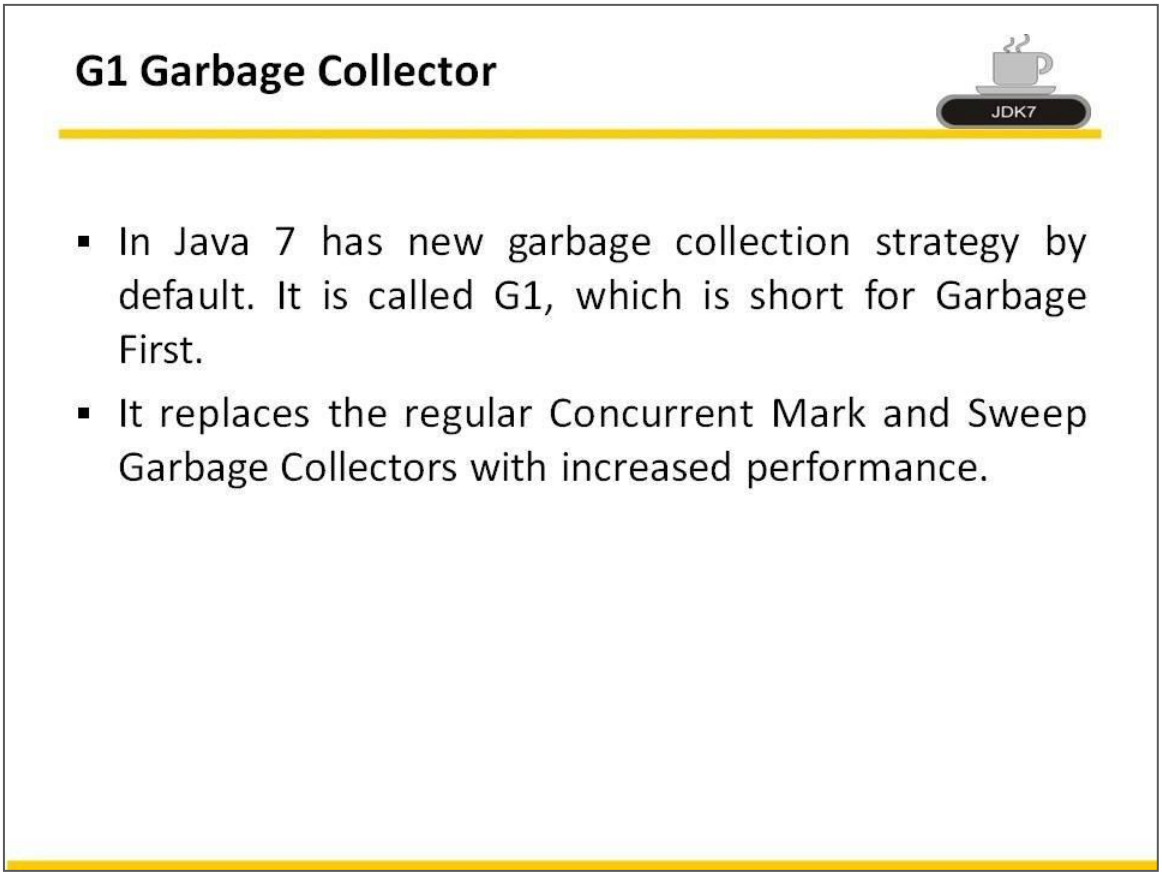


JAVA 6 gives you Garbage Collection Performance Enhancement to give better performance in application. Basically it comes with major changes:

Parallel Compaction Collector: To reduce the garbage collection overhead and better application performance with a large heap Parallel Compaction Collector concept is devised. It enables the collector to perform major collections in parallel. It is best suited to platforms with two or more processors or hardware threads.

Prior to Java 6, garbage was collected using a single thread; hence GC was needed to be called frequently, which resulted in performance loss.

Concurrent Low Pause Collector: To enhance the concurrent collection for the `System.gc()` and `Runtime.getRuntime().gc()` method instruction the Concurrent Mark Sweep (CMS) Collector is given by Java 6.Before to JAVA 6,when these two methods were called it stopped all application threads in order to garbage collect the heap objects. This process is usually lengthy, eats up a lot of time for applications with large heaps. This results in a performance hit for the application. This new feature enabling the collector to keep pauses as short as possible during heap collection.



The major changes in Java7 Garbage Collection are:

- Improvements on `Parallel Compaction Collector` to increase the performance of Garbage Collection process.
- Replacement to `concurrent Mark Sweep (CMS) collector`.
- The Java 7 Garbage collector is called as `'G1' i.e. 'Garbage-First'`.

G1 is considered as 'server centric' with following attributes:

- Use of all available CPU's and utilized the processing power.
- The performance and speed in GC.
- G1 has concurrency feature.
- To run Java threads to minimize the heap operations at stop pauses.

When objects are created, they are considered as young objects that have been in existence for some time (created before just sweep)are known as old objects.G1 is specifically targeted for multi-core CPUs as multiple processors can take care of mark/sweep, young/old object traversal simultaneously. This drastically improves the performance of the application as-a-whole.

Frequently memory allocation/deallocation may leave the memory area fragmented. G1 helps in eliminating the fragmentation as well.

Parameter Passing in Java

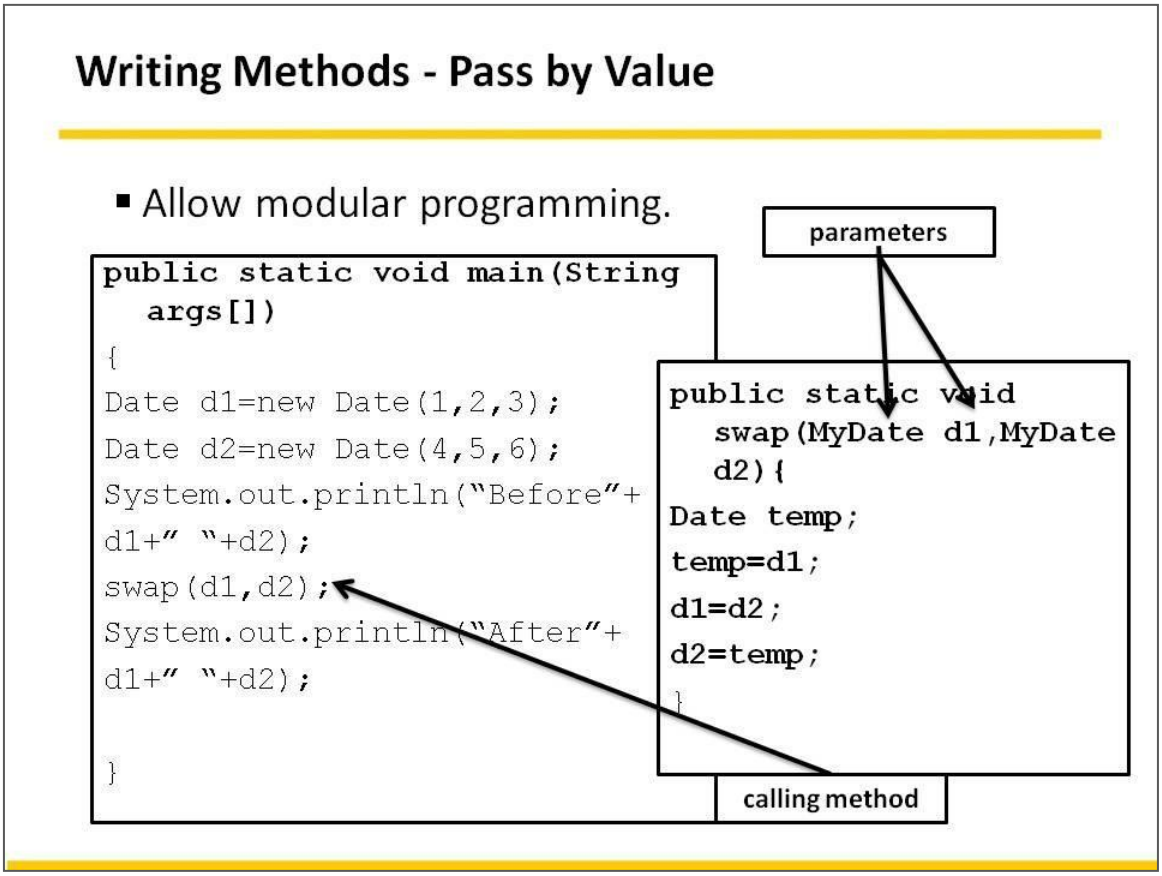
- In Java parameter passing is done "by value".
- While passing primitive data types a copy of the variable is created on the stack.
- While passing objects as parameters the references are passed by value(a copy of the reference "not the object" is created on stack).

All parameters to Java methods are passed by value. Values of parameter variables in a method are copies of the values the invoker specified by the invoker.

Code Example

```
//Code for pass by value
class PassByValue
{
    public static void main ( String [] args )
    {
        int x= 10;
        System.out.println("before" + x);
        doubleIt (x);
        System.out.println("after" + x);
    }
```

```
    public static void doubleIt (int arg )
    {
        arg *= 2;
    }
}
```

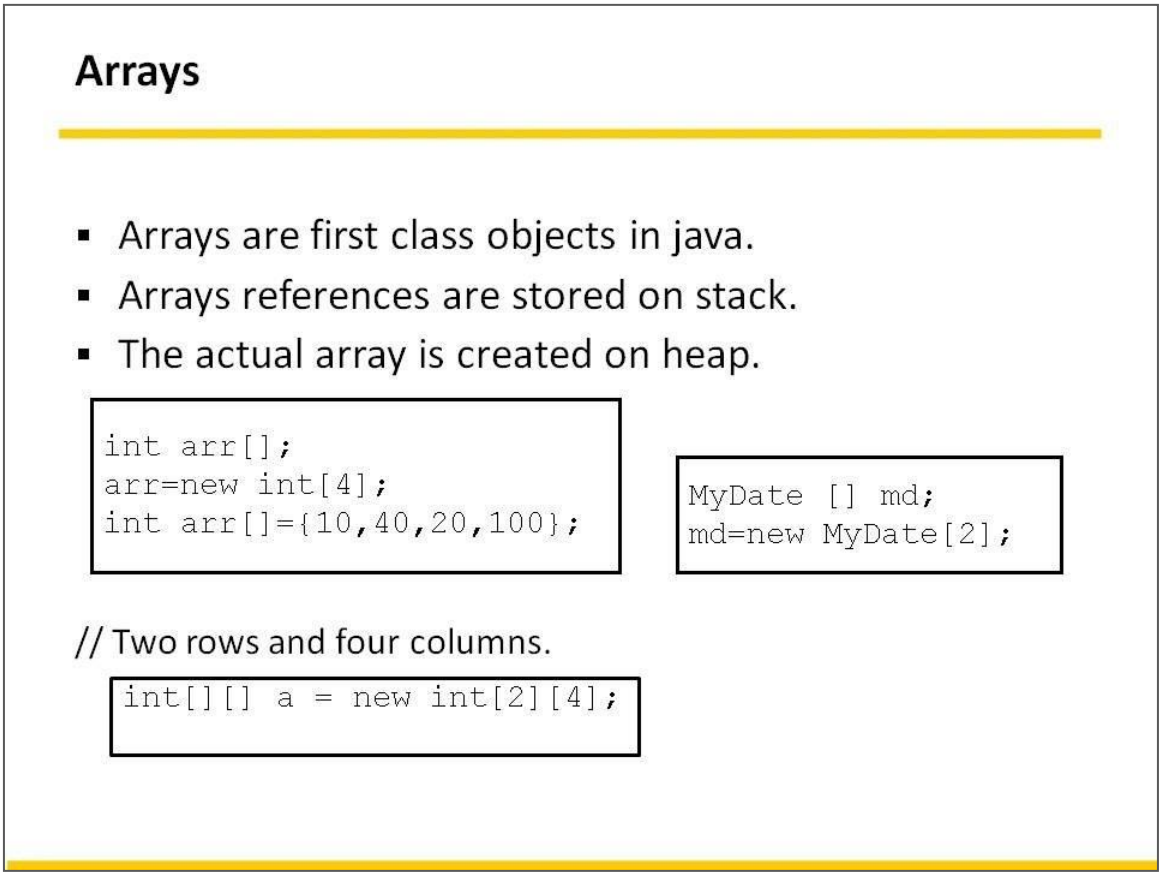


If such a function is written, the value of x before and after will be the same.

A simple swap function to interchange two `Date` objects `date1` and `date2` is written. This simple trick does not work and objects are not interchanged.

```
public static void swap (Date date1, Date date2)
{
    Date temp;
    temp = date1;
    date1 = date2;
    date2 = temp;          // does not work
}
```

When the parameter is an object reference, however, the object reference is what is passed by value, not the object itself. Thus the object a parameter refers to inside the method can be changed without affecting the reference that was passed. But if any fields of the object are changed or methods that change the object's states are invoked, the object is changed in the calling program also.



Arrays are first class objects in Java. What is meant by First class objects?, the objects need not be explicitly created using new as is mandatory for all other objects in Java.

Array references are stored on stack and actual array is stored on heap.

e.g.

```
int arr [];
arr = new int[10];
```

Here '`arr`' reference is stored on stack and the actual array is created on heap.

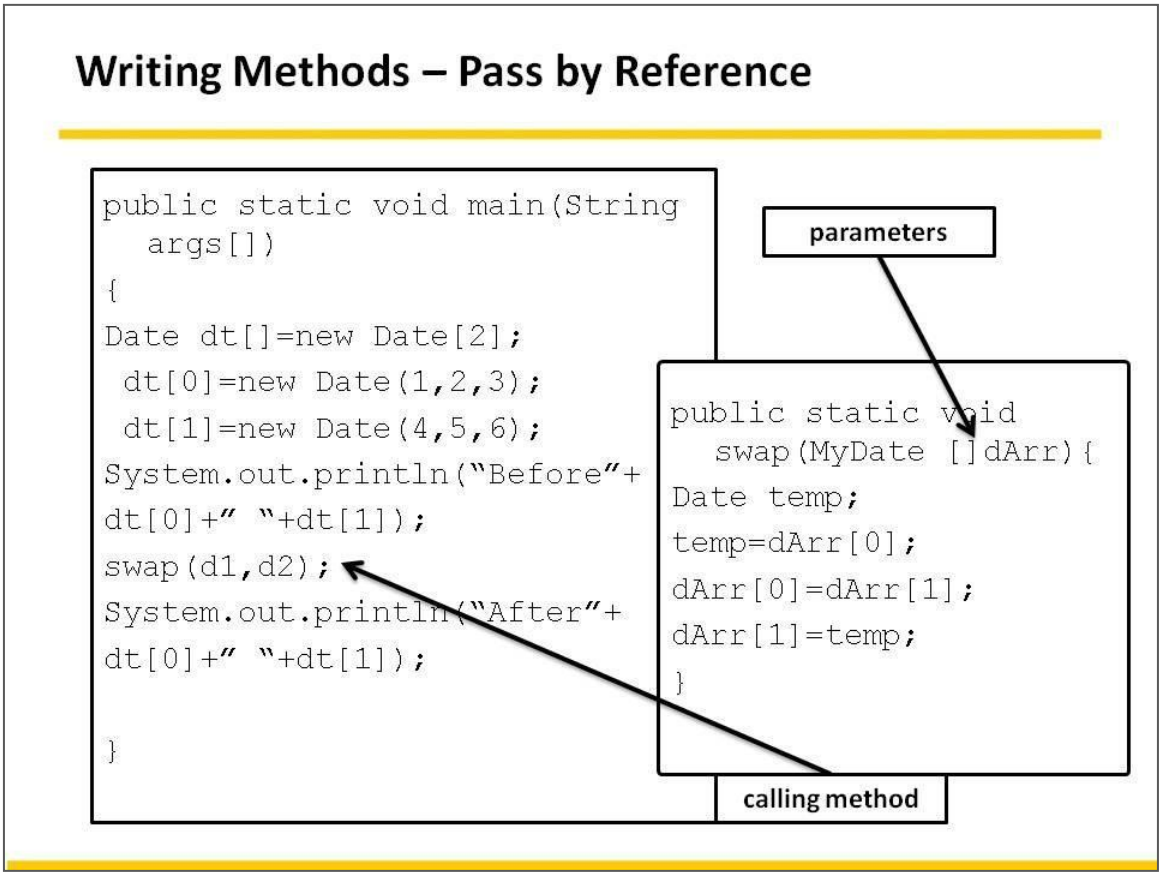
Multidimensional Array

Arrays can have more than one dimension. For example, the following declaration creates a two-dimensional array of three rows and two columns:

```
int [][] numbers = new int[][]{{1,2},{3,4},{5,6}};
```

Elements of 2D array can be printed using two for loops as shown below.

```
for(int i=0;i<3;i++)
 {
   for(int j=0;j<2;j++)
     {
        System.out.printf("\nElement
({%d},{%d})={%d}",i,j,numbers[i][j]);
      }
   }
```



The modification in swap helps us to overcome the parameter passing problem in the previous example.

```
public static void swap(MyDate [] dArr)
{
    MyDate temp = dArr[0];
    dArr[0] = dArr[1];
    dArr[1] = temp;
}
```

This works because even if array object references are passed as a value, the references are swapped inside the object array.

What is the difference between pass by value and pass by reference?

Enhanced For Loop(for each)

- for each loop is designed to cycle through a collection of objects, such as an array, in strictly sequential manner, from start to end.
- Traditional looping structure will work as follows:

```
int nums[]={1,2,3,4,5};
      int sum=0;
      for(int i=0;i<5;i++)
      {
        sum+=nums[i];
       }
```

- for each loop automates preceding loop. It eliminates loop counter, starting and ending value and manual indexing of array.

To iterate over the values generally programmer frequently refers to it as the “for loop”. The general form of `for` statement is as follows:

Syntax:

```
for (initialization; termination; increment) {
    statement(s)
}
```

This `for` statement block is divided into three parts:

- Initialization
- Condition or Termination
- Increment or Decrement

Refer the following code

```
class ForEachDemo
 {
```

```
   public static void main(String[] args)
   {
    for(int i=1; i<11; i++)
    {
     System.out.println("Count is: " + i);
    }
   }
 }
```

When a program has to iterate over Arrays or Collection then there is one form of for statement called enhanced for loop which is introduced in Java 5.

Syntax:

```
for(type var:collection)
```

Enhanced for loop has only two parameters that is type variable and collection type. Value is incremented internally and extra code is not required to write for it. To use enhance for loop variable must be of collection type.

Following are the rules to write enhanced for loop

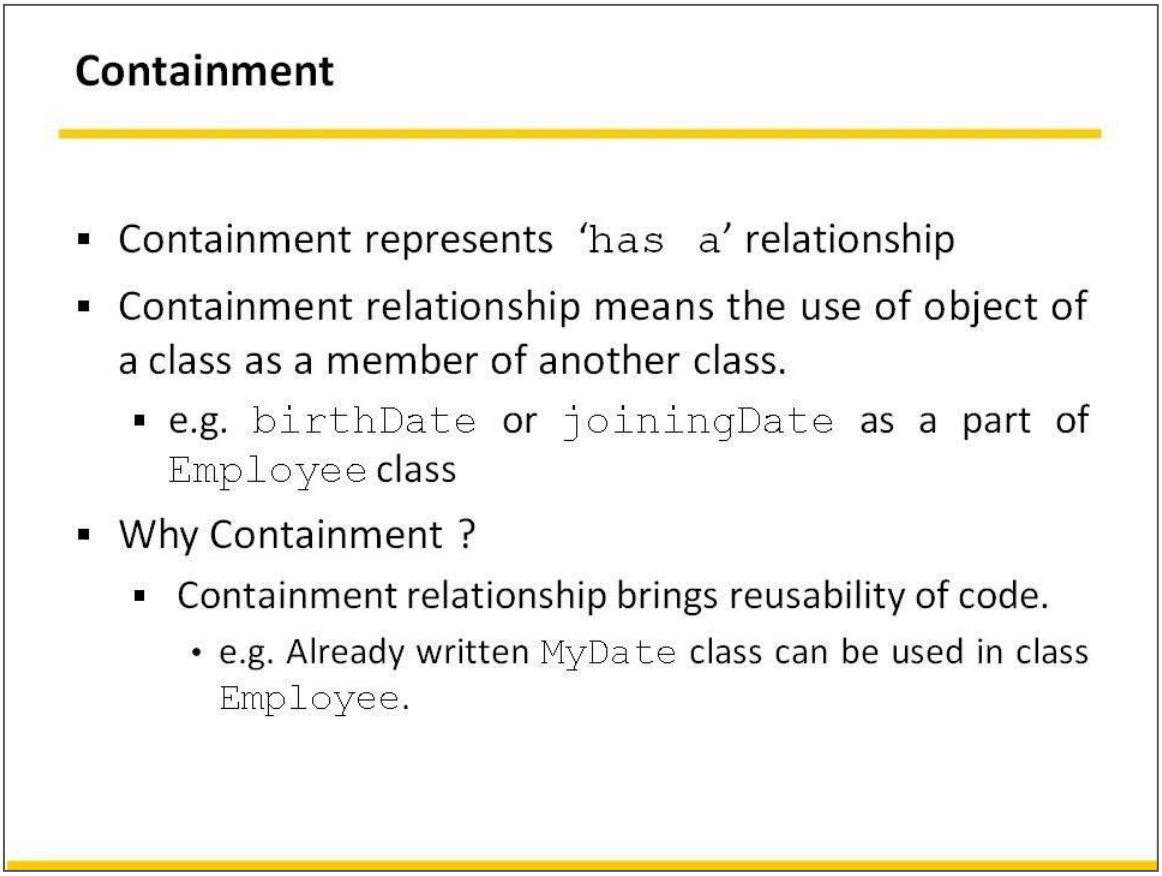
1. Cannot be used for filtering collection.
2. 'for each' loop gives read-only data and hides the iterator.
3. Cannot remove elements from collection.
4. Used to traverse over collection in forward direction.

Code Exmaple

```
class EnhancedForDemo
{
     public static void main(String[] args)
     {
          int[] numbers = {1,2,3,4,5,6,7,8,9,10};
          for (int item : numbers)
```

```
            {
              System.out.println ("Count is: " + item);
            }
      }
}
```

Enhanced for loop is read-only loop. You cannot use the for-each loop variable name to change the value of an item in the collection.



The basic need of containment is re-usability. A car manufacturing company designs and develops a car. Car has number of attributes like seat, tyre, engine etc. Manufacturer is not responsible for producing an engine/tyre. The company purchases these ready-made and embeds them into its cars. The company purchases these ready-made and embeds them into their car. The engine/tyre may turn have their own characteristics which the component is contained.

This is an example of re-usability by way of containing a ready-made component inside a class. This mechanism is also called containment.

Containment represents 'has-A' kind of relationship. The containment relationship brings reusability of code.

Code Example

To be precise in this example, if we consider Employee as class, then Employee has Birth Date or joining date. It means, Employee class can have say `MyDate` object as data member or instance variable. The container relationship brings reusability

of code. For example, If `Employee` is considered to be a class then `Employee` has birth date or joining date.

```
class Employee
{
 int empid;
 String empName;
 MyDate birthDate;
 public Employee ()
 {
  empid = 0;
  empName="";
  birthDate = new MyDate ();
 }
 public Employee(int i, String nm, MyDate dt)
 {
  empid = i;
  empname= nm;
  birthDate = dt;
  }
  public static void main (String args[])
  {
   Emp e1 = new Emp ();
   Emp e2 = new Emp (2, "abc", new MyDate (28, 6, 80));
  }
}
```

Final Variable

- Final Variable specifies that a variable is not modifiable; any attempt to do so flags an error.
- If a final variable is a reference to object ,it is the reference that must stay the same, not the object.
- Final variables must be initialized.
- e.g. private final float PI=3.14f;

If `final` keyword is attached to any variable then, it specifies that, a variable is not modifiable and any attempt to do so flag's an error.

If a `final` variable is a reference to an object, it is the reference that must stay the same, not the object.

Example

```
final MyDate bdate = new MyDate(1,1,80); //Assume that 1/1/80 is birthdate

bdate = new MyDate(1,2,80);   //illegal, since 'bdate' tries to change.

//Whereas,

//final variables must be initialized.

e.g.

private final float PI = 3.14f;
```

`final` variables are same as const's in C++ & #defined constants in C.

Enums

- An `enum` type is a type whose fields consist of a fixed set of constants.
  - Created using keyword `enum`.
  - It is a kind of class definition.
  - The possible values are listed in curly braces, separated by commas. By convention the value names are in upper case.
- `java.lang.Enum` class is an abstract class.

```
enum CoffeeSize
{
  HUGE,OVERWHELMING, BIG
}
```

The purpose of enumerated types is to enhance the readability of a program. By default, the constants in the enumeration list are assigned the integer values 0, 1, 2, and so on.

Enumerations are used to create a set of symbolic names that map to numerical values. By using the `enum` keyword a new "type" can be created and its can be specified. In general, enumerations may be defined as

```
enum tag {member1, member2… member n};
```

Once an enumeration is defined, variables of that type may be defined.

```
tag var1, var2, .  .  .varn;
```

In the example mentioned above, `WeekDays` is an enumeration. MONDAY, TUESDAY, and so on are the members of the enumeration which are called enumeration constants.

Enums specifies list of constants. In Java enums are `public static final Object` type. Enums are declared as default subclass of `java.lang.Enum`

class. Methods, variables, and constructors can be added to an enum, as they behave just like a normal class.

Enum API

Following are some important methods of `Enum` class

Method/Constructor	Description
`Enum(String name, int ordinal)`	Sole constructor.
`Object clone()`	Throws CloneNotSupportedException.
`int compareTo(E o)`	Compares this enum with the specified object for order.
`Class<E> getDeclaringClass()`	Returns the Class object corresponding to this enum constant's enum type.
`String name()`	Returns the name of this enum constant, exactly as declared in its enum declaration.
`int ordinal()`	Returns the ordinal of this enumeration constant (its position in its enum declaration, where the initial constant is assigned an ordinal of zero).
`static <T extends Enum<T>> T valueOf(Class<T> enumType, String name)`	Returns the enum constant of the specified enum type with the specified name.
`int compareTo(E o)`	use to compare ordinal values of two constants. If two ordinals are same it will return zero. If 1st ordinal is greater than 2nd it will return + ve value else it will return –ve value.

Following are the rules to write enums:

1. They can be declared inside or outside class but not within any method.
2. They are by default `final`.
3. If declared inside class they are `static`.
4. Their objects cannot be created explicitly as `new` keyword is not available.
5. Their constructor has to be private.
6. Enums cannot be `private, protected, static`, and `abstract`.
7. Enums constants are implicitly `public, static` and `final`.
8. The order of appearance of `enum` constants is called their NATURAL ORDER.

Code Example

```
//CoffeeSize.java
//Enum Declaration
package com.project.coffee;

enum CoffeeSize
{
  HUGE,OVERWHELMING, BIG
}
//OR
 package com.project.coffee;

 enum CoffeeSize
  {
    BIG(20),HUGE(25),OVERWHELMING(50) ;
    private int ml;
    CoffeeSize(int m)
    {
    ml=m;
```

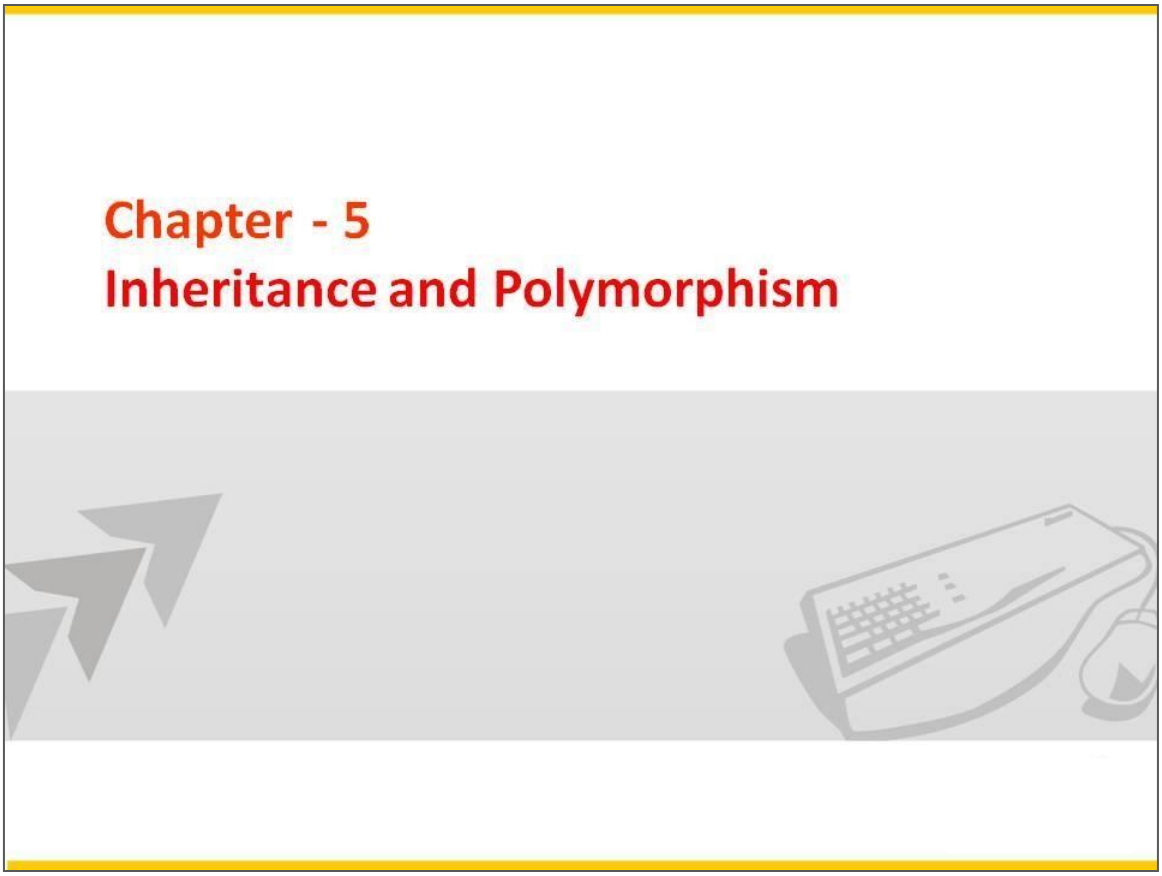
```
    }
    int getMl()
    {
    return ml;
    }
  }
//We can declare Constructor and Methods in side enum
```

```
//Coffee.java
//Declare size of type CoffeeSize
package com.project.coffee;

public class Coffee
{
 CoffeeSize size;
}
```

```
//CoffeeTest.java
//Test Class
package com.project.coffee;

public class CoffeeTest
{
 public static void main(String[] args)
 {
   Coffee drink=new Coffee();
   drink.size=CoffeeSize.BIG;
```

```
    drink.size=CoffeeSize.HUGE;
    System.out.println(drink.size);
    System.out.println("coffe is "+"
"+drink.size.getMl()+" "+"ml");
  }

}
```

Enumerations enhance the development and readability of the code.



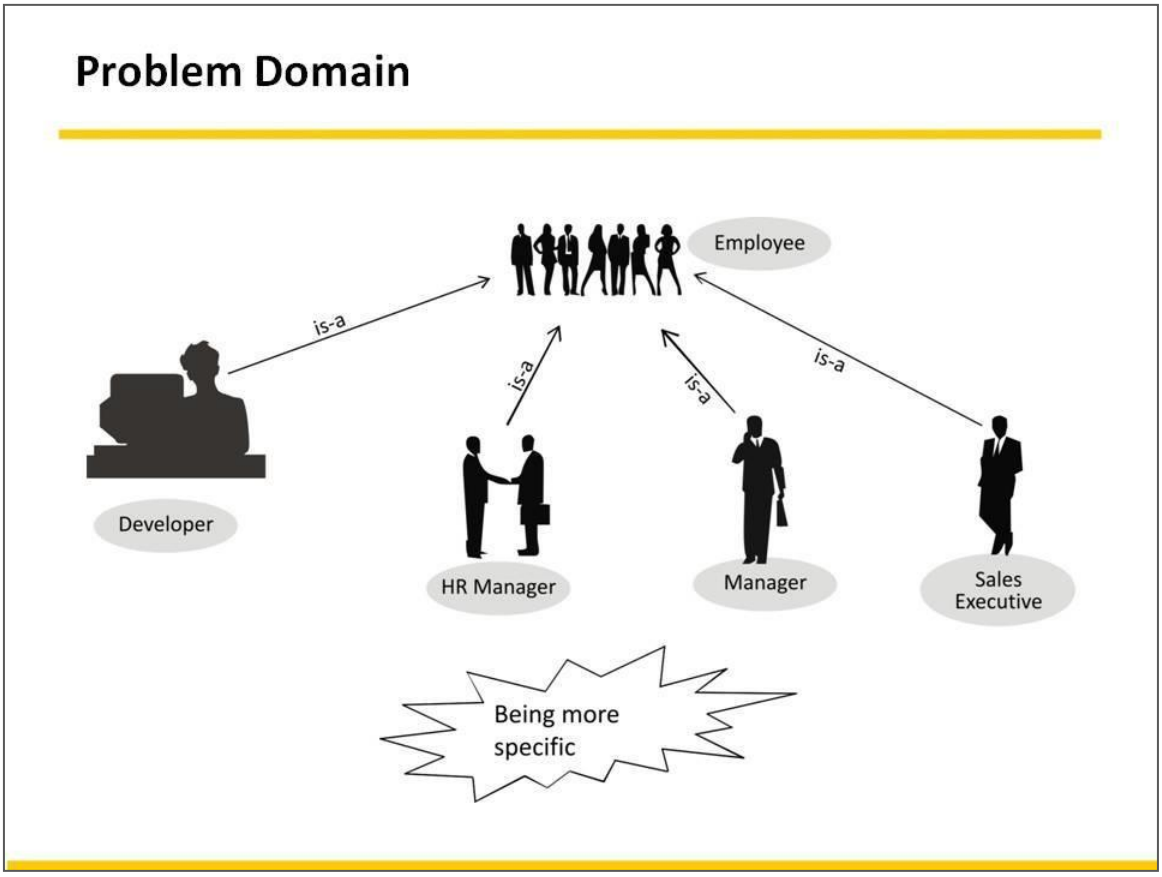
This chapter covers concept of inheritance and polymorphism. It covers `super`, `private`, `protected` keywords. It also covers the concept of covariant return type.

Objectives:

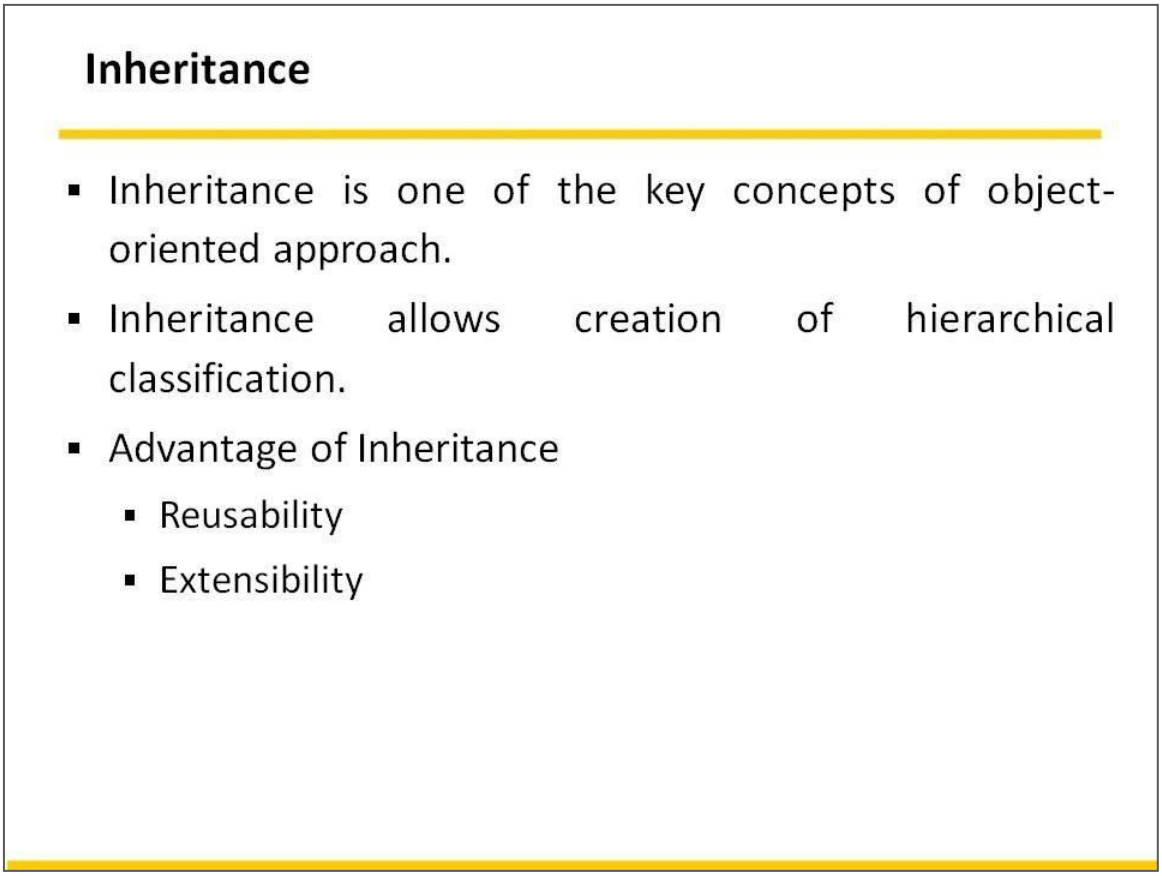
At the end of this chapter you will be able to

- Describe inheritance and its types.
- Construct "is-a" relationship using `Employee - Manager- SalesPerson` classes.
- Use super and protected keywords in building the hierarchy.
- Define polymorphism and state its use.
- Elaborate the difference between method overloading and overriding.
- `Instanceof` operator.
- Use covariant return type
- Stop inheritance by using final class.

- Stop overriding by final method.
- Use methods from Object class: `toString`, `equals`, `finalize`, `hashCode`.
- Create object hierarchy with abstract class.
- Create abstract method.
- Use abstract classes through overloading.
- State need for interface.
- Distinguish between `interface` and `abstract` class.
- Distinguish between `implements` and `extends` keywords.
- Create `Printable` interface to implement printing functionality in two unrelated object hierarchies like `Shape` and `Employee`.
- State concept of marker interface.
- Create copies of object using `Cloneable` interface.



Employees can be of different types as can be seen on the slide. There can be a developer, an HR manager, a sales executive, and so on. Each one may belong to a different problem domain but the basic characteristics of an employee are common to all. It can be said that, employee type can be made as the base and the remaining types can be derived from it.



Inheritance is one of the strongest features of object-oriented programming. It not only helps to reuse the old code but aids in extending the software. Inheritance is similar to parent-child relationship. In inheritance, each child has an 'is-a' relationship with its parent. For example, an apple 'is-a' fruit. A triangle 'is-a' shape. Red 'is-a' color.

Inheritance is a mechanism by which a new class is derived from an existing one. The new class, which is being created, is called as derived class and the class from which it is derived is called as base class. With the help of inheritance a hierarchy of classes can be created.

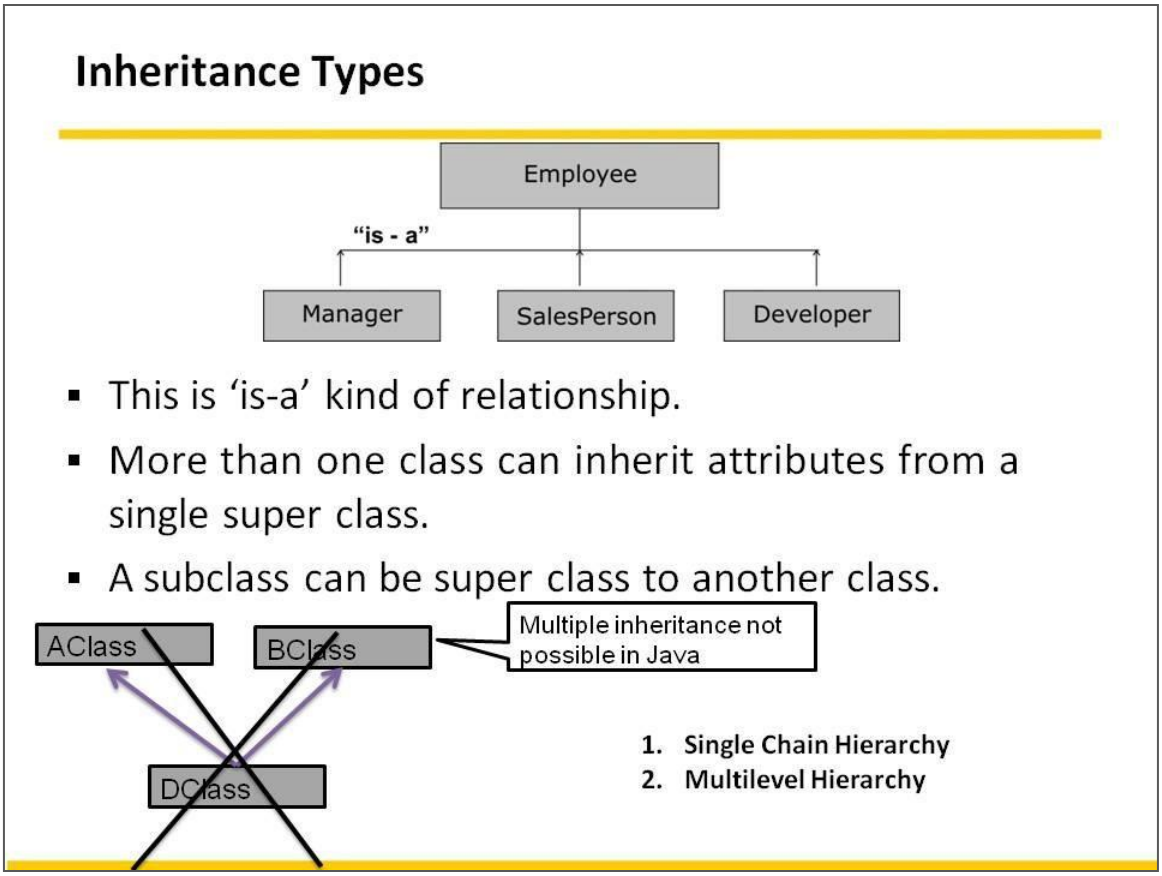
Features of Inheritance

- Object-oriented programming extends abstract data types to allow for type/subtype relationships. It is achieved through inheritance.
- To inherit means to receive properties of an already existing class.
- When an existing class is inherited, the derived class inherits all data members as well as the member methods from base class. But not all of them can be

accessed by the member methods of the derived class. The accessibility of base class members in the derived class depends on their access specifiers.

Advantages of Inheritance

- Reusability: Once a class is written and tested, it can be further used for creating new classes. These derived classes not only inherit the features of their base class, but also have their own individualistic features. This means that if the derived class wants to use its base class properties it can do so because these properties are also available to the derived class by the virtue of inheritance.
- Extensibility: It is the mechanism of being able to derive classes from existing classes. This provides the extensibility of adding and removing classes in a hierarchy as and when required. Any changes to data or operations contained within a base class are immediately inherited by all derived classes that have been inherited from the base classes.



In the example shown above, `Employee` is designed as the base class since it is the most general category of employees and has all the common attributes of all employees. `Manager`, `SalesPerson`, `Developer` classes are derived from `Employee` class.

A derived class can be used as a base class to derive other classes. For example, another class `SalesManager` can be derived from the `Manager` class.

"is - a" relationship is different than "has - a" relationship where one object is a part of another object. This is known as containment. For example, the employee type can have an address object as its data member. In this example, the relation between employee and the address will be "employee has an address".

Multiple inheritance is not possible in JAVA. If you know C++, recollect the problems associated with multiple inheritance.

Example of Inheritance

```
class Employee
 { . . .
     public double calculateSalary()
      {return basic_sal + hra + da ; }
 }
class Manager extends Employee
{ public double calculateIncentives()
  {
   //code to calculate incentives
  return incentives ;
   }
}
```

```
public static void main(string[] args)
{
 Manager mngr = new Manager();
 double inc = mngr.calculateIncentives();
 double Sal = mngr.calculateSalary();
 System.out.println("Incentives
            "+inc+".....SALARY="+Sal );
}
```

In the example shown above, `Employee` is designed as the base class since it is the most general category of employees and has all the common attributes of all employees. `Manager`, `SalesPerson`, `Developer` classes are derived from `Employee` class.

A derived class can be used as a base class to derive other classes. For example, another class `SalesManager` can be derived from the `Manager` class.

"is - a" relationship is different than "has - a" relationship where one object is a part of another object. This is known as containment. For example, the employee type can have an address object as its data member. In this example, the relation between employee and the address will be "employee has an address".

Inheritance Syntax

```
class super_class name
{
        // body of super class
}
```

```
class sub_class name extends super_class
{
        // body of derived class
}
```

The base class `Employee` defines a method `calculateSalary()` for calculating the salary. Since a `Manager` is a specialized employee, `Manager` class will inherit this method. This method is a public method and so can be invoked by object of `Manager` class. The `Manager` class can also define a new method `calculateIncentives()` specific to the class.

Accessibility of base class members in derived class depends on the type of modifier used to define them.

Access Specifier In Super class	Accessibility Within Sub Class	Accessibility Not related class
private	**X**	**X**
public	✓	✓
protected	✓	**X**
default	✓	✓

All members of super class are inherited by sub class. The accessibility of these members depends on their access specifier.

- `private` members are not accessible even by the immediate derived class within or outside (Not related) class.
- Other access specifiers like, `public`, `protected` are accessible to immediate sub classes within or outside the package.

You can use `super` keyword to access base class members to access in derived class.

Derived class Constructor

- Constructors are called in the sequence of super→sub
- When Manager is created the sequence in which constructors get invoked is
  - Employee →Manager

```
class Employee
 {
    Employee(…){
        //constructor here
        . . .
      }
 }
class Manager extends Employee
{
    Manager(…){
       super(…);
      }
}
```

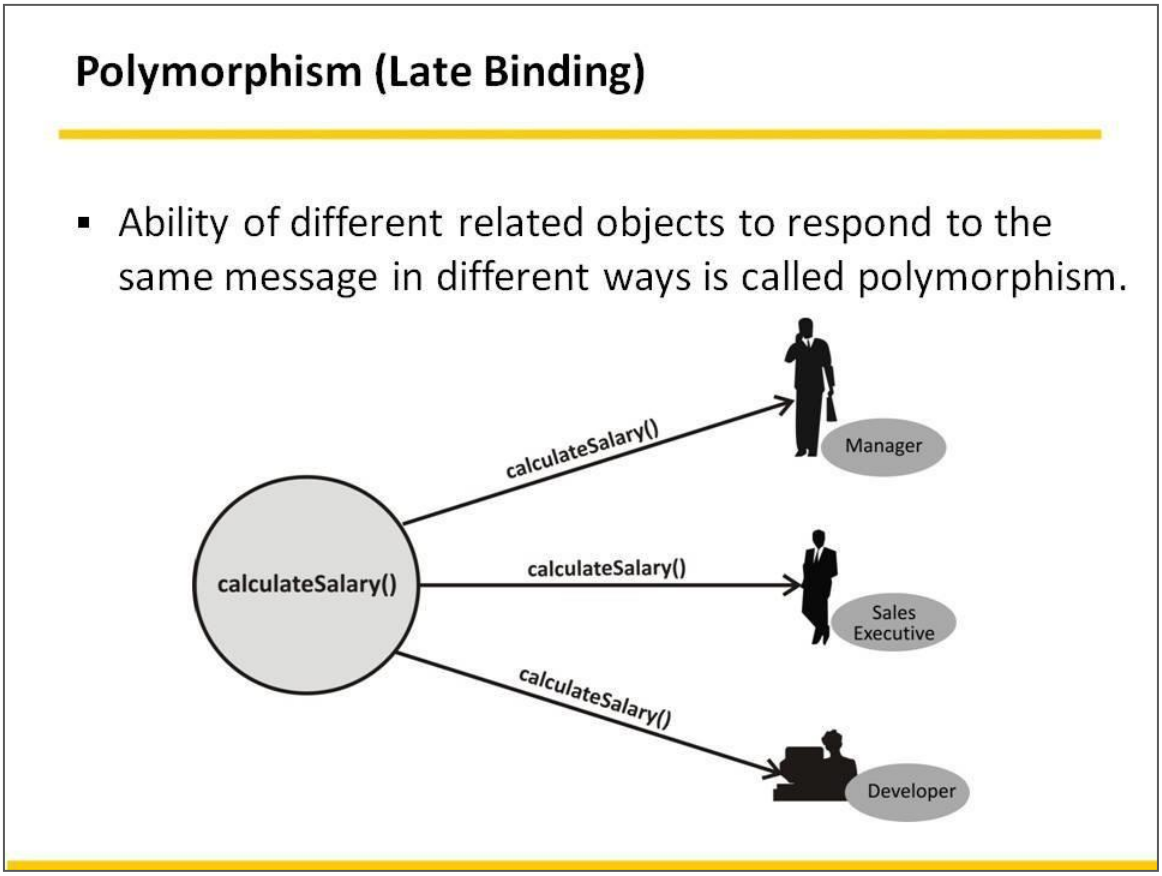
When a class is derived from another class, the member initializer list is the best way to initialize the super class data members. Member initialization of super class is achieved in derived class constructor using `super` keyword. `super` keyword is used to call any method of base class from the derived class.

Constructor of derived class first invokes the constructor of super class. If explicit call to parameterized constructor is not given, it calls the base class no-argument constructor. Then the code inside derived class constructor gets executed.

super class constructor should be explicitly called using `super` keyword otherwise as per the order of constructor call (super to sub), default constructor is called.

The super keyword is available in all non-static methods of an extended class. Using super is the only case where the type of the reference governs the selection of method implementation to be used. An invocation of super method always uses the super class's implementation of method, not any other overridden implementation. One can also access protected members of the super class using

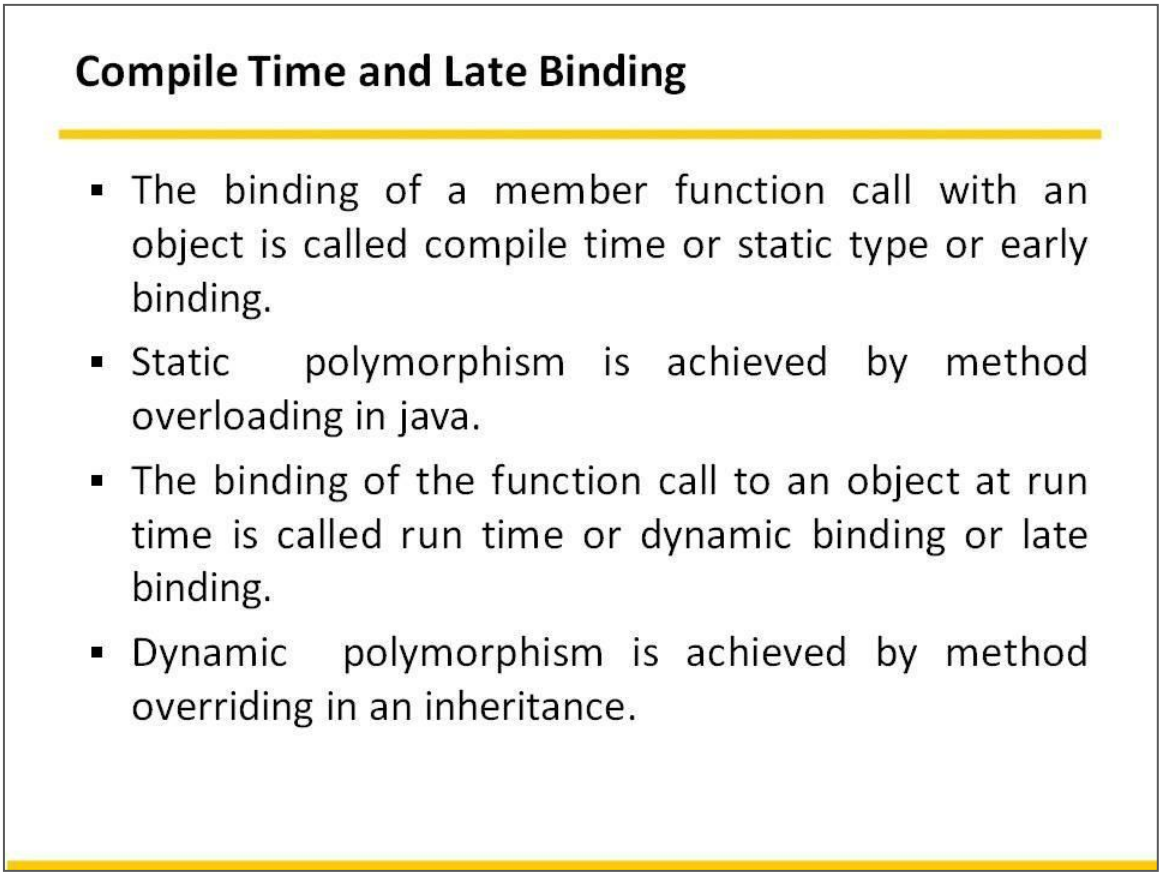
super keyword. Use of `this()` and `super()` must follow a particular sequence. If the `this()` is invoked before `super()` compiler tries to call the super class's default constructor, runs the remaining code till it encounters `super` once again. Since the object is being reconstructed, compiler gives an error.



The word polymorphism is combined using two words. “Poly” which means many and “morph” which means form. Ability to take more than one form is called polymorphism. In other words, one command may invoke different implementation for different related objects.

Polymorphism plays an important role in allowing different objects to share the same external interface although the implementation may be different. Thus, it is seen that polymorphism helps to design extensible software components as we can add new objects to the design without rewriting existing methods.

As shown in the figure above; method `calculateSalary()` can calculate salary for various objects like `Manager`, `SalesPerson` and `Developer`. But the ways in which all these objects would calculate the salary would be different.



Binding is the process of associating a function call to an object. There are two types of binding namely early binding and late binding.

Early Binding

When binding occurs at compile time, it is known as early binding. The argument passed to the method decides the method to be invoked at compile time. Early binding is achieved using method overloading or something called it as compile-time polymorphism.

Late Binding

When the binding process occurs at run time, it is called late binding. The appropriate method is invoked at run time by checking type of an object. Late binding is achieved using method overriding or is identified run-time polymorphism.

Method Overriding

- Polymorphism is achieved using override methods and inheritance.

```
class Employee
{
 public double calculateSalary()
 {return basic_sal + hra + da; }
}
class Manager extends Employee
{  public  double calculateSalary()
   {return (basic_sal + hra + da +
    allowances);}
}
public  static void main(string[] args){
   Manager mngr = new Manager();
System.out.println(mngr.calculateSalary());
   Employee empl = new Employee();
System.out.println(empl.calculateSalary());
}
```

Polymorphism provides a way for the derived class to give its own definition of a method that has already been defined by the base class. This process is called method overriding.

In method overriding, methods have same names, same signatures, same return types but are in different scopes (classes) in the hierarchy. There is a possibility that a method in the super class could be overridden in the derived class. The overridden method is required to have specific implementation in derived class.

Method overriding enables an object to resolve method invocation at runtime. It is called as late binding behavior of the object.

Rule for Method Overriding

- Overriding is directly related to sub classing.
- Overriding is required to modify the behavior of a super class's method to suit the new sub class.
- Overriding methods should have argument list of identical type and order- else they are simply treated as overloaded methods.

- Return types of overridden methods should be of same type; else compiler generates error.

Consider the hierarchy

`Employee` <- `-------- Manager`<--------`SalesManager`

All these classes contain `calculateSalary()` method which is overridden.

Method Overloading vs Overriding

	Overloading	Overriding
Scope	In the same class	In the inherited classes
Purpose	Handy for program design as different method names need not be remembered	Message is same but its implementation needs to be specific to the derived class
Signature of methods	Different for each method overloaded	Has to be same in derived class as in base class
Return Type	Can be same or different as it is not considered	Return type also needs to be same

What is the difference between method overloading and method overriding?

Dynamic data type - Method Selection

- Base class reference can refer to derived class object.
- Dynamic data type of base class reference governs method invocation.

```
static void displaySalary(Employee emp)
{
  emp.calculateSalary();
}
public static void main(string[]args)
{
  Manager man = new Manager();
  displaySalary(man);
}
```

emp has static data type Employee and dynamic data type Manager.

Inheritance implies 'is-a' relationship from the child to a parent. In an 'is-a' relationship, a super-class reference can refer to a derived object without needing a cast. This is termed as up-casting. This is the reason why an inheritance hierarchy diagram uses an up-arrow. But the derived class reference cannot refer to base class object.

In the hierarchy of `Employee` class, `Manager` is a sub class.

```
Employee emp = new Manager();
emp.calculateSalary();
```

`emp` is called as a super class reference referring to a sub class object. Static type of `emp` is '`Employee`'. But when the `calculateSalary()` method is called using base class reference '`emp`', method overridden in derived class i.e. '`Manager`' is called. The reason is that the dynamic data type of the base class governs method invocation. In the example mentioned, dynamic data type of '`emp`' is '`Manager`'.

```
static void displaySalary(Employee emp)
{
  emp.calculateSalary();
}
public static void main(String[]args)
{
  Manager man = new Manager();
  displaySalary(man);
}
```

This is true only for overridden methods. Binding of the method to an object takes place at run time. So this is called dynamic binding or late binding or polymorphism. So instead of static type of base class reference, actual object it refers to at run time decide the method invocation.

Specific methods of the derived class cannot be called using base class reference.

```
emp.specificMethodOfManager(); // compile-time error
```

Since the method is not virtual, compile-time binding takes place. In this case, static type of object reference is considered by the compiler to resolve the call. Since this method is not present in the base class, error occurs.

Explicitly casting is required as it is a downcast.

```
Manager man = (Manager)emp;
man.specificMethodOfManager();
```

Explicit casting is possible only if the two objects are in the same hierarchy.

instanceof operator

- Why `instanceof` operator?
  - Identifying dynamic data type of an Object.
  - To access objects polymorphically.
- Java runtime keeps track of the class to which each object belongs.
- This information is used by java to select the correct methods to execute at run time.

```
Employee emp = new Manager();
if(emp instanceof Manager)
 {
    // call to a method
 }
```

To identify dynamic data type of an object we required `instanceof` operator. Once the object is obtained the methods can be accessed.

Code Example

```
class Parent
{
 public Parent()
   {
   }
}
class Child extends Parent
{
 public Child()
  {
```

```
    super();
  }
 }
 public class TestClass
 {
   public static void main(String args[])
   {
    Parent c=new Child();
     if(c instanceof Child)
      {
        System.out.println("true");
      }
   }
}
```

Covariant Return Types

- In method overriding, overridden method in subclass should have signature same as that of its super class.
- In Java 5 return type in the overridden method can be changed as long as new return type is subtype of declared type.

```
class Employee{
     public Employee calSalary()
        {
          return new Employee();
        }
}
class Manager extends Employee{
      public Manager calSalary()
        {
           return new Manager();
        }
}
```

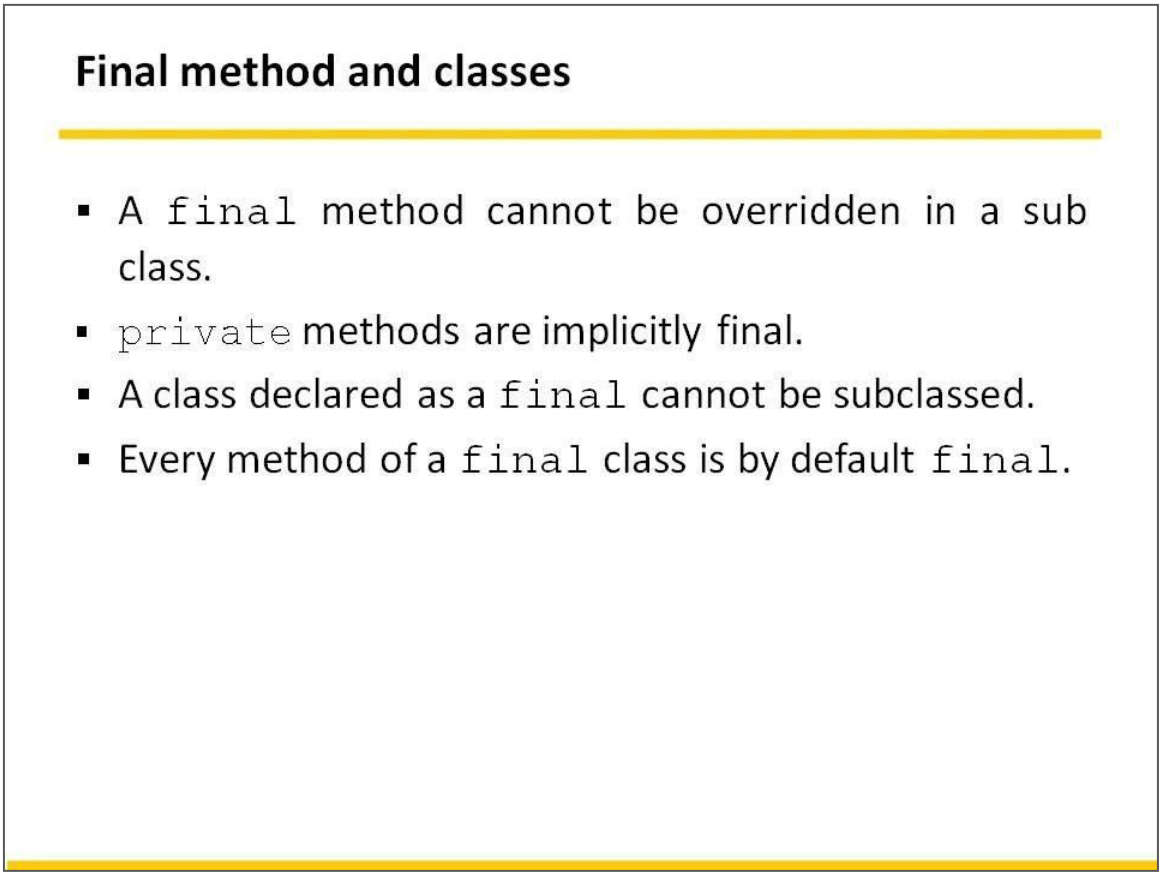
When a method in sub-class is overridden then the basic rule is that the method should have same signature as that of super class method. Otherwise that method is considered as different method.

Java 5 comes with new feature called 'covariant return type' which allows you to change the return type in the overridden method. Return type is a narrower type of declare type in subclass. Return type must be subtype of declared type.

Code Example

```
class Parent
{
     public Number getObject()
     {
          return new Number();
     }
}
```

```
class Child extends Parent
{
	public Integer getObject()
	{
		return new Integer();
	}
}
```



It is the other side of the design coin. The main purpose of a final class is to take away the inheritance feature from the user so that a class cannot be derived from a final class. In Java, `String` class is final class which is in `java.lang` package.

'`final`' modifier is used to prevent a class from being inherited and a method from being overridden in derived class.

'`private`' methods are implicitly final. A class declared as final cannot be sub-classed. Every method of a final class is by default final. Since final's method definition can never change, compiler can optimize the program by removing calls to methods & replacing them with the expanded code of their definitions at each method call. This is called as 'in lining the code'. In lining doesn't violate encapsulation or info hiding, but does improve performance because it eliminates the overhead of making a method call.

Code Example

Consider the following example.

```
class SinglyList
{
 public final double add()
   {
    //code to add a record in the linked list
   }
}
public class StringSinglyList extends SinglyList
{
 public double add()                    //final method
cannot be overridden
   {
     //code to add a record in the string linked list
   }
}

final class SinglyList
{
 public double add()
   {
    //code to add a record in the linked list
   }
}
public class StringSinglyList extends SinglyList
{
 public double add()                    //final class
cannot be a extended
   {
```

```
    //code to add a record in the string linked list
  }
}
```

What is a final class? In which scenario do you need to make the class as a final class?

Object class

- Object class is cosmic super class.
- Every class in Java implicitly extends Object.
- A variable of type Object can be used to refer to objects of any type.

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

- Available in java.lang package

Class Object is the root of the class hierarchy. Every class has Object as a super class. All objects, including arrays, implement the methods of this class.

The Object class defines basic state and behavior that the entire object must contain. All the user defined classes and pre-designed classes in the Java development environment. All Java classes directly or indirectly extends from the Object (base) class. This is true even though it is not included in "extend Object" in the class definition. For example,

```
public class MatchClass
 {
   ...
 }
    //is equivalent to
 public class MatchClass extends Object
 {
   ...
 }
```

This helps for a polymorphic referencing throughout. An array of objects can hold a reference of mix subclasses. The `Object` class also provides methods that can be overridden as per subclass requirement.

A variable of type `Object` can be used to refer to objects of any type.

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

Note: A variable of type `Object` is useful only as a generic place holder for arbitrary values. To do anything specific, type casting is necessary.

```
Employee e = (Employee) obj;
```

Following are the some of the important methods of `Object` class.

Method	Description
`String toString()`	Returns a string representation of the object.
`void finalize()`	Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
`boolean` **`equals`**`(Object obj)`	Indicates whether some other object is "equal to" this one.
`Object` **`clone`**`()`	Creates and returns a copy of this object.
`int` **`hashCode`**`()`	Returns a hash code value for the object.
`Class<?>` **`getClass`**`()`	Returns the runtime class of this Object.
`void` **`wait`**`()`	Causes the current thread to wait until another thread invokes the notify () method or the notifyAll () method for this object.
`void notify``()`	Wakes up a single thread that is waiting on this object's monitor.
`void notifyAll``()`	Wakes up all threads that are waiting on this object's monitor.

`toString()` method

`toString()` method of `Object` class is used for String representation of an object. It is an overridden method. The result should be a concise but informative representation that is easy for a person to read. The change in behavior of subclass is obtained by overriding the `toString()` method. Specify the `String` representation as per sub class needs in the `toString()` method.

Code Example

```
class Employee
{
 int eid,sal;
 String ename;
 Employee()
 {
   //constructor code here
 }
 // Custom toString () Method.
 public String toString()
 {
    return eid+" "+ename+" "+sal;
 }
 public static void main(String args[])
 {
  Employee emp=new Employee();
  System.out.println (emp);
 }
}
```

`finalize()` method

`finalize()` method is called by the garbage collector on an object when garbage collection determines that there are no more references to the object. A subclass overrides the `finalize()` method to dispose off system resources or to perform other cleanup.

The `finalize()` method of class `Object` performs no special action; it simply returns normally. Subclasses of Object may override this definition.

The `finalize()` method is never invoked more than once by JVM for any given object.

Code Example

```
//import specific package
class FileDemo
{
 FileDemo()
 {
 }
 void doOperationWithFile()
 {
   File f=new File("filename.txt");
   // write some file code here
   //file closing code here
 }
 public void finalize()
 {
  System.out.println("Finalize called here");
 }
 public static void main(String args[])
 {
```

```
  FileDemo fd=new FileDemo();
  fd.doOperationWithFile();
 }
}
```

`equals()` method

All the objects are having its own identity (it is a location in the memory) and state (it is an object data). The equality of the references can be checked with '=='operator.

Code Example

```
class MyDate
{
 int dd,mm,yy;
 MyDate(int dd,int mm,int yy)
  {
   //constructor code here
  }
public static void main(String[] args)
 {
   MyDate d1=new MyDate (1,2,2007);
   MyDate d2=new MyDate (1,2,2007);
    if(d1==d2)
    {
      System.out.println("Dates are  same");
    }
   else
    {
      System.out.println("Dates are different");
    }
 }
```

In the code snippet mentioned above, d1 and d2 are two references. When these are compared using '==' operator, two references are copied, so the output obtained is "Dates are different" than expected "Dates are same".

To compare the values of actual object, `equals()` method of `Object` class is overridden.

This method can be used to check for the equality of the state of two different objects. In this case, the output is "Dates are same".

Code Example

```
class MyDate
 {
  int dd,mm,yy;
  MyDate (int dd, int mm,int yy)
   {
    //constructor code here
   }
  public boolean equals(Object obj)
   {
    MyDate de=( MyDate)obj;
    if(dd==de.dd && mm==de.mm && yy==de.yy)
    return true;
    else
    return false;
   }
  public static void main(String[] args)
   {
    MyDate d1=new MyDate (1,2,2007);
    MyDate d2=new MyDate (1,2,2007);
    if(d1.equals(d2))
```

```
      {
     System.out.println ("Dates are same");
      }
    else
      {
       System.out.println ("Dates are different");
      }
    }
```

hashCode() method

This method returns a hashCode value for the object. This method is supported for the benefit of hash tables such as those provided by `HashMap`.

The general contract of hashCode is:

- If two objects are equal according to the `equals(Object)` method, then calling the hashCode method on each of the two objects must produce the same integer result.
- It is not required that if two objects are unequal according to the `equals(java.lang. Object)` method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

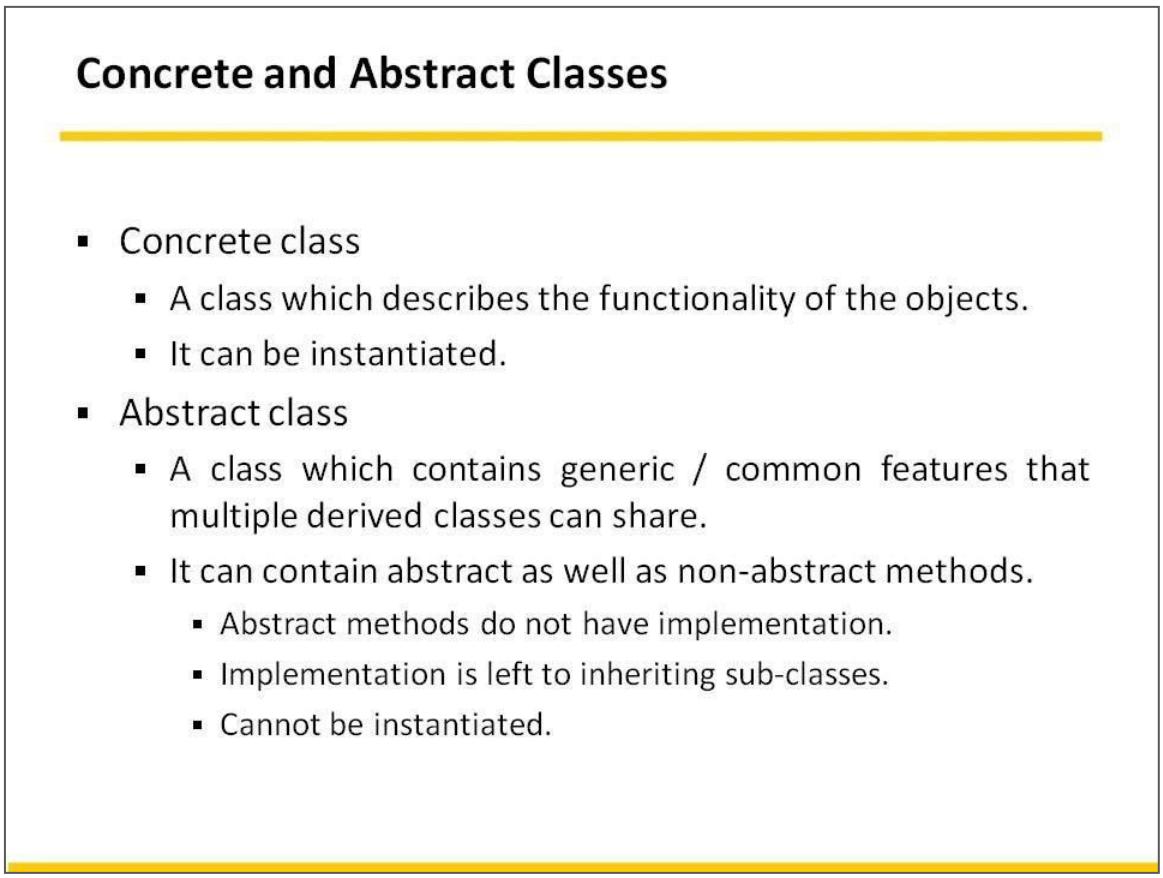
The `hashCode()` method defined by class `Object` does return distinct integers for distinct objects.

```
public int hashCode()
```

Code Example

```
class HashCodeDemo
{
 int dd,mm,yy;
 HashCodeDemo (int dd, int mm,int yy)
 {
```

```
  this.dd=dd;
  this.mm=mm;
  this.yy=yy;
 }
 public static void main(String[] args)
 {
  HashCodeDemo d1=new HashCodeDemo (1,2,2007);
  HashCodeDemo d2=new HashCodeDemo (1,2,2007);
  System.out.println(d1.hashCode());
  System.out.println(d2.hashCode());
 }
}
```



A class in which all the functionalities of an objects are defined is called a concrete class. Objects of such a class can be created.

An entity in problem domain needs to be mapped with a class. An entity which is conceptual (virtual) in real-life needs a mapping with a class in programming domain. For example, consider a shape has to be drawn. This cannot be done unless the type of shape such as rectangle or circle is specified. Thus shape is a virtual entity in the problem domain.

To map this in the programming domain, '`Shape`' needs to be defined as a class that has '`area`' method. No behavior can be associated with this method unless the shape is specified (rectangle or circle). In this scenario, '`Shape`' class is declared as an abstract class with '`area`' as a method with no implementation. Each of the derived classes of '`Shape`' such as '`Rectangle`' should have its own implementation of 'area' method.

A class which has at least one method that has no implementation is called an abstract class. Other methods can be abstract or non abstract. By design, objects

of abstract class cannot be created, but references can be created. So it is the responsibility of the developer to implement the abstract method in the derived class.

Abstract class is defined using `abstract` keyword. The `abstract` keyword enables creation of classes and class members solely for the purpose of inheritance – to define common features of derived classes.

Using abstract class, a hierarchy can be created that is easily extensible.

Consider the following example.

```
abstract class Shape
{
  public abstract void area();
}
public class Circle extends Shape
{
  int radius;
 // . . .other code
  public void area()
  {
     // code to find area of circle
  }
}
public class Rectangle extends Shape
{
 int len, bred;
 // . . . other code
  public void area()
  {
    // code to find area of rectangle
  }
}
```

This is the client code:

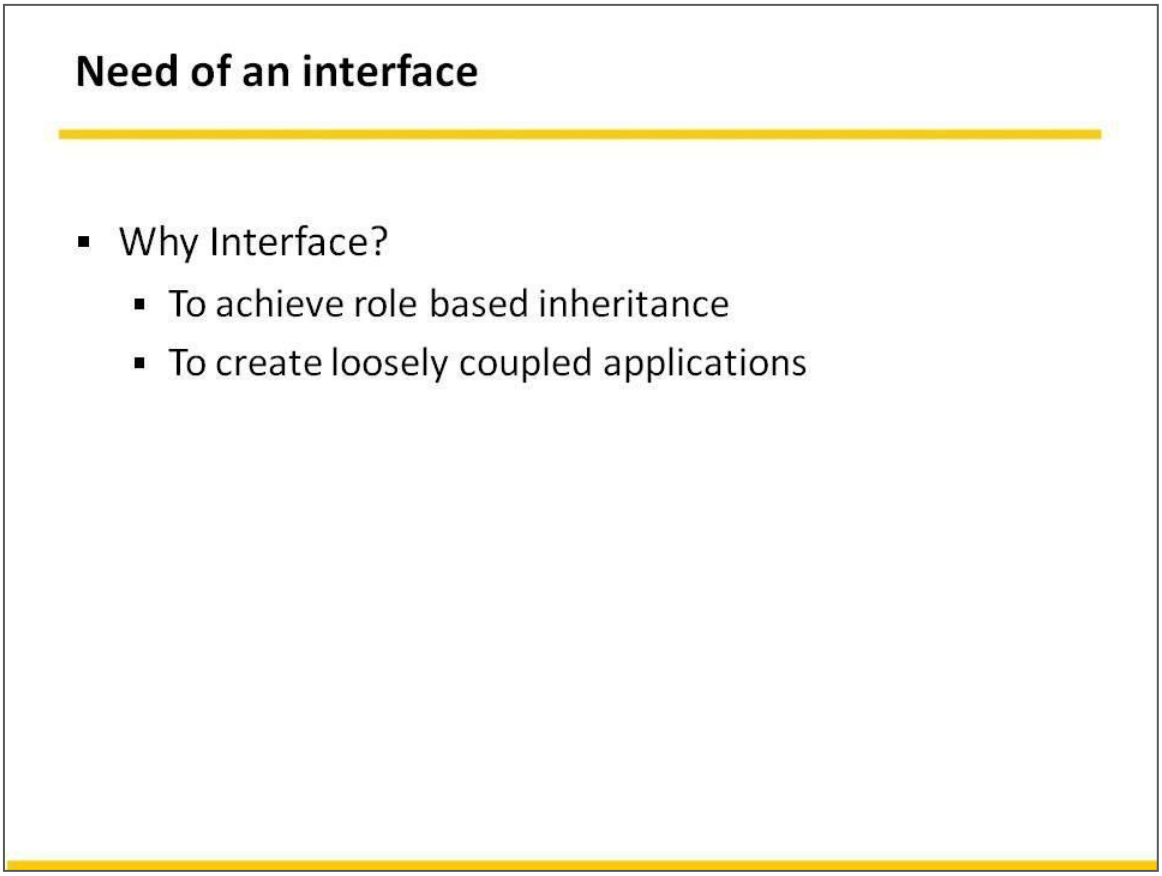
```
public static void main(String [] args)
{
  Circle cir = new Circle(5);
  cir.area();
  Rectangle rect = new Rectangle(3,5);
  rect.area();
  Shape s=new Circle();
  s.area();
  Shape s1=new Rectangle();
  s1.area();
  . . .
}
```

In the example above, the base class has one abstract method `area()` making it an abstract base class. This method has been implemented by its derived classes – `Circle` and `Rectangle`.

Following points should be noted regarding abstract classes:

- One cannot create objects of abstract classes. Any attempt to do so, results in compile time error.
- One can create references and therefore these classes support polymorphism.
- Abstract classes are useful when creating components because they allow specifying an invariant level of functionality in some methods, but leave the implementation of other methods until a specific implementation of that class is needed.
- Abstract methods do not have implementation.
- A class inheriting from an abstract class must provide implementation to all the abstract methods, else the class should be declared as abstract.
- static modifiers cannot be used with abstract methods.
- abstract modifier cannot be used for constructors.

Easy extensible hierarchy is created using abstract classes.



As already stated, Java does not support multiple inheritance, but the purpose of multiple inheritance in addition to code reusability from multiple classes is that, it provides an opportunity to the derived class to be any of the types.

For example, if a `SalesPerson` class derives from `Employee` and also from `TaxPayer`, it has a possibility of being an Employee at some time and also a TaxPayer at some other time. Java does not support multiple inheritance due to the conflict issues, but the benefit is worth incorporating into the language.

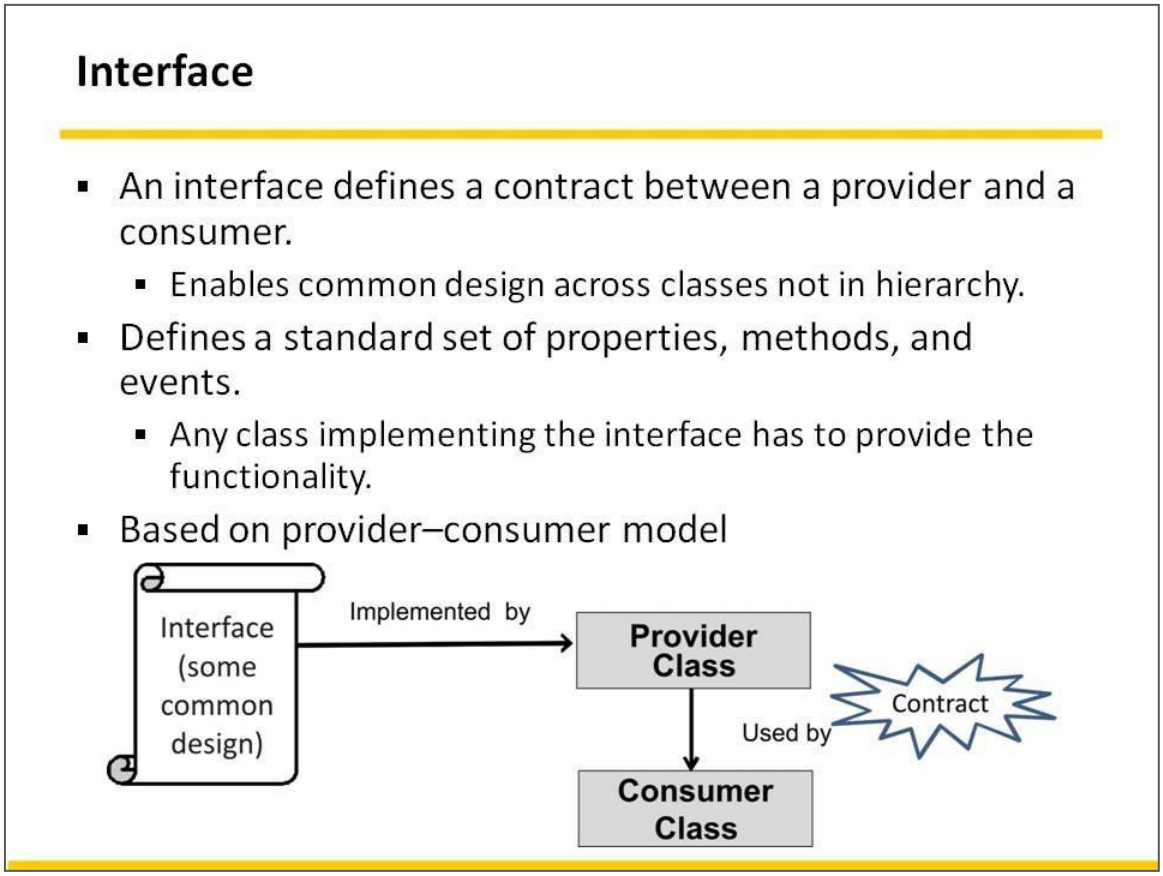
Observed closely, many operations (even in real life) are executed against a specific class type (role), and not necessarily against a specific class. For example, Tax on income is an operation which is applicable to any/everyone who is earning an income; it can be an employee/businessman/retired person. The operation also changes according to the targeted object. The tax calculation will be different for an employee, a businessman, a woman or a retired person.

Even in real life, a role does not provide any implementation; a role only defines the activities associated with it. Any entity that wants/has to get into that role will

compulsorily have to undertake the responsibility of performing those operations. It is also possible for entities belonging to two different hierarchy sets to belong to the same role. In effect, that means two entities inherit the role and not the implementation. This is called as role-based inheritance.

Translating this into programming domain, the role can be mapped to an interface. As will be seen later, an interface does not provide any implementation; it only provides the list of activities associated with that specific interface/role. Any class that will implement the interface will have to implement the operation enclosed within.

As the operations are distilled into an abstract entity, it also becomes possible for more than one solution/implementation to emerge, which ultimately provides solutions to the same problem. This will enable the developer to choose implementation depending on scenario. This will make the code pluggable. This is called a programming by contract.



Interfaces are suitable in plug-n-play like architectures where components can be interchanged as per the need. All the interchangeable components implement a common design. No extra programming is required. The common design forces each component to implement the functionality in its own way.

Interface is a contract between the consumer and the provider. It defines a standard set of properties, methods, and events. Any class which implements the interface has to provide the implementation to all the members of the interface. This class is called the provider class. The consumer class can use the functionality provided by the provider class. Interface thus allows classes to share a common design.

Considering an example from API, in-built class `ArrayList` defined in `java.util` package implements a number of interfaces.

```
public class ArrayList implements List, Serializable
```

Methods of these interfaces are provided explicit implementation in `ArrayList` class. For example, methods of `List` interface `add`, `clear`, `insert`, and `remove`

are provided explicit implementation in `ArrayList` class. Similarly, classes like `LinkedList` implements `List` and `Deque` interface, so it has to provide its own implementation of the methods of the two interfaces.

Following are the features of an interface:

- It is a reference type defined in JVM.
- It can contain data members. It contains only declaration of methods, properties, and events. These members do not have implementation. Access specifiers for these members is always public.
- A class which implements this interface has to provide the implementation for all the members.
- A class can implement multiple interfaces but can inherit only one class.
- An interface once defined and accepted must remain invariant, to protect applications written to use it. No changes can be made to interfaces once they are published.

Valid Combinations

- A class can inherit only one class.
- A class can implement one or more interfaces
- An interface can inherit another interface.

All other combinations are invalid.

Following are the rules while using/creating interfaces:

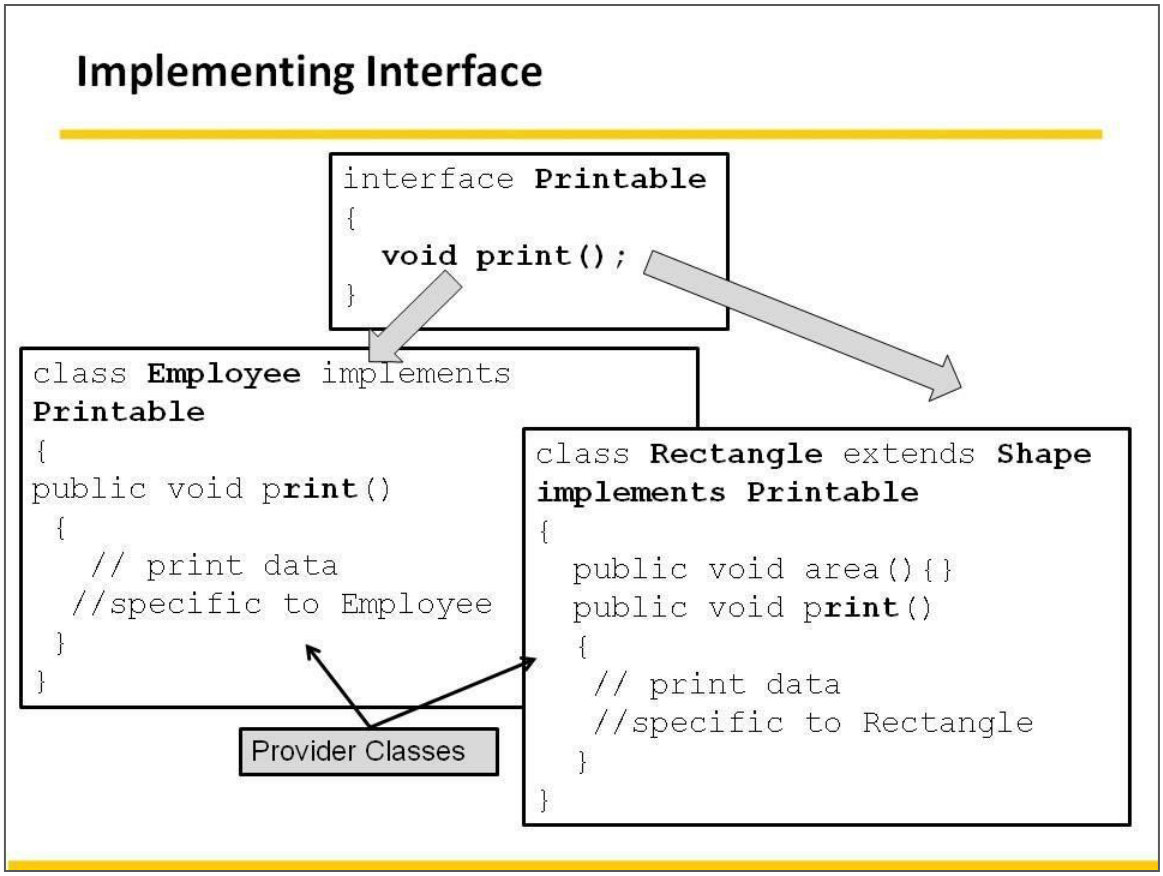
1. All the methods declared in an interface are implicitly `abstract`, and because an interface cannot provide an implementation of its declared methods, an interface does not need to declare the methods `abstract`.
2. Methods in an interface are always `public`. Interface methods may not be `static`, because `static` methods are class-specific, never `abstract`, and an `interface` can have only `abstract` methods. Fields in an interface are always `static` and `final`.
3. A variable can be declared as an interface type, and all the constants and methods declared in the interface can be accessed using this variable.

4. An interface can extend other interfaces, just as a class can extend another class. However, while a class can only extend one other class, an interface can extend any number of interfaces.
5. The class which implements an interface should give the implementation of all the methods declared in an interface otherwise, must declare as an `abstract`.
6. Implementing an Interface: A `class` declares all of the interfaces that it implements in the class declaration. To declare that your class implements one or more interfaces, use the keyword implements followed by a comma-delimited list of the interfaces implemented by your class.

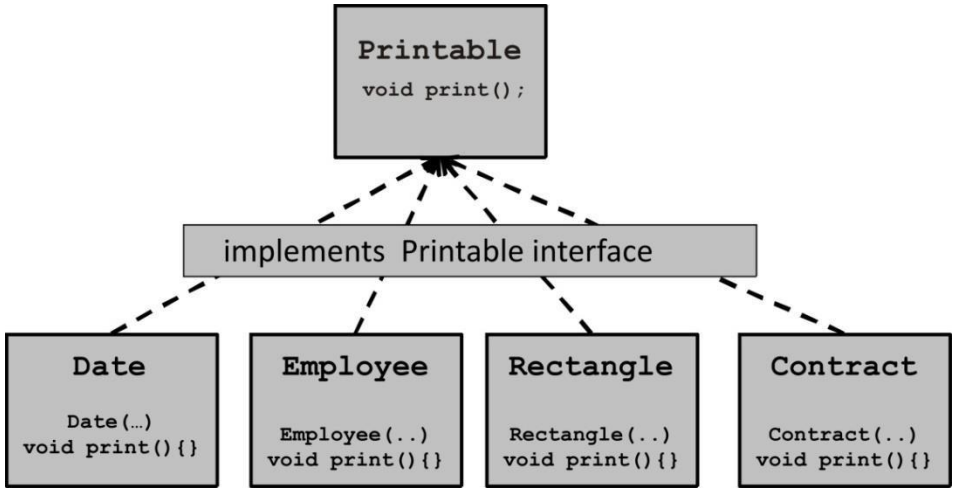
Write down some real life examples of interface.

Interface is a design pattern.

http://www.eventhelix.com/realtimemantra/patterncatalog/message_factory_and_interface_pattern.htm



`Printable` is an interface which has a `print()` method. If this is implemented by classes, a common message of printing is passed to the classes. What actually should be printed depends on the class that implements it.



`Employee`, `Date`, `Rectangle and Contract` are concrete classes which implement the interface. Each class implements the method in its own way.

Using an Interface

Since interface is a reference type, it can refer to the class implementing it. A method is called using this reference to get the desired result.

```
// in the client code (in the consumer class)
Printable prn;
Employee employee1 = new Employee(101,"Jane",20000);

employee1.print();
//or
prn = employee1;
prn.print();

Rectangle rect = new Rectangle(3,6);
rect.print();
```

Interface can be also used in case of polymorphism as in following code snippet. `display()` method takes `Printable` as an argument. `prn` reference can refer to any object whose type implements `Printable` interface. Accordingly `print()` method of that class would be called.

```
public static void display(Printable prn)
{
    prn.print();
}
```

```
// in the client code
Employee employee1 = new Employee(101,"Jane",20000);
```

```
display(employee1); //called with employee's instance
so Print() method
                    //of Employee class would be called
```

Tagging Interfaces

- Tagging interface is also called as Marker Interface.
- Tagging Interface does not contain any method or data member.
- It is an Empty Interface.

e.g. `Clonable` ,`Serializable` Interface etc.

Marker Interface

Marker Interface is used only for tagging or marking purpose. So, gives additional information about the behavior of a class.

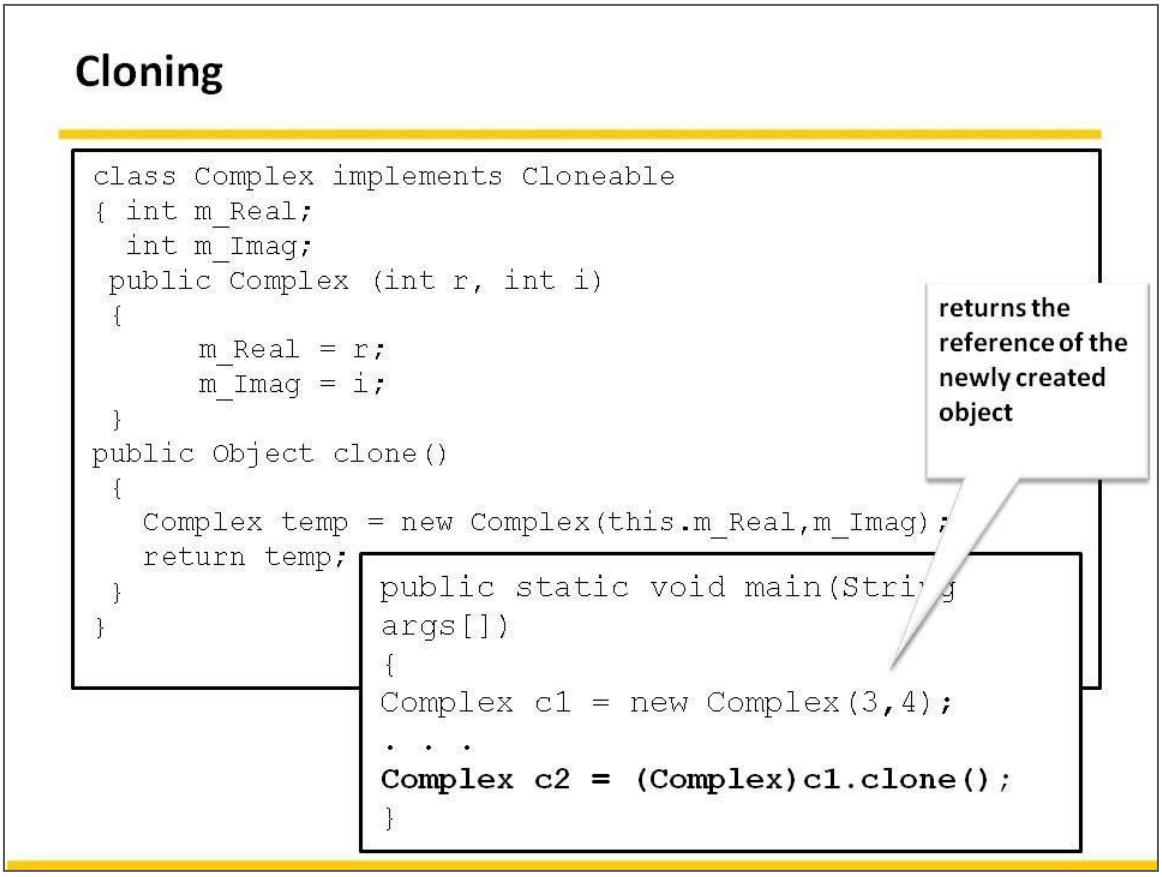
Abstract Class Vs. Interface

	Abstract class	Interface
Methods	Can have abstract as well as non-abstract methods	All methods are abstract
Best suited for	Objects closely related in hierarchy.	Contract based provider model
Component Versioning	By updating the base class all derived classes are automatically updated.	Interfaces are immutable

The choice to design a particular functionality as abstract class or interface is sometimes difficult. Abstract class and interface can be differentiated on certain points.

- Abstract class can have one or more methods as abstract. All methods in an interface are abstract.
- Abstract class is used for objects closely related in hierarchy. Interface is used to provide common functionality to unrelated classes.
- Abstract classes provide a simple and easy way to version the reusable software components. By updating the base class, all inheriting classes are automatically updated with the change. Interfaces are immutable; means they cannot be changed once created. If a new version of an interface is required, a whole new interface needs to be created, which has repercussions in all the classes implementing it.

What is the difference between abstract class and interface?



Cloning is the process of creating an object as a copy of another object.

- Cloning can be of two types shallow cloning: Shallow clone copies the references but not the referenced objects. A reference in the original object and the reference in a shallow-cloned object both refer to the same object.
- Deep cloning: Deep clone copies the referenced objects as well. A deep-cloned object contains a copy of everything directly or indirectly referenced by the object.

In the code snippet mentioned on the slide, complex class implements `Clonable` interface. The class has to implement `clone()` method. It indicates that the object is of their class is cloneable.

`Object` c1 is cloned. The `clone()` method returns a new object which is a copy of object. c2 refers to this newly created object returned from the method. The method returns an object. So it needs to be typecast to the appropriate type.

This chapter covers Java support APIs. This topic covers some basic concepts like `String`, `StringBuffer`, inner classes, reflection, `File` and `System` class.

Objectives

At the end of this chapter you will be able to

- Define inner class and its types.
- Use anonymous, simple and static inner class.
- Use wrapper classes.
- Use auto-boxing and auto-unboxing.
- Distinguish between `String` and `StringBuffer` and `StringBuilder` classes.
- Concept of `File` class.
- Concept of `System` class.

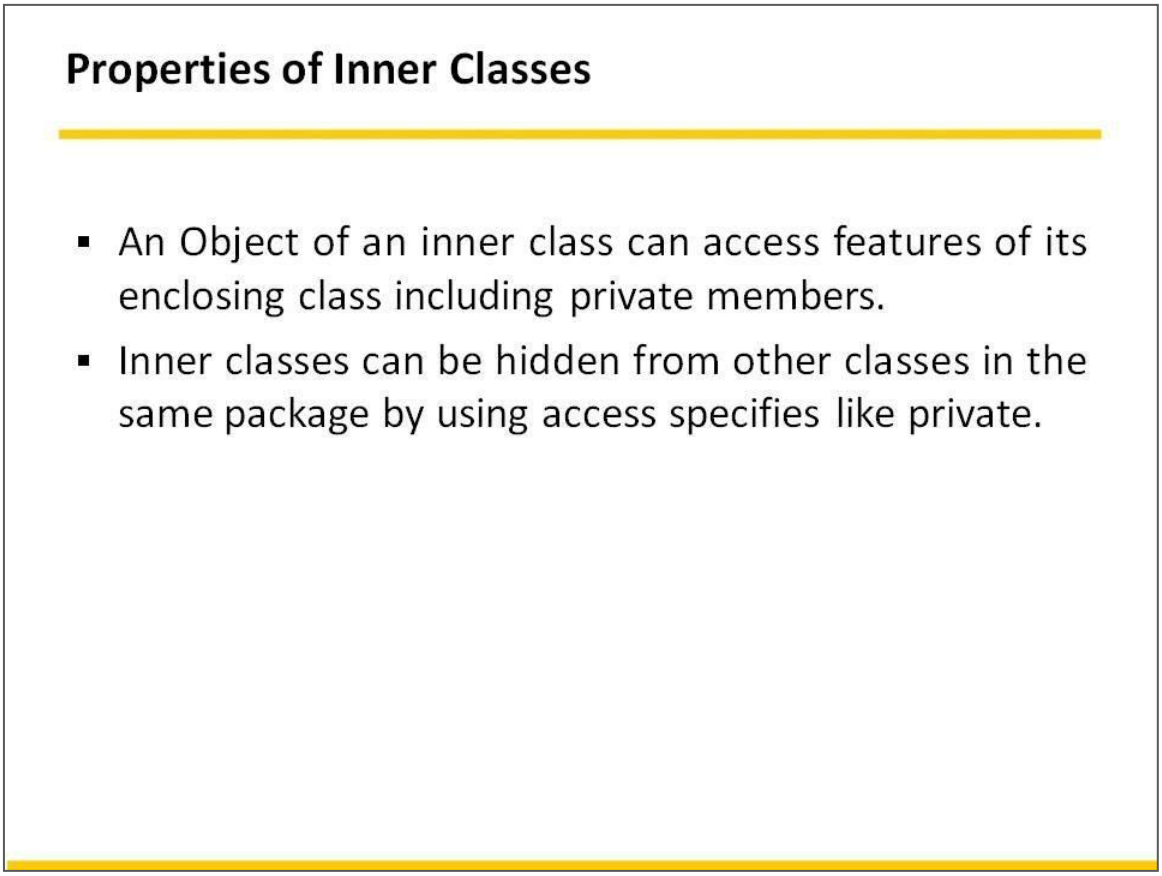
Inner Class

- Definition of one class within the definition of another class.
- Inner class is a member of the enclosing class just like other class members.

```
public class Outer{
//details of outer class
   public class Inner{
//details of inner class
}
//more details of outer class
}
```

It is possible to place a class definition within another class definition. Such a class is called an inner class. The inner class is a useful feature as it helps to group classes that need to be attached together.

This feature is provided to achieve control framework. Control framework is a particular type of application framework dominated by the need to respond to events (i.e. for event driven programming).



An object of an inner class can access the implementation of the object that created it including data that would otherwise be private, since it has a link to the enclosing object.

- Inner classes can be hidden from other classes in same package.
- Anonymous inner classes are handy when you want to define callbacks on the fly.
- Inner classes are very convenient when you are creating event driven programs.

When enclosing class is compiled the inner class is automatically compiled and the `.class` file name is enclosing- class_name$inner-class_name. If the inner class is an anonymous class then compiler generates unique number which is used as an inner class name, i.e.outerclass_name$1.

Inner classes should be avoided since they increase the complexity of code and put extra burden on the compiler. Inner class object construction requires outer class object.

Following points should be remembered while writing inner classes

- Inner class objects can be created without having outer class object only if the inner class is a static inner class.
- Inner classes can be defined
  - Within a method
  - Within a scope inside a method.
  - As an anonymous class
  - Class that extends a class
  - Class that implements an interface.

Anonymous Inner classes

- A class that is not assigned a name.
  - Declared by using the name of the class that they sub-class.
  - These classes are defined at the same time they are instantiated with `new`.
  - Facilities writing of event handlers.
  - Details of this we will discuss at the time of swings.
- Declaration of an anonymous inner class

```
Aclass( new AnonClass () {
                          void aMethod(){
                          …
                         } );
```

An inner class with no name is called anonymous inner classes.

Example: `actionPerformed` events utilize the Observer-Observable pattern -- An instance of the observer is added to the observable. Anonymous inner classes are very similar to named inner classes.

- Anonymous inner classes can override methods of the super class.
- The scope of anonymous inner classes is inside the private scope of the outer class. They can access the internal (private) properties and methods of the outer class.
- References to an inner class can be passed to other objects. Note that it still retains its scope.

There are some important Points related to Anonymous Inner class:

- Anonymous inner classes can be created using any constructor of super class.

- Since an object made from the inner class has a "life" independent of its outer class object, there is a problem with accessing local variables, especially method input parameters.
- Two ways to pass initializing parameters to an inner class:
- Initialize a property that the inner class uses -- properties have the cross-method-call persistence that local variables lack.
- Make the local variable "final" - compiler will automatically transfer the value to a more persistent portion of the computer memory. Disadvantage is that the value cannot be changed.

Usages of Anonymous Inner classes

- Very useful for controlled access to the insides of another class.
- Very useful when only want one instance of a special class is needed.

Following code snippet explains Anonymous inner class example.

```
class Demo
 {
  public Demo () {};
  public fun1()
  {
    ---
    ---
  }
  public fun2 ()
  {
    ---
    ---
  }
}
```

```
class TestAnonymous
 {
  public static void main (String args [])
   {
     new Thread (new Runnable ()
      {
       public void run ()
       {
         new Demo ().fun1 ();
         }
      }).start ();
    new Thread (new Runnable ()
     {
       public void run ()
     {
      new Demo().fun2();
       }).start();
}
```

`new` keyword has been adapted to create the Anonymous inner class.

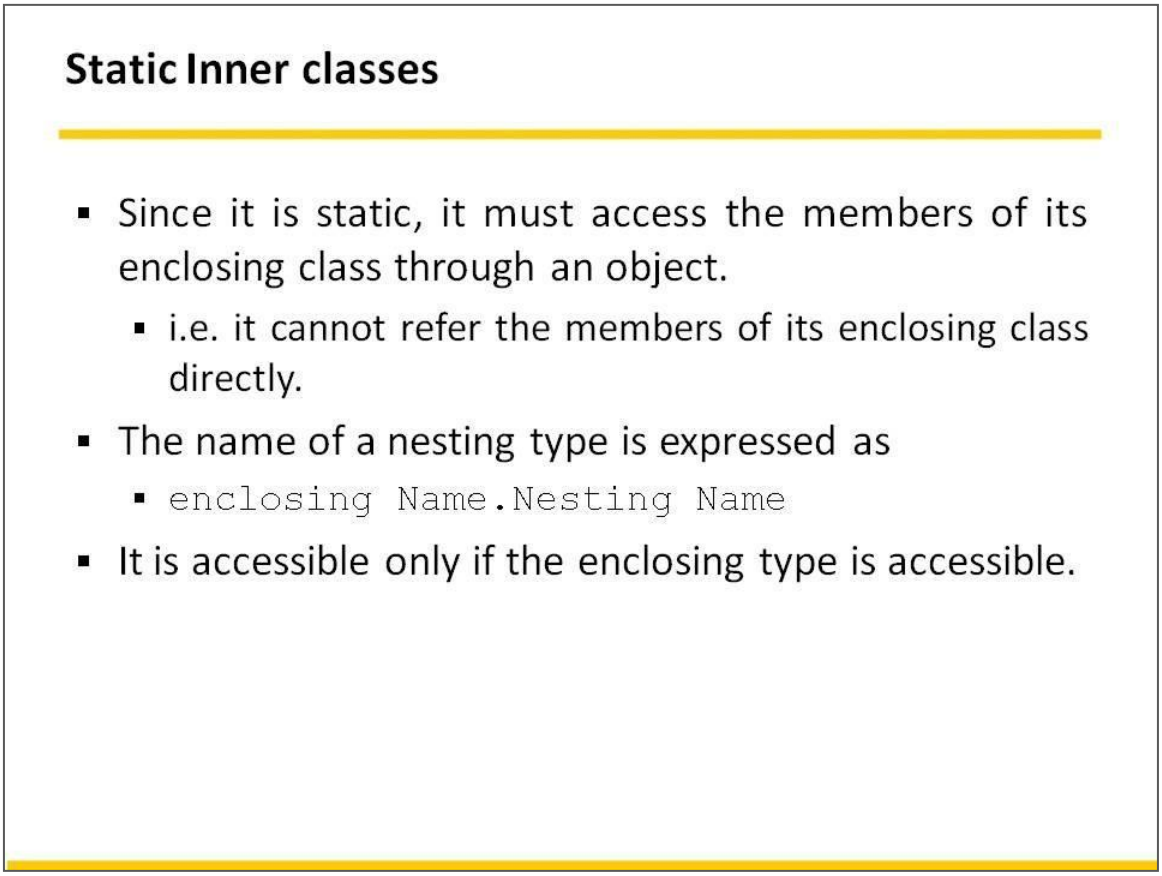
The `addWindowListener()` method's argument is an expression that defines and instantiates an anonymous inner class.

```
new WindowAdapter( ){…}
```

It indicates to compiler that the code between the braces defines an anonymous inner class. Further, the class extends the `WindowAdapter` class.

This class is not named, but it is automatically instantiated when the expression is executed.

Note: Event handlers are discussed in chapter "Swing".



Static inner class does not have a reference to the outer class unlike other inner classes.

- Only inner classes can be `private` or `static`. Other classes can never have these modifiers.
- Marking an inner class static with regards to accessing the fields of the enclosing class has some effects.
- There is only one instance of any variables, no matter how many instances of the outer class are created. In this situation the static inner class cannot determine which of non static outer class, need to be accessed.
- A static inner class cannot access instance variables of its enclosing class.
- The methods of a static inner class can access any static fields of its enclosing class as there is only one copy of these fields for all objects.

`String` class

- Java library contains a predefined class called String.
- The `String` type is not a primitive type.
- It is so important, that in certain areas Java treats it like one.

e.g.

- The ability to declare String literals instead of using `new` to instantiate a copy of the class

```
String s="seed";
```

- String is a 'First class object'.

```
public final class String extends Object implements
Serializable, Comparable
```

`String` class is from `java.lang` package. The `String` class represents character strings. All string literals in Java programs, such as "hello", are implemented as instances of this class. String is a first class object. There is no need to use `new` keyword to initialize it, and it can be initialized in following manner.

```
String s="abc";
//is equivalent to:
char data [] = {'a', 'b', 'c'};
String s = new String (data);
```

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 with all characters translated to uppercase or to lowercase. For details of these methods refer the Java Documentation.

String API

Following are the some of the methods of `String` class

Methods	Description
`String``()`	Initializes a newly created String object so that it represents an empty character sequence.
`String``(byte[] bytes)`	Constructs a new String by decoding the specified array of bytes using the platform's default charset.
`String``(byte[] bytes, Charset charset)`	Constructs a new String by decoding the specified array of bytes using the specified charset.
`String``(String original)`	Initializes a newly created String object so that it represents the same sequence of characters as the argument; in other words, the newly created string is a copy of the argument string.
`String``(StringBuffer buffer)`	Allocates a new string that contains the sequence of characters currently contained in the string buffer argument.
`String``(StringBuilder builder)`	Allocates a new string that contains the sequence of characters currently contained in the string builder argument.
`char charAt``(int index)`	Returns the char value at the specified index.
`int compareTo``(String anot`	Compares two strings lexicographically.

herString)	
int compareToIgnoreCase(String str)	Compares two strings lexicographically, ignoring case differences.
String concat(String str)	Concatenates the specified string to the end of this string.
boolean **contains**(CharSequences)	Returns true if and only if this string contains the specified sequence of char values.
boolean endsWith(String suffix)	Tests if this string ends with the specified suffix.
boolean **equals**(Object anObject)	Compares this string to the specified object.
boolean equalsIgnoreCase(String anotherString)	Compares this String to another String, ignoring case considerations.
static String **format**(Locale l, String format, Object... args)	Returns a formatted string using the specified locale, format string, and arguments.
static String **format**(String format, Object... args)	Returns a formatted string using the specified format string and arguments.
boolean **matches**(String regex)	Tells whether or not this string matches the given regular expression.
String replace(char oldChar, char newChar)	Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.
String []split(String regex)	Splits this string around matches of the

	given regular expression.
`String []`**`split`**`(String regex, int limit)`	Splits this string around matches of the given regular expression.
`boolean startsWith``(String prefix, int toffset)`	Tests if the substring of this string beginning at the specified index starts with the specified prefix.
`CharSequence` **`subSequence`**`(int beginIndex, int endIndex)`	Returns a new character sequence that is a subsequence of this sequence.
`String` **`substring`**`(int beginIndex)`	Returns a new string that is a substring of this string.
`String trim``()`	Returns a copy of the string, with leading and trailing whitespace omitted.
`static String` **`valueOf(`**`Object obj)`	Returns the string representation of the Object argument.

Code Example

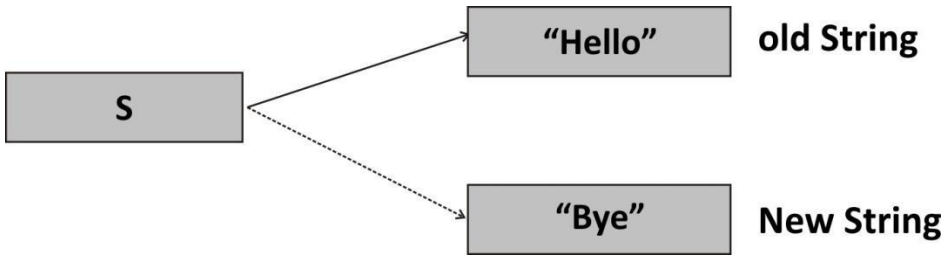
Following code snippet explains the simple methods of a `String`.

```
class StringDemo
{
 String oneString=”One”;
 StringDemo(){
 String twoString=new String(”One”);

 System.out.println(“Length of String”+”
“+oneString.length());

 if(oneString==twoString)
```

```
 {
    System.out.println("EQ");
 }
 else
  {
   System.out.println("NEQ");
   oneString ="Three";
   System.out.println("Length of String"+"
"+oneString.length());
  }
}
public static void main(String args[])
{
  StringDemo sd=new StringDemo();
}
```

String class represents an 'immutable'. The content of a String cannot be changed once it is instantiated.

Consider the following example:




```
String s="Hello"; ----------> 1
```

So, s is reference of String "Hello"

```
s="Bye" ;.....> 2
```

When a new String is assigned to String variable it discards the old string and replaces it with the modified one. This in effect is a new String.

String Pool

The strings are objects in Java.

```
String s1=new String (“Welcome”); //case1
String s2=“Welcome”;//case2
```

Every time when you create string literal the JVM checks the string literal pool. If pool is ready then a reference to pooled instance is returned. If pool is not available, a new String object instantiates then is placed in the pool.

Following code snippet explains the string pool concept.

Code Example

```
String s1=“Hi”; //case1
String s2=“Hi”;//case2


/*In case 1, s1 is created newly and kept in the pool.
But
  in case 2,  s2 refer the s1, it will not create new
one
  instead */
   if (s1 == s2) System.out.println ("equal");
 //Prints equal.

  String n1=new String(“Hello”);
  String n2=new String(“Hello”);
   if(n1 == n2) System.out.println("equal");
 //Not equal
```

The s1 and s2 are instances; but JVM optimizes while instantiating string literals to increase performance and decrease memory overhead, Instead of creating a new object every time, it maintains a pool of strings which can be re-used.

`String` in `switch` Statements (Prior to JDK 7)

- To use a String in `switch` statement, gives compilation error.
- Switch can work only with integer constants(static final int)or literals.
- To use, convert the String to char and use single quotes.
- Required equals() method with if …else technique.

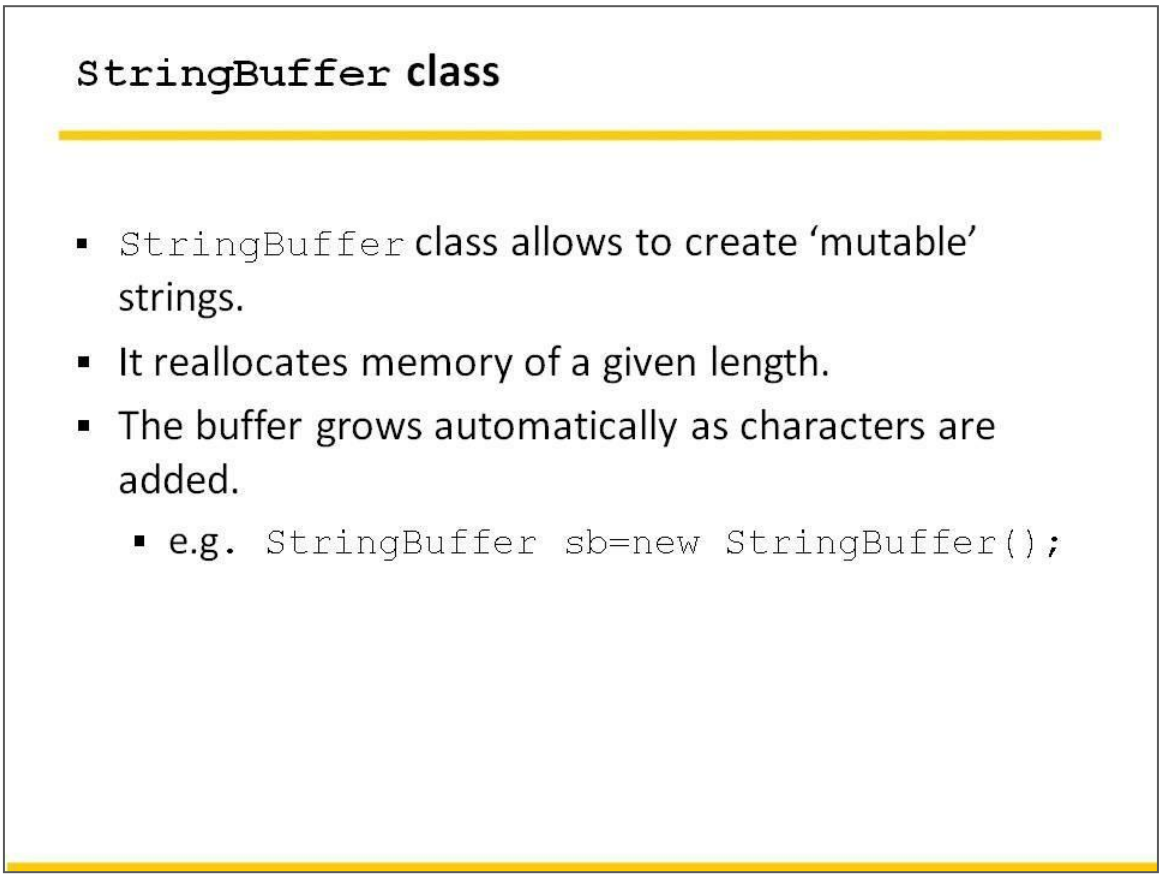
Before Java 7

```
String color = "red";

if (color.equals("red")) {

System.out.println("Color is Red");
} else if (color. equals("green")) {

System.out.println("Color is Green");
} else {

System.out.println("Color not found");
}
```

In Java 7

```
String color = "red";
  switch (color) {
 case "red":
   System.out.println("Color is Red");
   break;
 case "green":
  System.out.println("Color is Green");
   break;
 default:
     System.out.println("Color not found");
 }
```

Java 7 has a new feature called `String` in `switch` statements. Prior to Java 7 only Integer constants could be used in switch case. Using Java 7 it is possible use String in a switch statement.




```
public final class StringBuffer extends Object
implements Serializable
```

A `StringBuffer` implements a mutable sequence of characters. A `StringBuffer` is like a `String`, but can be modified. At any point in time it contains some particular sequence of characters, but the length and content of the sequence can be changed through certain method calls.

`StringBuffer` class is safe for use by multiple threads. The methods are synchronized where necessary so that all the operations on any particular instance behave as if they occur in some serial order that is consistent with the order of the method calls made by each of the individual threads involved.

String buffers are used by the compiler to implement the binary string concatenation operator +. For example, the code:

```
x = "a" + 4 + "c";
```

is compiled to the equivalent of:

```
x = new StringBuffer ().append ("a").append (4).append
("c"). toString ();
```

This creates a new string buffer (initially empty), appends the string representation of each operand to the string buffer in turn, and then converts the contents of the string buffer to a string. Overall, this avoids creating many temporary strings.

Every string buffer has a capacity. As long as the length of the character sequence contained in the string buffer does not exceed the capacity, it is not necessary to allocate a new internal buffer array. If the internal buffer overflows, it is automatically made larger.

Preallocation: by default it is the length of the string it can store before it must allocate more space

`StringBuffer` class is needed when transforming strings. Operations are faster.

Using `StringBuffer` an efficient way to read in characters and appends them to an existing String. Every time characters are appended to a string, they are added to the object of `String` class , which frees the need for a new memory space to hold the larger string. This affects the performance. More characters requires the string to be relocated again. The `StringBuffer` class avoids this problem. `StringBuffer` is a thread-safe, mutable sequence of characters.

StringBuffer API

Following are the some of the methods of `StringBuffer` class

Method	Description
`StringBuffer``()`	Constructs a string buffer with no characters in it and an initial capacity of 16 characters.
`StringBuffer``(CharSequence seq)`	Constructs a string buffer that contains the same characters as the specified CharSequence.
`StringBuffer``(int capacity)`	Constructs a string buffer with no

	characters in it and the specified initial capacity.
StringBuffer(String str)	Constructs a string buffer initialized to the contents of the specified string.
StringBuffer **append**(String str)	Appends the specified string to this character sequence.
int capacity()	Returns the current capacity.
char charAt(int index)	Returns the char value in this sequence at the specified index.
StringBuffer delete(int start, int end)	Removes the characters in a substring of this sequence.
StringBuffer **insert**(int offset, boolean b)	Inserts the string representation of the boolean argument into this sequence
int length()	Returns the length (character count).
StringBuffer **replace**(int start, int end, String str)	Replaces the characters in a substring of this sequence with characters in the specified String.
StringBuffer **reverse**()	Causes this character sequence to be replaced by the reverse of the sequence.
String **substring**(int start)	Returns a new String that contains a subsequence of characters currently contained in this character sequence.
String **substring**(int start, int end)	Returns a new String that contains a subsequence of characters currently contained in this sequence.

`String` **`toString`**`()`	Returns a string representing the data in this sequence.

Code Example

Following code snippet explains the simple methods of a `StringBuffer`.

```
public class StringBufferReverseExample
{
 public static void main(String[] args)
  {
   //create StringBuffer object
   StringBuffer sb = new StringBuffer("Java
StringBuffer Reverse Example");
   System.out.println("Original StringBuffer Content :
" + sb);
   //To reverse the content of the StringBuffer use
reverse method
   sb.reverse();
   System.out.println("Reversed StringBuffer Content :
" + sb);
  }
}
```

StringBuilder class

- StringBuilder is introduced in Java 5.0 version.
  - This is a only replacement with a StringBuffer class.
  - There is no difference between StringBuffer and StringBuilder.
- Methods of StringBuilder class are non synchronized.
- Recommended for faster applications.

```
public final class StringBuilder extends Object
implements Serializable, CharSequence
```

This class is introduced in Java 5 which is a mutable sequence of characters. This class provides an API compatible with `StringBuffer`, but it is not synchronized. This class is just for the replacement of `StringBuffer`. The main operations of `StringBuilder` are `append()` and `insert()`. The `append` method always adds these characters at the end of the builder; the `insert` method adds the characters at a specified point. `StringBuilder` is not thread-safe and none of its methods are synchronized.

Code Example

Following code snippet explains the simple methods of a `StringBuilder`.

```
class StringBuilderExample
{
 public void insertTextExample()
 {
  StringBuilder sb=new StringBuilder("Hello");
  sb.append("Add a String");
  System.out.println (sb.toString());
 }
 public static void main(String args[])
 {
  StringBuilderExample sb=new StringBuilderExample();
  sb.insertTextExample();
 }
}
```

The Methods of StringBuilder are the same of StringBuffer class only difference is non-synchronized.To refer methods of StringBuilder class refer

http://docs.oracle.com/javase/7/docs/api/

Wrapper Classes

- Java provides eight primitive data types. But sometimes there is a need to convert a primitive type to an object.
- All Java primitive have class counterparts called object wrappers or wrapper classes.
- Why Wrapper Classes?
  - To provide a home for methods & variables related to the type.
  - Create objects to hold values for generically written classes that know how to handle only object references.

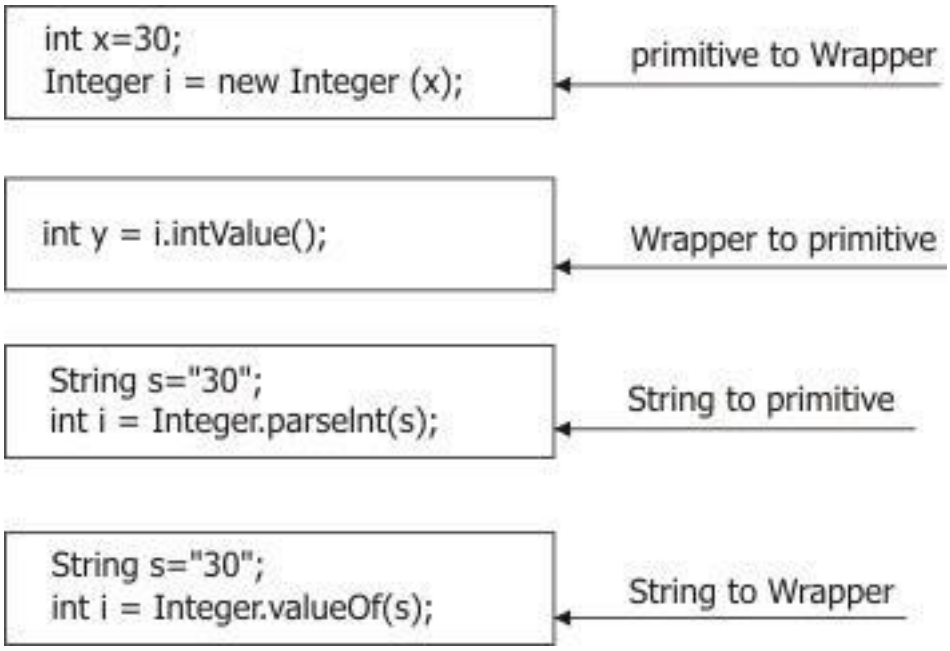
Wrapper classes provide a home for methods and variables related to the type. They create objects to hold values of a particular primitive type for generically written classes that only know how to handle Object references.

Java provides eight primitive data types. Following table shows the wrapper class for respective primitive data type.

Primitive Type	Respective Wrapper class
`byte`	Byte
`short`	Short
`char`	Character
`int`	Integer
`long`	Long

float	Float
double	Double
boolean	Boolean

Wrapper classes are final classes. They cannot be inherited. The constructor of wrapped class will allow creating primitive to respective wrapped class and from object to primitives and String to primitive.

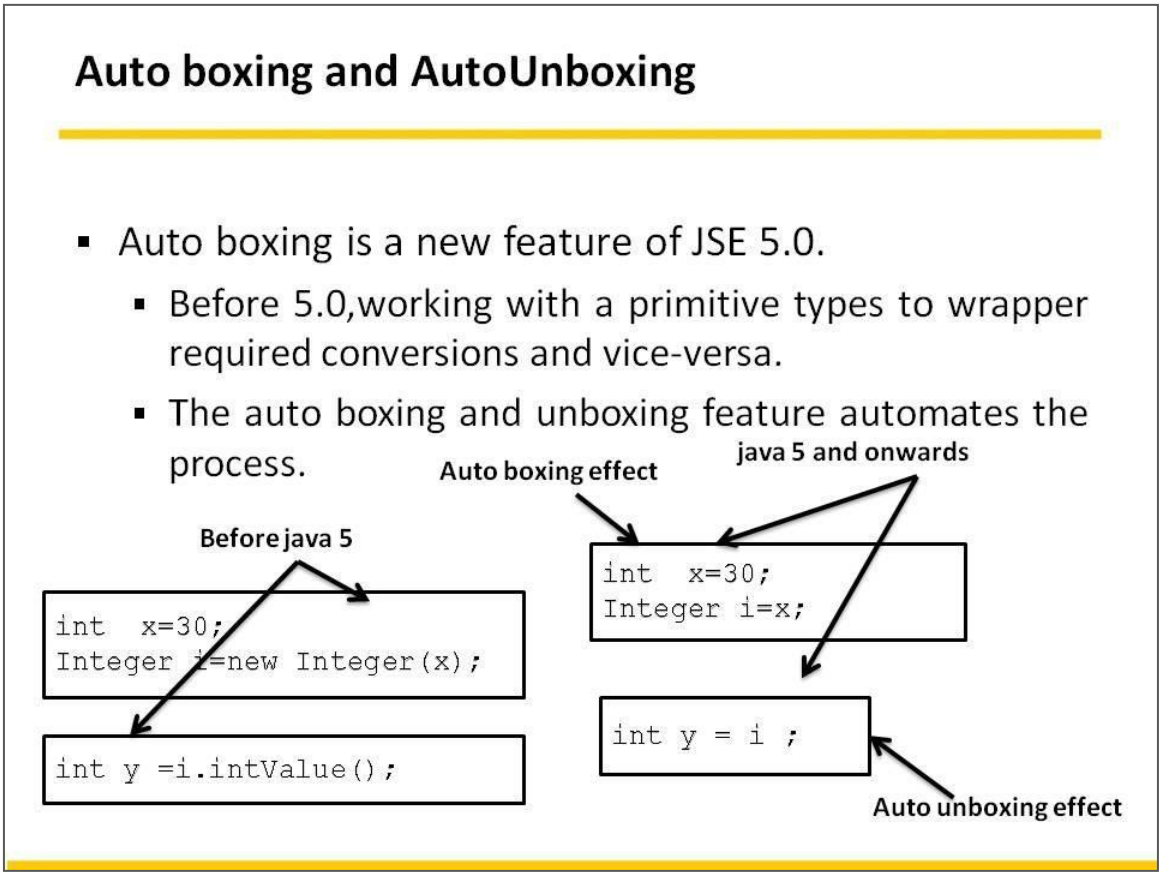


Code Example

Following code shows the use of wrapper class

```
class WrapperDemo
{
  int xvalue;
  WrapperDemo()
  {
   xvalue=10;
   //creating wrapper of int
```

```
    Integer xi=new Integer(xvalue);
    System.out.println("Output1"+" "+xi);
    //convert wrapper to primitive
    int yvalue=x.intValue();
    System.out.println("Output2"+" "+yvalue);
    }
  public static void main(String args[])
  {
     WrapperDemo w=new WrapperDemo();
    }
}
```



In Java primitive types cannot be used directly with collection classes. Collections hold objects. The values of primitive types need to be boxed. Prior to Java 5, wrapper class were used.

Prior to Java 5

```
void wrapperMeth(){
        int x=10;
        Integer i=new Integer(x); //Wrapping primitive
boxing effect.
        x=i.intValue ();       //Unwrap       unboxing
effect.
}
```

In the above code snippet int ‘x’ is wrapped using wrapper class `Integer` to ‘i’.

- Compiler gives an error if following statement is writing as primitive types cannot be wrapped automatically.

```
int x=10;
Integer i=x;
```

- Hence, `Integer` class is instantiated to wrap ‘x’ Then,

```
int x=10;
Integer i=new Integer(x);
```

- Compiler gives an error if following statement is writing as object types cannot be unwrapped automatically.

```
int x=i;
```

- Hence, have to call `xxxValue ()` method on object so that it will convert to respective primitive type.

```
int x=i.intValue()
```

So, that wrapper ‘i’ convert to primitive ‘`x`’.

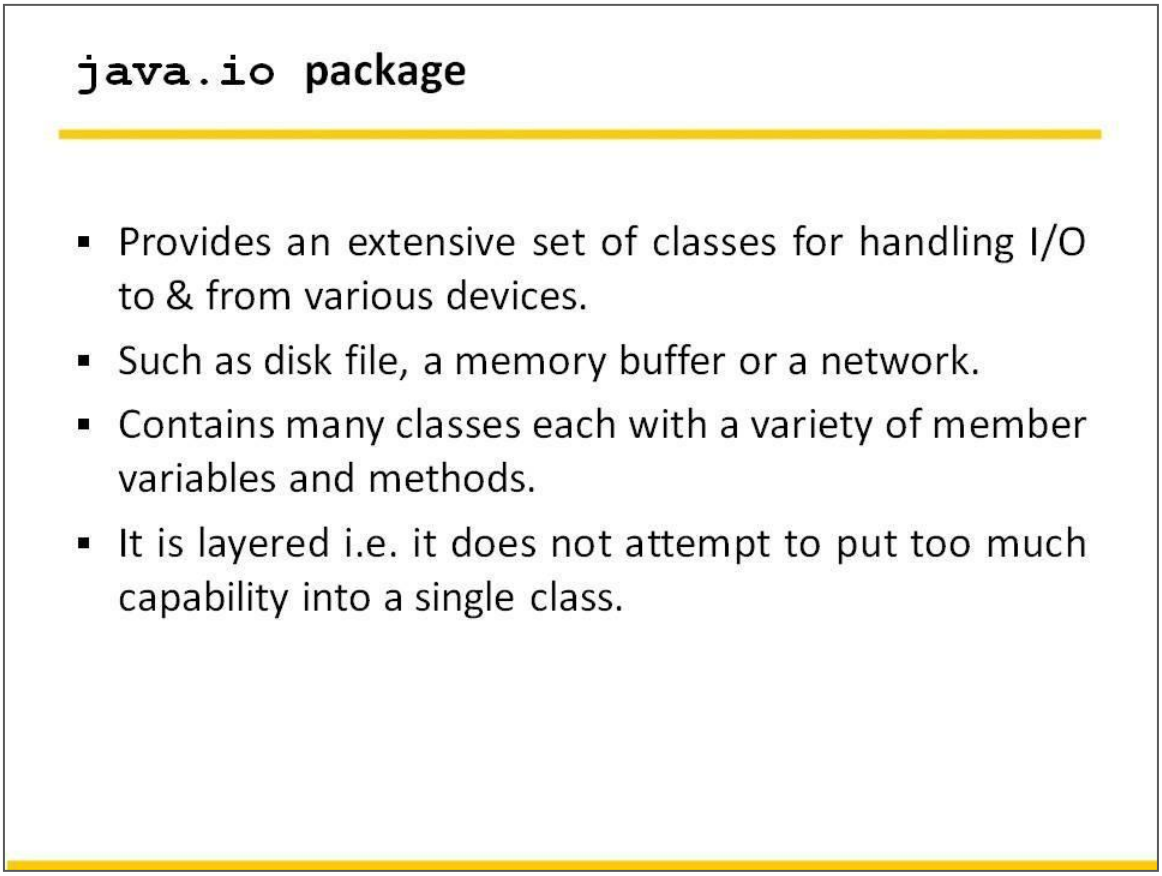
In Java 5 and onwards, same code can be written as

```
void wrapperMeth()
{
   int x=10;
   Integer i=x; //automatically wrapping primitive to
wrapper object autoboxing
                // effect.
   x=i;       //automatically Unwrap      auto unboxing
effect.
}
```

‘x’ is automatically wrapped to a wrapper object `Integer` class.

```
int x=i;
```

Statement unwrapped automatically to primitive type ‘`int`’. This automatic wrapping of primitive type to wrapper object is called ‘auto boxing’ and unwrapping is called ‘auto unboxing’.



Java provides an extensive library populated with a set of classes & interfaces under `java.io` package.

The classes provided under the `java.io` package are used to deal with various I/O devices to perform different I/O operations such as reading or writing data to an I/O device or even deleting some data on some I/O device.

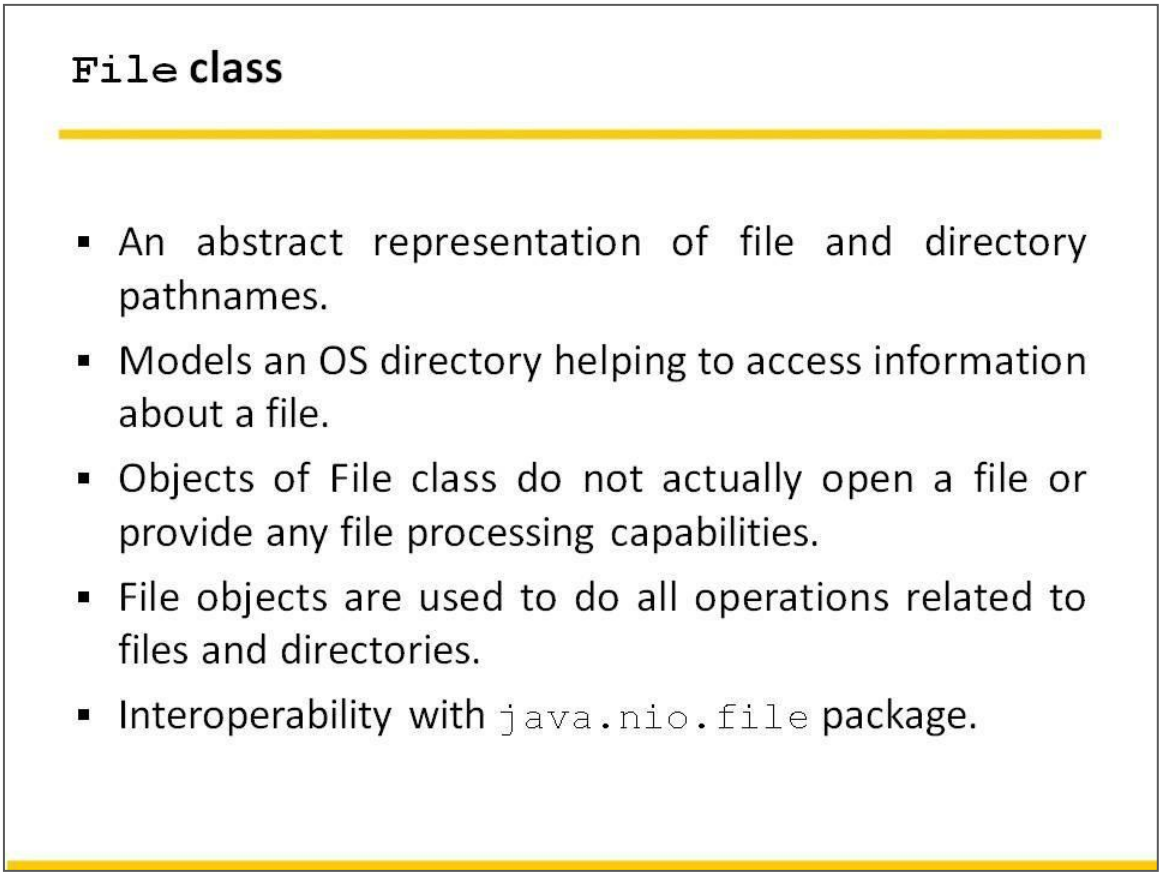
The advantage of using these classes is that, these classes do not have much functionality i.e. a single class is not capable of perform a big operation. So they are layered. You can achieve a number of features by layering one class over another. We will come across some examples of layering in further readings.

Java's stream-based I/O is built upon four abstract classes: `InputStream`, `OutputStream`, `Reader`, `Writer`. They are used to create several concrete stream subclasses. Although your programs perform their I/O operations through concrete subclasses, the top-level classes define the basic functionality common to all stream class.

Difference between Stream and Reader / Writer classes

Since byte-oriented streams are inconvenient for processing information stored in Unicode (Unicode uses two bytes per character), there is a separate hierarchy of classes for processing Unicode characters that inherit from the abstract `Reader` and `Writer` classes. These classes have read and write operations that are based on 2-byte Unicode characters rather than on single-byte characters.

`InputStream` and `OutputStream` are designed for byte streams. Reader and Writer are designed for character streams. The byte stream classes and the character stream classes form separate hierarchies. These four classes are also defined as root classes for input/output.



The Java API says that the class File is "An abstract representation of file and directory pathnames." The `File` class is not used to actually read or write data; it used to work at a higher level, making new empty files, searching for files, deleting files, making directories, and working with paths.

File API

Following are the some of the important methods of `File` class.

Method	Description
`File(File parent, String child)`	Creates a new File instance from a parent abstract pathname and a child pathname string.
`File(String pathname)`	Creates a new File instance by converting the given pathname string into an abstract pathname.

`File(String parent, String child)`	Creates a new File instance from a parent pathname string and a child pathname string.
`File(URI uri)`	Creates a new File instance by converting the given file: URI into an abstract pathname.
`boolean canRead()`	Tests whether the application can read the file denoted by this abstract pathname.
`boolean canWrite()`	Tests whether the application can modify the file denoted by this abstract pathname.
`boolean createNewFile()`	Atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist.
`boolean delete()`	Deletes the file or directory denoted by this abstract pathname.
`boolean exists()`	Tests whether the file or directory denoted by this abstract pathname exists.
`String getName()`	Returns the name of the file or directory denoted by this abstract pathname.
`boolean` `isHidden()`	Tests whether the file named by this abstract pathname is a hidden file.
`long` `length()`	Returns the length of the file denoted by this abstract pathname.
`URI toURI()`	Constructs a file: URI that represents this abstract pathname.

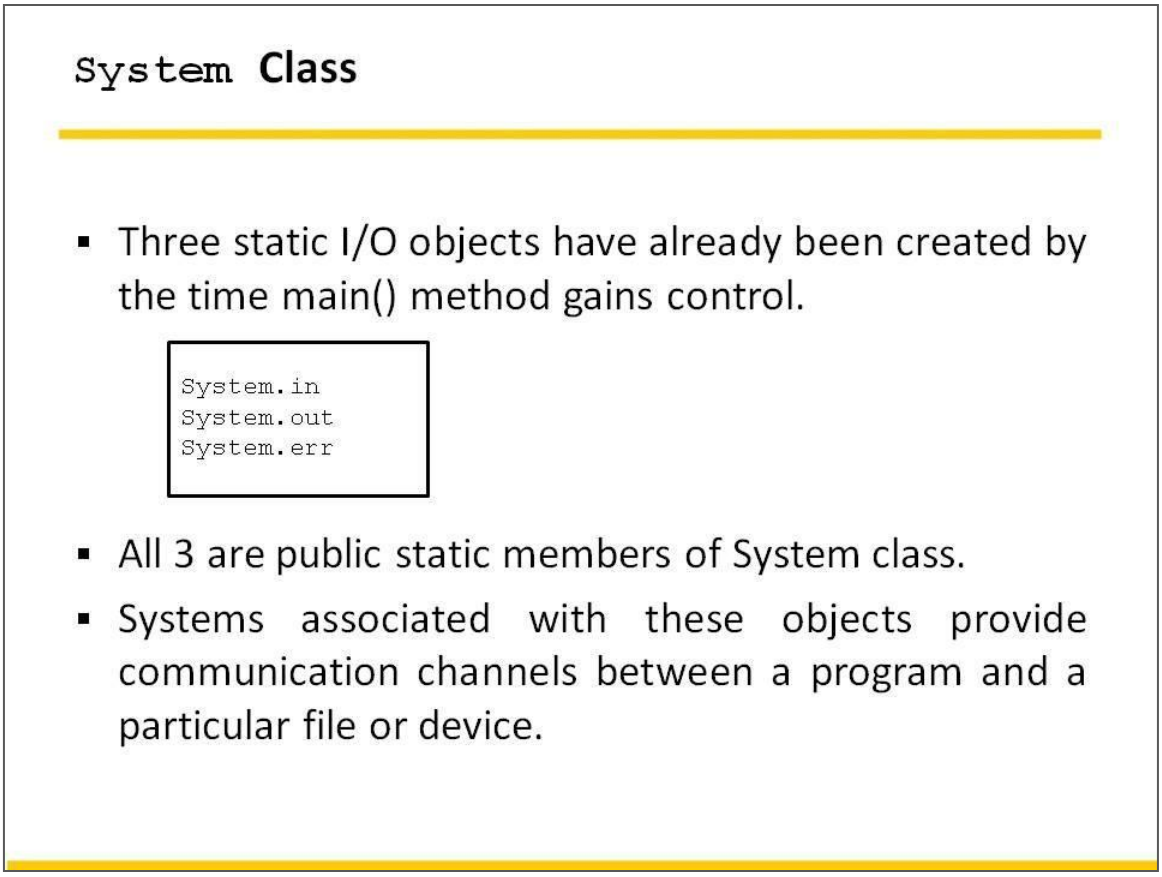
Code Example

Following code snippet explains the simple methods of a `File`.

```
import java.io.*;
class FileDemo
```

```
{
 File f;
 String fnm,path,abspath;
 long size;
  FileDemo()throws Exception
  {
   f=new File(“myFile.txt”);
     if(f.exists()==true)
      {
       System.out.println("file exists");
      }
     else
      {
        f.createNewFile();
      System.out.println("file does not exists");
      }
    System.out.println ("file name is:" + f.getName());
    System.out.println ("path is:" +f.getPath());
      System.out.println ("Absolute path
is"+f.getAbsolutePath());   if(f.canRead()==true)
       {
      System.out.println("u can read file");
       }
    if(f.canWrite()==true)
       {
      System.out.println("u can also write in file");
     }
    if(f.isAbsolute()==true)
```

```
        {
        System.out.println("file has absolute path");
        }
       size=f.length();
     System.out.println("file length is:" + size);


   }
   public static void main(String args[])throws
Exception
    {
    FileDemo fd=new FileDemo();
    }
}
```



The `System` class contains several useful class fields and methods.

The `System` class provides you with different facilities. Some of them are as follows:-

- Standard input, standard output, and error output streams.
- Access to externally defined properties and environment variables
- A means of loading files and libraries
- A utility method for quickly copying a portion of an array and so on.

The signature of the `System` class is as follows:

```
public final class System extends Object
```

Code Example

```
import java.io.*;
class ReadKeys
 {
```

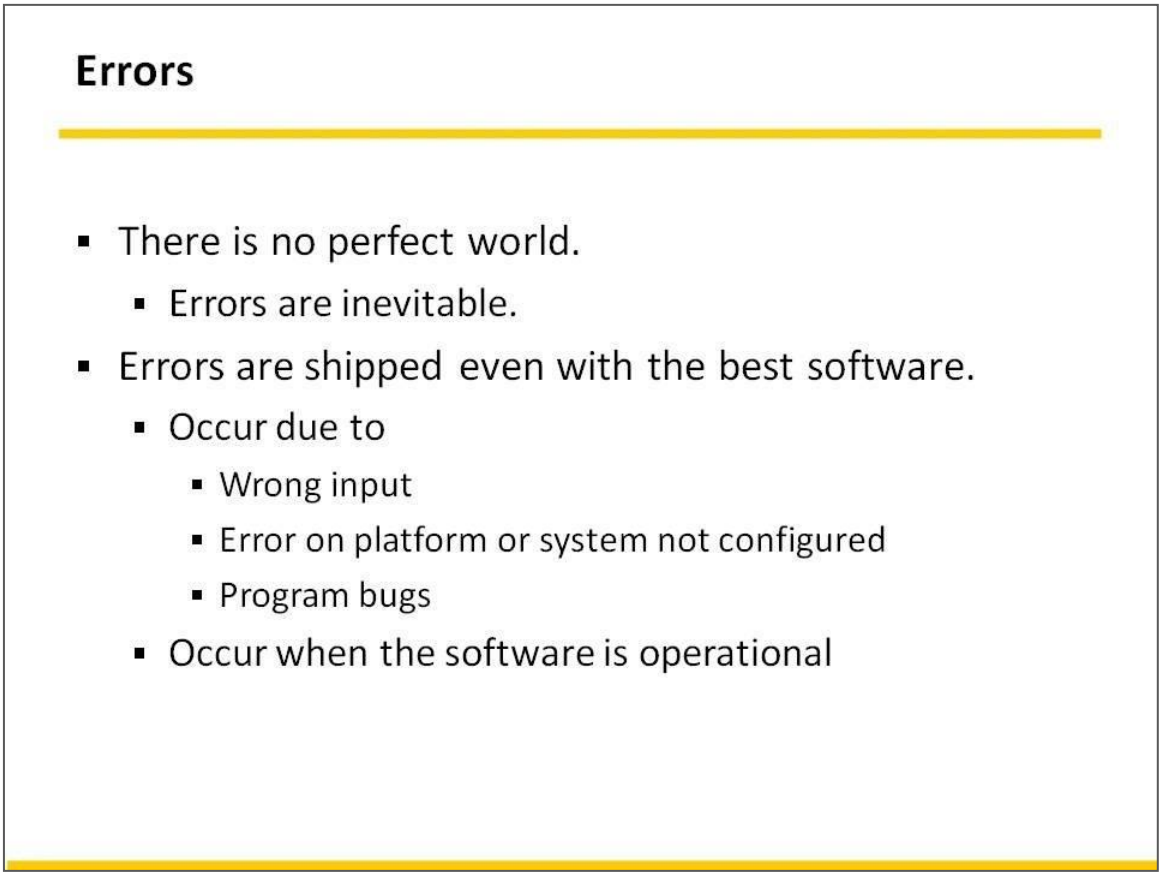
```
 public static void main(String args[])
  {
   StringBuffer sb = new StringBuffer();
   char ch;
   try
   {
   while((ch = (char) System.in.read())! = '\n')
     {
        sb.append(c) ;
     }
   }
   catch (Exception e) { ...  }
  String s = new String(sb);
  System.out.println(s);
   }
}
```

Exceptions are foreseen errors that may occur in programs. Most of the time, they can be detected and handled in the code. For example, validating user input, checking for null object references, and verifying the values returned from methods. This chapter covers limitations of C-style error handling mechanism, objected-oriented way of error handling, exception handling using the three built-in keywords `try`, `catch` and `throw`. It also covers writing user-defined exception classes.

Objectives

At the end of this chapter you will be able to

- State the limitations of C-style error handling.
- Define exception.
- Describe the mechanism of exception handling.
- Use `try`, `catch` and `finally`.
- Construct a code that handles exceptions.
- Describe the cascading of exceptions.
- Construct a user defined exception class.



Writing program code and errors go hand in hand. They can occur as a result of incorrect syntax, incorrect logic or can occur at run time.

There are different types of errors:

Logical Errors

Logical errors occur when there is something wrong with the logic used by the programmer. For example, while using the while loop, the developer has to give a terminating condition. But if the counter which leads to the terminating condition is not incremented / decremented, the loop will execute infinitely.

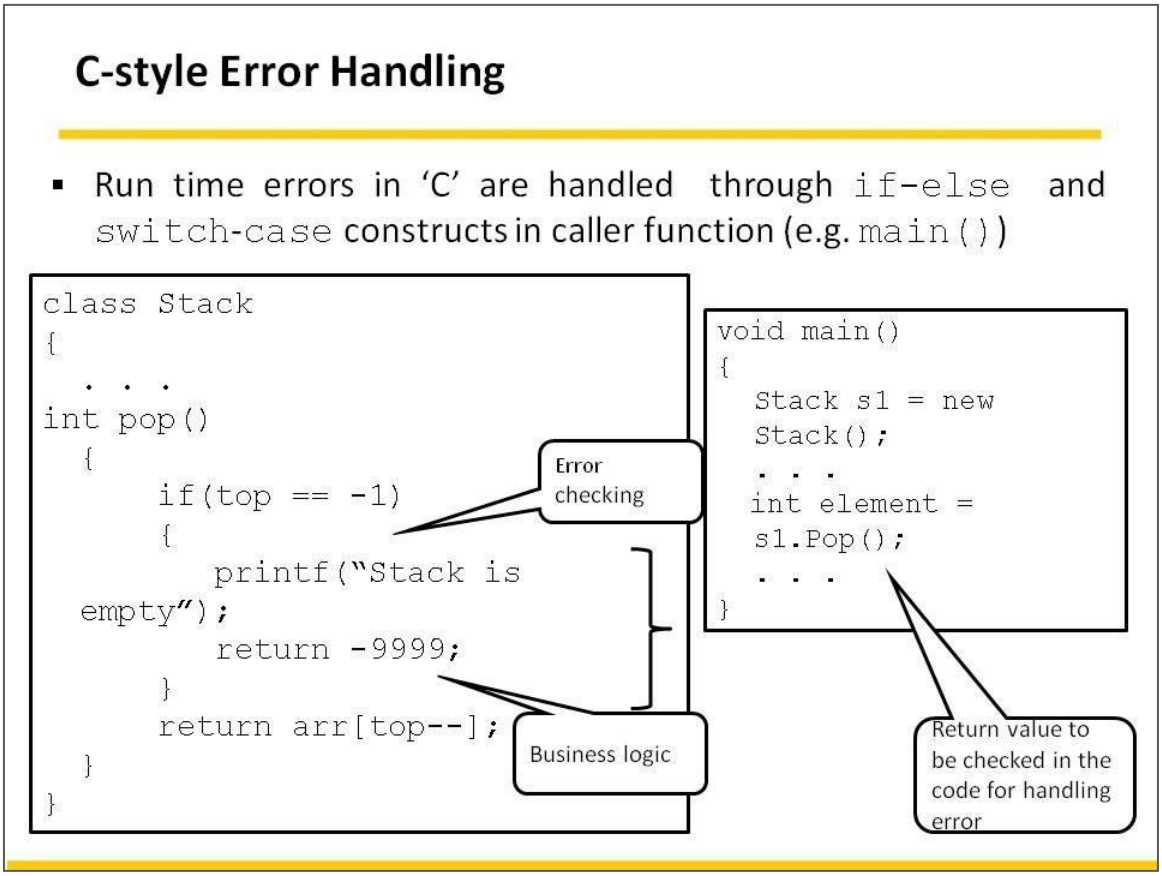
This type of errors are not detected by the compiler but can be corrected by the programmer using debug options or giving dry run to the program.

Run-time Errors

Run-time errors occur at run-time, when the program is executed. For example, when there is division to be performed and the denominator accepted from the user is zero, it will lead to a run-time error.

This type of errors can also be controlled by predicting the error and checking appropriate condition.

List some logical and run time errors.



Thoroughly tested software which works under all simulated conditions is said to be 'working software'. However, working software does not necessarily mean robust software. Robust software caters to unusual situations which might be encountered during the execution of the application. These situations are called run-time errors.

In C language, there is no separate mechanism of handling run time errors. Run time error handling is achieved by using one of the following methods:

- Return a status code with some values to indicate success or failure.
- Assign an error code to a global variable and let other functions examine it.
- Terminate the program by giving a message.

Any function that is likely to respond to a run-time error returns the error value on the occurrence. These error values are checked in a `switch case` or `if-else` statement in the caller function for taking proper action and generating appropriate error messages.

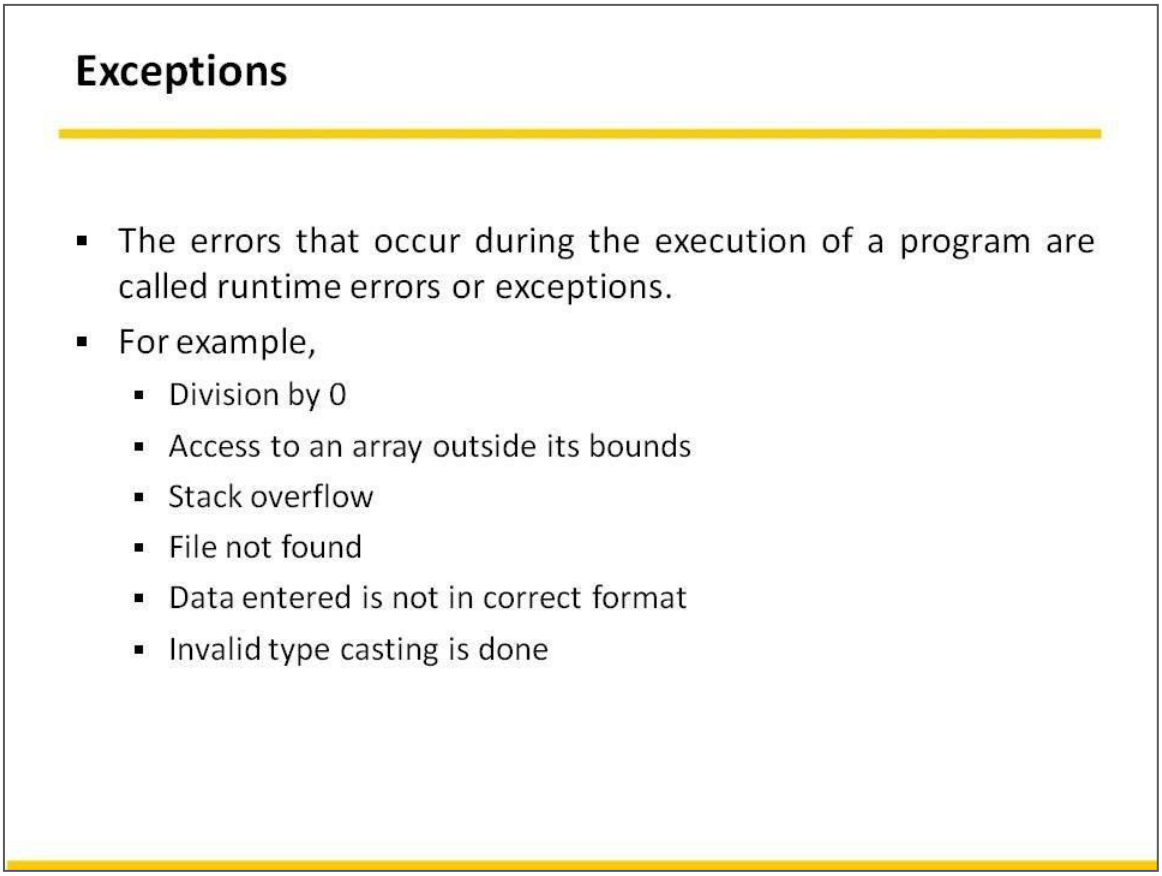
Therefore business logic gets secondary importance whereas error handling is given primary importance. This is necessary for successful execution of that program.

There is no standard way of tracking run time errors in C language.

Problems associated with C – style error handling:

- Error code returned needs to be checked again. If the returned error value is same as the data, there is a conflict.
- The business logic and error handling code are not separate.

So it is necessary that the name of the error, error message and information regarding the error bundled as one unit. It should be easy to program and extend. It should not be complicated like if-else and switch-case. Business logic and error handling code should be separate.



Java provides a structured way of handling these runtime errors; business logic and error handling code is separate. Exceptions are runtime errors which occur during the execution of a program. Some of the exceptions are:

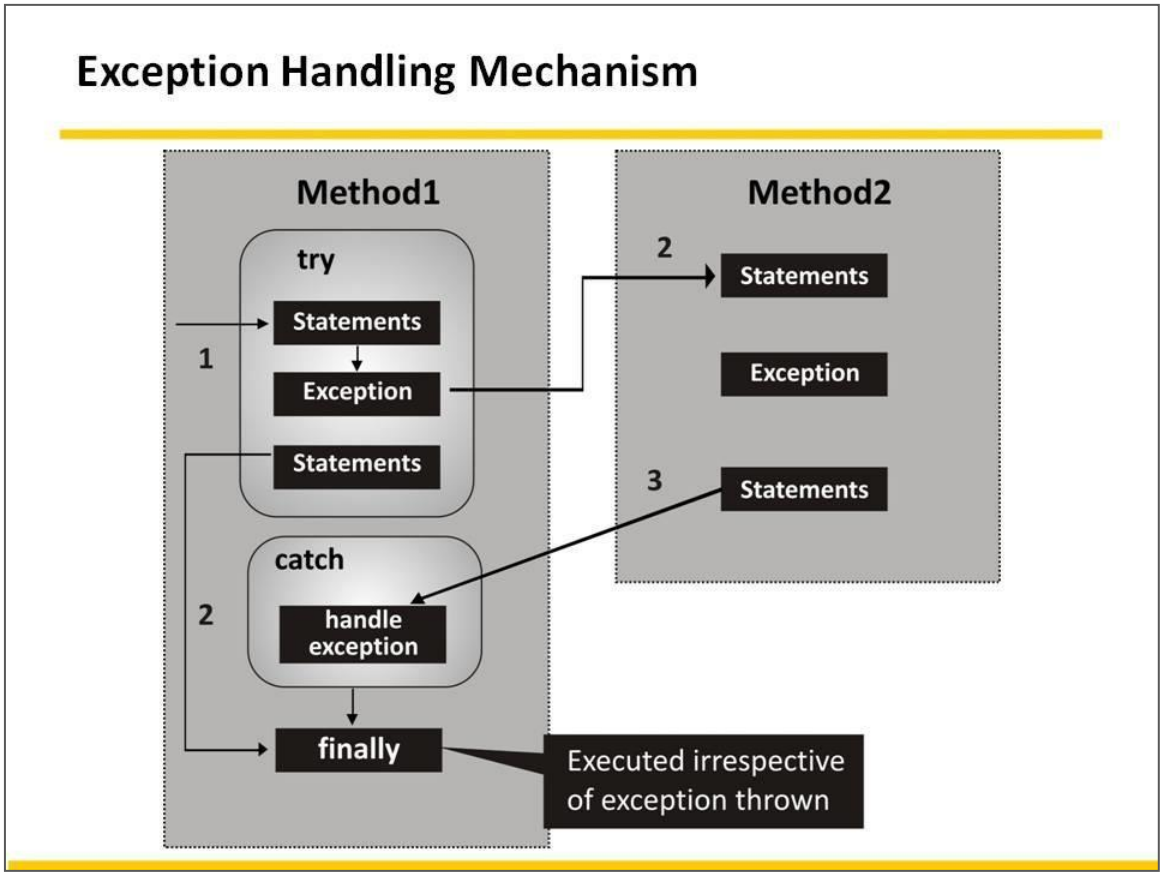
- Division by Zero exception. It occurs when the denominator is zero.
- Access to an array outside its bounds also causes a run time error to occur. This occurs when index of an array is more than its size or less than zero.
- Stack overflow exception occurs when there is no memory available to allocate on stack.
- File not found is the common exception which can occur while accessing a file from a particular location where it does not exist.

These exceptions abnormally terminate a program. To avoid this abnormal termination, Java provides an in-built language feature that takes care of all the run time anomalies. This feature is called as the exception handling mechanism. This mechanism is activated at run time whenever any erroneous situation arises in an application.

Exception handling mechanism provides a way to respond to run-time errors in the program by transferring control to special code called handler. This feature allows a clean separation between error detection code or business logic and error handling code.

The exception handling mechanism in Java language provides `try` and `catch` blocks to effectively handle runtime errors.

Exception handling code should always be a part of a robust application.



Any code that is written cannot be completely bug-free. It is the responsibility of the developer to see to it that there are minimum bugs in the code. Exception handling framework provided in Java provides following features:

- Identify the of block of code that may lead to error.
- Code is executed assuming that there are no errors. So, code that may throw an exception is written in the `try` block and the exception handling code is written in the `catch` block.
- Standard procedure to handle such errors.
- If error occurs in the `'try'` block, handling of error has to be done. The error handling code is written in `'catch'` block.

```
try
{
  FileInputStream fs = new
FileInputStream("Stud.txt");
}
```

```
catch(FileNotFoundException  ex)
{
   System.out.println("File does not exist… Do you
want to
                        create a new file?");
   . . .
}
```

- Differentiate between error messages
- There may be different errors in the `try` block that may give different error messages. It should be possible to differentiate between these error messages. Handling of individual errors should be done using multiple `catch` blocks. If there are no matching catch blocks, object of `Exception` class catches that exception.
- The code in the `try` block executes normally unless any exception occurs. When the `try` block executes normally, code in the `catch` block does not get executed.
- There can be multiple `catch` blocks for a single `try` block. Whenever an exception occurs, the system searches for the nearest `catch` block which matches the type of exception and handles it.

```
try
{
  . . .
}
catch(FileNotFoundException  ex)
{
   // . . . Some action to be taken
}
catch(SecurityException ex)
{
   // . . . Some action to be taken
}
```

- Catch all errors (cascading of exceptions)
- Object of `Exception` class catches all exceptions that are not caught by `catch` blocks. A catch clause that does not name an exception class handles any exception. This is known as "catch all" block. There can also be a user-defined `catch` block with an `Exception` type parameter to catch all exceptions. In Java, all exceptions are directly or indirectly inherited from the `Exception` class.

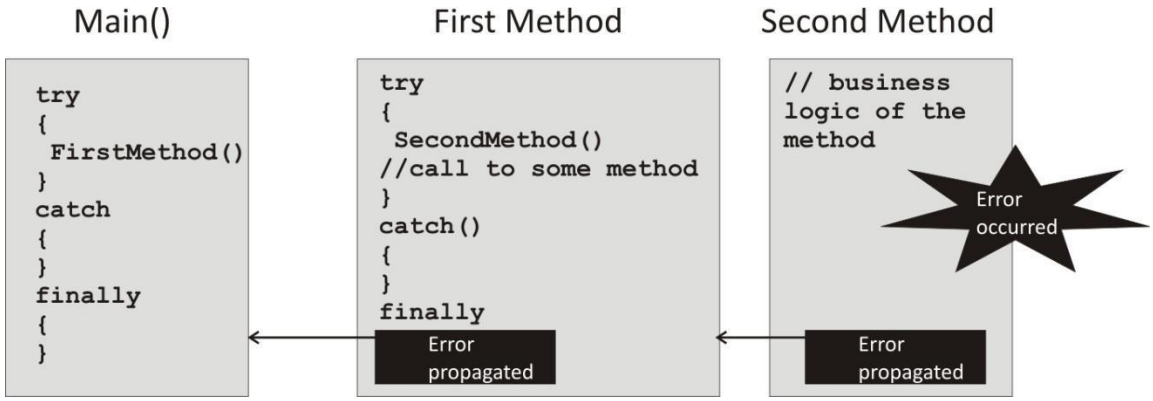
```
try
{
  FileInputStream fs = new FileInputStream(. . . );
}
catch(FileNotFoundException  ex){
   // . . . Some action to be taken
}
catch(SecurityException ex){
   // . . . Some action to be taken
}
catch(Exception ex)
{. . .
}
finally
{
   fs.close();  // release the resource
}
```

Catch all block

- Mechanism to check resource leakage. (Cleaning up)
- `try` and `catch` is a good mechanism to handle exceptions, but if an exception occurs in the application before cleaning up is done, then there has to be a way of doing it. Once the exception occurs, the control is transferred from the `try` block to catch block so, the cleanup code cannot be written inside the `try` block.
- One possibility is to place the cleanup code inside the `catch` block. This is not a good solution because each handler will then require its own copy of

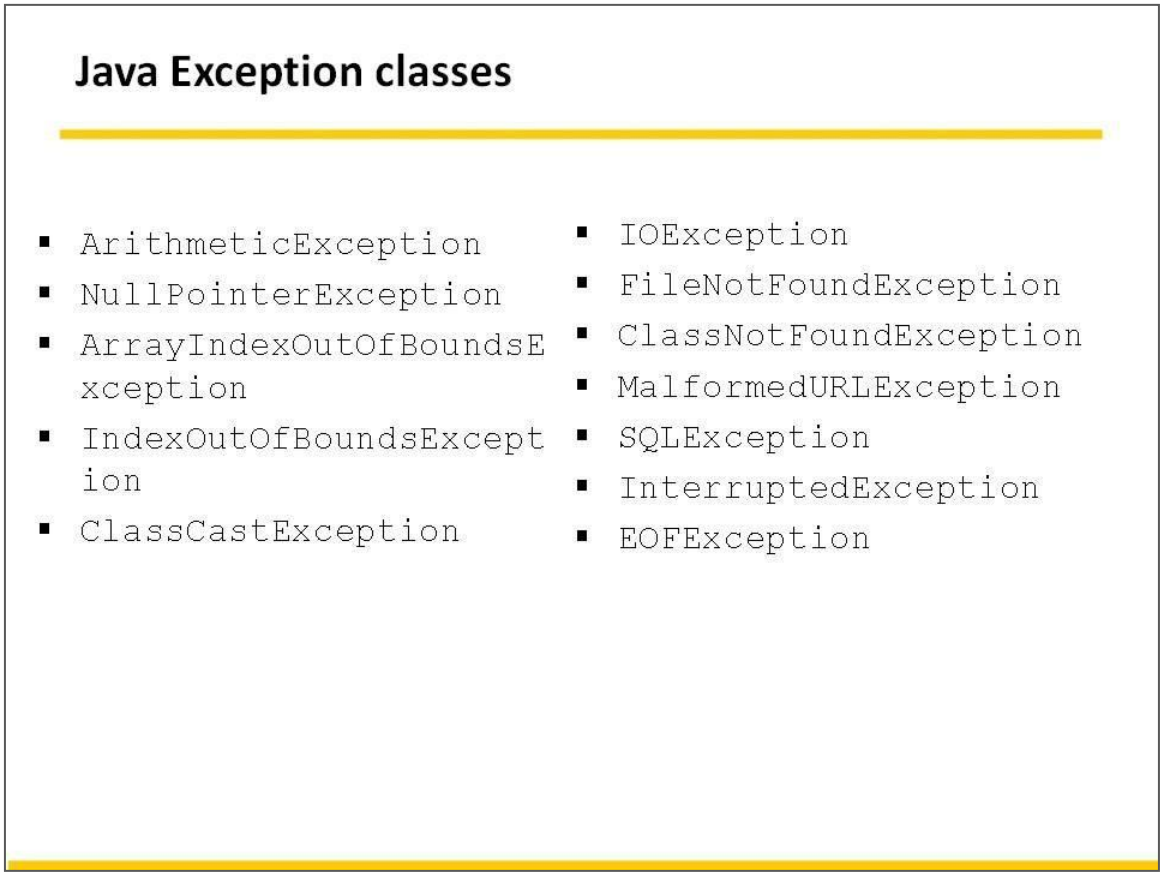
the cleanup code. A better solution is using the `finally` block. A `finally` block is always executed, whether an exception thrown or not.

- Irrespective of whether the exception is thrown, `finally` block is always executed. It is this block where one can write code to close a file or connection or release resources.
- Note that, `catch` or `finally` are optional blocks but one of them must be present otherwise compiler gives an error.
- Unhandled error
- In case the error is not handled, it is propagated up in the stack – to the calling function.



- Change the original error message
- If there is an error in catch block, then it needs to be **re-thrown** back to the caller method. This is done by using the **`throw`** keyword. A customized message that a user can understand, can then be displayed.
- Define user defined exception class
- To handle application specific exceptions, a user defined class can be created using `Exception` class.

Sometimes, in an application, no method is able to handle the exception because there is no `catch` block available. In Java, if a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against it. This is possible by including a `throws` clause in method's declaration. A `throws` clause lists the types of exceptions that a method might throw.



If a file is opened with the `java.iO.File read()` method, it can throw any of the following exceptions:

- `SecurityException`
- `ArgumentException`
- `ArgumentNullException`
- `DirectoryNotFoundException`
- `UnauthorizedAccessException`
- `FileNotFoundException`
- `NotSupportedException`

It is easy to find out which exceptions a method can raise by looking in the Java Frameworks SDK Documentation. Go to the Reference/Class Library section and look in the Namespace/Class/Method documentation for the methods used. The exceptions in the list above can be found by looking at the `read ()` method definition of the `File` class in the `java.io package`. Each exception has a hyperlink to its class definition to describe what that exception is about. Once this is known, only a handler to these exceptions needs to be placed in the code.

User-defined Exception Class

- Application specific exception class can be created using `ApplicationException` class.

```
class ApplicationException extends Exception
{
 public String message;
 public ApplicationException(String msg)
 {
    super(msg);
    message = msg;
 }
}
```

```
class MyApp
{
  . . .
  void myMethod (int ele) throws
ApplicationException
  {
     . . .
   if(some condition)
  throw new
   ApplicationException("Exception
occurred");
  }
}
```

User can define an exception class if it is required in an application. It is called user defined exception class or custom exception class. The class should be derived from `Exception` class.

A user defined exception class `ApplicationException` is defined above. `MyApp` class contains `push()` method which throws `ApplicationException` when the stack is full.

The client code is given below.

```
public static void main(String[] args)
{
  MyApp s1 = new MyApp();
  try{
      s1.myMethod(10);
      s1.myMethod(20);
```

```
       s1.myMethod(30);
      }
  catch (ApplicationException s)
    {
     System.out.println(s);
    }
}
```

If an `exception` is thrown, the calling method is automatically handed the convenience and power of Exception handling mechanism to respond to abnormal conditions. It will also be possible for the compiler to check if these are dealt with properly, since these abnormal conditions are declared in the `throws` clause of the method.

The common practice in a customized exception class is to subclass the `Exception` class. Because `exception` classes are class objects, they can have data members and methods defined within them.

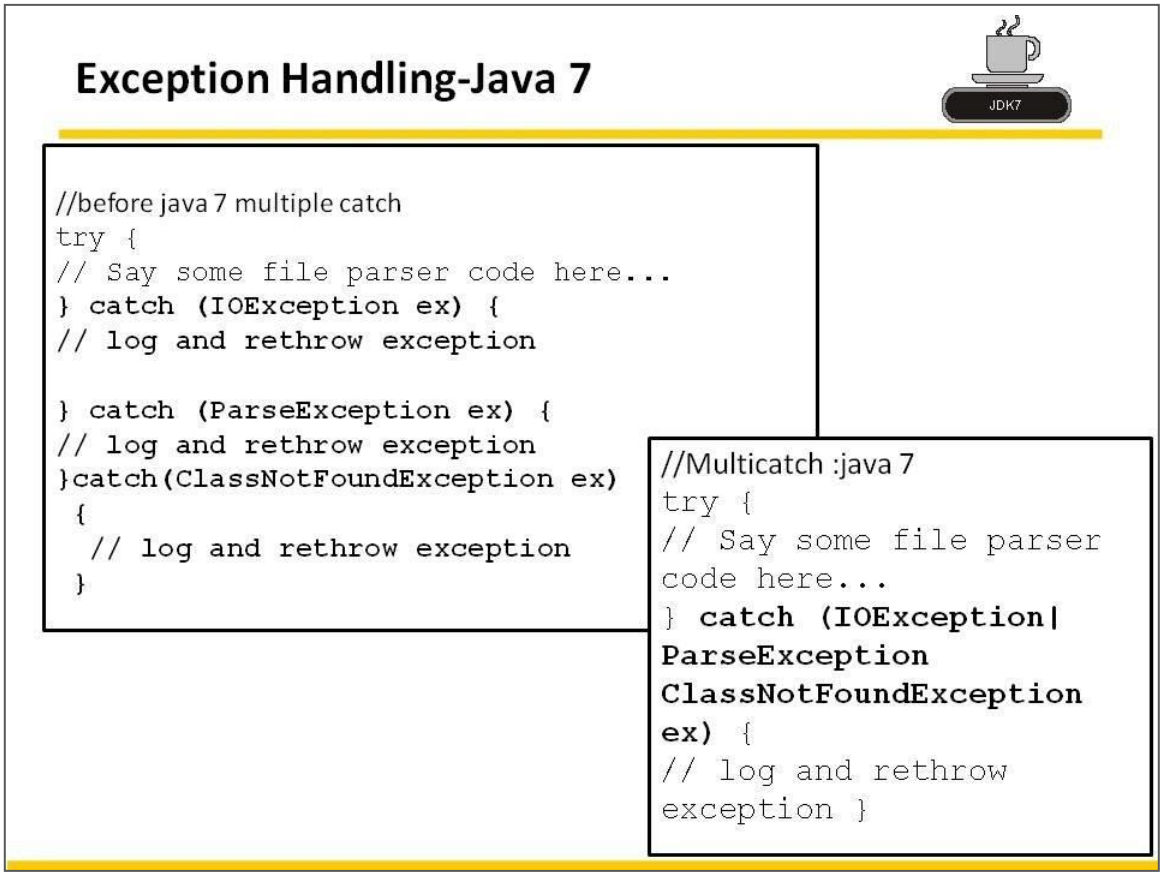
Exception Handling-Java 7

- Improved Exception Handling technique introduce by Java 7.
- This improvement adds two little improvements to exception handling:
- Multicatch : Multi exceptions types can be caught in one block
- Final Rethrow : Allows catching an exception type and its subtype and rethrowing it without having to add a throws clause to the method signature.

There is an improved in Exception handling technique introduced by Java 7.

The improvements are:

- Multicatch: can handle more than one exception in a same catch block.
- Final Rethrow: Without adding a throws clause to the method signature an exception type can be caught.



Before Java 7 is to write multiple catch blocks then to write a number of catch blocks one after another and it look messy. This is extremely inefficient and error prone.

Code Example

```
public  void myMultiCatchDemo()
{
  try { . . .}
  catch(Exception1 e) { . . .}
  catch (Exception2 |Exception3 e2) {. . .}
}
```

Java 7 has brought in a new language change.i.e Multicatch

Multicatch can handle more than one exception in the same catch block.

Exception Handling-Java 7 (ARM)

- ARM stands for Automatic Resource Management.
  - A new try-with-resources code-block.
- It automatically manages closeable resources.

```
static String readFirstLineFromFile2 (String path)
throws IOException {
   try (BufferedReader br = new BufferedReader(new
FileReader(path))
     {
         return br.readLine();
     }
}
```

The `try-with-resources` is also called as ARM (Automatic Resource Management). A resource is an object that must be closed after the program is finished. ARM or try-with-resources checks that each resource is closed at the end of the statement.

If the object implements interface called `java.lang.AutoCloseable`, which includes all objects which implement `java.io.Closeable`, it can be used as a resource.

Assertions

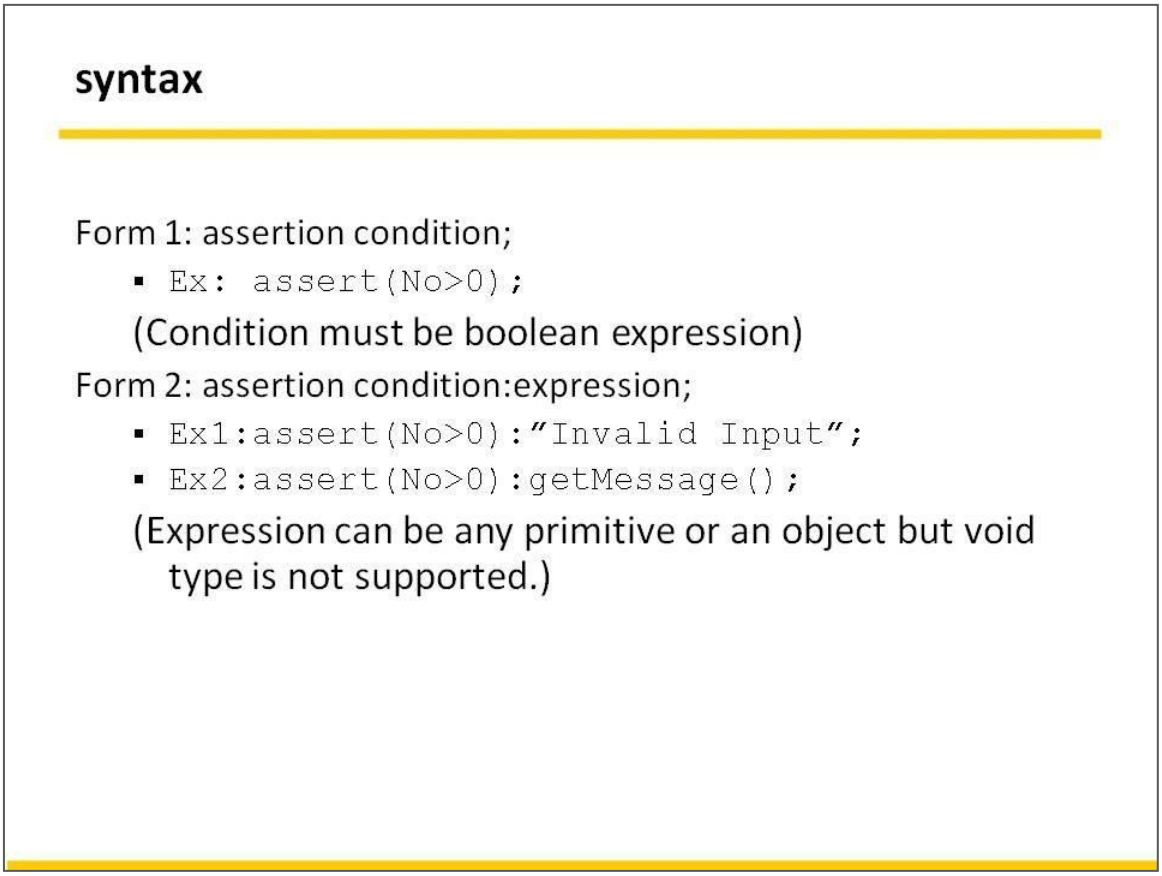
- Used to check assumptions to guarantee the application always executes complying specifications.
- Used as a debugging tool during development and testing.
- Till jdk 1.3 used as an identifier, jdk 1.4 onwards used as a keyword.
- Assertions ensure 'correctness' of application as per business rules whereas exception handling ensures 'robustness'.
- Also known as 'pre-contracting'.

To validate the assumptions of an application generally Assertions are required. It is a debugging tool during development and testing. Assertions are always boolean expressions if it returns false then application throws an error called `AssertionError`.

assert can be used as a keyword or an identifier, but not both.

To declare as assertions:

```
assert condition;
assert condition : expression;
```



The syntax of assert keyword to handle assert expressions.

Code Example

```
class AssertDemo
{
 static int val=3;
 //Ruturn an integer
 static int getnum()
 {
  return val--;
 }
 public static void main(String[] args)
 {
  int n;
```

```
  for(int i=0;i<10;i++)
    {
    n=getnum();
    assert n>0; //will fail when n is 0
    System.out.println("n is   "+n);
   }
 }
}
```

Compiling Assertions

- Java 7 compiler uses the `assert` keyword by default.
- Compiler gives an error message if it finds `assert` as identifier.
- Compile program using –source version command.
- Tells compiler in which version to compile it.

```
javac -source 1.3 OldJava.java
```

Java 7 compiler will use the assert keyword by default. The compiler will generate an error message if it finds the word `assert` used as an identifier.

If `assert` is to be used as an identifier then old compiler is required to compile the application.

So old version (jdk 1.3) code is to be confirmed that uses assert as an identifier.

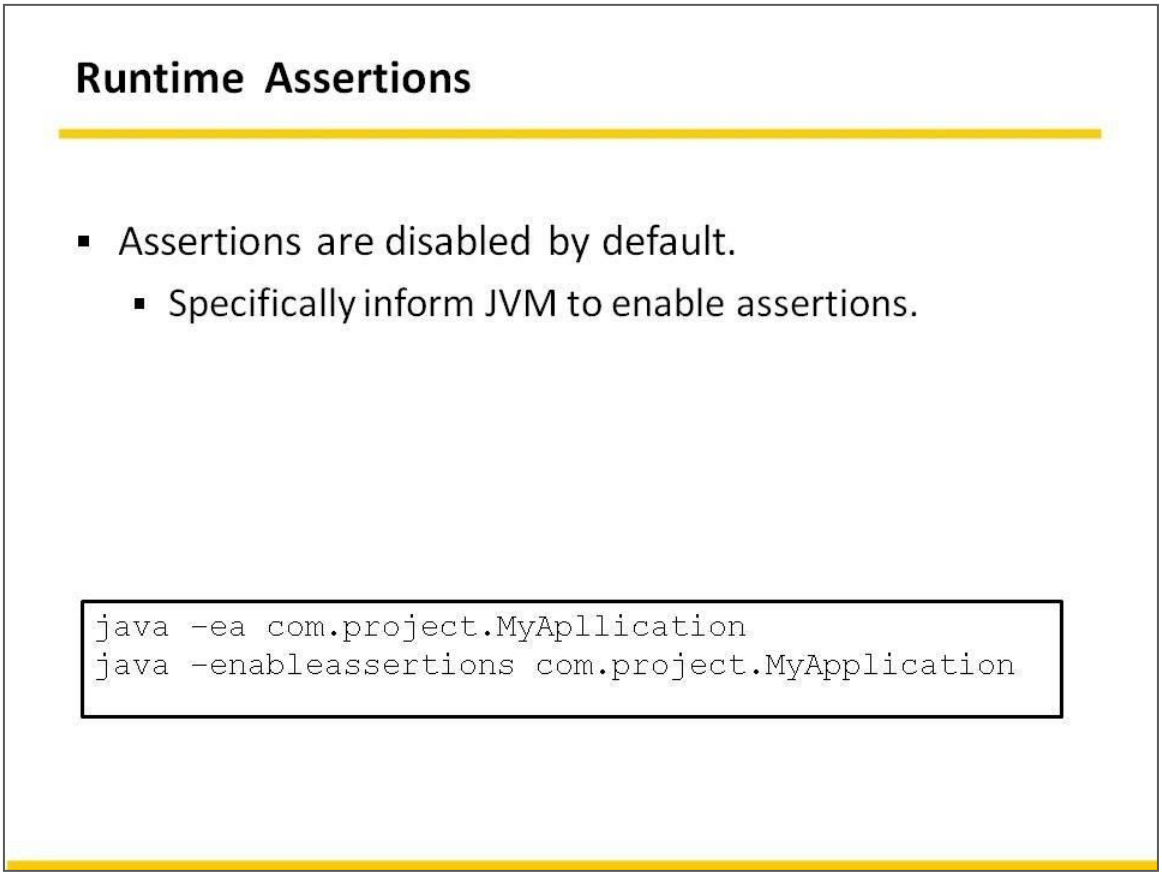
```
javac –source 1.3 AssertionDemo.java
```

This code will compile with a warning message.

If same application is compiled as

```
javac –source 1.4 AssertionDemo.java
javac –source 1.5 AssertionDemo.java
javac –source 1.6 AssertionDemo.java
```

and if assert is used as an identifier than compiler throws an error, because from jdk1.4 onwards `assert` is treated as keyword.

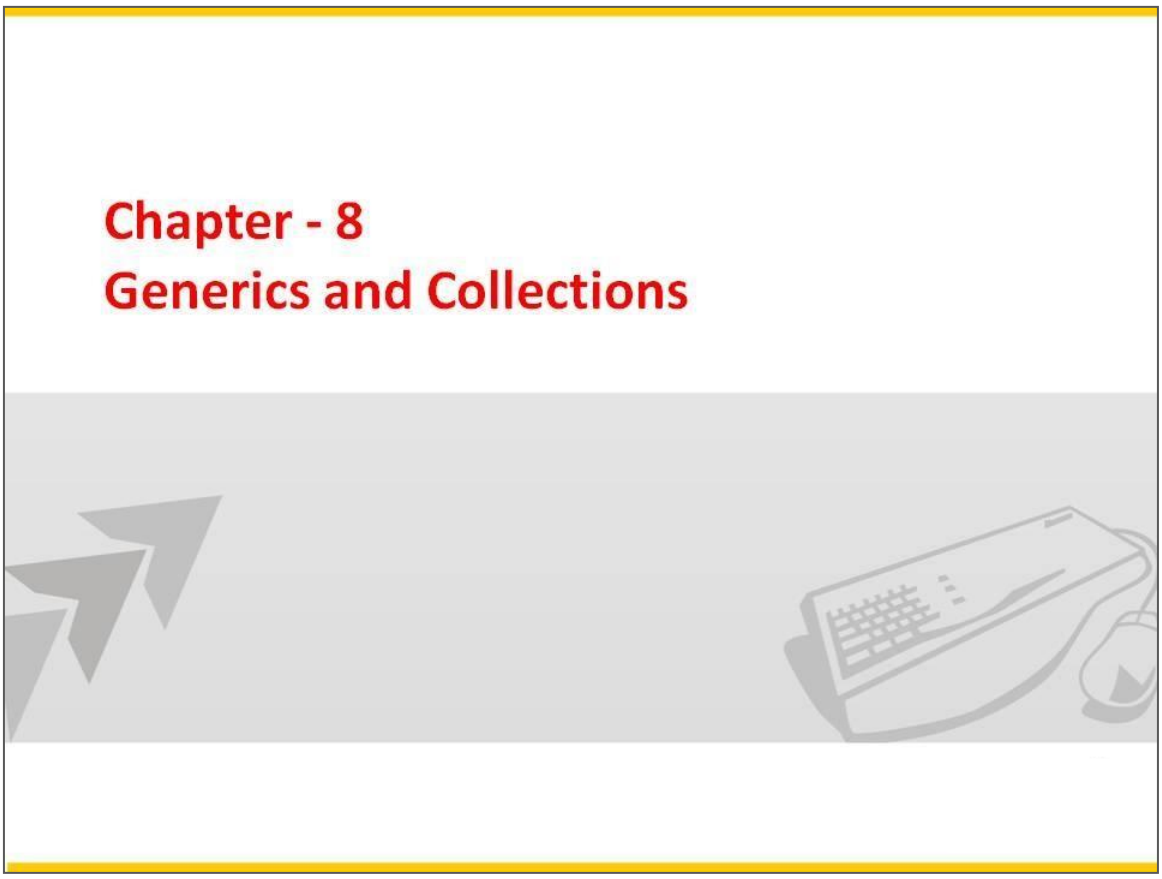


Assertions are by default disabled when application is deployed. Assertions are always in the code, but ignored by JVM. So, JVM has to be specifically informed to enable assertions.

```
//To enable Assertions
java  -ea  AssertionDemo
java -enableassertions AssertionDemo
//To disable Assertions
java  -da  AssertionDemo
java -disableassertions AssertionDemo
```

Rule for Assertions

1. Not to be used to validate command-line argument.
2. Not to be used to validate arguments of `public` methods.
3. Not to be used to call any method within it.

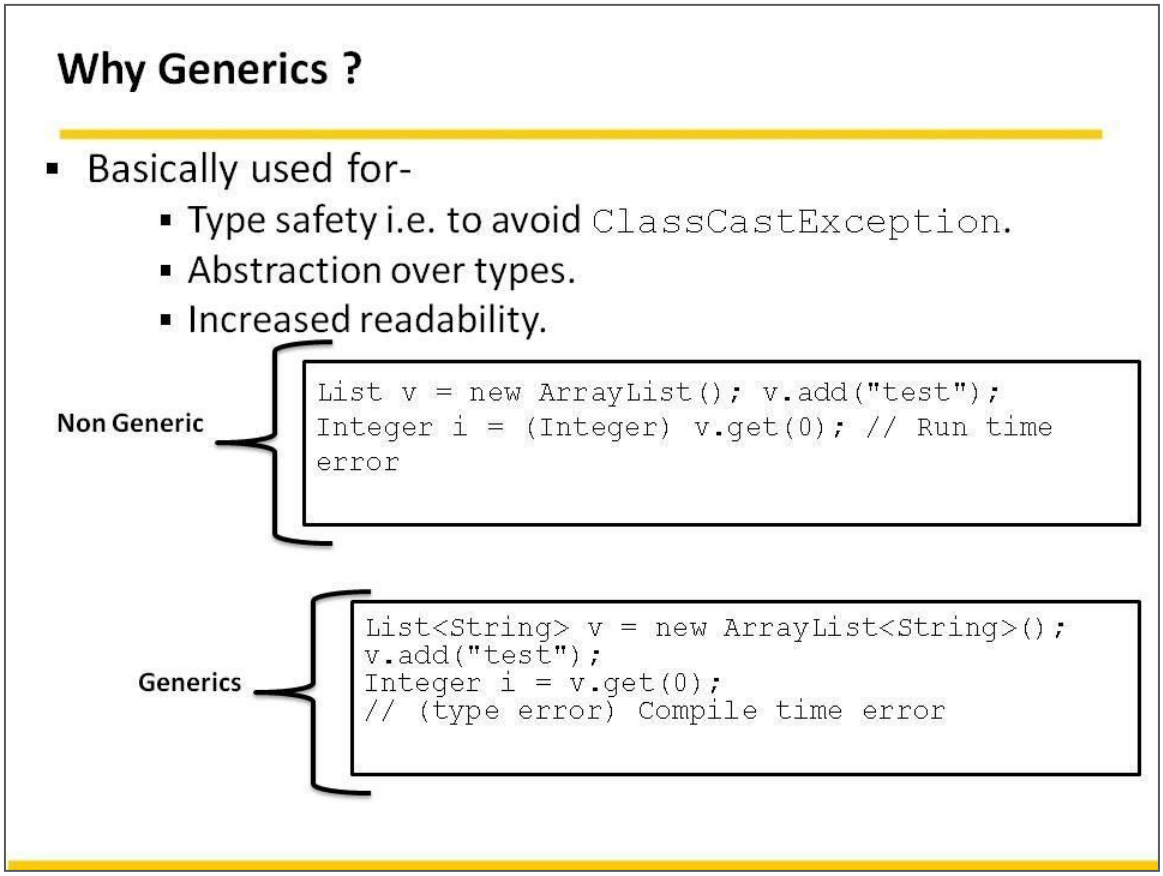


Closely related data can be handled very efficiently by grouping them together. Instead of handling individual objects, it is easier to write code to process these similar elements as a group. This group is called a collection. This chapter covers some collection classes defined in API. It covers collection interfaces which give a common behavior to the collection classes. It also focuses on generic (template) versions of these collections.

Objectives

At the end of this chapter you will be able to:

- Identify the need of collections.
- Define collections and list different collection classes in Java.
- List the collection interfaces.
- Implement `List, Set, Map, Queue` interfaces in user defined classes.
- Use iterator method to iterate through the collections.
- List the advantages of generics over their non - generic counterparts.
- Use generic classes like `List<T>`
- Use the bounded type and wildcard



There are two major disadvantages of non-generic object collections.

1. All objects in non generic collections are stored as `Object` type. If any primitive type is added to any of these collections, boxing takes place. That is, implicit conversion of value type to reference type. This hampers the performance. To retrieve this value from the collection, unboxing takes place. This requires essential type cast.
2. The non-generic object collections can store objects of different types simultaneously as shown. There is no guarantee of type safety with non-generic collections.

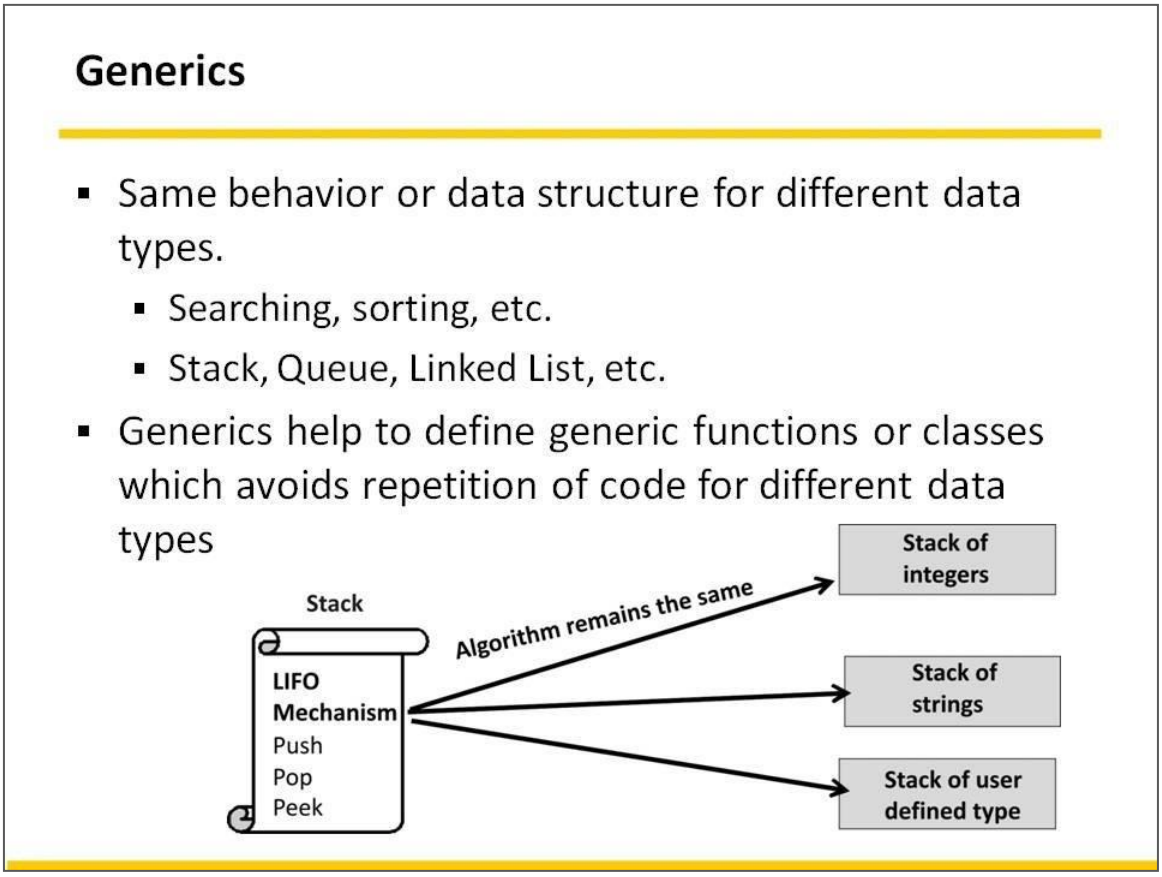
Code Example

```
ArrayList list = new ArrayList();
list.add(new Employee(10));
list.add(new Circle (3));
for (Object o :list)
```

```
{
    . . . //code here
}
```

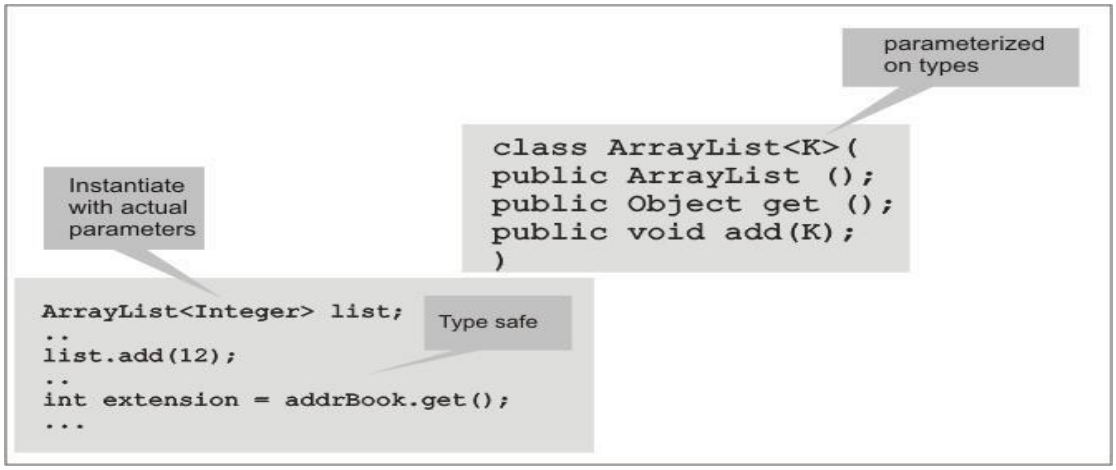
3. Moreover the garbage collector overhead increases as objects are allocated on heap.

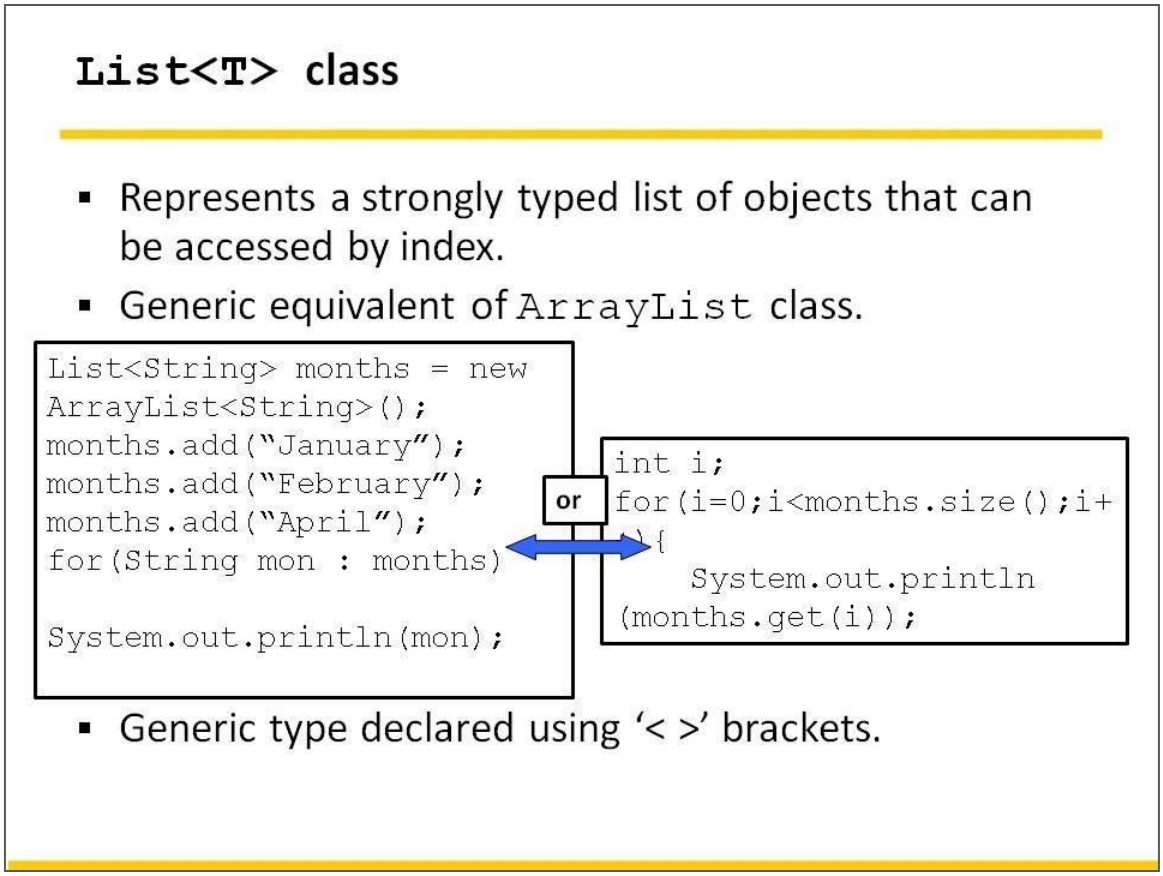
Above problems are overcome by using **generics** in Java.



Generics object collections are very useful when implementing generic constructs like searching, sorting, vectors, stacks, queues, lists, and so on. These constructs have a generic algorithm that can be implemented for any data type. Generic object collections or simply Generics in Java supports both primitive and reference types.

Data type has to be specified at the time of instantiation of generic classes, thus providing type safety. For example, `Integer` data type is specified to instantiate `ArrayList` class. The methods of `HashTable` class also take parameter of type `K`. `K` is a placeholder. Compiler generates type specific implementation. The compiler does not create a brand new implementation of the generic type. It addresses only those methods and properties of the generic type that are actually invoked. Boxing, unboxing, and casting are not required as the stored elements in the generic collection are of specified type.


parameterized on types
class ArrayList<K>(
public ArrayList ();
public Object get ();
public void add(K);
)
Instantiate with actual parameters
ArrayList<Integer> list;
..
list.add(12);
..
int extension = addrBook.get();
...
Type safe



`List<T>` interface represents a strongly typed list of objects that can be accessed by index. Its implementation is handled by a generic equivalent class `ArrayList`. Unlike an `ArrayList` class, `List<T>` interface is type safe and is also better performance wise. A reference type can also be used in place of type `T`.

```
public interface List<E> extends Collection<E>
```

`List<T>` accepts a `null` reference as a valid value for reference types and allows duplicate elements. `List<T>` class defines a list of generic methods for performing various operations on the list.

Generic class with two type parameter

- More than one type parameter can be declared in a generic type.
- To specify two or more type parameters, comma-separated list is used.

```
Map<Integer,String> months = new HashMap<Integer,String>();
months.put(0,"January");
months.put(1,"February"); months.put(3,"April");

for (String mon : months.values()){
   System.out.println(mon);
}
```

Suppose in an application month number (Integer type) and month name (String type) are to be added, it cannot handle requirements which call for two data types to be associated with the collection. When generic class is declared, it is a type T class. To handle such situations generics class with two type parameters can be declared.

The syntax is:

```
class ClassName<k,v> {}
```

```
Map<Integer, String> months=new HashMap<Integer,
String>();
months.put(0,”January”);
months.put(1,”February”);
```

Bounded Types

- When specifying a type parameter, an upper bound that declares the supperclass from which all type arguments must be derived can be created.
  - class stats<T extends Superclass>
  - e.g. class Stats<T extends Number>

```
__Bounded(){
__MyGen<Number> months =
      new __MyGen<Number>(1);
   months.print();
__MyGen<Number> months1 =
      new __MyGen<Number>(2.0);
   months1.print();
__MyGen<String> months2 =
     new __MyGen<String>("3");/*compile time error The type
String is not a valid substitute for the bounded parameter
<T extends Number> of the type __MyGen<T>*/
   months2.print();
   }
```

```
public class __MyGen<T extends
Number> {
T o;
 __MyGen(T obj){
 o=obj;
 }
 void print(){
System.out.println(o.getClass()
.getName()+" "+o.toString()); }
}
```

Generic has one advantage that provides abstraction over Types, but if abstraction over type allows adding or passing any kind of objects to erase the <T> and executes the application.

For example, a user wants to add some numbers and print the result. That number may be integer, float or double etc. so a generic type is required. At the same time, user can pass `String` type also which compiles successfully. Requirement is to add numbers, type has to restrict to number. So, bounded types come into the picture.

```
class Gen<T extends Number> {

 . . .
}
```

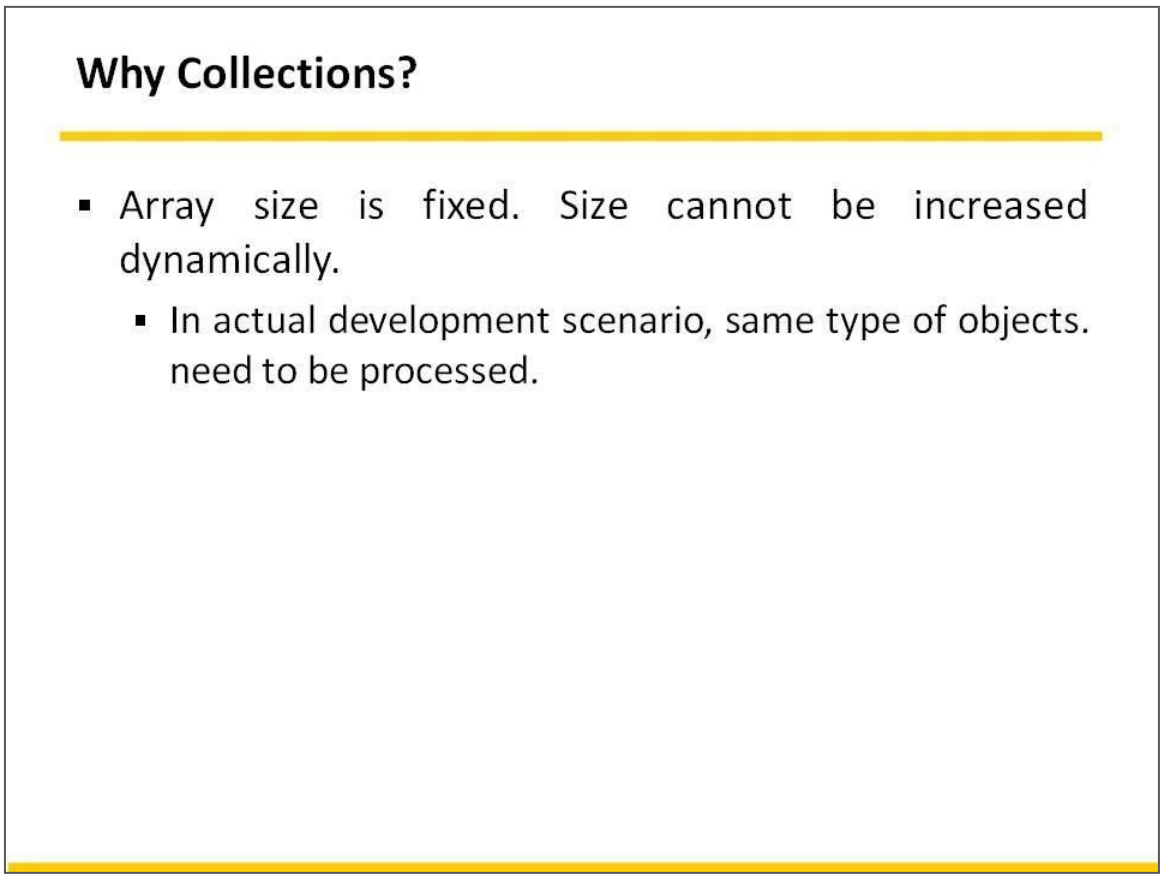
Why Wildcards?

- The wildcard argument is specified by the **?**
  - Represents an unknown type.

```
static void printCollection(Collection<?> c)
{
       for (Object o : c)
       System.out.println(o);
}

public static void main(String[] args)
{
       List<Integer> li = new ArrayList<Integer>(10);
       printCollection(li); // No Compile error
       Collection<String> cs = new Vector<String>();
       printCollection(cs); // No Compile error
}
```

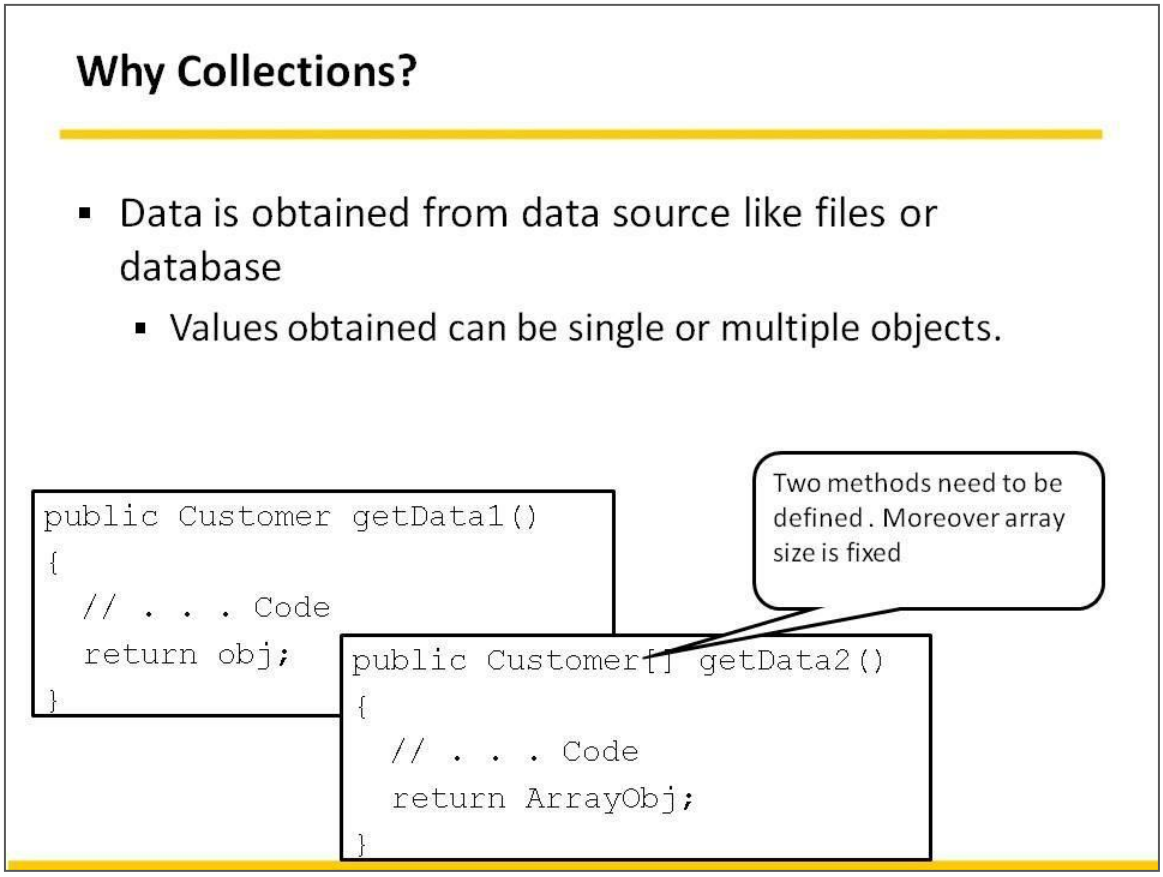
Consider the scenario, in which a generic class has a method called `sameAvg()`. The functionality of this method is to find out the average of the numbers and check that whether average is same or not. The number might be an integer, float or double which replaces generic type T at compile time. If the result returned is of type integer and needs to be compared with result type float, generic replacement under normal circumstances will not work as both the data types are different/unknown at design time. Hence to represent this unknown type the wild-card character (?) is used.

```
class SomeClass<T>
{ . . .
      boolean sameAvg(SomeClass<?> ob){
        if(average() == ob.average())
            return true;
        return false;
    }}
```



Arrays have fixed size. A set of elements can be stored in the array. The elements can be any built-in type or any user-defined type. The insertion and deletion of elements in the array is also costly in terms of performance.

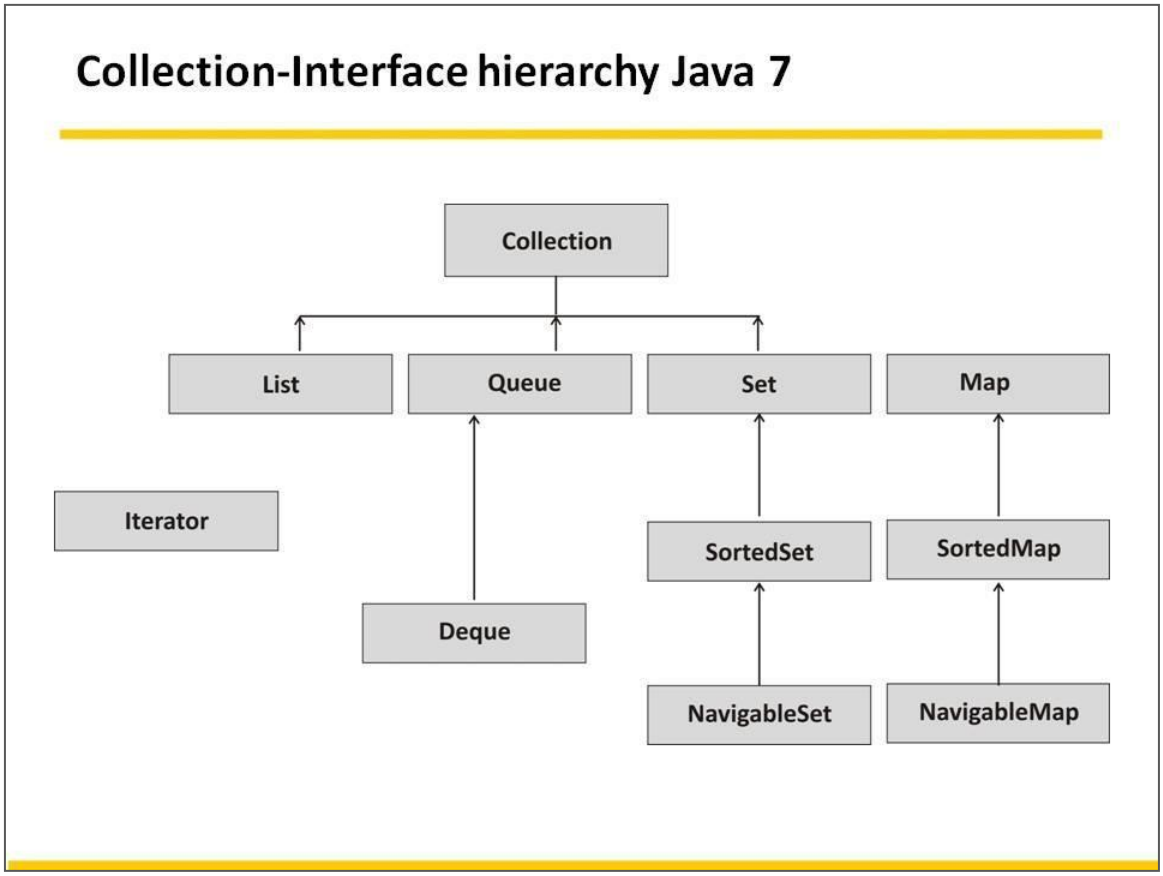
In actual development scenario, same types of objects need to be processed. They can be added and removed dynamically. So accordingly the size of the unit holding the objects should grow or shrink. Also the way these objects should be stored, can be different. There can be a requirement to store the objects sequentially, non-sequentially, sorted order, etc.



Collections come in handy as a return type in implementation of business methods. Generally when data is obtained from a data source like file or database, it can be a single object or multiple objects. When one is not sure, it is safe to define return type as collection since it can hold multiple objects. This works even if a single or no object is returned.

The advantage of defining an interface to return collection (typically the topmost class in the hierarchy) in such a way makes the client code more generic and the interface definitions do not require to be changed often. Otherwise one would end up having at least three methods with different return types like single object, array of objects and collection of objects. Just by making the method return collection eliminates the need to create the two methods.

Need of collections should be understood properly.



Collection is a data structure that can hold other objects. Collection encourages software reuse by providing convenient functionality. Collection Framework provides interfaces that define the operations performed generically on the various types of collections. `java.util` package contains specialized classes for storing and accessing the data. Collection interfaces define the operations that can performed on each type of collection.

Types of collections available in java are discussed below.

Arrays

Array class is defined in `java.util` package. Arrays can store any type of data but only one type at a time. The size of the array has to be specified at compile time. Operations such as insertion and deletion reduce the performance.

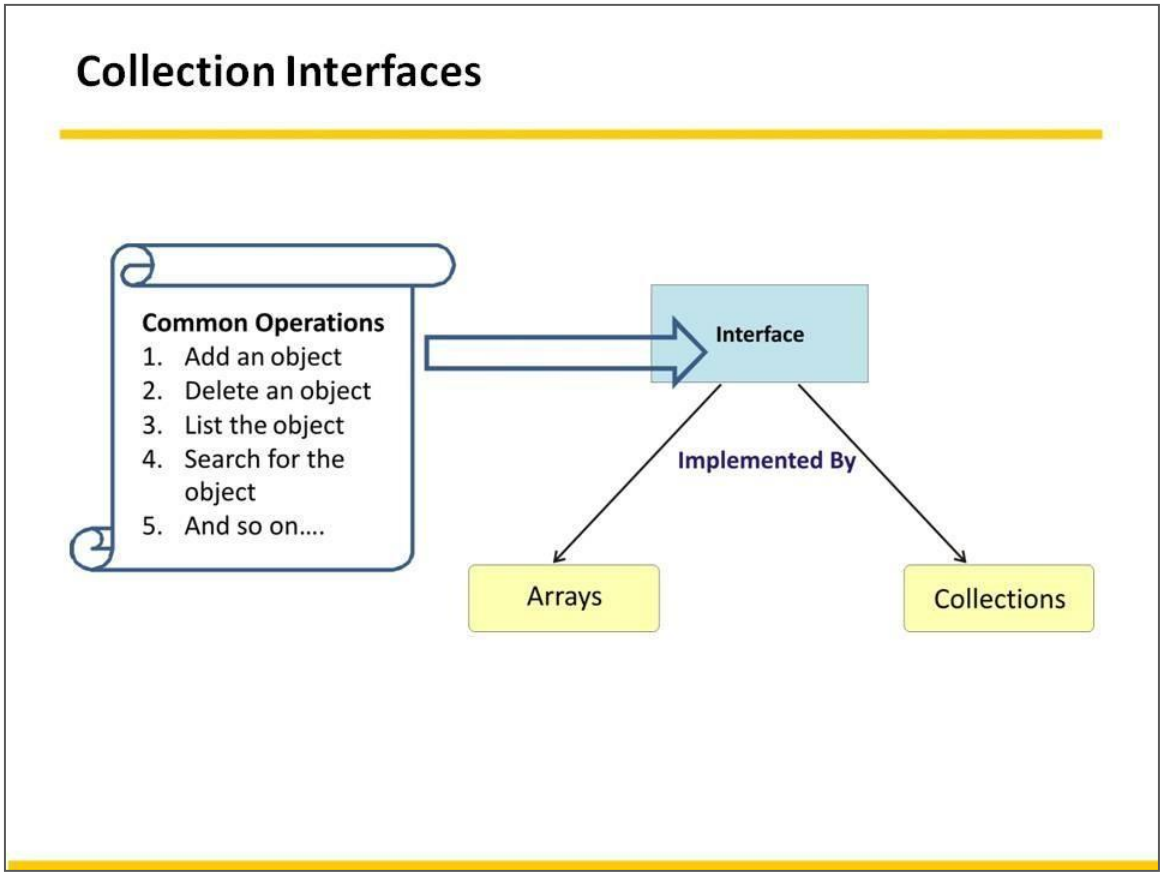
Advanced Collections

Array size is fixed at compile time. But many a times, actual number of elements can vary as per the need. Different operations are also required to process data.

These include searching, sorting, inserting, comparing, and so on. To perform these operations efficiently, the data needs to be organized in a specific way. This gives rise to advanced collections. Advanced collections are found in `java.util` package.

The super interface of all types of collections is `Collection` interface. Under the `Collection` interface `List`, `Queue` and `Set` are the three types of interfaces are offered. One more collection interface type is offered, called `Map`.

Every class in the collection has a unique feature. Some classes have sorting capability; some are based on a particular principle. For example, Stack is based on the principle, Last In-First-Out (LIFO) whereas Queue is based on First-In-First-Out (FIFO). Collections like hash table and dictionary are stored as key-value pairs. `ArrayList` is similar to an array but it can grow dynamically whenever a new element is added.



There are few basic operations that are possible on any collection.

1. Adding objects in the collection dynamically
2. Removing objects dynamically
3. Listing objects in the collection by iterating through it
4. Searching specific object in the collection.
5. Retrieving a particular object from collection.
6. Deleting a particular object from the collection

These common operations are defined in the form of interfaces. All the collection classes implement these interfaces to get the required functionality. The implementation can be different for different classes in the hierarchy. For example, adding objects in an array is different than adding them in `ArrayList` collection. `ArrayList` grows and shrinks dynamically. Therefore both array and `ArrayList` implement the functionality in their own way. Both collections need the functionality of iteration using `foreach` (enhanced for) loop to display the list of objects contained in them.

Collection Interfaces

- Allow collections to support a different behavior

Interface	Description
List	Represents a collection of objects that is accessed by an index. An Ordered collection .
Set	Represents a collection that contains no duplicate elements.An Order is not fix.
Map	Represents a collection of key-and-value pairs .
Queue	Represents a collection designed for holding elements prior to processing.
Iterator	Supports a simple iteration over collection.
Comparator	Comparison function, which imposes a total ordering on some collection of objects.

Collection interfaces provide common functionalities that the collection classes should implement to support different behavior. For example, `List` interface allows a collection to behave like a list and be indexed. There is a generic and a non-generic version of these interfaces. Custom collection classes can be written by implementing one or more interfaces. Some of the collection interfaces are explained below.

`List, List<E>` Interface

`List` interface represents a collection of objects that can be accessed by an index. Resizable-array implementation of the `List` interface implements all optional list operations, and permits all elements, including `null`. A class which implements this interface must provide implementation `add(),clear(),contains(),indexof(),remove()` methods.

Method	Description
`boolean add(E e)`	Appends the specified element to the end of this list (optional operation).
`void clear()`	Removes all of the elements from this list (optional operation).
`boolean contains(Object o)`	Returns true if this list contains the specified element.
`int indexOf(Object o)`	Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
`Iterator<E> iterator()`	Returns an iterator over the elements in this list in proper sequence.
`E remove(int index)`	Removes the element at the specified position in this list (optional operation).
`int size()`	Returns the number of elements in this list.

`Set, Set<E>` Interface

`Set` interface represents a collection that contains no duplicate elements. In `Set` interface an order is not fixed. A class which implements this interface must provide implementation a`dd(),clear(), contains(),remove()` methods.

Method	Description
`boolean add(E e)`	Adds the specified element to this set if it is not already present (optional operation).
`void clear()`	Removes all of the elements from this set (optional operation).
`boolean contains(Object o)`	Returns true if this set contains the specified element.
`Iterator<E> iterator()`	Returns an iterator over the elements in this set.
`boolean remove(Object o)`	Removes the specified element from this set if it is present (optional operation).
`int size()`	Returns the number of elements in this set (its cardinality).

`Map`, `Map<K,V>` Interface

`Map` interface represents a collection of key-and-value pairs. A class which implements this interface must provide implementation `put()`,`clear()`,`size()`,`get()`,`remove()` methods.

Method	Description
`V put(K key, V value)`	Associates the specified value with the specified key in this map (optional operation).
`void clear()`	Removes all of the mappings from this map (optional operation).
`V get(Object key)`	Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
`V remove(Object o)`	Removes the mapping for a key from this

	map if it is present (optional operation).
`int size()`	Returns the number of key-value mappings in this map.

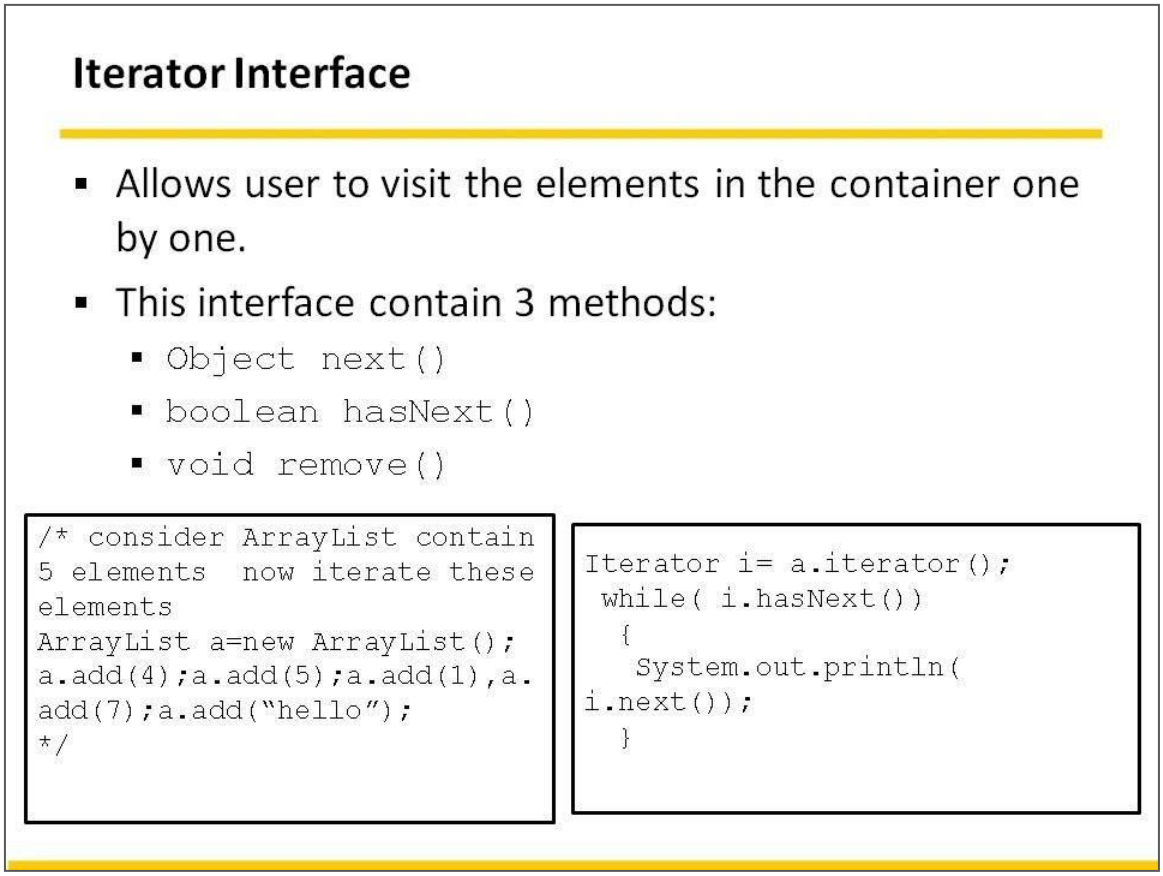
`Queue, Queue<E>` Interface

`Queue` interface represents a collection designed for holding elements prior to processing. `Queue` is support for all basic operations of Collection but it provides additional insertion, extraction and inspection operations. Each of these methods exists in two forms: One throws an exception if operation fails and the other returns special value i.e. null or false depending upon operation. The latter form of the insert operation is designed specifically for use with capacity-restricted `Queue` implementations. It is typically FIFO (First in First Out) collection. A class which implements this interface must provide implementation like `add(), element(), offer(), peek(), poll(),remove()` methods.

Method	Description
`boolean add(E e)`	Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions, returning true upon success and throwing an `IllegalStateException` if no space is currently available.
`E element()`	Retrieves, but does not remove, the head of this queue.
`boolean offer(E e)`	Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions.
`E peek()`	Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.

`E poll()`	Retrieves and removes the head of this queue, or returns null if this queue is empty.
`E remove()`	Retrieves and removes the head of this queue.

Collection Interfaces can be implemented in user defined classes too to get the functionality.



Iterator takes the place of Enumeration in the Java collections framework. `Iterator` interface supports a simple iteration over collection. Iterators differ from enumerations in two ways:

- Iterators allow the caller to remove elements from the underlying collection during the iteration with well-defined semantics.
- Method names are improved.

Iterator<E> API

Following are the important methods of Iterator interface.

Method	Description
`boolean hasNext()`	Returns true if the iteration has more elements.
`E next()`	Returns the next element in the iteration.
`void remove()`	Removes from the underlying collection the last element returned by the iterator (optional operation).

Code Example

Following code snippet explains some of the important methods of Iterator interface.

```
MyClass myObj=new MyClass();
Vector c = new Vector();
v.addElement(myObj);
Iterator it = c.iterator();
while(it.hasNext())
 {
  myObj = (MyClass)it.next();
  . . .
```

Comparator Interface

- A comparison function, which imposes a total ordering on a collection of objects.
- Comparable is used to
  - Provide an ordering for collections of objects that do not have a natural ordering.
  - Control the order of certain data structures (such as sorted sets or sorted maps).
- Comparators can be passed to a `sort` method
  - `Collections.sort()`
  - `Arrays.sort()` to maintain the control over the sort order.
- Method is `compare(Object o1,Object o2)`

`Comparator` interface gives a capability of sorting the data. To implement `Comparator` interface you have to write implementation for `compare()` has to be written .

`Comparator.compare()` method returns an `int` whose meaning is same as `Comparable.compareTo()`.

To sort employees on a specific criteria `compareTo()` method of `Comparable` interface is sufficient, but it is inadequate when sorting criteria consists of more than one characteristics for such instances `Comparator` interface provides a method called `compare`, which allows customized comparing mechanism.

Comparator<E> API

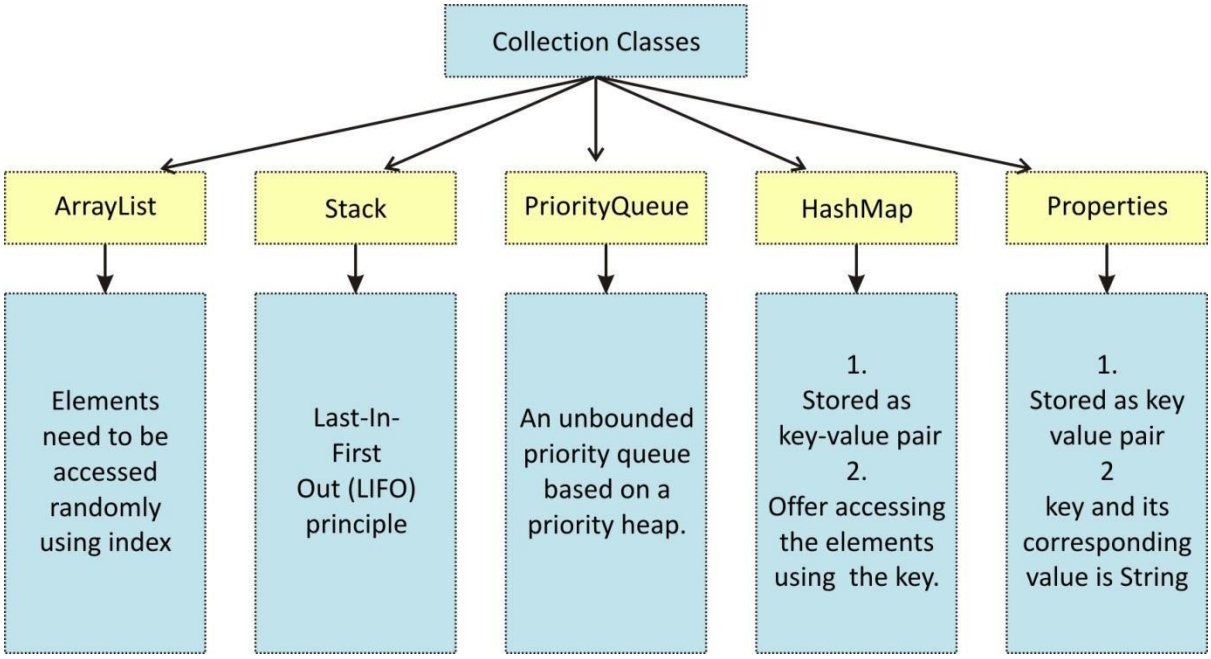
Following are some of the important methods of `Comparator` Interface.

Method	Description
`int compare(T o1,T o2)`	Compares its two arguments for order.
`boolean equals(Object obj)`	Indicates whether some other object is "equal to" this comparator.

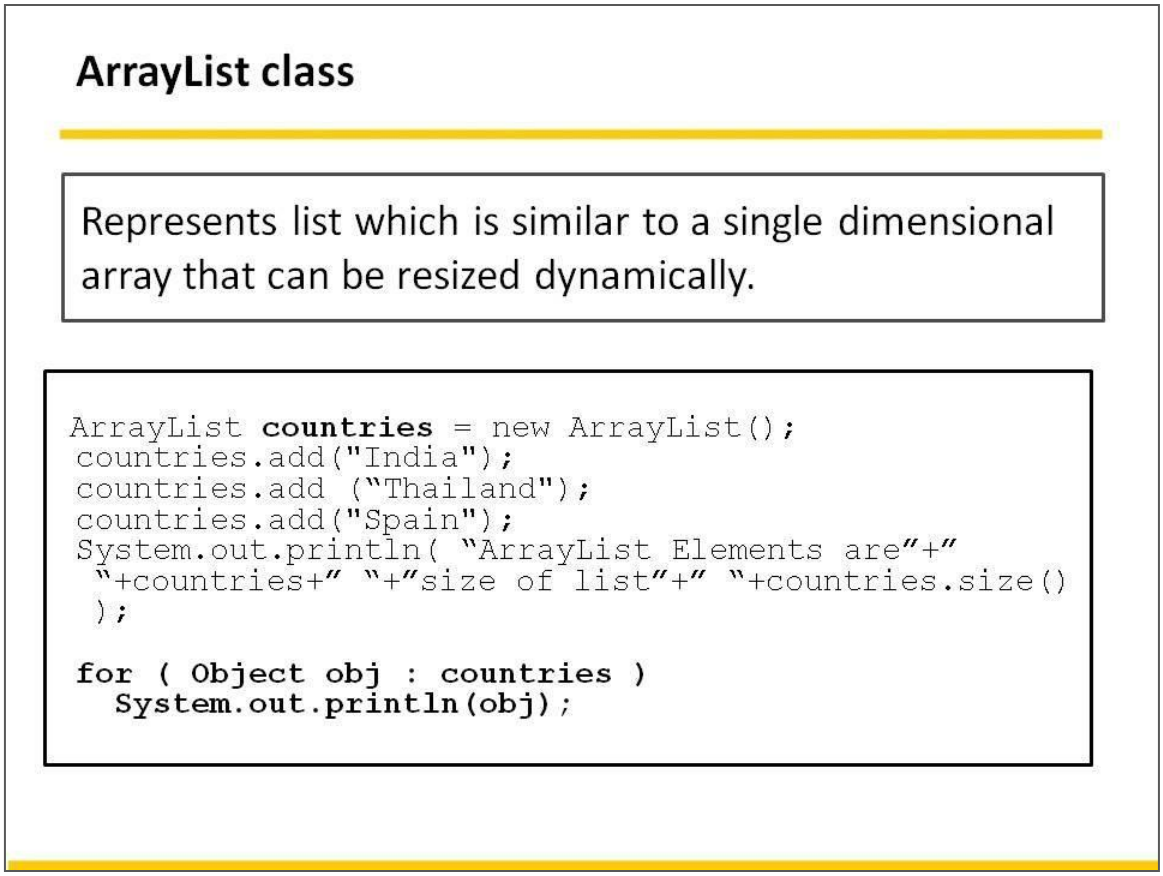
Code Example

Following code snippet explains some of the important methods of `Comparator` interface.

```
class EmpCompartor implements Comparator<Employee>
{
 public int  compare(Employee e1, Employee e2)
 {
  int emp1Age = e1.getAge();
  int emp2Age = e2.getAge();
   if(emp1Age > emp2Age)
    return 1;
   else if(emp1Age < emp2Age)
    return -1;
   else
    return 0;
  }
}
```



There are many collection classes in the collection framework. These classes provide specific functionality in terms of the way they are stored, accessed, updated, etc.



`ArrayList` is resizable-array implementation of the `List` interface. Implements all optional list operations, and permits all elements, including `null`. In addition to implementing the List interface, this class provides methods to manipulate the size of the array that is used internally to store the list. It is an ordered collection (by index).

`ArrayList<E>` API

Following are some of the important methods of `ArrayList` class.

Method	Description
`ArrayList()`	Constructs an empty list with an initial capacity of ten.
`ArrayList(Collection<? extends E> c)`	Constructs a list containing the elements of the specified collection, in the order they are returned by

	the collection's iterator.
`ArrayList(int initialCapacity)`	Constructs an empty list with the specified initial capacity.
`boolean add(E e)`	Appends the specified element to the end of this list.
`void add(int index, E element)`	Inserts the specified element at the specified position in this list.
`boolean contains(Object o)`	Returns true if this list contains the specified element.
`E get(int index)`	Returns the element at the specified position in this list.
`Iterator<E> iterator()`	Returns an iterator over the elements in this list in proper sequence.
`E remove(int index)`	Removes the element at the specified position in this list.
`Object[] toArray()`	Returns an array containing all of the elements in this list in proper sequence (from first to last element).

Code Example

Following code snippet explains some of the important methods of `ArrayList` class.

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

```
  {
   List l=new ArrayList ();
   l.add(3);
   l.add (1);
   l.add(3);
   l.add (4.0f);
   l.add("aaa");
   System.out.println ("ArrayList elements are : "+"
"+l);
  }
}
```

`ArrayList` and other collections are a convenient as a parameter to methods or returning from methods.

Stack class

- Represents a simple Last-In-First-Out (LIFO) non-generic collection of objects.

```
Stack numStack = new Stack();
numStack.push(10);
numStack.push(20);
numStack.push (30);
System.out.println("Element removed:"+
  numStack.pop());
```

Stack is a data structure which is based on Last-In-First-Out (LIFO) principle. For example, stack of coins, books, and so on. A book is always added to the top of the stack.

So, the element is added and removed from the top of the stack. The mechanism to add an element is known as 'Push' and to remove an element is known as 'Pop'. A stack takes a `null` reference as a valid value. It can also take duplicate values. Stack can grow dynamically.

Stack<E> API

The following table shows some of the methods of `Stack` class.

Method	Description
`Stack()`	Creates an empty Stack.
`boolean empty()`	Tests if this stack is empty.
`E peek()`	Looks at the object at the top of this stack without removing it from the stack.

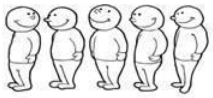
`E pop()`	Removes the object at the top of this stack and returns that object as the value of this function.
`E push(E item)`	Pushes an item onto the top of this stack.
`int search(Object o)`	Returns the 1-based position where an object is on this stack.

Code Example

Following code snippet explains some of the important methods of `Stack` class.

```
Stack numStack = new Stack();
numStack.push(10);
numStack.push(20);
numStack.push (30);
System.out.println("Element removed:"+ numStack.pop());
```

PriorityQueue



- An unbounded priority queue based on a priority heap.
- A priority queue relying on natural ordering also does not permit insertion of non-comparable objects.
- A priority queue does not permit null elements.
- Has an internal capacity governing the size of an array used to store the elements on the queue.

This class was introduced in Java 5. `LinkedList` implements the `Queue` interface; basic operation can be handled by `LinkedList` only. The purpose of `PriorityQueue` is to create to "priority-in, priority-out". `PriorityQueue` elements are ordered either by natural order or according to a Comparator.

PriorityQueue<E> API

The following table shows some of the methods of `PriorityQueue` class.

Method	Description
`PriorityQueue()`	Creates a `PriorityQueue` with the default initial capacity (11) that orders its elements according to their natural ordering
`PriorityQueue(Collection<? extends E> c)`	Creates a `PriorityQueue` containing the elements in the

	specified collection.
`PriorityQueue(int initialCapacity)`	Creates a `PriorityQueue` with the specified initial capacity that orders its elements according to their natural ordering.
`boolean add(E e)`	Inserts the specified element into this `PriorityQueue`.
`void clear()`	Removes all of the elements from this `PriorityQueue`.
`Comparator<? super E> comparator()`	Returns the `Comparator` used to order the elements in this queue, or null if this queue is sorted according to the natural ordering of its elements.
`boolean contains(Object o)`	Returns true if this queue contains the specified element.
`boolean offer(E e)`	Inserts the specified element into this priority queue.
`E peek()`	Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.
`E poll()`	Retrieves and removes the head of this queue, or returns null if this queue is empty.
`boolean remove(Object o)`	Removes a single instance of the specified element from this queue, if it is present.
`int size()`	Returns the number of elements in this collection.

Code Example

Following code snippet explains some of the important methods of `PriorityQueue` Class.

```
import java.util.*;
 public class PriorityQueueDemo
  {
   PriorityQueue stringQueue;
   void queueMethod()
     {
      stringQueue = new PriorityQueue();
      stringQueue.add ("b");
      stringQueue.add ("abcd");
      stringQueue.add ("abc");
      stringQueue.add ("a");
    stringQueue.add ("xyz");
      while( stringQueue.size() > 0)
      System.out.println( stringQueue.remove());
    }
    public static void main(String[] args)
     {
       PriorityQueueDemo p=new PriorityQueueDemo ();
       p.queueMethod();
      }
 }
```

Read about various scenarios where different collections can be used.