

Object Oriented Programming**Code: EC604A Contact: 3L Credits: 3****Prepared by: Ms. Palasri Dhar****Course Objective:**

Understand the basic concepts of object oriented programming

Able to develop solution using object oriented concept to real life problems.

Course Outcomes:

At the end of this course

Graduates will be able to understand the key concepts of object oriented programming and have an ability to design OO programs and appreciate the techniques of good design;

Graduates will be able to program presciently in an OO programming language;

Graduates will be able to have an ability to use application libraries, in this case the use of the java API;

Graduates will be able to familiar with good programming practices such as testing, debugging, documentation and version control;

Object oriented design [10 L]

Concepts of object oriented programming language, Major and minor elements, Object, Class, relationships among objects, aggregation, links, relationships among classes-association, aggregation, Generalization and Specialization. Difference between OOP and other conventional programming – advantages and disadvantages. Class, object, message passing, inheritance, encapsulation, polymorphism

Basic Java Program Execution [2L]

Structure of a Java program. Steps for executing a java program. JDK, JRE, JVM, API, byte code.

Class & Object properties [6L]

Basic concepts of java programming – advantages of java, data types, access specifiers, operators, control statements & loops, array, creation of class, object, constructor, finalize and garbage collection, use of method overloading, this keyword, use of objects as parameter & methods returning objects, call by value & call by reference, static variables & methods, garbage collection, nested & inner classes, basic string handling concepts- String (discuss charAt() , compareTo(), equals(), equalsIgnoreCase(), indexOf(), length() , substring(), toCharArray() , toLowerCase(), toString(), toUpperCase() , trim() , valueOf() methods) & StringBuffer classes (discuss append(), capacity(), charAt(), delete(), deleteCharAt(), ensureCapacity(), getChars(), indexOf(), insert(), length(), setCharAt(), setLength(), substring(), toString() methods), concept of mutable and immutable string, command line arguments, basics of I/O operations – keyboard input using BufferedReader & Scanner classes.

Reusability properties[8L] – Super class & subclasses including multilevel hierarchy, process of constructor calling in inheritance, use of super and final keywords with super() method, dynamic method dispatch, use of abstract classes & methods, interfaces, default and static method in interface, Creation of packages, importing packages, member access for packages. Meta class.

Exception handling & Multithreading [6L] – Exception handling basics, different types of exception classes, use of try & catch with throw, throws & finally, creation of user defined exception classes. Basics of multithreading, main thread, thread life cycle, creation of multiple threads, thread priorities, thread synchronization, inter-thread communication, deadlocks for threads, suspending & resuming threads.

Java File Handling [4L]

Basic file handling operations: Reading and Writing files.

Applet Programming (using swing) [4L] – Basics of applet programming, applet life cycle, difference between application & applet programming, parameter passing in applets, concept of delegation event model and listener, I/O in applets, use of repaint().

Module 1:

Object oriented design

1.1 Object Oriented Paradigm:

The object-oriented paradigm has gained great popularity in the recent decade. The primary and most direct reason is undoubtedly the strong support of encapsulation and the logical grouping of program aspects. These properties are very important when programs become larger and larger.

The underlying, and somewhat deeper reason to the success of the object-oriented paradigm is probably the conceptual anchoring of the paradigm. An object-oriented program is constructed with the outset in concepts, which are important in the problem domain of interest. In that way, all the necessary technicalities of programming come in second row.

Objects belong to *classes*. Typically, all the objects in a given class will have the same kinds of behavior.

Classes are usually arranged in some kind of *class hierarchy*. This hierarchy can be thought of as representing a "*kind of*" relation. For example, a computational model of the University might need a class person to represent the various people who make up the University. A sub-class of person might be a student; students are a kind of person. Another sub-class might be professor. Both students and professors can exhibit the same kinds of behaviour, since they are both people. They both eat drink and sleep, for example. But there are kinds of behaviour that are distinctive: professors pontificate for example.

1.2 History of Object Oriented Programming:

- The first object-oriented language was Simula (Simulation of real systems) that was developed in 1960 by researchers at the Norwegian Computing Center.
- In 1970, Alan Kay and his research group at Xerox PARC created a personal computer named Dynabook and the first pure object-oriented programming language (OOPL) - Smalltalk, for programming the Dynabook.
- In the 1980s, Grady Booch published a paper titled Object Oriented Design that mainly presented a design for the programming language, Ada. In the ensuing editions, he extended his ideas to a complete object-oriented design method.
- In the 1990s, Coad incorporated behavioral ideas to object-oriented methods.

1.3 Important Features of Object Programming:

Object-oriented programming (OOP) is a programming paradigm based upon objects (having both data and methods) that aims to incorporate the advantages of modularity and reusability. Objects, which are usually instances of classes, are used to interact with one another to design applications and computer programs.

The important features of object-oriented programming are:

- Bottom-up approach in program design
- Programs organized around objects, grouped in classes
- Focus on data with methods to operate upon object's data
- Interaction between objects through functions
- Reusability of design through creation of new classes by adding features to existing classes

1.4 Class

A class represents a collection of objects having same characteristic properties that exhibit common behavior. It gives the blueprint or description of the objects that can be created from it. Creation of an object as a member of a class is called instantiation. Thus, object is an instance of a class.

The constituents of a class are:

- A set of attributes for the objects that are to be instantiated from the class. Generally, different objects of a class have some difference in the values of the attributes.

Attributes are often referred as class data.

- A set of operations that portray the behavior of the objects of the class. Operations are also referred as functions or methods.

An object is a real-world element in an object-oriented environment that may have a physical or a conceptual existence. Each object has:

- Identity that distinguishes it from other objects in the system.
- State that determines the characteristic properties of an object as well as the values of the properties that the object holds.
- Behavior that represents externally visible activities performed by an object in terms of changes in its state.

Objects can be modeled according to the needs of the application. An object may have a physical existence, like a customer, a car, etc.; or an intangible conceptual existence, like a project, a process, etc.

1.5 Relationship among Classes

1.5.1 Aggregation

Aggregation is a special case of association. A directional association between objects. When an object ‘has-a’ another object, then you have got an aggregation between them. Direction between them specified which object contains the other object. Aggregation is also called a “Has-a” relationship.



Figure 1.2

Aggregation is whole-part relationship between an aggregate, the whole, and its parts. This relationship is often known as a has-a relationship, because the whole has its parts. For example, when you think of workers making up a team, you can say that a team has workers. Aggregation is shown using a hollow diamond attached to the class that represents the whole. The filled-in diamonds represent composition. As a UML rule, aggregation is used only with binary associations.

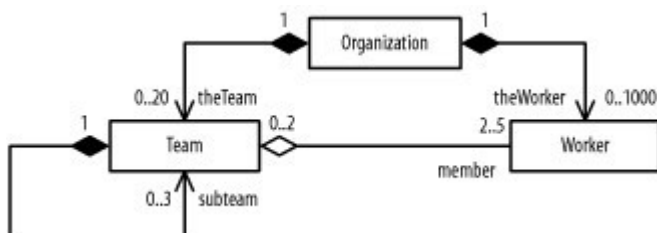


Figure 1.3

1.5.2 Link

A link represents a connection through which an object collaborates with other objects.

Rumbaugh has defined it as “a physical or conceptual connection between objects”. Through a link, one object may invoke the methods or navigate through another object. A link depicts the relationship between two or more objects.

A link is a specific relationship between objects. For example, the project management system involves various specific relationships, including specific manage, lead, execute, input, output, and other relationships between specific projects, managers, teams, work products, requirements, systems, and so forth. A link is an instance of an association, and the UML supports different types of links that correspond to the different types of associations.

The general rules for representing links in a UML diagram are as follows:

- Label links with their association names, and underline the names to show that they are specific instances of their respective associations.
- Ensure that link ends are consistent with their corresponding association ends.
- Translate association multiplicity into one or more specific links between specific objects.

1.5.2.1 Binary links

A binary link, which is a specific relationship between two objects, is shown as a solid-line path connecting the two objects in a UML object diagram. For example, a specific worker is related to specific units of work and work products in the project management system. A link may have its association name shown near the path (fully underlined), but links do not have instance names.

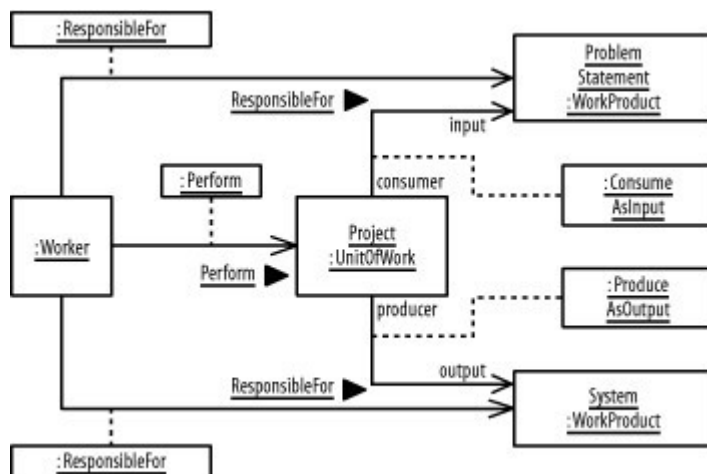


Figure 1.3

1.5.2.2 N-ary links

An n-ary link, a relationship between three or more objects, is shown as a large diamond with solid-line paths from the diamond to each object in a UML object diagram. For example, the utilization of a specific worker involves the worker, the worker's specific units of work, and the

worker's specific work products in the project management system. A link may have its association name shown near the path, and because a link is specific, its association name should be fully underlined. However, links do not have instance names. As a UML rule, aggregation, composition, and qualifiers may not be used with n-ary links.

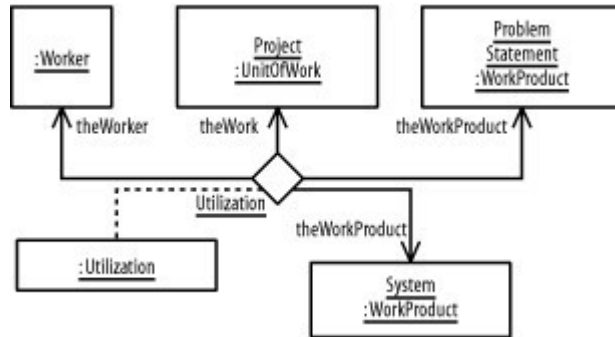


Figure 1.4

1.5.2.3 Link Objects

A link object is a specific instance of an association class, and thus has all the characteristics, including structural and behavioral features, defined by the association class. Structural features include attribute values and perhaps other links. Behavioral features include operations and methods, which are shared by all the links of an association class. Whenever an association has a related association class, each of its links has a corresponding link object. This link object defines attribute values for the link's structural features. In addition, the behavioral features defined by the link's association class apply to the link objects. In a UML object diagram, a link object is shown as an object rectangle attached by a dashed-line path to its link path in a binary link, or attached to its link diamond in an n-ary link. As with all UML elements representing specific objects or links, link object names should be fully underlined.

1.5.2.4 Link Ends

A link end, similar to an association end, is an endpoint of a link and connects the link to an object. A link end may show its association end's rolename, navigation arrow, aggregation or composition symbol, and values for its association end's qualifiers.

a. Rolenames

A link end's rolename must match its association end's rolename. For example, Figure 1.4 shows that a Worker is responsible for a WorkProduct. The specific association used is ResponsibleFor;

b. Navigation

Likewise, a link end's navigation must match its association end's navigation. For example, the arrows on the two ResponsibleFor links point to instances of WorkProduct(Fig-1.8). c.

Multiplicity

Multiplicity is shown only on association ends. This is because an association describes the multiplicity between two or more classes of objects. A link however, is between specific objects. Thus, in an object diagram, multiplicity manifests itself in terms of a specific number of links pointing to a specific number of discrete objects. (0..*) WorkProduct objects.

d. Aggregation

Aggregation is shown using a hollow diamond,

Figure 1.5 shows three teams named Eagle, Falcon, and Hawk. Jonathan, Andy, Nora, and Phillip are on the Eagle team, while Nora and Phillip are also on the Hawk team.

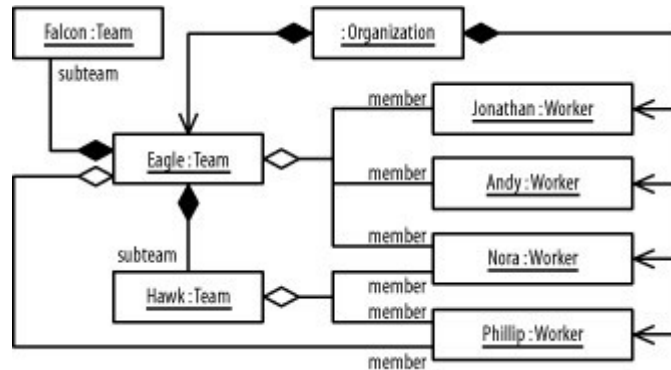


Figure 1.5

e. Composition

Composition may be shown using a filled diamond or graphical nesting Figure shows that the two teams, Falcon and Hawk, are subteams of the Eagle team. In addition, the filled-in diamond next to the Organization class indicates that all the individuals on these teams belong to the same organization, and that the Eagle team itself belongs to the organization.

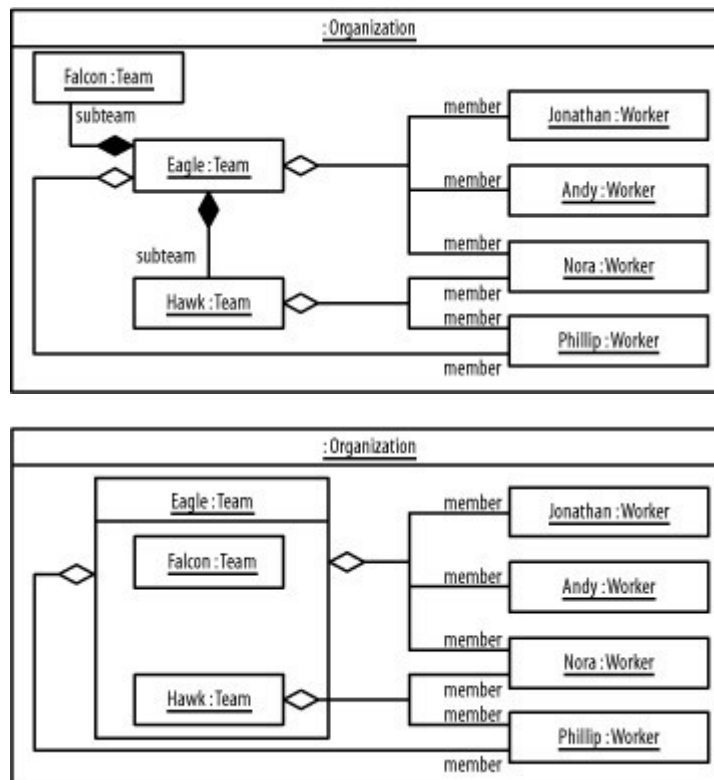


Figure 1.6

f. Qualifiers

Values for link qualifiers have the same notation as for object attribute values. Figure 1.7 shows how qualifier values associate a project with its problem statement (named Problem Statement) and system (named PM-System).

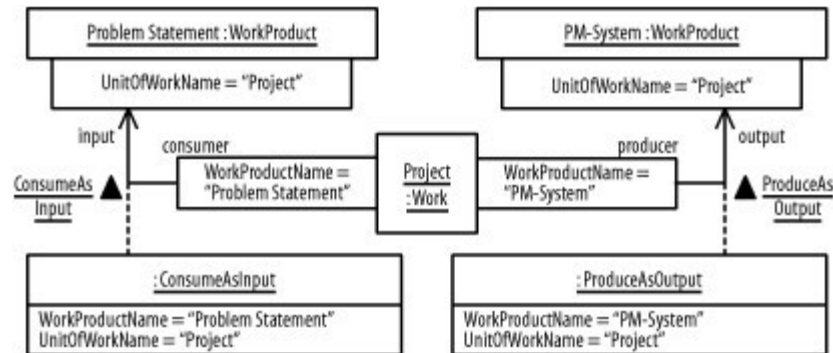


Figure 1.7

Lecture 3

1.6 Class Relationship: Association

An association defines a type of link and is a general relationship between classes. For example, the project management system involves various general relationships, including manage, lead, execute, input, and output between projects, managers, teams, work products, requirements, and systems. Consider, for example, how a project manager leads a team.

1.6.1 Binary associations

A binary association relates two classes. For example, one binary relationship in the project management system is between individual workers and their units of work, and another binary relationship is between individual workers and their work products.

In a UML class diagram, a binary association is shown as a solid-line path connecting the two related classes. A binary association may be labeled with a name. The name is usually read from left to right and top to bottom; otherwise, it may have a small black solid triangle next to it where the point of the triangle indicates the direction in which to read the name, but the arrow is purely descriptive, and the name of the association should be understood by the classes it relates.

Figure shows various associations within the project management system using the most basic notation for binary associations. The associations in the figure are as follows:

- A worker is responsible for work products and performs units of work

- Units of work consume work products as input and produce work products as output.

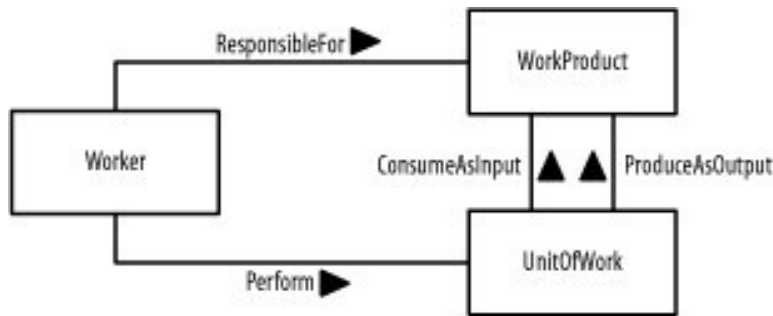


Figure 1.8

1.6.2 N-ary associations

An n-ary association relates three or more classes. For example, in the project management system, the use of a worker involves the worker, her units of work, and her associated work products.

In a UML class diagram, an n-ary association is shown as a large diamond with solid-line paths from the diamond to each class. An n-ary association may be labeled with a name. The name is read in the same manner as for binary associations, described in the previous section.

Figure shows an n-ary association associated with the project management system using the most basic notation for n-ary associations. This association states that utilization involves workers, units of work, and work products. As with a binary association, an n-ary association is also commonly named using a verb phrase. However, this is not always the case for example; the n-ary Utilization association shown in Figure is described using a noun rather than a verb, because it is named from our perspective rather than the perspective of one of the classes. That is, from our perspective, we want to understand a worker's utilization relative to the other classes. From the worker's perspective, a worker is responsible for work products and performs units of work.

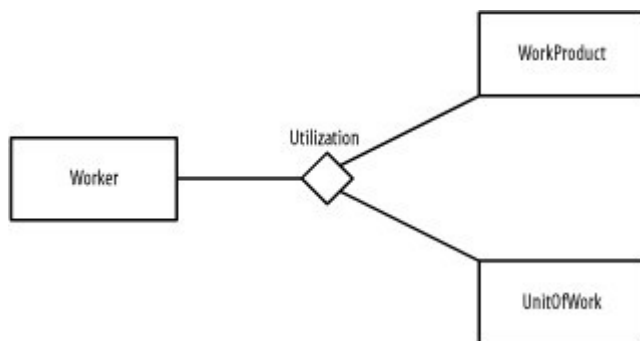


Figure 1.9

1.6.3 Association Classes

Association classes may be applied to both binary and n-ary associations. Similar to how a class defines the characteristics of its objects, including their structural features and behavioral features, an association class may be used to define the characteristics of its links, including their structural features and behavioral features. These types of classes are used when you need to maintain information about the relationship itself.

In a UML class diagram, an association class is shown as a class attached by a dashed-line path to its association path in a binary association or to its association diamond in an n-ary association. The name of the association class must match the name of the association.

The association classes track the following information:

- The reason a worker is responsible for a work product
- The reason a worker performs a unit of work
- A description of how a unit of work consumes a work product
- A description of how a unit of work produces a work product.

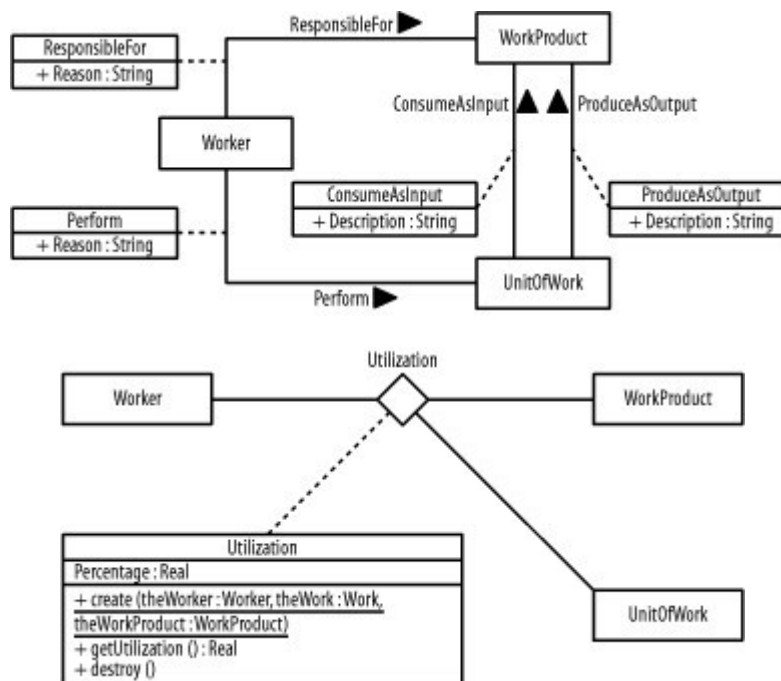


Figure 1.10

1.6.4 Association Ends

An association end is an endpoint of the line drawn for an association, and it connects the association to a class. An association end may include any of the following items to express more detail about how the class relates to the other class or classes in the association:

Rolename

Navigation arrow

Multiplicity specification

Aggregation or composition symbol

Qualifier

- a. Rolenames

A rolename is optional and indicates the role a class plays relative to the other classes in an association, how the other classes "see" the class or what "face" the class projects to the other classes in the relationship. A rolename is shown near the end of an association attached to a class.

For example, a work product is seen as input by a unit of work where the unit of work is seen as a consumer by the work product; a work product is seen as output by a unit of work where the unit of work is seen as a producer by the work product, Figure 3-13. Binary association ends

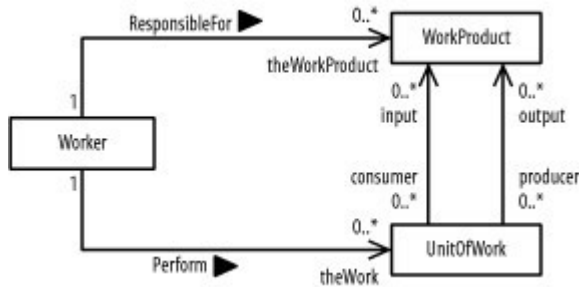


Figure 1.11

b. Navigation

Navigation is optional and indicates whether a class may be referenced from the other classes in an association. Navigation is shown as an arrow attached to an association end pointing toward the class in question. If no arrows are present, associations are assumed to be navigable in all directions, and all classes involved in the association may reference one another.

For example, given a worker, you can determine his work products and units of work. Given a unit of work, you can determine its input and output work products; but given a work product, you are unable to identify which worker is responsible for it or which units of work reference it as input or Given a worker, you can reference his work products and units of work to determine his utilization, but given a work product or unit of work, you are unable to determine its utilization by a worker.

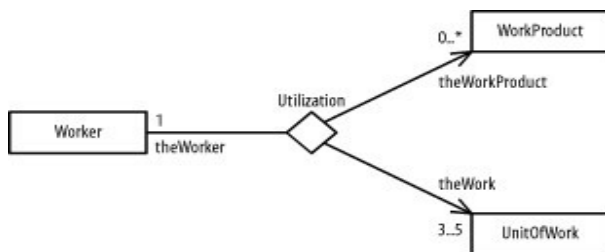


Figure 1.12

c. Multiplicity

Multiplicity (which is optional) indicates how many objects of a class may relate to the other classes in an association. Multiplicity is shown as a comma-separated sequence of the following:
 Integer intervals

Literal integer values

Intervals are shown as a lower-bound .. upper-bound string in which a single asterisk indicates an unlimited range. No asterisks indicate a closed range. For example, 1 means one, 1..5 means one to five, 1, 4 means one or four, 0..* and * mean zero or more (or many), and 0..1 and 0, 1 mean zero or one. There is no default multiplicity for association ends. Multiplicity is simply undefined, unless you specify it. For example:

A single worker is responsible for zero or more work products.

A single work product is the responsibility of exactly one worker.

A single worker performs zero or more units of work.

A unit of work is performed by exactly one worker.

A unit of work may input as a consumer zero or more work products and output as a producer zero or more work products.

A work product may be consumed as input by zero or more units of work and produced as output by zero or more units of work.

To determine the multiplicity of a class, ask yourself how many objects may relate to a single object of the class. The answer determines the multiplicity on the other end of the association. For example, using Another way to determine multiplicity is to ask how many objects of a class may relate to a single object of the class on the other end of an association, or to a single object of each class on the other ends of an n-ary association. The answer determines the multiplicity for the class.

d. Aggregation

Aggregation is whole-part relationship between an aggregate, the whole, and its parts. This relationship is often known as a has-a relationship, because the whole has its parts. For example, when you think of workers making up a team, you can say that a team has workers. Aggregation is shown using a hollow diamond attached to the class that represents the whole.

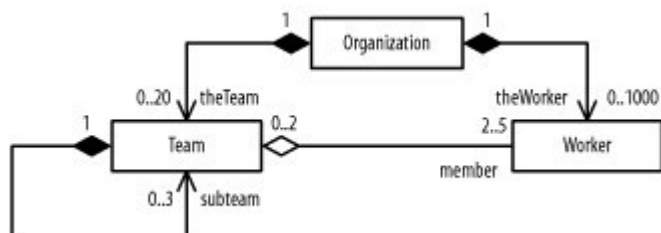


Figure 1.13

c. Composition

Composition, also known as composite aggregation, is a whole-part relationship between a composite (the whole) and its parts, in which the parts must belong only to one whole and the whole is responsible for creating and destroying its parts when it is created or destroyed. This relationship is often known as a contains-a relationship, because the whole contains its parts. For example, an organization contains teams and workers, and if the organization ceases to exist, its teams and workers also cease to exist. The specific individuals who represent the workers would still exist, but they would no longer be workers of the organization, because the organization

would no longer exist. Composition is shown using a filled diamond attached to the class that represents the whole. As a UML rule, composition is used only with binary associations.

It shows that an organization may contain 0 to 20 teams and 0 to 1,000 workers. Furthermore, each team has 2 to 5 workers and each worker may be a member of 0 to 2 teams. In addition, a team may contain 0 to 3 subteams.

To determine if you should use an aggregation or composition, ask yourself a few questions. First, if the classes are related to one another, use an association. Next, if one class is part of the other class, which is the whole, use aggregation; otherwise, use an association. For example, Figure shows that workers are part of a team and organization, teams are part of an organization, and subteams are part of teams. Finally, if the part must belong to one whole only, and the whole is responsible for creating and destroying its parts, use composition; otherwise, simply use aggregation.

Composition also may be shown by graphically nesting classes, in which a nested class's multiplicity is shown in its upper-right corner and its rolename is indicated in front of its class name. Separate the rolename from the class name using a colon.

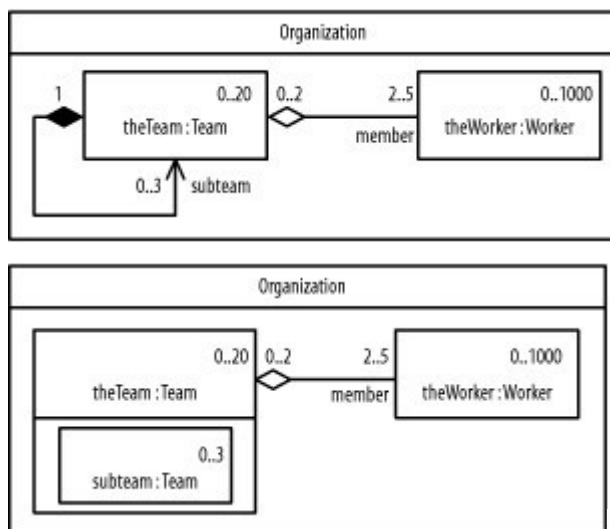


Figure 1.14

e. Qualifiers

A qualifier is an attribute of an association class that reduces the multiplicity across an association. For example, The multiplicity between work products and units of work is zero or more for both associations; that is, there may be many work products associated with a single unit of work and there may be many units of work associated with a single work product. Rather than simply say that there are "many" objects involved in the relationship, you can communicate a more finite number.

You can reduce the multiplicity between work products and units of work by asking yourself what you need to know about a unit of work so that you can define a more specific multiplicity one that isn't unbounded on the high-end. Likewise, you can ask yourself the same question about the association between work product and units of work. If you have a work product and the name of a unit of work, you can determine whether a relationship exists between the two; likewise, if you

have a unit of work and the name of a work product, you can determine whether a relationship exists between those two. The trick is to document precisely what information is needed so you can identify the objects on the other end of the relationship. This is where the qualifier comes into play.

Essentially, a qualifier is a piece of information used as an index to find the objects on the other end of an association. A qualifier is shown as a small rectangle attached to a class where an object of the class, together with a value for the qualifier, reduces the multiplicity on the other end of the association. Qualifiers have the same notation as attributes, have no initial values, and must be attributes of the association or the class on the other end of the association.

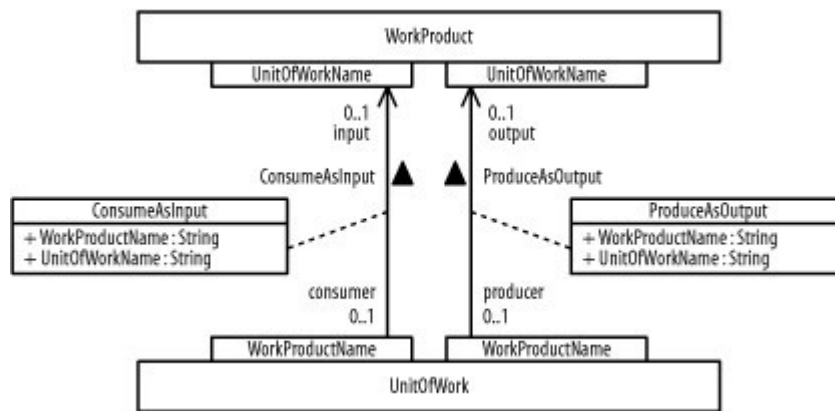


Figure 1.15

1.7 Aggregation

If a class has an entity reference, it is known as Aggregation. Aggregation represents HAS-A relationship.

Consider a situation; Employee object contains much information such as id, name, emailId etc. It contains one more object named address, which contains its own informations such as city, state, country, zipcode etc. as given below.

Aggregation or composition is a relationship among classes by which a class can be made up of any combination of objects of other classes. It allows objects to be placed directly within the body of other classes. Aggregation is referred as a “part-of” or “has-a” relationship, with the ability to navigate from the whole to its parts. An aggregate object is an object that is composed of one or more other objects.

In the relationship, “a car has-a motor”, car is the whole object or the aggregate, and the motor is a “part-of” the car. Aggregation may denote:

- Physical containment: Example, a computer is composed of monitor, CPU, mouse, keyboard, and so on.
- Conceptual containment: Example, shareholder has-a share.

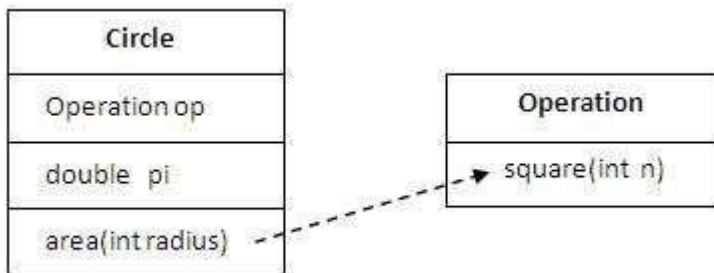


Figure 1.16

Multiple choice Questions

1. A Class consists of which of these abstractions
 - a) Set of the objects
 - b) Operations
 - c) Attributes
 - d) All of the mentioned
 - e) b, c
2. An attribute is a data item held by which of the following ?
 - a) Class
 - b) Object
 - c) All of the mentioned
 - d) None of the mentioned
3. What among these is true ?
 - a) Associations may also correspond to relation between instances of three or more classes
 - b) Association lines may be unlabeled or they may show association name
 - c) All of the mentioned
 - d) None of the mentioned
4. What is multiplicity for an association?
 - a) The multiplicity at the target class end of an association is the number of instances that can be associated with a single instance of source class
 - b) The multiplicity at the target class end of an association is the number of instances that can be associated with a number instance of source class
 - c) All of the mentioned
 - d) None of the mentioned
5. Which of these analysis are not acceptable ?
 - a) Object oriented design is far better approach compared to structural design
 - b) Object oriented design always dominates structural design
 - c) Object oriented design are given more preference than structural design
 - d) Object oriented uses more specific notations
6. Which these does not represent object oriented design ?

-
- a. It follows regular procedural decomposition in favor of class and object decomposition
 - b. Programs are thought of collection of objects
 - c. Central model represents class diagrams that show the classes comprising a program and their relationships to one another
 - d. Object-oriented methods incorporates Structural methods
7. An attribute is a data item held by which of the following ?
- a) Class
 - b) Object
 - c) All of the mentioned
 - d) None of the mentioned
8. In UML diagrams, relationship between object and component parts is represented by a.
- a. composition
 - b. aggregation
 - c. segregation
-

- d. increment
9. An operation can be described as?
- a) Object behavior
 - b) Class behavior
 - c) Functions
 - d) a,b
 - e. None of the mentioned
10. Class is a user fined variable
- a.TRUE
 - b.FALSE

Short Answer Type Questions

1. Discuss about different programming paradigm 2.
What do you mean by class and object?
3. What is association and aggregation?

1.5 Class and Object

A class is a blueprint or prototype from which objects are created. This section defines a class that models the state and behavior of a real-world object. It intentionally focuses on the basics, showing how even simple classes can cleanly model state and behavior.

An object is a software bundle of related state and behavior. Software objects are often used to model the real-world objects that you find in everyday life. This lesson explains how state and behavior are represented within an object, introduces the concept of data encapsulation, and explains the benefits of designing your software in this manner.

We need to understand the relationship between class and its object. Let think about how we can identify a cat , a dog or a bird?? We can identify them by recalling the image(definition) of the entities(objects) mapped in our brain.

Object can be defined as follows:

State (Features) – this would be the variables

Behaviors-this would be the method

1.6 Message Passing

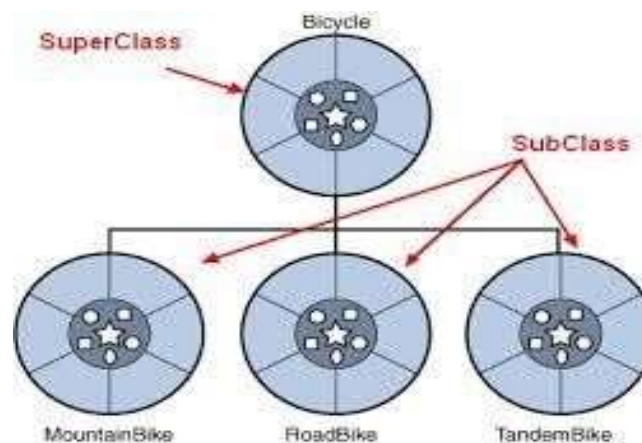
Message passing is a technique for invoking behavior (i.e., running a program) on a computer. In contrast to the traditional technique of calling a program by name, message passing uses an object model to distinguish the general function from the specific implementations. The invoking program sends a message and relies on the object to select and execute the appropriate code. The justifications for using an intermediate layer essentially falls into two categories: encapsulation and distribution.

Synchronous message passing is what typical object-oriented programming languages such as Java and Smalltalk use. Asynchronous message passing requires additional capabilities for storing and retransmitting data for systems that may not run concurrently.

Asynchronous message passing is generally implemented so that all the complexities that naturally occur when trying to synchronize systems and data are handled by an intermediary level of software. Commercial vendors who develop software products to support these intermediate levels usually call their software. One of the most common types of middleware to support asynchronous messaging is called Message-oriented middleware (MOM).

1.7 Inheritance

Inheritance is the process by which one object acquires the properties of another object. This is important because it supports the concept of hierarchical classification. As mentioned earlier, most knowledge is made manageable by hierarchical (that is, top-down) classifications. For example, a Golden Retriever is part of the classification dog, which in turn is part of the mammal class, which is under the larger class animal. Without the use of hierarchies, each object would need to define all of its characteristics explicitly. However, by use of inheritance, an object need only define those qualities that make it unique within its class. It can inherit its general attributes from its parent.



1.8 Polymorphism

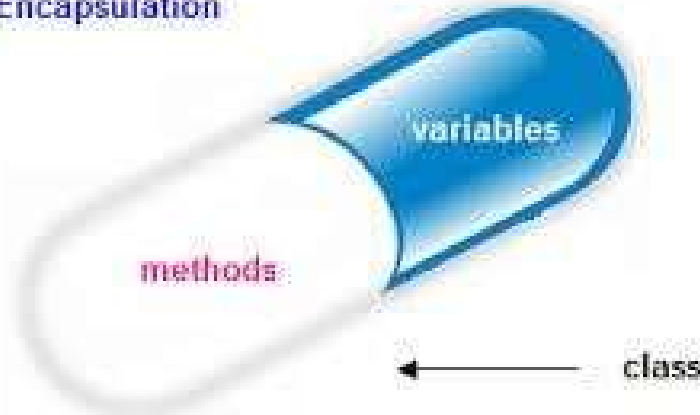
Polymorphism (from Greek, meaning “many forms”) is a feature that allows one interface to be used for a general class of actions. More generally, the concept of polymorphism is often expressed by the phrase “one interface, multiple methods.” This means that it is possible to design a generic interface to a group of related activities. This helps reduce complexity by allowing the same interface to be used to specify a general class of action.



1.9 Encapsulation

Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse. One way to think about encapsulation is as a protective wrapper that prevents the code and data from being arbitrarily accessed by other code defined outside the wrapper. Access to the code and data inside the wrapper is tightly controlled through a well-defined interface. Inheritance interacts with encapsulation as well.

Encapsulation



1.10 Difference between Procedural and Object Oriented Programming:

Procedural Programming	Object Oriented programming
Emphasis is on doing things (algorithms)	Emphasis is on data rather than on procedure
Large Programs are divided into smaller programs known as functions.	Programs are divided into objects. Data structures are designed in such a way that it characterizes the object
Data move openly around the system from function to function.	Functions that operate on data are ties together in a data structure called class.
Functions transform data from one form to another.	Objects may communicate to each other with the help of functions.
To add new data and function in POP is not so easy.	New data and functions can be easily added whenever necessary.
does not have any proper way for hiding data so it is less secure .	provides Data Hiding so provides more security .
Overloading is not possible.	Overloading is possible in the form of Method Overloading and Operator Overloading.
Employs top down approach in program design.	Follows bottom-up approach.
C, VB, FORTRAN, Pascal.	C++, JAVA, VB.NET, C#.NET.

1.11 Advantages and Disadvantages of Object Oriented Programming

Advantages

1. Improved software-development productivity: Object-oriented programming is modular, as it provides separation of duties in object-based program development. It is also extensible, as

objects can be extended to include new attributes and behaviors. Objects can also be reused within an across applications. Because of these three factors – modularity, extensibility, and reusability – object-oriented programming provides improved software-development productivity over traditional procedure-based programming techniques.

2. Improved software maintainability: For the reasons mentioned above, object oriented software is also easier to maintain. Since the design is modular, part of the system can be updated in case of issues without a need to make large-scale changes.
3. Faster development: Reuse enables faster development. Object-oriented programming languages come with rich libraries of objects, and code developed during projects is also reusable in future projects.
4. Lower cost of development: The reuse of software also lowers the cost of development. Typically, more effort is put into the object-oriented analysis and design, which lowers the overall cost of development.
5. Higher-quality software: Faster development of software and lower cost of development allows more time and resources to be used in the verification of the software. Although quality is dependent upon the experience of the teams, object oriented programming tends to result in higher-quality software.

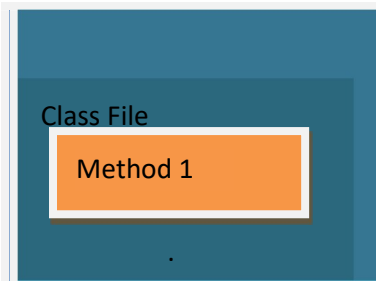
Some of the disadvantages of object-oriented programming include:

1. Steep learning curve: The thought process involved in object-oriented programming may not be natural for some people, and it can take time to get used to it. It is complex to create programs based on interaction of objects. Some of the key programming techniques, such as inheritance and polymorphism, can be challenging to comprehend initially.
2. Larger program size: Object-oriented programs typically involve more lines of code than procedural programs.
3. Slower programs: Object-oriented programs are typically slower than procedure based programs, as they typically require more instructions to be executed.
4. Not suitable for all types of problems: There are problems that lend themselves well to functional-programming style, logic-programming style, or procedure-based programming style, and applying object-oriented programming in those situations will not result in efficient programs.

Multiple Choice Questions

1. Class is a
 - a. Structure
 - b. Union
 - c. A user defined variable
 - d. A system defined variable
2. An object is
 - a. A class type variable
 - b. A user defined variable
 - c. A structure type variable
 - d. None of this
3. Message passing is like a method calling

-
- a. TRUE
 - b. FALSE
4. When variables are wrapping up inside a class is known as
 - a. Inheritance
 - b. Polymorphism
 - c. Encapsulation
 - d. None of these
 5. Which cannot be a content of a class
 - a. Class
 - b. Object
 - c. Variable
 - d. None of these
 6. A cat is an animal, which type of relationship is it?
 - a. Encapsulation
 - b. Inheritance
 - c. Polymorphism
 - d. Both b and c.
 7. Basic Encapsulation in java is



- a. Object
 - b. Method
 - c. Message
 - d. Class
8. Polymorphism allows you to create clean, sensible, readable, and resilient code
- a. TRUE
 - b. FALSE
9. Inheritance interacts with encapsulation as well.
- a. TRUE
 - b. FALSE
10. Inheritance is the process by which one object acquires the properties of another object
- a. TRUE
 - b. FALSE

Module 2

Java Program Execution:

2.1 Structure of a Simple Java Program

Put a class in source file

Put methods in class.

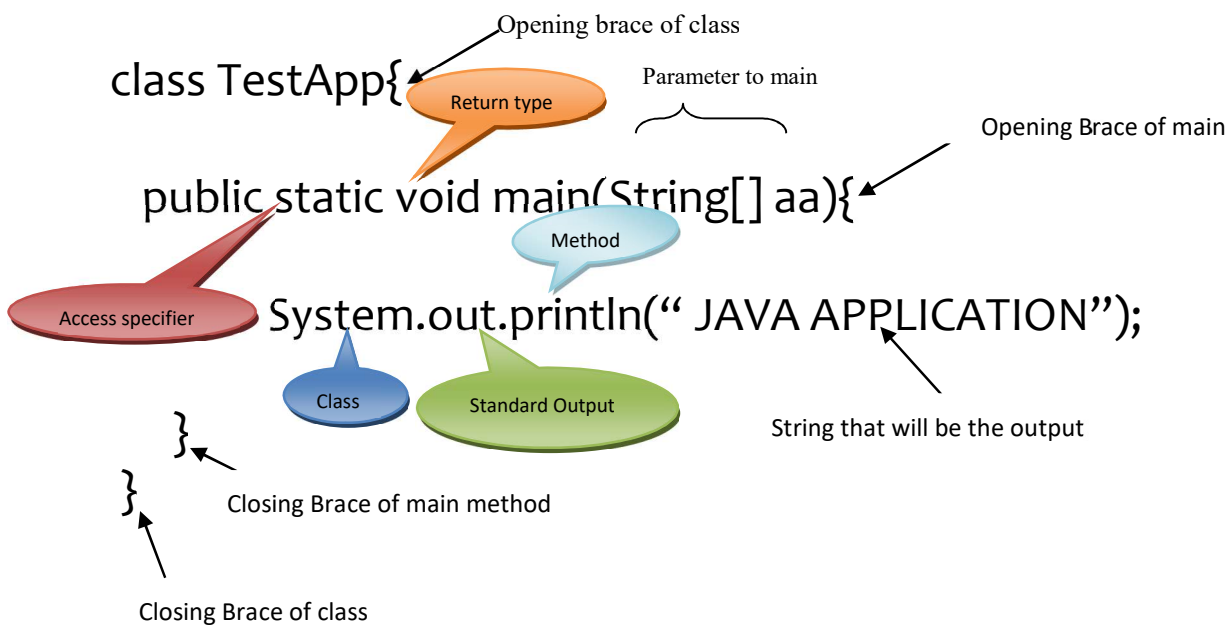
Put statements in method

Source file: with .java extension. Holds the class definition. There may be more than one class in a source file.

Inside class: Has one or more methods.

Inside methods: write instructions of how methods behave.

Every java application must have to have at least one class and one method(if we want to run the file). JVM runs every things written in main method.



2.2 Component of Java Development Kit

JDK is an acronym for Java Development Kit. It physically exists. It contains JRE + development tools.

It includes:

- 1.Appletviewer(It is used for viewing the applet)
- 2.Javac(It is a Java Compiler)
- 3.Java(It is a java interpreter)
- 4.Javap(Java diasssembler,which convert byte code into program description)
- 5.Javah(It is for java C header files)
- 6.Javadoc(It is for creating HTML document)
- 7.Jdb(It is Java debugger)

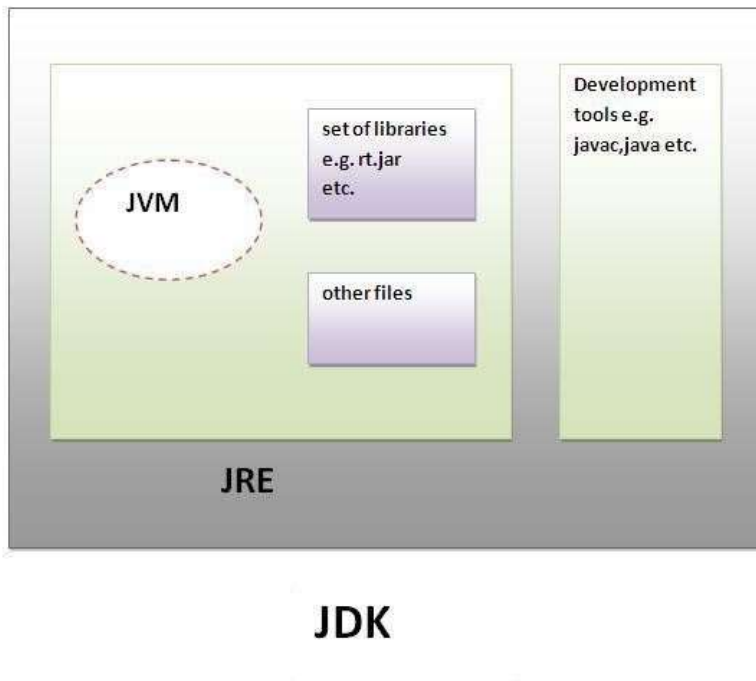
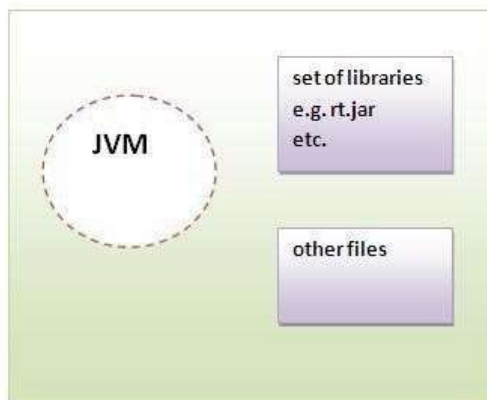


Figure 3.2

JRE:

JRE is an acronym for Java Runtime Environment. It is used to provide runtime environment. It is the implementation of JVM. It physically exists. It contains set of libraries + other files that JVM uses at runtime.

Implementation of JVMs is also actively released by other companies besides Sun Micro Systems.



JRE

JVM:

JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides runtime environment in which java bytecode can be executed.

JVMs are available for many hardware and software platforms. JVM, JRE and JDK are platform dependent because configuration of each OS differs. But, Java is platform independent.

The JVM performs following main tasks:

- Loads code
- Verifies code
- Executes code
- Provides runtime environment

2.3 Compiling and Running a Java Program

a) javac (Java compiler)

In java, we can use any text editor for writing program and then save that program with `—.java` extension. Java compiler convert the source code or program in byte code and Interpreter convert `—.java` file in `—.class` file. Syntax:

```
C:\javac filename.java
```

If my filename is `—abc.java` then the syntax will be

```
C:\javac abc.java
```

b) java (Java Interpreter)

As we learn that, we can use any text editor for writing program and then save that Program with `—.java` extension. Java compiler converts the source code or program in byte code and interpreter convert `—.java` file in `—.class` file. Syntax:

```
C:\java file name
```

If my filename is `abc.java` then the syntax will be

```
C:\java abc
```

2.4 public static void main(String[] aa):

public : public means that the method could be accessed from anywhere. To run a java program the jvm require to access the main method and it may be that the jvm and the main method is not in same folder then also jvm can access it because of public.

static :In object oriented concept every members other that static member need to be called by creating the object of the class. So if main method is not declared as static then jvm can only call the main method after creating the object of the class, but it is not possible to create object before execution starts so the main method is declared as static.

void : Return Type, It defined what this method can return. Which is void in this case it means that this method will not return anything. **main**: Name of the method. This method name is searched by JVM as starting point for an application.

String aa[] : Parameter to main method.

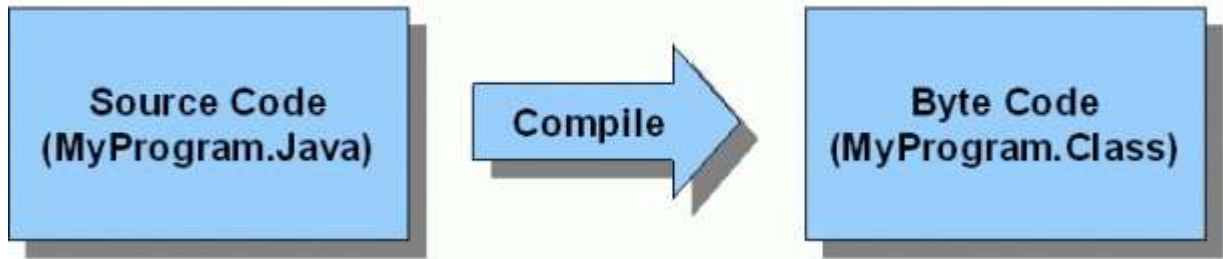
Why Java is a Platform Independent Language??

A platform is the hardware or software environment in which the program runs. This platform in real-life is the laptop and the underlying operating system runs on that laptop

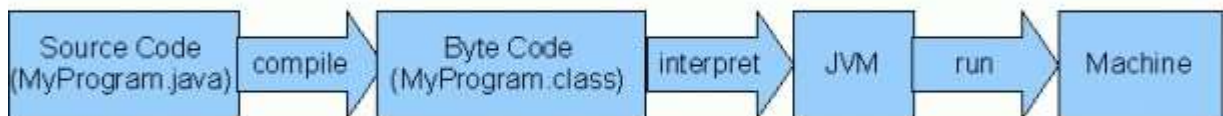


Any program that needs to be executed on a platform must be converted into machine instructions understandable by that platform. When developers write Java programs and compile it, the code is not directly converted into machine instructions but it is compiled into Byte Code.

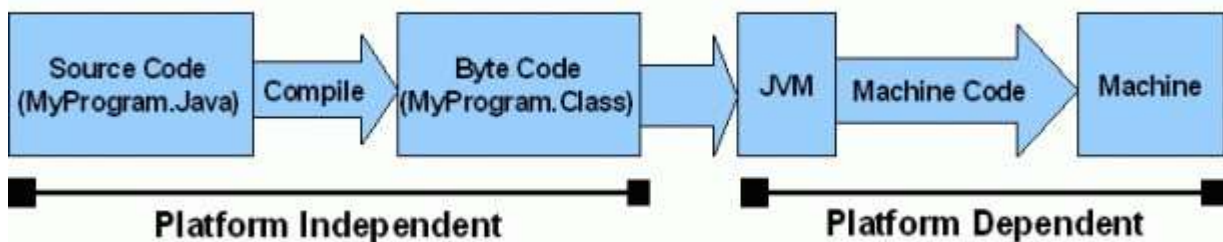
Illustration of how Bytecode is created



Byte code is not machine instructions that any platform like Windows, Mac will understand it directly. This Byte code is understood by Java made Virtual machines. Java Virtual Machines (JVM) interprets the Bytecode into machine specific instructions so that platform can understand its.



Java Virtual Machine (JVM) plays a key role in making the Byte code understandable to any underlying platform thus makes the Byte code platform independent. If JVM are not pre-installed on any operating system then byte code will fail to execute on that machine. This makes the Bytecode platform independent and Java Virtual Machine Platform (JVM) platform dependent.



Java programming language does not depend on any underlying platform and thus it makes Java platform independent. That's why it is said that Java programs are written once and executed anywhere.

Multiple Choice Questions

1. Java was invented by
 - i) James Gosling
 - ii) Denies Ritchie
 - iii) Jack Gosling
 - iv) Denies Robert
2. Current release of JDK is
 - i) JDK 1.1
 - ii) JDK 1.8
 - iii) JDK 8.1
 - iv) JDK 9
3. JRE stands for
 - i) Java Runtime Execution
 - ii) Java Runtime Environment
 - iii) Java Runnable execution
 - iv) Java Running Environment
4. JVM stands for
 - i) Java Virtual Machine
 - ii) Java Virtual Mechanism
 - iii) Java Variable Machine
 - iv) None of these
5. Java is plat form independent for
 - i) JVM
 - ii) bytecode
 - iii) excitable code
 - iv) Integrated Development Environment
6. Command for running a A.java program is
 - i) java A.java
 - ii) javac A
 - iii) javac A.java
 - iv) java A
7. bytecode is generated after
 - i) compilation
 - ii) Exceution
 - iii) Editing
 - iv) preprocessing
8. What is syntax to print output in Java
 - i) S.o.p(“ “)
 - ii) System.out.print(“ “)
 - iii) out.print(“ “)
 - iv) print(“ “)
9. What is extension of bytecode files
 - i) .java
 - ii).class
 - iii).Class
 - iv).Java
10. What is valid syntax of main
 - i) Public static void main(String[] a)
 - ii) void main(String[] a)
 - iii) public static void main(String[] a)
 - ii) public static void main(String a)

Module 3

Class & Object properties

3.1 Tokens

A Java program is basically a set of classes. A class is defined by a set of declaration statements and methods or functions. Most statements contain expressions, which express the actions carried out on information or data. Smallest indivisual thing in a program are known as tokens. The compiler recognizes them for building up expression and statements.

Tokens in Java:

There are five types of token as follows:

1. Literals
2. Identifiers
3. Keywords
4. Operators
5. Separators

3.1.1 Literals: Literals in Java are a sequence of characters (digits, letters and other characters) that characterize constant values to be stored in variables. Java language specifies five major types of literals are as follows:

1. Integer literals
2. Floating point literals
3. Character literals
4. String literals
5. Boolean literals

3.1.2 Identifiers:

Identifiers are programmer-created tokens. They are used for naming classes, methods, variables, objects, labels, packages and interfaces in a program. Java identifiers follow the following rules:

1. They can have alphabets, digits, and the underscore and dollar sign characters.
2. They must not start with a digit.
3. Uppercase and lowercase letters are individual.
4. They can be of any length.

Identifier must be meaningful, easily understandable and descriptive.

For example:

Private and local variables like —length.

Name of public methods and instance variables begin with lowercase letter like —addition

3.1.3 Keywords:

Keywords are important part of Java. Java language has reserved 50 words as keywords.

Keywords have specific meaning in Java. We cannot use them as variable, classes and method.

Following table shows keywords.

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

Table 4.1

3.1.4 Operator:

Java carries a broad range of operators. An operator is symbols that specify operation to be performed may be certain mathematical and logical operation. Operators are used in programs to operate data and variables. They frequently form a part of mathematical or logical expressions.

Categories of operators are as follows:

1. Arithmetic operators
2. Logical operators
3. Relational operators
4. Assignment operators
5. Conditional operators
6. Increment and decrement operators
7. Bit wise operators

3.1.4.1 Arithmetic operators:

Arithmetic operators are used to make mathematical expressions and the working out as same in algebra. Java provides the fundamental arithmetic operators. These can operate on built in data type of Java.

Following table shows the details of operators.

Operator	Importance/ significance
+	Addition
-	Subtraction
/	Division
*	Multiplication
%	Modulo division or remainder

Table 4.2

“+” operator in Java:

In this program, we have to add two integer numbers and display the result.

```
class AdditionInt
{
public static void main (String args[])
{
int a = 6;
int b = 3;
System.out.println("a = " + a);
System.out.println("b =" + b);
int c = a + b;
System.out.println("Addition = " + c);
}
}
```

Output:

a= 6

b= 3

Addition=9

“-” operator in Java:


```
class SubstractionInt
{
public static void main (String args[])
{
int a = 6;
int b = 3;
System.out.println("a = " + a);
System.out.println("b =" + b);
int c = a - b;
System.out.println("Subtraction= " + c);
}
}
```

Output:

a=6

b=3

Subtraction=3

“*” operator in Java:

```
Class MultiplicationInt
{
public static void main (String args[])
{
int a = 6;
int b = 3;
System.out.println("a = " + a);
System.out.println("b =" + b);
int c = a * b;
System.out.println("Multiplication= " + c);
}
}
```

Output:

a=6

b=3

Multiplication=18

“/” operator in Java:

```
Class DivisionInt
{
public static void main (String args[])
{
int a = 6;
int b = 3;
System.out.println("a = " + a);
System.out.println("b =" + b);
c = a / b;
System.out.println("division=" + c);
}
}
```

Remainder or modulus operator (%) in Java:

```
Class Remainderoptr
{
public static void main (String args[])
{
int a = 6;
int b = 3;
System.out.println("a = " + a);
System.out.println("b =" + b);
c = a % b;
System.out.println("remainder=" + c);
}
}
```

Output:

```
a=6
b=3
Remainder=0
```

When both operands in the expression are integers then the expression is called Integer expression and the operation is called Integer arithmetic.

When both operands in the expression are real then the expression is called Real expression and the operation is called Real arithmetic.

When one operand in the expression is integer and other is float then the expression is called Mixed Mode Arithmetic expression and the operation is called Mixed Mode Arithmetic operation.

3.1.4.2 Logical operators:

When we want to form compound conditions by combining two or more relations, then we can use logical operators.

Following table shows the details of operators.

Operators	Importance/ significance
	Logical – OR
&&	Logical –AND
!	Logical –NOT

Table 4.3

The logical expression defers a value of true or false. Following table shows the truth table of Logical – OR and Logical – AND.

Truth table for Logical – OR operator:

Operand1	Operand3	Operand1 Operand3
T	T	T
T	F	T
F	T	T
F	F	F

Table 4.4

T - True

F - False

Truth table for Logical – AND operator:

Operand1	Operand3	Operand1 && Operand3
T	T	T
T	F	F
F	T	F
F	F	F

Table 4.5

T - True

F – False

Now the following program shows the use of Logical operators.

```

class LogicalOptr
{
public static void main (String args[])
{
boolean a = true;
boolean b = false;
System.out.println("a||b = " +(a||b));
System.out.println("a&&b = "+(a&&b));
System.out.println("a! = "+(!a));

}
}

```

Output:

```

a||b = true
a&&b = false
a! = false

```

3.1.4.3 Relational Operators:

When evaluation of two numbers is performed depending upon their relation, assured decisions are made.

The value of relational expression is either true or false. If $A=7$ and $A < 10$ is true while $10 < A$ is false.

Operator	Importance/ significance
>	Greater than
<	Less than
!=	Not equal to
>=	Greater than or equal to

Table 4.6

Now, following examples show the actual use of operators.

- 1) If $10 > 30$ then result is false
- 2) If $40 > 17$ then result is true
- 3) If $10 >= 300$ then result is false
- 4) If $10 <= 10$ then result is true

Now the following program shows the use of operators.

```

class Reloptr1 {
public static void main (String args[])
{
int a = 10;
int b = 30; System.out.println("a||b = " +(a||b));
System.out.println("a>b = " +(a>b));
System.out.println("a<b = "+(a<b));
System.out.println("a<=b = "+(a<=b));
}
}

```

Output:

```

a>b = false
a<b = true
a<=b = true

```

3.1.4.4 Assignment Operators:

Assignment Operators is used to assign the value of an expression to a variable and is also called as Shorthand operators.

Variable_name binary_operator = expression Following table show the use of assignment operators.

Simple Assignment Operator	Statement with shorthand Operators
A=A+1	A+=1
A=A-1	A-=1
A=A/(B+1)	A/=(B+1)
A=A*(B+1)	A*=(B+1)
A=A/C	A/=C
A=A%C	A%=C

Table 4.7

These operators avoid repetition, easier to read and write.

Now the following program shows the use of operators.

```
class Assoptr {
public static void main (String args[])
{
int a = 10;
int b = 30;
int c = 30;
a+=1;
b-=3;
c*=7;
System.out.println("a = " +a);
System.out.println("b = "+b);
System.out.println("c = "+c);
}
}
```

Output:

```
a = 11
b = 18
c = 310
```

3.1.4.5 Conditional Operators:

The character pair ?: is a ternary operator of Java, which is used to construct conditional expressions of the following form: Expression1 ? Expression3 : Expression3 The operator ? : works as follows:

Expression1 is evaluated if it is true then Expression3 is evaluated and becomes the value of the conditional expression. If Expression1 is false then Expression3 is evaluated and its value becomes the conditional expression.

A=3;

B=4;

C=(A<B)?A:B;

C=(3<4)?3:4;

C=4

```

class Coptr
{
public static void main (String args[])
{
int a = 10;
int b = 30;
int c;
c=(a>b)?a:b;
System.out.println("c = " +c);
c=(a<b)?a:b;
System.out.println("c = " +c);
}
}

```

Output:

c = 30

c = 10

3.1.4.6 Increment and Decrement Operators:

The increment operator ++ adds 1 to a variable. Usually the variable is an integer type, but it can be a floating point type. The two plus signs must not be split by any character. Usually they are written immediately next to the variable.

Following table shows the use of operators.

Expression	Process	Example	end result
A++	Add 1 to a variable after use.	int A=10,B; B=A++;	A=11 B=10
++A	Add 1 to a variable before use.	int A=10,B; B=++A;	A=11 B=11
A--	Subtract 1 from a variable after use.	int A=10,B; B=A--;	A=9 B=10
--A	Subtract 1 from a variable before use.	int A=10,B; B=--A;	A=9 B=9

Table 4.8

```

class IncDecOp
{
public static void main(String args[]) {
int x=1;
int y=3;
int u;
int z;
u=++y;
z=x++;
System.out.println(x);
System.out.println(y);
System.out.println(u);
System.out.println(z);
}
}

```

Output:

```

3
4
4
1

```

3.1.4.7 Bit Wise Operators:

Bit wise operator execute single bit of their operands. Following table shows bit wise operator:

Operator	Importance/ significance
	Bitwise OR
&	Bitwise AND
&=	Bitwise AND assignment
=	Bitwise OR assignment
^	Bitwise Exclusive OR
<<	Left shift
>>	Right shift
~	One's complement

Table 4.9


```

Class Boptr3
{
public static void main (String args[])
{
int a = 16;
int b = a>>3;
System.out.println("a = " +a);
System.out.println("b = " +b);
}
}

```

Output:

```

a = 16
b = 3

```

3.1.5 Separator:

Separators are symbols. It shows the separated code. They describe function of our code.

Name	Use
()	Parameter in method definition, containing statements for conditions ,etc.
{}	It is used for define a code for method and classes
[]	It is used for declaration of array
;	It is used to show the separate statement
,	It is used to show the separation in identifier in variable declaration
.	It is used to show the separate package name from sub-packages and classes, separate variable and method from reference variable.

Table 4.10

3.1.6 Operator Precedence:

Highest			
()	[]	.	
++	--	~	
*	/	%	
+	-		
>>	>>>	<<	
>	>=	<	<=
==	!=		
&			
^			
&&			
?:			
=	op=		
Lowest			

Table 4.11

3.2 Data types:

A **data type** is a scheme for representing values. An example is int which is the Integer, a data type.

Values are not just numbers, but any manner of data that a computer can process.

The data type defines the kind of data that is represented by a variable.

As with the keyword class, Java data types are case sensitive.

There are two types of data types

- Primitive data type
- Non-primitive data type

In primitive data types, there are two categories numeric means Integer, Floating points

,Nonnumeric means

Character and Boolean

In non-primitive types, there are three categories

: classes arrays int interface

Datatype	Size(byte)	Range
byte	1	-128 to 127
boolean	1	True or false
char	2	A-Z,a-z,0-9,etc.
short	2	-32768 to 32767
Int	4	(about) -2 million to 2 million
long	8	(about) -10E18 to 10E18
float	4	-3.4E38 to 3.4E18
double	8	-1.7E308 to 1.7E308

Table 4.12

3.2.1 Integer data type:

Integer datatype can hold the numbers (the number can be positive number or negative number).

In

Java, there are four types of integer as follows:

- byte
- short
- int long We can make integer long by adding `_l` or `_L` at the end of

the number. **Floating point data type:**

It is also called as Real number and when we require accuracy then we can use it.

There are two types of floating point data type.

- float
- double

It is represent single and double precision numbers. The float type is used for single precision and it uses 4 bytes for storage space. It is very useful when we require accuracy with small degree of precision. But in double type, it is used for double precision and uses 8 bytes of storage space. It is useful for large degree of precision.

3.2.2 Character data type:

It is used to store single character in memory. It uses 2 bytes storage space.

3.2.3 Boolean data type:

It is used when we want to test a particular condition during the execution of the program.

There are only two values that a boolean type can hold: true and false.

Boolean type is denoted by the keyword boolean and uses only one bit of storage.

3.3 Variables:

Variables are labels that express a particular position in memory and connect it with a data type.

The first way to declare a variable: This specifies its data type, and reserves memory for it.

It assigns zero to primitive types and null to objects. **dataType variableName;**

The second way to declare a variable: This specifies its data type, reserves memory for it, and puts an initial value into that memory. The initial value must be of the correct data type.

dataType variableName = initialValue;

The first way to declare two variables: all of the same data type, reserves memory for each.

dataType variableNameOne, variableNameTwo;

The second way to declare two variables: both of the same data type, reserves

memory, and puts an initial value in each variable. **dataType**

variableNameI = initialValueI, variableNameII=initialValueII;

Variable name:

Use only the characters `_a` through `_z`, `_A` through `_Z`, `_0` through `_9`, character `_`, and character `_$`.

A name cannot include the space character.

Do not begin with a digit.

A name can be of any realistic length.

Upper and lower case count as different characters.

A name cannot be a reserved word (keyword).

A name must not previously be utilized in this block of the program.

3.3.1 Scope and Lifetime of variables:

So far, all of the variables used have been declared at the start of the main() method. However, Java allows variables to be declared within any block. A block defines a scope. Thus, each time you start a new block, you are creating a new scope. A scope determines what objects are visible to other parts of your program. It also determines the lifetime of those objects.

The scope defined by a method begins with its opening curly brace. However, if that method has parameters, they too are included within the method's scope. As a general rule, variables declared inside a scope are not visible (that is, accessible) to code that is defined outside that scope. Thus, when you declare a variable within a scope, you are localizing that variable and protecting it from unauthorized access and/or modification. Indeed, the scope rules provide the foundation for encapsulation. Scopes can be nested. For example, each time you create a block of code, you are creating a new, nested scope. When this occurs, the outer scope encloses the inner scope. This means that objects declared in the outer scope will be visible to code within the inner scope. However, the reverse is not true. Objects declared within the inner scope will not be visible outside it.

```
class Scope
{
public static void main(String args[]) {
int x; // known to all code within main x
= 10;
if(x == 10) { // start new scope int y
= 20; // known only to this block
// x and y both known here.
System.out.println("x and y: " + x + " " + y);
x = y * 2;
}
// y = 100; // Error! y not known here
// x is still known here.
System.out.println("x is " + x);
}}
```

3.4 Constant:

Constant means fixed value which is not change at the time of execution of program. In Java, there are two types of constant as follows:

Numeric Constants

Integer constant

Real constant

Character Constants

3.4.1 Integer Constant:

An Integer constant refers to a series of digits. There are three types of integer as follows:

a) Decimal integer

Embedded spaces, commas and characters are not allowed in between digits. For example:

23 411

7,00,000

17.33

b) Octal integer

It allows us any sequence of numbers or digits from 0 to 7 with leading 0 and it is called as octal integer.

For example:

011

00 0425

c) Hexadecimal integer

It allows the sequence which is preceded by 0X or 0x and it also allows alphabets from A to F or a to f (A to F stands for the numbers 10 to 15) it is called as Hexadecimal integer. For example:

0x7

00X

0A2B

3.4.2 Real Constant

It allows us fractional data and it is also called as floating point constant.

It is used for percentage, height and so on.

For example:

0.0234

0.777

-1.23

3.4.3 Character Constant

It allows us single character within pair of single quote.

For example:

'A'

'7'

'\'

3.4.4 String Constant

It allows us the series of characters within pair of double quote. For example:

—"WELCOME"

—"END OF PROGRAM "

—"BYE ...BYE "

—"A"

3.4.5 Symbolic constant:

In Java program, there are many things which is requires repeatedly and if we want to make changes

then we have to make these changes in whole program where this variable is used. For this purpose,

Java provides `_final` keyword to declare the value of variable as follows: Syntax: `final type Symbolic_name=value`; For example:

If I want to declare the value of `_PI` then:

```
final float PI=3.1459
```

the condition is, `Symbolic_name` will be in capital letter(it shows the difference between normal variable and sybolic name) and do not declare in method.

3.4.6 Backslash character constant:

Java support some special character constant which are given in following table.

Constant	Importance
<code>'\b'</code>	Back space
<code>'\t'</code>	Tab
<code>'\n'</code>	New line
<code>'\\'</code>	Backslash
<code>'\''</code>	Single coute
<code>'\"'</code>	Double coute

Table 4.13

3.5 Comments:

A **comment** is a note written to a human reader of a program. The program compiles and runs exactly the same with or without comments. Comments start with the two characters `—//` (slash slash). Those characters and everything that follows them on the same line are ignored by the java compiler.

Everything between the two characters `—/*` and the two characters `—*/` are unobserved by the compiler. There can be many lines of comments between the `—/*` and the `—*/`.

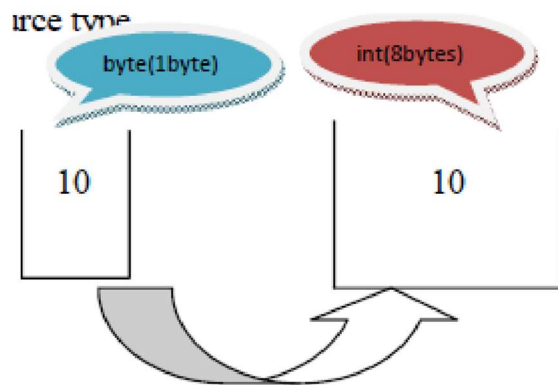
3.6 Type casting:

Type casting is basically an assignment of a value of one type to a variable of another type. There are two types of type casting : Implicit typecasting

- Explicit typecasting

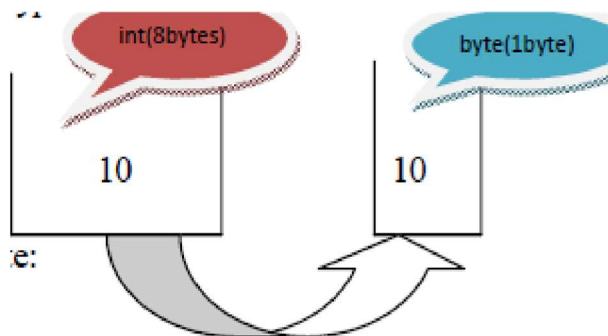
3.6.1 Implicit typecasting: when the destination type is larger than the source type

```
byte b=10;
int i=b//here the destination type is larger than source type
```



3.6.2 Explicit typecasting: when the source type is larger than destination type

```
int i=10;
byte b=i//here the destination type is smaller than source type
```



Here compiler will generate an error to overcome that we have to write:

```
int i=10;
byte b=(byte)i//explicitly convert the in type to byte type
```

3.6.3 Type Casting and Constant:

Constant value could not be changed, so we can store a constant bigger data type value to a smaller data type variable without type casing.

```
final int a=10; byte
```

```
b=a; // no error
```

Class:

The most important thing to understand about a class is that it defines a new data type. Once defined, this new type can be used to create objects of that type.

Class: A class is a user defined variable

A class can be defined as a template/blue print that describes the behaviors/states that object of its type support.

A class is a group of objects that has common properties. It is a template or blueprint from which objects are created.

A class in java can

- contain: ○ data
- member ○
 - method ○
 - constructor ○ block
- class and interface

```
class <class_name>{
  data member;  method;
}
```

3.7.2 Object: An entity that has state and behavior is known as an object. Object is an instance of a class.

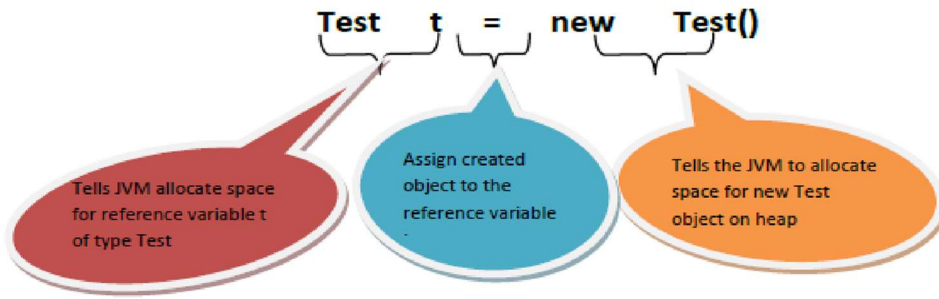
An object has three characteristics:

- state: represents data (value) of an object.
- behavior: represents the behavior (functionality) of an object such as deposit, withdraw etc.
- identity: Object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. But, it is used internally by the JVM to identify each object uniquely.

3.8 Instantiation : The process of creation of object is known as instantiation.

There are three steps when creating an object from a class:

- **Declaration:** A variable declaration with a variable name with an object type.
- **Instantiation:** The 'new' key word is used to create the object.
- **Assignment:** Assign the created object to the variable.



An object reference to another variable.

3.9 Primitive Variable:
1010

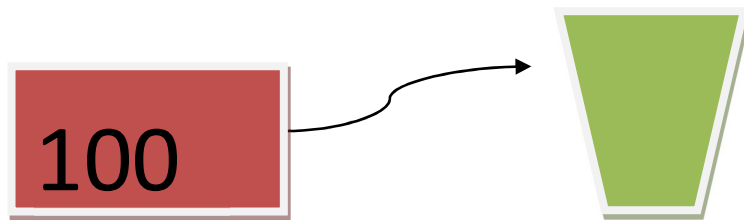


Figure 5.1

Primitive data value

Object: is a reference variable.

`Dog d=new Dog();`

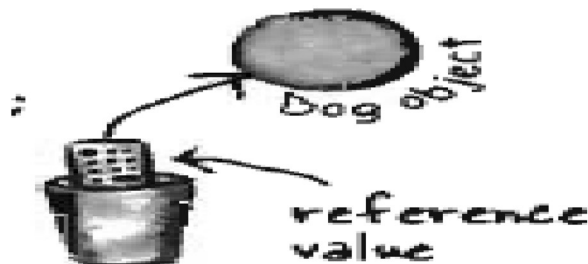


Figure 5.2

With primitive variables values are like: 1,2...1.1,'A' etc. and for the reference value of the variable is bits representing a way to get to a specific object.

A class can be defined as a template/blue print that describes the behaviors/states that object of its type support.

Class Contains:

- i. Member variable/constant
- ii. Method

3.10 A class may contain any numbers of following types of variables: **Local variables:** Variables defined inside methods, constructors or blocks are called local variables. The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed.

Instance variables: Instance variables are variables within a class but outside any method. These variables are instantiated when the class is loaded. Instance variables

can be accessed from inside any method, constructor or blocks of that particular class.

Class variables: Class variables are variables declared with in a class, outside any method, with the static keyword.

- A class can have any number of methods to access the value of various kinds of methods.

```

Class Test{
  int a; // instance variable
  static int b; // class or static variable
  void setValue(int i){ //method
    a=i;
  }
  void show(){ // method
    System.out.println(" the value of a="+a " and value of b="+b);
  }
  void testMethod(){ //method
    int a1=20;
    int b1=10;
    System.out.println("sum="+{a1+b1});
  }
}
public static void main(String[] aa){
  Test.b=90;
  Test t=new Test();
  t.setVal(80);
  t.show();
  t.testMethod();
}
}

```

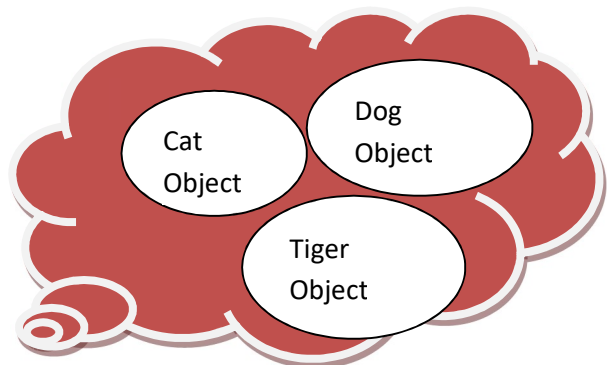
3.11 Stack and Heap

Heap : Java Heap space is used by java runtime to allocate memory to Objects and JRE classes. Whenever we create any object, it's always created in the Heap space. Any object created in the heap space has global access and can be referenced from anywhere of the application.

```

Cat c=new Cat();
Dog d=new Dog();
Tiger t=new Tiger();

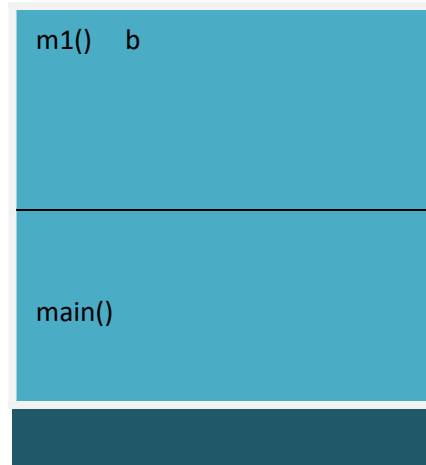
```



Stack :Java Stack memory is used for execution of a thread. They contain method specific values that are short-lived and references to other objects in the heap that are getting referred from the method. Stack memory is always referenced in LIFO (Last-In-First-Out) order. Whenever a method is invoked, a new block is created in the stack memory for the method to

hold local primitive values and reference to other objects in the method. As soon as method ends, the block becomes unused and become available for next method. Stack memory size is very less compared to Heap memory.

```
class Test {}  
public static void main(St  
    //  
}  
void m1(){  
    int b;  
}  
}
```



3.12 Local Variables:

Local variables reside in stack. They are of two types one is primitive and other is reference. The primitive local variables reside in stack and the reference local variable resides in stack referred to a real object resided in heap.

3.13 Instance Variable: They reside in the heap along with the object. They can also be of two types one in primitive and other is reference.

The primitive and reference instance variable resides in heap and the reference variable referred to the real object resides in object.

```
class Test { int a,b; // primitive
           instance variable Dog d; // reference
           instance variable void s(){ int
           c,d; // primitive local variable
           Cat c=new Cat(); // primitive reference variable
           }
           }
```

Difference between instance variable and Local variables:

1. Instance variables are declared inside class.
Local variables are declared inside method
2. Instance variables need not to be initialized before use (if they are not initialized default values are used)
Local variables must have to be initialized before use.

3.14 Method : Methods are the object oriented name of function

<modifier><return type> name of method(<parameters>)

- modifier – It defines the access type of the method and it is optional to use.
- returnType – Method may return a value.
- nameOfMethod – This is the method name. The method signature consists of the method name and the parameter list.

- Parameter List – The list of parameters, it is the type, order, and number of parameters of a method. These are optional, method may contain zero parameters.
- method body – The method body defines what the method does with the statements.

3.15 Constructor

Every class has a constructor. If we do not explicitly write a constructor for a class the Java compiler builds a default constructor for that class.

Each time a new object is created, at least one constructor will be invoked.

Features of Constructor:

- The name of constructor is same as class name where it is declared.
- It does not have any return type • It can not be called without new.
- It can not be static
- Constructors are called when we create object.

```
class Test_Const{  
  //  
  public static void main(String[] aa){  
    Test_Const tc=new Test_Const();//here default constructor of the class is called  
  }  
}
```

```
class Test_Const{  
  Test_Const(){ //user defined constructor  
    System.out.println("creating constructor");  
  }  
  public static void main(String[] aa){  
    Test_Const tc=new Test_Const();//here the user defined constructor is called  
  }  
}
```

Types of Constructor:

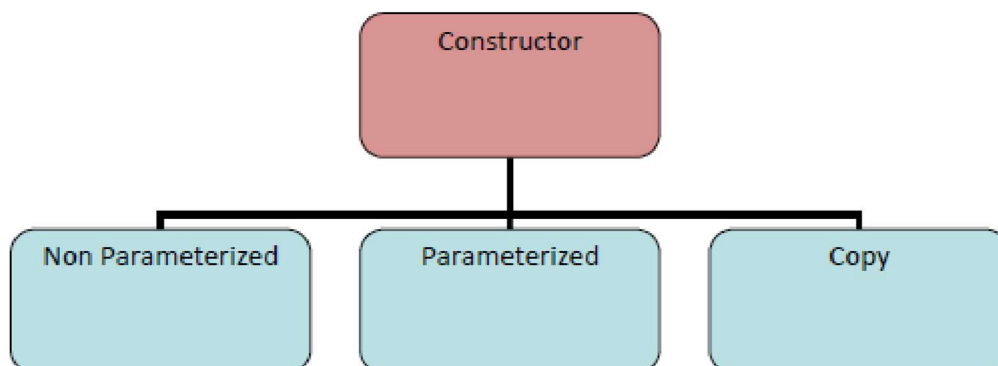


Figure 5.3

1. Non Parameterized Constructor:

The constructor which does not have any parameter is known as non parameterized constructor.

```
class Test_Const{
    Test_Const(){ //user defined non parameterized constructor
        System.out.println("creating constructor");
    }
    public static void main(String[] aa){
        Test_Const tc=new Test_Const();//here the user defined non parameterized constructor is called
    }
}
```

2. Parameterized Constructor:

The constructor which has parameter is known as parameterized constructor.

```
class Test_Const{
    Test_Const(int a){ //user defined parameterized constructor
        System.out.println("creating constructor");
    }
    public static void main(String[] aa){
        Test_Const tc=new Test_Const(10);//here the user defined parameterized constructor is called
    }
}
```

3. Copy Constructor

It is a special type of parameterized constructor where the parameter is the object of the same class. It is used to copy one object value to other object.

```
class Test_Const{
    Test_Const(int a){ //user defined parameterized constructor
        System.out.println("creating constructor");
    }
    Test_Const(Test_Const t){//user defined copy constructor
        t.a=this.a;
    }
    public static void main(String[] aa){
        Test_Const tc=new Test_Const(10);
        Test_Const tc1=new Test_Const(tc); //calling copy constructor
    }
}
```

All default constructors are non parametrized constructor but all non parameterized constructors are not default constructors.

Constructor to Initialization of Instance variable

Constructor can be used to initialize the instance variable during object creation time.

```
class Test_Const{
int a,b;
Test_Const(int a,int b){ //user defined parameterized constructor to initialize instance variable
this.a=a;
this.b=b;
}
public static void main(String[] aa){
Test_Const tc=new Test_Const(10,20);
}
}
```

this stands for the current object

Constructor Overloading

Constructor overloading is a feature by which a class can contain multiple constructors with different signature (Parameters) and different purpose.

Conditions for Constructor Overloading:

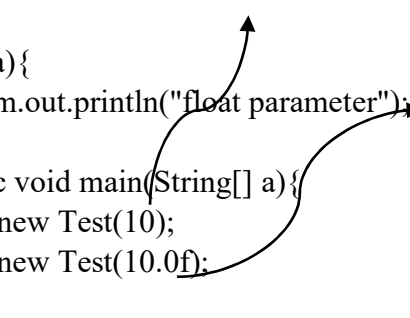
- No. of parameters different
- Types of Parameters different
- Sequence of occurrence of parameters different

No. of Parameters different:

```
class Test{
Test(){
System.out.println("I am a Non Parameterized Constructor");
}
Test(int a){
System.out.println("I am a Parameterized Constructor");
}
public static void main(String[] a){
Test t1=new Test();
Test t2=new Test(10);
}
}
```

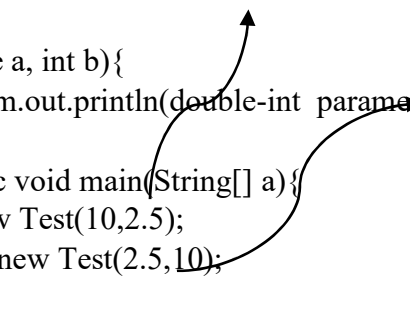

Types of Parameters different

```
class Test{
Test(int a){
    System.out.println("int parameter");
}
Test(float a){
    System.out.println("float parameter");
}
public static void main(String[] a){
    Test t1=new Test(10);
    Test t2=new Test(10.0f);
}
}
```



Sequence of occurrence of parameters different

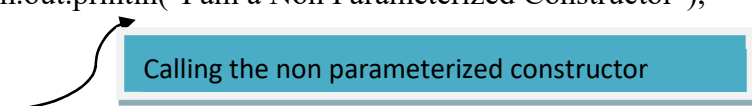
```
class Test{ Test(int a
, double b){
    System.out.println("int-double parameter");
}
Test(double a, int b){
    System.out.println("double-int parameter");
}
public static void main(String[] a){
    Test t1=new Test(10,2.5);
    Test t2=new Test(2.5,10);
}
}
```



3.16 this()- Constructor

this() is something by which we can call the constructor of the same class. Suppose by accessing one constructor, the programmer may require the functionality of other constructors also but by creating one object only. For this, Java comes with this(). "this()" is used to access one constructor from another "within the same class".

```
class Test{
    Test(){
        System.out.println("I am a Non Parameterized Constructor");
    }
    Test(int
a){
        this();
    }
}
```



```

        System.out.println("I am a Parameterized Constructor");
    }
    public static void main(String[] a){
        Test t1=new Test();
        Test t2=new Test(10);
    }
}

```

Rules of using this()

1. If included, this() statement must be the first one in the constructor. You cannot write anything before this() in the constructor.
2. With the above rule, there cannot be two this() statements in the same constructor (because both cannot be the first).
3. this() must be used with constructors only, that too to call the same class constructor (but not super class constructor).

Chaining of Constructor using this():

Calling one constructor from another one is known as chaining of constructor.

```

Class Test_Const{
int a,b,c;
Test_Const(int a,int b){
    this.a=a;
this.b=b;    }
Test_Const(int a,int b, int c){
    this(a,b); calling first constructor from the second one using this
this.c=c;
    }}

```

Constructor chaining can help us to avoid redundant code.

3.17 Garbage Collection

In java, garbage means unreferenced objects.

Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects.

To do so, we were using free() function in C language and delete() in C++. But, in java it is performed automatically. So, java provides better memory management.

Advantage of Garbage Collection

- It makes java memory efficient because garbage collector removes the unreferenced objects from heap memory.
- It is automatically done by the garbage collector (a part of JVM) so we don't need to make extra efforts.

Disadvantages

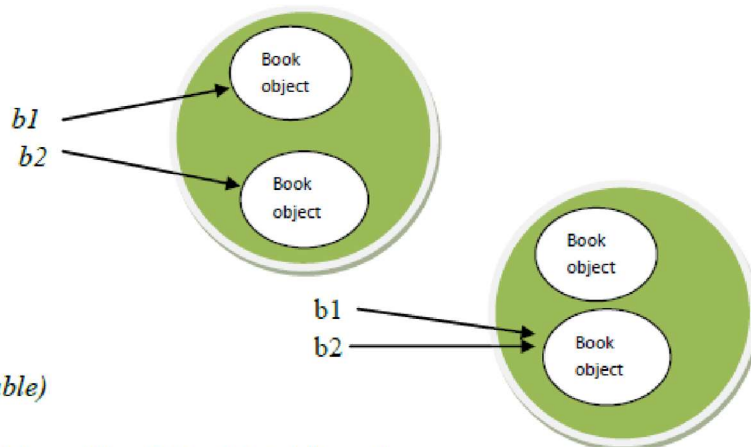
Typically, garbage collection has certain disadvantages, including consuming additional resources, performance impacts, possible stalls in program execution, and incompatibility with manual resource management.

```
Book b1=new Book();
Book b2=new Book();
```

Reference: 2
Object: 2

```
b1=b2;
```

Reference: 2
Active Object : 1
Abandoned Object : 1(not reachable)



Here the first object is not reachable so this will be deleted from the memory.

Runtime class has a method void gc() that initiate garbage collection

protected void finalize(): runs just before garbage collection

finalize() method :

The finalize() method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing. This method is defined in Object class as:

```
protected void finalize(){}
```

System.gc() :

The java.lang.System.gc() method runs the garbage collector. Calling this 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. Following is the declaration for `java.lang.System.gc()` method

3.18 Array

An *array* is a group of like-typed variables that are referred to by a common name. Arrays of any type can be created and may have one or more dimensions. A specific element in an array is accessed by its index. Arrays offer a convenient means of grouping related information

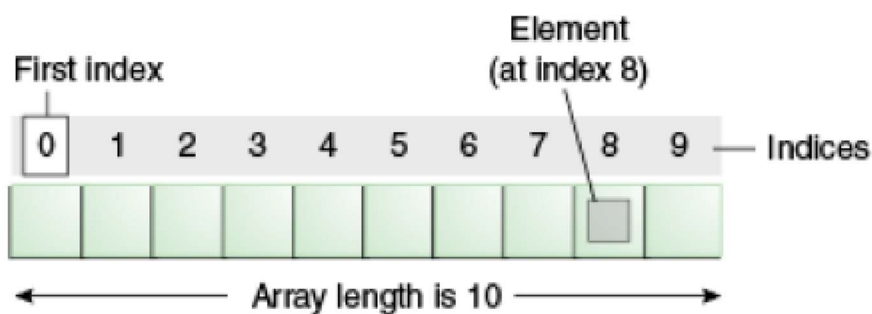


Figure 5.4

Each item in an array is called an element, and each element is accessed by its numerical index. As shown in the preceding illustration, numbering begins with 0. The 9th element, for example, would therefore be accessed at index 8.

Properties of Array in JAVA:

- Not Homogeneous
- Java Provides a final property 'length' that gives the length of array
- Array is treated as object of type `java.lang.Object` class
- Array has default values as:

0 int null

reference type at
creation time.

Creation of One dimensional array:

```
<datatype> [] <array name>= new <datatype> [<size>]
```

Followings are the way to create one dimensional array:

1. `int [] a;`
`a=new int[10];`
2. `int [] a=new int[10];`
3. `int a[]=new int[10];`
4. `int[] a={1,2,3,4} // initialization at time of creation.`

Program on array:

```
class Test_Array{
    public static void main(String[] aa){
        int[] a=new int[10];
        System.out.println("Initializing the array.....");
        for(int i=0;i<a.length;i++){
            a[i]=i*5;}
        System.out.println("Values in the array.....");
        for(int i=0;i<a.length;i++){
            System.out.println(a[i]);}
    }
}
```

`<datatype> [][] <arrayname>=new <datatype>[<rowsize>][<columnsize>];` Column size is optional.

We can create 2D array having different numbers of columns in each rows .A 2D array is nothing but array of array.

1. `int[][] a=new int[5][5];`
2. `int[][] a=new int[3][];`
`a[0]=new int[2];`
`a[1]=new int[3];`
`a[2]=new int[4];`

a[0][0]	a[0][1]		
a[1][0]	a[1][1]	a[1][2]	
a[2][0]	a[2][1]	a[2][2]	a[2][3]

Length : 2D array:

a.length: Numbers of rows in the array a[i].length:
Numbers of Columns in the ith row. Programme
on 2D array:

```
class Test_Array2D{
    public static void main(String[] aa){
        int[][] a=new int[2][];
        a[0]=new int[2];
        a[1]=new int[3];
        System.out.println("Initializing the array.....");
        for(int i=0;i<a.length;i++){
            for(int j=0;j<a[i].length;j++){
                a[i][j]=i*j;
            }
        }
        System.out.println("Values in the array.....");
        for(int i=0;i<a.length;i++){
            for(int j=0;j<a[i].length;j++){
                System.out.println(a[i][j]);
            }
        }
    }
}
```

Anonymous array:

In java it is possible to create an array without name.

```
System.out.println(new int[] {1,2,3,4}.length); //display 4.
```

Reusable array

In Java it is possible to reuse a already created array, i.e here x had initially size 3 but we can create an array of size 90 with same name and initial values will be lost.

```
int[] x={1,2,3};
x=new int[90];
```

Method is the object oriented name of function. Method is a block of code having similar functionality.

Method has two different phase:

i. Defining a method ii.

Calling a method.

Defining a method :

Method in java have the following signature :

<modifier><return type> name of method(<parameters>)

- modifier – It defines the access type of the method and it is optional to use.
- returnType – Method may return a value.
- nameOfMethod – This is the method name. The method signature consists of the method name and the parameter list.
- Parameter List – The list of parameters, it is the type, order, and number of parameters of a method. These are optional, method may contain zero parameters.
- method body – The method body defines what the method does with the statements.

Calling a Method:

Non static methods are called through the object of the class and static methods are called through class name

```
class Ret_Object{
    A m(){
        return new A();
    }
    static void s(){//}
}
public static void main(String[] aa){
    Ret_Objcet r=new Ret_Object();
    A o1=r.m();//calling non static method
    Ret_Object.s();// calling static method
}
}
class
A{
//}
```

Method of java are always call by value.

Passing Object to method:

```
class Length{
    Int feet,inches; Length(int
    feet,int inches){
```

```

this.feet=feet;
this.inches=inches;
boolean check(Length l){
return (if(this.feet*12+this.inches)>l.feet*12+l.inches)
} void display(boolean
b){ if(b==true)
System.out.println("calling object is greater"); else
System.out.println("passing object is greater"); }
public static void main(String[] aa){ Length
l1=new Length(2,3); Length l2=new
Length(3,4); l1.display(l1.check(l2));
}
}

```

Returning Object through a method

```

class Ret_Object{
A m(){ return
new A();
}
public static void main(String[] aa){
Ret_Objcet r=new Ret_Object();
A o1=r.m();
}
}
class A{
//}

```

Passing and returning Array through method:

```

class AddArray{
int []a1=new int[3]; int
[]a2=new int[3]; void
setValues(){
System.out.println("Initilizing two
arrays"); a1[0]=10; a1[1]=11; a1[2]=100;
a2[0]=12; a2[1]=11 a2[2]=101;} int[]
add(){ int[] r=new int[a1.length]; for(int
i=0;i<a1.length;i++){ r[i]=a1[i]+a2[i];
} return r; } void
display(int[] a){
System.out.print("The result= ");
for(int i=0;i<a.length;i++)
System.out.print(a[i]+" ");
} public static void main(string[]
aa){ AddArray a=new AddArray();

```



```
a.setValues(); int b[]=a.add();
a.display(b);
}
}
```

Method Overloading

If a class has multiple methods by same name but different signature, it is known as **Method Overloading**.

Condition of Method Overloading:

1. Number of Parameters should be different
2. Types of Parameters should be different
3. Sequence of occurrence of parameters should be different

```
class Method_Overloading{
    void show(){
        System.out.println("method with no argument");
    }
    void show(int a){
        System.out.println("method with one int argument");
    }
    void show(int a,double b){
        System.out.println("method with one int and one double argument");
    }
    void show(double b,int a){
        System.out.println("method with one double and one int argument");
    }
    public static void main(String[] aa){
        Method_Overloading m=new Method_Overloading();
        m.show();
        m.show(10);
        m.show(1,2.5);
        m.show(4.6,9);
    }
}
```

Return type does not play any role in method overloading:

```
void s(int a) {/}
int s(int a){/}
this is not valid overloading
```

Advantages of method overloading:

Method overloading **increases the readability of the program.**

Method Overloading with Type promotion (Implicit typecasting):

```
class Method_Overloading{
    void show(){
        System.out.println("method with no argument");
    }
    void show(double b){
        System.out.println("method with one double argument");
    }

    public static void main(String[] aa){
        Method_Overloading m=new Method_Overloading();
        m.show();
        m.show(10);
    }
}
```

In the above example there is no parameter with int parameter, so when we call show(10) , this 10 can be stored in a double value by implicit type casting so show(double) method will be called.

Method Overloading is a compile time polymorphism:

When we overload multiple methods then which version of the methods will be called that decision is taken at the compilation time so this is known as compile time polymorphism or static binding or early binding.

```
class Method_Overloading{
    void show(int a,double b){
        System.out.println("method with one int and one double argument");
    }
    void show(double b,int a){
        System.out.println("method with one double and one int argument");
    }
    public static void main(String[] aa){
        Method_Overloading m=new Method_Overloading();
        m.show(1,2);
    }
}
```

Here the called method support both the defined version so there is a ambiguity of which ver. of the method will be called and we will get the error at compilation time which proves that the decision of which ver. of methods will be called taken at compilation time so this is compile time polymorphism

main() can be overloaded

```
class Overloading{
    public static void main(int a){
        System.out.println(a);
    }
    public static void main(String[] aa){
        System.out.println("Calling main method");
        main(10);
    }
}
```

Output:

```
Calling main method
10
```

3.20 Static

Static Key word in java is basically used as a modifier. We can apply java static keyword with variables, methods, blocks and nested class. Static keyword is associated with class rather than object.

Non Static block:

Non static blocks are simple block enclosed in `{//}`. They executes every times an object is created.

```
class Test{
    Test(){
        System.out.println("Object created");
    }
    {
        System.out.println(" I am a non static block");
    }
    public static void main(String[] a){
        Test t1=new Test();
        Test t2=new Test();
    }
}
```

```
Output :
I am a non static block
Object created
I am a non static block
Object created
```

Static Block:

A *static block* is a normal block of code enclosed in braces, `{ }`, and preceded by the static keyword. Here is an example:

```
static {
// whatever code is needed for initialization goes here
}
```

A class can have any number of static initialization blocks, and they can appear anywhere in the class body. The runtime system guarantees that static initialization blocks are called in the order that they appear in the source code.

```
class Test{
    Test(){
        System.out.println("Constructor");
    }
    {
        System.out.println("Non Static Block");
    }
    static {
        System.out.println("Static Block");
    }
    public static void main(String[] aa){
        Test t=new Test();
        Test tt=new Test();
    }
}
```

Output:

Static Block

Non Static Block

Constructor

Non Static Block

Constructor

Whenever a class loaded into memory static block will execute in their order. This is only once as class is loaded only once during running.

For each object creation the non static blocks will execute.

If we initialize any static variable that static block also execute as the class loaded.

```
class T{
    static{ System.out.println("initialize");}
    static int a;
}
class B{
public static void main(String[] aa){
    T.a=10;
}
}
```

Output:

initialize

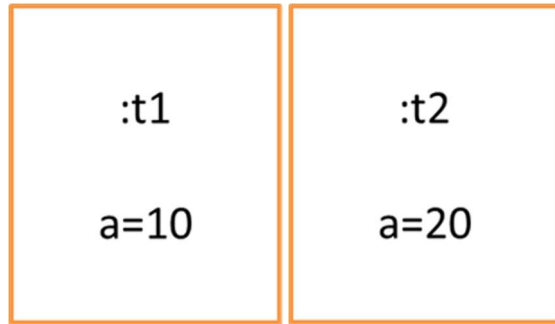
Static variable:

Static variables are the class level variables. All the instances of a class share the same copy of a static variable, which in turn actually saves memory. Usually the common properties of a class are stored in static variable. Static variables are initialized when a class is loaded, before any object is created.

```
class Test{
    int a;
    static int b;
    Test(int a){
        this.a=a;
    }
}
public static void main(String[] a){
    Test.b=100;
    Test t1=new Test(10);
    Test t2=new Test(20);
}
}
```

Here b is declared as static so single copy of b is shared by both two objects.

b=100



Static final variables are constant

Static Method:

They are the class level method. Behavior of static method does not depend on value of instance variables.

Static methods are called as

<classname>.<methodname>

```
class Test{
    void show(){
        System.out.println("non static method");
    }
    static void display(){
        System.out.println("static method");
    }
}
public static void main(String[] a){
    Test t1=new Test();
    Test t2=new Test();
    t1.show();
    Test.display();
}
```

Static method could not access the non static data directly. They are accessed with objects. Static methods could not identify which value of instance variable is associated with that method so it needs object to call it. It is true for the non static method also.

```
class Test{
    Test(){
        System.out.println("Constructor");
    }
    {
        System.out.println("Non Static Block");
    }
    static {
        System.out.println("Static Block");
    }
    public static void main(String[] aa){
        Test t=new Test();
        Test tt=new Test();
    }
}
```

Output:

```
Static Block
Non Static Block
Constructor
Non Static Block
Constructor
```

Whenever a class loaded into memory static block will execute in their order. This is only once as class is loaded only once during running.

For each object creation the non static blocks will execute.

If we initialize any static variable that static block also execute as the class loaded.

3.21 Nested Class

A nested class is a member of its enclosing class. Non-static nested classes (inner classes) have access to other members of the enclosing class, even if they are declared private.

Why Use Nested Classes?

- **It is a way of logically grouping classes that are only used in one place:** If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together. Nesting such "helper classes" makes their package more streamlined.
- **It increases encapsulation:** Consider two top-level classes, A and B, where B needs access to members of A that would otherwise be declared `private`. By hiding class B within class A, A's members can be declared private and B can access them. In addition, B itself can be hidden from the outside world.
- **It can lead to more readable and maintainable code:** Nesting small classes within top-level classes places the code closer to where it is used.

```
private static int a=10;
void show_Outer(){
    System.out.println(new Inner().aa);
    System.out.println("Outer class method");
}
class Inner { private
int aa=100;
    void show_Inner(){
        System.out.println(a);
        System.out.println("Inner class method");
    }
}
}
}
class Nested{
public static void main(String[] aa){
    new Outer().show_Outer();
    Outer.Inner o=new Outer().new Inner();
}
}
```

```
class Outer{
```

Static Nested Classes

As with class methods and variables, a static nested class is associated with its outer class. And like static class methods, a static nested class cannot refer directly to instance variables or methods defined in its enclosing class: it can use them only through an object reference.

A static nested class interacts with the instance members of its outer class (and other classes) just like any other top-level class. In effect, a static nested class is behaviorally a top-level class that has been nested in another top-level class for packaging convenience.

```
class Outer{
    private static int a=10;
    void show_Outer(){
        System.out.println(new Inner().aa);
        System.out.println("Outer class method");
        //show_Inner();
    }
    static class Nest{
        private int aa=100;
        static void show_Inner(){
            System.out.println(a);
            System.out.println("Nested class method");
        }
    }
}
class Nested{
    public static void main(String[] aa){
        new Outer().show_Outer();
        Outer.Nest o=new Outer.Nest();
        Outer.Nest.show_Inner();
    }
}
```


Local Class:

Local classes are classes that are defined in a *block*, which is a group of zero or more statements between balanced braces. You typically find local classes defined in the body of a method.

```
class
Test_Local{
public void s(){
class Local{ void
show(){
    System.out.println("LOCAL");
    }
Local l=new
Local(); l.show();
}
}
public static void main(String[]a
){ Test_Local t=new
Test_Local(); t.s();
}
}
```

Anonymous Class:

Anonymous classes enable you to make your code more concise. They enable you to declare and instantiate a class at the same time. They are like local classes except that they do not have a name. Use them if you need to use a local class only once.

```
class A1{
  interface
  I{
    void s();
  }public void s1(){ I i =new I(){
    public void s(){
      System.out.println("Anonymous class");
    }
  };
  i.s();
}
public static void main(String[]aa){
  new A1().s1();
}
}
```

Multiple Choice Questions

1. Which of these operators is used to allocate memory to array variable in Java? a) malloc
b) alloc
c) new
d) new malloc
2. What will this code print? `int arr[] = new int [5]; System.out.print(arr);` a) 0
b) value stored in `arr[0]`.
c) 00000
d) Class name@ hashCode in hexadecimal form
3. Which of these is an incorrect Statement?
a) It is necessary to use new operator to initialize an array.
b) Array can be initialized using comma separated expressions surrounded by curly braces.
c) Array can be initialized when they are declared.
d) None of the mentioned
4. Which of these keywords is used to make a class?
a) class
b) struct

- c) int
 - d) None of the mentioned
5. Which of the following is a valid declaration of an object of class Book?
- a) `Book obj = new Book();`
 - b) `Book obj = new Book;`
 - c) `obj = new Book();`
 - d) `new Book obj;`
6. Which of these statement is incorrect?
- a) Every class must contain a `main()` method.
 - b) Applets do not require a `main()` method at all.
 - c) There can be only one `main()` method in a program.
 - d) `main()` method must be made public.
7. Which of the following statements is correct?
- a) Public method is accessible to all other classes in the hierarchy
 - b) Public method is accessible only to subclasses of its parent class
 - c) Public method can only be called by object of its class.
 - d) Public method can be accessed by calling object of the public class.
8. What is the return type of a method that does not returns any value? a) int
- b) float
 - c) void
 - d) double
9. Which of the following is a method having same name as that of it's class? a) finalize
- b) delete
 - c) class
 - d) constructor
10. Which method can be defined only once in a program?
- a) main method
 - b) finalize method
 - c) static method
 - d) private method
11. Which of these statement is incorrect?
- a) All object of a class are allotted memory for the all the variables defined in the class.
 - b) If a function is defined public it can be accessed by object of other class by inheritance.
 - c) `main()` method must be made public.
 - d) All object of a class are allotted memory for the methods defined in the class.

12. What is the return type of Constructors?
 - a) int
 - b) float
 - c) void
 - d) None of the mentioned

13. Which of the following is a method having same name as that of its class?
 - a) finalize
 - b) delete
 - c) class
 - d) constructor

14. Which operator is used by Java run time implementations to free the memory of an object when it is no longer needed?
 - a) delete
 - b) free
 - c) new
 - d) None of the mentioned

15. Which function is used to perform some action when the object is to be destroyed?
 - a) finalize()
 - b) delete()
 - c) main()
 - d) None of the mentioned

16. Which of these can be overloaded?
 - a) Methods
 - b) Constructors
 - c) All of the mentioned
 - d) None of the mentioned

17. Which of these is correct about passing an argument by call-by-value process?
 - a) Copy of argument is made into the formal parameter of the subroutine.
 - b) Reference to original argument is passed to formal parameter of the subroutine.
 - c) Copy of argument is made into the formal parameter of the subroutine and changes made on parameters of subroutine have effect on original argument.
 - d) Reference to original argument is passed to formal parameter of the subroutine and changes made on parameters of subroutine have effect on original argument

18. What is the process of defining a method in terms of itself, that is a method that calls itself?
 - a) Polymorphism
 - b) Abstraction
 - c) Encapsulation
 - d) Recursion

```
19. void start() { A a = new A(); B b = new B(); a.s(b); b = null; /* Line 5 */ a =
    null; /* Line 6 */
    System.out.println("start completed"); /* Line 7 */
}
```

When is the B object, created in line 3, eligible for garbage collection?

1. After line 5
 2. After line 6
 3. After line 7
 4. There is no way to be absolutely certain.
20. Which is true about a method-local inner class?
- a) It must be marked final.
 - b) It can be marked abstract.
 - c) It can be marked public.
 - d) It can be marked static.

Module 4

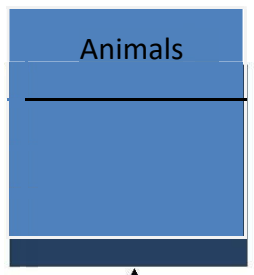
Reusable Properties

4.1 Inheritance

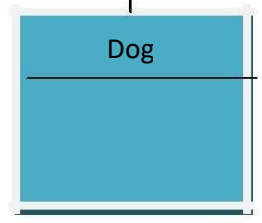
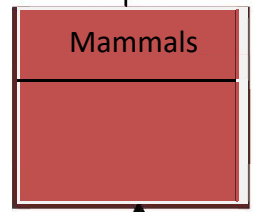
Inheritance is one of the cornerstones of object-oriented programming because it allows the creation of hierarchical classifications. Using inheritance, you can create a general class that defines traits common to a set of related items. This class can then be inherited

by other, more specific classes, each adding those things that are unique to it. In the terminology of Java, a class that is inherited is called a *super class*. The class that does the inheriting is called a *subclass*. Therefore, a subclass is a specialized version of a super class. It inherits all of the instance variables and methods defined by the super class and adds its own, unique elements.

Here Mammals inherits Animals and Dog inherits Mammals. Animals class is the generalized version and Dog class is the most specific version. Animals class is the super class for both Mammals and Dog and Mammals class is the super class for Dog and Dog is the sub class.

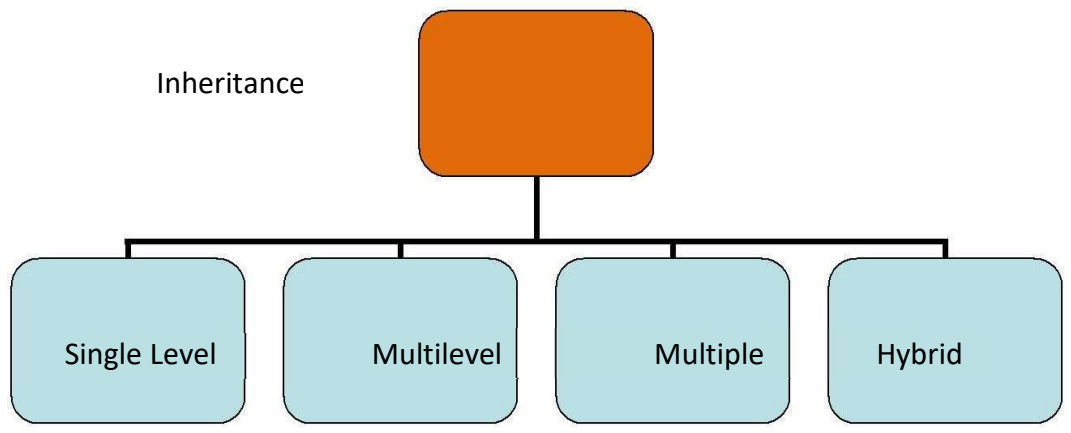


Object class is the universal super class for all java classes

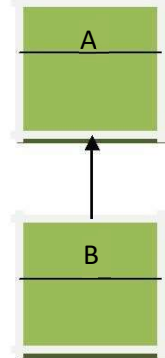


4.1.2 Types of

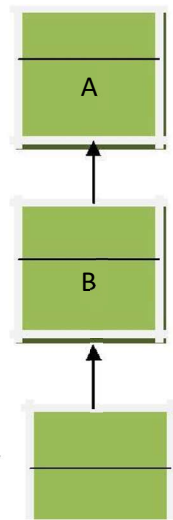
Inheritance:



Single Level Inheritance: When a class extends another one class only then we call it a single inheritance. The below flow diagram shows that class B extends only one class which is A. Here A is a **parent class** of B and B would be a **child class** of A.



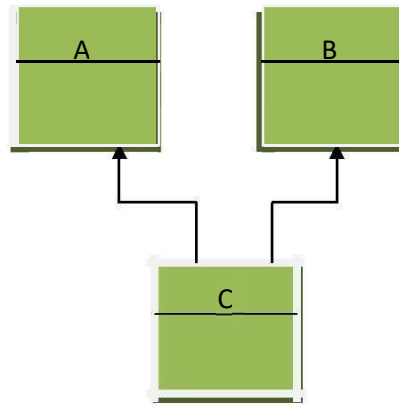
Multilevel Inheritance: **Multilevel inheritance** refers to a mechanism in OO technology where one can inherit from a derived class, thereby making this derived class the base class for the new class. As you can see in below flow diagram C is subclass or child class of B and B is a child class of A.



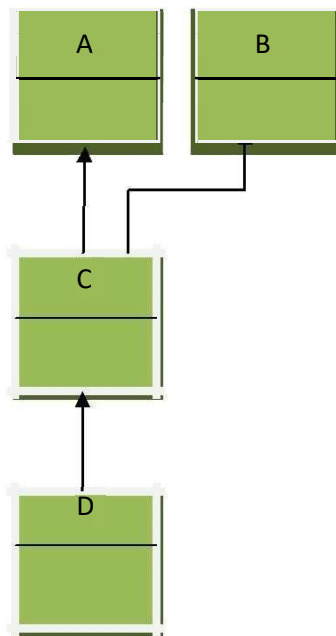


C

Multiple Inheritance : Multiple Inheritance refers to the concept of one class extending (Or inherits) more than one base class. The inheritance we learnt earlier had the concept of one base class or parent. The problem with “multiple inheritance” is that the derived class will have to manage the dependency on two base classes.



Hybrid Inheritance: It is the combination of any three of the above inheritances.



It is a combination of multilevel and multiple inheritance.

Java does not support multiple Inheritance

4.2 Java and Inheritance

Java supports single level and multi level inheritance. Keyword extends is used established inheritance relationship among classes. Derived class can access all non private properties of base class

```
class Base{
    void show_Base(){
        System.out.println("Base class");
    }
}
class Derived extends Base{
    void show_Derived(){
        System.out.println("Derived class");
    }
}
```

Derived class accessing Member Variable

```
class Base{
int a;
Base(int a){
    this.a=a;
}
void show_Base(){
    System.out.println("Base class");}
}
class Derived extends Base{
void show_Derived(){
    System.out.println("Derived class");}
}
class Test_Inheritance{
public static void main(String[] ar){
    Base b=new Base(10);
    Derived d=new Derived();
    System.out.println(d.a);
}
}
```

Derived class accessing Methods


```
class Base{
int a;
Base(int a){
this.a=a;
}
void show_Base(){
System.out.println("Base class");
}
}
class Derived extends Base{
void show_Derived(){
System.out.println("Derived class");
}
}
class Test_Inheritance{
public static void main(String[] ar){
Base b=new Base(10);
Derived d=new Derived();
d.show_Derived();
d.show_Base();
}
}
```

4.3 super

Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword super. Super has two general forms. The first calls the superclass' constructor. The second is used to access a member of the super class that has been hidden by a member of a subclass.

Accessing member variable using super

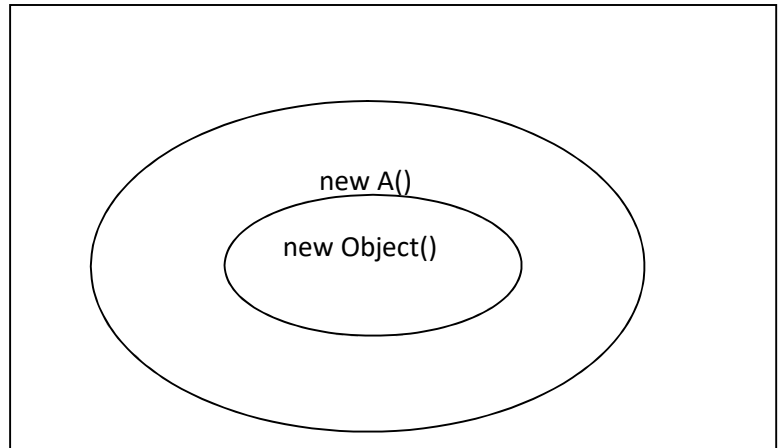
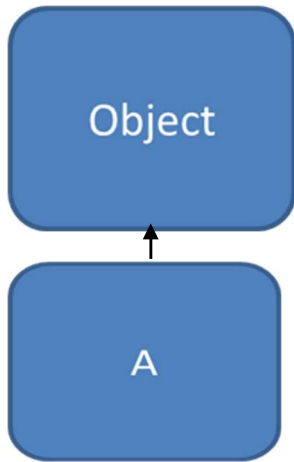
```
class Base{
int a;
void show_Base(){
System.out.println("Base class");
}
}
class Derived extends Base{
void show_Derived(){
super.a=100;
System.out.println("Derived class");
}
}
class Test_Inheritance{
public static void main(String[] ar){
Base b=new Base(10);
Derived d=new Derived();
d.show_Derived();
}
}
```

Accessing Method using super

```
class Base{
    int a;
    void show_Base(){
        System.out.println("Base class");
    }
}
class Derived extends Base{
    void show_Derived(){
        super.show_Base();
        System.out.println("Derived class");
    }
}
class Test_Inheritance{
    public static void main(String[] ar){
        Derived d=new Derived();
        d.show_Derived();
    }
}
```

java.lang.Object is the universal super class. When an object of a derived class is created, object of all its bases classes up to Object(class) is created.

```
class A { A() {
```



4.4 Role of Super Class Constructor in Object Life Cycle:

In a class hierarchy, constructors are called in order of derivation, from superclass to subclass. Further, since `super()` must be the first statement executed in a subclass' constructor, this order is the same whether or not `super()` is used. If `super()` is not used, then the default or parameterless constructor of each superclass will be executed.

Output:

Inside A's constructor
Inside B's constructor
Inside C's constructor

```
System.out.println("Inside A's constructor.");  
  
}  
}  
  
class B extends A { B() {  
System.out.println("Inside B's constructor.");  
}  
}  
  
class C extends B { C() {  
System.out.println("Inside C's constructor.");  
}  
}  
  
class CallingCons {    public static void  
    main(String args[]) { C c = new C();  
  
}  
}
```

How Polymorphism actually works

Base class reference can store the parent class object.

```
Animal[] a=new Animal[4]   a[0]=new Dog();  
a[1]=new Cow();   a[2]=new Lion();   a[4]=new  
Tiger();
```

Sometimes animal behaves like Dog, sometimes like Cow, sometimes like Lion and sometimes like Tiger. It is the polymorphic behavior

Both Base class and Derived class have same named method. Method associated with the object would be called.

4.6 Why Overridden Methods?

Overridden methods allow Java to support run-time polymorphism. Polymorphism is essential to object-oriented programming for one reason: it allows a general class to specify methods that will be common to all of its derivatives, while allowing subclasses to define the specific implementation of some or all of those methods. Overridden methods are another way that Java implements the “one interface, multiple methods” aspect of polymorphism.

Part of the key to successfully applying polymorphism is understanding that the super classes and subclasses form a hierarchy which moves from lesser to greater specialization. Used correctly, the super class provides all elements that a subclass can use directly. It also defines those methods that the derived class must implement on its own. This allows the subclass the flexibility to define its own methods, yet still enforces a consistent interface. Thus, by combining inheritance with overridden methods, a super class can define the general form of the methods that will be used by all of its subclasses.

Dynamic, run-time polymorphism is one of the most powerful mechanisms that object oriented design brings to bear on code reuse and robustness. The ability of existing code libraries to call methods on instances of new classes without recompiling while maintaining a clean abstract interface is a profoundly powerful tool.

```
class Base{
void s(){
    System.out.println("Base class method"); }
}
class Derived extends Base{
void s(){
    System.out.println("Derived class method"); }
}
class Super_Main{
public static void main(String[] a){
    Base b=new Base();
    Derived d=new Derived();
    b.s();
    d.s();
}
}
```

4.7 Condition for Overriding

- Argument list must be same.
- Derived class methods access specifier must be less restrictive than base class method.
- Return type compatible: Either same or return type of derived class method would be derived class of base class method return type.

```
class Base{
public void s(int a){
    System.out.println("Base class method");
}
}
class Derived extends Base{
void s(int b){
    System.out.println("Derived class method");
}
}
```

```
class Base{
public void s(int a){
    System.out.println("Base class method");
}
}
class Derived extends Base{
void s(int b){
    System.out.println("Derived class method");
}
}
```

```

class Base{
    Base s(int a){
        System.out.println("Base class method");
        return new Base();
    }
}
class Derived extends Base{
    Derived s(int b){
        System.out.println("Derived class method");
        return new Derived();
    }
}

```

4.8 Dynamic Method Dispatch:

Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time. Dynamic method dispatch is important because this is how Java implements *run-time polymorphism*.

In the previous example as we call show() method over the p object which is object of class Child but reference of class Parent, the child class version of show() is being called so method overriding is run time polymorphism i.e. which version of method will be called that decision is taken at run time.

It refers to as *dynamic binding* also.

Static and Overriding

Usually in overriding the method calls depend on the object type, but in case of static method, method calls depend on the reference type. So static methods could not be overridden.

```

class Parent {
    static void
    show(){

        System.out.println("Parent class");
    }
}
class Child extends Parent{
    static void show(){

        System.out.println("Child class");
    }
}
class Test_Overriding{
    public static void main(String[]
    aa){ Parent

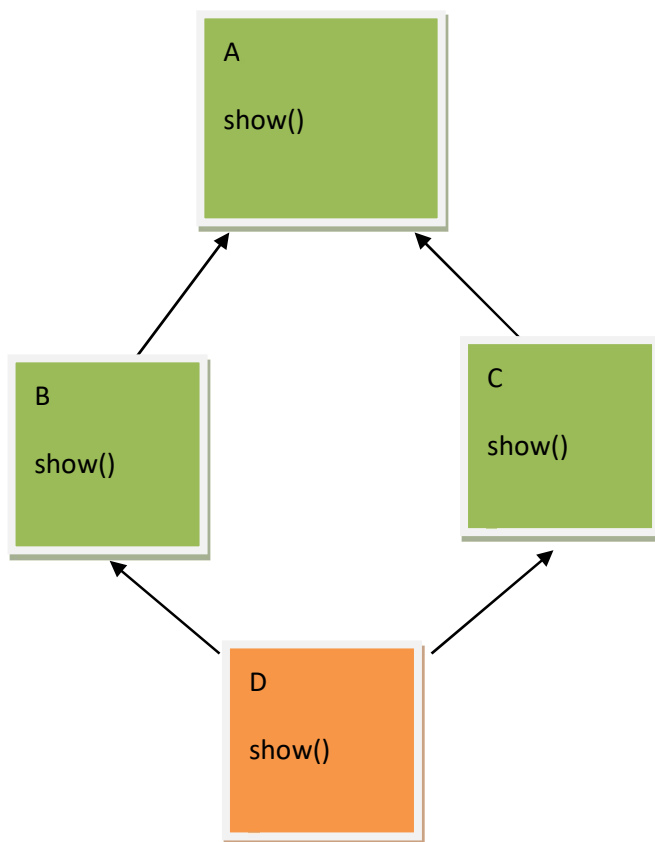
```

Output:

Parent
class


```
p=new Child();  
p.show();//It will call parent class version }
```

}



In this example say class A has a method `show()` and that is being overridden by two of its child class B and C. Now there is a class D which has two parents B and C, now if we call the `show()` method over C class object then there would be ambiguity i.e. which version of `show()` will be called. This problem is called Deadly Diamond of Death (DDD) problem. Java is supposed to be simple so it avoids multiple inheritance.

4.9 Final and Inheritance

- ⦿ If a class declared final it could not be inherited
- ⦿ If a method declared as final it could not be overridden.

Why Java does not support multiple Inheritance:

4.10 Abstract Class

There are situations in which you will want to define a super class that declares the structure of a given abstraction without providing a complete implementation of every method. That is, sometimes you will want to create a super class that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details. Such a class determines the nature of the methods that the subclasses must implement. One way this situation can occur is when a super class is unable to create a meaningful implementation for a method.

You can require that certain methods be overridden by subclasses by specifying the **abstract** type modifier.

4.11 Abstract Method: If a method does not have any definition then the method is abstract method.

Abstract class: An abstract class is a class that is declared abstract—it may or may not include abstract methods. Abstract classes cannot be instantiated, but they can be sub classed.

4.11.1 Features of Abstract class:

1. Abstract class may or may not contain abstract methods
2. If a class contains one abstract method it must be declared as abstract
3. It is the responsibility of the child class that extends abstract class to define all the abstract methods otherwise the child class must be declared as abstract.
4. Abstract class cannot be instantiated. 5. Abstract class cannot be final.

```
abstract class Abs{ abstract
void show();
void dis(){
System.out.println("It is a non abstract method");}
}
class Con extends Abs{
void show(){
System.out.println("It is a non abstract method"); }
}
class Test_Abs{
public static void main(String [] aa){
Con c=new Abs(); c.show();
c.dis();} }
```

6.11.2 Advantages and Disadvantages of Abstract class:

Advantages:

The advantage of using an abstract class is that you can group several related classes together as siblings.

Grouping classes together is important in keeping a program organized and understandable. An Abstract class is a way to organize inheritance, sometimes it makes no sense to implement every method in a class (i.e. a predator class), it may implement some to pass to its children but some methods cannot be implemented without knowing what the class will do (i.e. eat() in a Tiger class). Therefore, abstract classes are templates for future specific classes.

Disadvantages:

A disadvantage is that abstract classes cannot be instantiated, but most of the time it is logical not to create a object of an abstract class. **Why the abstract class could not be final:**

Abstract class may or may not contain abstract methods. If it contains any method that method should be defined by the child class of the abstract class, but if an abstract class is final it could not be inherited so the undefined method remains undefined, so abstract class could not be final.

Illegal use of static abstract:

If a method is abstract it could not be declared as static, because static methods could not be overridden, i.e always the parent class version of the method would be called, but in case of abstract method the parent class does not have any body so always the child class version must be called, so it is illegal to use static abstract modifier together.

Lecture 4

4.12 Interface

Interface is a fully abstract class. An interface is a collection of abstract methods.

Features of Interface:

1. All the methods are by default public abstract
2. All the members are static and final
3. Interface cannot be instantiated
4. The child class which implements the interface should define all the methods of the interface.
5. One interface can extends other interface.

4.13 Implementing Interface

Interface avoids DDD problem:

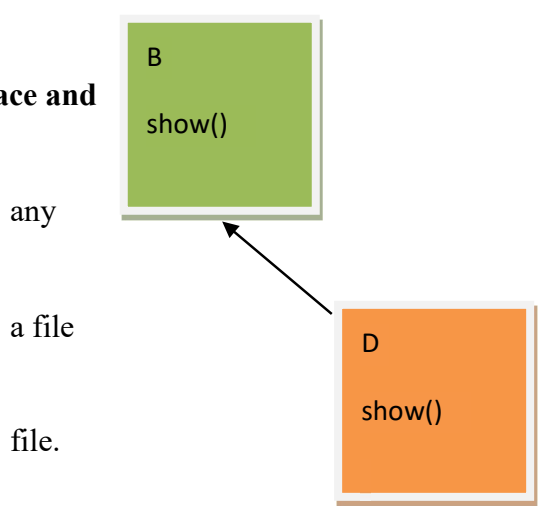
Here A and B are interfaces and C is a class that implements both A and B. Both A and B has method named show() but as they are interface the methods are undefined now it's the responsibility of C to define the show() , so there is no ambiguity of which version of show () will be called as show() is

```
interface Inter1 { public abstract void show(); static
final int A=10;
}
class Con implements Inter1 { public void show(){
System.out.println("implementing interface");
}}
class Test_Inter{
public static void main(String[] aa){
Con c=new Con(); c.show();
}}
```

A class can implements more than one interfaces

Here A and B are interfaces and C is a class that implements both A and B. Both A and B has method named show() but as they are interface the methods are undefined now it's the responsibility of C to define the show() , so there is no ambiguity of which version of show () will be called as show in only defined in C.

4.14 Interface and



Similarities of Class:

- An interface can contain number of methods.
- An interface is written in with a **.java** extension, with the name of the interface matching the name of the
- The byte code of an interface appears in a **.class** file.

- Interfaces appear in packages, and their corresponding byte code file must be in a directory structure that matches the package name.

4.15 Dissimilarities of Interface and Class:

- You cannot instantiate an interface.
- An interface does not contain any constructors.
- All of the methods in an interface are abstract.
- An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
- An interface is not extended by a class; it is implemented by a class.
- An interface can extend multiple interfaces.

4.16 Advantages and Disadvantages of Abstract class:

Advantages:

- Interfaces are mainly used to provide polymorphic behavior.
- Interfaces function to break up the complex designs and clear the dependencies between objects.

Disadvantages:

- Java interfaces are slower and more limited than other ones.
- Interface should be used multiple number of times else there is hardly any use of having them.

4.17 Package

A Package can be defined as a grouping of related types (classes, interfaces, enumerations and annotations) providing access protection and name space management.

Some of the existing packages in Java are::

- **java.lang** - bundles the fundamental classes
- **java.io** - classes for input , output functions are bundled in this package

Programmers can define their own packages to bundle group of classes/interfaces, etc. It is a good practice to group related classes implemented by you so that a programmer can easily determine that the classes, interfaces, enumerations, annotations are related.

Since the package creates a new namespace there won't be any name conflicts with names in other packages. Using packages, it is easier to provide access control and it is also easier to locate the related classes.

Packages are categorized in two form:

1. Built in Packages (java.lang, java.io etc)
2. Userdefined packages

4.18 Creating a package:

To create a package is quite easy: simply include a package command as the first statement in a Java source file. Any classes declared within that file will belong to the specified package. The package statement defines a name space in which classes are stored

This is the general form of the package statement:


```
package pkg;
```

Here, pkg is the name of the package. For example, the following statement creates a package called MyPackage.

```
package MyPackage;
```

You can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period. The general form of a multileveled package statement is shown here:

```
package pkg1[.pkg2[.pkg3]];
```

A package hierarchy must be reflected in the file system of your Java development system. For example, a package declared as

```
package java.awt.image;
```

needs to be stored in java\awt\image in a Windows environment. Be sure to choose your package names carefully. You cannot rename a package without renaming the directory in which the classes are stored.

4.19 Compile and Running :

```
package jis1;  
public class  
T1{ public void
```

```
s(){  
    System.out.println("Creating a package");  
}  
  
public static void main(String[]  
a){ T1 t=new T1();  
    t.s();  
}
```

//SUPPOSE PROGRAMS STORE IN D:\JP//

To Compile

```
D:\JP\jis1>javac T1.java
```

To run the file:

```
<<Set class path>>  
D:\JP> set classpath=D:\JP
```

Run:

```
D:\JP> java jis1.T1
```

The import Keyword:

If we want to access classes from other packages we need import statement.

import <packageName>.<subPackageName>.*
importing all the classes under the package.

or

import <packageName>.<subPackageName>.<class name>
importing specific under the package.

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

```
public class T1{  
    public static void main(String[] a){  
        Date d=new Date();  
        System.out.println(d.toString());  
    }  
}
```

If we are using import statement then we have to use fully qualified class name:

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

4.20 Accessing class from user defined different package:

```
package jis1;
public class
T1{ public
void s(){
    System.out.println("Creating a package");
}
}
```

```
package jis2;
import jis1.*;
public class
T2{
    public static void main(String[]
a){ T1 t=new T1();
    t.s();
}
}
```

4.21 Compilation and Running:

//SUPPOSE PROGRAMS STORE IN D:\JP//

To Compile

1. D:\JP\jis1>javac
T1.java <<Set class
path>>
D:\JP> set classpath=D:\JP

2. D:\JP\jis2>javac T2.java

To run the file:

D:\JP>java jis2.T2

***** Running Package Program without creating package manually*****

1. D:\JP> javac -d.T1.java
2. D:\JP> javac -d.T2.java
3. D:\java jis2.T2

Java's import statement doesn't include the file, as #include in C/C++ does, it just tells the compiler where to look for its class references.

4.12 Static import:

In order to access static members, it is necessary to qualify references with the class they came from. For example, one must say: `double r = Math.cos(Math.PI * theta);`

In order to get around this, people sometimes put static members into an interface and inherit from that interface. This is a bad idea. In fact, it's such a bad idea that there's a name for it: the *Constant Interface Antipattern*. The problem is that a class's use of the static members of another class is a mere implementation detail. When a class implements an interface, it becomes part of the class's public API. Implementation details should not leak into public APIs.

The static import construct allows unqualified access to static members *without* inheriting from the type containing the static members. Instead, the program *imports* the members, either individually:

```
import static java.lang.Math.PI;  
or en masse: import static  
java.lang.Math.*;
```

Once the static members have been imported, they may be used without qualification: `double r = cos(PI * theta);`

The static import declaration is analogous to the normal import declaration. Where the normal import declaration imports classes from packages, allowing them to be used without package qualification, the static import declaration imports static members from classes, allowing them to be used without class qualification.

Only use it when you'd otherwise be tempted to declare local copies of constants, or to abuse inheritance (the Constant Interface Antipattern). In other words, use it when you require frequent access to static members from one or two classes. If you overuse the static import feature, it can make your program unreadable and unmaintainable, polluting its namespace with all the static members you import. Readers of your code (including you, a few months after you wrote it) will not know which class a static member comes from. Importing all of the static members from a class can be particularly harmful to readability; if you need only one or two members, import them individually. Used appropriately, static import can make your program more readable, by removing the boilerplate of repetition of class names.

4.13 Access specifiers

	Private	Default	Protected	Public
Same class	YES	YES	YES	YES
Same package subclass	NO	YES	YES	YES
Same package non-subclass	NO	YES	YES	YES
Different package subclass	NO	NO	YES	YES
Different package <u>non-subclass</u>	NO	NO	NO	YES

Classes and packages are both means of encapsulating and containing the name space and scope of variables and methods. Packages act as containers for classes and other subordinate packages. Classes act as containers for data and code. The class is Java's smallest unit of abstraction. Because of the interplay between classes and packages, Java addresses four categories of visibility for class members:

- Subclasses in the same package
- Non-subclasses in the same package
- Subclasses in different packages
- Classes that are neither in the same package nor subclasses

The three access specifiers, **private**, **public**, and **protected**, provide a variety of ways to produce the many levels of access required by these categories.

Multiple Choice Questions

1. Which of this keyword must be used to inherit a class?
 - a) super
 - b) this
 - c) extent
 - d) extends
2. Which of these keywords is used to refer to member of base class from a sub class? a)
 - a) upper
 - b) super
 - c) this
 - d) None of the mentioned
3. A class member declared protected becomes member of subclass of which type? a)
 - a) public member
 - b) private member
 - c) protected member
 - d) static member
4. Which of these is correct way of inheriting class A by class B?
 - a) class B + class A {}
 - b) class B inherits class A {}
 - c) class B extends A {}
 - d) class B extends class A {}
5. Which of these keyword can be used in subclass to call the constructor of superclass? a)
 - a) super
 - b) this
 - c) extent
 - d) extends
6. What is the process of defining a method in subclass having same name & type signature as a method in its superclass? a) Method overloading
 - a) Method overloading
 - b) Method overriding
 - c) Method hiding
 - d) None of the mentioned
7. Which of these keywords can be used to prevent Method overriding? a) static
 - a) static
 - b) constant
 - c) protected
 - d) final

8. Which of these is correct way of calling a constructor having no parameters, of superclass A by subclass B? a) `super(void);`
b) `superclass.();`
c) `super.A();`
d) `super();`
9. Which of these is supported by method overriding in Java?
a) Abstraction
b) Encapsulation
c) Polymorphism
d) None of the mentioned
10. Which of these keywords is used to define packages in Java?
a) `pkg`
b) `Pkg`
c) `package`
d) `Package`
11. Which of these is a mechanism for naming and visibility control of a class and its content?
a) Object
b) Packages
c) Interfaces
d) None of the Mentioned.
12. Which of this access specifier can be used for a class so that its members can be accessed by a different class in the same package? a) `Public`
b) `Protected`
c) `No Modifier`
d) `All of the mentioned`
13. Which of these access specifiers can be used for a class so that its members can be accessed by a different class in the different package? a) `Public`
b) `Protected`
c) `Private`
d) `No Modifier`
14. Which of the following is correct way of importing an entire package 'pkg'? a) `import pkg.`
b) `Import pkg.`
c) `import pkg.*`
d) `Import pkg.*`
15. Which of the following is incorrect statement about packages?

- a) Package defines a namespace in which classes are stored.
- b) A package can contain other package within it.
- c) Java uses file system directories to store packages.
- d) A package can be renamed without renaming the directory in which the classes are stored.

Module 5

Exception and Multithreading

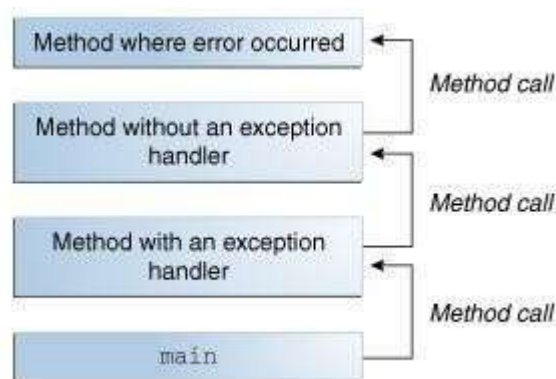
5.1 Exception

The term *exception* is shorthand for the phrase "exceptional event."

An *exception* is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.

When an error occurs within a method, the method creates an object and hands it off to the runtime system. The object, called an *exception object*, contains information about the error, including its type and the state of the program when the error occurred. Creating an exception object and handing it to the runtime system is called *throwing an exception*.

After a method throws an exception, the runtime system attempts to find something to handle it. The set of possible "something" to handle the exception is the ordered list of methods that had been called to get to the method where the error occurred. The list of methods is known as the *call stack* (see the next figure).



The runtime system searches the call stack for a method that contains a block of code that can handle the exception. This block of code is called an *exception handler*. The search begins with the method in which the error occurred and proceeds through the call stack in the reverse order in which the methods were called. When an appropriate handler is found, the runtime system passes the exception to the handler. An exception handler is considered appropriate if the type of the exception object thrown matches the type that can be handled by the handler.

The exception handler chosen is said to *catch the exception*. If the runtime system exhaustively searches all the methods on the call stack without finding an appropriate exception handler, as shown in the next figure, the runtime system (and, consequently, the program) terminates.

Using exceptions to manage errors has some advantages over traditional error-management techniques.

To understand how exception handling works in Java, you need to understand the three categories of exceptions:

5.1.1 Checked exceptions: A checked exception is an exception that is typically a user error or a problem that cannot be foreseen by the programmer. For example, if a file is to be opened, but the file cannot be found, an exception occurs. These exceptions cannot simply be ignored at the time of compilation.

5.1.2 Runtime exceptions: A runtime exception is an exception that occurs that probably could have been avoided by the programmer. As opposed to checked exceptions, runtime exceptions are ignored at the time of compilation.

Errors: These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise.

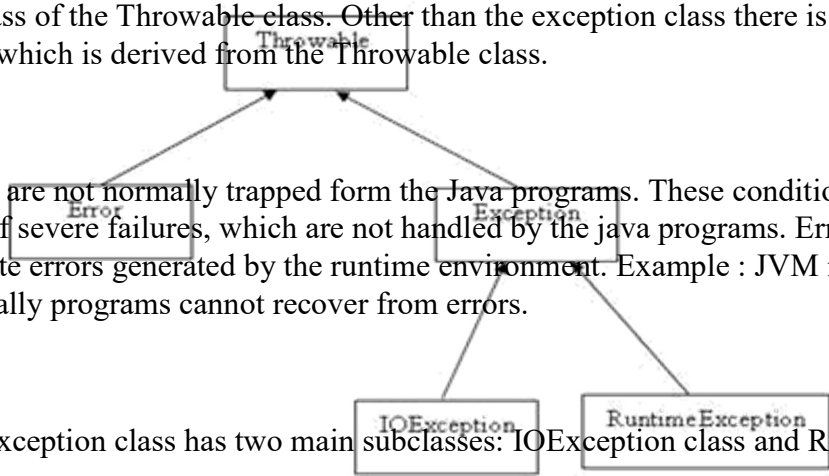
They are also ignored at the time of compilation.

5.2 Exception Hierarchy:

All exception classes are subtypes of the java.lang.Exception class. The exception class is a subclass of the Throwable class. Other than the exception class there is another subclass called Error which is derived from the Throwable class.

Errors are not normally trapped from the Java programs. These conditions normally happen in case of severe failures, which are not handled by the java programs. Errors are generated to indicate errors generated by the runtime environment. Example : JVM is out of Memory. Normally programs cannot recover from errors.

The Exception class has two main subclasses: IOException class and RuntimeException Class.



5.3 Examples of some Unchecked(Runtime) checked and Exception

Runtime Exception

Exception	Description
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
NullPointerException	Invalid use of a null reference.

Checked Exception

Exception	Description
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the Cloneable interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.

5.4 Exception Handling

5.4.1 try-catch

If an exceptional condition occurs in code the code will terminate. But it is possible to handle the exception so that rest of the code will execute properly:

```
try{ code that can generate exceptions
} catch(Exception e)
{ code that execute if exception occurs}
```

```
class Test_Exp{
public static void main(String[] aa){
int a=10;
try{
int c=a/0;
System.out.println("Main method"); }catch(Exception e){ e.printStackTrace();}
System.out.println("End of main"); }
}
```

Here as exception is handled so "End of main this statement will execute"

Multiple catch with try:

It is possible to use multiple catch statement associated with one try.

Conditions to use multiple catch with single try:

The order of the caught exception class should be from child class to parent class, reverse is not valid.

i.e This is valid

```
try{// }
catch(
Arith
meticE
xcepti
on
ae){
ae.prin
tStack
Trace(
);}
catch(
ArrayI
ndexO
utOfB
ounds
Except
ion
ae){
ae.prin
tStack
Trace(
);}
catch(
Except
ion e){
```



```
e.print  
StackT  
race();  
}
```

This is not valid

```
try{// }
```

```
catch(Exception e){ e.printStackTrace();} catch(ArithmeticException  
ae){ ae.printStackTrace();}
```



```
catch(ArrayIndexOutOfBoundsException ae){
    ae.printStackTrace();}class Test_Exp{    public static void
main(String[] aa){ int a=10; int b[]=new int[5]; try{ int c=a/0; int d=b[9];
System.out.println("Main method");
} catch(ArithmeticException ae){ ae.printStackTrace();}
catch(ArrayIndexOutOfBoundsException ae){ ae.printStackTrace();} catch(Exception e){
e.printStackTrace();}
System.out.println("End of main");
} }
class Test_Exp{
public static void main(String[] aa){
int a=10; int b[]=new int[5]; try{ int
c=a/0; int d=b[9];
System.out.println("Main method");
} catch(ArithmeticException ae){ ae.printStackTrace();}
catch(ArrayIndexOutOfBoundsException ae){ ae.printStackTrace();} catch(Exception e){
e.printStackTrace();}
System.out.println("End of main"); }
}
```

Nested try-catch :

The **try** statement can be nested. That is, a **try** statement can be inside the block of another **try**. Each time a **try** statement is entered, the context of that exception is pushed on the stack. If an inner **try** statement does not have a **catch** handler for a particular exception, the stack is unwound and the next **try** statement's **catch** handlers are inspected for a match. This continues until one of the **catch** statements succeeds, or until all of the nested **try** statements are exhausted.

```
class Test_Exp{
public static void main(String[] aa){
int a=10;
int b[]=new int[5];
try{
int c=a/0;
try{
int d=b[9];
}catch(ArrayIndexOutOfBoundsException ae){ ae.printStackTrace();}
System.out.println("Main method");
}catch(ArithmeticException ae){ ae.printStackTrace();}
System.out.println("End of main");
}
}
```

If inner block does not having a matching catch block it will look for the matching catch block in the outer catch block, and if finds a matching catch exception will handle properly.

But if outer catch block does not have any matching catch block it cannot look for the inner catch block so exception will not handle.

5.5 throw:

You can throw explicitly exception(user defined or built-in) by throw statement.

Throwing Built-in Exception:

```
class Test_Exp{
public static void main(String[] aa){
int a=10;
try{
throw new ArithmeticException()
System.out.println("Main method");
}catch(Exception e){ e.printStackTrace();}
System.out.println("End of main");
}
```

5.6 Throwing User defined Exception:

```
class MyExp extends Exception{
public void show(){
System.out.println("User defined Exception");
}}
class Test_Exp{
public static void main(String[] aa){
try{
int a=10;
if(a>=10)
{
MyExp m=new MyExp();
throw m;
}}catch(MyExp m){m.show();}
}}
```

5.7 Rethrow of Exception:

It is possible to throw of one exception from another exception

```
Class Exp extends Exception{
public static void show(){
try{ throw new ArithmeticException();
}catch(ArithmeticException ae){
System.out.println(ArithmeticExp”);
throw e;
}

public static void main(String[] aa){ try{
show();
} catch(ArithmeticException ae){
System.out.println(“Rethrow exp caught”);
}
}
```

5.8 Chained Exception:

The chained exception feature allows you to associate another exception with an exception. This second exception describes the cause of the first exception. For example, imagine a situation in which a method throws an **ArithmeticException** because of an attempt to divide by zero. However, the actual cause of the problem was that an I/O error occurred, which caused the divisor to be set improperly.

The chained exception methods added to Throwable are `getCause()` and `initCause()`.

These methods are shown in Table 10-3 and are repeated here for the sake of discussion.

Throwable `getCause()`

Throwable `initCause(Throwable causeExc)`

The `getCause()` method returns the exception that underlies the current exception. If there

is no underlying exception, null is returned. The `initCause()` method associates `causeExc` with the invoking exception and returns a reference to the exception. Thus, you can associate a cause with an exception after the exception has been created. However, the cause exception can be set only once. Thus, you can call `initCause()` only once for each exception object. Furthermore, if the cause exception was set by a constructor, then you can't set it again.

using `initCause()`.

```
class Exp extends Exception{
public void show(){
ArithmeticException ae=new ArithmeticException();
ae.initCause(new IOException());
throw ae;
}
public static void main(String[] aa){
try{
new Exp().show();
}catch(ArithmeticException a){System.out.println(a.getSource());}
}
}
```

Output:

IOException

Thread is a light weight process. It Runs on JVM. Java platform is designed to support concurrent programming with basic concurrency support.

Computer execution unit is basically divided into two category Process and Thread.

5.9.1 Process:

An application consists of one or more processes. A *process*, in the simplest terms, is an executing program. Processes have their own copy of the data segment of the parent process. Each process is started with a single thread, known as the primary thread. A process can have multiple threads in addition to the primary thread.

5.9.1 Thread:

A *thread* is the basic unit to which the operating system allocates processor time. A thread can execute any part of the process code, including parts currently being executed by another thread. Threads have direct access to the data segment of its process

Processes are heavily dependent on system resources available while threads require minimal amounts of resource, so a process is considered as heavyweight while a thread is termed as a lightweight process.

5.10 Thread Life Cycle

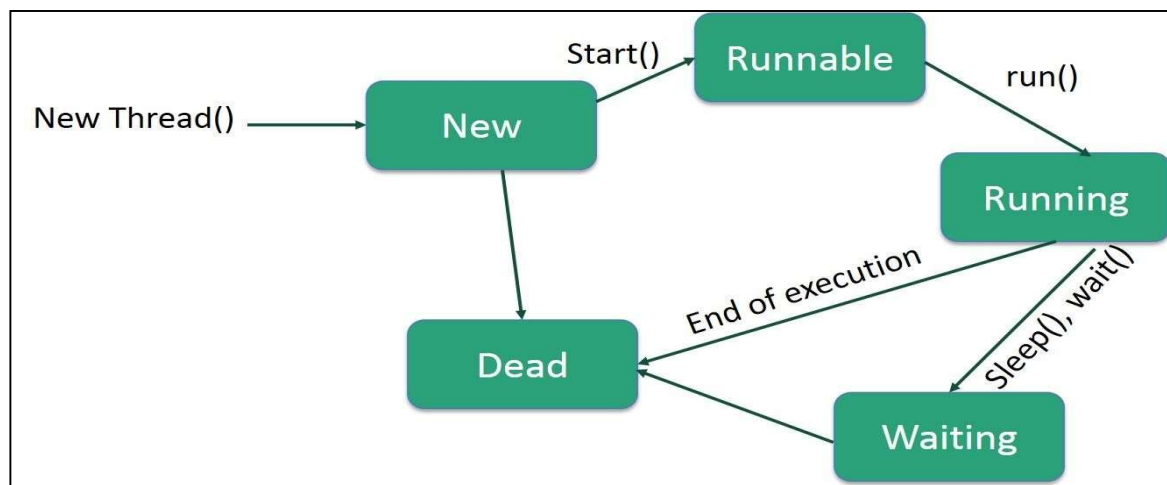


Figure 8.3

New – A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread.

Runnable – After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.

Waiting – Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.

Timed Waiting – A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.

Terminated (Dead) – A runnable thread enters the terminated state when it completes its task or otherwise terminates.

Current Thread: Every Java program works on a thread. If user dose not create any thread still there is main thread.

```
class Test_Thread{
    public static void main(String[] aa){
        Thread t=Thread.currentThread();
        System.out.println(t);
        t.setName("JIS");
        System.out.println(t);
    }
}
```

5.11 Thread Call Stack

Each thread maintains their separate call stack. When start method is called on a thread a separate call stack creates.

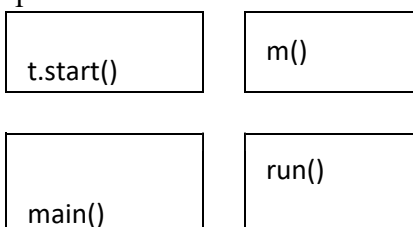


Figure 8.4

5.12 Important Methods used for developing Thread Application

Thread Class Constructors :

Thread() : allocates new thread object
 Thread (java.lang.Runnable r): allocates new thread object and r is the object whose run method will be called.

Public Thread(java.lang.String s) : allocates new thread object and n is the name of new thread.

Method	Purpose
void start()	Causes this thread to begin execution; the Java Virtual Machine calls the run method of this thread.
void join()	Waits for this thread to die.
void run()	If this thread was constructed using a separate Runnable run object, then that Runnable object's run method is called; otherwise, this method does nothing and returns.
void stop()	Stopping a thread with Thread.stop causes it to unlock all of the monitors that it has locked (as a natural consequence of the unchecked
void sleep(long millis)	Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers.
void interrupt()	Interrupts this thread.
void isAlive()	Tests if this thread is alive.
void setName(name)	Changes the name of this thread to be equal to the argument name.
void setPriority(priority)	Changes the priority of this thread.
String getName()	Returns this thread's name.
int getPriority()	Returns this thread's priority.
static void yeild()	A hint to the scheduler that the current thread is willing to yield its current use of a processor.

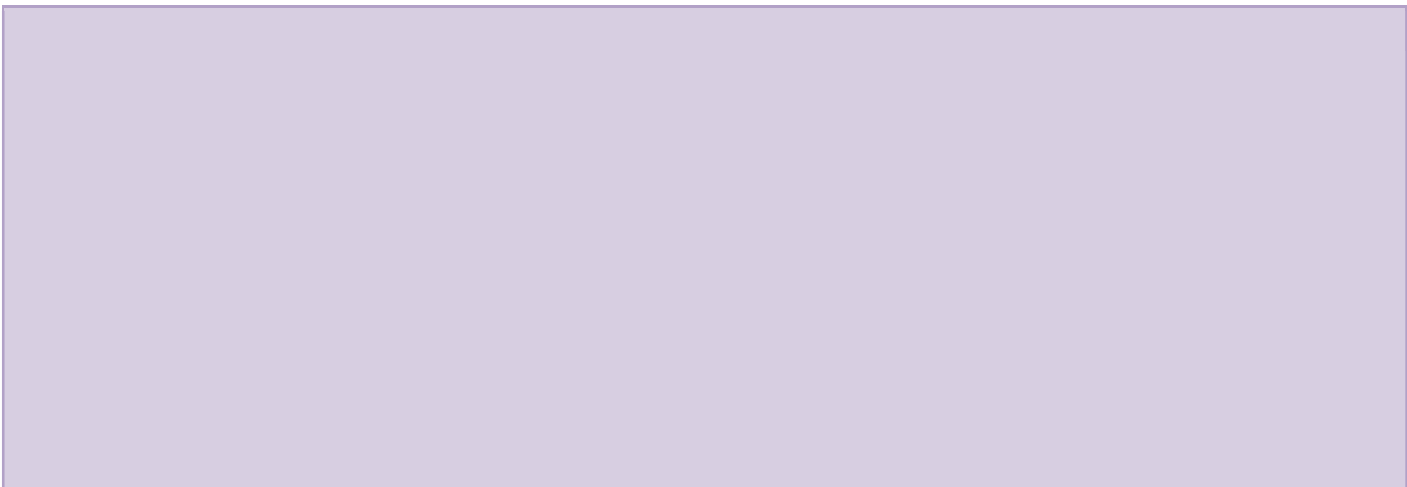
Table 8.3

java.lang.Object.notify() : wakes up a single thread that is waiting on this object's monitor.

void wait() : This method causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.

5.13 Creation of Multiple Thread Creation:

Thread in a Java program can be created either by extending the thread class or implementing the Runnable interface.



```
class MyThread1 extends Thread {
    String name;
    MyThread(String name) {
        this.name=name;
    }
    public void run(){
        for(int i=0;i<5;i++){
            System.out.println(name);
            try{
                Thread.sleep(2000);
            }
        }
    }
}
```

Creating Multiple Thread by extending the Thread Class

Threads are created by calling the start() method.



```
class MyThread implements Runnable {
    String s;
    MyThread(String s) {
        this.s=s;
    }
    public void run() {
        for(int i=0;i<5;i++){

            System.out.println(s);
```

5.14 Creating Multiple Threads by implementing the Runnable Interface

Runnable interface contains only the run() method .

It is always better to implements Runnable than extending the Thread.

5.15 Thread Priorities

Java assigns to each thread a priority that determines how that thread should be treated with respect to the others. Thread priorities are integers that specify the relative priority of one thread to another. As an absolute value, a priority is meaningless; a higher-priority thread doesn't run any faster than a lower-priority thread if it is the only thread running. Instead, a thread's priority is used to decide when to switch from one running thread to the next. This is called a context switch.

A thread can voluntarily relinquish control. This is done by explicitly yielding, sleeping, or blocking on pending I/O. In this scenario, all other threads are examined, and the highestpriority thread that is ready to run is given the CPU.

A thread can be preempted by a higher-priority thread. In this case, a lower-priority thread that does not yield the processor is simply preempted—no matter what it is doing— by a higherpriority thread. Basically, as soon as a higher-priority thread wants to run, it does. This is called preemptive multitasking. In cases where two threads with the same priority are competing for

CPU cycles, the situation is a bit complicated. For operating systems such as Windows, threads class

```
TestPriority{ public static void main(String[] a){    Mythread m=new Mythread();
    m.start();
} } class Mythread extends Thread{
public void run(){    setPriority(2);
System.out.println(getPriority());
}
}
```


of equal priority are time-sliced automatically in round-robin fashion. For other types of operating systems, threads of equal priority must voluntarily yield control to their peers. If they don't, the other threads will not run.

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called synchronization. As you will see, Java provides unique, language-level support for it. Key to synchronization is the concept of the monitor (also called a semaphore). A monitor is an object that is used as a mutually exclusive lock, or mutex. Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor. These other threads are said to be waiting for the monitor. A thread that owns a monitor can reenter the same monitor if it so desires.

```

void show(String s){
    System.out.print(s+" JIS");
    try{
        Thread.sleep(2000);
    } catch (Exception e){}
    System.out.println("CE");
}

String s;
Test t;
MyThread (String s, Thread t){
    this.s=s;
    this.t=t;
}
public void run(){

```

5.16 Synchronization

```

Mark C:\Windows\system32\cmd.exe
E:\JavaPro>java Test_Thread3
welcome to JISwelcome to JISwelcome to JISCE
CE
CE
E:\JavaPro>

```

Without Synchronization class

```
Test{
```

```
    } class MyThread extends
```

```
    Thread{
```

```
        t.show(s);
```

```
    }
```

```
}
```

```
    class Test_Thread3 {        public
```

```
static void main(String[] a){
```

```
    Test t=new Test();
```

```
    MyThread m1=new MyThread("welcome to",t);
```

```
    MyThread m2=new MyThread("welcome to",t);
```

```
    MyThread m3=new MyThread("welcome to",t);
```

```
    m1.start();          m2.start();
```

```
    m3.start();
```

```
    }
```

```
}
```

Using Synchronized Methods

Synchronization is easy in Java, because all objects have their own implicit monitor associated with them. To enter an object's monitor, just call a method that has been modified with the synchronized keyword. While a thread is inside a synchronized method, all other threads that try to call it (or any other synchronized method) on the same instance have to wait. To exit the monitor and relinquish control of the object to the next waiting thread, the owner of the monitor simply returns from the synchronized method.

```
class Test{ synchronized void
```

```
show(String s){
```

```
    System.out.print(s+" JIS");    try{
```

```
        Thread.sleep(2000);
```

```
    }catch(Exception e){}
```

```
    System.out.println("CE");
```

```
    } }
```

```
class MyThread extends Thread{
```

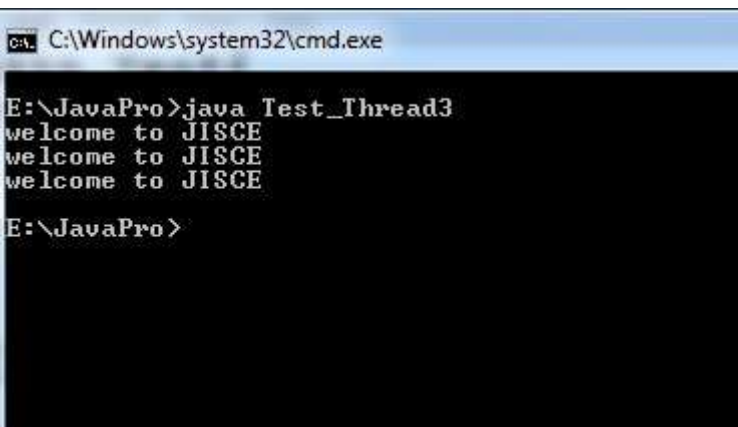
```
    String s;
```

```
    Test t;
```

```

MyThread (String s, Thread
t){  this.s=s;  this.t=t;
}
public void run(){
synchronized(t){
t.show(s);
}
} }
class Test_Thread3 {
public static void main(String[] a){
Test t=new Test();
MyThread m1=new MyThread("welcome to",t);
MyThread m2=new MyThread("welcome to",t);
MyThread m3=new MyThread("welcome to",t);
m1.start();  m2.start();  m3.start();
}
}

```



```

C:\Windows\system32\cmd.exe
E:\JavaPro>java Test_Thread3
welcome to JISCE
welcome to JISCE
welcome to JISCE
E:\JavaPro>

```

5.17 final void join() throws InterruptedException

This method waits until the thread on which it is called terminates. Its name comes from the concept of the calling thread waiting until the specified thread joins it. Additional forms of join() allow you to specify a maximum amount of time that you want to wait for the specified thread to terminate.

```

class NewThread implements Runnable {
String name; // name of thread
Thread t;
NewThread(String threadname) {
name = threadname; t = new
Thread(this, name);
System.out.println("New thread: " + t); t.start();
// Start the thread
}
}

```

```
}
// This is the entry point for thread.
public void run() { try {
for(int i = 5; i > 0; i--) {
System.out.println(name + ": " + i);
Thread.sleep(1000);
}
} catch (InterruptedException e) {
System.out.println(name + " interrupted.");
}
System.out.println(name + " exiting.");
}
}
class DemoJoin {
public static void main(String args[]) {
NewThread ob1 = new NewThread("One");
NewThread ob2 = new NewThread("Two");
NewThread ob3 = new NewThread("Three");
System.out.println("Thread One is alive: "
+ ob1.t.isAlive());
System.out.println("Thread Two is alive: "
+ ob2.t.isAlive());
System.out.println("Thread Three is alive: "
+ ob3.t.isAlive()); // wait
for threads to finish
try {
System.out.println("Waiting for threads to finish.");
ob1.t.join(); ob2.t.join();
ob3.t.join();
} catch (InterruptedException e) {
System.out.println("Main thread Interrupted");
}
System.out.println("Thread One is alive: "
+ ob1.t.isAlive());
System.out.println("Thread Two is alive: "
+ ob2.t.isAlive());
System.out.println("Thread Three is alive: "
+ ob3.t.isAlive());
System.out.println("Main thread exiting.");
}
}
```

5.8.18 Inter thread Communication

Inter-thread communication or **Co-operation** is all about allowing synchronized threads to communicate with each other.

Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.

It is implemented by following methods of **Object class**:

- wait()
- notify()
- notifyAll()

wait() method

Causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed. The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

notify() method

Wakes up a single thread that is waiting on this object's monitor.

If any threads are waiting on this object, one of them is chosen to be awakened.

notifyAll() method

Wakes up all threads that are waiting on this object's monitor.

5.19 Difference between wait and sleep

wait()	sleep()
wait() method releases the lock	sleep() method doesn't release the lock.
is the method of Object class	is the method of Thread class
is the non-static method	is the static method
should be notified by notify() or notifyAll() methods	after the specified amount of time, sleep is completed.

Example

```

class Def{
    boolean
    b=true; synchronized
    void q(){ while(!b){
    try{ wait();
        }catch(Exception e){}
        }
        System.out.print("What is your ");
    try{
        Thread.sleep(2000);
        }catch(Exception e){}
    System.out.println("name? ");
        b=false;
    notify();
    } synchronized void
    a(){ while(b){
    try{ wait();
        }catch(Exception e){}
        }
        System.out.print("Annwasha ");
    try{
        Thread.sleep(2000);
        }catch(Exception e){}
    System.out.println("Banerjee");
        b=true;
    notify();
    }
}

```

```
class Th1 extends Thread{
    Def d;
    Th1(Def d){
    this.d=d; }
    public void run(){
    for(int i=0;i<3;i++){
    d.q();
        }
    }
}
class Th2 extends Thread{
    Def d;
    Th2(Def d){
    this.d=d;
    }
    public void run(){
        for(int i=0;i<3;i++){
            d.a();
        }
    }
}
class Test_Comm{
    public static void main(String[] aa){
        Def d=new Def();
        Th1 t1=new Th1(d);
        Th2 t2=new Th2(d);
        t1.start();
        t2.start();
    }
}
```

Multiple Choice Questions

Choose the correct alternatives:

1. When dose the Exceptions in Java arises in code sequence?
 - a) Run Time
 - b) Compilation Time
 - c) Can occur any time
 - d) None of the mentioned above.
2. Which of these is a super class of all exceptional type classes? a)String

- b)RuntimeExceptions
 - c)Throwable
 - d) Cachable
3. Which of these class is related to all the exceptions that can be caught by using catch?
 - a)Error
 - b)Exception
 - c)RuntimeException
 - d) All of the mentione
 4. Which of these class is related to all the exceptions that cannot be caught?
 - a)Error
 - b)Exception
 - c)RuntimeException
 - d) All of the mentioned
 5. Which of these handles the exception when no catch is used?
 - a)DefaultHandler
 - b)finally
 - c)throw
 - d) Java run time system
 6. Which of these keywords is used to manually throw an exception?
 - a)try
 - b)finally
 - c)throw
 - d) catch
 7. Which of these keywords must be used to handle the exception thrown by try block in some rational manner?
 - a)try
 - b)finally
 - c)throw
 - d) catch
 8. Which of these operator is used to generate an instance of an exception than can be thrown by using throw?
 - a)new
 - b)malloc
 - c)alloc
 - d) thrown
 9. Which of these keywords is used to by the calling function to guard against the exception that is thrown by called function?
 - a)try
 - b)throw
 - c)throws
 - d) catch
 10. Which of these class is used to make a thread?
 - a)String
 - b)System
 - c)Thread
 - d) Runnable
 11. Which of these interface is implemented by Thread class?
 - a)Runnable
 - b)Connections
 - c)Set
 - d) MapConnections

12. Which of these method of Thread class is used to find out the priority given to a thread?
- get()
 - ThreadPriority()
 - getPriority()
 - getThreadPriority()
13. What is multithreaded programming?
- It's a process in which two different processes run simultaneously.
 - It's a process in which two or more parts of same process run simultaneously.
 - Its a process in which many different process are able to access same information.
 - Its a process in which a single process can access information from many sources.
14. Which of these are types of multitasking?
- Process based
 - Thread based
 - Process and Thread based
 - None of the mentioned
15. Thread priority in Java is? a)Integer
b)Float
c)double
d) long

Module 6

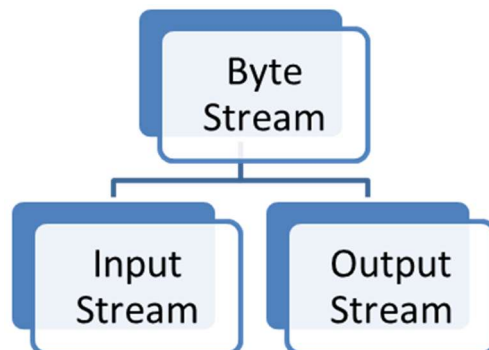
Java File Handling

6.1 Java I/O (Input and Output) is used *to process the input and produce the output*. **Java I/O** (Input and Output) is used *to process the input and produce the output*.

Java defines two types of streams

- **Byte Stream:** It provides a convenient means for handling input and output of byte.
- **Character Stream:** It provides a convenient means for handling input and output of characters. Character stream uses Unicode and therefore can be internationalized.

6.2 Byte Stream



Programs use *byte streams* to perform input and output of 8-bit bytes. All byte stream classes are descended from `InputStream` and `OutputStream`. There are many byte stream classes.

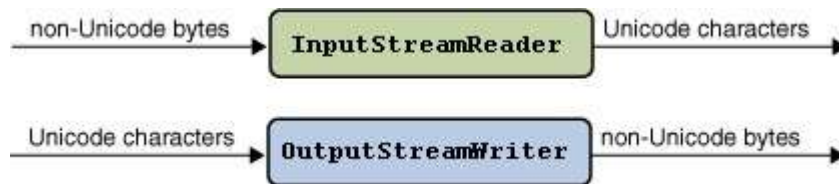


Figure 9.2

6.3 Byte Stream Classes

Stream class	Description
<code>BufferedInputStream</code>	Used for Buffered Input Stream.
<code>BufferedOutputStream</code>	Used for Buffered Output Stream.
<code>DataInputStream</code>	Contains method for reading java standard datatype
<code>DataOutputStream</code>	An output stream that contain method for writing java standard data type
<code>FileInputStream</code>	Input stream that reads from a file
<code>FileOutputStream</code>	Output stream that write to a file.
<code>InputStream</code>	Abstract class that describe stream input.
<code>OutputStream</code>	Abstract class that describe stream output.

6.4 Using Byte Streams

We'll explore `FileInputStream` and `FileOutputStream` by examining an example program named `CopyBytes`, which uses byte streams to copy `xanadu.txt`, one byte at a time.

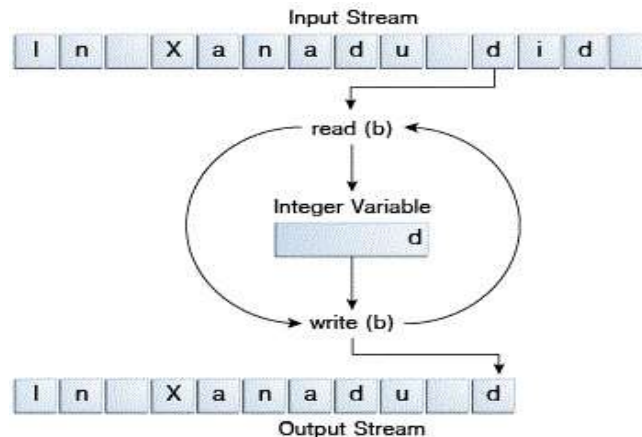
```
import java.io.FileInputStream; import
java.io.FileOutputStream;
import java.io.IOException;

public class CopyBytes {
    public static void main(String[] args) throws IOException {

        FileInputStream in = null;
        FileOutputStream out = null;
        try
        {
            in = new FileInputStream("xanadu.txt");
            out = new FileOutputStream("outagain.txt");
            int c;

            while ((c = in.read()) != -1) {
                out.write(c);
            }
        } finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    }
}
```

`CopyBytes` spends most of its time in a simple loop that reads the input stream and writes the output stream, one byte at a time, as shown in the following figure.



Always Close Streams

Closing a stream when it's no longer needed is very important — so important that CopyBytes uses a finally block to guarantee that both streams will be closed even if an error occurs. This practice helps avoid serious resource leaks.

One possible error is that CopyBytes was unable to open one or both files. When that happens, the stream variable corresponding to the file never changes from its initial null value. That's why CopyBytes makes sure that each stream variable contains an object reference before invoking close.

When Not to Use Byte Streams

CopyBytes seems like a normal program, but it actually represents a kind of low-level I/O that you should avoid. There are also streams for more complicated data types. Byte streams should only be used for the most primitive I/O.

6.5 Character Streams

The Java platform stores character values using Unicode conventions. Character stream I/O automatically translates this internal format to and from the local character set. In Western locales, the local character set is usually an 8-bit superset of ASCII.

For most applications, I/O with character streams is no more complicated than I/O with byte streams. Input and output done with stream classes automatically translates to and from the local character set. A program that uses character streams in place of byte streams automatically adapts to the local character set and is ready for internationalization — all without extra effort by the programmer.

If internationalization isn't a priority, you can simply use the character stream classes without paying much attention to character set issues. Later, if internationalization becomes a priority, your program can be adapted without extensive recoding.

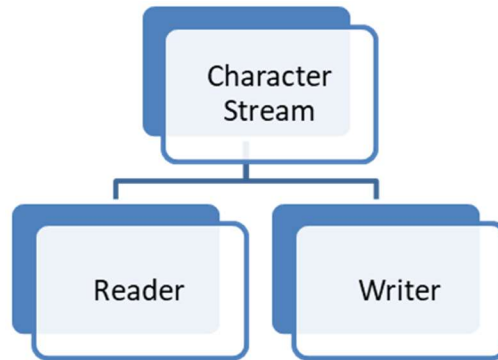


Figure 9.4

6.6 Some Character Stream Classes

Stream class	Description
BufferedReader	Handles buffered input stream.
BufferedWriter	Handles buffered output stream.
FileReader	Input stream that reads from file.
FileWriter	Output stream that writes to file.
InputStreamReader	Input stream that translate byte to character
OutputStreamReader	Output stream that translate character to byte.
PrintWriter	Output Stream that contain print() and println() method.
Reader	Abstract class that define character stream input
Writer	Abstract class that define character stream output

6.7 Reading Console input using BufferedReader


BufferedReader is character stream class.

Need to convert the byte stream to character stream as from console bytes are read.

It reads String data

6.8 Creating Object of BufferedReader

```
BufferedReader br=new BufferedReader(new InputStramReader(System.in));
```



Convert byte
stream to
character stream

Example

```
import java.io.*; class
IO1 {
    public static void main(String[] aa) throws Exception {
        BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
        System.out.println("Enter name");
        String n=br.readLine();
        System.out.println("name="+n);
    }
}
```

Example : Getting non String InputExample

```
import java.io.*; class
IO1 {
    public static void main(String[] aa) throws Exception {
        BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
        System.out.println("Enter name");
        String n=br.readLine();
        System.out.println("Enter roll");    int
r=Integer.parseInt(br.readLine());
        System.out.println("name="+n);
        System.out.println("Roll="+r);
    }
}
```

```
}
```

6.9 Formatted IO

Stream objects that implement formatting are instances of either `PrintWriter`, a character stream class, or `PrintStream`, a byte stream class.

Like all byte and character stream objects, instances of `PrintStream` and `PrintWriter` implement a standard set of write methods for simple byte and character output. In addition, both `PrintStream` and `PrintWriter` implement the same set of methods for converting internal data into formatted output. Two levels of formatting are provided:

- `print` and `println` format individual values in a standard way.
- `format` formats almost any number of values based on a format string, with many options for precise formatting.

6.10 The `print` and `println` Methods

Invoking `print` or `println` outputs a single value after converting the value using the appropriate `toString` method. We can see this in the `Root` example:

```
public class Root {
    public static void main(String[] args) {
        int i = 2;
        double r = Math.sqrt(i);

        System.out.print("The square root of ");
        System.out.print(i);
        System.out.print(" is ");
        System.out.print(r);
        System.out.println(".");

        i = 5;
        r = Math.sqrt(i);
        System.out.println("The square root of " + i + " is " + r + ".");
    }
}
```

Here is the output of `Root`:

```
The square root of 2 is 1.4142135623730951.
The square root of 5 is 2.23606797749979.
```


6.11 The format Method

The format method formats multiple arguments based on a format string. The format string consists of static text embedded with format specifiers; except for the format specifiers, the format string is output unchanged.

Format strings support many features. In this tutorial, we'll just cover some basics. For a complete description, see format string syntax in the API specification. The Root2 example formats two values with a single format invocation:

```
public class Root2 {
    public static void main(String[] args) {
        int i = 2;
        double r = Math.sqrt(i);

        System.out.format("The square root of %d is %f.%n", i, r);
    }
}
```

Here is the output:

The square root of 2 is 1.414214.

Like the three used in this example, all format specifiers begin with a % and end with a 1- or 2-character conversion that specifies the kind of formatted output being generated. The three conversions used here are:

- d formats an integer value as a decimal value.
- f formats a floating point value as a decimal value.
- n outputs a platform-specific line terminator.

Here are some other conversions:

- x formats an integer as a hexadecimal value.
- s formats any value as a string.
- tB formats an integer as a locale-specific month name.

Except for %% and %n, all format specifiers must match an argument. If they don't, an exception is thrown.

In the Java programming language, the \n escape always generates the linefeed character (\u000A). Don't use \n unless you specifically want a linefeed character. To get the correct line separator for the local platform, use %n.

```
public class Format {
    public static void main(String[] args) {
        System.out.format("%f, %1$+020.10f %n", Math.PI);
    }
}
```

Here's the output:

```
3.141593, +00000003.1415926536
```

The additional elements are all optional. The following figure shows how the longer specifier breaks down into elements.



The elements must appear in the order shown. Working from the right, the optional elements are:

- Precision. For floating point values, this is the mathematical precision of the formatted value. For s and other general conversions, this is the maximum width of the formatted value; the value is right-truncated if necessary.

- Width. The minimum width of the formatted value; the value is padded if necessary. By default the value is left-padded with blanks.
- Flags specify additional formatting options. In the Format example, the + flag specifies that the number should always be formatted with a sign, and the 0 flag specifies that 0 is the padding character. Other flags include - (pad on the right) and , (format number with locale-specific thousands separators). Note that some flags cannot be used with certain other flags or with certain conversions.
- The Argument Index allows you to explicitly match a designated argument. You can also specify < to match the same argument as the previous specifier. Thus the example could have said: `System.out.format("%f, %<+020.10f %n", Math.PI);`

6.12Scanner Class

- The Java Scanner class breaks the input into tokens using a delimiter that is whitespace by default.
- It provides many methods to read and parse various primitive values.

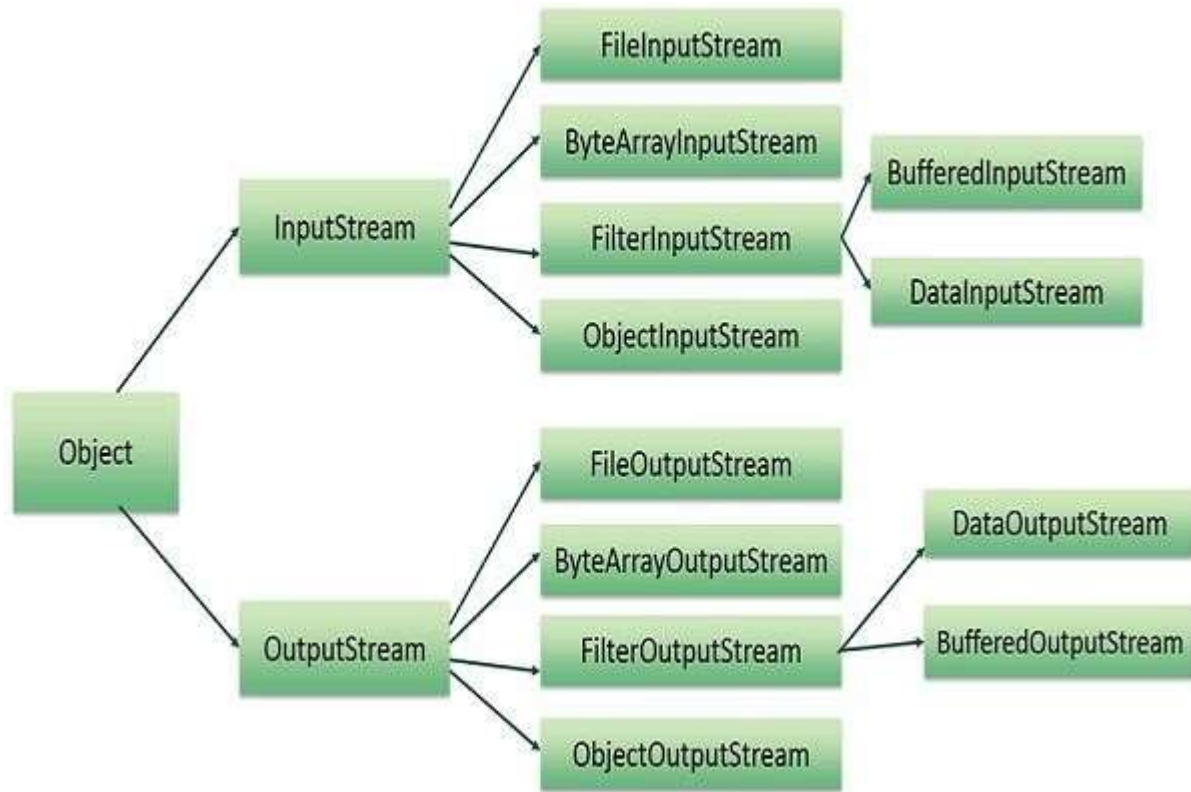
6.12 Commonly used methods of Scanner class

Method	Description
<code>public String next()</code>	it returns the next token from the scanner.
<code>public String nextLine()</code>	it moves the scanner position to the next line and returns the value as a string.
<code>public byte nextByte()</code>	it scans the next token as a byte.
<code>public short nextShort()</code>	it scans the next token as a short value.
<code>public int nextInt()</code>	it scans the next token as an int value.
<code>public long nextLong()</code>	it scans the next token as a long value.
<code>public float nextFloat()</code>	it scans the next token as a float value.
<code>public double nextDouble()</code>	it scans the next token as a double value.

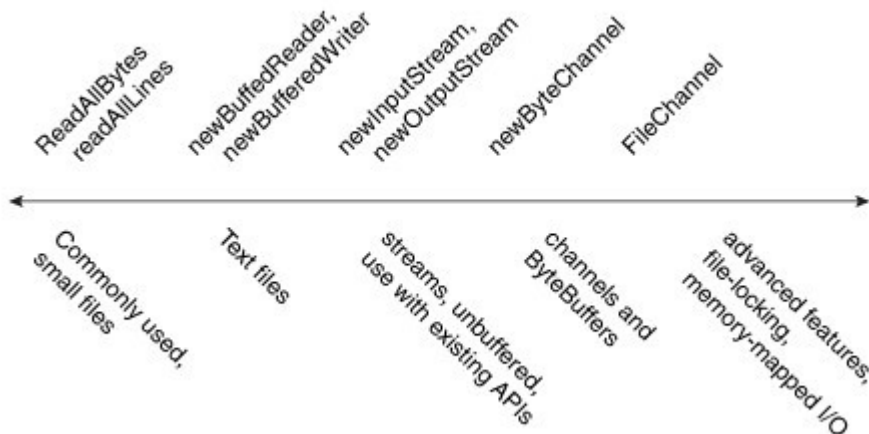
Example

```
import java.util.Scanner; class
ScannerTest{ public static void
main(String args[]){ Scanner sc=new
Scanner(System.in);
System.out.println("Enter your rollno");
int rollno=sc.nextInt();
System.out.println("Enter your name");
String name=sc.next();
System.out.println("Enter your fee");
double fee=sc.nextDouble();
System.out.println("Rollno:"+rollno+" name:"+name+" fee:"+fee);
sc.close();
}
}
```

6.13 File IO



6.14 Reading, Writing, and Creating Files



On the far left of the diagram are the utility methods readAllBytes, readAllLines, and the write methods, designed for simple, common cases. To the right of those are the methods used to iterate

over a stream or lines of text, such as `newBufferedReader`, `newBufferedWriter`, then `newInputStream` and `newOutputStream`. These methods are interoperable with the `java.io` package. To the right of those are the methods for dealing with `ByteChannels`, `SeekableByteChannels`, and `ByteBuffers`, such as the `newByteChannel` method. Finally, on the far right are the methods that use `FileChannel` for advanced applications needing file locking or memory-mapped I/O.

6.15 `FileInputStream`

This stream is used for reading data from the files.

- `InputStream f = new FileInputStream("C:/java/hello");`

Method

Sr.No.	Method & Description
1	public void close() throws IOException{ This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException.
2	protected void finalize()throws IOException { This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException.
3	public int read(int r)throws IOException{ This method reads the specified byte of data from the InputStream. Returns an int. Returns the next byte of data and -1 will be returned if it's the end of the file.
4	public int read(byte[] r) throws IOException{ This method reads r.length bytes from the input stream into an array. Returns the total number of bytes read. If it is the end of the file, -1 will be returned.
5	public int available() throws IOException{ Gives the number of bytes that can be read from this file input stream. Returns an int.

Table 9.4

6.16 File OutputStream

- ⦿ FileOutputStream is used to create a file and write data into it.
- ⦿ The stream would create a file, if it doesn't already exist, before opening it for output.
 - File f = new File("C:/java/hello"); OutputStream f = new FileOutputStream(f);

Methods

Sr.No.	Method & Description
--------	----------------------

1	<code>public void close() throws IOException {}</code> This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException.
2	<code>protected void finalize()throws IOException {}</code> This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException.
3	<code>public void write(int w)throws IOException {}</code> This methods writes the specified byte to the output stream.
4	<code>public void write(byte[] w)</code> Writes w.length bytes from the mentioned byte array to the OutputStream.

Example

```
byte []b={1,2,3};
    File f=new File("Test.txt");
    OutputStream o=new FileOutputStream(f);
for(int i=0;i<b.length;i++)        o.write(b);
    InputStream is = new FileInputStream("Test.txt");
int size = is.available();        for(int i = 0; i < size; i++) {
    System.out.print((char)is.read() + " ");
    }
    }
    catch(Exception e){}
```

6.17 Commonly Used Methods for Small Files

Reading All Bytes or Lines from a File

If you have a small-ish file and you would like to read its entire contents in one pass, you can use the `readAllBytes(Path)` or `readAllLines(Path, Charset)` method. These methods take care of most of the work for you, such as opening and closing the stream, but are not intended for handling large files. The following code shows how to use the `readAllBytes` method:

```
Path file = ...;
byte[] fileArray;
fileArray = Files.readAllBytes(file);
```

Writing All Bytes or Lines to a File

You can use one of the write methods to write bytes, or lines, to a file.

- `write(Path, byte[], OpenOption...)`
- `write(Path, Iterable< extends CharSequence>, Charset, OpenOption...)`

The following code snippet shows how to use a write method.

```
Path file = ...;

byte[] buf = ...;

Files.write(file, buf);
```

6.18 Buffered I/O Methods for Text Files

The `java.nio.file` package supports channel I/O, which moves data in buffers, bypassing some of the layers that can bottleneck stream I/O.

Reading a File by Using Buffered Stream I/O

The `newBufferedReader(Path, Charset)` method opens a file for reading, returning a `BufferedReader` that can be used to read text from a file in an efficient manner.

The following code snippet shows how to use the `newBufferedReader` method to read from a file.

The file is encoded in "US-ASCII." `Charset charset = Charset.forName("US-ASCII");`

```
try (BufferedReader reader = Files.newBufferedReader(file, charset)) {
    String line = null;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
} catch (IOException x) {
    System.err.format("IOException: %s%n", x); }
```

Writing a File by Using Buffered Stream I/O

You can use the `newBufferedWriter(Path, Charset, OpenOption...)` method to write to a file using a `BufferedWriter`.

The following code snippet shows how to create a file encoded in "US-ASCII" using this method:

```
Charset charset = Charset.forName("US-ASCII");
String s = ...;
try (BufferedWriter writer = Files.newBufferedWriter(file, charset)) {
    writer.write(s, 0, s.length());
} catch (IOException x) {
    System.err.format("IOException: %s%n", x); }
```

6.19 Methods for Unbuffered Streams and Interoperable with `java.io` APIs

Reading a File by Using Stream I/O

To open a file for reading, you can use the `newInputStream(Path, OpenOption...)` method. This method returns an unbuffered input stream for reading bytes from the file.

```
Path file = ...;
try (InputStream in = Files.newInputStream(file);
    BufferedReader reader =
        new BufferedReader(new InputStreamReader(in))) {
    String line = null;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
} catch (IOException x) {
    System.err.println(x);
}
```

Creating and Writing a File by Using Stream I/O

You can create a file, append to a file, or write to a file by using the `newOutputStream(Path, OpenOption...)` method. This method opens or creates a file for writing bytes and returns an unbuffered output stream.

The method takes an optional `OpenOption` parameter. If no open options are specified, and the file does not exist, a new file is created. If the file exists, it is truncated. This option is equivalent to invoking the method with the `CREATE` and `TRUNCATE_EXISTING` options.

The following example opens a log file. If the file does not exist, it is created. If the file exists, it is opened for appending.

```
import static java.nio.file.StandardOpenOption.*;
```

```
import java.nio.file.*;
import java.io.*;

public class LogFileTest {

    public static void main(String[] args) {

        // Convert the string to a
        // byte array.
        String s = "Hello World! ";
        byte data[] = s.getBytes();    Path p
        = Paths.get("./logfile.txt");

        try (OutputStream out = new BufferedOutputStream(
        Files.newOutputStream(p, CREATE, APPEND))) {
            out.write(data, 0, data.length);
        } catch (IOException x) {
            System.err.println(x);
        }
    }
}
```

Multiple Choice Questions

1. Which of these packages contain classes and interfaces used for input & output operations of a program?
 - a) java.util
 - b) java.lang
 - c) java.io
 - d) All of the mentioned
2. Which of these class is not a member class of java.io package?
 - a) String
 - b) StringReader
 - c) Writer
 - d) File
3. Which of these interface is not a member of java.io package?
 - a) DataInput
 - b) ObjectInput
 - c) ObjectFilter
 - d) FileFilter
4. Which of these class is not related to input and output stream in terms of functioning?
 - a) File
 - b) Writer
 - c) InputStream
 - d) Reader

5. Which of these classes is used for input and output operation when working with bytes?
 - a)InputStream
 - b)Reader
 - c)Writer
 - d) All of the mentioned
6. Which of these class is used to read and write bytes in a file?
 - a)FileReader
 - b)FileWriter
 - c)FileInputStream
 - d) InputStreamReader
7. Which of these method of InputStream is used to read integer representation of next available byte input?
 - a)read()
 - b)scanf()
 - c)get()
 - d) getInteger()
8. Which of these data type is returned by every method of OutputStream?
 - a)int
 - b)float
 - c)byte
 - d) None of the mentioned
9. Which of these is a method to clear all the data present in output buffers?
 - a)clear()
 - b)flush()
 - c)fflush()
 - d) close()
10. Which of these method(s) is/are used for writing bytes to an outputstream?
 - a)put()
 - b)print() and write()
 - c)printf()
 - d) write() and read()

Module 7

Applet Programming

7.1 Applet

Applet are small application that are accessed on an Internet Server transported over the Internet automatically installed and run as part of a web document.

After an applet arrives on the client it has limited access to resources so that it can produce a graphical user interface and run computation without introducing the risk of viruses or loss of data integrity.

- Applets do not need a main() method.
- Applets must be run under an applet viewer or a Java-compatible browser.
- User I/O is not accomplished with Java's stream I/O classes. Instead, applets use the interface provided by the AWT or Swing.

```
import java.awt.*; import java.applet.*; /*
```

```
<applet code="MyApplet.class" height=200 width=200">
```

```
</applet>
```

```
*/
```

```
public class MyApplet extends Applet{
```

```
String s=""; public void init(){
```

```
s+="...call init()";
```

```
}
```

```
public void start(){ s+="....call start()";
```

```
}
```

```
public void stop(){ s+="....call stop()";
```

```
}
```

```
public void paint(Graphics g){ g.drawString(s,100,100);
```

```
}
```

All applets are subclasses (either directly or indirectly) of **Applet**. Applets are not stand-alone programs. Instead, they run within either a web browser or an applet viewer. The illustrations shown in this chapter were created with the standard applet viewer, called `appletviewer`, provided by the JDK.

Compile :

```
java MyApplet.java
```

Run

```
appletviewer MyApplet.java
```

Execution of an applet does not begin at `main()`. Actually, few applets even have `main()`

methods. Instead, execution of an applet is started and controlled with an entirely different mechanism, which will be explained shortly. Output to your applet's window is not performed by `System.out.println()`. Rather, in non-Swing applets, output is handled with various AWT methods, such as `drawString()`, which outputs a string to a specified X,Y location. Input is also handled differently than in a console application.

To use an applet, it is specified in an HTMLfile. One way to do this is by using the APPLET

tag. The applet will be executed by a Java-enabled web browser when it encounters the APPLET tag within the HTMLfile. To view and test an applet more conveniently, simply include a comment at the head of your Java source code file that contains the APPLET tag. This way, your code is documented with the necessary HTML

statements needed by your applet, and you can test the compiled applet by starting the applet viewer with your Java source code file specified as the target.

```
/*  
  
<applet code="MyApplet" width=200 height=60>  
  
</applet>  
  
*/
```

This comment contains an APPLET tag that will run an applet called MyApplet in a window that is 200 pixels wide and 60 pixels high. Because the inclusion of an APPLET command makes testing applets easier, all of the applets shown in this book will contain the appropriate APPLET tag embedded in a comment.

7.2 The HTML APPLET Tag

As mentioned earlier, Sun currently recommends that the APPLET tag be used to start an applet from both an HTML document and from an applet viewer. An applet viewer will execute each APPLET tag that it finds in a separate window, while web browsers will allow many applets on a single page. So far, we have been using only a simplified form of the APPLET tag. Now it is time to take a closer look at it.

The syntax for a fuller form of the APPLET tag is shown here. Bracketed items are optional.

```
< APPLET
```

```

[CODEBASE = codebaseURL]
CODE = appletFile
[ ALT = alternateText]
[NAME = appletInstanceName]
WIDTH = pixels HEIGHT =
pixels
[ALIGN = alignment]
[VSPACE = pixels] [HSPACE = pixels]
>
[< PARAM NAME = AttributeName VALUE = AttributeValue>]
[< PARAM NAME = AttributeName2 VALUE =
AttributeValue>]
...
[HTML Displayed in the absence of Java
  ] </APPLET>

```

Let's take a look at each part now.

CODEBASE CODEBASE is an optional attribute that specifies the base URL of the applet code, which is the directory that will be searched for the applet's executable class file (specified by the CODE tag). The HTML document's URL directory is used as the CODEBASE if this attribute is not specified. The CODEBASE does not have to be on the host from which the HTML document was read.

CODE CODE is a required attribute that gives the name of the file containing your applet's compiled **.class** file. This file is relative to the code base URL of the applet, which is the directory that the HTML file was in or the directory indicated by CODEBASE if set.

ALT The ALT tag is an optional attribute used to specify a short text message that should be displayed if the browser recognizes the APPLETTAG tag but can't currently run Java applets.

This is distinct from the alternate HTML you provide for browsers that don't support applets.

NAME NAME is an optional attribute used to specify a name for the applet instance.

Applets must be named in order for other applets on the same page to find them by name and communicate with them. To obtain an applet by name, use **getApplet()**, which is defined by the **AppletContext** interface.

WIDTH and HEIGHT WIDTH and HEIGHT are required attributes that give the size (in pixels) of the applet display area.

ALIGN ALIGN is an optional attribute that specifies the alignment of the applet. This attribute is treated the same as the HTML IMG tag with these possible values: LEFT, RIGHT, TOP, BOTTOM, MIDDLE, BASELINE, TEXTTOP, ABSMIDDLE, and ABSBOTTOM.

VSPACE and HSPACE These attributes are optional. VSPACE specifies the space, in pixels, above and below the applet. HSPACE specifies the space, in pixels, on each side of the applet.

They're treated the same as the IMG tag's VSPACE and HSPACE attributes.

PARAM NAME and VALUE The PARAM tag allows you to specify applet-specific arguments

in an HTML page. Applets access their attributes with the **getParameter()** method.

7.3 Applet Life Cycle Method

init() is used to initialize the Applet. It is invoked only once. **start()** is invoked after the **init()** method or browser is maximized. It is used to start the Applet.

stop() is used to stop the Applet. It is invoked when Applet is stop or browser is minimized.

destroy()

The **destroy()** method is called when the environment determines that your applet needs to be removed completely from memory. At this point, you should free up any resources the applet may be using. The **stop()** method is always called before **destroy()**.

7.4 Other Method:

Overriding update()

In some situations, your applet may need to override another method defined by the AWT, called **update()**. This method is called when your applet has requested that a portion of its window be redrawn. The default version of **update()** simply calls **paint()**. However, you can override the **update()** method so that it performs more subtle repainting. In general, overriding **update()** is a specialized technique that is not applicable to all applets, and the examples in this book do not override **update()**.

Requesting Repainting

The **repaint()** method is defined by the AWT. It causes the AWT run-time system to execute a call to your applet's **update()** method, which, in its default implementation, calls **paint()**. Thus, for another part of your applet to output to its window, simply store the output and then call **repaint()**. The AWT will then execute a call to **paint()**, which can display the stored information. For example, if part of your applet needs to output a string, it can store this string in a String variable and then call **repaint()**. Inside **paint()**, you will output the string using **drawString()**.

The **repaint()** method has four forms. Let's look at each one, in turn. The simplest version of **repaint()** is shown here:

```
void repaint()
```

This version causes the entire window to be repainted. The following version specifies a region that will be repainted:

```
void repaint(int left, int top, int width, int height)
```

Here, the coordinates of the upper-left corner of the region are specified by **left** and **top**, and the width and height of the region are passed in **width** and **height**. These dimensions are specified in pixels. You save time by specifying a region to repaint. Window updates are costly in terms of time. If you need to update only a small portion of the window, it is more efficient to repaint only that region.

Calling `repaint()` is essentially a request that your applet be repainted sometime soon.

However, if your system is slow or busy, `update()` might not be called immediately. Multiple requests for repainting that occur within a short time can be collapsed by the AWT in a manner such that `update()` is only called sporadically

```
import java.awt.*; import java.applet.*; /*
<applet code="SimpleBanner" width=300 height=50>
</applet>
*/
public class SimpleBanner extends Applet implements Runnable { String msg = " A
Simple Moving Banner."; Thread t = null; int state; boolean stopFlag;
*** Set colors and initialize thread. public void init() { setBackground(Color.cyan);
setForeground(Color.red);
}
*** Start thread public void
start() { t = new Thread(this);
stopFlag = false; t.start();

}
// Entry point for the thread that runs the banner. public void run() { char
ch;
// Display banner
for(;;) { try { repaint();
Thread.sleep(250); ch = msg.charAt(0); msg =
msg.substring(1, msg.length()); msg += ch;
if(stopFlag) break;
} catch(InterruptedException e) {}
}
}
}
}
// Pause the banner. public void stop() { stopFlag =
true;
t = null;
}
// Display the banner.
public void paint(Graphics g) { g.drawString(msg, 50, 30);
}
}
```

7.5 Advantages and Disadvantages of Applet:

Advantages

- It is simple to make it work on Linux, Microsoft Windows and OS X i.e. to make it cross platform. Applets are supported by most web browsers.
- The same applet can work on “all” installed versions of Java at the same time, rather than just the latest plug-in version only. However, if an applet requires a later version of the Java Runtime Environment (JRE) the client will be forced to wait during the large download.
- Most web browsers cache applets so will be quick to load when returning to a web page. Applets also improve with use: after a first applet is run, the JVM is already running and starts quickly (the JVM will need to restart each time the browser starts afresh). It should be noted that JRE versions 1.5 and greater stop the JVM and restart it when the browser navigates from one HTML page containing an applet to another containing an applet.
- It can move the work from the server to the client, making a web solution more scalable with the number of users/clients.
- If a standalone program (like Google Earth) talks to a web server, that server normally needs to support all prior versions for users which have not kept their client software updated. In contrast, a properly configured browser loads (and caches) the latest applet version, so there is no need to support legacy versions.
- The applet naturally supports the changing user state, such as figure positions on the chessboard.
- Developers can develop and debug an applet direct simply by creating a main routine (either in the applet’s class or in a separate class) and calling `init()` and `start()` on the applet, thus allowing for development in their favorite Java SE development environment. All one has to do after that is re-test the applet in the AppletViewer program or a web browser to ensure it conforms to security restrictions.
- An untrusted applet has no access to the local machine and can only access the server it came from. This makes such an applet much safer to run than a standalone executable that it could replace. However, a signed applet can have full access to the machine it is running on if the user agrees.

- Java applets are fast – and can even have similar performance to native installed software.

Disadvantages:

- It requires the Java plug-in.
- Some browsers, notably mobile browsers running Apple iOS or Android do not run Java applets at all.
- Some organizations only allow software installed by the administrators. As a result, some users can only view applets that are important enough to justify contacting the administrator to request installation of the Java plug-in.
- As with any client-side scripting, security restrictions may make it difficult or even impossible for an untrusted applet to achieve the desired goals. However, simply editing the java.policy file in the JAVA JRE installation, one can grant access to the local filesystem or system clipboard for example, or to other network sources other than the network source that served the applet to the browser.
- Some applets require a specific JRE. This is
- discouraged.

If an applet requires a newer JRE than available on the system, or a specific JRE, the user running it the first time will need to wait for the large JRE download to complete. □ Java automatic installation or update may fail if a proxy server is used to access the web. This makes applets with specific requirements impossible to run unless Java is manually updated. The Java automatic updater that is part of a Java installation also may be complex to configure if it must work through a proxy.

- Unlike the older applet tag, the object tag needs workarounds to write a cross-browser HTML document.
- There is no standard to make the content of applets available to screen readers. Therefore, applets can harm the accessibility of a web site to users with special needs.

7.6 Some Methods of Graphics:

[draw3DRect](#)(int, int, int, int, boolean)

Draws a highlighted 3-D rectangle.

• [drawArc](#)(int, int, int, int, int, int)

Draws an arc bounded by the specified rectangle from startAngle to endAngle.

- [drawImage](#)(Image, int, int, ImageObserver)

Draws the specified image at the specified coordinate (x, y). • [drawImage](#)(Image, int, int, int, int, ImageObserver)

Draws the specified image inside the specified rectangle.

[drawLine](#)(int, int, int, int)

Draws a line between the coordinates (x1,y1) and (x2,y2). • [drawOval](#)(int, int, int, int)

Draws an oval inside the specified rectangle using the current color. [drawPolygon](#)(int[], int[], int)

Draws a polygon defined by an array of x points and y points. - [drawPolygon](#)(Polygon)

Draws a polygon defined by the specified point. [drawRect](#)(int, int, int, int)

Draws the outline of the specified rectangle using the current color. [drawRoundRect](#)(int, int, int, int, int)

Draws an outlined rounded corner rectangle using the current color. [drawString](#)(String, int, int)

Draws the specified String using the current font and color. • [fill3DRect](#)(int, int, int, int, boolean)

Paints a highlighted 3-D rectangle using the current color. • [fillArc](#)(int, int, int, int, int, int)

Fills an arc using the current color. • [fillOval](#)(int, int, int, int)

Fills an oval inside the specified rectangle using the current color. • [fillPolygon](#)(int[], int[], int)

Fills a polygon with the current color. - [fillPolygon](#)(Polygon)

Fills the specified polygon with the current color. • [fillRect](#)(int, int, int, int)

Fills the specified rectangle with the current color. • [fillRoundRect](#)(int, int, int, int, int, int)

Draws a rounded rectangle filled in with the current color.

• [getColor](#)()

Gets the current color. •

[getFont](#)()

Gets the current font.

• [setColor](#)(Color)

Sets the current color to the specified color.

• [setFont](#)(Font)

Sets the font for all subsequent text-drawing operations.

7.7 Passing Parameters to Applets

As just discussed, the APPLET tag in HTML allows you to pass parameters to your applet. To retrieve a parameter, use the **getParameter()** method. It returns the value of the specified import java.awt.*; import java.applet.*;

```
/*  
  
  <APPLET CODE="ParaExample" WIDTH=200  
  HEIGHT=100> <param name="param1"  
value="Hello">  
  
  <param    name="param2" value="14">  
    <param name="param3"    value="2">  
  </APPLET  
  
*/  
public class ParaExample extends Applet  
{
```

```

String parameter1;
int parameter2; int
parameter3; int
result; public
void init()
{
parameter1 = getParameter("param1");

parameter2 =
Integer.parseInt(getParameter("param2")); parameter3
= Integer.parseInt(getParameter("param3"));

    result = parameter2 + parameter3;
}
public void paint(Graphics g)
{
    g.drawString("Parameter 1 is: " + parameter1,20,20);
    g.drawString("Parameter 2 is: " + parameter2,20,40);
    g.drawString("Parameter 3 is: " + parameter3,20,60);
    g.drawString("Parameter 2 + parameter 3 is: " +
    result,20,80);
}
}

```

7.8 `getDocumentBase()` and `getCodeBase()`

Often, you will create applets that will need to explicitly load media and text. Java will allow the applet to load data from the directory holding the HTML file that started the applet (the *document base*) and the directory from which the applet's class file was loaded (the *code base*). These directories are returned as **URL** objects (described in Chapter 20) by **`getDocumentBase()`** and **`getCodeBase()`**. They can be concatenated with a string that names the file you want to load. To actually load another file, you will use the **`showDocument()`** method defined by the **`AppletContext`** interface, discussed in the next section. `import java.awt.*; import java.applet.*; import java.net.*;`

```

/*
<applet code="MyCode" height=100
width=100> </applet> */
public class MyCode extends Applet{

    public void paint(Graphics
g){    String    s="";    URL
        url=getCodeBase();
        s=url.toString();

```

```

g.drawString(s,100,100);
url=getDocumentBase();
s=url.toString();g.drawString(s,100,1
50);

}
}

```

7.9 The Delegation Event Model

The modern approach to handling events is based on the *delegation event model*, which defines standard and consistent mechanisms to generate and process events. Its concept is quite simple: a *source* generates an event and sends it to one or more *listeners*. In this scheme, the listener simply waits until it receives an event. Once an event is received, the listener processes the event and then returns. The advantage of this design is that the application logic that processes events is cleanly separated from the user interface logic that generates those events. A user interface element is able to “delegate” the processing of an event to a separate piece of code.

In the delegation event model, listeners must register with a source in order to receive an event notification. This provides an important benefit: notifications are sent only to listeners that want to receive them. This is a more efficient way to handle events than the design used by the old Java 1.0 approach. Previously, an event was propagated up the containment hierarchy until it was handled by a component. This required components to receive events that they did not process, and it wasted valuable time. The delegation event model eliminates this overhead.

Events

In the delegation model, an *event* is an object that describes a state change in a source. It can be generated as a consequence of a person interacting with the elements in a graphical user interface.

Event Sources

A source is an object that generates an event. This occurs when the internal state of that object changes in some way.

Event Listeners

A listener is an object that is notified when an event occurs. It has two major requirements. First, it must have been registered with one or more sources to receive notifications about specific types of events. Second, it must implement methods to receive and process these notifications.

```

import java.awt.*;
import
java.applet.*;
import
java.awt.event.*;
//import
javax.swing.*;

```

```
/*<applet code="MyApp_Listn" height=100
width=100> </applet>
*/ public class MyApp_Listn extends Applet
implements
ActionListener{ Button b1;
    Button b2;
    Label l1;
    TextField t;
    public void
    init(){

        b1=new Button("RED");
        b2=new Button("PINK");
        l1=new Label("click on
        button");
t=new
    TextField(10);
    add(b1);
    add(b2);
    add(l1);
    add(t);
    b1.addActionListener(this);
    b2.addActionListener(this);
}
    public void actionPerformed(ActionEvent ae){

        if(ae.getSource()==b1){
            setBackground(Color.red);
        }
        if(ae.getSource()==b2){
            setBackground(Color.pink);
        }
    }
}
```

7.10 Layout Manager

Layout means the arrangement of components within the container. In other way we can say that placing the components at a particular position within the container. The task of laying out the controls is done automatically by the Layout Manager.

The layout manager automatically positions all the components within the container. If we do not use layout manager then also the components are positioned by the default layout manager.

It is possible to layout the controls by hand but it becomes very difficult because of the following two reasons.

- It is very tedious to handle a large number of controls within the container.
- Often the width and height information of a component is not given when we need to arrange them.

Java provide us with various layout manager to position the controls. The properties like size, shape and arrangement varies from one layout manager to other layout manager. When the size of the applet or the application window changes the size, shape and arrangement of the components also changes in response i.e. the layout managers adapt to the dimensions of appletviewer or the application window.

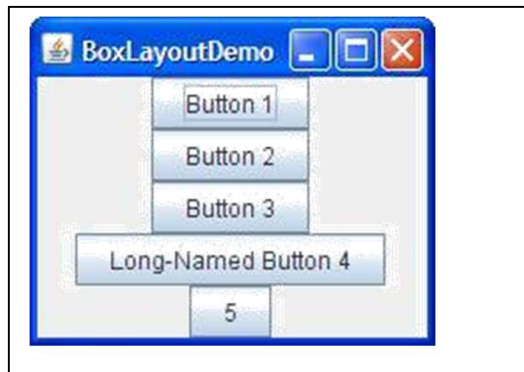
The layout manager is associated with every Container object. Each layout manager is an object of the class that implements the `LayoutManager` interface.

BorderLayout



Every content pane is initialized to use a `BorderLayout`. A `BorderLayout` places components in up to five areas: top, bottom, left, right, and center. All extra space is placed in the center area. Tool bars that are created using [JToolBar](#) must be created within a `BorderLayout` container, if you want to be able to drag and drop the bars away from their starting positions..

BoxLayout



The BoxLayout class puts components in a single row or column. It respects the components' requested maximum sizes and also lets you align components..

CardLayout



The CardLayout class lets you implement an area that contains different components at different times. A CardLayout is often controlled by a combo box, with the state of the combo box determining which panel (group of components) the CardLayout displays. An alternative to using CardLayout is using a [tabbed pane](#), which provides similar functionality but with a pre-defined GUI

FlowLayout



FlowLayout is the default layout manager for every JPanel. It simply lays out components in a single row, starting a new row if its container is not sufficiently wide. Both panels in CardLayoutDemo, shown [previously](#), use FlowLayout.

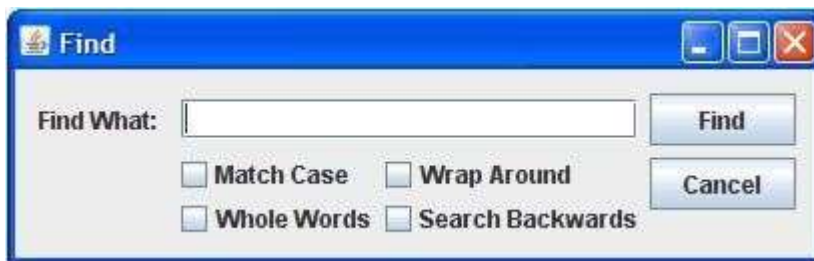
GridBagLayout

GridBagLayout is a sophisticated, flexible layout manager. It aligns components by placing them within a grid of cells, allowing components to span more than one cell. The rows in the grid can have different heights, and grid columns can have different widths.

GridLayout



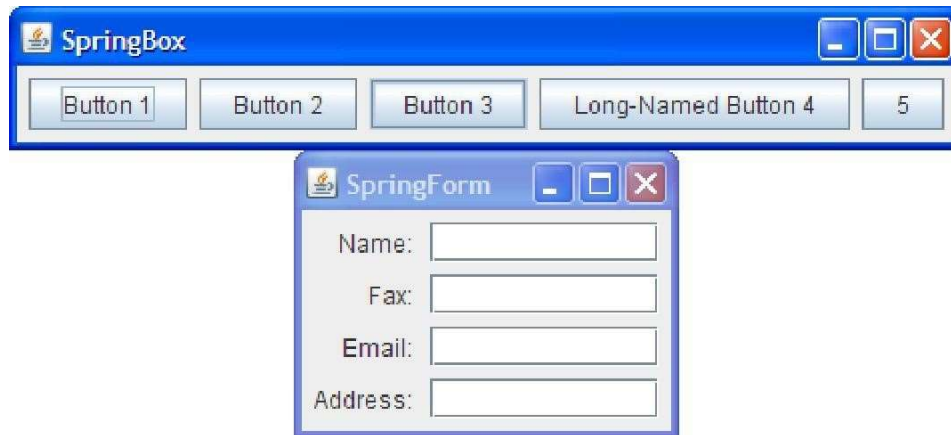
GridLayout simply makes a bunch of components equal in size and displays them in the requested number of rows and columns.



GroupLayout

GroupLayout is a layout manager that was developed for use by GUI builder tools, but it can also be used manually. GroupLayout works with the horizontal and vertical layouts separately. The layout is defined for each dimension independently. Consequently, however, each component needs to be defined twice in the layout. The Find window shown above is an example of a GroupLayout.

SpringLayout



SpringLayout is a flexible layout manager designed for use by GUI builders. It lets you specify precise relationships between the edges of components under its control. For example, you might define that the left edge of one component is a certain distance (which can be dynamically calculated) from the right edge of a second component. SpringLayout lays out the children of its associated container according to a set of constraints.

7.11 Creation of buttons (JButton class only) & text fields:

Swing-based applets are similar to AWT-based applets, but with an important difference: A Swing applet extends JApplet rather than Applet. JApplet is derived from Applet. Thus, JApplet includes all of the functionality found in Applet and adds support for Swing. JApplet is a toplevel Swing container, which means that it is not derived from JComponent. Because JApplet is a top-level container, it includes the various panes described earlier. This means that all components are added to JApplet's content pane in the same way that components are added to JFrame's content pane.

Swing applets use the same four lifecycle methods `init()`, `start()`, `stop()`, and `destroy()`. Of course, you need override only those methods that are needed by your applet. Painting is accomplished differently in Swing than it is in the AWT, and a Swing applet will not normally override the `paint()` method.

`import javax.swing.*; import java.awt.*; import java.awt.event.*; /* This HTML can be used to launch the applet:`

```
<object code="MySwingApplet" width=220 height=90> </object>
```

```
*/
```

```
public class MySwingApplet extends JApplet { JButton jbtnAlpha;
```

```
JButton jbtnBeta; JLabel jlab;

// Initialize the applet. public void init() {

try {

SwingUtilities.invokeLaterAndWait(new Runnable () { public void run() { makeGUI();
// initialize the GUI

}

});
} catch(Exception exc) {

System.out.println("Can't create because of "+ exc);
}

}
// This applet does not need to override start(), stop(),

// or destroy().
// Set up and initialize the GUI.

private void makeGUI() {

// Set the applet to use flow layout. setLayout(new FlowLayout());

// Make two buttons.

jbtnAlpha = new JButton("Alpha"); jbtnBeta = new JButton("Beta");

// Add action listener for Alpha. jbtnAlpha.addActionListener(new ActionListener() { public
void actionPerformed(ActionEvent le) { jlab.setText("Alpha was pressed.");

}
});

// Add action listener for Beta. jbtnBeta.addActionListener(new ActionListener() { public
void actionPerformed(ActionEvent le) { jlab.setText("Beta was pressed.");

}
});
});
```

```
// Add the buttons to the content pane. add(jbtnAlpha); add(jbtnBeta);

// Create a text-based label.

jlab = new JLabel("Press a button."); // Add the label to the content pane. add(jlab);

}
}
```

Lecture 4:

7.12 IO Applet

Applets work in graphical environment. Therefore applets treat input as text strings. We must first create an area of the screen in which user can type and edit input items. We can do this by using the TextField class of the applet package. The values of the fields can be given even editor after the creation of input fields. Next step is to retrieve the items from the fields for display of calculations.

Applet Taking User Input

```
public class TestApp extends Applet implements ActionListener{
    TextField tf1,tf2; Button b;
public void init() {    tf1=new
TextField(8);    tf2=new
TextField(8);
tf1.setBounds(50,50,20,50);
tf2.setBounds(50,120,20,50);
    b=new Button("ADD");
tf1.setText("0");
tf2.setText("0");
    }
    public void start(){
        add(tf1);
add(tf2);
add(b);
    }
    public void actionPerformed(ActionEvent ae){

    }
    public void paint(Graphics g){
int sum=0;
    g.drawString("enter numbers",10,50);
try{
        sum=Integer.parseInt(tf1.getText()+Integer.parseInt(tf2.getText());
    }catch(Exception e)
```

Multiple Choice Questions

1. Which of these functions is called to display the output of an applet? a)display()
b)paint()
c)displayApplet()
d)PrintApplet() View

Answer

2. Which of these methods can be used to output a sting in an applet? a)display()
b)print()
c)drawString()
d)transient()

View Answer

3. What does AWT stands for?
a)All Window Tools
b)All Writing Tools
c)Abstract Window Toolkit
d)Abstract Writing Toolkit

View Answer

4. Which of these methods is a part of Abstract Window Toolkit (AWT) ? a)display()
b)paint()
c)drawString()
d)transient()

View Answer

5. Which of these modifiers can be used for a variable so that it can be accessed from any thread or parts of a program? a)transient
b)volatile
c)global
d)No modifier is needed

View Answer

6. Which of these operators can be used to get run time information about an object?
a)getInfo
b)Info
c)instanceof
d)getinfoof

7. Which of these functions is called to display the output of an applet?

- a) display()
b) print()
c) displayApplet()
d) PrintApplet()

8. Which of these methods can be used to output a sting in an applet?

- a) display()

- b) print()
- c) drawString()
- d) transient()

9. What is the Message is displayed in the applet made by this program? `import java.awt.*; import java.applet.*; public class myapplet extends Applet { public void paint(Graphics g) { g.drawString("A Simple Applet", 20, 20); } }`

- a). A Simple Applet
- b). A Simple Applet 20 20
- c). Compilation Error
- d). Runtime Error

10. What is the length of the application box made by this program? `import java.awt.*; import java.applet.*; public class myapplet extends Applet { Graphic g; g.drawString("A Simple Applet", 20, 20); }`

- a). 20
- b). Default value
- c). Compilation Error
- d). Runtime Error

