## AVN INSTITUTE OF ENGINEERING & TECHNOLOGY::HYDERABAD

## DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

OOPs through JAVA Lecture Notes By DAYAKAR GURRAM

## <u>1-UNIT</u>

#### Need for oop paradigm

- The object oriented paradigm is a methodology for producing reusable software components
- The object-oriented paradigm is a programming methodology that promotes the efficient design and development of software systems using reusable components that can be quickly and safely assembled into larger systems.
- Object oriented programming has taken a completely different direction and will place an emphasis on object s and information. With object oriented programming, a problem will be broken down into a number of units .these are called objects .The foundation of oop is the fact that it will place an emphasis on objects and classes. There are number of advantages to be found with using the oop paradigm, and some of these are oop paradigm
- Object oriented programming is a concept that was created because of the need to overcome the problems that were found with using structured programming techniques. While structured programming uses an approach which is top down, oop uses an approach which is bottom up.
- A **paradigm** is a way in which a computer language looks at the problem to be solved. We divide computer languages into four paradigms: *procedural*, *object-oriented*, *functional* and *declarative*
- A paradigm shift from a <u>function-centric</u> approach to an <u>object-centric</u> approach to software development
- A program in a procedural paradigm is an active agent that uses passive objects that we refer to as data or data items.
- The basic unit of code is the **class** which is a template for creating run-time objects.
- Classes can be composed from other classes. For example, Clocks can be constructed as an aggregate of Counters.
- The object-oriented paradigm deals with active objects instead of passive objects. We encounter many active objects in our daily life: a vehicle, an automatic door, a dishwasher and so on. The actions to be performed on these objects are included in the object: the objects need only to receive the appropriate stimulus from outside to perform one of the actions.
- A file in an object-oriented paradigm can be packed with all the procedures—called methods in the object-oriented paradigm—to be performed by the file: printing, copying,

deleting and so on. The program in this paradigm just sends the corresponding request to the object

• Java provides automatic garbage collection, relieving the programmer of the need to ensure that unreferenced memory is regularly deallocated.

## **Object Oriented Paradigm – Key Features**

- Encapsulation
- Abstraction
- Inheritance
- Polymorphis

A Way of viewing World- Agents

- The word *agent* has found its way into a number of technologies. It has been applied to aspects of artificial intelligence research and to constructs developed for improving the experience provided by collaborative online social environments (MUDS, MOOs, and the like). It is a branch on the tree of distributed computing. There are agent development toolkits and agent programming languages.
- The Agent Identity class defines agent identity. An instance of this class uniquely identifies an agent. Agents use this information to identify the agents with whom they are interested in collaborating.
- The Agent Host class defines the agent host. An instance of this class keeps track of every agent executing in the system. It works with other hosts in order to transfer agents.
- The Agent class defines the agent. An instance of this class exists for each agent executing on a given agent host.
- OOP uses an approach of treating a real world agent as an object.
- Object-oriented programming organizes a program around its data (that is, objects) and a set of well-defined interfaces to that data.
- An object-oriented program can be characterized as data controlling access to code by switching the controlling entity to data.

#### **Responsibility**

- In object-oriented design, the chain-of-responsibility pattern is a design pattern consisting of a source of command objects and a series of processing objects..
- Each processing object contains logic that defines the types of command objects that it can handle; the rest are passed to the next processing object in the chain. A mechanism also exists for adding new processing objects to the end of this chain.

- Primary motivation is the need for a platform-independent (that is, architecture- neutral) language that could be used to create software to be embedded in various consumer electronic devices, such as microwave ovens and remote controls.
- Objects with clear responsibilities
- Each class should have a clear responsibility.
- If you can't state the purpose of a class in a single, clear sentence, then perhaps your class structure needs some thought.
- In object-oriented programming, the single responsibility principle states that every class should have a single responsibility, and that responsibility should be entirely encapsulated by the class. All its services should be narrowly aligned with that responsibility.

#### Messages

- Message implements the Part interface. Message contains a set of attributes and a "content".
- Message objects are obtained either from a Folder or by constructing a new Message object of the appropriate subclass. Messages that have been received are normally retrieved from a folder named "INBOX".
- A Message object obtained from a folder is just a lightweight reference to the actual message. The Message is 'lazily' filled up (on demand) when each item is requested from the message.
- Note that certain folder implementations may return Message objects that are pre-filled with certain user-specified items. To send a message, an appropriate subclass of Message (e.g., Mime Message) is instantiated, the attributes and content are filled in, and the message is sent using the Transport. Send method.
- We all like to use programs that let us know what's going on. Programs that keep us informed often do so by displaying status and error messages.
- These messages need to be translated so they can be understood by end users around the world.
- The Section discusses translatable text messages. Usually, you're done after you move a message String into a Resource Bundle.
- If you've embedded variable data in a message, you'll have to take some extra steps to prepare it for translation.

## **Methods**

- The only required elements of a method declaration are the method's return type, name, a pair of parentheses, (), and a body between braces, {}.
- Two of the components of a method declaration comprise the *method signature*—the method's name and the parameter types.
- More generally, method declarations have six components, in order:
- Modifiers—such as public, private, and others you will learn about later.

- The return type—the data type of the value returned by the method, or void if the method does not return a value.
- The method name—the rules for field names apply to method names as well, but the convention is a little different.
- The parameter list in parenthesis—a comma-delimited list of input parameters, preceded by their data types, enclosed by parentheses, (). If there are no parameters, you must use empty parentheses.
- The method body, enclosed between braces—the method's code, including the declaration of local variables, goes here.

#### Naming a Method

Although a method name can be any legal identifier, code conventions restrict method names. By convention, method names should be a verb in lowercase or a multi-word name that begins with a verb in lowercase, followed by adjectives, nouns, etc. In multi-word names, the first letter of each of the second and following words should be capitalized. Here are some examples:

run

run Fast getBackground getFinalData compareTo setX isEmpty



Typically, a method has a unique name within its class. However, a method might have the same name as other methods due to *method overloading*.

#### **Overloading Methods**

• The Java programming language supports *overloading* methods, and Java can distinguish between methods with different *method signatures*. This means that methods within a class can have the same name if they have different parameter lists (there are some qualifications to this that will be discussed in the lesson titled "Interfaces and Inheritance").

- In the Java programming language, you can use the same name for all the drawing methods but pass a different argument list to each method. Thus, the data drawing class might declare four methods named draw, each of which has a different parameter list.
- Overloaded methods are differentiated by the number and the type of the arguments passed into the method.
- You cannot declare more than one method with the same name and the same number and type of arguments, because the compiler cannot tell them apart.
- The compiler does not consider return type when differentiating methods, so you cannot declare two methods with the same signature even if they have a different return type.
- Overloaded methods should be used sparingly, as they can make code much less readable.

#### <u>Classes</u>

- In object-oriented terms, we say that your bicycle is an *instance* of the *class of objects* known as bicycles. A *class* is the blueprint from which individual objects are created.
- Java classes contain *fields* and *methods*. A field is like a C++ data member, and a method is like a C++ member function. In Java, each class will be in its own .java file.

Each field and method has an *access level*:

- <u>private</u>: accessible only in this class
- (<u>package</u>): accessible only in this package
- protected: accessible only in this package and in all subclasses of this class
- <u>public</u>: accessible everywhere this class is available
- Each class has one of two possible access levels:
- (package): class objects can only be declared and manipulated by code in this package
- <u>Public</u>: class objects can be declared and manipulated by code in any package.
- <u>Object</u>: Object-oriented programming involves *inheritance*. In Java, all classes (built-in or user-defined) are (implicitly) subclasses of Object. Using an array of Object in the List class allows any kind of Object (an instance of any class) to be stored in the list. However, primitive types (int, char, etc) cannot be stored in the list.
- A method should be made static when it does not access any of the non-static fields of the class, and does not call any non-static methods.
- Java class objects exhibit the properties and behaviors defined by its class. A class can contain fields and methods to describe the behavior of an object. Current states of a class's corresponding object are stored in the object's instance variables.
  - Creating a class:

A class is created in the following way

Class <class name> { Member variables; Methods;

• 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.

#### Class Variables – Static Fields

• We use class variables also know as Static fields when we want to share characteristics across all objects within a class. When you declare a field to be static, only a single instance of the associated variable is created common to all the objects of that class. Hence when one object changes the value of a class variable, it affects all objects of the class. We can access a class variable by using the name of the class, and not necessarily using a reference to an individual object within the class. Static variables can be accessed even though no objects of that class exist. It is declared using static keyword.

#### <u>Class Methods – Static Methods</u>

Class methods, similar to Class variables can be invoked without having an instance of the class. Class methods are often used to provide global functions for Java programs. For example, methods in the java.lang.Math package are class methods. You cannot call non-static methods from inside a static method.

Bundling code into individual software objects provides a number of benefits, including:

- <u>Modularity:</u> The source code for an object can be written and maintained independently of the source code for other objects. Once created, an object can be easily passed around inside the system.
- <u>Information-hiding</u>: By interacting only with an object's methods, the details of its internal implementation remain hidden from the outside world.
- <u>Code re-use</u>: If an object already exists (perhaps written by another software developer), you can use that object in your program. This allows specialists to implement/test/debug complex, task-specific objects, which you can then trust to run in your own code.
- <u>Pluggability and debugging ease</u>: If a particular object turns out to be problematic, you can simply remove it from your application and plug in a different object as its replacement. This is analogous to fixing mechanical problems in the real world. If a bolt breaks, you replace *it*, not the entire machine.

An instance or an object for a class is created in the following way <class name> <object name>=new <constructor>();

#### **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.
- To relate this to the real world, consider the automatic transmission on an automobile.
- It encapsulates hundreds of bits of information about your engine, such as how much we are accelerating, the pitch of the surface we are on, and the position of the shift.
- The power of encapsulated code is that everyone knows how to access it and thus can use it regardless of the implementation details—and without fear of unexpected side effects.

## **Polymorphism:**

*Polymorphism* (from the Greek, meaning —many forms $\parallel$ ) is a feature that allows one interface to be used for a general class of actions.

- The specific action is determined by the exact nature of the situation. Consider a stack (which is a last-in, first-out list). We might have a program that requires three types of stacks. One stack is used for integer values, one for floating-point values, and one for characters. The algorithm that implements each stack is the same, even though the data being stored differs.
- In Java we can specify a general set of stack routines that all share the same names. 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.*
- Polymorphism allows us to create clean, sensible, readable, and resilient code.

#### class Hierarchies (Inheritance):

- Object-oriented programming allows classes to *inherit* commonly used state and behavior from other classes. Different kinds of objects often have a certain amount in common with each other.
- In the Java programming language, each class is allowed to have one direct superclass, and each superclass has the potential for an unlimited number of *subclasses*:
- Mountain bikes, road bikes, and tandem bikes, for example, all share the characteristics of bicycles (current speed, current pedal cadence, current gear). Yet each also defines additional features that make them different: tandem bicycles have two seats and two sets of handlebars; road bikes have drop handlebars; some mountain bikes have an additional chain ring, giving them a lower gear ratio. In this example, Bicycle now becomes the *super class* of Mountain Bike, Road Bike, and Tandem Bike.
- The syntax for creating a subclass is simple. At the beginning of your class declaration, use the extends keyword, followed by the name of the class to inherit from:

# class <sub class> extends <super class> { // new fields and methods defining a sub class would go here }

The different types of inheritance are

- 1. Single level Inheritance.
- 2. Multilevel Inheritance.
- 3. Hierarchical inheritance.
- 4. Multiple inheritance.
- 5. Hybrid inheritance.

Multiple, hybrid inheritance is not used in the way as other inheritances but it needs a special concept called interfaces.

#### Method Binding:

- Binding denotes association of a name with a class.
- Static binding is a binding in which the class association is made during compile time. This is also called as early binding.
- Dynamic binding is a binding in which the class association is not made until the object is created at execution time. It is also called as late binding.

#### Abstraction:

Abstraction in Java or Object oriented programming is a way to segregate implementation from interface and one of the five fundamentals along with Encapsulation, Inheritance, Polymorphism, Class and Object.

- An essential component of object oriented programming is Abstraction
- Humans manage complexity through abstraction.
- For example people do not think a car as a set of tens and thousands of individual parts. They think of it as a well defined object with its own unique behavior.
- This abstraction allows people to use a car ignoring all details of how the engine, transmission and braking systems work.
- In computer programs the data from a traditional process oriented program can be transformed by abstraction into its component objects.
- A sequence of process steps can become a collection of messages between these objects. Thus each object describes its own behavior.

#### **Overriding:**

- In a class hierarchy when a sub class has the same name and type signature as a method in the super class, then the method in the subclass is said to override the method in the super class.
- When an overridden method is called from within a sub class, it will always refer to the version of that method defined by the sub class.
- The version of the method defined by the super class will be hidden.

#### **Exceptions**:

- An exception is an abnormal condition that arises in a code sequence at run time.
- In other words an exception is a run time error.
- A java exception is an object that describes an exceptional condition that has occurred in a piece of code.
- When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error.
- Now the exception is caught and processed.

#### Summary of oops concepts

- **Object-oriented programming** (**OOP**) is a programming paradigm that represents concepts as "objects" that have data fields (attributes that describe the object) and associated procedures known as methods. Objects, which are usually instances of classes, are used to interact with one another to design applications and computer programs.
- Object-oriented programming is an approach to designing modular, reusable software systems.
- The goals of object-oriented programming are:
- Increased understanding.
- Ease of maintenance
- Ease of evolution.
- Object orientation eases maintenance by the use of encapsulation and information hiding.

#### **Object-Oriented Programming – Summary of Key Terms**

#### Definitions of some of the key concepts in Object Oriented Programming(OOP).

Term	Definition
Abstract Data Type	A user-defined data type, including both attr <u>ibutes (its state)</u> and <u>methods</u> (its behaviour). An object oriented language will include means to define new types (see class) and create instances of those classes (see object). It will also provide a number of <u>primitive types</u> .
Aggregation	Objects that are made up of other objects are known as aggregations. The relationship is generally of one of two types:
	<ul> <li>Composition – the object is composed of other objects. This form of aggregation is a form of code reuse. <i>E.g. A Car is composed of Wheels, a Chassis and an Engine</i></li> <li>Collection – the object contains other objects. <i>E.g. a List contains several Items; A Set several Members.</i></li> </ul>
Attribute	A characteristic of an object. Collectively the attributes of an object describe its state. <i>E.g. a Car may have attributes of</i>

Speed, Direction, Registration Number and Driver.         Class       The definition of objects of the same abstract data type. In Java class is the keyword used to define new types.         Dynamic (Late)       The identification at run time of which version of a method is being called (see polymorphism). When the class of an object cannot be identified at compile time, it is impossible to use static binding to identify the correct object method, so dynamic binding must be used.         Encapsulation       The combining together of attributes (data) and methods (behaviour/processes) into a single abstract data type with a public interface and a private implementation. This allows the implementation to be altered without affecting the interface.         Inheritance       The derivation of one class from another so that the attributes and methods of one class are part of the definition of another class. The first class is often referred to the base or parent class. The child is often referred to as a derived or sub-class.         Derived classes generally add to the attributes and/or behaviour of the base class. Inderivate some common attributes and behaviour but may add their own that are unique to that particular type.         Interface       The behaviour that a class exposes to the outside world; its public face. Also called its_contract <sup>*</sup> . In Java interface is also a keyword similar to class. However a Java interface contains on implementation. It is imply describes the behaviour expected of a particular type of object, it doesn't so how that behaviour should be implemented.         Member       See attribute       The invoking of a method of an object. In an object-oriented application objects send each other messages (i.e. execute ea		
Class       The definition of objects of the same <u>abstract data type</u> . In Java class is the keyword used to define new types.         Dynamic (Late)       The identification at run time of which version of a method is being called (see <u>polymorphism</u> ). When the class of an object cannot be identified at compile time, it is impossible to use <u>static binding</u> to identify the correct object method, so dynamic binding must be used.         Encapsulation       The combining together of attributes (data) and methods (behaviour/processes) into a single abstract data type with a public <u>interface</u> and a private implementation. This allows the implementation to be altered without affecting the interface.         Inheritance       The derivation of one class from another so that the attributes and methods of one class is often referred to the base or parent class. The child is often referred to as a derived or sub-class. Derived classes generally add to the attributes and/or behaviour of the base class. Inheritance is one form of object-oriented code reuse. <i>E.g. Both Motorbikes and Cars are kinds of MotorVehicles and therefore share some common attributes and behaviour that a class_exposes to the outside world; its public face. Also called its _contract'. In Java interface to a keyword similar to class. However a Java interface to salor a keyword similar to class. However a Java interface of a particular type of object, it doesn't so how that behaviour should be implemented.         Member       See <u>attribute</u>         Variable       The implementation of some behaviour of an <u>object</u>. The invoking of a method of an object. In an object-oriented application object send each other messages (i.e. execute each others methods) to achieve the desired behaviour.    </i>		Speed, Direction, Registration Number and Driver.
Dynamic (Late)The identification at run time of which version of a method is being called (see polymorphism). When the class of an object cannot be identified at compile time, it is impossible to use static binding to identify the correct object method, so dynamic binding must be used.EncapsulationThe combining together of attributes (data) and methods (behaviour/processes) into a single abstract data type with a public interface and a private implementation. This allows the implementation to be altered without affecting the interface.InheritanceThe derivation of one class from another so that the attributes and methods of one class are part of the definition of another class. The first class is often referred to the base or parent class. The child is often referred to as a derived or sub-class. Derived classes generally add to the attributes and behaviour of the base class. Berived class exposes to the outside world; its public face. Also called its _contract'. In Java interface is also a keyword similar to class. However a Java interface cotains no implementation: it simply describes the behaviour expected of a particular type of object, it doesn't so how that behaviour should be implemented.MemberSee attribute variableMethodThe implementation of some behaviour of an object-oriented application object seed of an object. In an object-oriented application object, it doesn't so how that behaviour should be implemented.	Class	The definition of objects of the same <u>abstract data type</u> . In Java class is the keyword used to define new types.
EncapsulationThe combining together of attributes (data) and methods (behaviour/processes) into a single abstract data type with a public interface and a private implementation. This allows the implementation to be altered without affecting the interface.InheritanceThe derivation of one class from another so that the attributes and methods of one class are part of the definition of another class. The first class is often referred to the base or parent class. The child is often referred to as a derived or sub-class. Derived classes generally add to the attributes and/or behaviour of the base class. Inheritance is one form of object-oriented code reuse. E.g. Both Motorbikes and Cars are kinds of MotorVehicles and therefore share some common attributes and behaviour but may add their own that are unique to that particular type.InterfaceThe behaviour that a class exposes to the outside world; its public face. Also called its _contract'. In Java interface is also a keyword similar to class. However a Java interface is also a keyword similar to class. However a Java interface contains no implementation: it simply describes the behaviour expected of a particular type of object, it doesn't so how that behaviour should be implemented.MemberSee attribute attribute The invoking of a method of an object. In an object-oriented application objects send each other messages (i.e. execute each others methods) to achieve the desired behaviour.	Dynamic (Late) Binding	The identification at run time of which version of a method is being called (see <u>polymorphism</u> ). When the class of an object cannot be identified at compile time, it is impossible to use <u>static binding</u> to identify the correct object method, so dynamic binding must be used.
InheritanceThe derivation of one class from another so that the attributes and methods of one class are part of the definition of another class. The first class is often referred to the base or parent class. The child is often referred to as a derived or sub-class. Derived classes are always _a kind of their base classes. Derived classes generally add to the attributes and/or behaviour of the base class. Inheritance is one form of object-oriented code reuse. E.g. Both Motorbikes and Cars are kinds of MotorVehicles and therefore share some common attributes and behaviour but may add their own that are unique to that particular type.InterfaceThe behaviour that a class exposes to the outside world; its public face. Also called its _contract'. In Java interface is also a keyword similar to class. However a Java interface contains no implementation: it simply describes the behaviour should be implemented.MemberSee attribute The implementation of some behaviour of an object.MethodThe implementation of some behaviour of an object.MessageThe invoking of a method of an object. In an object-oriented application objects send each other messages (i.e. execute each others methods) to achieve the desired behaviour.	Encapsulation	The combining together of att <u>ributes (data)</u> and methods (behaviour/processes) into a single abstract data type with a public <u>interface</u> and a private implementation. This allows the implementation to be altered without affecting the interface.
InterfaceThe behaviour that a class exposes to the outside world; its public face. Also called its _contract'. In Java interface is also a keyword similar to class. However a Java interface contains no implementation: it simply describes the behaviour expected of a particular type of object, it doesn't so how that behaviour should be implemented.MemberSee attribute VariableMethodThe implementation of some behaviour of an object.MessageThe invoking of a method of an object. In an object-oriented application objects send each other messages (i.e. execute each others methods) to achieve the desired behaviour.	Inheritance	The derivation of one class from another so that the attributes and methods of one class are part of the definition of another class. The first class is often referred to the base or parent class. The child is often referred to as a derived or sub-class. Derived classes are always _a kind of their base classes. Derived classes generally add to the attributes and/or behaviour of the base class. Inheritance is one form of object-oriented code reuse. <i>E.g. Both Motorbikes and Cars are kinds of</i> <i>MotorVehicles and therefore share some common attributes</i> <i>and behaviour but may add their own that are unique to that</i>
MemberSee attributeVariableImage: Constraint of the second sec	Interface	The behaviour that a class exposes to the outside world; its public face. Also called its _contract'. In Java interface is also a keyword similar to class. However a Java interface contains no implementation: it simply describes the behaviour expected of a particular type of object, it doesn't so how that behaviour should be implemented.
VariableMethodThe implementation of some behaviour of an <u>object</u> .MessageThe invoking of a method of an object. In an object-oriented application objects send each other messages (i.e. execute each others methods) to achieve the desired behaviour.	Member	See <u>attribute</u>
MethodThe implementation of some behaviour of an object.MessageThe invoking of a method of an object. In an object-oriented application objects send each other messages (i.e. execute each others methods) to achieve the desired behaviour.	Variable	
MessageThe invoking of a method of an object. In an object-oriented application objects send each other messages (i.e. execute each others methods) to achieve the desired behaviour.	Method	The implementation of some behaviour of an <u>object</u> .
	Message	The invoking of a method of an object. In an object-oriented application objects send each other messages (i.e. execute each others methods) to achieve the desired behaviour.

Object	An instance of a class. Objects have state, identity and behaviour.
Overloading	Allowing the same method name to be used for more than one implementation. The different versions of the method vary according to their parameter lists. If this can be determined at compile time then static binding is used, otherwise dynamic binding is used to select the correct method as runtime.
Polymorphism	Generally, the ability of different classes of object to respond to the same message in different, class-specific ways. Polymorphic methods are used which have one name but different implementations for different classes. <i>E.g. Both the</i> <i>Plane and Car types might be able to respond to a turnLeft</i> <i>message. While the behaviour is the same, the means of</i> <i>achieving it are specific to each type.</i>
Primitive Type	The basic types which are provided with a given object- oriented programming language. <i>E.g. int, float, double, char,</i> <i>Boolean</i>
Static(Early) Binding	The identification at compile time of which version of a polymorphic method is being called. In order to do this the compiler must identify the <u>class</u> of an object.

Java was conceived by James gosling, Patrick Naughton, chriswarth, Ed frank and Mike Sheridan at sun Microsystems.

The original impetus for java was not internet instead primary motivation was the need for a platform independent (i.e. Architectural neutral) independent language.

#### Java's Byte code:

The key that allows java to solve the both security and portability problems is that the output of a java compiler is not executable code rather it is byte code.

Byte code is highly optimized set of instructions designed to be executed by java runtime systems, which is called Java Virtual Machine (JVM). JVM is interpreter for byte code. Translating a java program into byte code helps makes it much easier to run a Program in a wide variety of environments. The reason is straightforward: only the JVM needs to be implemented for each platform. Once the run-time package exists for a given system, any Java program can run on it.

#### The Java Buzzwords:

No discussion of the genesis of Java is complete without a look at the Java buzzwords. Although the fundamental forces that necessitated the invention of Java are portability and security, other factors also played an important role in mol ding the final form of the language. The key considerations were summed up by the Java team in the following list of buzzwords:

- Simple
- Secure
- Portable
- Object-oriented
- Robust
- Multithreaded
- Architecture-neutral
- Interpreted
- High performance
- Distributed
- Dynamic

#### Simple

Java was designed to be easy for the professional programmer to learn and use effectively. Assuming that you have some programming experience, you will not find Java hard to master. If you already understand the basic concepts of object-oriented programming, learning Java will be even easier. Best of all, if you are an experienced C++ programmer, moving to Java will require very little effort. Because Java inherits the C/C++ syntax and many of the object-oriented features of C++

#### <u>Robust</u>

The multiplatform environment of the Web places extraordinary demands on a program, because the program must execute reliably in a variety of systems. Thus, the ability to create robust programs were given a high priority in the design of Java.

To gain reliability, Java restricts you in a few key areas, to force you to find your mistakes early in program development. At the same time, Java frees you from having to worry about many of the most common causes of programming errors. Because Java is a strictly typed language, it checks your code at compile time. However, it also checks your code at run time. To better understand how Java is robust, consider two of the main reasons for program failure: memory management mistakes and mishandled exceptional conditions (that is, run-time errors). Memory management can be a difficult, tedious task in traditional programming environments. For example, in C/C++, the programmer must manually allocate and free all dynamic memory. This sometimes leads to problems, because programmers will either forget to free memory that has

been previously allocated or, worse, try to free some memory that another part of their code is still using. Java virtually eliminates these problems by managing memory allocation and deallocation for you. (In fact, deallocation is completely automatic,

SVECW IT DEPT 13

Page

because Java provides garbage collection for unused objects.) Exceptional conditions in traditional environments often arise in situations such as division by zero or —file not found, || and they must be managed with clumsy and hard-to-read constructs. Java helps in this area by providing object-oriented exception handling. In a well-written Java program, all run-time errors can—and should—be managed by your program.

#### **Multithreaded**

Java was designed to meet the real-world requirement of creating interactive, networked programs. To accomplish this, Java supports multithreaded programming, which allows you to write programs that do many things simultaneously.

#### Architecture-Neutral

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

#### **Interpreted and High Performance**

Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java byte code. This code can be interpreted on any system that provides a Java Virtual Machine. Most previous attempts at cross platform solutions have done so at the expense of performance. Java was engineered for interpretation, the Java byte code was carefully designed so that it would be easy to translate directly into native machine code for very high performance by using a just-in-time compiler. Java run-time systems that provide this feature lose none of the benefits of the platform-independent code. —High-performance cross-platform|| is no longer an oxymoron.

#### **Distributed**

Java is designed for the distributed environment of the Internet, because it handles TCP/IP protocols. In fact, accessing a resource using a URL is not much different from accessing a file. The original version of Java (Oak) included features for intraaddress-space messaging. This allowed objects on two different computers to execute procedures remotely. Java revived these interfaces in a package called *Remote Method Invocation* (*RMI*). This feature brings an unparalleled level of abstraction to client/server programming.

#### **Dynamic**

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

#### <u>Security</u>

Every time that you download a —normal|| program, you are risking a viral infection. Even so, most users still worried about the possibility of infecting their systems with a virus. In addition to viruses, another type of malicious program exists that must be guarded against. This type of program can gather private information, such as credit card numbers, bank account balances, and passwords, by searching the contents of your computer's local file system. Java answers both of these concerns by providing a —firewall|| between a networked application and your computer. When you use a Java-compatible Web browser, you can safely download Java applets without fear of viral infection or malicious intent. Java achieves this protection by confining a Java program to the Java execution environment and not allowing it access to other parts of the computer.

#### **Portability**

Many types of computers and operating systems are in use throughout the world—and many are connected to the Internet. For programs to be dynamically downloaded to all the various types of platforms connected to the Internet, some means of generating portable executable code is needed. As you will soon see, the same mechanism that helps ensure security also helps create portability.

#### Data Types:

Java defines eight simple types of data: **byte**, **short**, **int**, **long**, **char**, **float**, **double**, and **Boolean**. These can be put in four groups:

- Integers this group includes **byte**, **short**, **int**, and **long**, which are for whole valued signed numbers.
- Floating-point numbers this group includes **float** and **double**, which represent numbers with fractional precision.
- Characters this group includes **char**, which represents symbols in a character set, like letters and numbers.
- Boolean this group includes **Boolean**, which is a special type for representing true/false values.

#### **Integers:**

The width and ranges of these integer types vary widely, as shown in this table:

Name	Width	Range
long	64	-9,223,372,036,854,775,808 to
int	32	9,223,372,036,854,775,807 -2,147,483,648 to 2,147,483,647
short	16	-32,768 to 32,767
byte	8	-128 to 127

#### **Floating-point:**

There are two kinds of floating-point types, **float** and **double**, which represent single- and double-precision numbers, respectively. Their width and ranges are shown here:

Name	Width in Bits	Approximate Range
double	64	4.9e-324 to 1.8e+308
float	32	1.4e-045 to 3.4e+038

#### Variables:

The variable is the basic unit of storage in a Java program. A variable is defined by the combination of an identifier, a type, and an optional initializer. In addition, all variables have a scope, which defines their visibility, and a lifetime. In Java, all variables must be declared before they can be used. The basic form of a variable declaration is shown here:

type identifier [ = value][, identifier [= value] ...];

The *type* is one of Java's atomic types, or the name of a class or interface. The *identifier* is the name of the variable. You can initialize the variable by specifying an equal sign and a value. To declare more than one variable of the specified type, use a comma-separated list.

Here are several examples of variable declarations of various types. Note that some include an initialization.

int a, b, c; // declares three ints, a, b, and c. int d = 3, e, f = 5; // declares three more ints byte z = 22; // initializes z. double pi = 3.14159; // declares an approximation of pi. char x = 'x'; // the variable x has the value 'x'.

#### The Scope and Lifetime of Variables:

All of the variables used till now have been declared at the start of the **main()** method. However, Java allows variables to be declared within any block. A block is begun with an opening curly brace and ended by a closing curly brace. A block defines a *scope*. Thus, each time you start a new block, you are creating Scope determines what objects are visible to other parts of your program. It also determines the lifetime of those objects. Most other computer languages define two general categories of scopes: global and local. 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. 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. 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.

To understand the effect of nested scopes, consider the following program:

```
// demonstrate block
scope. 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);
   } }
```

As the comments indicate, the variable  $\mathbf{x}$  is declared at the start of **main**()'s scope and is accessible to all subsequent code within **main**(). Within the **if** block,  $\mathbf{y}$  is declared. Since a block defines a scope,  $\mathbf{y}$  is only visible to other code within its block. This is why outside of its block, the line  $\mathbf{y} = \mathbf{100}$ ; is commented out. If you remove the leading comment symbol, a compile-time error will occur, because  $\mathbf{y}$  is not visible outside of its block. Within the **if** block,  $\mathbf{x}$  can be used because code within a block (that is, a nested scope) has access to variables declared by an enclosing scope.

Here is another important point to remember: variables are created when their scope is entered and destroyed when their scope is left. This means that a variable will not hold its value once it has gone out of scope. Therefore, variables declared within a method will not hold their values between calls to that method. Also, a variable declared within a block will lose its value when the block is left. Thus, the lifetime of a variable is confined to its scope. If a variable declaration includes an initializer, then that variable will be reinitialized each time the block in which it is declared is entered. For example, consider the next program.

```
// Demonstrate lifetime of a
variable. class Lifetime
{
    public static void main(String args[])
    {
        int x;
    }
}
```

```
for(x = 0; x < 3; x++)
{
    int y = -1; // y is initialized each time block is entered
    System.out.println("y is: " + y); // this always prints -1
    y = 100;
    System.out.println("y is now: " + y);
    }
}</pre>
```

The output generated by this program is shown here: y is: -1 y is now: 100 y is: -1 y is now: 100 y is: -1 y is now: 100 As you can see y is always reinitialized to -1 each

As you can see,  $\mathbf{y}$  is always reinitialized to -1 each time the inner **for** loop is entered. Even though it is subsequently assigned the value 100, this value is lost.One last point: Although blocks can be nested, you cannot declare a variable to have the same name as one in an outer scope.

#### Arrays:

An *array* is a group of similar-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.

#### **One-Dimensional Arrays**

A *one-dimensional array* is a list of like-typed variables. To create an array, you first must create an array variable of the desired type. The general form of a one dimensional array declaration is

type var-name[];

Here, *type* declares the base type of the array. For example, the following declares an array named **month** with the type —array of int||:

int month [];

Although this declaration establishes the fact that **month** is an array variable, no array actually exists. In fact, the value of **month** is set to **null**, which represents an array with no value. To link **month** with an actual, physical array of integers, you must allocate one using **new** and assign it to **month**. **new** is a special operator that allocates memory. The general form of **new** as it applies to one-dimensional arrays appears as follows:

*array-var* = new *type*[*size*];

Here, *type* specifies the type of data being allocated, *size* specifies the number of elements in the array, and *array-var* is the array variable that is linked to the array. That is, to use **new** to

allocate an array, you must specify the type and number of elements to allocate. The elements in the array allocated by **new** will automatically be initialized to zero.

This example allocates a 12-element array of integers and links them to **month** month = new int[12];

After this statement executes, **month** will refer to an array of 12 integers. Further, all elements in the array will be initialized to zero.

Another way to declare an arrayin single step is type arr-name=new type[size];

Arrays can be initialized when they are declared. The process is much the same as that used to initialize the simple types. An *array initializer* is a list of comma-separated expressions surrounded by curly braces. The commas separate the values of the array elements. The array will automatically be created large enough to hold the number of elements you specify in the array initializer. There is no need to use **new**. For example, to store the number of days in each month, we do as

follows // An improved version of the previous program.

```
class AutoArray
{
    public static void main(String args[])
    {
        int month[] = { 31, 28, 31, 30, 31, 30, 31, 30, 31, 30, 31, 30, 31 };
        System.out.println("April has " + month[3] + " days."); }
```

```
}
```

When you run this program, in the output it prints the number of days in April. As mentioned, Java array indexes start with zero, so the number of days in April is **month[3]** or 30.

Here is one more example that uses a one-dimensional array. It finds the average of a set of numbers.

```
// Average an array of values.
class Average
{
    public static void main(String args[])
    {
        double nums[] = {10.1, 11.2, 12.3, 13.4,
        14.5}; double result = 0;
```

```
int i;
for(i=0; i<5; i++)
result = result + nums[i];
System.out.println("Average is " + result / 5);
}
Output: Average is:12.3
```

```
Multidimensional Arrays
```

In Java, multidimensional arrays are actually arrays of arrays. To declare a multidimensional array variable, specify each additional index using another set of square brackets. For example, the following declares a two-dimensional array variable called **twoD**.

```
int twoD[][] = new int[4][5];
This allocates a 4 by 5 array and assigns it to twoD.
program :
// Demonstrate a two-dimensional
array. class TwoDArray
ł
  public static void main(String args[])
  {
    int twoD[][]= new
    int[4][5]; int i, j, k = 0;
    for(i=0; i<4; i++)
    for(j=0; j<5; j++)
    ł
    twoD[i][j] =
    k; k++;
   for(i=0; i<4; i++)
   for(j=0; j<5; j++)
   System.out.print(twoD[i][j] + "
   "); System.out.println();
   }
  }
Output:
01234
56789
10 11 12 13 14
15 16 17 18 19
```

When you allocate memory for a multidimensional array, you need only specify the memory for the first (leftmost) dimension. You can allocate the remaining dimensions separately. We can allocates the second dimension manually.

```
int twoD[][] = new int[4][];
twoD[0] = new int[5];
twoD[1] = new int[5];
twoD[2] = new int[5];
twoD[3] = new int[5];
we can creates a two dimensional array in which the sizes of the second dimension are
unequal.
// Manually allocate differing size second dimensions.
class TwoDAgain
  public static void main(String args[])
   int twoD[][] = new
   int[4][]; twoD[0] = new
   int[1]; twoD[1] = new
   int[2]; twoD[2] = new
   int[3]; twoD[3] = new
   int[4];
   int i, j, k = 0;
   for(i=0; i<4; i++)
   for(j=0; j<i+1; j++)
   twoD[i][j] =
   k; k++;
   ł
   for(i=0; i<4; i++)
   for(j=0; j<i+1; j++)
   System.out.print(twoD[i][j] + "
   "); System.out.println();
   }
ł
Output:
0
12
345
678
We can create a three-dimensional array where first index specifies the number of tables,
second one number of rows and the third number of columns.
// Demonstrate a three-dimensional
array. class threeDMatrix
{
   public static void main(String args[])
```

```
int threeD[][][] = new int[3][4][5];
```

```
int i, j, k;
   for(i=0; i<3; i++)
   for(j=0; j<4; j++)
   for(k=0; k<5; k++)
   threeD[i][j][k] = i * j * k;
   for(i=0; i<3; i++)
   {
    for(j=0; j<4; j++)
    for(k=0; k<5; k++)
    System.out.print(threeD[i][j][k] + " ");
   System.out.println();
  }
 System.out.println();
 }
Output:
00000
00000
00000
00000
00000
01234
02468
036912
00000
02468
0481216
0 6 12 18 24
Alternative Array Declaration Syntax
There is a second form that may be used to declare an array:
type[ ] var-name;
Here, the square brackets follow the type specifier, and not the name of the array
variable. For example, the following two declarations are equivalent: int al[] = new
int[3];
```

int[] a2 = new int[3];

The following declarations are also equivalent:

char twod1[][] = new char[3][4]; char[][] twod2 = new char[3][4];

#### **Operators:**

Operators are special symbols that perform specific operations on one, two, or three *operands*, and then return a result. The operators in the following table are listed according to precedence order. The closer to the top of the table an operator appears, the higher its precedence. Operators with higher precedence are evaluated before operators with relatively lower precedence. Operators on the same line have equal precedence. When operators of equal precedence appear in the same expression, a rule must govern which is evaluated first. All binary operators except for the assignment operators are evaluated right to left.

Operator Precedence		
Operators	Precedence	
Postfix	expr++, expr	
Unary	$++exprexpr +expr -expr \sim !$	
multiplicative	* / %	
additive	+ -	
shift	<< >> >>>	
relational	<><=>= instanceof	
equality	== !=	
bitwise AND	&	
bitwise	٨	
exclusive OR		
bitwise		
inclusive OR	I	
logical AND	&&	
logical OR		
ternary	?:	
assignment	= += -= *= /= %= &= ^=  = <<= >>= >>=	

In general-purpose programming, certain operators tend to appear more frequently than theirs; for example, the assignment operator "=" is far more common than the unsigned right shift operator ">>>".

#### Expressions:

An *expression* is a construct made up of variables, operators, and method invocations, which are constructed according to the syntax of the language that evaluates to a single value.

```
int a = 0;
arr[0] = 100;
System.out.println("Element 1 at index 0: " + arr[0]);
```

```
int result = 1 + 2; // result is now 3
if(value1 == value2)
System.out.println("value1 == value2");
```

The data type of the value returned by an expression depends on the elements used in the expression. The expression a= 0 returns an int because the assignment operator returns a value of the same data type as its left-hand operand; in this case, cadence is an int. As you can see from the other expressions, an expression can return other types of values as well, such as boolean or String.

For example, the following expression gives different results, depending on whether you perform the addition or the division operation first:

x + y / 100 // ambiguous

You can specify exactly how an expression will be evaluated using balanced parenthesis rewrite the expression as

(x + y) / 100 // unambiguous, recommended

If you don't explicitly indicate the order for the operations to be performed, the order is determined by the precedence assigned to the operators in use within the expression. Operators that have a higher precedence get evaluated first. For example, the division operator has a higher precedence than does the addition operator. Therefore, the following two statements are equivalent:

x + y / 100

x + (y / 100) // unambiguous, recommended

When writing compound expressions, be explicit and indicate with parentheses which operators should be evaluated first.

#### Control Statements:

#### **IF Statement**

The general form of the **if** statement:

*if* (*condition*) *statement1;* 

Here, each *statement* may be a single statement or a compound statement enclosed in curly braces (that is, a *block*). The *condition* is any expression that returns a **boolean** value.

If the *condition* is true, then *statement1* is executed. Otherwise, *statement2* is executed.

#### **IF**-**ELSE** Statement

The general form of the **if** statement:

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

The if-then-else statement provides a secondary path of execution when an "if" clause evaluates to false.

```
void applyBrakes()
```

```
{
    if (isMoving)
    {
        currentSpeed--;
    }
    else
    {
        System.err.println("The bicycle has already stopped!");
    }
}
```

The switch Statement

Unlike if-then and if-then-else, the switch statement allows for any number of possible execution paths. A switch works with the byte, short, char, and int primitive data types. It also works with *enumerated types*.

Program: displays the name of the month, based on the value of month, using the switch statement.

```
class SwitchDemo
{
   public static void main(String[] args) {
     int month = 8;
     switch (month) {
        case 1: System.out.println("January");
            break;
        case 2: System.out.println("February");
        }
}
```

```
break;
```

```
case 3: System.out.println("March");
                break:
       case 4: System.out.println("April");
                break;
       case 5: System.out.println("May");
                break;
       case 6: System.out.println("June");
                break;
       case 7: System.out.println("July");
                 break:
       case 8: System.out.println("August");
                break;
       case 9: System.out.println("September");
                break;
       case 10: System.out.println("October");
                 break;
       case 11: System.out.println("November");
                 break;
       case 12: System.out.println("December");
                 break;
       default: System.out.println("Invalid month.");
                 break;
     ļ
}
```

```
Output:"August"
```

The body of a switch statement is known as a *switch block*. Any statement immediately contained by the switch block may be labeled with one or more case or default labels. The switch statement evaluates its expression and executes the appropriate case.

#### The while and do-while Statements

The while statement continually executes a block of statements while a particular condition is true.

Its syntax can be expressed as:

```
while (expression)
```

```
statement(s)
```

}

{

The while statement evaluates *expression*, which must return a boolean value. If the expression evaluates to true, the while statement executes the *statement*(s) in the while block. The while statement continues testing the expression and executing its block until

the expression evaluates to false. Using the while statement to print the values from 1 through 10 can be accomplished as in the following program:

```
class WhileDemo
{
   public static void main(String[] args)
  {
      int count = 1:
      while (count < 11)
{
         System.out.println("Count is: " + count);
         count++;
      }
   }
}
do-while statement
Its syntax can be expressed as:
do
ł
   statement(s)
} while (expression);
```

The difference between do-while and while is that do-whil

The difference between do-while and while is that do-while evaluates its expression at the bottom of the loop instead of the top. Therefore, the statements within the do block are always executed at least once.

```
Program:
class DoWhileDemo
{
    public static void main(String[] args){
        int count = 1;
        do {
            System.out.println("Count is: " + count);
            count++;
        } while (count <= 11);
    }
}</pre>
```

## The for Statement

The for statement provides a compact way to iterate over a range of values. Programmers often refer to it as the "for loop" because of the way in which it repeatedly loops until a particular condition is satisfied. The general form of the for statement can be expressed as follows:

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

When using the for statement, we need to remember that

- The *initialization* expression initializes the loop; it's executed once, as the loop begins.
- When the *termination* expression evaluates to false, the loop terminates.
- The *increment* expression is invoked after each iteration through the loop; it is perfectly acceptable for this expression to increment *or* decrement a value.

#### Type Conversion and Casting:

We can assign a value of one type to a variable of another type. If the two types are compatible, then Java will perform the conversion automatically. For example, it is always possible to assign an int value to a long variable. However, not all types are compatible, and thus, not all type conversions are implicitly allowed. For instance, there is no conversion defined from double to byte.

But it is possible for conversion between incompatible types. To do so, you must use a *cast, which* performs an explicit conversion between incompatible types.

Java's Automatic Conversions

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

- The two types are compatible.
- The destination type is larger than the source type.

When these two conditions are met, a widening conversion takes place. For example, the int type is always large enough to hold all valid byte values, so no explicit cast statement is required.

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

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

Casting Incompatible Types

The automatic type conversions are helpful, they will not fulfil all needs. For example, if we want to assign an int value to a byte variable. This conversion will not be performed automatically, because a byte is smaller than an int. This kind of conversion is sometimes called a *narrowing conversion*, since you are explicitly making the value narrower so that

it will fit into the target type. To create a conversion between two incompatible types, you must use a cast. A *cast* is simply an explicit type conversion. It has this general form:

(target-type) value Here, target-type specifies the desired type to convert the specified value to. Example: int a; byte b; // ... b = (byte) a; A different type of conversion will occur when a floating-point value is assigned to an integer type: truncation. As integers do not have fractional components so,when a floating-point value is assigned to an integer type, the fractional component is lost.

#### **Program:**

```
class Conversion
 public static void main(String args[])
 ł
  byte b;
  int i = 257;
  double d = 323.142;
  System.out.println("\nConversion of int to byte.");
  b = (byte) i;
  System.out.println("i and b " + i + " " + b);
  System.out.println("\nConversion of double to
  int."); i = (int) d;
 System.out.println("d and i " + d + " " + i);
 System.out.println("\nConversion of double to
 byte."); b = (byte) d;
 System.out.println("d and b " + d + " " + b);
 }
}
```

Output: Conversion of int to byte. i and b 257 1 Conversion of double to int. d and i 323.142 323 Conversion of double to byte. d and b 323.142 67 byte b = 50; b = (byte)(b \* 2);

### **The Type Promotion Rules:**

In addition to the elevation of bytes and shorts to int, Java defines several *type promotion rules* that apply to expressions. They are as follows. First, all byte and short values are promoted to int. Then, if one operand is a long, the whole expression is promoted to long. If one operand is a float, the entire expression is promoted to float. If any of the operands is double, the result is double.

The following program demonstrates how each value in the expression gets promoted to match the second argument to each binary operator:

#### class Promote

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

{

byte b = 42;

char c = 'a';

short s = 1024;

int i = 50000;

float f = 5.67f;

double d = .1234;

double result = (f * b) + (i / c) - (d * s);

System.out.println((f * b) + " + " + (i / c) + " - " + (d * s));

System.out.println("result = " + result); }
```

#### }

double result = (f \* b) + (i / c) - (d \* s);

In the first sub expression, f \* b, b is promoted to a float and the result of the sub expression is float. Next, in the sub expression i / c, c is promoted to int, and the result is of type int. Then, in d \* s, the value of s is promoted to double, and the type of the sub expression is double. Finally, these three intermediate values, float, int, and double, are considered. The outcome of float plus an int is a float. Then the resultant float minus the last double is promoted to double, which is the type for the final result of the expression.

Simple Java Program:

```
class Example
```

```
{
```

public static void main(String args[])

```
{
```

System.out.println("This is a simple Java program.");

Here public is an access modifier, which means this method can be accessed by any one out side the class.

Static allows the main () method to be called without initiating any instance for the class. Void tells the compiler that main() doesnot return any type.

String args[] declares a parameter named args, which is an array of instances of class string.

args takes the arguments for a command line.

#### **Classes and Objects:**

In the real world, you'll often find many individual objects all of the same kind. There may be thousands of other bicycles in existence, all of the same make and model. Each bicycle was built from the same set of blueprints and therefore contains the same components. In object-oriented terms, we say that your bicycle is an *instance* of the *class of objects* known as bicycles. A *class* is the blueprint from which individual objects are created.

#### **Declaring Member Variables**

There are several kinds of variables:

- Member variables in a class—these are called *fields*.
- Variables in a method or block of code—these are called *local variables*.
- Variables in method declarations—these are called *parameters*.

```
A class is declared by use of the class
keyword class classname
{
type instance-variable1;
type instance-variable2;
// ...
type instance-variableN;
type methodname1(parameter-list)
{
// body of method
}
type methodname2(parameter-list)
{
// body of method
}
// ...
type methodnameN(parameter-list)
{
```

The data, or variables, defined within a **class** are called *instance variables*. The code is contained within *methods*. Collectively, the methods and variables defined within a class are called *members* of the class. In most classes, the instance variables are acted upon and accessed by the methods defined for that class. Variables defined within a class are called

instance variables because each instance of the class (that is, each object of the class) contains its own copy of these variables. Thus, the data for one object is separate and unique from the data for another.

```
Example:

Class demo

{

    int x=1;;

    int y=2;

    float z=3;

    void display()

    {

        System.out.println(--values of x, y and z are:||+x+|| --+y||+|| --+z);

    }

}
```

## **Declaring Objects**

when you create a class, you are creating a new data type. You can use this type to declare objects of that type. However, obtaining objects of a class is a two-step process. First, you must declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can *refer* to an object. Second, you must acquire an actual, physical copy of the object and assign it to that variable. You can do this using the **new** operator. The **new** operator dynamically allocates (that

is, allocates at run time) memory for an object and returns a reference to it. This reference is, more or less, the address in memory of the object allocated by **new**.

This reference is then stored in the variable. Thus, in Java, all class objects must be dynamically allocated.

In the above programs to declare an object of type demo:

Demo d1 = new demo();

This statement combines the two steps just described. It can be rewritten like this

#### to show each step more clearly:

demo d1; // declare reference to object

d1 = new demo(); // allocate a demo object

The first line declares **d1 a**s a reference to an object of type demo. After this line executes, d1 contains the value **null**, which indicates that it does not yet point to an actual object. Any attempt to use d1 at this point will result in a compile-time error. The next line allocates an actual object and assigns a reference to it to **d1**. After the second line executes; you can use d1 **as** if it were a demo object. But in reality, d1 simply holds the memory address of the actual demo object. The effect of these two lines of code is depicted in Figure.

#### **Constructors:**

A class contains constructors that are invoked to create objects from the class blueprint. Constructor declarations look like method declarations—except that they use the name of the class and have no return type. A constructor initializes an object immediately upon creation.

Constructors can be default or parameterized constructors.

A default constructor is called when an instance is created for a class.

```
Example
Class demo
{
int x;
int y;
float z;
demo()
{
X=1;
Y=2;
Z=3;
}
void display()
System.out.println(—values of x, y and z are: ||+x+|| - |+y|| + || - |+z|;
Class demomain
Public static void main(String args[])
demo d1=new demo(); // this is a call for the above default constructor
d1.display();
}
}
Parameterized constructor:
Class demo
{
int x;
int y;
float z;
demo(int x1,int y1,int z1)
{
 x=x1;
```

```
y=y1;
z=z1;
}
void display()
{
System.out.println(--values of x, y and z are: ||+x+|| --+y||+|| --+z);
}
Class demomain
{
Public static void main(String args[])
{
demo d1=new demo(1,2,3); // this is a call for the above parameterized constructor
d1.display();
}
```

## This Keyword:

Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines the **this** keyword. **this** can be used inside any method to refer to the *current* object. That is, **this** is always a reference to the object on which the method was invoked. You can use **this** anywhere a reference to an object of the current class' type is permitted. Example:

```
Class demo
{
int x;
int y;
float z;
demo(int x,int y,int z)
ł
 this.x=x;
 this.y=y;
 this.z=z;
void display()
System.out.println(—values of x, y and z are: ||+x+|| —+y||+|| —+z);
Class demomain
Public static void main(String args[])
demo d1=new demo(1,2,3); // this is a call for the above parameterized constructor
d1.display();
}
```
}

Output:

Values of x, y and z are:1 2 3

To differentiate between the local and instance variables we have used this keyword int the constructor.

## **Garbage Collection:**

Since objects are dynamically allocated by using the **new** operator, objects are destroyed and their memory released for later reallocation. In some languages, such as C++, dynamically allocated objects must be manually released by use of a **delete** operator. Java takes a different approach; it

handles deallocation for you automatically. The technique that accomplishes this is called *garbage collection*. It works like this: when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed. There is no explicit need to destroy objects as in C++. The finalize() Method

Sometimes an object will need to perform some action when it is destroyed. For example, if an object is holding some non-Java resource such as a file handle or window character font, then you might want to make sure these resources are freed before an object is destroyed. To handle such situations, Java provides a mechanism called *finalization*. By using finalization, you can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector.

To add a finalizer to a class, you simply define the finalize() method. The Java run time calls that method whenever it is about to recycle an object of that class. Inside the finalize() method you will specify those actions that must be performed before an object is destroyed. The garbage collector runs periodically, checking for objects that are no longer referenced by any running state or indirectly through other referenced objects. Right before an asset is freed, the Java run time calls the finalize() method on the object. The finalize() method has this general form:

protected void finalize()

{

// finalization code here

}

Here, the keyword **protected** is a specifier that prevents access to **finalize(**) by code defined outside its class.

## **Overloading Methods:**

The Java programming language supports *overloading* methods, and Java can distinguish between methods with different *method signatures*. This means that methods within a class can have the same name if they have different parameter lists.

Suppose that you have a class that can use calligraphy to draw various types of data (strings, integers, and so on) and that contains a method for drawing each data type. It is cumbersome to use a new name for each method—for example, drawString, drawInteger, drawFloat, and so on. In the Java programming language, you can use the same name for all the drawing methods but pass a different argument list to each method. Thus, the data drawing class might declare four methods named draw, each of which has a different parameter list.

```
public class DataArtist {
```

```
public void draw(String s) {
    ...
}
public void draw(int i) {
    ...
}
public void draw(double f) {
    ...
}
public void draw(int i, double f) {
    ...
}
```

Overloaded methods are differentiated by the number and the type of the arguments passed into the method. In the code sample, draw(String s) and draw(int i) are distinct and unique methods because they require different argument types. You cannot declare more than one method with the same name and the same number and type of arguments, because the compiler cannot tell them apart. The compiler does not consider return type when differentiating methods, so you cannot declare two methods with the same signature even if they have a different return type.

# **Overloading Constructors:**

```
We can overload constructor methods
class Box {
  double width;
  double height;
  double depth;
  // This is the constructor for Box.
  Box(double w, double h, double d)
  { width = w;
  height =
  h; depth =
  d;
  }
  // compute and return volume
  double volume() {
  return width * height * depth;
```

```
}
```

}

As you can see, the **Box()** constructor requires three parameters. This means that all declarations of **Box** objects must pass three arguments to the **Box()** constructor. For example, the following statement is currently invalid: Box ob = new Box();

Since **Box(**) requires three arguments. /\* Here, Box defines three constructors to initialize the dimensions of a box various ways. \*/

```
class Box {
double width:
double height;
double depth;
// constructor used when all dimensions specified
Box(double w, double h, double d) {
width
         =
w; height
= h; depth
=d;
}
// constructor used when no dimensions specified
Box() {
width = -1; // use -1 to
indicate height = -1; // an
uninitialized depth = -1; // box
}
// constructor used when cube is created
Box(double len) {
width = height = depth = len;
}
// compute and return volume
double volume() {
return width * height * depth;
class OverloadCons
public static void main(String args[]) {
// create boxes using the various constructors
Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box();
Box mycube = new
Box(7); double vol;
// get volume of first box
vol = mybox1.volume();
```

System.out.println("Volume of mybox1 is " + vol);

```
// get volume of second box vol =
mybox2.volume(); System.out.println("Volume of
mybox2 is " + vol); //get volume of cube
```

```
vol = mycube.volume();
System.out.println("Volume of mycube is " +
vol);
}
Output:
Volume of mybox1 is 3000.0
Volume of mybox2 is -1.0
Volume of mycube is 343.0
```

# Parameter Passing:

In general, there are two ways that a computer language can pass an argument to a subroutine.

The first way is *call-by-valueI*. In *t*his method copies the *value* of an argument into the formal parameter of the subroutine. Therefore, changes made to the parameter of the subroutine have no effect on the argument.

The second way an argument can be passed is *call-by-reference*. In this method, a reference to an argument (not the value of the argument) is passed to the parameter. Inside the subroutine, this reference is used to access the actual argument specified in the call. This means that changes made to the parameter will affect the argument used to call the subroutine.

Java uses both approaches, depending upon what is passed.

In Java, when you pass a simple type to a method, it is passed by value. Thus, what occurs to the parameter that receives the argument has no effect outside the method. For example, consider the following program:

// Simple types are passed by value.

```
class Test {
void meth(int i, int j)
{ i *= 2;
    j /= 2;
}
class CallByValue
{
public static void main(String args[])
{
Test ob = new
Test(); int a = 15, b =
20;
System.out.println("a and b before call: " + a + " " +
b); ob.meth(a, b);
System.out.println("a and b after call: " + a + " " + b);
```

```
}
}
output :
a and b before call: 15 20
a and b after call: 15 20
we can see, the operations that occur inside meth() have no effect on the values of a and
b used in the call; their values here did not change to 30 and 10.
When we pass an object to a method, the situation changes dramatically, because objects
are passed by reference. Keep in mind that when you create a variable of a class type, you
are only creating a reference to an object. Thus, when you pass this reference to a
method, the parameter that receives it will refer to the same object as that referred to by
the argument. This effectively means that objects are passed to methods by use of call-
by-reference. Changes to the object inside the method do affect the object used as an
argument.
Example:
// Objects are passed by
reference. class Test {
int a, b;
Test(int i, int j) {
a = i;
b = j;
}
// pass an object
void meth(Test o)
o.a *= 2;
o.b /= 2;
class CallByRef
public static void main(String args[])
Test ob = new Test(15, 20);
System.out.println("ob.a and ob.b before call: "+ob.a + " " +
ob.b); ob.meth(ob);
System.out.println("ob.a and ob.b after call: "+ob.a + " " + ob.b);
}
output:
ob.a and ob.b before call: 15 20
ob.a and ob.b after call: 30 10
in this case, the actions inside meth() have affected the object used as an argument.
```

# **Recursion:**

Java supports *recursion*. Recursion is the process of defining something in terms of itself. As it relates to Java programming, recursion is the attribute that allows a method to call itself. A method that calls itself is said to be *recursive*.

The classic example of recursion is the computation of the factorial of a number. The factorial of a number *N* is the product of all the whole numbers between 1 and *N*. For example, 3 factorial is  $1 \times 2 \times 3$ , or 6. Here is how a factorial can be computed by use of a recursive method:

```
// A simple example of recursion.
class Factorial
// this is a recursive function
int fact(int n)
int result:
if(n==1) return 1;
result = fact(n-1) * n;
return result;
}
}
class Recursion {
public static void main(String args[]) { Factorial f
= new Factorial(); System.out.println("Factorial
of 3 is " + f.fact(3));
System.out.println("Factorial of 4 is " +
f.fact(4)); System.out.println("Factorial of 5 is "
+ f.fact(5)); } }
Output:
Factorial of 3 is 6
Factorial of 4 is 24
Factorial of 5 is 120
```

# **String Handling:**

In Java a *string* is a sequence of characters. But, unlike many other languages that implement strings as character arrays, Java implements strings as objects of type **String**. Implementing strings as built-in objects allows Java to provide a full complement of features that make string handling convenient.

when you create a **String** object, you are creating a string that cannot be changed. That is, once a **String** object has been created, you cannot change the characters that comprise that string. The

difference is that each time you need an altered version of an existing string, a new **String** object is created that contains the modifications. The original string is left unchanged. This approach is used because fixed, immutable strings can be implemented more efficiently than changeable ones. For those cases in which a modifiable string is

desired, there is a companion class to **String** called **StringBuffer**, whose objects contain strings that can be modified after they are created.

Both the **String** and **StringBuffer** classes are defined in **java.lang**. Thus, they are available to all programs automatically. Both are declared **final**, which means that neither of these classes may be subclassed.

#### The String Constructors:

The **String** class supports several constructors. To create an empty **String**, you call the default constructor.

For example,

String s = new String();

will create an instance of **String** with no characters in it.Frequently, you will want to create strings that have initial values. The **String** class provides a variety of constructors to handle this. To create a **String** initialized by an array of characters, use the constructor shown here:

String(char chars[ ])

Example: char chars[] = { 'a', 'b', 'c' }; String s = new String(chars); This constructor initializes s with the string —abc||.

You can specify a subrange of a character array as an initializer using the following constructor: String(char *chars*[], int *startIndex*, int *numChars*) Here, *startIndex* specifies the index at which the subrange begins, and *numChars* specifies the number of characters to use. Example:

char chars[] = { 'a', 'b', 'c', 'd', 'e', 'f' }; String s = new String(chars, 2, 3); This initializes s with the characters cde.

You can construct a **String** object that contains the same character sequence as another **String** object using this constructor: String(String *strObj*)

Here, *strObj* is a **String** object.

#### String Length:

The length of a string is the number of characters that it contains. To obtain this value, call the **length()** method. int length()

## **Example:**

char chars[] = { 'a', 'b', 'c' }; String s = new String(chars); System.out.println(s.length()); It prints 3 as the output since the string as 3 characters.

## charAt( )

To extract a single character from a **String**, you can refer directly to an individual character via the **charAt(**) method. It has this general form: char charAt(int *where*)

## getChars()

If you need to extract more than one character at a time, you can use the **getChars()** method. It has this general form:

void getChars(int sourceStart, int sourceEnd, char target[], int targetStart)

#### equals()

To compare two strings for equality, use **equals()**. It has this general form: boolean equals(Object *str*)

Here, *str* is the **String** object being compared with the invoking **String** object. It returns **true** if the strings contain the same characters in the same order, and **false** otherwise. The comparison is case-sensitive.

To perform a comparison that ignores case differences, call **equalsIgnoreCase(**). When it compares two strings, it considers **A-Z** to be the same as **a-z**. It has this general form: boolean equalsIgnoreCase(String *str*)

Here, *str* is the **String** object being compared with the invoking **String** object.

## compareTo()

to know whether two strings are identical. For sorting applications, you need to know which is *less than, equal to,* or *greater than* the next. A string is less than another if it comes before the other in dictionary order. A string is

greater than another if it comes after the other in dictionary order. The **String** method **compareTo()** serves this purpose. It has this general form: int compareTo(String *str*)

Here, *str* is the **String** being compared with the invoking **String**.

## indexOf()

The String class provides two methods that allow you to search a string for a specified character or substring:

- **indexOf(**) Searches for the first occurrence of a character or substring.
- **lastIndexOf()** Searches for the last occurrence of a character or substring.

## substring( )

>>we can extract a substring using **substring()**. It has two forms. The first is String substring(int *startIndex*)

>>Here, *startIndex* specifies the index at which the substring will begin. This form returns a copy of the substring that begins at *startIndex* and runs to the end of the invoking string.

>>The second form of **substring(**) allows you to specify both the beginning and ending index of the substring:

## >>String substring(int *startIndex*, int *endIndex*)

Here, *startIndex* specifies the beginning index, and *endIndex* specifies the stopping point. The string returned contains all the characters from the beginning index, up to,but not including, the ending index.

## replace()

The **replace(**) method replaces all occurrences of one character in the invoking string with another character. It has the following general form: String replace(char *original*, char *replacement*)

## **StringBuffer Constructors:**

**StringBuffer** defines these three constructors: StringBuffer() StringBuffer(int *size*) StringBuffer(String *str*) The default constructor (the one with no param

- The default constructor (the one with no parameters) reserves room for 16 characters without reallocation.
- The second version accepts an integer argument that explicitly sets the size of the buffer. The third version accepts a **String** argument that sets the initial contents of the **StringBuffer** object and reserves room for 16 more characters without reallocation.
- **StringBuffer** allocates room for 16 additional characters when no specific buffer length is requested, because reallocation is a costly process in terms of time. Also, frequent reallocations can fragment memory. By allocating room for a few extra characters, **StringBuffer** reduces the number of reallocations that take place.

# length( ) and capacity( )

The current length of a **StringBuffer** can be found via the **length()** method, while the total allocated capacity can be found through the **capacity()** method. They have the following general forms: int length()

int length( )

#### int capacity()

## ensureCapacity( )

If you want to preallocate room for a certain number of characters after a **StringBuffer** has been constructed, you can use **ensureCapacity()** to set the size of the buffer. **ensureCapacity()** has this general form: void ensureCapacity(int *capacity*) Here, *capacity* specifies the size of the buffer

## append()

The **append()** method concatenates the string representation of any other type of data to the end of the invoking **StringBuffer** object. It has overloaded versions for all the built-in types and for **Object**. Here are a few of its forms: StringBuffer append(String *str*)

StringBuffer append(int num)
StringBuffer append(Object obj)
String.valueOf() is called for each parameter to obtain its string representation.

## insert( )

The **insert(**) method inserts one string into another. StringBuffer insert(int *index*, String *str*) StringBuffer insert(int *index*, char *ch*) StringBuffer insert(int *index*, Object *obj*)

Here, *index* specifies the index at which point the string will be inserted into the invoking **StringBuffer** object

## reverse()

You can reverse the characters within a **StringBuffer** object using **reverse()**, shown here:

StringBuffer reverse( )

This method returns the reversed object on which it was called

# delete( ) and deleteCharAt( )

to delete characters using the methods **delete(**) and **deleteCharAt(**). These methods are shown here:

StringBuffer delete(int *startIndex*, int *endIndex*) StringBuffer deleteCharAt(int *loc*)

The **delete(**) method deletes a sequence of characters from the invoking object. Here, *startIndex* specifies the index of the first character to remove, and *endIndex* specifies an

index one past the last character to remove. Thus, the substring deleted runs from *startIndex* to *endIndex*–1. The resulting **StringBuffer** object is returned. The **deleteCharAt()** method deletes the character at the index specified by *loc*. It returns the resulting **StringBuffer** object

#### replace()

to replaces one set of characters with another set inside a **StringBuffer** object. Its signature is shown here:

StringBuffer replace(int *startIndex*, int *endIndex*, String *str*)

The substring being replaced is specified by the indexes *startIndex* and *endIndex*. Thus, the substring at *startIndex* through *endIndex*-1 is replaced.

## **Hierarchical Abstractions**

Example:



- Abstraction consists of eliminating unnecessary detail and concentrating on essential features
- The concept of an evergreen tree is an abstraction
- Fir trees, Thuya trees, Pine trees, ... have features that are common to all evergreen trees

- The concept of a deciduous tree is an abstraction
- Maple trees, Oak trees, Apple trees, ... have features that are common to all deciduous trees

# **INHERITANCE:**

Inheritance is a Java language feature, used to model hierarchical abstraction Inheritance means taking a class (called the base class) and defining a new class (called the subclass) by specializing the state and behaviors of the base class Subclasses specialize the behaviors of their base class

- The subclasses inherit the state and behaviors of their base class
- They can have additional state and behaviors Inheritance is mainly used for code reusability and to reduce the complexity of the program.

## Base Class:

```
A class that is being inherited is known as Base class
For ex:
Class x
{
Void method ();
}
class y extends x
{
}
In the above example the class x is known as the base class.
When we create an object for the base class then that object
```

When we create an object for the base class then that object is known as base class object. **Sub class:** 

The class that is inheriting the base class Is known as sub class.

In the example above since class y is inheriting the class x, class y is known as the subclass to class x.

## **Definition of substitutability:**

- Assigning derived class object to parent class reference variables is known as substitutability.
- The concept of substitutability is fundamental to many of the powerful software development techniques in object oriented programming.
- The idea is that the type given in a declaration of variable doesn't have to match the type associated with value the variable is holding.
- It can also be used through interfaces.

## FORMS OF INHERITANCE:

The choices between inheritance and overriding, subclass and subtypes, mean that inheritance can be used in a variety of different ways and for different purposes. Many of these types of inheritance are given their own special names. We will describe some of these specialized forms of inheritance.

- Specialization
- Specification
- Construction
- Generalization or Extension
- Limitation
- Variance

**Specialization Inheritance:** By far the most common form of inheritance is for specialization . A good example is the Java hierarchy of Graphical components in the AWT:

- Component
- Label
- Button
- TextComponent
- TextArea
- TextField
- CheckBox
- Scroll bar

Each child class overrides a method inherited from the parent in order to specialize the class in some way.

# **Specification Inheritance:**

• If the parent class is abstract, we often say that it is providing a specification for the child class, and therefore it is specification inheritance (a variety of specialization inheritance). Example: Java Event Listeners, ActionListener, MouseListener, and so on specify behavior, but must be subclassed.

# **Inheritance for Construction:**

• If the parent class is used as a source for behavior, but the child class has no is-a relationship to the parent, then we say the child class is using inheritance for construction. An example might be subclassing the idea of a Set from an existing List class.

• Generally not a good idea, since it can break the principle of substituability, but nevertheless sometimes found in practice. (More often in dynamically typed languages, such as Smalltalk).

# **Inheritance for Generalization or Extension:**

- If a child class generalizes or extends the parent class by providing more functionality, and overrides any method, we call it inheritance for generalization.
- The child class doesn't change anything inherited from the parent, it simply adds new features is called extension.

An example is Java Properties inheriting form Hashtable.

# Inheritance for Limitation:

- If a child class overrides a method inherited from the parent in a way that makes it unusable (for example, issues an error message), then we call it inheritance for limitation. For example, you have an existing List data type that allows items to be inserted at either end, and you override methods allowing insertion at one end in order to create a Stack.
- Generally not a good idea, since it breaks the idea of substitution. But again, it is sometimes found in practice.

# **Inheritance for Variance:**

• Two or more classes that seem to be related, but its not clear who should be the parent and who should be the child.

# Example: Mouse and TouchPad and JoyStick

• Better solution, abstract out common parts to new parent class, and use subclassing for specialization.

# Summary of Forms of Inheritance:

**Specialization:** The child class is a special case of the parent class; in other words, the child class is a subtype of the parent class.

**Specification:** The parent class defines behavior that is implemented in the child class but not in the parent class.

**<u>Construction</u>**: The child class makes use of the behavior provided by the parent class, but is not a subtype of the parent class.

**<u>Generalization</u>**: The child class modifies or overrides some of the methods of the parent class.

**Extension:** The child class adds new functionality to the parent class, but does not change any inherited behavior.

**<u>Limitation</u>**: The child class restricts the use of some of the behavior inherited from the parent class.

**Variance:** The child class and parent class are variants of each other, and the class-subclass relationship is arbitrary.

**<u>Combination</u>**: The child class inherits features from more than one parent class. This is multiple inheritance and will be the subject of a later chapter.

# **BENEFITS OF INHERITANCE:**

- Software Reuse
- Code Sharing
- Improved Reliability
- Consistency of Interface
- Rapid Prototyping
- Polymorphism
- Information Hiding

# **COSTS OF INHERITANCE:**

- Execution speed
- Program size
- Message Passing Overhead
- Program Complexity

This does not mean you should not use inheritance, but rather than you must understand the benefits, and weigh the benefits against the costs.

# **TYPES OF INHERITANCE:**

• <u>Single Level Inheritance:</u>

When one base class is being inherited by one sub class then that kind of inheritance is known as single level inheritance.

# <u>Multi Level Inheritance:</u>

When a sub class is in turn being inherited then that kind of inheritance is known as multi level inheritance.

## • <u>Hierarchical Inheritance:</u>

When a base class is being inherited by one or more sub class then that kind of inheritance is known as hierarchical inheritance.

A sub class uses the keyword —extends|| to inherit a base class.

#### **Example for Single Inheritance:**

```
class x
ł
    int a;
   void display()
   {
       a=0;
       System.out.println(a);
   }
}
class y extends x
{
      int b:
       void show()
     {
         B=1;
        System.out.println(b);
     }
}
class show_main
{
   Public static void main(String args[])
   ł
       y y1=new y();
       y1.display();
       y1.show();
   }
}
Output:
0
1
```

Since the class y is inheriting class x, it is able to access the members of class x. Hence the method display() can be invoked by the instance of the class y.

## **Example for multilevel inheritance:**

```
class x
{
```

```
int a:
 void display()
  {
      a=0;
      System.out.println(a);
  }
}
class y extends x
{
   int b;
   void show()
   {
        B=1;
       System.out.println(b);
   }
}
class z extends y
ł
 int c;
 show1()
 {
       c=2;
      System.out.println(c);
  }
}
class show_main
{
  Public static void main(String args[])
  {
       z z1=new z();
        z1.display();
    z1.show();
  }
}
```

# <u>Output</u>

01

2

Since class z is inheriting class y which is in turn a sub class of the class x, indirectly z can access the members of class x.

Hence the instance of class z can access the display () method in class x, the show () method in class y.

# Problems:

In java multiple level inheritance is not possible easily. We have to make use of a concept called interfaces to achieve it.

## Access Specifers:

The different access specifiers used are

- Public
- Private
- Protected
- Default
- Privateprotected

1. Private members can be accessed only within the class in which they are declared.

2. Protected members can be accessed inside the class in which they are declared and also in the sub classes in the same package and sub classes in the other packages.

3. Default members are accessed by the methods in their own class and in the sub classes of the same package.

4. Public members can be accessed anywhere.

# Super Keyword:

Whenever a sub class needs to refer to its immediate super class, we can use the super keyword.

Super has two general forms

- The first calls the super class constructor.
- The second is used to access a member of the super class that has been hidden by a member of a sub class

# Syntax:

A sub class can call a constructor defined by its super class by use of the following form of super.

super(arg-list);

here arg-list specifies any arguments needed by the constructor in the super class .

The second form of super acts like a —this|| keyword. The difference between —this|| and —super|| is that —this|| is used to refer the current object where as the super is used to refer to the super class.

The usage has the following general form: super.member;

```
Example:
Class x
```

ł

```
int
a; x()
{
a=0;
}
void display()
```

```
{
    System.out.println(a);
  }
}
class y extends x
{
    int b;
  y()
  {
   super();
   b=1;
  }
 Void display()
   Super.display();
   System.out.println(b);
  }
}
class super_main
 Public static void main(String args[])
  {
     y y1=new y();
     y.display();
  }
}
```

# **Using final with inheritance:**

The keyword final has three uses.

- First it can be used to create the equivalent of a named constant.
- To prevent overriding.
- To prevent inheritance.

# Using final to prevent overriding:

To disallow a method from being overridden, specify final as a modifier at the start of the declaration.

Methods declared as final cannot be overridden.

Syntax:

final <return type> <method name> (argument list);

# **Using final with inheritance:**

Some times we may want to prevent a class from being inherited. In order to do this we must precede the class declaration with final. Declaring a class as final implicitly declares all its methods as final.

# Example:

```
final class A
{
.....//members
}
```

## Abstract Classes:

Abstract methods:

We can require that some methods be overridden by sub classes by specifying the abstract type modifier.

These methods are sometimes referred to as sub classer responsibility as they have no implementation specified in the super class.

Thus a sub class must override them.

To declare an abstract method we have:

abstract type name (parameter list);

Any class that contains one or more abstract methods must also be declared abstract..

Such types of classes are known as abstract classes.

Abstract classes can contain both abstract and non-abstract methods.

Let us consider the following example:

abstract class A

```
abstract void callme();
void call()
```

```
System.out.println(—HELLO||);
```

}

class B extends A

Void callme()

```
System.out.println(—GOOD MORNING||);
```

```
}
```

class abstractdemo

```
{
  Public static void main(String args[]){
    B b=new B();
    b.callme();
```

```
b.call();
```

```
}
```

}

Output: GOOD MORINING HELLO

# **UNIT- II** Packages and Interfaces

Packages and interfaces are two of the basic components of a Java program. In general, a Java source file can contain any (or all) of the following four internal parts:

- A single package statement (optional)
- Any number of import statements (optional)
- A single public class declaration (required)
- Any number of classes private to the package (optional)

Java provides a mechanism for partitioning the class name space into more manageable chunks. This mechanism is the package. The package is both a naming and a visibility control mechanism. You can define classes inside a package that are not accessible by code outside that package. You can also define class members that are only exposed to other members of the same package.

## **Defining a Package:**

- Creating 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.
- If you omit the **package** statement, the class names are put into the default package, which has no name. While the default package is fine for short, sample programs, it is inadequate for real applications. Most of the time, you will define a package for your code.

## **Creating a Package:**

- 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**;

Java uses file system directories to store packages. For example, the **.class** files for any classes you declare to be part of **MyPackage** must be stored in a directory called **MyPackage**. Remember that case is significant, and the directory name must match the package name exactly.

More than one file can include the same **package** statement. The **package** statement

simply specifies to which package the classes defined in a file belong. It does not exclude other classes in other files from being part of that same package. 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. For example, a package declared as **package java.awt.image;** needs to be stored in **java/awt/image**, **java\awt\image**,

Or

## java:awt:imag

e on your

**UNIX, Windows, or Macintosh** file system, respectively. Be sure to choose your package names carefully.

• You cannot rename a package without renaming the directory in which the classes are stored.

# Finding Packages and CLASSPATH:

Java run-time system know where to look for packages that you create? The answer has two parts.

- First, by default, the Java run-time system uses the current working directory as its starting point. Thus, if your package is in the current directory, or a subdirectory of the current directory, it will be found.
- Second, you can specify a directory path or paths by setting the **CLASSPATH** environmental variable.

For example, consider the following package specification.

# package MyPack;

In order for a program to find **MyPack**, one of two things must be true. Either the program is executed from a directory immediately above **MyPack**, or **CLASSPATH** must be set to include the path to **MyPack**.

The first alternative is the easiest (and doesn't require a change to **CLASSPATH**), but the second alternative lets your program find **MyPack** no matter what directory the program is in.

Create the package directories below your current development directory, put the **.class** files into the appropriate directories and then execute the programs from the development directory.

```
Example:
// A simple package
package MyPack;
class Balance
       String name;
       double bal;
       Balance(String n, double b)
       name = n;
       bal = b;
       void show()
       if(bal<0)
       System.out.print("--> ");
System.out.println(name + ": $" + bal);
}
class AccountBalance
public static void main(String args[])
       Balance current[] = new Balance[3];
       current[0] = new Balance("K. J. Fielding", 123.23);
       current[1] = new Balance("Will Tell", 157.02);
       current[2] = new Balance("Tom Jackson", -12.33);
       for(int i=0; i<3; i++)
       current[i].show();
}
```

Save this file as **AccountBalance.java**, and put it in a directory called **MyPack**. Next, compile the file. Make sure that the resulting **.class** file is also in the **MyPack** directory. Executing the **AccountBalance** class, using the following command line: **java MyPack.AccountBalance** 

Remember, we should be in the directory above **MyPack** when you execute this command, or to have your **CLASSPATH** environmental variable set appropriately. **AccountBalance** is now part of the package **MyPack**. This means that it cannot be executed by itself. That is, you cannot use this command line:

## java AccountBalance

AccountBalance must be qualified with its package name.

## **Access Protection:**

Classes and packages are both means of encapsulating and containing the name space and scope of variables and methods. Packages act as containers for classes 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

	Private	No modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non- subclass	No	Yes	Yes	Yes
Different Package subclass	No	No	Yes	Yes
Different package non- subclass	No	No	No	Yes

#### Table: Class member access

The three access specifiers, **private**, **public**, and **protected**, provide a variety of ways to produce the many levels of access required by these categories. A class has only two possible access levels: default and public. When a class is declared as **public**, it is accessible by any other code. If a class has default access, then it can only be accessed by other code within its same package.

```
Example:
   This is file
•
   Protection.java: package
   p1;
   public class Protection
    {
           int n = 1;
           private int n_{pri} = 2;
           protected int n_pro =
           3; public int n_pub =
           4; public Protection()
           System.out.println("base constructor");
           System.out.println("n = " + n);
           System.out.println("n_pri = " + n_pri);
           System.out.println("n_pro = " + n_pro);
           System.out.println("n_pub = " + n_pub);
   }
   This is file
   Derived.java: package
   p1;
   class Derived extends Protection
   {
           Derived()
           {
           System.out.println("derived constructor");
           System.out.println("n = " + n);
           // class only
          // System.out.println("n_pri = " + n_pri);
           System.out.println("n_pro = " + n_pro);
           System.out.println("n_pub = " + n_pub);
           }
   }
```

```
This is file
   SamePackage.java: package
   p1;
   class SamePackage
   SamePackage()
           Protection p = new Protection();
          System.out.println("same package constructor");
           System.out.println("n = " + p.n);
          // class only
          // System.out.println("n_pri = " + p.n_pri);
          System.out.println("n_pro = " + p.n_pro);
          System.out.println("n_pub = " + p.n_pub);
   }
   This is file Protection2.java:
   package p2;
   class Protection2 extends p1.Protection
   {
           Protection2()
           ł
          System.out.println("derived other package constructor");
          // class or package only
          // System.out.println("n = " + n);
          // class only
          // System.out.println("n_pri = " + n_pri);
          System.out.println("n_pro = " + n_pro);
           System.out.println("n_pub = " + n_pub);
   }
•
   This is file OtherPackage.java:
   package p2;
   class OtherPackage
   {
           OtherPackage()
           p1.Protection p = new p1.Protection();
          System.out.println("other package constructor");
          // class or package only
          // System.out.println("n = " + p.n);
          // class only
          // System.out.println("n_pri = " + p.n_pri);
          // class, subclass or package only
```

```
// System.out.println("n_pro = " +
p.n_pro); System.out.println("n_pub = " +
p.n_pub);
}
```

}

• If you wish to try these two packages, here are two test files you can use. The one for package **p1** is shown here:

# **Importing Packages:**

Java includes the **import** statement to bring certain classes, or entire packages, into visibility. Once imported, a class can be referred to directly, using only its name. The **import** statement is convenient.

In a Java source file, **import** statements occur immediately following the **package** statement (if it exists) and before any class definitions. This is the general form of the **import** statement:

```
import pkg1[.pkg2].(classname|*);
```

Here, pkg1 is the name of a top-level package, and pkg2 is the name of a subordinate package inside the outer package separated by a dot (.). There is no practical

limit on the depth of a package hierarchy, except that imposed by the file system. Finally, you specify either an explicit *classname* or a star (\*), which indicates that the Java compiler should import the entire package. This code fragment shows both forms in use: **import java.util.Date; import java.io.\*;** 

The star form may increase compilation time—especially if you import several large packages. For this reason it is a good idea to explicitly name the classes that you want to use rather than importing whole packages. However, the star form has absolutely no effect on the run-time performance or size of your classes.

All of the standard Java classes included with Java are stored in a package called **java**. The basic language functions are stored in a package inside of the **java** package called **java.lang**. Normally, you have to import every package or class that you want to use.

If a class with the same name exists in two different packages that you import using the star form, the compiler will remain silent, unless you try to use one of the classes. In that case, you will get a compile-time error and have to explicitly name the class specifying its package.

when a package is imported, only those items within the package declared as **public** will be available to non-subclasses in the importing code.

For example, if you want the **Balance** class of the package **MyPack** shown earlier to be available as a stand-alone class for general use outside of **MyPack**, then you will need to declare it as **public** and put it into its own file, as shown here: package MyPack;

/\* Now, the Balance class, its constructor, and its show() method are public. This means that they can be used by non-subclass code outside their package.\*/ public class Balance

```
{
```

}

```
String name;
double bal;
public Balance(String n, double b)
{
  name = n;
  bal = b;
  }
  public void show()
  {
  if(bal<0)
  System.out.print("--> ");
  System.out.println(name + ": $" + bal);
  }
```

As you can see, the **Balance** class is now **public**. Also, its constructor and its **show()** method are **public**, too. This means that they can be accessed by any type of code outside the **MyPack** package. For example, here **TestBalance** imports **MyPack** and is then able to make use of the **Balance** class:

```
import MyPack.*;
class TestBalance
{
```

public static void main(String args[])

/\* Because Balance is public, you may use Balance class and call its constructor. \*/

```
Balance test = new Balance("J. J. Jaspers", 99.88);
test.show(); // you may also call show() }
```

}

## Interfaces

Using the keyword interface, you can fully abstract a class' interface from its implementation. That is, using interface, you can specify what a class must do, but not how it does it. Interfaces are syntactically similar to classes, but they lack instance varia

bles, and their methods are declared without any body. In practice, this means that you can define interfaces which don't make assumptions about how they are implemented. Once it is defined, any number of classes can implement an interface. Also, one class can implement any number of interfaces.

To implement an interface, a class must create the complete set of methods defined by the interface. Each class can determine the details of its own implementation. By providing the interface keyword, Java allows you to fully utilize the —one interface, multiple methods|| aspect of polymorphism.

Interfaces are designed to support dynamic method resolution at run time. Normally, in order for a method to be called from one class to another, both classes need to be present at compile time so the Java compiler can check to ensure that the method signatures are compatible. This requirement by itself makes for a static and no extensible classing environment. Inevitably in a system like this, functionality gets pushed up higher and higher in the class hierarchy so that the mechanisms will be available to more and more subclasses. Interfaces are designed to avoid this problem. They disconnect the definition of a method or set of methods from the inheritance hierarchy. Since interfaces are in a different hierarchy from classes, it is possible for classes that are unrelated in terms of the class hierarchy to implement the same interface. This is where the real power of interfaces is realized.

Interfaces add most of the functionality that is required for many applications which would normally resort to using multiple inheritance in a language such as C++.

#### **Defining an Interface:**

- An interface is defined much like a class.
- This is the general form of an interface: access interface name
  - {

return-type method-name1(parameterlist); return-type methodname2(parameter-list); type finalvarname1 = value; type final-varname2 = value; return-type method-nameN(parameter-list); type finalvarnameN = value;

}

Here, access is either public or not used. When no access specifier is included, then default access results, and the interface is only available to other members of the package in which it is declared. When it is declared as public, the interface can be used by any other code. name is the name of the interface, and can be any valid identifier. Notice that the methods which are declared have no bodies. They end with a semicolon after the parameter list. They are, essentially, abstract methods; there can be no default implementation of any method specified within an interface. Each class that includes an interface must implement all of the methods.

Variables can be declared inside of interface declarations. They are implicitly final and static, meaning they cannot be changed by the implementing class. They must also be initialized with a constant value. All methods and variables are implicitly public if the interface, itself, is declared as public.

An example of an interface definition. It declares a simple interface which contains one method called **callback()** that takes a single integer parameter.

## **Implementing Interfaces:**

Once an interface has been defined, one or more classes can implement that interface.

To implement an interface, include the implements clause in a class definition, and then create the methods defined by the interface.

• The general form of a class that implements the interface:

access class classname [extends superclass][implements interface [,interface...]] { // class-body }

Here, access is either public or not used. If a class implements more than one interface, the interfaces are separated with a comma. If a class implements two interfaces that declare the same method, then the same method will be used by clients of either interface. The methods that implement an interface must be declared public. Also, the type signature of the implementing method must match exactly the type signature specified in the interface definition.

**Example**: class that implements the Callback interface shown earlier.

class Client implements Callback

{

public void callback(int p)
{
 System.out.println("callback called with " + p);

}

}

Notice that callback() is declared using the public access specifier. When you implement an interface method, it must be declared as public.

For example, the following version of Client implements callback() and adds the method nonIfaceMeth():

class Client implements Callback

public void callback(int p)
{
 System.out.println("callback called with " + p);
}
void nonIfaceMeth()
{

System.out.println("Classes that implement interfaces"+"may also define other members, too.");

ì

#### Accessing Implementations Through Interface References:

We can declare variables as object references that use an interface rather than a class type. Any instance of any class that implements the declared interface can be referred to by such a variable. When you call a method through one of these references, the correct version will be called based on the actual instance of the interface being referred to. This is one of the key features of interfaces. The method to be executed is looked up dynamically at run time, allowing classes to be created later than the code which calls methods on them. The calling code can dispatch through an interface without having to know anything about the —callee. || This process is similar to using a superclass reference to access a subclass object.

Because dynamic lookup of a method at run time incurs a significant overhead when compared with the normal method invocation in Java, you should be careful not to use interfaces casually in performance-critical code.

The following example calls the callback( ) method via an interface reference variable:

```
class TestIface
```

}

```
public static void main(String args[])
{
Callback c = new Client();
c.callback(42);
}
```

### **Output:**

callback called with 42.

Notice that variable c is declared to be of the interface type Callback, yet it was assigned an instance of Client. Although c can be used to access the callback() method, it cannot access any other members of the Client class. An interface reference variable only has knowledge of the methods declared by its interface declaration.

Thus, c could not be used to access nonIfaceMeth() since it is defined by Client but not Callback.

The preceding example shows, mechanically, how an interface reference variable can access an implementation object, it does not demonstrate the polymorphic power of such a reference. To sample this usage, first create the second implementation of Callback, shown here:

// Another implementation of Callback.

```
class AnotherClient implements Callback
```

```
{
    public void callback(int p)
        {
            System.out.println("Another version of
            callback"); System.out.println("p squared is " +
            (p*p)); }
}
class TestIface2
{
            public static void main(String args[])
            {
                 Callback c = new Client();
                 AnotherClient ob = new AnotherClient();
                c.callback(42);
                 c = ob; // c now refers to AnotherClient object
                c.callback(42);
                 }
            }
}
```

## **Output:**

callback called with 42 Another version of callback p squared is 1764

As you can see, the version of callback() that is called is determined by the type of object that c refers to at run time.

If a class includes an interface but does not fully implement the methods defined by that interface, then that class must be declared as abstract.

```
For example:
```

abstract class Incomplete implements Callback

```
int a, b;
void show()
{
System.out.println(a + " " + b);
}
```

Here, the class Incomplete does not implement callback( ) and must be declared as abstract. Any class that inherits Incomplete must implement callback( ) or be declared abstract itself.

# **Applying Interfaces:**

We define a stack interface, leaving it to each implementation to define the specifics. Let's look at two examples.

First, here is the interface that defines an integer stack. Put this in a file called IntStack.java. This interface will be used by both stack implementations. Example:

```
// Define an integer stack
```

interface. interface IntStack

{

void push(int item); // store an item
int pop(); // retrieve an item

```
}
```

The following program creates a class called FixedStack that implements a fixedlength version of an integer stack: Example:

```
// An implementation of IntStack that uses fixed storage. class FixedStack implements IntStack
```

```
{
    private int stck[];
    private int tos;
    FixedStack(int size)
    stck = new int[size];
    tos = -1;
}
// Push an item onto the
stack public void push(int
item)
{
    if(tos==stck.length-1) // use length member
}
```

```
System.out.println("Stack is full.");
       else
       stck[++tos] = item;
}
// Pop an item from the stack
public int pop()
       if(tos < 0)
       System.out.println("Stack underflow.");
       return 0;
        }
       else
       return stck[tos--];
        }
}
class IFTest
{
       public static void main(String args[])
       FixedStack mystack1 = new FixedStack(5);
       // push some numbers onto the stack
       for(int i=0; i<5; i++) mystack1.push(i);</pre>
       for(int i=0; i<8; i++) mystack2.push(i);</pre>
       // pop those numbers off the stack
       System.out.println("Stack in
       mystack1:"); for(int i=0; i<5; i++)
       System.out.println(mystack1.pop());
       System.out.println("Stack in
       mystack2:"); for(int i=0; i<8; i++)
       System.out.println(mystack2.pop());
}
```

Following is another implementation of IntStack that creates a dynamic stack by use of the same interface definition. In this implementation, each stack is constructed with an initial length. If this initial length is exceeded, then the stack is increased in size. Each time more room is needed, the size of the stack is doubled. Example:

```
// Implement a "growable" stack.
class DynStack implements IntStack
ł
private int stck[];
private int tos;
// allocate and initialize stack
DynStack(int size)
{
stck = new int[size];
tos = -1;
}
// Push an item onto the
stack public void push(int
item)
{
// if stack is full, allocate a larger
stack if(tos==stck.length-1)
int temp[] = new int[stck.length * 2]; // double size
for(int i=0; i<stck.length; i++)</pre>
temp[i] = stck[i];
stck = temp;
stck[++tos] =
item;
}
else
stck[++tos] = item;
}
// Pop an item from the stack
public int pop()
if(tos < 0) {
System.out.println("Stack underflow.");
return 0;
}
else
return stck[tos--];
}
class IFTest2
public static void main(String args[])
DynStack mystack1 = new DynStack(5);
DynStack mystack2 = new DynStack(8);
// these loops cause each stack to grow
for(int i=0; i<12; i++) mystack1.push(i);</pre>
for(int i=0; i<20; i++) mystack2.push(i);</pre>
```
System.out.println("Stack in
mystack1:");

```
for(int i=0; i<12; i++)
System.out.println(mystack1.pop());
System.out.println("Stack in mystack2:");
for(int i=0; i<20; i++)
System.out.println(mystack2.pop());
The following class uses both the FixedStack and DynStack implementations. It does so
through an interface reference. This means that calls to push() and pop() are resolved at
run time rather than at compile time.
/* Create an interface variable and access stacks through it.
*/
class IFTest3
public static void main(String args[])
IntStack mystack; // create an interface reference
variable DynStack ds = new DynStack(5); FixedStack fs
= new FixedStack(8);
mystack = ds; // load dynamic stack
// push some numbers onto the stack
for(int i=0; i<12; i++)
mystack.push(i);
mystack = fs; // load fixed
stack for(int i=0; i<8; i++)
mystack.push(i);
mystack = ds;
System.out.println("Values in dynamic stack:");
for(int i=0; i<12; i++)
System.out.println(mystack.pop());
mystack = fs;
System.out.println("Values in fixed
stack:"); for(int i=0; i<8; i++)
System.out.println(mystack.pop());
}
}
```

In this program, mystack is a reference to the IntStack interface. Thus, when it refers to ds, it uses the versions of push() and pop() defined by the DynStack implementation. When it refers to fs, it uses the versions of push() and pop() defined by FixedStack.

These determinations are made at run time. Accessing multiple implementations of an interface through an interface reference variable is the most powerful way that Java achieves run-time polymorphism.

#### Variables in Interfaces:

Interfaces can be used to import shared constants into multiple classes by simply declaring an interface that contains variables which are initialized to the desired values. When we include that interface in a class (that is, when you —implement|| the interface), all of those variable names will be in scope as constants. This is similar to using a header file in C/C++ to create a large number of #defined constants or const declarations. If an interface contains no methods, then any class that includes such an interface doesn't actually implement anything. It is as if that class were importing the constant variables into the class name space as final variables

```
Example:
import java.util.Random;
interface SharedConstants
{
int NO = 0;
int YES = 1:
int MAYBE = 2;
int LATER = 3;
int SOON = 4;
int NEVER = 5;
}
class Question implements SharedConstants
Random rand = new Random();
int ask()
{
int prob = (int) (100 * rand.nextDouble());
if (\text{prob} < 30)
return NO; // 30%
else if (\text{prob} < 60)
return YES; // 30%
else if (\text{prob} < 75)
return LATER; // 15%
else if (prob < 98)
return SOON; // 13%
else
return NEVER; // 2%
class AskMe implements SharedConstants
static void answer(int result)
switch(result)
{
case NO:
```

```
System.out.println("No");
break:
case YES:
System.out.println("Yes");
break;
case MAYBE:
System.out.println("Maybe");
break;
case LATER:
System.out.println("Later");
break:
case SOON:
System.out.println("Soon");
break;
case NEVER:
System.out.println("Never");
break;
public static void main(String args[])
Question q = new Question();
answer(q.ask());
answer(q.ask());
answer(q.ask());
answer(q.ask());
}
}
Output:
Later
Soon
No
Yes
```

This program makes use of one of Java's standard classes: Random. This class provides pseudorandom numbers. It contains several methods which allow you to obtain random numbers in the form required by your program. In this example, the method nextDouble() is used. It returns random numbers in the range 0.0 to 1.0. In this sample program, the two classes, Question and AskMe, both implement the SharedConstants interface where NO, YES, MAYBE, SOON, LATER, and NEVER are defined. Inside each class, the code refers to these constants as if each class had defined or inherited them directly.

#### **Interfaces Can Be Extended:**

```
One interface can inherit another by use of the keyword extends. The syntax is the
same as for inheriting classes. When a class implements an interface that inherits another
interface, it must provide implementations for all methods defined within the interface
inheritance chain.
Example:
// One interface can extend
another. interface A
{
void meth1();
void meth2();
}
// B now includes meth1() and meth2() -- it adds
meth3(). interface B extends A
ł
void meth3();
}
// this class must implement all of A and B
class MyClass implements B
public void meth1()
System.out.println("Implement meth1().");
public void meth2()
System.out.println("Implement meth2().");
public void meth3()
System.out.println("Implement meth3().");
class IFExtend
public static void main(String arg[])
MyClass ob = new MyClass();
ob.meth1();
ob.meth2();
ob.meth3();
Any class that implements an interface must implement all methods defined by that
```

Any class that implements an interface must implement all methods defined by tha interface, including any that are inherited from other interfaces.

#### EXPLORING JAVA.IO PACKAGE:

**java.io**, which provides support for I/O operations Data is retrieved from an *input* source. The results of a program are sent to an *output* destination. In Java, these sources or destinations are defined very broadly. For example, a network connection, memory buffer, or disk file can be manipulated by the Java I/O classes. Although physically different, these devices are all handled by the same abstraction: the *stream*. A stream is a logical entity that either produces or consumes information. A stream is linked to a physical device by the Java I/O system. All streams behave in the same manner, even if the actual physical devices they are linked to differ.

Some of the I/O classes defined by **java.io** are:

FileWriter, BufferedOutputStream, FilterInputStream, BufferedReader, FilterOutputStream, BufferedWriter, FilterReader, DataInputStream, RandomAccessFile DataOutputStream.

#### File:

most of the classes defined by **java.io** operate on streams, the **File** class does not. It deals directly with files and the file system. That is, the **File** class does not specify how information is retrieved from or stored in files; it describes the properties of a file itself. A **File** object is used to obtain or manipulate the information associated with a disk file, such as the permissions, time, date, and directory path, and to navigate subdirectory hierarchies.

Files are a primary source and destination for data within many programs. Files are still a central resource for storing persistent and shared information. A directory in Java is treated simply as a **File** with one additional property—a list of filenames that can be examined by the **list()** method.

The following constructors can be used to create **File** objects:

File(String *directoryPath*)
File(String *directoryPath*, String *filename*)
File(File *dirObj*, String *filename*)
File(URI *uriObj*)
Here, *directoryPath* is the path name of the file, *filename* is the name of the file, *dirObj* is a File object that specifies a directory, and *uriObj* is a URI object that describes a file.

The following example demonstrates several of the **File** methods: // Demonstrate File. import java.io.File; class FileDemo

```
{
static void p(String s)
     System.out.println(s);
public static void main(String args[])
File f1 = new File("/java/COPYRIGHT");
p("File Name: " + f1.getName());
p("Path: " + f1.getPath());
p("Abs Path: " + f1.getAbsolutePath());
p("Parent: " + f1.getParent());
p(f1.exists() ? "exists" : "does not exist");
p(f1.canWrite() ? "is writeable" : "is not writeable");
p(f1.canRead() ? "is readable" : "is not readable");
p("is " + (f1.isDirectory() ? "" : "not" + " a directory"));
p(f1.isFile() ? "is normal file" : "might be a named pipe");
p(f1.isAbsolute() ? "is absolute" : "is not absolute");
p("File last modified: " + f1.lastModified());
p("File size: " + f1.length() + " Bytes");
}
When you run this program, you will see something similar to the following:
```

When you run this program, you will see something similar to the following File Name: COPYRIGHT Path: /java/COPYRIGHT Parent: /java exists is writeable is readable is not a directory is normal file is absolute File last modified: 812465204000 File size: 695 Bytes

#### The Stream Classes:

Java's stream-based I/O is built upon four abstract classes: **InputStream**, **OutputStream**, **Reader**, and **Writer**. They are used to create several concrete stream

subclasses. Although your programs perform their I/O operations through concrete subclasses, the top-level classes define the basic functionality common to all stream classes.

InputStream and OutputStream are designed for byte streams.

Reader and Writer are designed for character streams.

The byte stream classes and the character stream classes form separate hierarchies. In general, you should use the character stream classes when working with characters or strings, and use the byte stream classes when working with bytes or other binary objects.

#### **Byte Stream Classes:**

## InputStream

**InputStream** is an abstract class that defines Java's model of streaming byte input. All of the methods in this class will throw an **IOException** on error conditions. Some of the methods in this class are:

- int available(): Returns the number of bytes of input currently available for reading.
- void close(): Closes the input source. Further read attempts will generate an IOException.
- int read(): Returns an integer representation of the next available byte of input. -1 is returned when the end of the file is encountered.
- int read(byte *buffer*[]): Attempts to read up to *buffer.length* bytes into *buffer* and returns the actual number of bytes that were successfully read. -1 is returned when the end of the file is encountered.

## • <u>OutputStream</u>

**OutputStream** is an abstract class that defines streaming byte output. All of the methods in this class return a **void** value and throw an **IOException** in the case of errors. Some of the methods in this class are:

- void close(): Closes the output stream. Further write attempts will generate an IOException.
- void write(int *b*): Writes a single byte to an output stream. Note that the parameter is an **int**, which allows you to call **write**() with expressions without having to cast them back to **byte**.

• void write(byte *buffer*[]): Writes a complete array of bytes to an output stream.

## <u>FileInputStream</u>

The **FileInputStream** class creates an **InputStream** that you can use to read bytes from a file. Its two most common constructors are shown here:

FileInputStream(String *filepath*)

FileInputStream(File *fileObj*)

Either can throw a **FileNotFoundException**. Here, *filepath* is the full path name of a file, and *fileObj* is a **File** object that describes the file.

#### <u>FileOutputStream</u>

FileOutputStream creates an OutputStream that you can use to write bytes to a file. Its most commonly used constructors are shown here:

FileOutputStream(String *filePath*) FileOutputStream(File *fileObj*) FileOutputStream(String *filePath*, boolean *append*) FileOutputStream(File *fileObj*, boolean *append*)

They can throw a FileNotFoundException or a SecurityException. Here, *filePath* is the full path name of a file, and *fileObj* is a File object that describes the file. If *append* is true, the file is opened in append mode

## The Character Streams:

While the byte stream classes provide sufficient functionality to handle any type of I/O

operation, they cannot work directly with Unicode characters. Since one of the main purposes of Java is to support the —write once, run anywhere|| philosophy, it was necessary to include direct I/O support for characters. In this section, several of the character I/O classes are discussed. As explained earlier, at the top of the character stream hierarchies are the **Reader** and **Writer** abstract classes.

#### <u>Reader</u>

**Reader** is an abstract class that defines Java's model of streaming character input. All of the methods in this class will throw an **IOException** on error conditions. Table 17-3 provides a synopsis of the methods in **Reader**.

#### <u>Writer</u>

**Writer** is an abstract class that defines streaming character output. All of the methods in this class return a **void** value and throw an **IOException** in the case of errors. Table 17-4 shows a synopsis of the methods in **Writer**.

#### <u>FileReader</u>

The **FileReader** class creates a **Reader** that you can use to read the contents of a file. Its two most commonly used constructors are shown here: FileReader(String *filePath*)

FileReader(File *fileObj*)

## • <u>FileWriter</u>

**FileWriter** creates a **Writer** that you can use to write to a file. Its most commonly used constructors are shown here:

FileWriter(String *filePath*) FileWriter(String *filePath*, boolean *append*) FileWriter(File *fileObj*)

FileWriter(File *fileObj*, boolean *append*)

They can throw an **IOException**. Here, *filePath* is the full path name of a file, and *fileObj* is a **File** object that describes the file. If *append* is **true**, then output is appended to the end of the file.

# **UNIT – III Exception handling – Multithreading**

## Exceptional Handling

An exception is a problem that arises during the execution of a program. An exception can occur for many different reasons, including the following:

- $\Box$  A user has entered invalid data.
- $\Box$  A file that needs to be opened cannot be found.
- □ A network connection has been lost in the middle of communications, or the JVM has run out of memory

"Exceptional Handling is a task to maintain normal flow of the program. For this we should try to catch the exception object thrown by the error condition and then display appropriate message for taking corrective actions"

## **Types of Exceptions**

- 1. Checked Exception: A checked exception is an exception that is typically a user error or a problem that cannot be foreseen by the programmer. Checked exception can also be defined as "*The classes that extend the Throwable class except RuntimeException and Error are known as Checked Exceptions*". For example, if a file is to be opened, but the file cannot be found, an exception occurs. These exceptions are checked at compile-time and cannot simply be ignored at the time of compilation. Example of Checked Exception are IOException, SQLException etc.
- 2. Unchecked Exception: Also known as Runtime Exceptions and they are ignored at the time of compilation but checked during execution of the program. Unchecked Exceptions can also be defined as "*The Classes that extend the RuntimeException class are known as Unchecked Exceptions*". Example are ArithmeticException, NullPointerException etc.
- **3.** Error: 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.

## **Hierarchy of Exception**

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.



## **Table of JAVA – Built in Exceptions**

Exception	Description	
ArithmeticException	Arithmetic error, such as divide-by-zero.	
ArrayIndexOutOfBoundsExcepti	Array index is out-of-bounds.	
on		
ArrayStoreException	Assignment to an array element of an incompatible	
	type.	
ClassCastException	Invalid cast.	
IllegalArgumentException	Illegal argument used to invoke a method.	
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an	
	unlocked thread.	
IllegalStateException	Environment or application is in incorrect state.	
IllegalThreadStateException	Requested operation not compatible with current	
	thread state.	
IndexOutOfBoundsException	Some type of index is out-of-bounds.	
NegativeArraySizeException	Array created with a negative size.	
NullPointerException	Invalid use of a null reference.	
NumberFormatException	Invalid conversion of a string to a numeric format.	
SecurityException	Attempt to violate security.	
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.	
UnsupportedOperationException	An unsupported operation was encountered.	

Following is the list of Java Unchecked RuntimeException

Exception	Description
ClassNotFoundException	Class not found.
CloneNotSupportedExcepti on	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.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.

Following is the list of Java Checked Exceptions Defined in java.lang

## **Handling Exceptions in Java**

Following five keywords are used to handle an exception in Java:

- 1. try
- 2. catch
- 3. finally
- 4. throw
- 5. throws

## try -catch block

A method catches an exception using a combination of the **try and catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following:



A catch statement involves declaring the type of exception you are trying to catch. If an exception occurs in protected code, the catch block (or blocks) that follows the try is checked. If the type of exception that occurred is listed in a catch block, the exception is passed to the

catch block much as an argument is passed into a method parameter.

#### **Example of Program without Exceptional Handling**



#### **Output:**

```
C:\Achin Jain>java excep1 12
Value of i is:8
C:\Achin Jain>java excep1 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
at excep1.main<excep1.java:5>
```

#### Same Program with Exception Handling

```
class excep1
ł
    public static void main(String args[])
    {
         try
                                                               Now as the statement which can
         ł
                                                               cause error condition is wrapped
              int i=100/Integer.parseInt(args[0]);
                                                               under try block and catch block is also
              System.out.println("Value of j is:"+i);
                                                               present to handle the exception
                                                               object thrown. In this case even if
         catch(ArithmeticException e)
                                                               there is an error rest of the program
         ł
                                                               will execute normally
              System.out.println(e);
         System.out.println("Code after try-catch");
    }
}
```

#### **Output**

```
C:\Achin Jain>java excep1 12
Value of i is:8
Code after try-catch
C:\Achin Jain>java excep1 0
java.lang.ArithmeticException: / by zero
Code after try-catch
```

## **Multiple Catch Blocks:**

A try block can be followed by multiple catch blocks, but when we use multiple catch statements it is important that exception subclasses must come before any of their superclasses. The reason is "*a catch statement with superclass will catch exceptions of that type plus any of its subclass, thus causing a catch statement with subclass exception a non-reachable code which is error in JAVA"*.

#### **Example:**



#### **Output**



However if the order of the catch blocks is reversed like shown below, then program will execute normally

```
catch(ArithmeticException e)
{
    System.out.println(e);
}
catch(Exception e1)
{
    System.out.println(e1);
}
```

## **Finally Block**

The finally keyword is used to create a block of code that follows a try block. A finally block of code always executes, whether or not an exception has occurred. Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code.



#### **Example of Finally Statement**

```
class excep1
Ł
    public static void main(String args[])
    ł
        try
        Ł
            int i=100/Integer.parseInt(args[0]);
            System.out.println("Value of i is:"+i);
        ł
        catch(ArithmeticException e1)
        £
            System.out.println(e1);
        finally
        ł
            System.out.println("Finally Code Executed");
        System.out.println("Code after try-catch");
}
```

#### Output 1

In the first case no command line arguments are passed which will throw ArrayIndexOutOfBoundsException and in the above code we are handling only ArithmeticException which will cause the system to terminate and remaining program will not run. But in this case also the statement written in the finally block will gets executed as shown below:

```
C:\Achin Jain>java excep1
Finally Code Executed
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
at excep1.main<excep1.java:?>
```

In second case '0' is passed as command line argument to let program throw ArithmeticException which will eventually be handled by catch block. See the output below which clearly shows that remaining part of the code will also run along with finally statement.

```
C:\Achin Jain>java excep1 Ø
j<u>ava.lang.ArithmeticEx</u>ception: ∕ by zero
Finally Code Executed
Code after try-catch
```

In third case '5' is passed as command line argument which is perfectly fine and in this case no exception will be throws. Now see the output below, in this case also finally statement will get executed.

## **Throw Keyword**

The throw keyword is used to explicitly throw an exception. We can throw either checked or unchecked exception. The throw keyword is normally used to throw custom exception.

## **Example**

In the example shown below a method validage(int i) is used which will check the value of passed parameter *i* and if the value is less than 18 than a ArithmeticException is thrown. Now as you can see when we have called the method no try catch block is used which results in termination of the program and message is displayed as "not valid" which is passed during throwing of ArithmeticException object.

```
class excep1
ł
    static void validage(int i)
    ł
        if(i<18)
        £
             throw new ArithmeticException("not valid");
        ł
        else
        ł
            System.out.println("Welcome");
    public static void main(String args[])
    Ł
        validage(12);
        System.out.println("Code after try-catch");
}
```

## <u>Output</u>

```
C:\Achin Jain>javac excep1.java
C:\Achin Jain>java excep1
Exception in thread "main" java.lang.ArithmeticException: not valid
at excep1.validage(excep1.java:7)
at excep1.main(excep1.java:16)
```

However if during call of validage method try-catch block has been used then the program will

run normally

```
try
{
    validage(12);
}
catch(ArithmeticException e)
{
    System.out.println(e);
}
```

#### **Output**

```
C:\Achin Jain>javac excep1.java
C:\Achin Jain>java excep1
java.lang.ArithmeticException: not valid
Code after try-catch
```

## **Throws Keyword**

The throws keyword is used to declare the exception, it provide information to the programmer that there may occur an exception so during call of that method, and programmer must use exceptional handling mechanism. Throws keyword is also used to propagate checked exception.

## **Example**

In this example, exception is created by extending Exception class and the custom exception is declared in the method validage(int i)

```
class ajexception extends Exception
                                                Code to create custom exception with
  ł
                                                name "ajexception"
      ajexception(String s)
      £
          super(s);
      3
  - }
  class excep2
  ł
      static void validage(int i) throws ajexception
      ł
          if(i<18)
           ł
               throw new ajexception("not valid");
      public static void main(String args[])
          validage(12);
           System.out.println("Code after try-catch");
DAYAKAR GURRAM ASST. PROFESSOR CSE AVNIET
```

<u>Case 1:</u> During call of validage method exceptional handling is not used and code looks like this and error is displayed in the compilation of the code.

```
public static void main(String args[])
{
    validage(12);
    System.out.println("Code after try-catch");
}
```

## <u>Output</u>

```
C:\Achin Jain>javac excep2.java
excep2.java:19: error: unreported exception ajexception; must be caught or decla
red to be thrown
validage(12);
```

<u>**Case 2:**</u> During call of method validage exceptional handling is used with try-catch keyword like this and the program runs as expected.

```
try
{
    validage(12);
}
catch(ajexception aj)
{
    System.out.println(aj);
}
```

## Output:

```
C:∖Achin Jain≻javac excep2.java
C:\Achin Jain≻java excep2
ajexception: not valid
Code after try-catch
```

<u>Case 3:</u> During call of method validage exceptional handling is used without try-catch keyword and throws keyword is used in main method as shown below

```
public static void main(String args[]) throws ajexception
{
    validage(12);
    System.out.println("Code after try-catch");
}
```

There will be no error now during compile time, but program will gets terminated when exception event takes place.

#### <u>Output</u>

```
C:\Achin Jain>javac excep2.java
C:\Achin Jain>java excep2
Exception in thread "main" ajexception: not valid
at excep2.validage(excep2.java:14)
at excep2.main(excep2.java:19)
```

**<u>Case 4:</u>** Try to propagate custom exception not of type RuntimeException without declaring in method using throws keyword. This will give compile time error



#### **Output:**



<u>Case 5:</u> Make custom exception by extending RunTimeException Class and try the same method as use for Case 4. There will be no error now and the program runs as expected

```
class ajexception extends RuntimeException
{
    ajexception(String s)
    {
        super(s);
    }
}
```

#### **Output**



## **Important Points in Exceptional Handling**

- A catch clause cannot exist without a try statement.
- It is not compulsory to have finally clauses whenever a try/catch block is present.
- The try block cannot be present without either catch clause or finally clause.
- Any code cannot be present in between the try, catch, finally blocks.

## **Declaring your Own Exceptions**

You can create your own exceptions in Java. Keep the following points in mind when writing your own exception classes:

- All exceptions must be a child of Throwable.
- If you want to write a checked exception that is automatically enforced by the Handle or Declare Rule, you need to extend the Exception class.
- If you want to write a runtime exception, you need to extend the RuntimeException class.

**Example to create custom exception** is shown in the section above.

## **Multithreading**

Multithreading is a conceptual programming concept where a program (process) is divided into two or more subprograms (process), which can be implemented at the same time in parallel. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution. A *process* consists of the memory space allocated by the operating system that can contain one or more threads. A thread cannot exist on its own; it must be a part of a process.

There are two distinct types of <u>Multitasking</u> i.e. Processor-Based and Thread-Based multitasking.

**Q:** What is the difference between thread-based and process-based multitasking?

**Ans:** As both are types of multitasking there is very basic difference between the two. **Process-Based multitasking** is a feature that allows your computer to run two or more programs concurrently. For example you can listen to music and at the same time chat with your friends on Facebook using browser. In Thread-based multitasking, thread is the smallest unit of code, which means a single program can perform two or more tasks simultaneously. For example a text editor can print and at the same time you can edit text provided that those two tasks are perform by separate threads.

Q: Why multitasking thread requires less overhead than multitasking processor?

**Ans:** A multitasking thread requires less overhead than multitasking processor because of the following reasons:

- Processes are heavyweight tasks where threads are lightweight
- Processes require their own separate address space where threads share the address space
- Interprocess communication is expensive and limited where Interthread communication is inexpensive, and context switching from one thread to the next is lower in cost.

## **Benefits of Multithreading**

- 1. Enables programmers to do multiple things at one time
- 2. Programmers can divide a long program into threads and execute them in parallel which eventually increases the speed of the program execution
- 3. Improved performance and concurrency
- 4. Simultaneous access to multiple applications

## Life Cycle of Thread

A thread can be in any of the five following states

- 1. Newborn State: When a thread object is created a new thread is born and said to be in Newborn state.
- **2. Runnable State:** If a thread is in this state it means that the thread is ready for execution and waiting for the availability of the processor. If all threads in queue are of same priority then they are given time slots for execution in round robin fashion
- **3. Running State:** It means that the processor has given its time to the thread for execution. A thread keeps running until the following conditions occurs
  - a. Thread give up its control on its own and it can happen in the following situations
    - i. A thread gets suspended using **suspend**() method which can only be

- ii. A thread is made to sleep for a specified period of time using sleep(time) method, where time in milliseconds
- iii. A thread is made to wait for some event to occur using wait () method.In this case a thread can be scheduled to run again using notify () method.
- b. A thread is pre-empted by a higher priority thread
- **4. Blocked State:** If a thread is prevented from entering into runnable state and subsequently running state, then a thread is said to be in Blocked state.
- **5. Dead State:** A runnable thread enters the Dead or terminated state when it completes its task or otherwise terminates.



Fig: Life Cycle of Thread

## Main Thread

Every time a Java program starts up, one thread begins running which is called as the <u>main</u> <u>thread</u> of the program because it is the one that is executed when your program begins.

- Child threads are produced from main thread
- Often it is the last thread to finish execution as it performs various shut down operations

#### **Creating a Thread**

Java defines two ways in which this can be accomplished:

- You can implement the Runnable interface.
- You can extend the Thread class, itself.

#### **Create Thread by Implementing Runnable**

The easiest way to create a thread is to create a class that implements the Runnable interface. To implement Runnable, a class need only implement a single method called **run()**, which is declared like this:

#### public void run( )

You will define the code that constitutes the new thread inside **run**() method. It is important to understand that **run**() can call other methods, use other classes, and declare variables, just like the main thread can.

After you create a class that implements Runnable, you will instantiate an object of type Thread from within that class. Thread defines several constructors. The one that we will use is shown here:

#### Thread(Runnable threadOb, String threadName);

Here threadOb is an instance of a class that implements the Runnable interface and the name of the new thread is specified by threadName. After the new thread is created, it will not start running until you call its **start()** method, which is declared within Thread. The start() method is shown here:

void start( );

#### **Example to Create a Thread using Runnable Interface**

```
class t1 implements Runnable
{
    public void run()
    {
        System.out.println("Thread is Running");
    }
    public static void main(String args[])
    {
        t1 obj1 = new t1();
        Thread t = new Thread(obj1);
        t.start();
    }
}
```

#### **Output:**

C:\NIEC Java>javac t1.java C:\NIEC Java>java t1 Thread is Running C:\NIEC Java>

#### **Create Thread by Extending Thread**

The second way to create a thread is to create a new class that extends Thread, and then to create an instance of that class. The **extending class must override the run() method**,

which is the entry point for the new thread. It must also call start() to begin execution of the new thread.

#### **Example to Create a Thread by Extending Thread Class**

```
class t2 extends Thread
{
    public void run()
    {
        System.out.println("Thread is Running");
    }
    public static void main(String args[])
    {
        t2 obj1 = new t2();
        obj1.start();
    }
}
```

#### **Output:**

```
C:\NIEC Java>javac t2.java
C:\NIEC Java>java t2
Thread is Running
C:\NIEC Java>
```

## **Thread Methods**

SN	Methods with Description
1	public void start()
	Starts the thread in a separate path of execution, then invokes the run() method
	on this Thread object.
2	public void run()
	If this Thread object was instantiated using a separate Runnable target,
	the run() method is invoked on that Runnable object.
3	public final void setName(String name)
	Changes the name of the Thread object. There is also a getName() method
	for retrieving the name.
4	public final void setPriority(int priority)
	Sets the priority of this Thread object. The possible values are between 1 and 10.
5	public final void setDaemon(boolean on)
	A parameter of true denotes this Thread as a daemon thread.
6	public final void join(long millisec)
	The current thread invokes this method on a second thread, causing the current thread
	to block until the second thread terminates or the specified number of milliseconds
	passes.
7	public void interrupt()
	Interrupts this thread, causing it to continue execution if it was blocked for any
0	reason.
0	Public final boolean isAnve() Paturns true if the thread is alive, which is any time after the thread has been
	started but before it runs to completion

**Q:** Can we start a thread twice?

**Ans:** No, if a thread is started it can never be started again, if you do so, an illegalThreadStateException is thrown. Example is shown below in which a same thread is coded to start again



## Use of Yield() Method

Causes the currently running thread to yield to any other threads of the same priority that are waiting to be scheduled

#### Example

As you can see in the output below, thread A gets started and when condition if(i==2) gets satisfied yield() method gets evoked and the control is relinquished from thread A to thread B which run to its completion and only after that thread a regain the control back.

## Output



## Use of stop() Method

The stop() method kills the thread on execution

## Example

```
class C extends Thread
Ł
    public void run()
    £
        for(int i=1; i<=5; i++)</pre>
         ł
                                               Condition is checked and when i==2
             if(i==2) stop();
                                               stop() method is evoked causing
             System.out.println("A:" +i);
                                               termination of thread execution
         }
        System.out.println("Exit from A");
    public static void main(String args[])
    Ł
        C c = new C();
        c.start();
}
```

## Output

C:∖NIEC Java≻java C A:1

## Use of sleep() Method

Causes the currently running thread to block for at least the specified number of milliseconds.

You need to handle exception while using sleep() method.

#### Example

```
class C extends Thread
ł
    public void run()
    £
         for(int i=1; i<=5; i++)</pre>
              try
                                                 Condition is checked and when i==2
              ł
                                                 sleep() method is evoked which halts the
                  if(i==2) sleep(1000);
                                                 execution of the thread for 1000
              3
                                                 milliseconds. When you see output there
              catch(Exception e)
                                                 is no change but there is delay in
              ł
                                                 execution.
              System.out.println("A:" +i);
         }
         System.out.println("Exit from A");
     }
    public static void main(String args[])
    ł
         C c = new C();
         c.start();
     3
}
```

#### Output



## Use of suspend() and resume() method

A suspended thread can be revived by using the resume() method. This approach is useful when we want to suspend a thread for some time due to certain reason but do not want to kill it.

Following is the example in which two threads C and A are created. Thread C is started ahead of Thread A, but C is suspended using suspend() method causing Thread A to get hold of the processor allowing it to run and when Thread C is resumed using resume() method it runs to its completion.

## Example

```
class C extends Thread
{
    public void run()
    ł
         for(int i=1; i<=5; i++)</pre>
         ł
             System.out.println("C:" +i);
         System.out.println("Exit from C");
}
class A extends Thread
ł
    public void run()
    ł
         for(int i=1; i<=5; i++)</pre>
         ł
             System.out.println("A:" +i);
         }
         System.out.println("Exit from A");
}
class suspendtest
{
    public static void main(String args[])
    ł
         C c = new C();
        A = new A();
         c.start();
         a.start();
                                        Although Thread 'C' is started earlier than
         c.suspend();
                                        Thread 'A' but due to suspend method
         c.resume();
                                        Thread 'A' gets completed ahead of
                                        Thread 'C'
    3
1
Output
C:\Achin Jain>java suspendtest
A:1
A:2
A:3
 1:4
A:5
Exit from A
 xit from C
```

#### **Thread Priority**

Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.

Java priorities are in the range between MIN\_PRIORITY (a constant of 1) and MAX\_PRIORITY (a constant of 10). By default, every thread is given priority NORM\_PRIORITY (a constant of 5).

Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute and very much platform dependent.

## Example

```
class prioritytest
{
    public static void main(String args[])
    {
        C c = new C();
        A a = new A();
        a.setPriority(10);
        c.setPriority(1);
        c.start();
        a.start();
    }
}
```

In the above code, you can see Priorities of Thread is set to maximum for Thread A which lets it to run to completion ahead of C which is set to minimum priority.

#### **Output:**



#### Use of isAlive() and join() method

The java.lang.Thread.isAlive() method tests if this thread is alive. A thread is alive if it has been started and has not yet died. Following is the declaration for java.lang.Thread.isAlive() method

#### public final boolean isAlive()

This method returns true if this thread is alive, false otherwise.

**join() method** waits for a thread to die. It causes the currently thread to stop executing until the thread it joins with completes its task.

#### Example



#### **Synchronization**

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 synchronization is achieved is called thread synchronization. The synchronized keyword in Java creates a block of code referred to as a critical section. Every Java object with a critical section of code gets a lock associated with the object. To enter a critical section, a thread needs to obtain the corresponding object's lock.

#### synchronized(object)

## { // statements to be synchronized }

## Problem without using Synchronization

In the following example method updatesum() is not synchronized and access by both the threads simultaneously which results in inconsistent output. Making a method synchronized, Java creates a "monitor" and hands it over to the thread that calls the method first time. As long as the thread holds the monitor, no other thread can enter the synchronized section of the code. Writing the method as synchronized will make one thread enter the method and till execution is not complete no other thread can get access to the method.

```
synchronized void updatesum(int i)
{
   Thread t = Thread.currentThread();
   for(int n=1; n<=5; n++)
   {
      System.out.println(t.getName()+" : "+(i+n));
   }
}</pre>
```



#### **Output**

C:\Achin	Jain≻java syntest
Thread A	: 11
Thread B	: 11
Thread A	: 12
Thread B	: 12
Thread A	: 13
Thread B	: 13
Thread A	: 14
Thread B	: 14
Thread A	: 15
Thread B	: 15

## **Interthread Communication**

It is all about making synchronized threads communicate with each other. It is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter in the same critical section to be executed. It is implemented by the following methods of Object Class:

- wait(): This method tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify().
- **notify():** This method wakes up the first thread that called wait() on the same object.
- **notifyAll():** This method wakes up all the threads that called wait() on the same object. The highest priority thread will run first.

These methods are implemented as final methods in Object, so all classes have them. All three methods can be called only from within a synchronized context.

## Example

```
class customer
ł
int amount = 0;
int flag = 0;
public synchronized int withdraw(int amount)
Ł
    System.out.println(Thread.currentThread().getName()+" : is going to withdraw");
    if(flag==0)
    ł
        try
        {
            System.out.println("Waiting....");
            wait();
        ł
        catch(Exception e)
        }
                                                       If both these methods are commented which
    this.amount-=amount;
                                                       means there is no communication, output
    System.out.println("Withdraw Complete");
                                                       will be inconsistent. See Output 2
    return amount;
}
public synchronized void deposit(int amount)
    System.out.println(Thread.currentThread().getName()+" : is going to Deposit");
    this.amount+=amount;
    notifyAll();
    System.out.println("Deposit Complete");
    flag=1;
}
ł
```
```
class threadcomm
ł
    public static void main(String args[])
    ł
        final customer c = new customer();
        Thread t1 = new Thread()
            public void run()
            ł
                c.withdraw(5000);
                System.out.println("After withdraw Amount is :"+ c.amount);
        };
        Thread t2 = new Thread()
            public void run()
            ł
                c.deposit(10000);
                System.out.println("After Deposit Amount is :"+ c.amount);
            }
        };
        t1.start();
        t2.start();
}
```

**Output 1:** 

C:\Achin Jain>java threadcomm Thread-0 : is going to withdraw Waiting.... Thread-1 : is going to Deposit Deposit Complete After Deposit Amount is :10000 Withdraw Complete After withdraw Amount is :5000

**Output 2:** 

C:\Achin Jain≻java threadcomm Thread-0 : is going to withdraw Waiting.... Withdraw Complete After withdraw Amount is :-5000 Thread-1 : is going to Deposit Deposit Complete After Deposit Amount is :5000

# UNIT-V

# **GUI Programming with Swing- Event Handling- Applets** <u>INHERITANCE HIERARCHY FOR APPLETS :</u>

Using HTML program: In order to run the applet through HTML program we must use the following tag.

# <u>Syntax:</u>

<applet code =".class file of the applet" height = height value width = width value> </applet>

File name: MyApp.html

<HTML>

<HEAD>

<TITLE> My applet example </TITLE>

</HEAD>

<BODY>

<APPLET code=''MyApp'' height=100 width=150>

</APPLET>
</BODY>

</HTML>

APPLET	APPLICATION
<ul> <li>Applets must have GUI.</li> <li>Applet starts execution after init() method.</li> <li>Applets can be embedded in HTML pages and downloaded over the Internet.</li> <li>Applets can only be executed inside a java compatible container, such as a browser or appletviewer.</li> <li>Applets execute under strict security limitations that disallow certain operations (sandbox model security).</li> <li>Applet cannot communicate with any server other than which it is downloaded from.</li> <li>Applet is an application that can be run on web browser.</li> <li>An applet can react to the user input and dynamically change.</li> </ul>	<ul> <li>Applications may not have GUI.</li> <li>Application starts execution after main() method.</li> <li>Applications have no special support in HTML for embedding or downloading.</li> <li>Applications are executed at command line JVM by java.exe or jview.exe.</li> <li>Applications have no inherent security restrictions.</li> <li>They do not have such security restrictions as applets.</li> <li>Application is a program that runs in a standalone machine.</li> <li>An application can be an execution of a program in the host system.</li> <li>Applications are more general purpose programs.</li> </ul>

# **APPLET LIFE CYCLE:**



## **PUBLIC VOID INIT ():**

- 1. which is called by the browser only one time after loading the applet.
- 2. This is the first method to execute.
- 3. It is an ideal place to initialize variables.
- 4. It is the best place to define the GUI Components (buttons, text fields, scrollbars, etc.), lay them out, and add listeners to them.
- 5. Almost every applet you ever write will have an init() method.

## **PUBLIC VOID START ():**

- 1. After calling the init method, the next method which is from second request to subsequent requests the start method only will be called i.e., short method will be called each and every time. In this method we write the block of statement which provides business logic.
- 2. Called after init().
- 3. Used mostly in conjunction with stop().
- 4. start() and stop() are used when the Applet is doing time-consuming calculations that you don't want to continue when the page is not in front.

## **PUBLIC VOID STOP():**

- 1. Called when the browser leaves the page.
- 2. Called just before destroy().
- 3. Use stop() if the applet is doing heavy computation that you don't want to continue when the browser is on some other page.
- 4. Used mostly in conjunction with start().

## **PUBLIC VOID DESTROY():**

- 1. Seldom needed.
- 2. Called after stop().
- 3. Use to explicitly release system resources (like threads).
- 4. System resources are usually released automatically.

## PUBLIC VOID PAINT(GRAPHICS G) :

- 1. Needed if you do any drawing or painting other than just using standard GUI Components.
- 2. Any painting you want to do should be done here, or in a method you call from here.
- 3. Painting that you do in other methods may or may not happen.

## **REPAINT() METHOD:**

- 1. Call repaint() when you have changed something and want your changes to show up on the screen.
- 2. repaint() is a request--it might not happen.
- 3. When you call repaint(), Java schedules a call to update(Graphics g).

## **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 parameter in the form of a **String** object. Thus, for numeric and **boolean** values, you will need to convert their string representations into their internal formats. Here is an example

that demonstrates passing parameters:

## // Use Parameters

```
import java.awt.*;
import java.applet.*;
/*
<applet code="ParamDemo" width=300 height=80>
<param name=fontName value=Courier>
<param name=fontSize value=14>
<param name=leading value=2>
<param name=accountEnabled value=true>
</applet>
*/
public class ParamDemo extends Applet
      String fontName;
      int fontSize;
      float leading;
      boolean active;
      // Initialize the string to be displayed.
      public void start()
```

```
{
      String param;
      fontName = getParameter("fontName");
      if(fontName == null)
      fontName = "Not Found";
      param = getParameter("fontSize");
      try
       {
             if(param != null) // if not found
             fontSize = Integer.parseInt(param);
             else
                    fontSize = 0;
       }
       catch(NumberFormatException e)
             fontSize = -1;
       param = getParameter("leading");
      try
       {
             if(param != null) // if not found
                    leading = Float.valueOf(param).floatValue();
             else
                    leading = 0;
       ł
      catch(NumberFormatException e)
             leading = -1;
       param = getParameter("accountEnabled");
      if(param != null)
             active = Boolean.valueOf(param).booleanValue();
}
// Display parameters.
public void paint(Graphics g)
       g.drawString("Font name: " + fontName, 0, 10);
      g.drawString("Font size: " + fontSize, 0, 26);
      g.drawString("Leading: " + leading, 0, 42);
      g.drawString("Account Active: " + active, 0, 58);
}
```

#### Sample output from this program is shown here:

🚔 Applet Viewer: ParamDemo	
Applet	
Font name: Courier	
Font size: 14	
Leading: 2.0	
Account Active: true	
Applet started.	

As the program shows, you should test the return values from getParameter().

If a parameter isn't available, **getParameter()** will return **null**. Also, conversions to numeric types must be attempted in a **try** statement that catches **NumberFormatException**.

Uncaught exceptions should never occur within an applet.

#### **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.
- Some of the activities that cause events to be generated are pressing a button, entering a character via the keyboard, selecting an item in a list, and clicking the mouse.
- Events may also occur that are not directly caused by interactions with a 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. Sources may generate more than one type of event.
- Asource must register listeners in order for the listeners to receive notifications about a specific type of event.
- Each type of event has its own registration method. Here is the general form:

#### public void addTypeListener(TypeListener el)

Here, *Type* is the name of the event, and *el* is a reference to the event listener.

## **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.
- The methods that receive and process events are defined in a set of interfaces found in java.awt.event.

For example, the MouseMotionListener interface defines two methods to receive notifications when the mouse is dragged or moved.

## **Event Classes**

- ◆ The classes that represent events are at the core of Java's event handling mechanism.
- ★ At the root of the Java event class hierarchy is **EventObject**, which is in **java.util**.
- $\diamond$  it is the superclass for all events. Its one constructor is shown here:

#### EventObject(Object src)

Here, *src* is the object that generates this event.

**EventObject** contains two methods:

getSource() and toString().The getSource() method returns the source of the event. Its general form is shown here:

#### Object getSource()

- ★ As expected, **toString()** returns the string equivalent of the event.
- \* The class **AWTEvent**, defined within the **java.awt** package, is a subclass of **EventObject**.
- It is the superclass (either directly or indirectly) of all AWT-based events used by the delegation event model.
- Its getID() method can be used to determine the type of the event. The signature of this method is shown here:

#### int getID()

- **EventObject** is a superclass of all events.
- ✤ AWTEvent is a superclass of all AWT events that are handled by the delegation event model.
- The package java.awt.event defines many types of events that are generated by various user interface elements.

#### **The ActionEvent Class**

An ActionEvent is generated when a button is pressed, a list item is double-clicked, or a menu item is selected. The ActionEvent class defines four integer constants that can be used to identify any modifiers associated with an action event:

# ALT\_MASK, CTRL\_MASK,

META\_MASK, and SHIFT MASK.

In addition, there is an integer constant, **ACTION\_PERFORMED**, which can be used to identify action events.

Event Class	Description
ActionEvent	Generated when a button is pressed, a list item is double-clicked,
	or a menu item is selected
AdjustmentEvent	Generated when a scroll bar is manipulated.
ComponentEvent	Generated when a component is hidden, moved, resized, or
	becomes visible.
ContainerEvent	Generated when a component is added to or removed from a
	container.
FocusEvent	Generated when a component gains or loses keyboard focus.
InputEvent	Abstract superclass for all component input event classes.
ItemEvent	Generated when a check box or list item is clicked; also occurs
	when a choice selection is made or a checkable menu item is
	selected or deselected.
KeyEvent	Generated when input is received from the keyboard.
MouseEvent	Generated when the mouse is dragged, moved, clicked, pressed,
	or released;
	also generated when the mouse enters or exits a component.
MouseWheelEvent	Generated when the mouse wheel is moved.
TextEvent	Generated when the value of a text area or text field is changed.
WindowEvent	Generated when a window is activated, closed, deactivated,
	deiconified, iconified, opened, or quit.

• ActionEvent has these three constructors:

<u>ActionEvent(Object src, int type, String cmd)</u> <u>ActionEvent(Object src, int type, String cmd, int modifiers)</u> ActionEvent(Object src, int type, String cmd, long when, int modifiers)

You can obtain the command name for the invoking **ActionEvent** object by using the **getActionCommand()** method, shown here:

String getActionCommand( )

The getModifiers() method returns a value that indicates which modifier keys (ALT, CTRL, META, and/or SHIFT) were pressed when the event was generated. Its form is shown here:

## int getModifiers()

The method **getWhen()** returns the time at which the event took place. This is called the event's *timestamp*. The **getWhen()** method is shown here:

long getWhen( )

## The AdjustmentEvent Class

An AdjustmentEvent is generated by a scroll bar. There are five types of adjustment events.
The AdjustmentEvent class defines integer constants that can be used to identify them.
The constants and their meanings are shown here:

COMPONENT_HIDDEN	The component was hidden.
------------------	---------------------------

BLOCK_DECREMENT	The user clicked inside the scroll bar to decrease its value.
BLOCK_INCREMENT	The user clicked inside the scroll bar to increase its value.
TRACK	The slider was dragged.
UNIT_DECREMENT	The button at the end of the scroll bar was clicked to decrease its value.
UNIT_INCREMENT	The button at the end of the scroll bar was clicked to increase its value.

Here is one **AdjustmentEvent** constructor:

AdjustmentEvent(Adjustable src, int id, int type, int data)

The **getAdjustable()** method returns the object that generated the event. Its form is shown here:

<u>Adjustable getAdjustable()</u> int getValue()

## The ComponentEvent Class

✤ A ComponentEvent is generated when the size, position, or visibility of a component is changed. There are four types of component events.

The constants and their meanings are shown here:

COMPONENT_MOVED	The component was moved.
COMPONENT_RESIZED	The component was resized.
COMPONENT_SHOWN	The component became visible.

**ComponentEvent** has this constructor:

ComponentEvent(Component src, int type)

**ComponentEvent** is the superclass either directly or indirectly of **ContainerEvent**, **FocusEvent**, **KeyEvent**, **MouseEvent**, and **WindowEvent**.

The **getComponent()** method returns the component that generated the event. It is shown here:

Component getComponent( )

#### The ContainerEvent Class

✤ A ContainerEvent is generated when a component is added to or removed from a container. There are two types of container events.

#### COMPONENT\_ADDED and COMPONENT\_REMOVED.

They indicate that a component has been added to or removed from the container.
 ContainerEvent is a subclass of ComponentEvent and has this constructor:

ContainerEvent(Component src, int type, Component comp)

You can obtain a reference to the container that generated this event by using the **getContainer()** method, shown here:

#### Container getContainer()

The **getChild()** method returns a reference to the component that was added to or removed from the container. Its general form is shown here:

Component getChild()

#### The FocusEvent Class

A FocusEvent is generated when a component gains or loses input focus. These events are identified by the integer constants FOCUS\_GAINED and FOCUS\_LOST.
 FocusEvent is a subclass of ComponentEvent and has these constructors:

FocusEvent(Component *src*, int *type*) FocusEvent(Component *src*, int *type*, boolean *temporaryFlag*) FocusEvent(Component *src*, int *type*, boolean *temporaryFlag*, Component *other*) You can determine the other component by calling **getOppositeComponent(**), shown here:

#### Component getOppositeComponent( )

The opposite component is returned. The **isTemporary()** method indicates if this focus change is temporary. Its form is shown here:

#### boolean isTemporary()

The method returns **true** if the change is temporary. Otherwise, it returns **false**.

#### The InputEvent Class

The abstract class InputEvent is a subclass of ComponentEvent and is the superclass for component input events. Its subclasses are KeyEvent and MouseEvent.

The **InputEvent** class defined the following eight values to represent the modifiers:

ALT_MASK	BUTTON2_MASK	META_MASK
ALT_GRAPH_MASK	BUTTON3_MASK	SHIFT_MASK
BUTTON1_MASK	CTRL_MASK	

The following extended modifier values were added because of possible conflicts between the modifiers used by keyboard events and mouse events, and other issues.

ALT_DOWN_MASK	BUTTON2_DOWN_MASK	META_DOWN_MASK
ALT_GRAPH_DOWN_MASK	BUTTON3_DOWN_MASK	SHIFT_DOWN_MASK
BUTTON1_DOWN_MASK	CTRL_DOWN_MASK	

The forms of these methods are shown here:

boolean isAltDown() boolean isAltGraphDown() boolean isControlDown() boolean isMetaDown() boolean isShiftDown()

You can obtain a value that contains all of the original modifier flags by calling the getModifiers() method. It is shown here:

#### int getModifiers()

♦ You can obtain the extended modifiers by calling **getModifiersEx(**), which is shown here:

#### int getModifiersEx()

#### The ItemEvent Class

An ItemEvent is generated when a check box or a list item is clicked or when checkable menu item is selected or deselected. There are two types of item events, which are identified by the following integer constants:

DESELECTED	The user deselected an item.
SELECTED	The user selected an item.

In addition, ItemEvent defines one integer constant, ITEM\_STATE\_CHANGED, that signifies a change of state. ItemEvent has this constructor:

#### ItemEvent(ItemSelectable src, int type, Object entry, int state)

The getItem() method can be used to obtain a reference to the item that generated an event. Its signature is shown here:

#### Object getItem()

the getItemSelectable() method can be used to obtain a reference to the ItemSelectable object that generated an event. Its general form is shown here:

#### ItemSelectable getItemSelectable( )

Lists and choices are examples of user interface elements that implement the ItemSelectable interface.

The getStateChange() method returns the state change (that is, SELECTED or DESELECTED) for the event. It is shown here:

#### int getStateChange( )

#### The KeyEvent Class

- A KeyEvent is generated when keyboard input occurs. There are three types of key events, which are identified by these integer constants: KEY\_PRESSED, KEY\_RELEASED, and KEY\_TYPED.
- The first two events are generated when any key is pressed or released. The last event occurs only when a character is generated.
- There are many other integer constants that are defined by KeyEvent. For example, VK\_0 through VK\_9 and VK\_A through VK\_Z define the ASCII equivalents of the numbers and letters. Here are some others:

VK_ALT	VK_DOWN	VK_LEFT	VK_RIGHT
VK_CANCEL	VK_ENTER	VK_PAGE_DOWN	VK_SHIFT
VK_CONTROL	VK_ESCAPE	VK_PAGE_UP	VK_UP

The VK constants specify *virtual key codes* and are independent of any modifiers, such as control, shift, or alt.KeyEvent is a subclass of InputEvent. Here is one of its constructors:

KeyEvent(Component src, int type, long when, int modifiers, int code, char ch)

The KeyEvent class defines several methods, but the most commonly used ones are getKeyChar(), which returns the character that was entered, and getKeyCode(), which returns the key code. Their general forms are shown here:

<u>char getKeyChar()</u> <u>int getKeyCode()</u>

If no valid character is available, then getKeyChar() returns CHAR\_UNDEFINED. When a KEY\_TYPED event occurs, getKeyCode() returns VK\_UNDEFINED.

## The MouseEvent Class

There are eight types of mouse events. The MouseEvent class defines the following integer constants that can be used to identify them:

MOUSE_CLICKED	The user clicked the mouse.
MOUSE_DRAGGED	The user dragged the mouse.
MOUSE_ENTERED	The mouse entered a component.
MOUSE_EXITED	The mouse exited from a component.
MOUSE_MOVED	The mouse moved.
MOUSE_PRESSED	The mouse was pressed.
MOUSE_RELEASED	The mouse was released.
MOUSE_WHEEL	The mouse wheel was moved.

MouseEvent is a subclass of InputEvent. Here is one of its constructors:

<u>MouseEvent(Component src, int type, long when, int modifiers,</u> <u>int x, int y, int clicks, boolean triggersPopup)</u>

Two commonly used methods in this class are getX() and getY(). These return the X and Y coordinates of the mouse within the component when the event occurred. Their forms are shown here:

int getX() int getY()

Alternatively, you can use the **getPoint(**) method to obtain the coordinates of the mouse. It is shown here:

Point getPoint( )

It returns a **Point** object that contains the X,Y coordinates in its integer members: x and y.The **translatePoint()** method changes the location of the event. Its form is shown here:

#### void translatePoint(int x, int y)

- $\bullet$  Here, the arguments x and y are added to the coordinates of the event.
- The getClickCount() method obtains the number of mouse clicks for this event. Its signature is shown here:

#### int getClickCount( )

Also available is the **getButton()** method, shown here:

#### int getButton()

It returns a value that represents the button that caused the event. The return value will be one of these constants defined by MouseEvent:

✤ Java SE 6 added three methods to MouseEvent that obtain the coordinates of the mouse relative to the screen rather than the component. They are shown here:

Point getLocationOnScreen()
int getXOnScreen()
int getYOnScreen()

The getLocationOnScreen() method returns a Point object that contains both the X and Y coordinate. The other two methods return the indicated coordinate.

#### The MouseWheelEvent Class

The MouseWheelEvent class encapsulates a mouse wheel event. It is a subclass of MouseEvent.Not all mice have wheels.

WHEEL_BLOCK_SCROLL	A page-up or page-down scroll event occurred.
WHEEL_UNIT_SCROLL	A line-up or line-down scroll event occurred.

Here is one of the constructors defined by **MouseWheelEvent**:

<u>MouseWheelEvent(Component src, int type, long when, int modifiers,</u> <u>int x, int y, int clicks, boolean triggersPopup,</u> int scrollHow, int amount, int count)

MouseWheelEvent defines methods that give you access to the wheel event. To obtain the number of rotational units, call getWheelRotation(), shown here:

int getWheelRotation( )

It returns the number of rotational units. If the value is positive, the wheel moved counterclockwise. If the value is negative, the wheel moved clockwise.
 To obtain the type of scroll, call getScrollType(), shown below:

#### int getScrollType( )

It returns either WHEEL\_UNIT\_SCROLL or WHEEL\_BLOCK\_SCROLL. If the scroll type is WHEEL\_UNIT\_SCROLL, you can obtain the number of units to scroll by calling getScrollAmount(). It is shown here:

#### int getScrollAmount()

#### The TextEvent Class

 Instances of this class describe text events. These are generated by text fields and text areas when characters are entered by a user or program. TextEvent defines the integer constant TEXT\_VALUE\_CHANGED. The one constructor for this class is shown here:

#### TextEvent(Object src, int type)

Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type*.

- The TextEvent object does not include the characters currently in the text component that generated the event. Instead, your program must use other methods associated with the text component to retrieve that information.
- This operation differs from other event objects discussed in this section. For this reason, no methods are discussed here for the **TextEvent** class. Think of a text event notification as a signal to a listener that it should retrieve information from a specific text component.

## The WindowEvent Class

There are ten types of window events. The WindowEvent class defines integer constants that can be used to identify them. The constants and their meanings are shown here:

WINDOW_ACTIVATED	The window was activated.
WINDOW_CLOSED	The window has been closed.
WINDOW_CLOSING	The user requested that the window be closed
WINDOW_DEACTIVATED	The window was deactivated.
WINDOW_DEICONIFIED	The window was deiconified.
WINDOW_GAINED_FOCUS	The window gained input focus.
WINDOW_ICONIFIED	The window was iconified.
WINDOW_LOST_FOCUS	The window lost input focus.
WINDOW_OPENED	The window was opened.

WindowEvent is a subclass of ComponentEvent. It defines several constructors. The first is

#### WindowEvent(Window src, int type)

Here, *src* is a reference to the object that generated this event. The type of the event is *type*. The next three constructors offer more detailed control:

<u>WindowEvent(Window src, int type, Window other)</u> WindowEvent(Window src, int type, int fromState, int toState)</u> WindowEvent(Window src, int type, Window other, int fromState, int toState)

✤ A commonly used method in this class is getWindow(). It returns the Window object that generated the event. Its general form is shown here:

#### Window getWindow()

WindowEvent also defines methods that return the opposite window (when a focus or activation event has occurred), the previous window state, and the current window state. These methods are shown here:

<u>Window getOppositeWindow()</u> <u>int getOldState()</u> <u>int getNewState()</u>

#### **Sources of Events**

Event Source	Description
Button	Generates action events when the button is pressed.
Check box	Generates item events when the check box is selected or deselected.
Choice	Generates item events when the choice is changed.
List	Generates action events when an item is double-clicked; generates item events when an item is selected or deselected.
Menu Item	Generates action events when a menu item is selected; generates item events when a checkable menu item is selected or deselected.
Scroll bar	Generates adjustment events when the scroll bar is manipulated.
Text components	Generates text events when the user enters a character.
Window	Generates window events when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

## **Event Listener Interfaces**

- ♦ As explained, the delegation event model has two parts: sources and listeners.
- Listeners are created by implementing one or more of the interfaces defined by the java.awt.event package.When an event occurs, the event source invokes the appropriate method defined by the listener and provides an event object as its argument.

Interface	Description		
ActionListener	Defines one method to receive action events.		
AdjustmentListener	Defines one method to receive adjustment events.		
ComponentListener	Defines four methods to recognize when a component is		
_	hidden, moved, resized, or shown.		
ContainerListener	Defines two methods to recognize when a component is added to		
	or removed from a container.		
FocusListener	Defines two methods to recognize when a component gains or		
	loses keyboard focus.		
ItemListener	Defines one method to recognize when the state of an item		
	changes.		
KeyListener	Defines three methods to recognize when a key is pressed,		
	released, or typed.		
MouseListener	Defines five methods to recognize when the mouse is clicked,		
	enters a component, exits a component, is pressed, or is released.		
MouseMotionListener	Defines two methods to recognize when the mouse is dragged or		
	moved.		
MouseWheelListener	Defines one method to recognize when the mouse wheel is		
	moved.		
TextListener	Defines one method to recognize when a text value changes.		
WindowFocusListener	Defines two methods to recognize when a window gains or loses		
	input focus.		
WindowListener	Defines seven methods to recognize when a window is		
	activated, closed, deactivated, deiconified, iconified, opened, or		
	quit.		

## The ActionListener Interface

This interface defines the actionPerformed() method that is invoked when an action event occurs. Its general form is shown here:

## void actionPerformed(ActionEvent ae)

## The AdjustmentListener Interface

This interface defines the adjustmentValueChanged() method that is invoked when an adjustment event occurs. Its general form is shown here:

## void adjustmentValueChanged(AdjustmentEvent ae)

## The ComponentListener Interface

This interface defines four methods that are invoked when a component is resized, moved, shown, or hidden. Their general forms are shown here:

<u>void componentResized(ComponentEvent *ce*)</u> <u>void componentMoved(ComponentEvent *ce*)</u> <u>void componentShown(ComponentEvent *ce*)</u>

#### void componentHidden(ComponentEvent ce)

#### The ContainerListener Interface

This interface contains two methods. When a component is added to a container, componentAdded() is invoked.

When a component is removed from a container, componentRemoved() is invoked. Their general forms are shown here:

void componentAdded(ContainerEvent ce)
void componentRemoved(ContainerEvent ce)

#### The FocusListener Interface

- This interface defines two methods. When a component obtains keyboard focus, focusGained() is invoked.
- When a component loses keyboard focus, focusLost() is called. Their general forms are shown here:

void focusGained(FocusEvent fe)
void focusLost(FocusEvent fe)

## The ItemListener Interface

This interface defines the itemStateChanged() method that is invoked when the state of an item changes. Its general form is shown here:

#### void itemStateChanged(ItemEvent ie)

## The KeyListener Interface

- This interface defines three methods. The keyPressed() and keyReleased() methods are invoked when a key is pressed and released, respectively.
- The keyTyped() method is invoked when a character has been entered. The general forms of these methods are shown here:

void keyPressed(KeyEvent ke)
void keyReleased(KeyEvent ke)
void keyTyped(KeyEvent ke)

## The MouseListener Interface

- This interface defines five methods. If the mouse is pressed and released at the same point, mouseClicked() is invoked.
- When the mouse enters a component, the mouseEntered() method is called. When it leaves, mouseExited() is called.
- The mousePressed() and mouseReleased() methods are invoked when the mouse is pressed and released, respectively. The general forms of these methods are shown here:

void mouseClicked(MouseEvent me) void mouseEntered(MouseEvent me) void mouseExited(MouseEvent me) void mousePressed(MouseEvent me) void mouseReleased(MouseEvent me)

#### The MouseMotionListener Interface

- This interface defines two methods. The mouseDragged() method is called multiple times as the mouse is dragged.
- The mouseMoved() method is called multiple times as the mouse is moved. Their general forms are shown here:

void mouseDragged(MouseEvent me)
void mouseMoved(MouseEvent me)

#### The MouseWheelListener Interface

This interface defines the mouseWheelMoved() method that is invoked when the mouse wheel is moved. Its general form is shown here:

#### void mouseWheelMoved(MouseWheelEvent mwe)

#### The TextListener Interface

This interface defines the textChanged() method that is invoked when a change occurs in a text area or text field. Its general form is shown here:

void textChanged(TextEvent te)

#### The WindowFocusListener Interface

This interface defines two methods: windowGainedFocus() and windowLostFocus(). These are called when a window gains or loses input focus. Their general forms are shown here:

void windowGainedFocus(WindowEvent we)
void windowLostFocus(WindowEvent we)

## The WindowListener Interface

- This interface defines seven methods. The windowActivated() and windowDeactivated() methods are invoked when a window is activated or deactivated, respectively.
- ◆ If a window is iconified, the **windowIconified**() method is called.
- When a window is deiconified, the **windowDeiconified**() method is called.
- When a window is opened or closed, the windowOpened() or windowClosed() methods are called, respectively.

The windowClosing() method is called when a window is being closed. The general forms of these methods are

void windowActivated(WindowEvent we) void windowClosed(WindowEvent we) void windowClosing(WindowEvent we) void windowDeactivated(WindowEvent we) void windowDeiconified(WindowEvent we) void windowIconified(WindowEvent we) void windowOpened(WindowEvent we)

## **Using the Delegation Event Model**

Applet programming using the delegation event model is actually quite easy. Just follow these two steps:

- Implement the appropriate interface in the listener so that it will receive the type of event desired.
- Implement code to register and unregister (if necessary) the listener as a recipient for the event notifications.

# Handling Mouse Events

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="MouseEvents" width=300 height=100>
</applet>
*/
public class MouseEvents extends Applet implements MouseListener, MouseMotionListener
       String msg = "";
       int mouse X = 0, mouse Y = 0;
       // coordinates of mouse
       public void init()
              addMouseListener(this);
              addMouseMotionListener(this);
       }
       // Handle mouse clicked.
       public void mouseClicked(MouseEvent me)
       {
```

```
// save coordinates
       mouse X = 0;
       mouseY = 10;
       msg = "Mouse clicked.";
       repaint();
}
// Handle mouse entered.
public void mouseEntered(MouseEvent me)
ł
       // save coordinates
       mouse X = 0:
       mouseY = 10;
       msg = "Mouse entered.";
       repaint();
}
// Handle button pressed.
public void mousePressed(MouseEvent me)
ł
       // save coordinates
       mouseX = me.getX();
       mouseY = me.getY();
       msg = "Down";
       repaint();
ł
// Handle button released.
public void mouseReleased(MouseEvent me)
{
       // save coordinates
       mouseX = me.getX();
       mouseY = me.getY();
       msg = "Up";
       repaint();
}
// Handle mouse exited.
public void mouseExited(MouseEvent me)
{
       // save coordinates
       mouse X = 0;
       mouseY = 10;
       msg = "Mouse exited.";
       repaint();
}
// Handle mouse dragged.
public void mouseDragged(MouseEvent me)
{
       // save coordinates
       mouseX = me.getX();
```

```
mouseY = me.getY();
msg = "*";
showStatus("Dragging mouse at " + mouseX + ", " + mouseY);
repaint();
}
// Handle mouse moved.
public void mouseMoved(MouseEvent me)
{
    // show status
    showStatus("Moving mouse at " + me.getX() + ", " + me.getY());
}
// Display msg in applet window at current X,Y location.
public void paint(Graphics g)
{
    g.drawString(msg, mouseX, mouseY);
}
```

## **Adapter Classes**

}

- An adapter class provides an empty implementation of all methods in an event listener interface.
- For example, the MouseMotionAdapter class has two methods, mouseDragged() and mouseMoved().

```
// Demonstrate an adapter.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="AdapterDemo" width=300 height=100>
</applet>
*/
public class AdapterDemo extends Applet
      public void init()
             addMouseListener(new MyMouseAdapter(this));
               addMouseMotionListener(new MyMouseMotionAdapter(this));
class MyMouseAdapter extends MouseAdapter
   AdapterDemo adapterDemo;
   public MyMouseAdapter(AdapterDemo adapterDemo)
      this.adapterDemo = adapterDemo;
```

```
// Handle mouse clicked.
   public void mouseClicked(MouseEvent me)
      adapterDemo.showStatus("Mouse clicked");
   }
}
class MyMouseMotionAdapter extends MouseMotionAdapter
   AdapterDemo adapterDemo;
   public MyMouseMotionAdapter(AdapterDemo adapterDemo)
   {
      this.adapterDemo = adapterDemo;
   }
   // Handle mouse dragged.
   public void mouseDragged(MouseEvent me)
   ł
      adapterDemo.showStatus("Mouse dragged");
   }
}
```

# **INTRODUCION OF SWINGS**

Although the AWT is still a crucial part of Java, its component set is no longer widely used to create graphic user interfaces. Today, most programmers use Swing for this purpose. Swing is a set of Classes that provides more powerful and flexible GUI components than does the AWT.Simply put, Swing provides the look and feel of the modern Java GUI.



AWT	Swing
AWT stands for Abstract windows toolkit.	Swing is also called as JFC's (Java Foundation classes).
AWT components are called Heavyweight component.	Swings are called light weight component because swing components sits on the top of AWT components and do the work.
AWT components require java.awt package.	Swing components require javax.swing package.
AWT components are platform dependent.	Swing components are made in purely java and they are platform independent.
This feature is not supported in AWT.	We can have different look and feel in Swing.
These feature is not available in AWT.	Swing has many advanced features like JTabel, Jtabbed pane which is not available in AWT. Also. Swing components are called "lightweight" because they do not require a native OS object to implement their functionality. JDialog and JFrame are heavyweight, because they do have a peer. So components like JButton, JTextArea, etc., are lightweight because they do not have an OS peer.
With AWT, you have 21 "peers" (one for each control and one for the dialog itself). A "peer" is a widget provided by the operating system, such as a button object or an entry field object.	With Swing, you would have only one peer, the operating system's window object. All of the buttons, entry fields, etc. are drawn by the Swing package on the drawing surface provided by the window object. This is the reason that Swing has more code. It has to draw the button or other control and implement its behavior instead of relying on the host operating system to perform those functions.
AWT is a thin layer of code on top of the OS.	Swing is much larger. Swing also has very much richer functionality.
Using AWT, you have to implement a lot of things yourself.	Swing has them built in.

#### :CONTAINERS: :JFrame:

Frame encapsulates what is commonly thought of as a "window."It is a subclass of Window and has a titlebar,menubar,borders,andresizing corners. If you create a Frame object from within an applet, it will contain a warning message, such as "Java Applet Window," to the user that an applet window has been created. This message warns users that the window they see was started by an applet and not by software running on their computer. When a Frame window is created by a stand-alone application rather than an applet, a normal window is created.

## **Working with Frame Windows:**

After the applet, the type of window you will most often create is derived from Frame. You will use it to create child windows with in applets, and top-level or child windows for stand-alone applications. As mentioned, it creates a standard-style window.

# Here are two of Frame's constructors:

## Frame( ) Frame(String title)

The first form creates a standard window that does not contain a title. The second form creates a window with the title specified by title. Notice that you cannot specify the dimensions of the window. Instead, you must set the size of the window after it has been created.

There are several key methods you will use when working with Frame windows. They are examined here. Setting the Window's Dimensions:

- The setSize() method: is used to set the dimensions of the window.Its signature is shown here:void setSize(int newWidth, int newHeight)
- The getSize() method: is used to obtain the current size of a window. Its signature is shown here: Dimension getSize()
- The setTitle() method: Setting a Window's Title You can change the title in a frame window.
- > The windowClosing() method: To intercept a window-close event.

```
// Create a child frame window from within an applet.
import java.awt.*;
import java.awt.event.*:
import java.applet.*;
/*
<applet code="AppletFrame" width=300 height=50>
</applet>
*/
// Create a subclass of Frame.
class SampleFrame extends Frame
       SampleFrame(String title)
       ł
              super(title);
              // create an object to handle window events
              MyWindowAdapter adapter = new MyWindowAdapter(this);
              // register it to receive those events
              addWindowListener(adapter);
       public void paint(Graphics g)
              g.drawString("This is in frame window", 10, 40);
class MyWindowAdapter extends WindowAdapter
{
```

```
SampleFrame sampleFrame;
       public MyWindowAdapter(SampleFrame sampleFrame)
              this.sampleFrame = sampleFrame;
       public void windowClosing(WindowEvent we)
              sampleFrame.setVisible(false);
}
// Create frame window.
public class AppletFrame extends Applet
ł
       Frame f;
       public void init()
              f = new SampleFrame("A Frame Window");
              f.setSize(250, 250); f.setVisible(true);
       public void start()
              f.setVisible(true);
       public void stop()
              f.setVisible(false);
       public void paint(Graphics g)
              g.drawString("This is in applet window", 10, 20);
```

# . JApplet:

Fundamental to Swing is the JApplet classApplets that use Swing must be subclasses of JApplet.JApplet supports various "panes," such as the contentpane, the glass pane, and the root pane.

When adding a component to an instance of JApplet, do not invoke the add() method of the applet. Instead, call add() for the content pane of the Japplet object.

# **Container getContentPane()**

• add() method of Container can be used to add a component to a content pane

# void add(comp)

# :JDialog:

A Dialog window is an independent **subwindow** meant to carry temporary notice apart from the main Swing Application Window. **Most Dialogs present an error message or warning to a user**, but Dialogs can present images, directory trees, or just about anything compatible with the main Swing Application that manages them.



## :JPanel:

JPanel is a Swing's lightweight container which is used to group a set of components together. JPanel is a pretty simple component which, normally, does not have a GUI (except when it is being set an opaque background or has a visual border).

#### **Constructor:**

- > **JPanel**() : Create a new JPanel with a double buffer and a flow layout
- > JPanel(boolean isDoubleBuffered)

Create a new JPanel with FlowLayout and the specified buffering strategy.

> JPanel(java.awt.LayoutManager layout)

Create a new buffered JPanel with the specified layout manager

> JPanel(java.awt.LayoutManager layout, boolean isDoubleBuffered)

Creates a new JPanel with the specified layout manager and buffering strategy.

```
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
public class PanelWithComponents
ł
      public static void main(String[] a)
             JPanel panel = new JPanel();
             JButton okButton = new JButton("OK");
             panel.add(okButton);
             JButton cancelButton = new JButton("Cancel");
             panel.add(cancelButton);
             JFrame frame = new JFrame("Oval Sample");
             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
             frame.add(panel);
             frame.setSize(300, 200);
             frame.setVisible(true);
      }
}
```

🛃 Oval Sam	ple		
	ок	Cancel	

#### OVERVIEW OF SOME SWING COMPONENTS: :JBUTTON:

The JButton class provides the functionality of a push button. JButton allows an icon, a string, or both.

- JButton(Icon i)
- > JButton(String s)
- JButton(String s, Icon i)

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JButtonDemo" width=250 height=300>
</applet>
*/
public class JButtonDemo extends JApplet implements ActionListener {
      JTextField jtf;
      public void init()
      {
             // Get content pane
             Container contentPane = getContentPane();
             contentPane.setLayout(new FlowLayout());
             // Add buttons to content pane
             ImageIcon france = new ImageIcon("france.gif");
             JButton jb = new JButton(france);
             jb.setActionCommand("France");
             jb.addActionListener(this);
             contentPane.add(jb);
             ImageIcon germany = new ImageIcon("germany.gif");
             jb = new JButton(germany);
             jb.setActionCommand("Germany");
             jb.addActionListener(this);
             contentPane.add(jb);
             ImageIcon italy = new ImageIcon("italy.gif");
             jb = new JButton(italy);
```



public void actionPerformed(ActionEvent ae) {
 jtf.setText(ae.getActionCommand());



}



# :JLabel:

Swing labels are instances of the JLabel class, which extends JComponent. It can display text and/or an icon:

- JLabel(Icon i)
- Label(String s)
- JLabel(String s, Icon i, int align)

The align argument is either LEFT, RIGHT, CENTER, LEADING, or TRAILING .These constants are defined in the SwingConstants interface.

```
/*
```

```
<applet code="JLabelDemo" width=250 height=150>
```

```
</applet>
*/
public class JLabelDemo extends Japplet
      public void init()
      {
            // Get content pane
            Container contentPane = getContentPane();
            // Create an icon
            ImageIcon ii = new ImageIcon("france.gif");
            // Create a label
            JLabel jl = new JLabel("France", ii, JLabel.CENTER);
            // Add label to the content pane
            contentPane.add(jl);
      }
}
           Applet Viewer: JLabelDemo
                                                      - 0 ×
            Applet
                                               France
           Applet started.
```

# :Text Fields:

The Swing text field is encapsulated by the JTextComponent class, which extends Jcomponent JTextField, which allows you to edit one line of text.

- > JTextField()
- JTextField(int cols)
- > JTextField(String s, int cols)
- JTextField(String s)

```
import java.awt.*;
import java.awt.*;
import javax.swing.*;
/*
<applet code=''JTextFieldDemo'' width=300 height=50>
</applet>
*/
public class JTextFieldDemo extends Japplet
{
```

```
JTextField jtf;
public void init()
{
     // Get content pane
     Container contentPane = getContentPane();
     contentPane.setLayout(new FlowLayout());
     // Add text field to content pane
     jtf = new JTextField(15);
     contentPane.add(jtf);
}
```

Applet Viewer: JTextFieldDemo

## :JTextArea:

The JTextArea class is used to create a text area. It is a multiline area that displays the plain text only.

## **Commonly used Constructors:**

**JTextArea**(): creates a text area that displays no text initially.

JTextArea(String s): creates a text area that displays specified text initially.

**JTextArea**(**int row, int column**): creates a text area with the specified number of rows and columns that displays no text initially.

JTextArea(String s, int row, int column): creates a text area with the specified number of rows and columns that displays specified text.

## **Commonly used methods of JTextArea class:**

1) public void setRows(int rows): is used to set specified number of rows.

2) public void setColumns(int cols): is used to set specified number of columns.

3) public void setFont(Font f): is used to set the specified font.

4) **public void insert(String s, int position):** is used to insert the specified text on the specified position.

5) **public void append(String s):** is used to append the given text to the end of the document. **import java.awt.Color;** 

import javax.swing.\*;

public class TArea

JTextArea area; JFrame f; TArea(){

```
f=new JFrame();
area=new JTextArea(300,300);
area.setBounds(10,30,300,300);
area.setBackground(Color.black);
area.setForeground(Color.white);
f.add(area);
f.setSize(400,400);
f.setSize(400,400);
f.setLayout(null);
f.setVisible(true); }
public static void main(String[] args)
{
    new TArea();
}
```

# LAYOUT MANAGERS

## <u>Def:</u>

The standard procedures which exactly specifies the way in which we align the components on the container are known as Layout Manager.

## **TYPES OF LAYOUT MANAGERS:**

- BorderLayout
- BoxLayout
- ➢ CardLayout
- ➢ FlowLayout
- GridBagLayout
- ➢ GridLayout
- GroupLayout
- SpringLayout

## :BorderLayout():

BorderLayout(int horz, int vert)  $\rightarrow$  The second allows you to specify the horizontal and vertical space left between components in horz and vert.

**void add**(**Component** *compObj*, *Object region*); *compObj is the component to be added*, **regions:** 

BorderLayout.CENTER BorderLayout.EAST BorderLayout.NORTH BorderLayout.SOUTH BorderLayout.WEST

```
import java.awt.*;
import java.applet.*;
import java.util.*;
/*
<applet code=''BorderLayoutDemo'' width=400 height=200>
</applet>
*/
public class BorderLayoutDemo extends Applet
{
```

```
public void init()
      setLayout(new BorderLayout());
      add(new Button("This is across the top."),
      BorderLayout.NORTH);
      add(new Label("The footer message might go here."),
      BorderLayout.SOUTH);
      add(new Button("Right"), BorderLayout.EAST);
      add(new Button("Left"), BorderLayout.WEST);
   String msg = "The reasonable man adapts " +
   "himself to the world:\n" +
   "the unreasonable one persists in " +
   "trying to adapt the world to himself.\n" + " Therefore all progress depends " +
   "on the unreasonable man.\n\n" +
   " - George Bernard Shaw\n\n";
   add(new TextArea(msg), BorderLayout.CENTER);
}
```



# :GridLayout:

}

The container will be divided into specific no of rows and columns of equal size **GridLayout has two constructors:** 

> public GridLayout( int rows, int cols );

creates a grid layout with the specified rows and columns.

> public GridLayout( int rows, int cols, int hgap, int vgap );

creates a grid layout with the specified rows and columns ,also lets you specify a minimal horizontal and vertical gap to appear between the components.Each division is called cell or grid

計算器 张雨萌				
888			reset	
0	1	2	3	
4	5	6	7	
8	9	+	-	
*	1	=		

# :FLOW LAYOUT:

- FlowLayout is the default layout manager Components are laid out from the upper-left corner, left to right and top to bottom.
- FlowLayout() → which centers components and leaves five pixels of space between each component
- > FlowLayout(int how)  $\rightarrow$  specify how each line is aligned.

FlowLayout.LEFT FlowLayout.CENTER FlowLayout.RIGHT

FlowLayout(int how, int horz, int vert)

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="FlowLayoutDemo" width=250 height=200>
</applet>
*/
public class FlowLayoutDemo extends Applet implements
ItemListener
ł
      String msg = "";
      Checkbox Win98, winNT, solaris, mac;
      public void init()
             // set left-aligned flow layout
             setLayout(new FlowLayout(FlowLayout.LEFT));
             Win98 = new Checkbox("Windows 98/XP", null, true);
             winNT = new Checkbox("Windows NT/2000");
             solaris = new Checkbox("Solaris");
             mac = new Checkbox("MacOS");
             add(Win98);
             add(winNT);
```
```
add(solaris);
add(mac);
// register to receive item events
Win98.addItemListener(this);
winNT.addItemListener(this);
solaris.addItemListener(this);
mac.addItemListener(this);
}
public void itemStateChanged(ItemEvent ie)
{
repaint();
}

Applet Viewer: FlowLayoutDemo _____Applet

// repaint();
// Constant StateChanged(ItemEvent ie)
```

Current state: Windows 98/XP: true Windows NT/2000: false Solaris: false Mac: false

Applet started.

}