

Unit - 1 Object Oriented Programming concepts

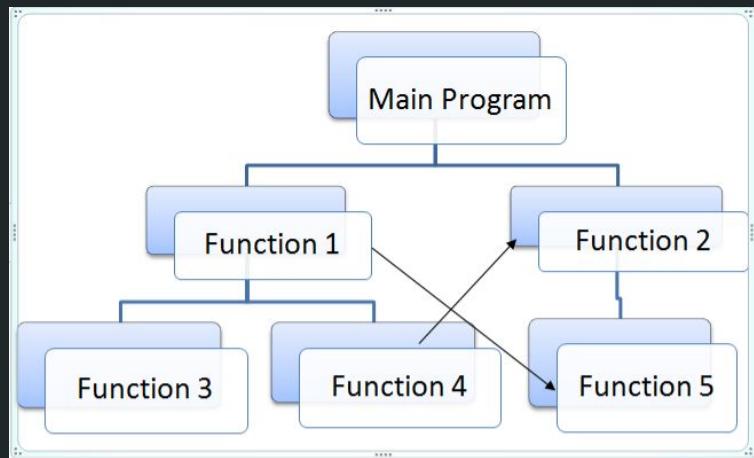
Introduction



Introduction

- CHANGE is one of the contrast characteristics in today's software industry.
- In programming many approaches have been tried like modular programming, top-down programming, bottom-up programming and structured programing to handle the increasing complexity, reliability and maintainability of program.
- Object-Oriented Programming is an approach to program organization and development, which attempts to eliminate some of the pitfalls of conventional programming methods.

Procedure Oriented Programming



Procedure-Oriented Programming

- In the procedure oriented approach, the problem is viewed as the sequence of things to be done such as reading, calculating and printing.
- The primary focus is on functions.

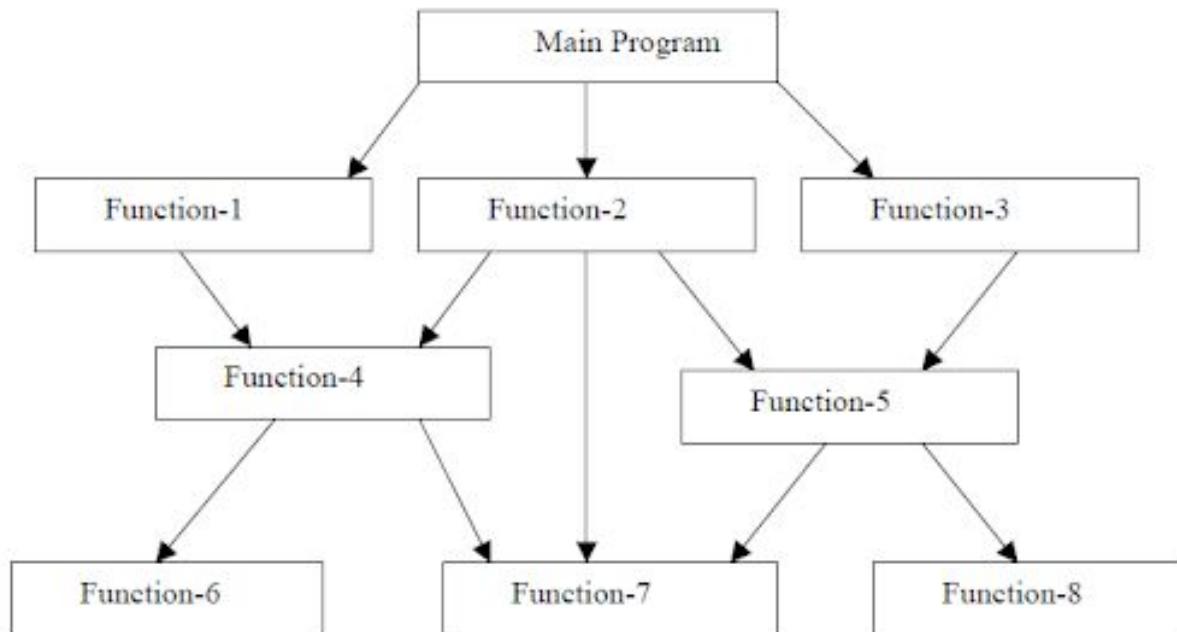


Fig. 1.2 Typical structure of procedural oriented programs

Procedure-Oriented Programming

- Procedure oriented programming basically consists of writing a list of instructions for the computer to follow, and organizing these instructions into groups known as functions.
- We normally use flowcharts to organize these actions and represent the flow of control from one action to another.
- In a multi-function program, many important data items are placed as global so that they may be accessed by all the functions. In a large program it is very difficult to identify what data is used by which function. In case we need to revise an external data structure, we also need to revise all functions that access the data.

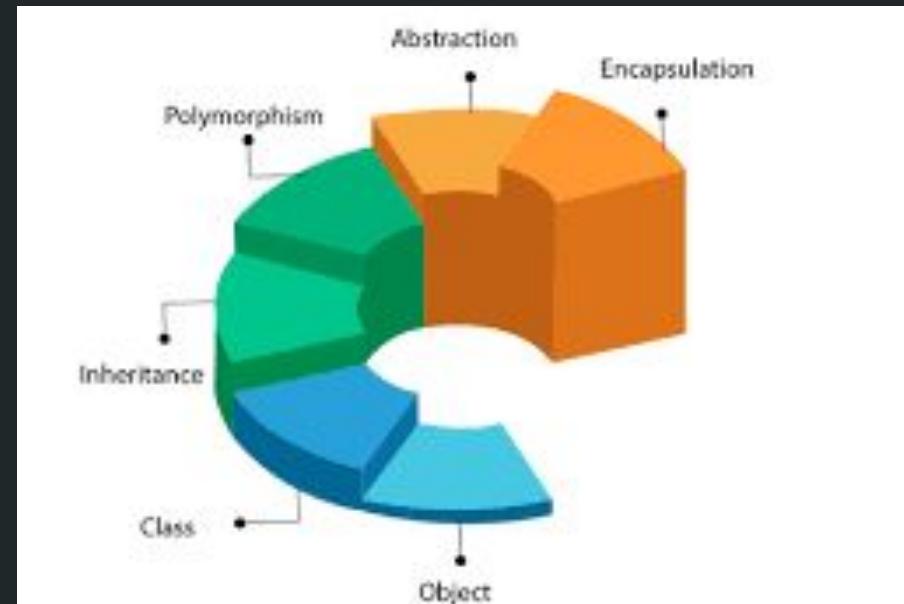
Procedure-Oriented Programming

Another serious concern with the procedural approach is that we do not model real world problems very well. This is because functions are action-oriented and do not really correspond to the element of the problem.

Some Characteristics exhibited by procedure-oriented programming are:

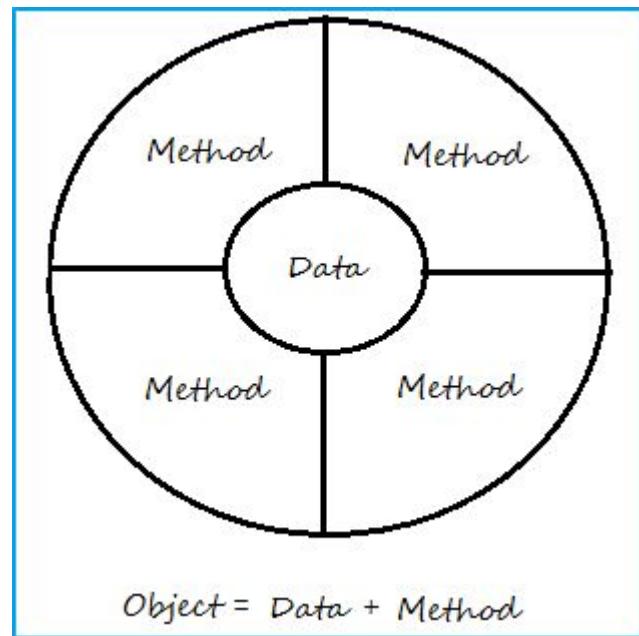
- Emphasis is on doing things (algorithms).
- Large programs are divided into smaller programs known as functions.
- Most of the functions share global data.
- Data move openly around the system from function to function.
- Functions transform data from one form to another.
- Employs top-down approach in program design.

Object Oriented Paradigm



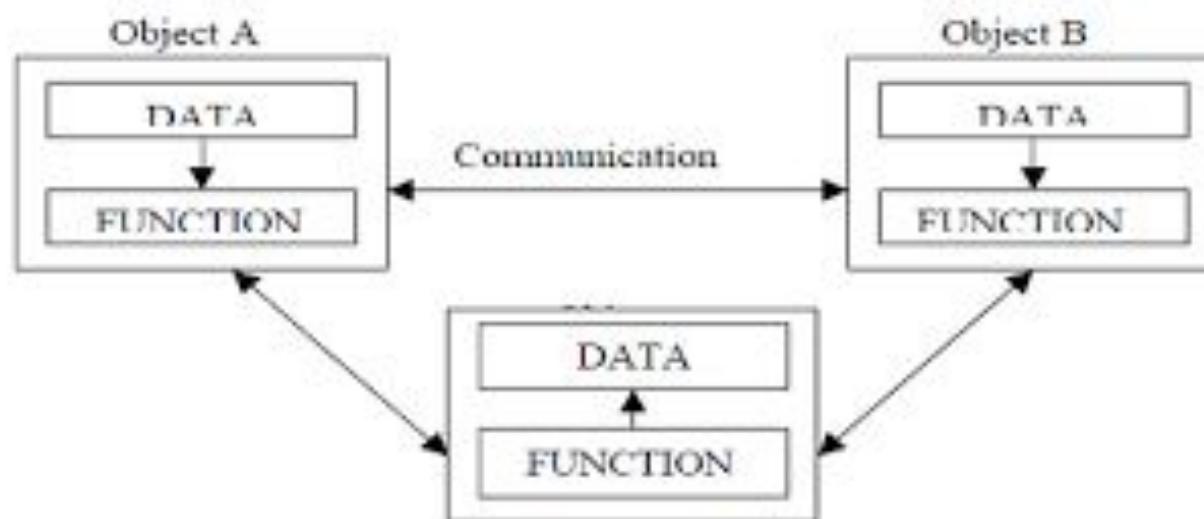
Object-Oriented Paradigm

- The major motivating factor in the invention of object-oriented approach is to remove some of the flaws encountered in the procedural approach.
- OOP treats data as a critical element in the program development and does not allow it to flow freely around the system. It ties data more closely to the function that operate on it, and protects it from accidental modification from outside function.
- OOP allows decomposition of a problem into a number of entities called objects and then builds data and function around these objects.



Object-Oriented Paradigm

The data of an object can be accessed only by the function associated with that object. However, function of one object can access the function of other objects.



Organization of data and function in OOP

Object-Oriented Paradigm

Some of the features of object oriented programming are:

- Emphasis is on data rather than procedure.
- Programs are divided into what are known as objects.
- Data structures are designed such that they characterize the objects.
- Functions that operate on the data of an object are tied together in the data structure.
- Data is hidden and cannot be accessed by external function.
- Objects may communicate with each other through function.
- New data and functions can be easily added whenever necessary.
- Follows bottom up approach in program design.

Object-oriented programming is the most recent concept among programming paradigms and still means different things to different people.

O

Object

O

Oriented

P

Programming

Object Oriented Programming

Object-oriented programming is an approach that provided a way of modularizing programs by creating partitioned memory area for both data and functions that can be used as templates for creating copies of such modules on demand.

This means that an object is considered to be a partitioned area of computer memory that stores data and a set of operations that can access the data.

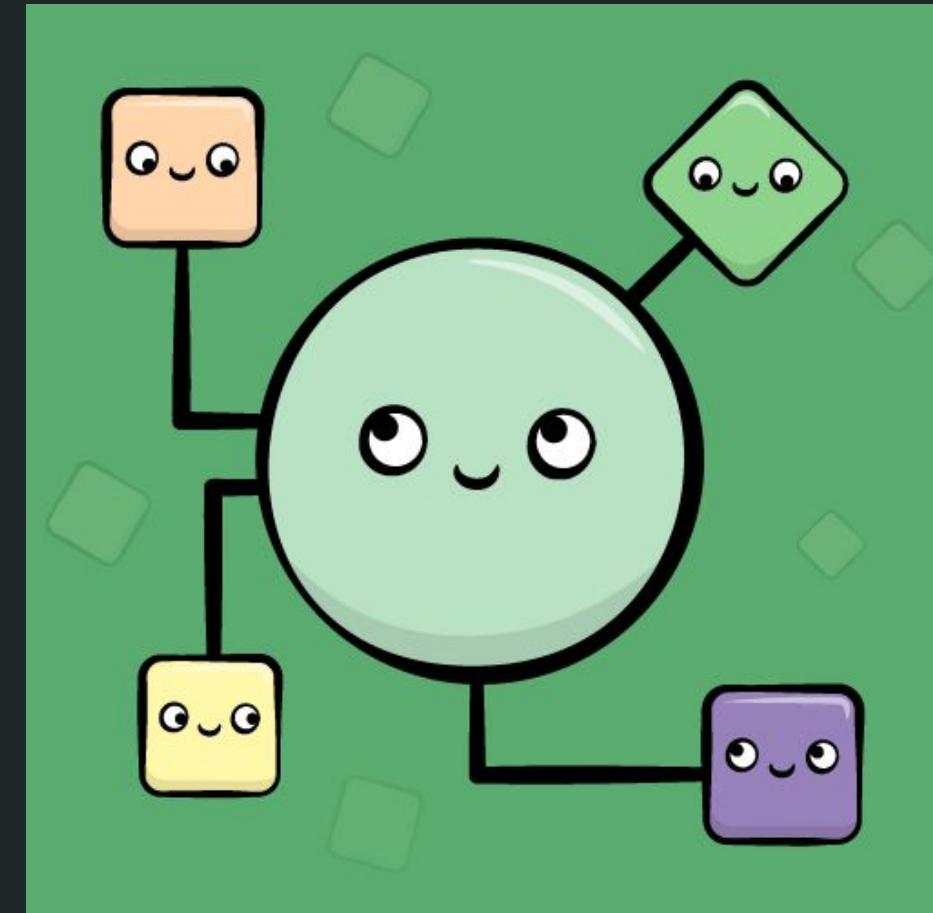
Basics Concepts

Object Oriented Programming



- Objects
- Classes
- Data abstraction and encapsulation
- Inheritance
- Polymorphism
- Dynamic binding
- Message passing

Objects



Objects

- Objects are the basic run time entities in an object-oriented system.
- They may represent a person, a place, a bank account, a table of data or any item that the program has to handle. They may also represent user-defined data such as vectors, time and lists.
- Programming problem is analyzed in term of objects and the nature of communication between them. Program objects should be chosen such that they match closely with the real-world objects.
- Objects take up space in the memory and have an associated address like a record in Pascal, or a structure in c.

Objects

- When a program is executed, the objects interact by sending messages to one another.
- For example, if “customer” and “account” are to object in a program, then the customer object may send a message to the count object requesting for the bank balance.
- Each object contain data, and code to manipulate data. Objects can interact without having to know details of each other’s data or code. It is sufficient to know the type of message accepted, and the type of response returned by the objects.

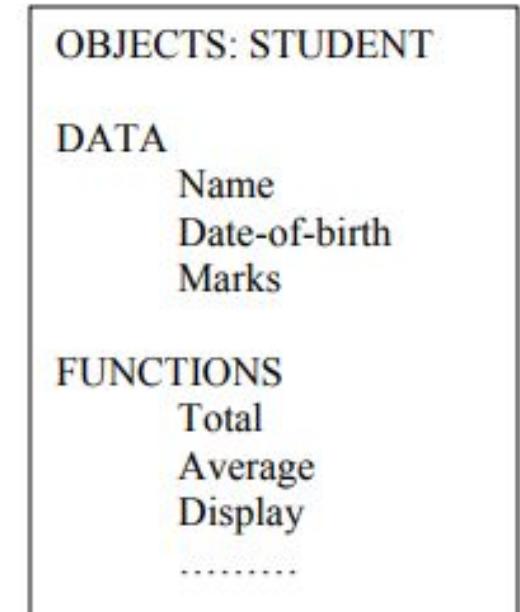


Fig. 1.5 representing an object

Objects

Object is a real world entity with specific features and behaviour. Objects in a program are generally used to model the real-world items.

- Object can be tangible like person, place, car, house and tree etc.
- It can be logical entity such as date of birth, job profile, bank account etc.

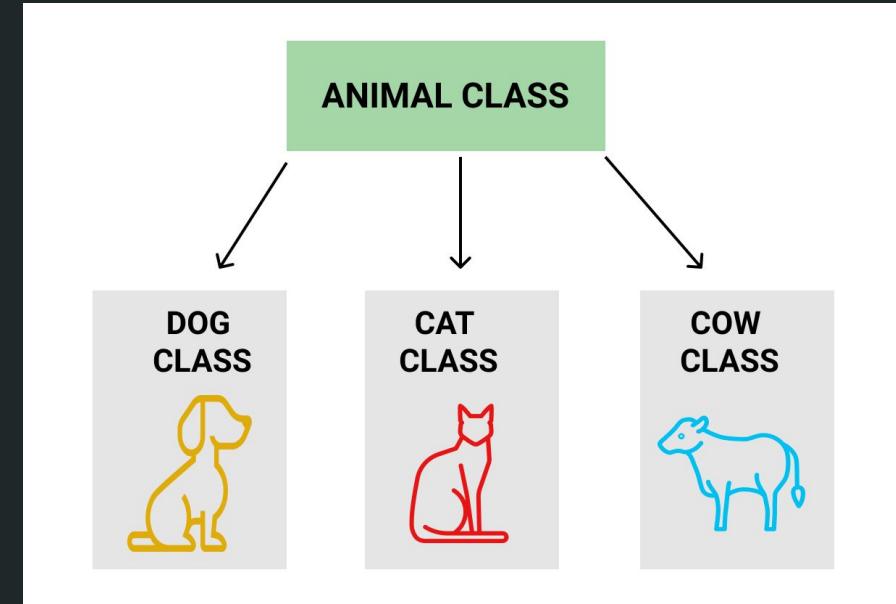
An object is defined with two major items

- Attributes of object, Examples name, colour, shape, size or other features
- Behaviour of object defined by operations or functions, Examples read, drive, display, update_balance etc.

Objects

Name	Attributes/ Data	Behaviour / Functions
Person	Name, age, DOB, phone	Walk(), sleep(), work(), work()
Car	Type, make, model, gear	Start(), stop(), setspeed(), setgear()
Time	Hours, min, sec	Set_time(), current_time(), get_time()

Classes



Classes

- We just mentioned that objects contain data, and code to manipulate that data.
- The entire set of data and code of an object can be made a user-defined data type with the help of class.
- In fact, objects are variables of the type class. Once a class has been defined, we can create any number of objects belonging to that class.
- Each object is associated with the data of type class with which they are created. A class is thus a collection of objects similar types.
- For examples, Mango, Apple and orange members of class fruit. Classes are user-defined that types and behave like the built-in types of a programming language.

Classes

The real world objects, such as television have many features and properties. For example every television is a complete unit in itself even if it is attached to some other device like DVD player it remains a television.

There is an interface like buttons and remote control to operate television. All televisions have some common properties and operations. Collection of similar objects is known as class. And instances of that class are called objects.

- Name of class – television
- Data – brand, model, type, volume_level, channel_number etc.
- Functions – switch_on(), switch_off(), change_channel(), set_volume() etc.

Classes

- Class is a collection of similar objects.
- Class defines all the properties (data and functions) of an object.
- Class provides a well-defined interface to use the functions of that class.
- A class must be complete and well-documented.
- The programming code of a class should be robust

Class	Object
user defined data type	variable of type class
group of similar objects	Instance of a class
Class defines properties i.e. data and function members	Objects hold actual values
Blue print of the object	Number of objects are created using class

Classes

Class – television with attributes as brand_name, model, size and colour.

Object1 - commonhall_tv with values as Sony, 2015, 40”, black

Object2- room_tv with values as Samsung, 2016, 32”, white

Object3 - kitchen_tv with values as Sony, 2014, 22”, blue

Data Abstraction and Encapsulation



Data Abstraction and Encapsulation

- The wrapping up of data and function into a single unit (called class) is known as encapsulation.
- Data and encapsulation is the most striking feature of a class. The data is not accessible to the outside world, and only those functions which are wrapped in the class can access it.
- These functions provide the interface between the object's data and the program. This insulation of the data from direct access by the program is called data hiding or information hiding.

Data Abstraction and Encapsulation

- Abstraction refers to the act of representing essential features without including the background details or explanation.
- Classes use the concept of abstraction and are defined as a list of abstract attributes such as size, wait, and cost, and function operate on these attributes.
- They encapsulate all the essential properties of the object that are to be created. The attributes are some time called data members because they hold information. The functions that operate on these data are sometimes called methods or member function.

Data Abstraction and Encapsulation

Abstraction means providing only the essential or relevant information and hiding the background or technical implementation details of something.

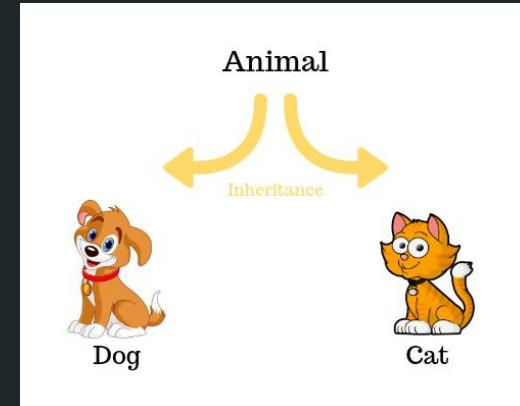
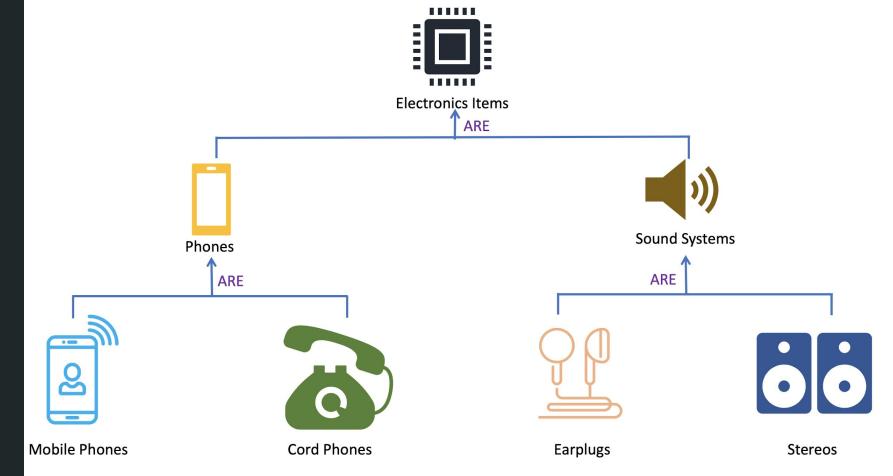
For example in the class TV

- Interface part is through remote buttons (i.e. visible part methods like Changechannel(), Change-volume(), Power-on())
- Implementation part is mechanism behind these switches (i.e. hidden part methods like Decode-signal(), Display-picture())

Data Abstraction and Encapsulation

Abstraction	Encapsulation
Is about identifying necessary components required to build a system	Is about hiding the internal complexities of a system to make it user friendly
Abstraction takes place at design level as it focuses on what should be implemented in the system	Encapsulation happens at implementation level as it focuses on how it should be done or actually implemented
Abstraction is achieved through encapsulation	Encapsulation is achieved by hiding data and information in class

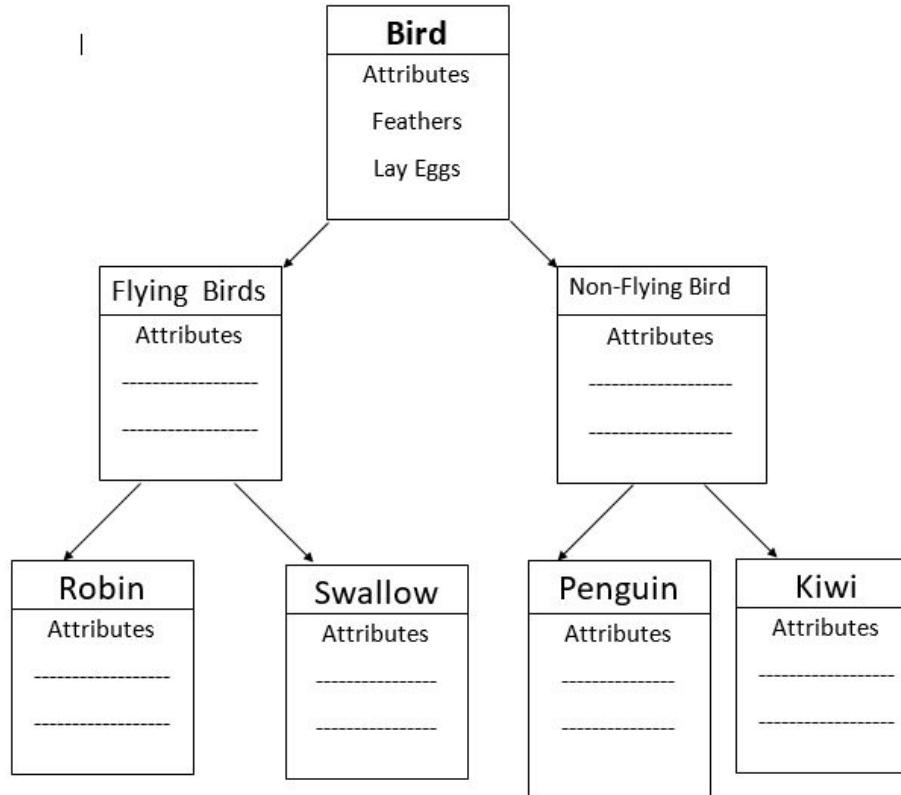
Inheritance



Inheritance

- Inheritance is the process by which objects of one class acquired the properties of objects of another classes.
- It supports the concept of hierarchical classification. For example, the bird, ‘robin’ is a part of class ‘flying bird’ which is again a part of the class ‘bird’.
- The principal behind this sort of division is that each derived class shares common characteristics with the class from which it is derived.

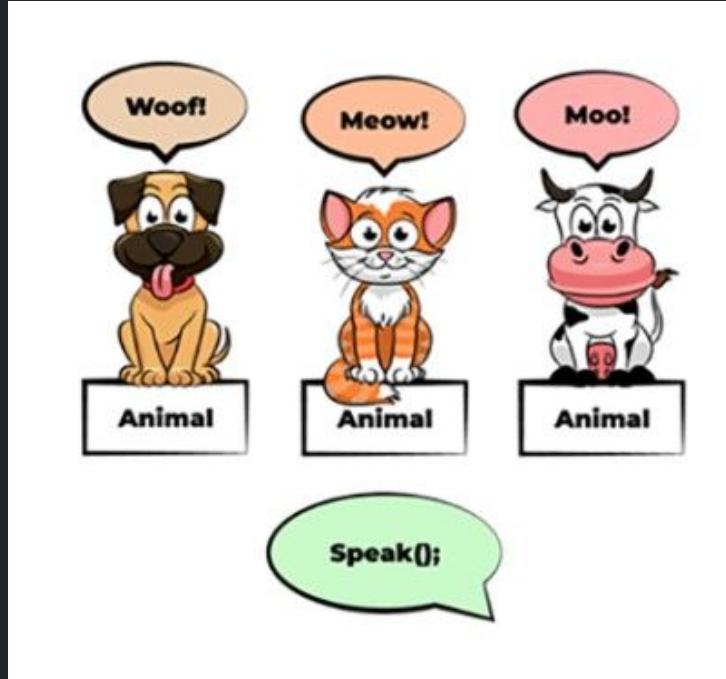
Inheritance



Inheritance

- In OOP, the concept of inheritance provides the idea of reusability.
- This means that we can add additional features to an existing class without modifying it. This is possible by deriving a new class from the existing one. The new class will have the combined feature of both the classes.
- The real appeal and power of the inheritance mechanism is that it allows the programmer to reuse a class i.e almost, but not exactly, what he wants, and to tailor the class in such a way that it does not introduce any undesirable side-effects into the rest of classes.

Polymorphism

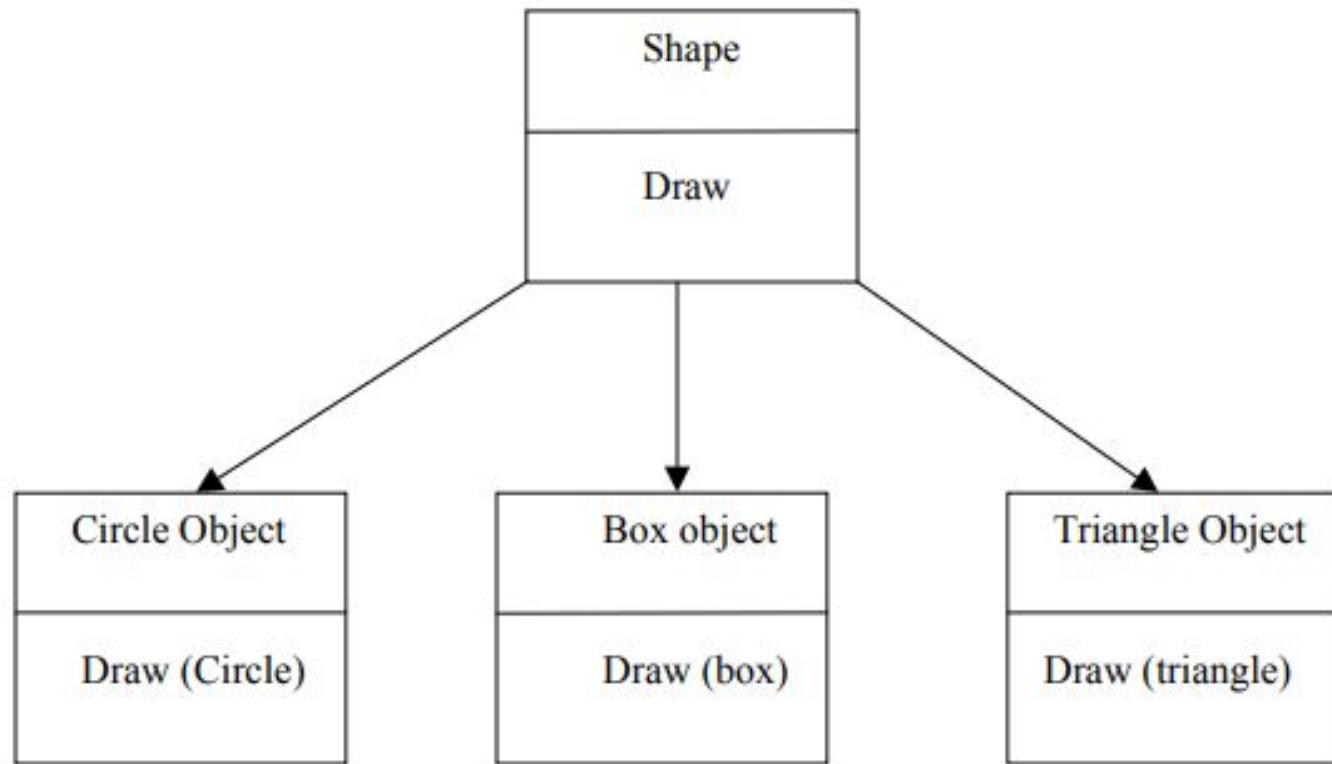


Polymorphism

- Polymorphism is another important OOP concept. Polymorphism, a Greek term, means the ability to take more than one form.
- An operation may exhibit different behavior in different instances.
- The behavior depends upon the types of data used in the operation.
For example, consider the operation of addition. For two numbers, the operation will generate a sum. If the operands are strings, then the operation would produce a third string by concatenation.
- The process of making an operator to exhibit different behaviors in different instances is known as operator overloading.

Polymorphism

Fig. illustrates that a single function name can be used to handle different number and different types of argument. This is something similar to a particular word having several different meanings depending upon the context. Using a single function name to perform different type of task is known as function overloading.



Polymorphism

- Polymorphism plays an important role in allowing objects having different internal structures to share the same external interface.
- This means that a general class of operations may be accessed in the same manner even though specific action associated with each operation may differ. Polymorphism is extensively used in implementing inheritance.

Dynamic Binding

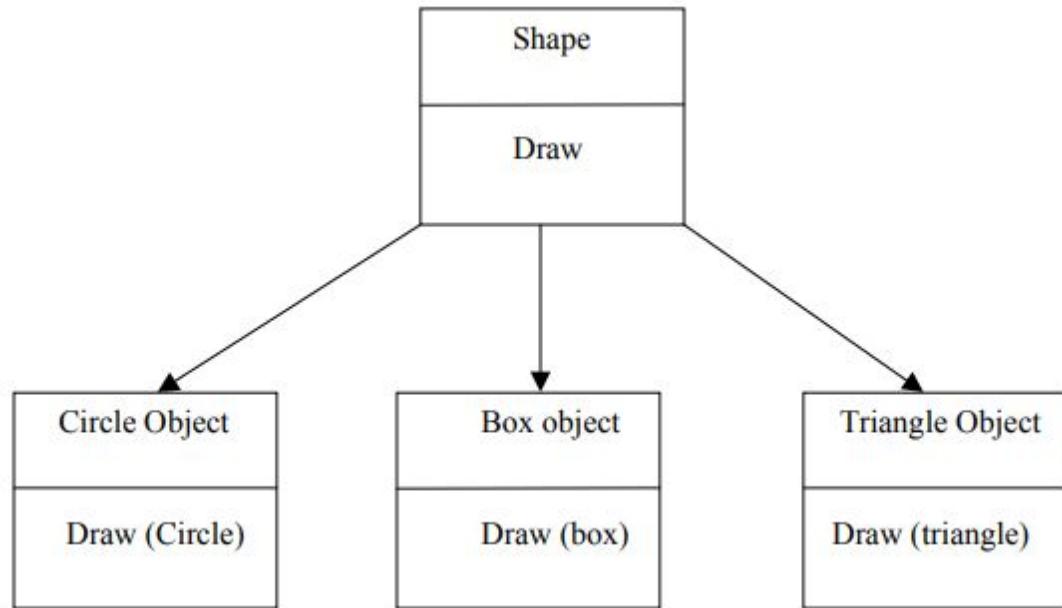


Dynamic Binding

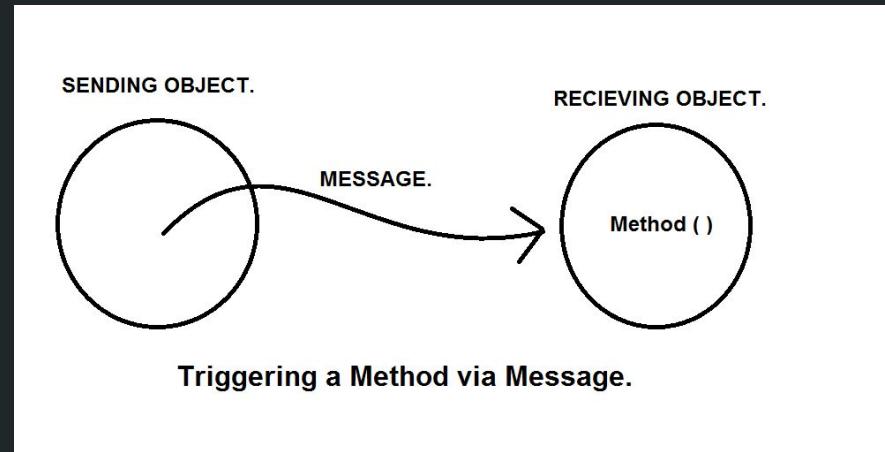
- Binding refers to the linking of a procedure call to the code to be executed in response to the call.
- Dynamic binding means that the code associated with a given procedure call is not known until the time of the call at run time.
- It is associated with polymorphism and inheritance.
- A function call associated with a polymorphic reference depends on the dynamic type of that reference.

Dynamic Binding

- Consider the procedure “draw” in fig.
- By inheritance, every object will have this procedure.
- Its algorithm is, however, unique to each object and so the draw procedure will be redefined in each class that defines the object. At run-time, the code matching the object under current reference will be called.



Message Passing



Message Passing

An object-oriented program consists of a set of objects that communicate with each other.

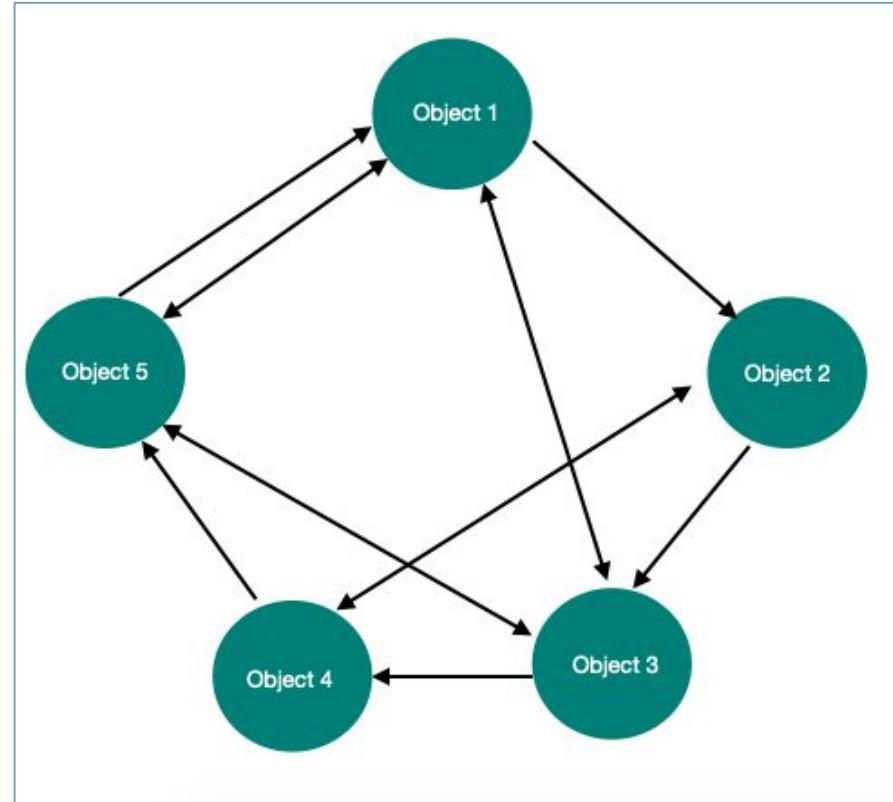
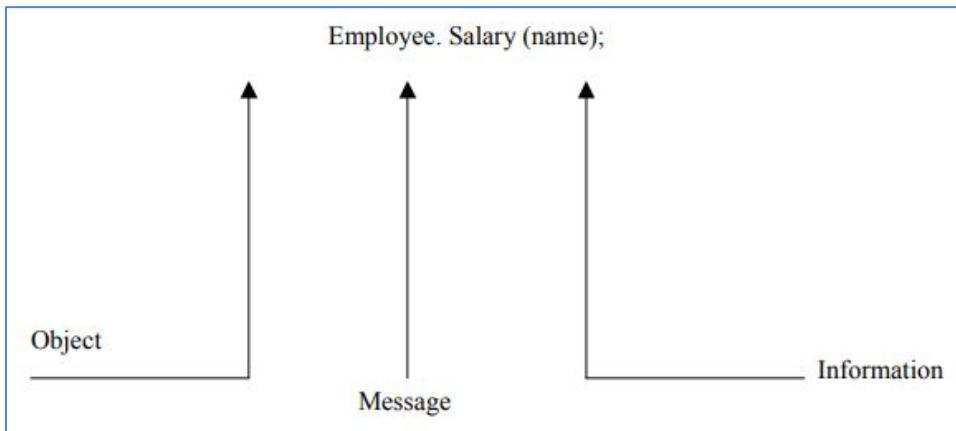
The process of programming in an object-oriented language, involves the following basic steps:

- 1. Creating classes that define object and their behavior,**
- 2. Creating objects from class definitions, and**
- 3. Establishing communication among objects.**

Message Passing

- Objects communicate with one another by sending and receiving information much the same way as people pass messages to one another.
- The concept of message passing makes it easier to talk about building systems that directly model or simulate their real-world counterparts.
- A Message for an object is a request for execution of a procedure, and therefore will invoke a function (procedure) in the receiving object that generates the desired results.
- Message passing involves specifying the name of object, the name of the function (message) and the information to be sent.

Message Passing



Benefits of OOP



Benefits of OOP

- Through inheritance, we can eliminate redundant code extend the use of existing Classes.
- We can build programs from the standard working modules that communicate with one another, rather than having to start writing the code from scratch. This leads to saving of development time and higher productivity.
- The principle of data hiding helps the programmer to build secure program that can not be invaded by code in other parts of a programs.
- It is possible to have multiple instances of an object to co-exist without any interference.
- It is possible to map object in the problem domain to those in the program.

Benefits of OOP

- It is easy to partition the work in a project based on objects.
- The data-centered design approach enables us to capture more detail of a model can implemental form.
- Object-oriented system can be easily upgraded from small to large system.
- Message passing techniques for communication between objects makes to interface descriptions with external systems much simpler.
- Software complexity can be easily managed.

Benefits of OOP

- While it is possible to incorporate all these features in an object-oriented system, their importance depends on the type of the project and the preference of the programmer.
- There are a number of issues that need to be tackled to reap some of the benefits stated above.
- For instance, object libraries must be available for reuse. The technology is still developing and current product may be superseded quickly. Strict controls and protocols need to be developed if reuse is not to be compromised.

Benefits of OOP

- While it is possible to incorporate all these features in an object-oriented system, their importance depends on the type of the project and the preference of the programmer.
- There are a number of issues that need to be tackled to reap some of the benefits stated above. For instance,
 - Object libraries must be available for reuse.
 - The technology is still developing and current product may be superseded quickly.
 - Strict controls and protocols need to be developed if reuse is not to be compromised.

Object Oriented Languages

A language that is specially designed to support the OOP concepts makes it easier to implement them.

The languages should support several of the OOP concepts to claim that they are object-oriented. Depending upon the features they support, they can be classified into the following two categories:

1. Object-based programming languages,
2. Object-oriented programming languages.

Object-based programming

Object-based programming is the style of programming that primarily supports encapsulation and object identity.

Major feature that are required for object based programming are:

- Data encapsulation
- Data hiding and access mechanisms
- Automatic initialization and clear-up of objects
- Operator overloading

Languages that support programming with objects are said to the objects-based programming languages. They do not support inheritance and dynamic binding. Ada is a typical object-based programming language.

Object-oriented programming

Object-oriented programming language incorporates all of object-based programming features along with two additional features, namely, inheritance and dynamic binding. Object-oriented programming can therefore be characterized by the following statements:

Object-based features + inheritance + dynamic binding

Application of OOP



Application of OOP

Hundreds of windowing systems have been developed, using the OOP techniques. Real-business system are often much more complex and contain many more objects with complicated attributes and method. OOP is useful in these types of application because it can simplify a complex problem. The promising areas of application of OOP include:

- Real-time system
- Simulation and modeling
- Object-oriented databases
- Hypertext, Hypermedia, and expertext
- AI and expert systems
- Neural networks and parallel programming
- Decision support and office automation systems
- CIM/CAM/CAD systems

Object-oriented technology is certainly going to change the way the software engineers think, analyze, design and implement future system.

THE
END

Introduction of JAVA

Java is a programming language and a platform. Java is a high level, robust, object-oriented and secure programming language.

Introduction of JAVA

- Java programming language was originally developed by Sun Microsystems which was initiated by James Gosling and released in 1995 as core component of Sun Microsystems' Java platform (Java 1.0 [J2SE]).
- Java runs on a variety of platforms, such as Windows, Mac OS, and the various versions of UNIX.
- The latest release of the Java Standard Edition is Java SE 8. With the advancement of Java and its widespread popularity, multiple configurations were built to suit various types of platforms.
- For example: J2EE for Enterprise Applications, J2ME for Mobile Applications.

Introduction of JAVA

- The new J2 versions were renamed as Java SE, Java EE, and Java ME respectively. Java is guaranteed to be **Write Once, Run Anywhere.**

Features of JAVA

- **Object Oriented** – In Java, everything is an Object. Java can be easily extended since it is based on the Object model.
- **Platform Independent** – Unlike many other programming languages including C and C++, when Java is compiled, it is not compiled into platform specific machine, rather into platform independent byte code. This byte code is distributed over the web and interpreted by the Virtual Machine (JVM) on whichever platform it is being run on.
- **Simple** – Java is designed to be easy to learn. If you understand the basic concept of OOP Java, it would be easy to master.
- **Secure** – With Java's secure feature it enables to develop virus-free, tamper-free systems. Authentication techniques are based on public-key encryption.

Features of JAVA

- **Architecture-neutral** – Java compiler generates an architecture-neutral object file format, which makes the compiled code executable on many processors, with the presence of Java runtime system.
- **Portable** – Being architecture-neutral and having no implementation dependent aspects of the specification makes Java portable. Compiler in Java is written in ANSI C with a clean portability boundary, which is a POSIX subset (IEEE Standard).
- **Robust** – Java makes an effort to eliminate error prone situations by emphasizing mainly on compile time error checking and runtime (dynamic) checking. (eg Java has a pointer model that eliminates the possibility of overwriting memory and corrupting data.)

Features of JAVA

- **Multithreaded** – With Java's multithreaded feature it is possible to write programs that can perform many tasks simultaneously. This design feature allows the developers to construct interactive applications that can run smoothly.
- **Interpreted** – Java byte code is translated on the fly to native machine instructions and is not stored anywhere. The development process is more rapid and analytical since the linking is an incremental and light-weight process.
- **High Performance** – With the use of Just-In-Time compilers, Java enables high performance.

Features of JAVA

- **Distributed** – Java is designed for the distributed environment of the internet.
- **Dynamic** – Java is considered to be more dynamic than C or C++ since it is designed to adapt to an evolving environment. Java programs can carry extensive amount of run-time information that can be used to verify and resolve accesses to objects on run-time.

History of Java

- James Gosling initiated Java language project in June 1991 for use in one of his many set-top box projects. The language, initially called ‘Oak’ after an oak tree that stood outside Gosling's office, also went by the name ‘Greentalk’ and ended up later being renamed as Java, from a list of random words.
- Sun released the first public implementation as Java 1.0 in 1995. It promised **Write Once, Run Anywhere** (WORA), providing no-cost run-times on popular platforms.
- On 13 November, 2006, Sun released much of Java as free and open source software under the terms of the GNU General Public License (GPL).
- On 8 May, 2007, Sun finished the process, making all of Java's core code free and open-source, aside from a small portion of code to which Sun did not hold the copyright.

How JAVA differs from C & C++

Java and C

- JAVA is Object-Oriented while C is procedural.
- Java is an Interpreted language while C is a compiled language.
- C is a low-level language while JAVA is a high-level language
- C uses the top-down **{sharp & smooth}** approach while JAVA uses the bottom-up **{on the rocks}** approach.
- The Memory Management (Garbage Collection) with JAVA & The User-Based Memory Management in C.
- Unlike C, JAVA does not support Preprocessors, & does not really them.
- Exception Handling in JAVA and the errors & crashes in C.

How JAVA differs from C & C++

Java and C++

- C++ supports multiple inheritance. Java doesn't support multiple inheritance through class. It can be achieved by interfaces in java.
- C++ is mainly used for system programming. Java is mainly used for application programming. It is widely used in window, web-based, enterprise and mobile applications.
- C++ supports operator overloading. Java doesn't support operator overloading.
- C++ uses compiler only. Java uses compiler and interpreter both.

How JAVA differs from C & C++

Java and C++

- C++ supports both call by value and call by reference. Java supports call by value only. There is no call by reference in java.
- C++ supports structures and unions. Java doesn't support structures and unions.
- C++ doesn't support documentation comment. Java supports documentation comment (`/** ... */`) to create documentation for java source code.
- Java Doesn't have Destructor like C++ Instead Java Has finalize Method

Application

- Desktop Applications such as acrobat reader, media player, antivirus, etc.
- Web Applications such as irctc.co.in, javatpoint.com, etc.
- Enterprise Applications such as banking applications.
- Mobile
- Embedded System
- Smart Card
- Robotics
- Games, etc.

Types of Java Applications

- Standalone Application
- Web Application
- Enterprise Application
- Mobile Application

Java Platforms / Editions

- Java SE (Java Standard Edition)
- Java EE (Java Enterprise Edition)
- Java ME (Java Micro Edition)
- JavaFX

Java Environment

JDK contains

- Appletviewer (for viewing applets)
- javac (Java Compiler)
- java (Java Interpreter)
- javap (Java disassembler)
- javah (for C header files)
- javadoc (creating HTML documents)
- jdb (Java Debugger)

Tools

- you will need a Pentium 200-MHz computer with a minimum of 64 MB of RAM (128/512 MB of RAM recommended).
- You will also need the following softwares –
 - Linux 7.1 or Windows xp/7/8/10 operating system
 - Java JDK 8 or higher
 - Microsoft Notepad or any other text editor

First Java Program

```
public class MyFirstJavaProgram
{
    public static void main(String []args)
    {
        System.out.println("Hello World");
    }
}
```

Editors

- **Popular Java Editors**
- To write your Java programs, you will need a text editor. There are even more sophisticated IDEs available in the market. But for now, you can consider one of the following
 - **Notepad** – On Windows machine, you can use any simple text editor like Notepad (Recommended for this tutorial), TextPad.
 - **Netbeans** – A Java IDE that is open-source and free which can be downloaded from
<https://www.netbeans.org/index.html>.
 - **Eclipse** – A Java IDE developed by the eclipse open-source community and can be downloaded from
<https://www.eclipse.org/>

Fundamental Concepts of JAVA

- Java is an Object-Oriented Language. As a language that has the Object-Oriented feature, Java supports the following fundamental concepts –
- Polymorphism
- Inheritance
- Encapsulation
- Abstraction
- Classes
- Objects
- Dynamic Binding
- Message Passing

Fundamental Concepts of JAVA

- **Object** – Objects have states and behaviors. Example: A dog has states - color, name, breed as well as behaviors – wagging the tail, barking, eating. An object is an instance of a class.
- **Class** – A class can be defined as a template/blueprint that describes the behavior/state that the object of its type support.

Fundamental Concepts of JAVA

- **Classes in Java**
- A class is a blueprint from which individual objects are created.

```
public class Dog
{
    String breed;    int age;      String color;
    void barking()
    {
    }
    void hungry()
    {
    }
    void sleeping()
    {
    }
}
```

Fundamental Concepts of JAVA

- A class can contain any of the following variable types.
- **Local variables** – Variables defined inside methods, constructors or blocks are called local variables. The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed.
- **Instance variables** – Instance variables are variables within a class but outside any method. These variables are initialized when the class is instantiated. Instance variables can be accessed from inside any method, constructor or blocks of that particular class.

Fundamental Concepts of JAVA

- **Class variables** – Class variables are variables declared within a class, outside any method, with the static keyword.

Constructors

- When discussing about classes, one of the most important sub topic would be constructors. Every class has a constructor. If we do not explicitly write a constructor for a class, the Java compiler builds a default constructor for that class.
- Each time a new object is created, at least one constructor will be invoked. The main rule of constructors is that they should have the same name as the class. A class can have more than one constructor.

Object

- **Creating an Object**
- An object is created from a class. In Java, the new keyword is used to create new objects.
- There are three steps when creating an object from a class –
 - **Declaration** – A variable declaration with a variable name with an object type.
 - **Instantiation** – The 'new' keyword is used to create the object.
 - **Initialization** – The 'new' keyword is followed by a call to a constructor. This call initializes the new object.

Object

- **Accessing Instance Variables and Methods**
- Instance variables and methods are accessed via created objects. To access an instance variable, following is the fully qualified path –

```
/* First create an object */
```

```
ObjectReference = new Constructor();
```

```
/* Now call a variable as follows */
```

```
ObjectReference.variableName;
```

```
/* Now you can call a class method as follows */
```

```
ObjectReference.MethodName();
```

- Java is a whole platform, with a huge library, containing lots of reusable code, and an execution environment that provides services such as security, portability across operating systems, and automatic garbage collection.
- Any hardware or software environment in which a program runs, is known as a platform. Since Java has a runtime environment (JRE) and API, it is called a platform.

- Java has everything
 - a good language, a high-quality execution environment, and a vast library. That combination is what makes Java an irresistible proposition to so many programmers.

DATA TYPES & VARIABLES

DATA TYPES

Java defines eight simple types:

- 1) byte - 8-bit integer type
- 2) short - 16-bit integer type
- 3) int - 32-bit integer type
- 4) long - 64-bit integer type
- 5) float - 32-bit floating-point type
- 6) double - 64-bit floating-point type
- 7) char - symbols in a character set
- 8) boolean - logical values true and false

DATA TYPES

byte: 8-bit integer type.

Range: -128 to 127.

Example: byte b = -15;

Usage: particularly when working with data streams.

short: 16-bit integer type.

Range: -32768 to 32767.

Example: short c = 1000;

Usage: probably the least used simple type.

DATA TYPES

int: 32-bit integer type.

Range: -2147483648 to 2147483647.

Example: int b = -50000;

Usage:

- 1) Most common integer type.
- 2) Typically used to control loops and to index arrays.
- 3) Expressions involving the byte, short and int values are promoted to int before calculation.

DATA TYPES

`long`: 64-bit integer type.

Range: -9223372036854775808
to
9223372036854775807.

Example: `long l =
10000000000000000000;`

Usage: useful when `int` type
is not large enough to hold
the desired value

`float`: 32-bit floating-point
number.

Range: `1.4e-045` to `3.4e+038`.

Example: `float f = 1.5;`

Usage:

- 1) fractional part is needed
- 2) large degree of precision
is not required

DATA TYPES

double: 64-bit floating-point number.

Range: 4.9e-324 to 1.8e+308.

Example: double pi = 3.1416;

Usage:

- 1) accuracy over many iterative calculations
- 2) manipulation of large-valued numbers

DATA TYPES

char: 16-bit data type used to store characters.

Range: 0 to 65536.

Example: char c = 'a';

Usage:

- 1) Represents both ASCII and Unicode character sets;
Unicode defines a character set with characters found in
(almost) all human languages.
- 2) Not the same as in C/C++ where char is 8-bit and
represents ASCII only

DATA TYPES

boolean: Two-valued type of logical values.

Range: values true and false.

Example: boolean b = (1<2);

Usage:

- 1) returned by relational operators, such as 1<2
- 2) required by branching expressions such as if or for

VARIABLES

VARIABLES

declaration – how to assign a type to a variable

initialization – how to give an initial value to a variable

scope – how the variable is visible to other parts of the program

lifetime – how the variable is created, used and destroyed

type conversion – how Java handles automatic type conversion

type casting – how the type of a variable can be narrowed down

VARIABLES

- Java uses variables to store data.
- To allocate memory space for a variable JVM requires:
 - 1) to specify the data type of the variable
 - 2) to associate an identifier with the variable
 - 3) optionally, the variable may be assigned an initial value

All done as part of variable declaration.

BASIC VARIABLE DECLARATION

datatype identifier [=value];

- datatype must be
 - A simple datatype
 - User defined datatype (class type)
- Identifier is a recognizable name confirm to identifier rules
- Value is an optional initial value.

VARIABLE DECLARATION

We can declare several variables at the same time:

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

Examples:

```
int a, b, c;
```

```
int d = 3, e, f = 5;
```

```
byte g = 22;
```

```
double pi = 3.14159;
```

```
char ch = 'x';
```

VARIABLE SCOPE

- Scope determines the visibility of program elements with respect to other program elements.
- In Java, scope is defined separately for classes and methods:
 - 1) variables defined by a class have a global scope
 - 2) variables defined by a method have a local scope
- A scope is defined by a block: { ... }
- A variable declared inside the scope is not visible outside:

```
{ int n; } n = 1;// this is illegal
```

VARIABLE LIFETIME

- Variables are created when their scope is entered by control flow and destroyed when their scope is left:
- A variable declared in a method will not hold its value between different invocations of this method.
- A variable declared in a block loses its value when the block is left.
- Initialized in a block, a variable will be reinitialized with every re-entry. Variables lifetime is confined to its scope!

ARRAYS

ARRAY

An array is a group of liked-typed variables referred to by a common name, with individual variables accessed by their index.

Arrays are:

- 1) declared
- 2) created
- 3) initialized
- 4) used

Also, arrays can have one or several dimensions.

ARRAY DECLARATION

Array declaration involves:

- 1) declaring an array identifier
- 2) declaring the number of dimensions
- 3) declaring the data type of the array elements

Two styles of array declaration:

type array-variable[];

or

type [] array-variable;

ARRAY CREATION

- After declaration, no array actually exists.
- In order to create an array, we use the new operator:

```
type array-variable[];  
array-variable = new type[size];
```

- This creates a new array to hold size elements of type type, which reference will be kept in the variable array-variable.

ARRAY INDEXING

- Later we can refer to the elements of this array through their indexes:
`array-variable[index]`
- The array index always starts with zero!
- The Java run-time system makes sure that all array indexes are in the correct range, otherwise raises a runtime error.

ARRAY INITIALIZATION

- Arrays can be initialized when they are declared:

```
int monthDays[] =  
{31,28,31,30,31,30,31,31,30,31,30,31};
```

Note:

- 1) there is no need to use the new operator
- 2) the array is created large enough to hold all specified elements

MULTIDIMENSIONAL ARRAYS

Multidimensional arrays are arrays of arrays:

1) declaration: int array[][];

2) creation: int array = new int[2][3];

3) initialization

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

Operators in java

- Arithmetic Operators
- Increment Decrement Operators
- Relational Operators
- Bitwise Operators
- Logical Operators
- Assignment Operators
- Misc (Special)Operators

Arithmetic Operators

Operator	Description	Example
+	Addition - Adds values on either side of the operator	A + B will give 30
-	Subtraction - Subtracts right hand operand from left hand operand	A - B will give -10
*	Multiplication - Multiplies values on either side of the operator	A * B will give 200
/	Division - Divides left hand operand by right hand operand	B / A will give 2
%	Modulus - Divides left hand operand by right hand operand and returns remainder	B % A will give 0

Increment/Decrement Operators

Operator	Description	Example
<code>++</code>	Increment - Increase the value of operand by 1	<code>B++</code> gives 21
<code>--</code>	Decrement - Decrease the value of operand by 1	<code>B--</code> gives 19

```
int a = 10;  
int d = 25;  
System.out.println("a++ = " + (a++));  
System.out.println("b-- = " + (a--)); // Check the difference in d++ and ++d  
System.out.println("d++ = " + (d++));  
System.out.println("++d = " + (++d));
```

Output:-

```
a++ = 10  
b-- = 11  
d++ = 25  
++d = 27
```

Relational Operators

Assume variable A holds 10 and variable B holds 20 then:

Operator	Description	Example
<code>==</code>	Checks if the value of two operands are equal or not, if yes then condition becomes true.	$(A == B)$ is not true.
<code>!=</code>	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	$(A != B)$ is true.
<code>></code>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	$(A > B)$ is not true.
<code><</code>	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	$(A < B)$ is true.
<code>>=</code>	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	$(A >= B)$ is not true.
<code><=</code>	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	$(A <= B)$ is true.

Bitwise Operator

- Java defines several bitwise operators which can be applied to the integer types, long, int, short, char, and byte.
- Bitwise operator works on bits and perform bit by bit operation.
Assume if

a = 60;

b = 13;

- Now in binary format they will be as follows:

a = 0011 1100

b = 0000 1101

- $a \& b = 0000\ 1100$
- $a | b = 0011\ 1101$
- $a ^ b = 0011\ 0001$
- $\sim a = 1100\ 0011$

Bitwise Operators

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
	Binary OR Operator copies a bit if it exists in either operand.	(A B) will give 61 which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -60 which is 1100 0011
<<	Binary Left Shift Operator. The left operand's value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operand's value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 1111
>>>	Shift right zero fill operator. The left operand's value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros.	A >>>2 will give 15 which is 0000 1111

Logical Operators

Assume boolean variables A holds true and variable B holds false then

Operator	Description	Example
<code>&&</code>	Called Logical AND operator. If both the operands are non zero then then condition becomes true.	$(A \&\& B)$ is false.
<code> </code>	Called Logical OR Operator. If any of the two operands are non zero then then condition becomes true.	$(A B)$ is true.
<code>!</code>	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	$!(A \&\& B)$ is true.

Assignment Operators

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	$C = A + B$ will assigne value of $A + B$ into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	$C += A$ is equivalent to $C = C + A$
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	$C -= A$ is equivalent to $C = C - A$
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	$C *= A$ is equivalent to $C = C * A$
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	$C /= A$ is equivalent to $C = C / A$

Assignment Operators

Operator	Description	Example
<code>%=</code>	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	<code>C %= A</code> is equivalent to <code>C = C % A</code>
<code><=></code>	Left shift AND assignment operator	<code>C <=> 2</code> is same as <code>C = C << 2</code>
<code>>=></code>	Right shift AND assignment operator	<code>C >=> 2</code> is same as <code>C = C >> 2</code>
<code>&=</code>	Bitwise AND assignment operator	<code>C &= 2</code> is same as <code>C = C & 2</code>
<code>^=</code>	bitwise exclusive OR and assignment operator	<code>C ^= 2</code> is same as <code>C = C ^ 2</code>
<code> =</code>	bitwise inclusive OR and assignment operator	<code>C = 2</code> is same as <code>C = C 2</code>

Misc Operators

- **Conditional Operator (? :):**
- Conditional operator is also known as the ternary operator. This operator consists of three operands and is used to evaluate boolean expressions. The goal of the operator is to decide which value should be assigned to the variable. The operator is written as :
- variable x = (expression) ? value if true : value if false
- Following is the example:

```
public class Test
{
    public static void main(String args[])
    {
        int a , b; a = 10; b = (a == 1) ? 20: 30;
        System.out.println( "Value of b is : " + b );
        b = (a == 10) ? 20: 30;
        System.out.println( "Value of b is : " + b );
    }
}
```

This would produce following result:

- **Value of b is : 30**
- **Value of b is : 20**

instanceOf Operator:

- This operator is used only for object reference variables. The operator checks whether the object is of a particular type(class type or interface type).
- instanceof operator is written as:
- (Object reference variable) instanceof (class/interface type)
- If the object referred by the variable on the left side of the operator passes the IS-A check for the class/interface type on the right side then the result will be true.
- Following is the example:
- String name = "James";
- boolean result = name instanceof String;
 // This will return true since name is type of String

instanceOf operator

- This operator will still return true if the object being compared is the assignment compatible with the type on the right.
- Following is one more example:

```
class Vehicle {}  
public class Car extends Vehicle  
{  
    public static void main(String args[])  
    {  
        Vehicle a = new Car();  
        boolean result = a instanceof Car;  
        System.out.println( result);  
    }  
}
```

This would produce following result:
`true`

Precedence of Java Operators:

Category	Operator	Associativity
Postfix	() [] . (dot operator)	Left to right
Unary	++ - - ! ~	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	>> >>> <<	Left to right
Relational	> >= < <=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR		Left to right
Logical AND	&&	Left to right
Logical OR		Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= =	Right to left
Comma	,	Left to right

Java Decision Making

- There are two types of decision making statements in Java.

They are:

1. if statements
2. switch statements

if Statement:

Syntax:

```
if(Boolean_expression)
```

```
{
```

```
//Statements will execute if the Boolean expression is true
```

```
}
```

If the boolean expression evaluates to true then the block of code inside the if statement will be executed. If not the first set of code after the end of the if statement(after the closing curly brace) will be executed.

if Statement:

```
public class Test
```

```
{
```

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

```
{
```

```
        int x = 10;
```

```
        if( x < 20 )
```

```
{
```

```
            System.out.print("This is if statement");
```

```
}
```

```
}
```

```
}
```

Output:-
This is if statement

if..else Statement:

- An if statement can be followed by an optional *else* statement, which executes when the Boolean expression is false.

Syntax:

```
if(Boolean_expression)
{
    //Executes when the Boolean expression is true
}
else
{
    //Executes when the Boolean expression is false
}
```

- If the boolean expression evaluates to true then the block of code inside the if statement will be executed. If not the first set of code after the end of the if statement(after the closing curly brace) will be executed.

If...else Statement:

```
public class Test
{
    public static void main(String args[])
    {
        int x = 30;
        if( x < 20 )
        {
            System.out.print("This is if statement");
        }
        else
        {
            System.out.print("This is else statement");
        }
    }
}
```

Output:-
This is else statement

if...else if...else Statement:

- An if statement can be followed by an optional *else if...else* statement, which is very useful to test various conditions using single if...else if statement.
- **Syntax:**

```
if(Boolean_expression 1)
{
    //Executes when the Boolean expression 1 is true
}
else if(Boolean_expression 2)
{
    //Executes when the Boolean expression 2 is true
}
else if(Boolean_expression 3)
{
    //Executes when the Boolean expression 3 is true
}
else
{
    //Executes when the none of the above condition is true.
}
```

if...else if...else Statement

```
public class Test
{
    public static void main(String args[])
    {
        int x = 30;
        if( x == 10 )
        {
            System.out.print("Value of X is 10");
        }
        else if( x == 20 )
        {
            System.out.print("Value of X is 20");
        }
        else if( x == 30 )
        {
            System.out.print("Value of X is 30");
        }
        else
        {
            System.out.print("This is else statement");
        }
    }
}
```

Output:-
Value of X is 30

Nested if...else Statement

- It is always legal to nest if-else statements, which means you can use one if or else if statement inside another if or else if statement.
- **Syntax:**

```
if(Boolean_expression 1)
{
    //Executes when the Boolean expression 1 is true
    if(Boolean_expression 2)
    {
        //Executes when the Boolean expression 2 is true
    }
}
```

You can nest *else if...else* in the similar way as we have nested *if* statement.

Nested if...else Statement

```
public class Test
{
    public static void main(String args[])
    {
        int x = 30;
        int y = 10;
        if( x == 30 )
        {
            if( y == 10 )
            {
                System.out.print("X = 30 and Y = 10");
            }
        }
    }
}
```

Output:-

X = 30 and Y = 10

Switch case

- A *switch* statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.

The following rules apply to a switch statement:

- The variable used in a switch statement can only be a byte, short, int, or char.
- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- The value for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.

Switch case

- When the variable being switched on is equal to a case, the statements following that case will execute until a *break* statement is reached.
- When a *break* statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
- Not every case needs to contain a break. If no break appears, the flow of control will *fall through* to subsequent cases until a break is reached.
- A *switch* statement can have an optional default case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

Syntax of switch case

```
switch(expression)
{
    case value :
        //Statements break;//optional
    case value :
        //Statements break; //optional
        //You can have any number of cases.
    default : //Optional //Statements
}
```

Example of switch case

```
public static void main(String args[])
{
    char grade = args[0].charAt(0);
switch(grade)
{
    case 'A' : System.out.println("Excellent!");
break;
    case 'B' :
    case 'C' : System.out.println("Well done");
        break;
    case 'D' : System.out.println("You passed");
    case 'F' : System.out.println("Better try again")
        break;
    default : System.out.println("Invalid grade");
}
System.out.println("Your grade is " + grade);
}
```

```
$ java Test a
Invalid grade
Your grade is a a
```

```
$ java Test A
Excellent!
Your grade is a A
```

```
$ java Test C
Well done
Your grade is a C
```

Loops In Java

- Java has very flexible three looping mechanisms.
- You can use one of the following three loops:
 - while Loop
 - do...while Loop
 - for Loop

while loop

- The syntax of a while loop is:

```
while(Boolean_expression)
{
    //Statements
}
```

- When executing, if the *boolean_expression* result is true then the actions inside the loop will be executed. This will continue as long as the expression result is true.
- Here key point of the *while* loop is that the loop might not ever run. When the expression is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

while loop

```
public class Test
{
    public static void main(String args[])
    {
        int x= 10;
        while( x < 20 )
        {
            System.out.print("value of x : " + x );
            x++;
        }
    }
}
```

value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19

do....while loop

- The syntax of a do....while loop is:

```
do  
{  
    //Statements  
}while(Boolean_expression);
```

- Notice that the Boolean expression appears at the end of the loop, so the statements in the loop execute once before the Boolean is tested.
- If the Boolean expression is true, the flow of control jumps back up to do, and the statements in the loop execute again. This process repeats until the Boolean expression is false.

do.....while loop

```
public class Test
{
    public static void main(String args[])
    {
        int x= 10;
        do
        {
            System.out.print("value of x : " + x );
            System.out.print("\n");
        }while( x < 20 );
    }
}
```

value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19

for loop

- A for loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.
- The syntax of a for loop is:

```
for(initialization;  
Boolean_expression;update)  
{  
    //Statements  
}
```

for loop

- Here is the flow of control in a for loop:
- The initialization step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.
- Next, the Boolean expression is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement past the for loop.
- After the body of the for loop executes, the flow of control jumps back up to the update statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the Boolean expression.
- The Boolean expression is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then update step, then Boolean expression). After the Boolean expression is false, the for loop terminates.

for loop

```
public class Test
```

```
{
```

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

```
{
```

```
        for(int x = 10; x < 20; x = x+1)
```

```
{
```

```
            System.out.print("value of x : "+x );
```

```
}
```

```
}
```

```
}
```

```
value of x : 10  
value of x : 11  
value of x : 12  
value of x : 13  
value of x : 14  
value of x : 15  
value of x : 16  
value of x : 17  
value of x : 18  
value of x : 19
```

for each loop

- As of java 5 the enhanced for loop was introduced. This is mainly used for Arrays.

- **Syntax:**

```
for(declaration : expression)
{
    //Statements
}
```

- **Declaration** . The newly declared block variable, which is of a type compatible with the elements of the array you are accessing. The variable will be available within the for block and its value would be the same as the current array element.
- **Expression** . This evaluate to the array you need to loop through. The expression can be an array variable or method call that returns an array.

Foreach loop

```
public class Test
{
    public static void main(String args[])
    {
        int [] numbers = {10, 20, 30, 40, 50};
        for(int x : numbers )
        {
            System.out.print( x );
        System.out.print(",");
        }
        System.out.print("\n");
        String [] names = {"James", "Larry", "Tom", "Lacy"};
        for( String name : names )
        {
            System.out.print( name );
            System.out.print(",");
        }
    }
}
```

Output:
10,20,30,40,50,
James,Larry,Tom,Lacy,

break

- The *break* keyword is used to stop the entire loop. The break keyword must be used inside any loop or a switch statement.
- The break keyword will stop the execution of the innermost loop and start executing the next line of code after the block.

Syntax:

```
break;
```

break

```
public class Test
{
    public static void main(String args[])
    {
        int [] numbers = {10, 20, 30, 40, 50};
        for(int x : numbers )
        {
            if( x == 30 )
            {
                break;
            }
            System.out.print( x );
            System.out.print("\n");
        }
    }
}
```

Output:
10
20

continue

- The *continue* keyword can be used in any of the loop control structures. It causes the loop to immediately jump to the next iteration of the loop.
- In a for loop, the continue keyword causes flow of control to immediately jump to the update statement.
- In a while loop or do/while loop, flow of control immediately jumps to the Boolean expression.
- **Syntax:**

continue;

continue

```
public class Test
{
    public static void main(String args[])
    {
        int [] numbers = {10, 20, 30, 40, 50};
        for(int x : numbers )
        {
            if( x == 30 )
            {
                continue;
            }
            System.out.print( x );
            System.out.print("\n");
        }
    }
}
```

Output:

10

20

40

50

String

String CONSTRUCTORS

String() Constructs a new empty String.

String(String) Constructs a new String that is a copy of the specified String.

String(char[]) Constructs a new String whose initial value is the specified array of characters.

String(char[], int, int) Constructs a new String whose initial value is the specified sub array of characters.

String(byte[], int, int) Constructs a new String whose initial value is the specified sub array of bytes.

String(byte[], int) Constructs a new String whose initial value is the specified array of bytes.

- **charAt(int)** Returns the character at the specified index.
- **compareTo(String)** Compares this String to another specified String.
- **concat(String)** Concatenates the specified string to the end of this String.
- **copyValueOf(char[], int, int)** Returns a String that is equivalent to the specified character array.
- **copyValueOf(char[])** Returns a String that is equivalent to the specified character array.
- **endsWith(String)** Determines whether the String ends with some suffix.
- **equals(Object)** Compares this String to the specified object.
- **equalsIgnoreCase(String)** Compares this String to another object.

- **getChars**(int, int, char[], int) Copies characters from this String into the specified character array.
- **indexOf**(String) Returns the index within this String of the first occurrence of the specified substring.
- **indexOf**(String, int) Returns the index within this String of the first occurrence of the specified substring.
- **lastIndexOf**(int, int) Returns the index within this String of the last occurrence of the specified character.
- **lastIndexOf**(String) Returns the index within this String of the last occurrence of the specified substring.
- **length()** Returns the length of the String.

- **replace**(char, char) Converts this String by replacing all occurrences of oldChar with newChar.
- **startsWith**(String) Determines whether this String starts with some prefix.
- **toCharArray**() Converts this String to a character array.
- **toLowerCase**() Converts all of the characters in this String to lower case.
- **toString**() Converts this String to a String.
- **toUpperCase**() Converts all of the characters in this String to upper case.
- **trim**() Trims leading and trailing whitespace from this String.
- **valueOf**(Object) Returns a String that represents the String value of the object.

Java String

Java String

In Java, string is basically an object that represents sequence of char values. An array of characters works same as Java string.

String

```
char[] ch = {'m','c','a','','d','d','u'};
```

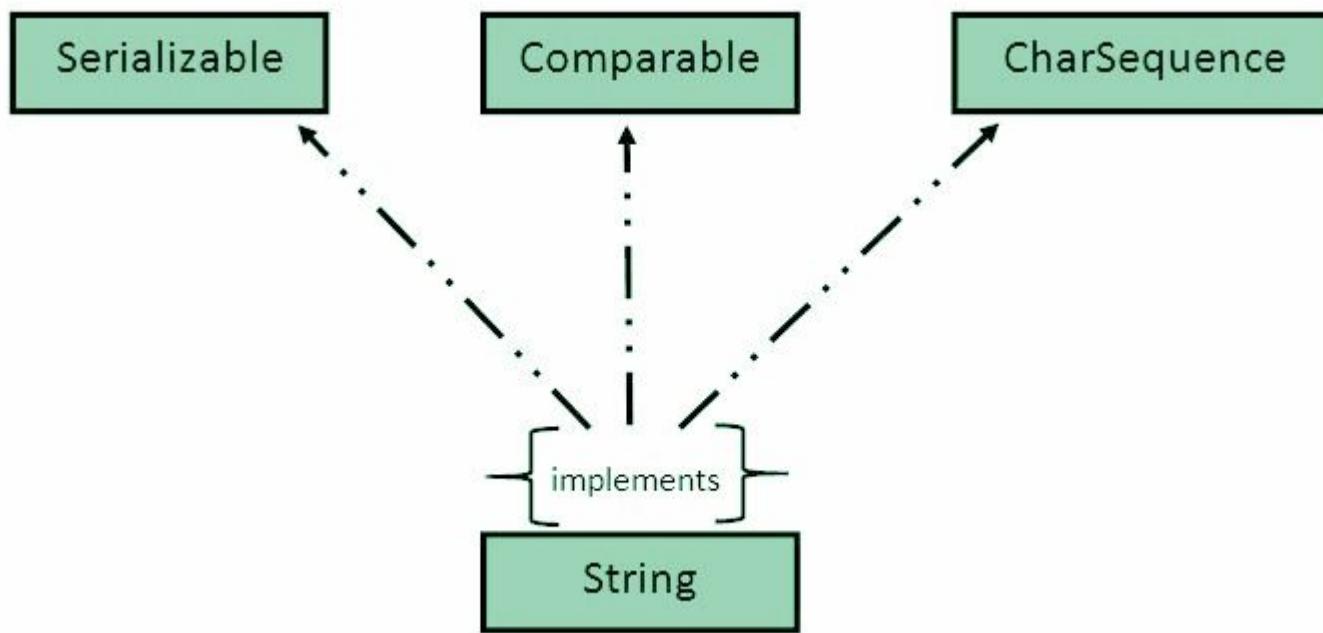
```
String s = new String(ch);
```

```
String s = "mca ddu";
```

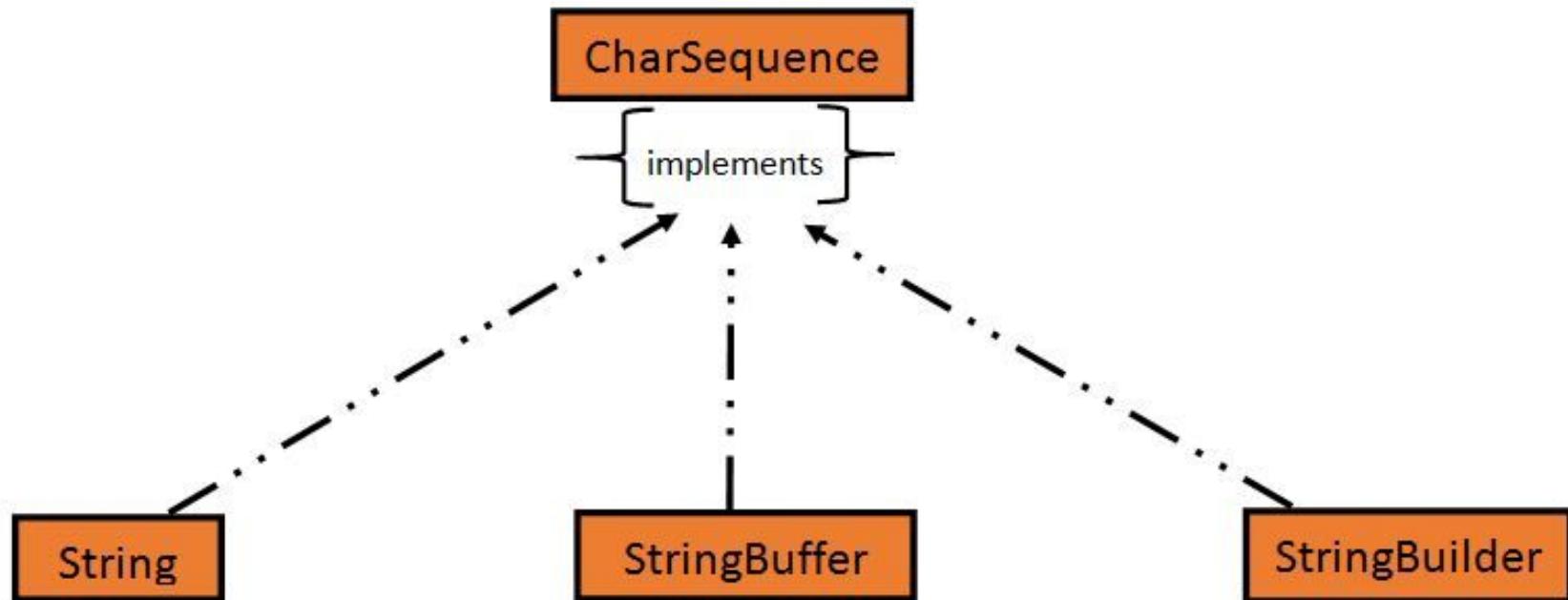
String is an object that represents sequence of characters. In Java, String is represented by String class which is located into **java.lang** package

One important thing to notice about string object is that string objects are **immutable** that means once a string object is created it cannot be changed.

String



CharSequence Interface is used for representing a sequence of characters.



Creation of String

There are two ways to create String object:

- By string literal

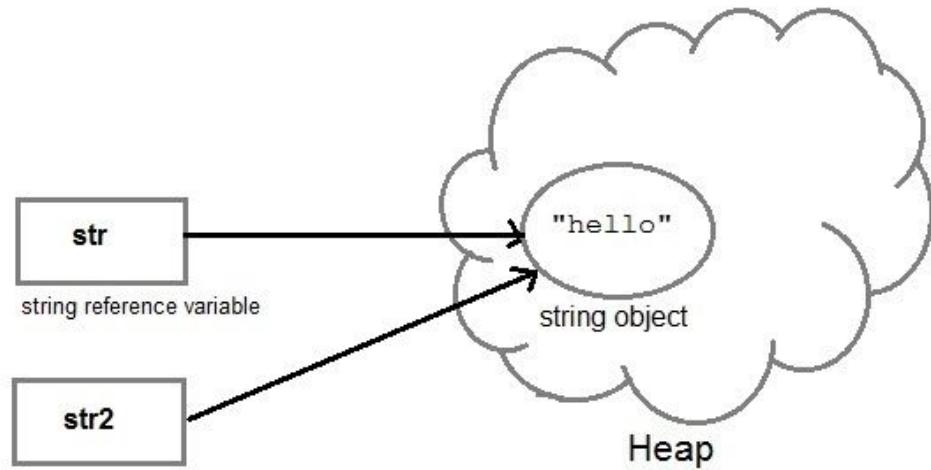
```
String s1 = "Hello DDU";
```

- By new keyword

```
String s1 = new String("Hello DDU");
```

String Literals

- String objects are stored in a special memory area known as string constant pool inside the heap memory.
- Each time you create a string literal, the JVM checks the "string constant pool" first.
- If the string already exists in the pool, a reference to the pooled instance is returned. If the string doesn't exist in the pool, a new string instance is created and placed in the pool.



```
String s1="hello";
String s2="hello";//It doesn't create a new instance
//String s2 = s1
```

String Object

```
String s=new String("hello");
```

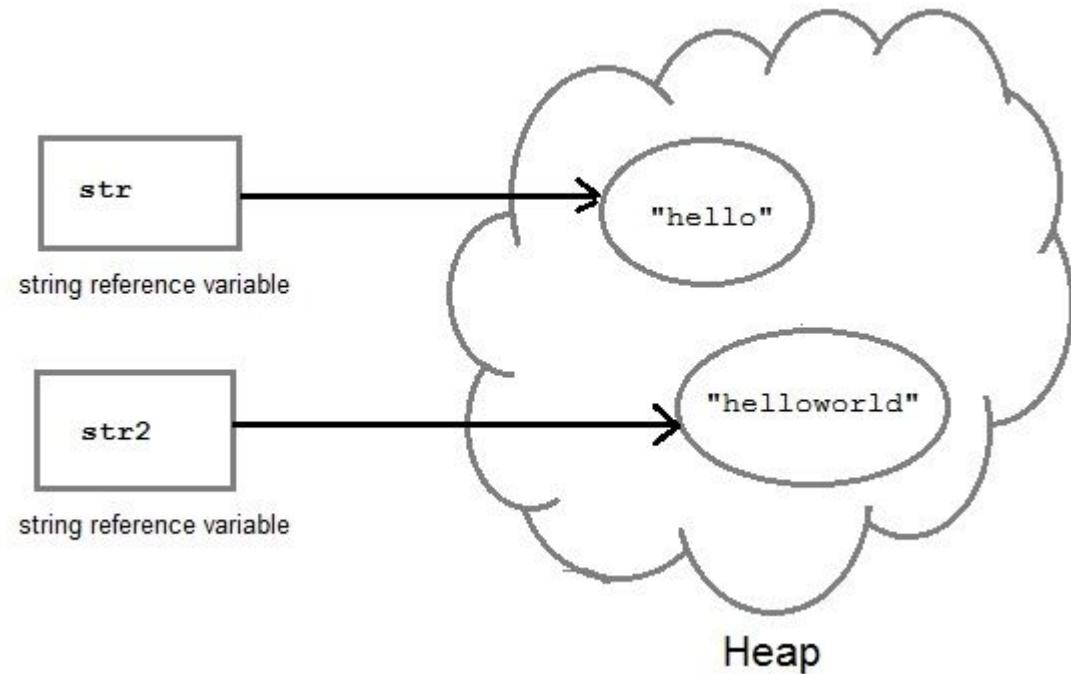
In such case, JVM will create a new string object in normal (non-pool) heap memory, and the literal "Welcome" will be placed in the string constant pool. The variable s will refer to the object in a heap (non-pool).

Change in String will Change the reference

```
String str= "hello";
```

```
String str2 = str;
```

```
str2=str2.concat("world");
```



Immutable object

- An object whose state cannot be changed after it is created is known as an Immutable object. String, Integer, Byte, Short, Float, Double and all other wrapper classes objects are immutable.
- In java, string objects are immutable. Immutable simply means unmodifiable or unchangeable.
- Once string object is created its data or state can't be changed but a new string object is created.

```
String s="DharmsinhDesaiUniversity";
```

```
s.concat("MCADepartment");//concat() method appends the string at the end
```

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

Why immutable?

- Java uses the concept of string literal.
- Suppose there are 5 reference variables, all refers to one object "DDU".
- If one reference variable changes the value of the object, it will be affected to all the reference variables.
- That is why string objects are immutable in java.

String compare

We can compare string in java on the basis of content and reference.

- authentication (by equals() method)
- sorting (by compareTo() method)
- reference matching (by == operator)

compare by equals()

The String equals() method compares the original content of the string. It compares values of string for equality. String class provides two methods:

```
public boolean equals(Object obj)
```

//compares this string to the specified object.

```
public boolean equalsIgnoreCase(String str)
```

//compares this String to another string, ignoring case.

compare by == operator

The == operator compares references not values.

```
String s1="DDU";
```

```
String s2="DDU";
```

```
String s3=new String("DDU");
```

```
System.out.println(s1==s2); //true (because both refer to same instance)
```

```
System.out.println(s1==s3); //false(because s3 refers to instance created in nonpool)
```

compare by compareTo()

The String compareTo() method compares values lexicographically (alphabetical order.) and returns an integer value that describes if first string is less than, equal to or greater than second string.

Suppose s1 and s2 are two string variables. If:

- $s1 == s2 : 0$
- $s1 > s2 : \text{positive value}$
- $s1 < s2 : \text{negative value}$

StringBuffer class



StringBuffer class

StringBuffer class is used to create a mutable string object. It means, it can be changed after it is created. It represents growable and writable character sequence.

StringBuffer()

//creates an empty string buffer with the initial capacity of 16.

StringBuffer(String str)

//creates a string buffer with the specified string.

StringBuffer(int capacity)

//creates an empty string buffer with the specified capacity as length.

StringBuffer(charSequence []ch):

//It creates a StringBuffer object from the charsequence array.

- `append(Object)` - Appends an object to the end of this buffer.
- `capacity()` - Returns the current capacity of the String buffer.
- `charAt(int)` - Returns the character at the specified index.
- `ensureCapacity(int)` - Ensures that the capacity of the buffer is at least equal to the specified minimum.

```
StringBuffer stringBuffer = new StringBuffer("Hello World");
```

```
System.out.println(stringBuffer.length()); //11
```

```
System.out.println(stringBuffer.capacity()); //27
```

- `getChars(int, int, char[], int)` - Copies the characters of the specified substring (determined by `srcBegin` and `srcEnd`) into the character array, starting at the array's `dstBegin` location.
- `insert(int Object)` - Inserts an object into the String buffer.
- `length()` - Returns the length (character count) of the buffer.
- `setCharAt(int, char)` - Changes the character at the specified index to be `ch`.
- `toString()` - Converts to a String representing the data in the buffer

StringBuilder class

- StringBuilder is identical to StringBuffer except for one important difference that it is not synchronized, which means it is not thread safe.
- StringBuilder also used for creating string object that is mutable and non synchronized. The StringBuilder class provides no guarantee of synchronization. StringBuffer and StringBuilder both are mutable but if synchronization is not required then it is recommended to use StringBuilder class.

StringBuffer class	StringBuilder class
StringBuffer is synchronized.	StringBuilder is not synchronized.
Because of synchronisation, StringBuffer operation is slower than StringBuilder.	StringBuilder operates faster.
StringBuffer is thread-safe	StringBuilder is not thread-safe
StringBuffer is less efficient as compare to StringBuilder	StringBuilder is more efficient as compared to StringBuffer.
Its storage area is in the heap	Its storage area is the stack
It is mutable	It is mutable
Methods are synchronized	Methods are not synchronized
It is alternative of string class	It is more flexible as compared to the string class

Introduced in Java 1.0

Its performance is moderate

It consumes more memory

Introduced in Java 1.5

Its performance is very high

It consumes less memory

Modifiers

JAVA

Access modifiers

Introduction

Do you define how people would access some of your properties? You would not want anyone using your properties. but, your close friends and relatives can have some access.. Similarly Java controls access, too.

As the name suggests access modifiers in Java helps to restrict the scope of a class, constructor, variable, method, or data member.

Or

In Java, access modifiers are used to set the accessibility (visibility) of classes, interfaces, variables, methods, constructors, data members, and the setter methods.

Introduction

Java provides 4 levels/ types of access modifiers.

- Private: The access level of a private modifier is **only within the class**. It cannot be accessed from outside the class.
- Default: The access level of a default modifier is **only within the package**. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
- Protected: The access level of a protected modifier is **within the package and outside the package through child class**. If you do not make the child class, it cannot be accessed from outside the package.
- Public: The access level of a public modifier is **everywhere**. It can be accessed from within the class, outside the class, within the package and outside the package.

Access Modifier	within class	within package by subclass	within package by non subclass	outside package by subclass only	outside package
Private	Y	N	N	N	N
Default	Y	Y	Y	N	N
Protected	Y	Y	Y	Y	N
Public	Y	Y	Y	Y	Y

Default access modifier

- When we do not mention any access modifier, it is called default access modifier.
- The scope of this modifier is limited to the package only. This means that if we have a class with the default access modifier in a package, only those classes that are in this package can access this class. No other class outside this package can access this class.
- Similarly, if we have a default method or data member in a class, it would not be visible in the class of another package.

```
package abcpackage;  
  
public class Addition {  
  
    int addTwoNumbers(int a, int b){  
  
        return a+b;  
  
    }  
  
}
```

Exception in thread "main" java.lang.Error:
Unresolved compilation problem:
The method addTwoNumbers(int, int) from the type
Addition is not visible
at xyzpackage.Test.main

```
package xyzpackage;  
  
import abcpackage.*;  
  
public class Test {  
  
    public static void main(String args[]){  
  
        Addition obj = new Addition();  
  
        /* It will throw error because we are trying to  
        access the default method in another package*/  
  
        obj.addTwoNumbers(10, 21); }  
  
}
```

Private access modifier

The scope of private modifier is limited to the class only.

- Private Data members and methods are only accessible within the class
- Class and Interface cannot be declared as private
- If a class has private constructor then you cannot create the object of that class from outside of the class.

```
class ABC{  
  
    private double num = 100;  
  
    private int square(int a){  
  
        return a*a;  
  
    }  
  
}
```

```
public class Example{  
  
    public static void main(String args[]){  
  
        ABC obj = new ABC();  
  
        System.out.println(obj.num);  
  
        System.out.println(obj.square(10));  
  
    }  
  
}
```

Protected Access Modifier

Protected data member and method are only accessible by the classes of the same package and the subclasses present in any package.

You can also say that the protected access modifier is similar to default access modifier with one exception that it has visibility in sub-classes.

Classes cannot be declared protected. This access modifier is generally used in a parent child relationship.

```
package abcpackage;

public class Addition {
    protected int addTwoNumbers(int a, int b){
        return a+b; }
}
```

- class Test which is present in another package is able to call the addTwoNumbers() method, which is declared protected.
- This is because the Test class extends class Addition and the protected modifier allows the access of protected members in subclasses

```
package xyzpackage;
import abcpackage.*;

public class Test extends Addition{
    public static void main(String args[]){
        Addition obj = new Addition();
        obj.addTwoNumbers(10, 21); }
}
```

Public access modifier

The members, methods and classes that are declared public can be accessed from anywhere. This modifier doesn't put any restriction on the access.

```
package abcpackage;  
  
public class Addition {  
  
    public int addTwoNumbers(int a, int b){  
  
        return a+b; }  
  
}
```

```
package xyzpackage;  
  
import abcpackage.*;  
  
public class Test{  
  
    public static void main(String args[]){  
  
        Addition obj = new Addition();  
  
        obj.addTwoNumbers(10, 21); }  
  
}
```

- Test is able to access this method without even extending the Addition class. This is because public modifier has visibility everywhere.

Non Access Modifiers

Introduction

Java provides a number of non-access modifiers to achieve many other functionality.

- The **static** modifier for creating class methods and variables
- The **final** modifier for finalizing the implementations of classes, methods, and variables.
- The **abstract** modifier for creating abstract classes and methods.
- The **synchronized** and **volatile** modifiers, which are used for threads.

static Modifier

Variables:

- The static keyword is used to create variables that will exist independently of any instances created for the class. Only one copy of the static variable exists regardless of the number of instances of the class.
- Static variables are also known as class variables. Local variables cannot be declared static.

static Modifier

Methods:

- The static keyword is used to create methods that will exist independently of any instances created for the class.
- Static methods do not use any instance variables of any object of the class they are defined in. Static methods take all the data from parameters and compute something from those parameters, with no reference to variables.
- Class variables and methods can be accessed using the class name followed by a dot and the name of the variable or method.

```
public class InstanceCounter { private static int  
    numInstances = 0; protected static int getCount()  
    { return numInstances; } private static void  
    addInstance() { numInstances++; }  
    InstanceCounter() {  
        InstanceCounter.addInstance(); } public static void  
    main(String[] arguments) {  
        System.out.println("Starting with " +  
            InstanceCounter.getCount() + " instances"); for  
            (int i = 0; i < 500; ++i){ new InstanceCounter(); }  
        System.out.println("Created " +  
            InstanceCounter.getCount() + " instances"); } }
```

- This would produce following result:
- Started with 0 instances Created 500 instances

Final Modifier:

Final is the modifier applicable for classes, methods and variables.

Methods:

- Whatever the methods parent has by default available to the child.
- If the child is not allowed to override any method, that method we have to declare with final in parent class. That is final methods cannot overridden.

Class:

- If a class declared as the final then we cannot creates the child class that is inheritance concept is not applicable for final classes.

Variables:

- Final variable are similar to symbolic constants. They cannot be declared in methods.

Final Modifier:

Variables:

- A final variable can be explicitly initialized only once. A reference variable declared final can never be reassigned to refer to an different object.
- However the data within the object can be changed. So the state of the object can be changed but not the reference.
- With variables, the final modifier often is used with static to make the constant a class variable.

```
class Test{  
    final int value=10; // The following are examples of declaring constants:  
    public static final int BOXWIDTH = 6;  
    static final String TITLE = "Manager";  
    public void changeValue(){  
        value = 12; //will give an error } }
```

Final Modifier:

Methods:

- A final method cannot be overridden by any subclasses. As mentioned previously the final modifier prevents a method from being modified in a subclass.
- The main intention of making a method final would be that the content of the method should not be changed by any outsider.

```
class Test{  
    public final void changeName()  
    { // body of method }  
}
```

Final Modifier:

Class:

- The main purpose of using a class being declared as final is to prevent the class from being subclassed. If a class is marked as final then no class can inherit any feature from the final class.

```
public final class Test
```

```
{
```

```
    // body of class
```

```
}
```

abstract Modifier:

Abstract is the modifier applicable only for methods and classes but not for variables.

Methods:

- Even though we don't have implementation still we can declare a method with abstract modifier.
- That is abstract methods have only declaration but not implementation.
- Hence abstract method declaration should compulsory ends with semicolon.

Class:

- For any java class if we are not allow to create an object such type of class we have to declare with abstract modifier that is for abstract class instantiation is not possible.

abstract Modifier:

Class:

- An abstract class can never be instantiated. If a class is declared as abstract then the sole purpose is for the class to be extended.
- A class cannot be both abstract and final. (since a final class cannot be extended). If a class contains abstract methods then the class should be declared abstract. Otherwise a compile error will be thrown.
- An abstract class may contain both abstract methods as well normal methods.

```
abstract class Caravan{  
    private double price;  
    private String model;  
    private String year;  
    public abstract void goFast(); //an abstract method  
    public abstract void changeColor();  
}
```

abstract Modifier:

Methods:

- An abstract method is a method declared without any implementation. The method's body (implementation) is provided by the subclass. Abstract methods can never be final or strict.
- Any class that extends an abstract class must implement all the abstract methods of the super class unless the subclass is also an abstract class.
- If a class contains one or more abstract methods then the class must be declared abstract. An abstract class does not need to contain abstract methods.
- The abstract method ends with a semicolon. Example: public abstract sample();

```
public abstract class SuperClass{
```

```
    abstract void m(); //abstract method }
```

```
class SubClass extends SuperClass{ // implements the abstract method
```

```
void m(){ ..... }}
```

synchronized Modifier:

The synchronized keyword used to indicate that a method can be accessed by only one thread at a time. The synchronized modifier can be applied with any of the four access level modifiers.

```
public synchronized void showDetails(){ ..... }
```



Java File Structure

Introduction

1. A java Program can contain any no. Of classes but at most one class can be declared as public.
2. "If there is a public class the name of the Program and name of the public class must be matched otherwise we will get compile time error".
3. If there is no public class then any name we can give for java source file.

class A{}

class B{}

class C{}

Case 1:

- If there is no public class then we can use any name for java source file there are no restrictions.

Case 2:

- If class B declared as public then the name of the Program should be B.java otherwise we will get compile time error saying "class B is public, should be declared in a file named B.java".

Case 3:

- If both B and C classes are declared as public and name of the file is B.java then we will get compile time error saying "class C is public, should be declared in a file named C.java".
- It is highly recommended to take only one class for source file and name of the Program (file) must be same as class name. This approach improves readability and understandability of the code.

- We can compile a java Program but not java class. In that Program for every class one dot class file will be created.
- We can run a java class but not java source file. Whenever we are trying to run a class the corresponding class main method will be executed.
- If the class won't contain main method then we will get runtime exception saying "NoSuchMethodError: main".
- If we are trying to execute a java class and if the corresponding .class file is not available then we will get runtime execution saying "NoClassDefFoundError: Sample".

Import Statements

Types of Import Statements:

There are 2 types of import statements.

1) Explicit class import

Import java.util.Scanner

- This type of import is highly recommended to use because it improves readability of the code.

2) Implicit class import.

import java.util.;*

- It is not recommended to use because it reduces readability of the code.

Whenever we are using fully qualified name it is not required to use import statement. Similarly whenever we are using import statements it is not required to use fully qualified name.

- We may get the ambiguity problem because it may be available in multiple packages.
- While resolving class names compiler will always gives the importance in the following order.
 1. Explicit class import
 2. Classes present in current working directory.
 3. Implicit class import.
- Whenever we are importing a package all classes and interfaces present in that package are by default available but not sub package classes. Explicit import is required.
- In any java Program the following 2 packages are not require to import.
 1. `java.lang` package
 2. default package(current working directory)

"Import statement is totally compile time concept" if more no of imports are there then more will be the compile time but there is "no change in execution time".

- In the case of C language #include all the header files will be loaded at the time of include statement hence it follows static loading.
- But in java import statement no ".class" will be loaded at the time of import statements in the next lines of the code whenever we are using a particular class then only corresponding ".class" file will be loaded. Hence it follows "dynamic loading" or "load-on -demand" or "load-on-fly".

Usually we can access static members by using class name but whenever we are using static import it is not require to use class name we can access directly.

```
System.out.println(Math.sqrt(4));  
System.out.println(Math.max(10,20));  
System.out.println(Math.random());
```

With static import:

```
import static java.lang.Math.sqrt;  
import static java.lang.Math.*;  
System.out.println(sqrt(4));  
System.out.println(max(10,20));  
System.out.println(random());
```

Packages In Java

Packages

- A **Package** can be defined as a grouping of related types (classes, interfaces, enumerations, annotations and sub-packages) providing access protection and name space management.
- Packages are use to control access of classes, interface, enumeration etc. and avoid namespace collision.
- There can not be two classes with same name in a same Package
- But two packages can have a class with same name.
- There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

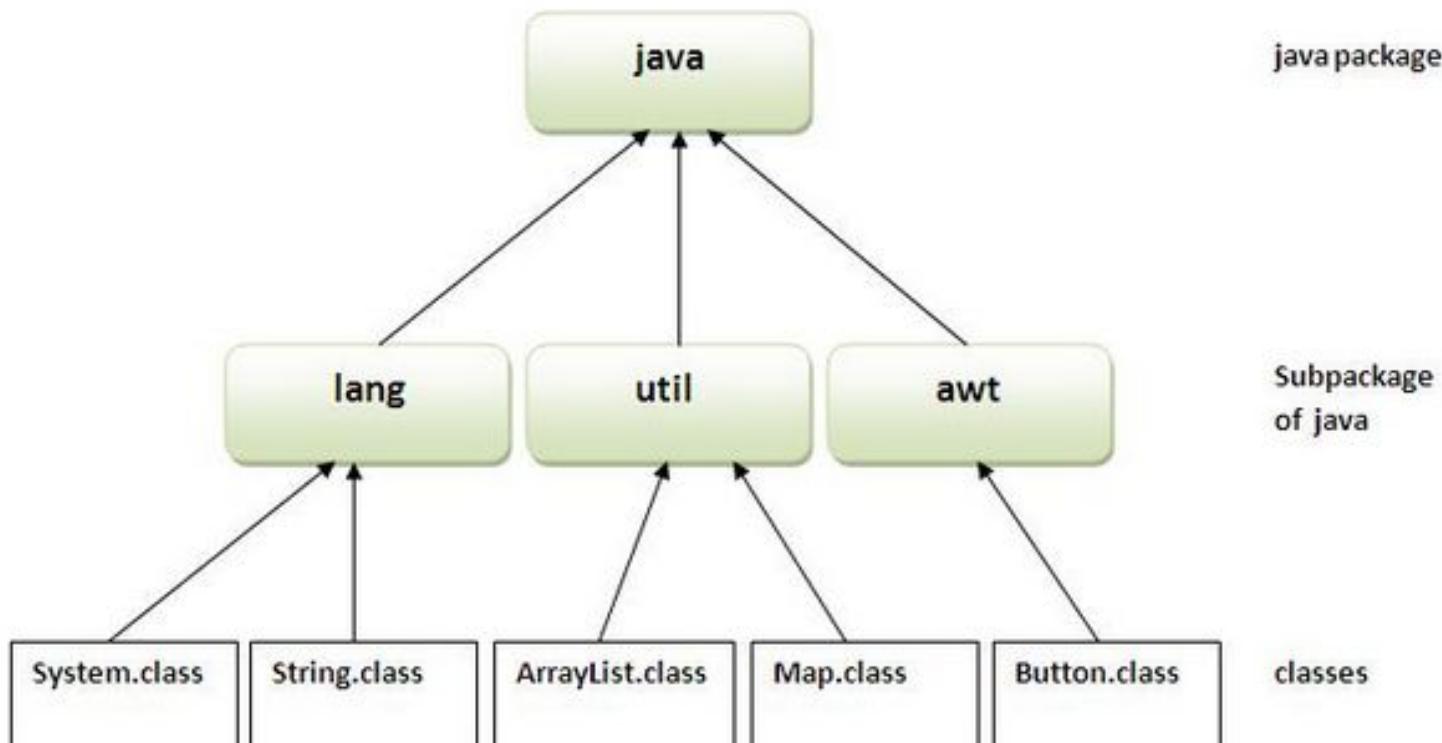
Packages

- Package names are dot separated, e.g., java.lang.
- Package names have a correspondence with the directory structure.
- Exact Name of the class is identified by its package structure.
- <> Fully Qualified Name>>
- java.lang.String ;
- java.util.Arrays;
- java.io.BufferedReader ;
- java.util.Date

Advantage of Java Package

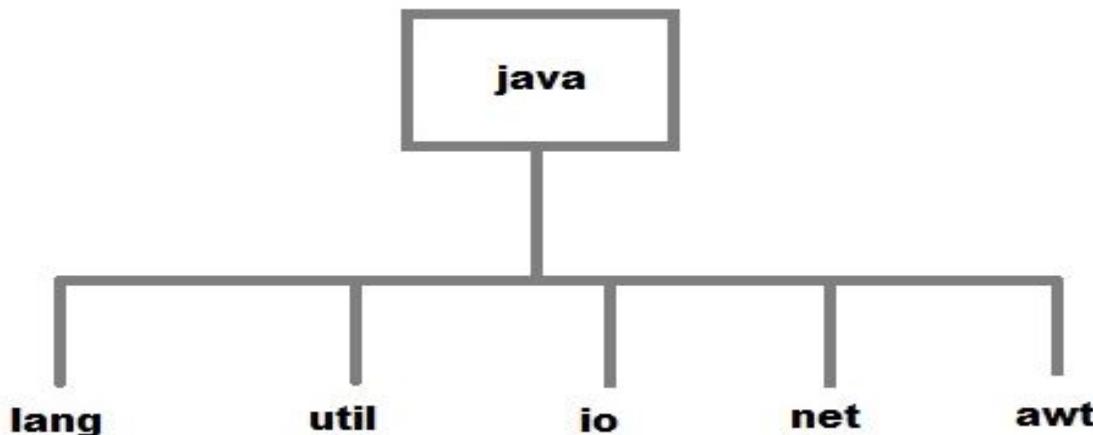
- Java package is used to categorize the classes and interfaces so that they can be easily maintained.
- Java package provides access protection.
- Java package removes naming collision.

Package Hierarchy



Con't

- Packages are categorized into two forms:
 - **Built-in java package** : java.lang, java.util etc.

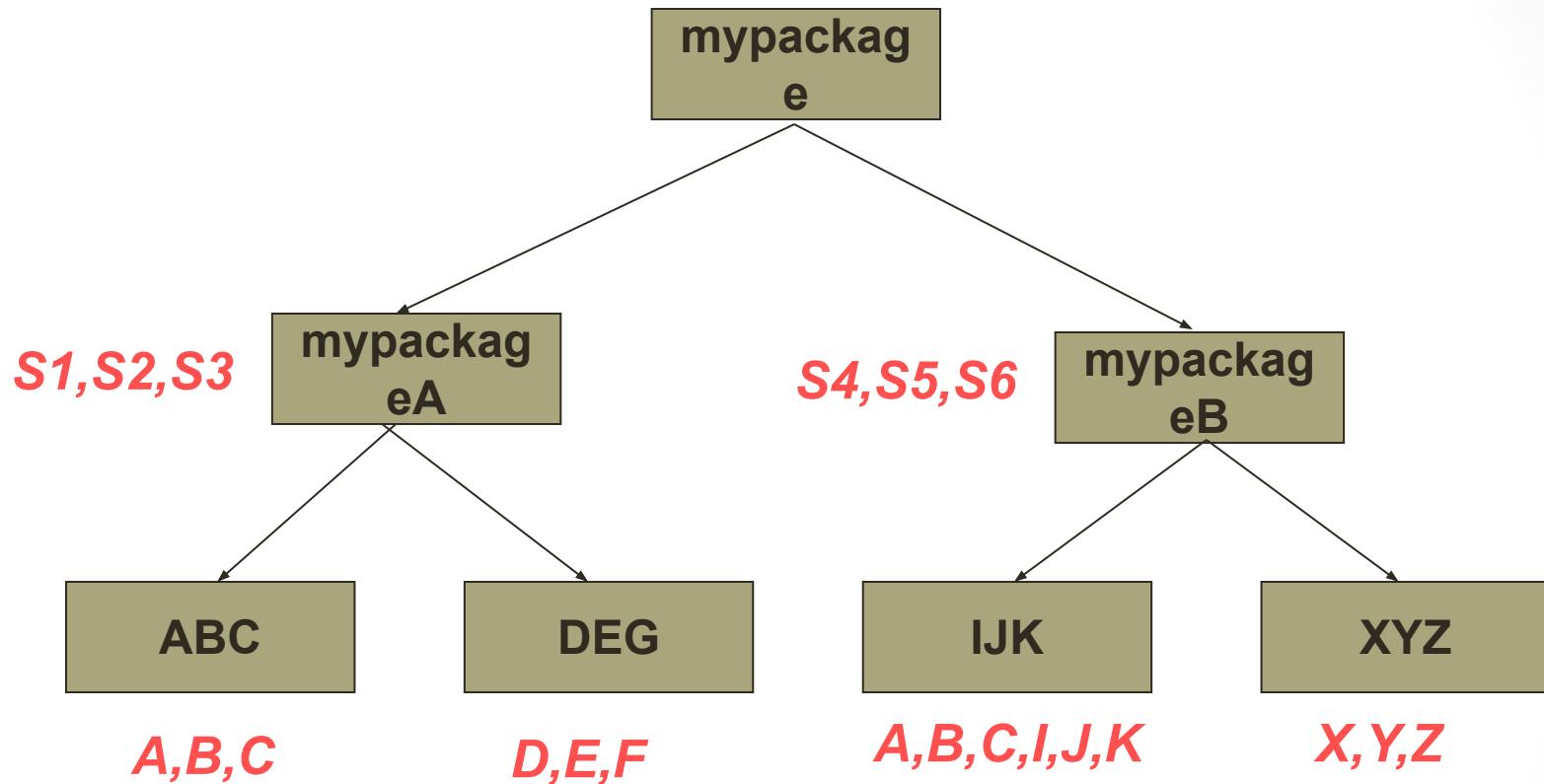


- **User-defined packages** : Java Package created by user to categorize classes and interface.

How To Create a Package

- To create a package, First we have to create a directory /directory structure that matches the package hierarchy.
 - To make a class belongs to a particular package include the package statement as the first statement of source file.
 - There can be only one package statement in each source file, and it applies to all types in the file.
-
- I:\5\mypack\Student.java
 - In mypack folder, file Student.java (we need to be in - I:\5)
 - Compile: javac mypack/Student.java (compiler operates on files)
 - Run : java mypack.Student (interpreter loads a class)

Creating Packages



- Package ABC and IJK have classes with same name.
- A class in ABC has name `mypackage.mypackageA.ABC.A`
- A class in IJK has name `mypackage.mypackageB.IJK.A`

How to make a class Belong to a Package

- Include a proper package statement as first line in source file

Make class S1 belongs to mypackageA

```
package mypackage.mypackageA;  
public class S1 {  
    public S1( ) {  
        System.out.println("This is Class S1");  
    }  
}
```

Name the source file as S1.java and compile it and store the S1.class file in mypackageA directory

Make class S2 belongs to mypackageA

```
package mypackage.mypackageA;
public class S2
{
    public S2( )
    {
        System.out.println("This is Class S2");
    }
}
```

Name the source file as `S2.java` and compile it and store the `S2.class` file in `mypackageA` directory

Make class A belongs to IJK

```
package mypackage.mypackageB.IJK;
public class A
{
    public A( )
    {
        System.out.println("This is Class A in IJK");
    }
}
```

Name the source file as A.java and compile it and store the A.class file in IJK directory

<< Same Procedure For all classes>>

Import keyword

- A class can use all classes from its own package and all public classes from other packages.

3 ways to refer to a class present in different package:

- Using fully qualified name
 - Class MyDate extends java.util.Date { ... }
- Import the only class you want to use
 - import java.util.date;
 - class MyDate extends Date{ ... }
- Import all the classes from the particular package
 - import java.util.*;
 - class MyDate extends Date{ ... }
- Import statement is kept after package statement
 - E.g. package mypack;
import java.util.*;

Importing the Package

- import statement allows the importing of package
- Library packages are automatically imported irrespective of the location of compiling and executing program
- JRE looks at two places for user created packages
 - (i) Under the current working directory
 - (ii) At the location specified by CLASSPATH environment variable
- Most ideal location for compiling/executing a program is immediately above the package structure.

Employee.java

Boss.java

Example importing

```
import mypackage.mypackageA.ABC;  
import mypackage.mypackageA.ABC.*;  
class packagetest  
{  
    public static void main(String args[])  
    {  
        B b1 = new B();                                << packagetest.java>>  
        C c1 = new C();                                This is Class B  
    }                                                 This is Class C  
}
```

<< Store it in location above the package structure. Compile and Execute it from there>>

```
import mypackage.mypackageA.ABC.*;
import mypackage.mypackageB.IJK.*;
class packagetest
{
    public static void main(String args[])
    {
        A a1 = new A();
    }
}
```

<< What's Wrong Here>>

```
mypackage.mypackageA.ABC.A a1 = new
mypackage.mypackageA.ABC.A();
```

OR

```
mypackage.mypackageB.IJK.A a1 = new mypackage.mypackageB.IJK.A();
```

**<< class A is present in both the imported packages ABC and IJK.
So A has to be fully qualified in this case>>**

CLASSPATH Environmental Variables

- CLASSPATH Environmental Variable lets you define path for the location of the root of the package hierarchy.
- Consider the following statement :

```
package mypack;
```

What should be true in order for the program to find mypack.

- (i) Program should be executed from the location immediately above mypack.

- I:\5> javac mypackage/mypackageA/ABC/A.java
- I:\5> javac mypackage.mypackageA.ABC.A

OR

CLASSPATH Environmental Variables

(ii) mypack should be listed in the set of directories for CLASSPATH.

- I:\5\mypack>javac Student.java
 - I:\5\mypack>set classpath = .;I:\5\;
 - I:\5\mypack>java mypack.Student
-
- I:\5\mypackage\mypackageA\ABC>javac A.java
 - I:\5\mypack>set classpath = .;I:\5\;
 - I:\5\mypackage\mypackageA\ABC>java mypackage.mypackageA.ABC.A

The Directory Structure of Packages:

- In general, a company uses its reversed Internet domain name for its package names.
 - Eg: A company's Internet domain name is apple.com, then all its package names would start with com.apple.
- Example: The company had a com.apple.computers package that contained a Dell.java source file.
 - // File Name: Dell.java
 - package com.apple.computers;
 - public class Dell{ }
 - class Ups{ }

What is jar files? How to create it?

- JAR : It is a java archive file used to package classes, files etc. as single file.
 - This is similar to zip file in windows.
- To create a jar file with all class files under the current directory.
 - `jar -cvf jarfilename.jar *.class`
 - c - to *create* a JAR file.
 - f - the output goes to a *file* rather than to stdout.
 - v -Produces *verbose* output on stdout while the JAR file is being built.
 - The verbose output tells you the name of each file as it's added to the JAR file.

What is jar files? How to create it?

- Can jar code be retrieved from byte code?
 - Yes, class files can be decompiled using utilities like JAD(JAvA Decompiler). This takes the class files as an input and generates a java file.

How to run?

- You need to specify a Main-Class in the jar file manifest.
- You need two files: java/class files to be included in jar file and manifest.mf file to indicate the main class.
- Note that the text file must end with a new line or carriage return.
- The last line will not be parsed properly if it does not end with a new line or carriage return.
 - javac Test.java
 - jar cfm test.jar manifest.mf Test.class
 - java -jar test.jar

static import

- To import static member of class.
- Using static import it is possible to refer static member without even using class name.
- Import static package.class-name.static-member-name;
 - E.g. import static java.lang.Math.sqrt;
- Import static package.class-type-name.*;
 - E.g. import static java.lang.Math.*;
- Test.java

What's the use of static imports?

- Static imports are used to save your time and typing.
- If you hate to type same thing again and again then you may find such imports interesting.
- Lets understand this with the help of below examples:
- **Example 1: Without Static Imports**
- class Demo1{

```
    public static void main(String args[]) {  
        double var1= Math.sqrt(5.0);  
        double var2= Math.tan(30);  
        System.out.println("Square of 5 is:"+ var1);  
        System.out.println("Tan of 30 is:"+ var2);  
    }  
}
```

- **Output:**
 - Square of 5 is:2.23606797749979 Tan of 30 is:-6.405331196646276

What's the use of static imports?

- **Example 2: Using Static Imports**

- import static java.lang.System.out;
- import static java.lang.Math.*;

```
class Demo2{
    public static void main(String args[]) {
        //instead of Math.sqrt need to use only sqrt
        double var1= sqrt(5.0); //instead of Math.tan need to use only tan
        double var2= tan(30); //need not to use System in both the below statements
        out.println("Square of 5 is:"+var1);
        out.println("Tan of 30 is:"+var2);
    }
}
```

Output:

Square of 5 is:2.23606797749979 Tan of 30 is:-6.405331196646276

When to use static imports?

- If you are going to use static variables/methods a lot then it's fine to use static imports.
- For example if you want to write a code with lot of mathematical calculations then you may want to use static import.
- **Drawbacks**
 - It makes the code confusing and less readable so if you are going to use static members very few times in your code then probably you should avoid using it.
 - You can also use wildcard(*) imports.

Java - Inheritance

...

Inheritance

- Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object.
- The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.
- Inheritance gives you the opportunity to add some functionality that does not exist in the original class.
- The subclass and the superclass has an "is-a" relationship / parent-child relationship.
 - Vehicle is super class of Car.
 - Car is sub class of Vehicle.
 - Car IS-A Vehicle.

Inheritance

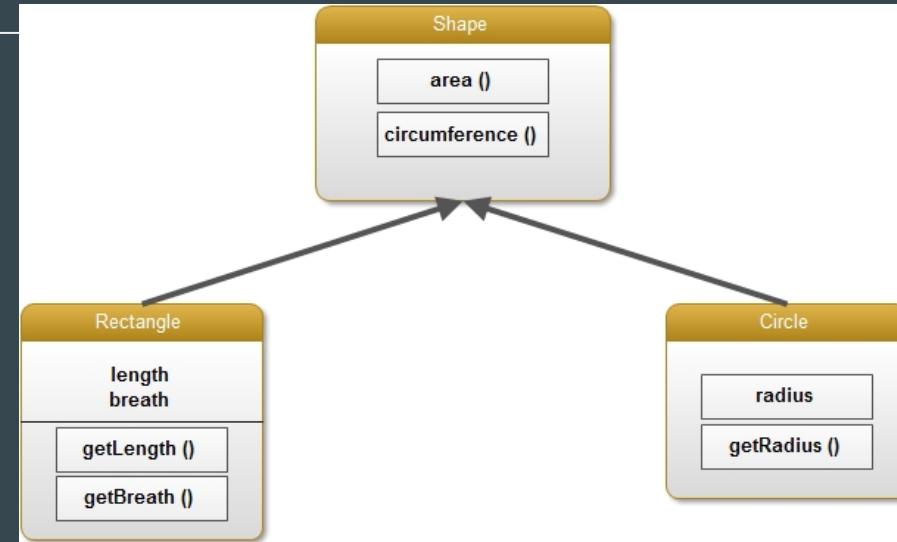
- Within a subclass you can access its superclass's public and protected methods and fields, but not the superclass's private methods.
- If the subclass and the superclass are in the same package, you can also access the superclass's default methods and fields.
- Using inheritance the two classes (parent and child class) gets tightly coupled.
- This means that if we change code of parent class, it will affect to all the child classes which is inheriting/deriving the parent class, and hence, it cannot be independent of each other

Inheritance - Why?

1. For Method Overriding (so runtime polymorphism can be achieved).
 2. It promotes the code reusability i.e the same methods and variables which are defined in a parent/super/base class can be used in the child/sub/derived class.
-
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
 - **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class

Inheritance - extends keyword

```
class Subclass-name extends Superclass-name  
{  
    //methods and fields  
}
```



- “extends” indicates that you are making a new class that derives from an existing class. [“extends” means to increase the functionality].
- A class which is inherited is called a parent or superclass, and the new class is called child or subclass.

Inheritance - extend keyword

```
class Animal{  
  
    private String type;  
    public Animal(String aType){  
        type = aType;  
    }  
    public String toString(){  
        return "This is a " + type;  
    }  
}
```

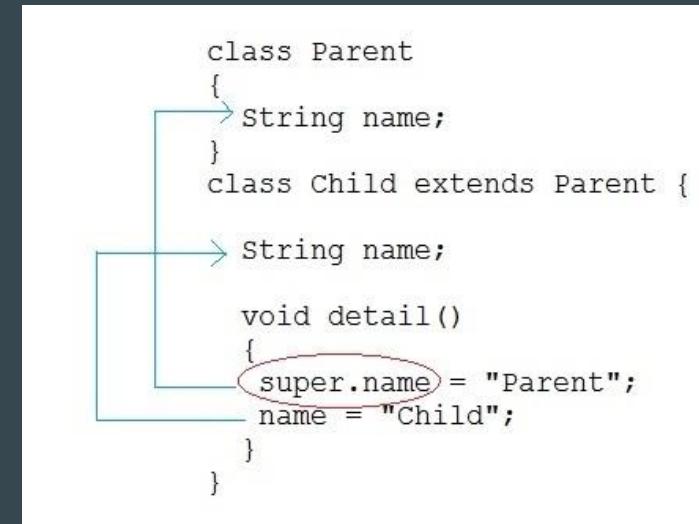
```
class Dog extends Animal{  
  
    private String breed;  
  
    public Dog(String name){  
        super(name);  
    }  
}
```

Inheritance - super Keyword

- The keyword super represents an instance of the direct superclass of the current object.
- You can explicitly call the parent's constructor from a subclass's constructor by using the super keyword.
- 'super' must be the first statement in the constructor.

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

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



Inheritance - Example

```
class Animal {  
  
    private String type;  
    public Animal(String aType) {  
        type = new String(aType);  
    }  
    public String toString() {  
        return "This is a " + type;  
    }  
}
```

```
class Dog extends Animal {  
  
    private String name;  
    private String breed;  
    public Dog(String aName) {  
        super("Dog");  
        name = aName;  
        breed = "Unknown";  
    }  
    public Dog(String aName, String aBreed) {  
        super("Dog");  
        name = aName;  
        breed = aBreed;  
    }  
}
```

Inheritance - Example

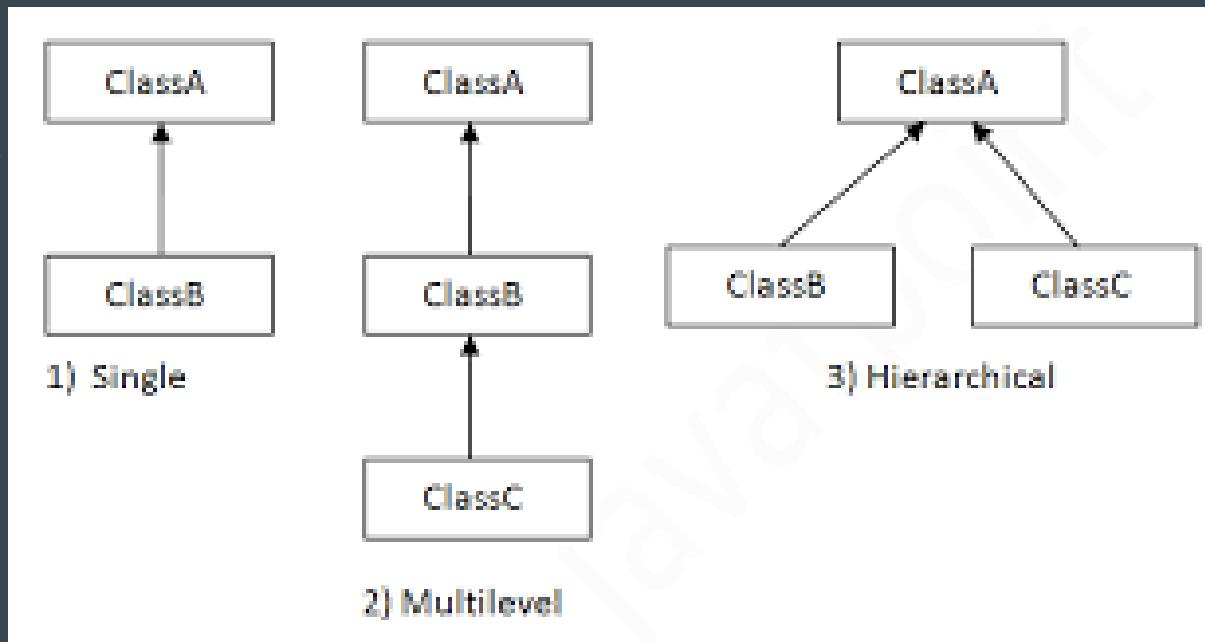
```
class Parent{  
    String name;  
}  
  
public class Child extends Parent {  
    String name;  
    public void details(){  
        super.name = "Parent"; //refers  
        to parent class member  
        name = "Child";  
  
        System.out.println(super.name+ " and  
        "+name);  
    }  
    public static void main(String[] args){  
        Child cobj = new Child();  
        cobj.details();  
    }  
}
```

```
class Parent{  
    String name;  
    public void details(){  
        name = "Parent";  
        System.out.println(name);  
    }  
}  
  
public class Child extends Parent {  
    String name;  
    public void details(){  
        super.details(); //calling Parent  
        class details() method  
        name = "Child";  
        System.out.println(name);  
    }  
    public static void main(String[] args){  
        Child cobj = new Child();  
        cobj.details();  
    }  
}
```

Inheritance - Types

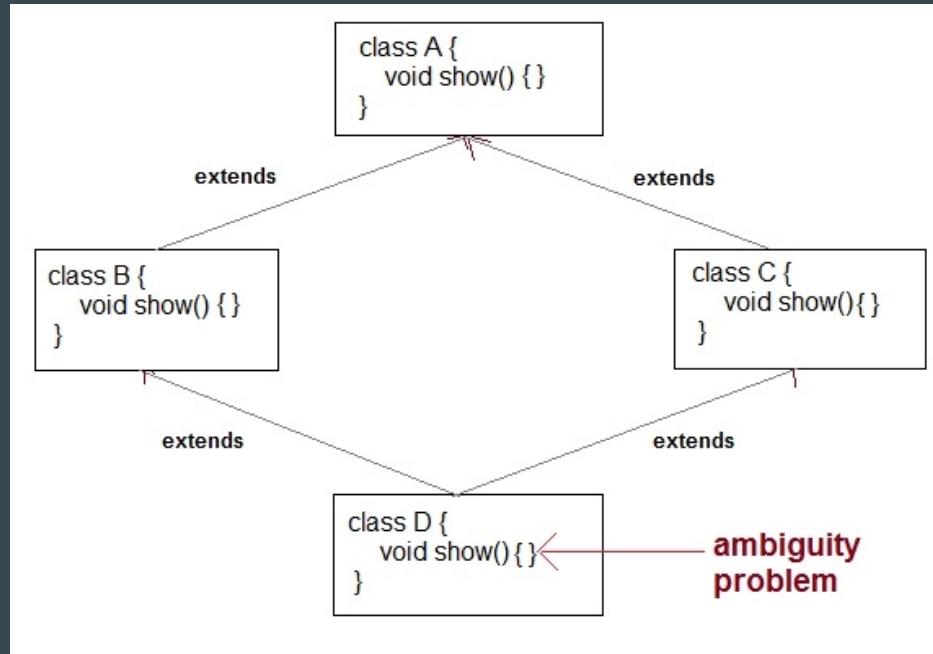
Java mainly supports only three types of inheritance that are listed below.

- Single Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance



multiple inheritance in Java?

- To remove ambiguity.
- To provide more maintainable and clear design.



Inheritance - Method Overriding

- When you extends a class, you can change the behavior of a method in the parent class.
- This is called method overriding.
- This happens when you write in a subclass a method that has the same signature as a method in the parent class.
- If only the name is the same but the list of arguments is not, then it is method overloading.

Inheritance - Method Overriding

```
class Animal {  
  
    private String type;  
    public Animal(String aType) {  
        type = new String(aType);  
    }  
    public String toString() {  
        return "This is a " + type;  
    }  
}
```

```
class Dog extends Animal {  
    private String name;  
    private String breed;  
    public Dog(String aName) {  
        super("Dog");  
        name = aName;  
        breed = "Unknown";  
    }  
    public Dog(String aName, String aBreed) {  
        super("Dog");  
        name = aName;  
        breed = aBreed;  
    }  
    public String toString() {  
        return "It's " + name + " the " + breed;  
    }  
}
```

Inheritance - Type Casting

- With objects, you can cast an instance of a subclass to its parent class.
- Casting an object to a parent class is called upcasting.
- To upcast a Child object, all you need to do is assign the object to a reference variable of type Parent. The parent reference variable cannot access the members that are only available in Child.

```
Child child = new Child();
```

```
Parent parent = child;
```

- Because parent references an object of type Child, you can cast it back to Child. It is called downcasting because you are casting an object to a class down the inheritance hierarchy. Downcasting requires that you write the child type in brackets. For example:

```
Child child2 = (Child) parent;
```

```
class A {  
    public void play() {  
    }  
  
    static void tune(A i) {  
        i.play();  
    }  
}  
  
// Wind objects are instruments  
// because they have the same interface:  
class B extends A {  
}  
  
public class MainClass {  
    public static void main(String[] args) {  
        B flute = new B();  
        A.tune(flute); // Upcasting  
    }  
}
```

Inheritance - Example

```
class A {  
    A(int i) {  
        System.out.println("A constructor");  
    }  
}
```

```
class B extends A {  
    B(int i) {  
        super(i);  
        System.out.println("B constructor");  
    }  
}
```

```
class C extends B {  
    C() {  
        super(11);  
        System.out.println("C constructor");  
    }  
}
```

```
public class D{  
    public static void main(String[] args) {  
        C x = new C();  
    }  
}
```

Inheritance - Example

```
class A {  
    A() {  
        System.out.println("Inside A's  
constructor.");  
    }  
}
```

```
class B extends A {  
    B() {  
        System.out.println("Inside B's  
constructor.");  
    }  
}
```

```
class C extends B {  
    C() {  
        System.out.println("Inside C's  
constructor.");  
    }  
}
```

```
class D{  
    public static void main(String args[]) {  
        C c = new C();  
    }  
}
```

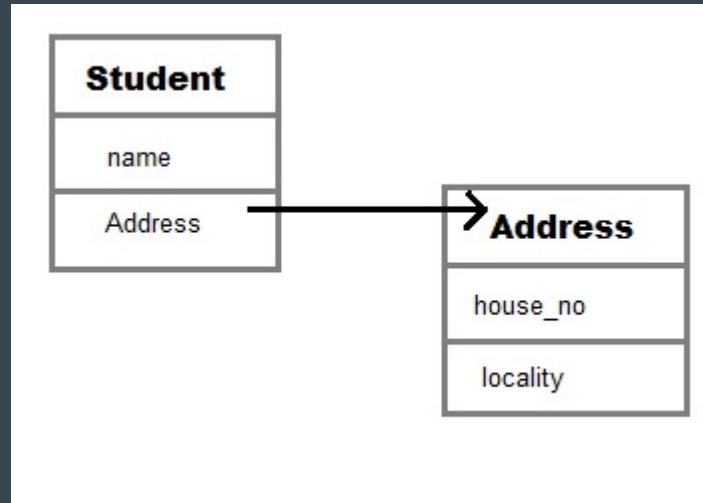
Inheritance - Example

```
class Animal {  
    protected void displayInfo() {  
        System.out.println("I am an animal.");  
    }  
}  
  
class Dog extends Animal {  
    public void displayInfo() {  
        System.out.println("I am a dog.");  
    }  
}  
  
class Main {  
    public static void main(String[] args) {  
        Dog d1 = new Dog();  
        d1.displayInfo();  
    }  
}
```

Aggregation (HAS-A relationship) in Java

- Aggregation is a term which is used to refer one way relationship between two objects. For example, Student class can have reference of Address class but vice versa does not make sense.
- In Java, aggregation represents HAS-A relationship, which means when a class contains reference of another class known to have aggregation.
- The HAS-A relationship is based on usage, rather than inheritance. In other words, class A has-a relationship with class B, if class A has a reference to an instance of class B.
- The main advantage of using aggregation is to maintain code reusability. If an entity has a relation with some other entity than it can reuse code just by referring that.

```
Class Address{  
    int street_no;  
    String city;  
    String state;  
    int pin;  
    Address(int street_no, String city, String state, int pin ){  
        this.street_no = street_no;  
        this.city = city;  
        this.state = state;  
        this.pin = pin;  
    }  
}  
class Student  
{  
    String name;  
    Address ad;  
}
```



Java – Abstract Classes

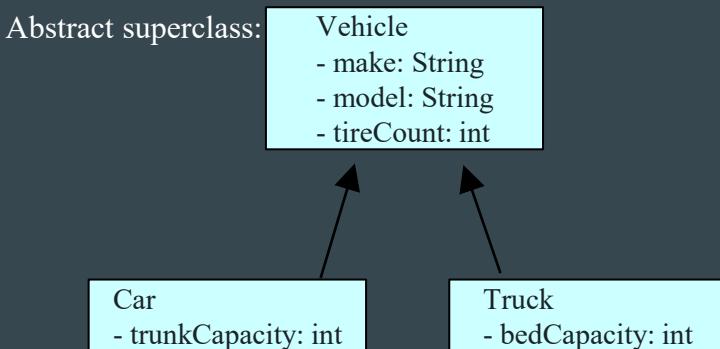
...

What is an Abstract class?

- Abstraction refers to the ability to make a class abstract in OOP.
- Superclasses are created through the process called "generalization"
 - Common features (methods or variables) are factored out of object classifications (ie. classes).
 - Those features are formalized in a class. This becomes the superclass
 - The classes from which the common features were taken become subclasses to the newly created super class
- Often, the superclass does not have a "meaning" or does not directly relate to a "thing" in the real world
- It is an artifact of the generalization process
- Because of this, abstract classes cannot be instantiated
- They act as place holders for abstraction

Abstract Class Example

- In the following example, the subclasses represent objects taken from the problem domain.
- The superclass represents an abstract concept that does not exist "as is" in the real world.



Defining Abstract Classes

- Inheritance is declared using the "extends" keyword
 - If inheritance is not defined, the class extends a class called Object

```
public abstract class Vehicle
{
    private String make;
    private String model;
    private int tireCount;
    [...]
```

Vehicle
- make: String
- model: String
- tireCount: int

```
public class Car extends Vehicle
{
    private int trunkCapacity;
    [...]
```

Car
- trunkCapacity: int

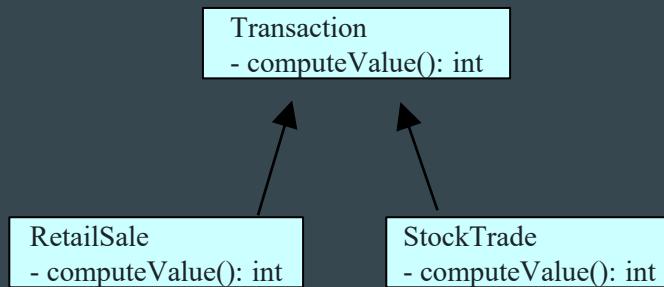
Truck
- bedCapacity: int

```
public class Truck extends Vehicle
{
    private int bedCapacity;
    [...]
```

Often referred to as "concrete" classes

Abstract Method Example

- In the following example, a Transaction's value can be computed, but there is no meaningful implementation that can be defined within the Transaction class.
 - How a transaction is computed is dependent on the transaction's type
(This is polymorphism)



Defining Abstract Methods

```
public abstract class Transaction  
{  
    public abstract int computeValue();
```

Note: no implementation

Transaction
- computeValue(): int

```
public class RetailSale extends Transaction  
{  
    public int computeValue()  
    {  
        [...]
```

RetailSale
- computeValue(): int

```
public class StockTrade extends Transaction  
{  
    public int computeValue()  
    {  
        [...]
```

StockTrade
- computeValue(): int

Abstract Method Example

```
public abstract class Employee
{
    private String name;
    private String address;
    private int number;
    public abstract double computePay();
}
```

//If Salary is extending Employee class then it is required to implement computePay() method as follows:

```
public class Salary extends Employee
{
    private double salary; //Annual salary
    public double computePay()
    {
        System.out.println("Computing salary pay for " + getName());
        return salary/52;
    }
}
```

Abstract Method

- If you want a class to contain a particular method but you want the actual implementation of that method to be determined by child classes, you can declare the method in the parent class as abstract.
- The abstract keyword is also used to declare a method as abstract. An abstract methods consist of a method signature, but no method body.
- The class must also be declared abstract. If a class contains an abstract method, the class must be abstract as well.
- Any child class must either override the abstract method or declare itself abstract.
- A child class that inherits an abstract method must override it. If they do not, they must be abstract, and any of their children must override it.

Abstract Class

- An abstract class cannot be instantiated.
- All other functionality of the class still exists, and its fields, methods, and constructors are all accessed in the same manner.
- You just cannot create an instance of the abstract class.
- If a class is abstract and cannot be instantiated, the class does not have much use unless it is subclassed.
- Use the `abstract` keyword to declare a class abstract. The keyword appears in the class declaration somewhere before the `class` keyword.

An Employee class is an abstract class

```
public class AbstractDemo
```

```
{
```

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

```
{
```

```
        /* Following is not allowed and would raise error */
```

```
        Employee e = new Employee("Rahul");
```

```
}
```

```
}
```

When you compile above class then you would get following error:

```
Employee.java:46: Employee is abstract; cannot be instantiated
```

```
        Employee e = new Employee("Rahul");
```

```
        ^ 1 error!
```

Abstract Class Example

Java - Interface

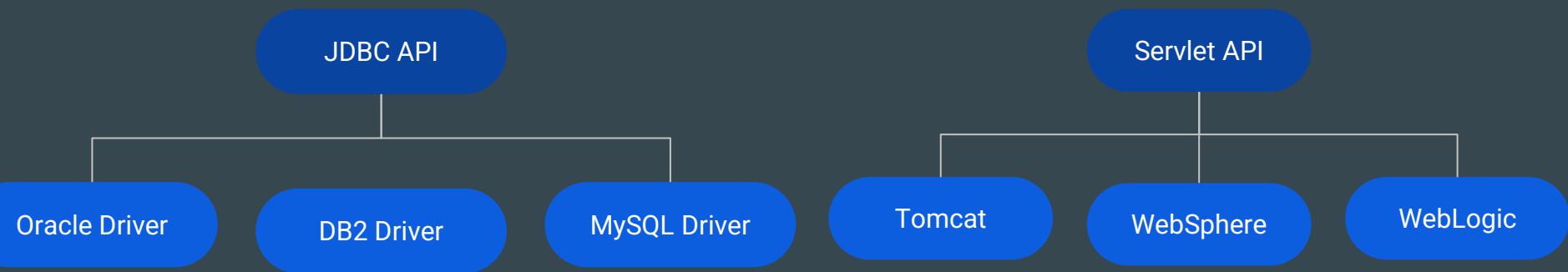
...

What is an Interface?

- An interface is similar to an abstract class with the following exceptions:
 - All methods defined in an interface are abstract. Interfaces can contain no implementation
 - Interfaces cannot contain instance variables. However, they can contain public static final variables (ie. constant class variables)
- Interfaces are declared using the "interface" keyword
 - If an interface is public, it must be contained in a file which has the same name.
- Interfaces are more abstract than abstract classes
- Interfaces are implemented by classes using the "implements" keyword.

Interface

- Any service requirement specification is called an interface.
- Example1: JDBC API.
- Example2: Servlet API.
- From the client point of view an interface defines the set of services what is expecting.
- From the service provider point of view an interface defines the set of services offering.
- Hence an interface is considered as a contract between client and service provider.



Interface

- Inside interface every method is always abstract whether we are declaring or not hence interface is considered as 100% pure abstract class.
- An interface is a collection of abstract methods.
- Whenever we are implementing an interface compulsory for every method of that interface we should provide implementation otherwise we have to declare class as abstract in that case child class is responsible to provide implementation for remaining methods.
- Whenever we are implementing an interface method compulsory it should be declared as public otherwise we will get compile time error.

Declaring an Interface

```
public interface Steerable
{
    public void turnLeft(int degrees);
    public void turnRight(int degrees);
}
```

When a class "implements" an interface, the compiler ensures that it provides an implementation for all methods defined within the interface.

```
public class Car extends Vehicle implements Steerable
{
    public int turnLeft(int degrees)
    {
        [...]
    }

    public int turnRight(int degrees)
    {
        [...]
    }
}
```

Declaring an Interface

In Car.java:

```
public class Car extends Vehicle implements Steerable, Driveable
{
    public int turnLeft(int degrees)
    {
        [...]
    }

    public int turnRight(int degrees)
    {
        [...]
    }

    // implement methods defined within the Driveable interface
}
```

Implementing Interfaces

- A Class can only inherit from one superclass. However, a class may implement several Interfaces
 - The interfaces that a class implements are separated by commas
- Any class which implements an interface must provide an implementation for all methods defined within the interface.
 - NOTE: if an abstract class implements an interface, it NEED NOT implement all methods defined in the interface. HOWEVER, each concrete subclass MUST implement the methods defined in the interface.
- Interfaces can inherit method signatures from other interfaces.

Multiple Inheritance?

- Some people (and textbooks) have said that allowing classes to implement multiple interfaces is the same thing as multiple inheritance
- This is NOT true. When you implement an interface:
 - The implementing class does not inherit instance variables
 - The implementing class does not inherit methods (none are defined)
 - The Implementing class does not inherit associations
- Implementation of interfaces is not inheritance. An interface defines a list of methods which must be implemented.

Abstract Classes Versus Interfaces

- When should one use an Abstract class instead of an interface?
 - If the subclass-superclass relationship is genuinely an "is a" relationship.
 - If the abstract class can provide an implementation at the appropriate level of abstraction

- When should one use an interface in place of an Abstract Class?
 - When the methods defined represent a small portion of a class
 - When the subclass needs to inherit from another class
 - When you cannot reasonably implement any of the methods

Object as a Universal Super Class and Wrapper Classes

Object class

- ❖ Object is a universal superclass in java
- ❖ The **Object class** is the parent class of all the classes in java by default. In other words, it is the topmost class of java.
- ❖ The Object class is beneficial if you want to refer any object whose type you don't know.
 - Notice that parent class reference variable can refer the child class object, known as upcasting.

Object class

- ❖ E.g., there is getObject() method that returns an object but it can be of any type like Employee, Student etc, we can use Object class reference to refer that object. For example:
- ❖ `Object obj = getObject();`
 - we don't know what object would be returned from this method
- ❖ The Object class provides some common behaviors to all the objects such as object can be compared, object can be cloned, object can be notified etc.

Con't

- ❖ Of course, a variable of type Object is only useful as a generic holder for arbitrary values.
- ❖ If you have some knowledge about the original type and then apply a cast:
 - ❖ Employee e = (Employee) obj;
 - ❖ Only primitive types (numbers, characters, and Boolean values) are not objects.
 - ❖ All array types no matter whether they are arrays of objects or arrays of primitive types, are class types that extend the object class.

Method	Description
public final Class getClass()	returns the Class class object of this object. The Class class can further be used to get the metadata of this class.
public int hashCode()	returns the hashcode number for this object.
public boolean equals(Object obj)	compares the given object to this object.
protected Object clone() throws CloneNotSupportedException	creates and returns the exact copy (clone) of this object.
public String toString()	returns the string representation of this object.
public final void notify()	wakes up single thread, waiting on this object's monitor.
public final void notifyAll()	wakes up all the threads, waiting on this object's monitor.
public final void wait(long timeout) throws InterruptedException	causes the current thread to wait for the specified milliseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait(long timeout,int nanos) throws InterruptedException	causes the current thread to wait for the specified milliseconds and nanoseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait() throws InterruptedException	causes the current thread to wait, until another thread notifies (invokes notify() or notifyAll() method).
protected void finalize() throws Throwable	is invoked by the garbage collector before object is

java.lang.Object

- ❖ The Java super class **java.lang.Object** has two very important methods defined in it. They are -
 - public boolean equals(Object obj)
 - public int hashCode()
- ❖ The equals() method checks if some other object passed to it as an argument is *equal* to the object on which this method is invoked.
- ❖ The default implementation of this method in Object class simply checks if two object references x and y refer to the same object.
 - i.e. It checks if $x == y$.

public Boolean equals(Object obj)

- ❖ Object class has no data members that define its state, So, it simply performs shallow comparison.
- ❖ However, the classes providing their own implementations of the equals method are supposed to perform a "deep comparison";
 - by actually comparing the relevant data members.
- ❖ The equals method for class Object implements the most discriminating possible equivalence relation on objects;
 - that is, for any reference values x and y, this method returns true if and only if x and y refer to the same object.

What JDK 1.4 API documentation says about the equals method?

- ❖ The equals method implements an equivalence relation: It is **reflexive**: for any reference value x , $x.equals(x)$ should return true.
- ❖ It is **symmetric**: for any reference values x and y , $x.equals(y)$ should return true if and only if $y.equals(x)$ returns true.
- ❖ It is **transitive**: for any reference values x , y , and z , if $x.equals(y)$ returns true and $y.equals(z)$ returns true, then $x.equals(z)$ should return true.
- ❖ It is **consistent**: for any reference values x and y , multiple invocations of $x.equals(y)$ consistently return true or consistently return false, provided no information used in equals comparisons on the object is modified.
- ❖ For any non-null reference value x , $x.equals(null)$ should return false.

public int hashCode()

- ❖ This method returns the hash code value for the object on which this method is invoked.
- ❖ It returns the hash code value as an integer and is supported for the benefit of hashing based collection classes such as Hashtable, HashMap, HashSet etc.
- ❖ This method must be overridden in every class that overrides the equals method.

The general contract of hashCode

- ❖ Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified.
 - This integer need not remain consistent from one execution of an application to another execution of the same application.

The general contract of hashCode

- ❖ If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.
- ❖ It is not required that if two objects are unequal according to the equals(java.lang.Object) method, then calling the hashCode method on each of the two objects must produce distinct integer results.

The `toString` Method

- ❖ Object's `toString` method returns a String representation of the object.
- ❖ You can use `toString` to display an object.
 - For example, you could display a String representation of the current Thread like this:
 - `System.out.println(Thread.currentThread().toString());`
- ❖ The String representation for an object is entirely dependent on the object.
- ❖ The String representation of an Integer object is the integer value displayed as text.
- ❖ The `toString` method is very useful for debugging and you should override this method in all your classes.

Con't

- ❖ The `toString` method of the `Point` class returns a string:

- `java.awt.Point[x=10,y=20]`

```
Class Employee {
```

```
    public String toString() {
```

```
        return getClass().getName() + "[name=" + name + ",salary=" + salary +  
               ",hireDay=" + hireDay + "]";
```

```
}
```

```
}
```

```
class Manager extends Employee {
```

```
    public String toString() {
```

```
        return super.toString() + "[bonus=" + bonus + "]";
```

```
}
```

```
}
```

Output: Manager object will be printed as

`Manager[name=...,salary=...,hireDay=...][bonus=...]`

Con't

- ❖ Whenever an object is concatenated with a string by the "+" operator, the Java compiler automatically invokes the `toString` method to obtain a string representation of the object.
- ❖ Example:
 - `Point p = new Point(10, 20);`
 - `String message = "The current position is " + p;`
 - `// automatically invokes p.toString()`
- ❖ `x.toString()`, you can write `"" + x`
 - `x` can be of primitive type.
- ❖ If `x` is any object and you call
 - `System.out.println(x);`
 - calls `x.toString()` and prints the resulting string.

Wrapper Classes

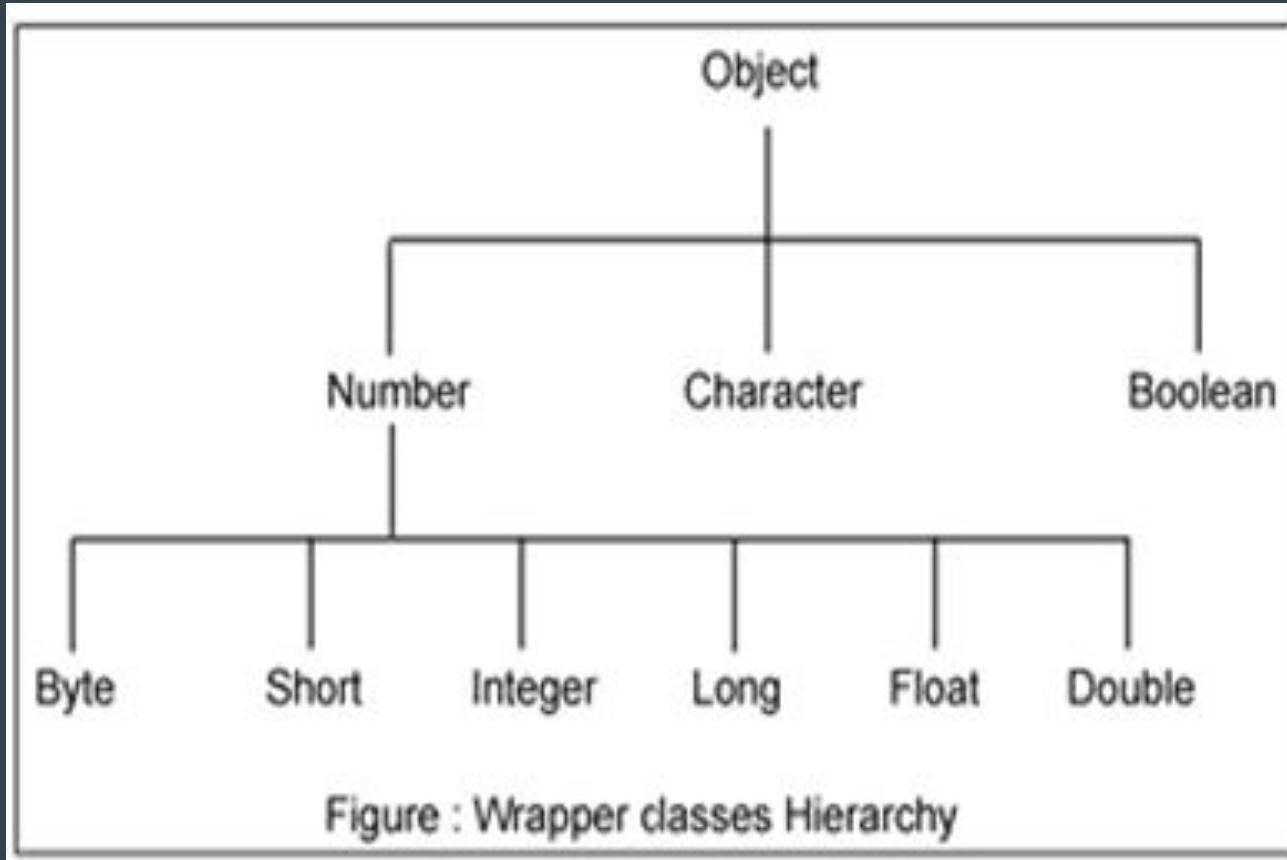
- ❖ **Wrapper class in java** provides the mechanism *to convert primitive into object and object into primitive.*
- ❖ Since J2SE 5.0, **autoboxing** and **unboxing** feature converts primitive into object and object into primitive automatically.
- ❖ The automatic conversion of primitive into object is known as autoboxing and vice-versa unboxing.

Wrapper Classes

- ◆ The list of eight wrapper classes are given below:

Primitive Type	Wrapper class
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

Wrapper class hierarchy



Wrapper class Example: Primitive to Wrapper

```
◊ public class WrapperExample1{  
    public static void main(String args[]){  
        //Converting int into Integer  
        int a=20;  
        Integer i=Integer.valueOf(a); //converting int into Integer  
        //autoboxing, now compiler will write Integer.valueOf(a) internally  
        Integer j=a;  
        System.out.println (a+" "+i+" "+j);  
    }  
}
```

Wrapper class Example: Wrapper to Primitive

```
◊ public class WrapperExample2{  
    public static void main(String args[]){  
        //Converting Integer to int  
        Integer a=new Integer(3);  
        int i=a.intValue();//converting Integer to int  
        //unboxing, now compiler will write a.intValue() internally  
        int j=a;  
        System.out.println(a+" "+i+" "+j);  
    }  
}
```

Object Cloning

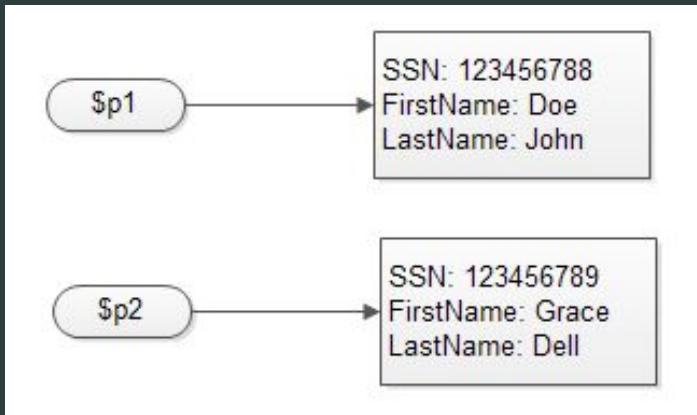
Object Cloning

- ◆ We've seen before that making a copy of an object variable just makes a second reference to that object, it doesn't "copy" it

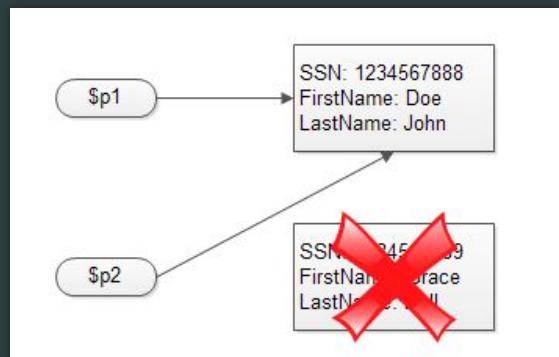
```
Person somePerson=new Person("Student1");
Person otherPerson;
otherPerson=somePerson; //both otherPerson and somePerson point to the
                      //same object. A change in either will occur in
                      //both.
```

- ◆ This just creates two references to the same exact thing. This isn't very useful.

Clone



```
$p2 = $p1;
```



An Example

- In this example, both Date references point to the same Object. A change in one will effect both

```
Date someDate=new Date(1234567);
Date newDate=someDate;
newDate.setTime(newDate.getTime()+1);
System.out.println(someDate.getTime()); //outputs 1234568
System.out.println(newDate.getTime()); //outputs 1234568
```

- In this example, we create a full new object that starts in the same state. Now they are two unique objects

```
Date someDate=new Date(1234567);
Date newDate=(Date)someDate.clone();
newDate.setTime(newDate.getTime()+1);
System.out.println(someDate.getTime()); //outputs 1234567
System.out.println(newDate.getTime()); //outputs 1234568
```

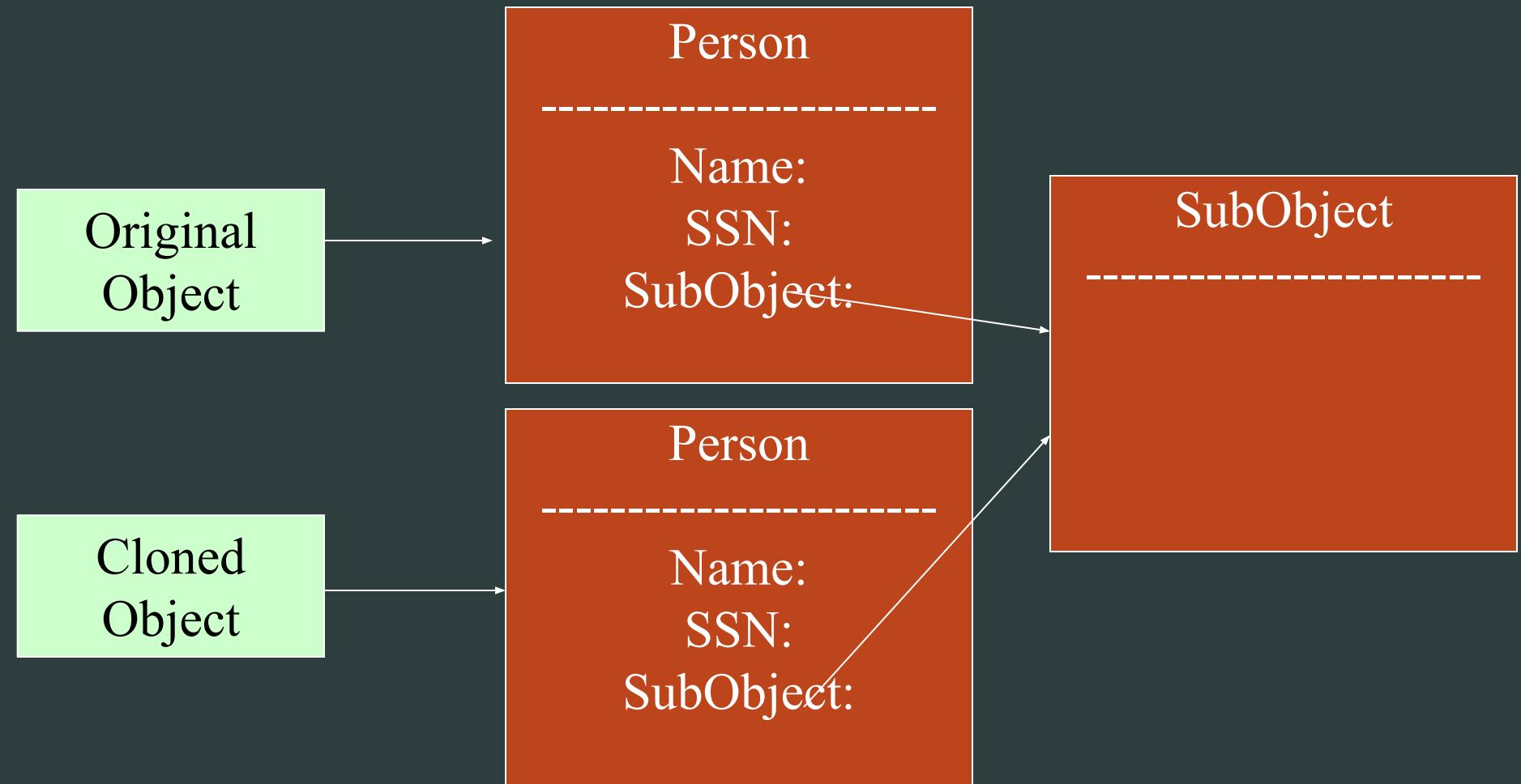
The Clone Method

- ❖ This works with Date objects because Date objects implement the Cloneable interface.
- ❖ The `clone()` method is actually inherited from the Object superclass, and is protected by default.
- ❖ Since `clone()` is already implemented in Object, it means that the Cloneable interface is a *tagged interface* – it actually contains no methods.
- ❖ Since `clone()` is inherited from Object, it doesn't know anything about its subclasses (which is every class in Java).
- ❖ Is this a problem?

The Clone Method

- ❖ Since the `clone()` method belongs to the `Object` superclass, it doesn't know anything about the object it's copying.
- ❖ Because of this, it can only do a field-by-field copy of the `Object`.
- ❖ This is fine if all the fields are primitive types or immutable objects, but if the `Object` contained other mutable sub objects, only the reference will be copied, and the objects will share data.
- ❖ This is called shallow copying.

The Clone method



The Cloning Decision

- ❖ For every class, you need to decide whether or not
 - 1) The default shallow copy clone is acceptable
 - 2) You need to deep-copy the class
 - 3) The object is not allowed to be cloned
- ❖ For either of the first two choices, you must
 - Implement the Cloneable interface
 - Redefine the clone() method with the public access modifier

- ❖ The **object cloning** is a way to create exact copy of an object. The `clone()` method of Object class is used to clone an object.
- ❖ The **java.lang.Cloneable interface** must be implemented by the class whose object clone we want to create. If we don't implement Cloneable interface, `clone()` method generates **CloneNotSupportedException**.
- ❖ The **clone() method** is defined in the Object class.

A Cloning Example

- ◆ To implement the default clone, you must create a `clone()` method with a public access modifier and add it to the class

```
public class cloneExample implements Cloneable {  
    public Object clone() {  
        try {  
            return super.clone();  
        }  
        catch (CloneNotSupportedException exp)  
        {  
            return null; }  
    }  
}
```

Deep Copy Vs Shallow Copy

- ❖ Deep Copy vs Shallow Copy
 - Shallow copy is the method of copying an object and is followed by default in cloning. In this method, the fields of an old object X are copied to the new object Y. While copying the object type field the reference is copied to Y i.e object Y will point to the same location as pointed out by X. If the field value is a primitive type it copies the value of the primitive type.
 - Therefore, any changes made in referenced objects in object X or Y will be reflected in other objects.
 - Shallow copies are cheap and simple to make. In the above example, we created a shallow copy of the object.

Deep Copy Vs Shallow Copy

- ❖ Usage of clone() method – Deep Copy
- If we want to create a deep copy of object X and place it in a new object Y then a new copy of any referenced objects fields are created and these references are placed in object Y. This means any changes made in referenced object fields in object X or Y will be reflected only in that object and not in the other. In the below example, we create a deep copy of the object.
- A deep copy copies all fields and makes copies of dynamically allocated memory pointed to by the fields. A deep copy occurs when an object is copied along with the objects to which it refers.

Deep copying

- ◊ If you need to make a deep copy of a class, your clone method must copy any mutable objects within the class. This is more work, but necessary.
- ◊ Any objects within your class that implement Cloneable themselves should be cloned – if they do not, use the new operator to call the constructors, and use = from Strings and primitive types.

Deep copying

```
public class cloneExample implements Cloneable {  
    private Date someDate;      //cloneable  
    private JSlider theSlider;   //not cloneable  
    public Object clone() {  
        try {  
            cloneExample cloned=(cloneExample)super.clone();  
            cloned.someDate=(Date)someDate.clone();  
            cloned.theSlider=new  
JSlider(theSlider.getMinimum(),theSlider.getMaximum());  
            return cloned;  
        }  
        catch (CloneNotSupportedException e) { return null;}  
    }  
}
```

Cloning via Serialization

- ❖ As you can see, cloning can be tedious.
- ❖ There is a way to automagically clone something using something called *Serialization*. (we'll cover Serialization later)
- ❖ Java provides a way to save objects and their state to files. (for use in stuff like RMI, among others)
- ❖ We can use this as a way to make a Deep Copy without going through rewriting the Cloneable() method for every class
- ❖ To do this, our class must implement the Cloneable AND Serializable interfaces.

Serialized cloning

```
public class Serialtest implements Cloneable, Serializable {  
    public Object clone() {  
        try {  
            //write out a copy of the object to a byte array  
            ByteArrayOutputStream bout= new ByteArrayOutputStream();  
            ObjectOutputStream out = new ObjectOutputStream(bout);  
            out.writeObject(this);  
            out.close();  
            //read a clone of the object from the byte array  
            ByteArrayInputStream bin = new  
            ByteArrayInputStream(boat.toByteArray());  
            ObjectInputStream in = new ObjectInputStream(bin);  
            Object ret = in.readObject();  
            in.close();  
            return ret;  
        }  
        catch (Exception exp) { return null; }  
    }  
}
```

Serialized cloning

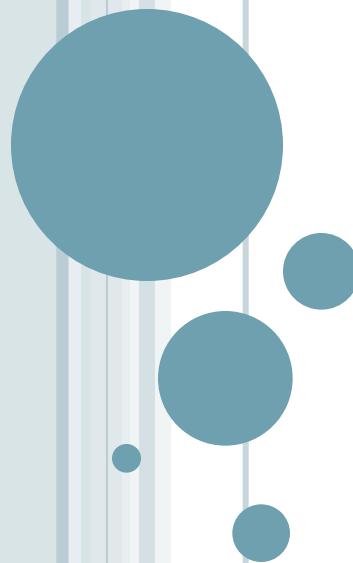
- ❖ This produces a deep copy of object, without all the tedious work
- ❖ So why not just do this every time? Why didn't the Java creators just implement this themselves?
- ❖ Mainly because this way will usually be much slower than a clone method that explicitly constructs a new object and copies or clones the fields itself.

- ❖ The **clone()** method saves the extra processing task for creating the exact copy of an object. If we perform it by using the new keyword, it will take a lot of processing time to be performed that is why we use object cloning.
- ❖ Advantage
 - You don't need to write lengthy and repetitive codes. Just use an abstract class with a 4- or 5-line long clone() method.
 - It is the easiest and most efficient way for copying objects, especially if we are applying it to an already developed or an old project. Just define a parent class, implement Cloneable in it, provide the definition of the clone() method and the task will be done.
 - Clone() is the fastest way to copy array.

Disadvantages

- To use the `Object.clone()` method, we have to change a lot of syntaxes to our code, like implementing a `Cloneable` interface, defining the `clone()` method and handling `CloneNotSupportedException`, and finally, calling `Object.clone()` etc.
- We have to implement `cloneable` interface while it doesn't have any methods in it. We just have to use it to tell the JVM that we can perform `clone()` on our object.
- `Object.clone()` is protected, so we have to provide our own `clone()` and indirectly call `Object.clone()` from it.
- `Object.clone()` doesn't invoke any constructor so we don't have any control over object construction.
- If you want to write a clone method in a child class then all of its superclasses should define the `clone()` method in them or inherit it from another parent class. Otherwise, the `super.clone()` chain will fail.
- `Object.clone()` supports only shallow copying but we will need to override it if we need deep cloning.

JAVA REFLECTION



JAVA LOOKING AT JAVA

- One of the unusual capabilities of Java is that a program can examine itself
 - You can determine the class of an object
 - You can find out all about a class: its access modifiers, superclass, fields, constructors, and methods
 - You can find out what is there in an interface
 - Even if you don't know the names of things when you write the program, you can:
 - Create an instance of a class
 - Get and set instance variables
 - Invoke a method on an object
 - Create and manipulate arrays



WHAT IS REFLECTION FOR?

- In “normal” programs you don’t need reflection
- You *do* need reflection if you are working with programs that process programs
- Typical examples:
 - A class browser
 - A debugger
 - A GUI builder
 - An IDE, such Netbeans or eclipse
 - A program to grade student programs



USING REFLECTION TO ANALYZE THE CAPABILITIES OF CLASSES

- The three classes Field, Method, and Constructor in the java.lang.reflect package describe the fields, methods, and constructors of a class, respectively
- All three classes have a method called getName() that returns the name of the item. The Field class has a method getType() that returns an object, again of type Class, that describes the field type
- The Method and Constructor classes have methods to report the types of the parameters, and the Method class also reports the return type



CONT..

- All three of these classes also have a method called `getModifiers()` that returns an integer, with various bits turned on and off, that describes the modifiers used, such as `public` and `static`
- Use methods like `isPublic()`, `isPrivate()`, or `isFinal()` in the `Modifier` class to tell whether a method or constructor was `public`, `private`, or `final`. All you have to do is have the appropriate method in the `Modifier` class work on the integer that `getModifiers()` returns. You can also use the `Modifier.toString()` method to print the modifiers

CONT..

- The `getFields()`, `getMethods()`, and `getConstructors()` methods of the `Class` class return arrays of the public fields, methods, and constructors that the class supports. This includes public members of superclasses
- The `getDeclaredFields()`, `getDeclaredMethods()`, and `getDeclaredConstructors()` methods of the `Class` class return arrays consisting of all fields, operations, and constructors that are declared in the class. This includes private and protected members, but not members of superclasses

THE **CLASS** CLASS

- While your program is running, the Java runtime system always maintains what is called runtime type identification on all objects
- This information keeps track of the class to which each object belongs. Runtime type information is used by the virtual machine to select the correct methods to execute
- You can also access this information by working with a special Java class. The class that holds this information is called, somewhat confusingly, Class. The getClass() method in the Object class returns an instance of Class type



THE CLASS CLASS

- To find out about a class, first get its **Class** object
 - If you have an object **obj**, you can get its class object with
Class c = obj.getClass();
 - You can get the class object for the superclass of a Class **c** with
Class sup = c.getSuperclass();
 - If you know the name of a class (say, **Button**) at compile time, you can get its class object with
Class c = Button.class;
 - If you know the name of a class at run time (in a String variable **str**), you can get its class object with
Class c = class.forName(str);

GETTING THE CLASS NAME

- If you have a class object c, you can get the name of the class with `c.getName()`
- `getName` returns the fully qualified name; that is,

```
Class c = Button.class;
String s = c.getName();
System.out.println(s);
```

will print
`java.awt.Button`
- Class `Class` and its methods are in `java.lang`, which is always imported and available



GETTING ALL THE SUPERCLASSES

- `getSuperclass()` returns a **Class** object (or **null** if you call it on **Object**, which has no superclass)
- The following code is from the Sun tutorial:
 - `static void printSuperclasses(Object o) {`
 - `Class subclass = o.getClass();`
 - `Class superclass = subclass.getSuperclass();`
 - `while (superclass != null) {`
 - `String className = superclass.getName();`
 - `System.out.println(className);`
 - `subclass = superclass;`
 - `superclass = subclass.getSuperclass();`
 - `}`

GETTING THE CLASS MODIFIERS I

- The modifiers (e.g., public, final, abstract etc.) of a **Class** object is encoded in an int and can be queried by the method **getModifiers()**.
- To decode the **int** result, we need methods of the **Modifier** class, which is in **java.lang.reflect**, so:
import java.lang.reflect.*;
- Then we can do things like:
**if (Modifier.isPublic(m))
 System.out.println("public");**



GETTING THE CLASS MODIFIERS II

□ **Modifier** contains these methods (among others):

- `public static boolean isAbstract(int)`
- `public static boolean isFinal(int)`
- `public static boolean isInterface(int)`
- `public static boolean isPrivate(int)`
- `public static boolean isProtected(int)`
- `public static boolean isPublic(int)`
- `public static String toString(int)`

□ This will return a string such as
"public final synchronized strictfp"



GETTING INTERFACES

- A class can implement zero or more interfaces
- `getInterfaces()` returns an *array* of **Class** objects
- Ex:

```
static void printInterfaceNames(Object o) {  
    Class c = o.getClass();  
    Class[] theInterfaces = c.getInterfaces();  
    for (Class inf: interfaces) {  
        System.out.println(inf.getName());    }}
```

- Note the convenience of enhanced for-loop



EXAMINING CLASSES AND INTERFACES

- The class `Class` represents both classes and interfaces
- To determine if a given `Class` object `c` is an interface, use `c.isInterface()`
- To find out more about a class object, use:
 - `getModifiers()`
 - `getFields()` // "fields" == "instance variables"
 - `getConstructors()`
 - `getMethods()`
 - `isArray()`



GETTING FIELDS

- `public Field[] getFields() throws SecurityException`
 - Returns an array of *public* Fields (including inherited fields).
 - The length of the array may be zero
 - The fields are not returned in any particular order
 - Both locally defined and inherited instance variables are returned, but *not* static variables.
- `public Field getField(String name)`
`throws NoSuchFieldException, SecurityException`
 - Returns the named *public* Field
 - If no immediate field is found, the superclasses and interfaces are searched recursively

USING FIELDS, I

- If *f* is a **Field** object, then
 - *f.getName()* returns the simple name of the field
 - *f.getType()* returns the type (**Class**) of the field
 - *f.getModifiers()* returns the **Modifier**s of the field
 - *f.toString()* returns a String containing access modifiers, the type, and the fully qualified field name
 - Example: `public java.lang.String Person.name`
 - *f.getDeclaringClass()* returns the **Class** in which this field is declared
 - note: `getFields()` may return superclass fields.



USING FIELDS, II

- The fields of a particular object *obj* may be accessed with:
 - `boolean f.getBoolean(obj)`, `int f.getInt(obj)`, `double f.getDouble(obj)`, etc., return the value of the field, assuming it is that type or can be widened to that type
 - `Object f.get(obj)` returns the value of the field, assuming it is an Object
 - `void f.set(obj, value)`, `void f.setBoolean(obj, bool)`, `void f.setInt(obj, i)`, `void f.setDouble(obj, d)`, etc. set the value of a field



GETTING CONSTRUCTORS OF A CLASS

- if c is a Class, then
- `c.getConstructors()` : `Constructor[]` return an array of all public constructors of class c.
- `c.getConstructor(Class ... paramTypes)` returns a constructor whose parameter types match those given paramTypes.

Ex:

- `String.class.getConstructors().length`
➤ 15;
- `String.class.getConstructor(char[].class, int.class, int.class).toString()`
➤ `String(char[], int,int).`



CONSTRUCTORS

- If `c` is a `Constructor` object, then
 - `c.getName()` returns the name of the constructor, as a `String` (this is the same as the name of the class)
 - `c.getDeclaringClass()` returns the `Class` in which this constructor is declared
 - `c.getModifiers()` returns the `Modifier`s of the constructor
 - `c.getParameterTypes()` returns an array of `Class` objects, in declaration order
 - `c.newInstance(Object... initargs)` creates and returns a new instance of class `c`
 - Arguments that should be primitives are automatically unwrapped as needed



EXAMPLE

- Constructor c = String.class.getConstructor(char[].class, int.class, int.class).toString()
- String(char[], int,int).

- String s = c.newInstance(
 new char[] {'a','b','c','d'}, 1, 2);
- assert s == “bc”;



METHODS

- **public Method[] getMethods()
throws SecurityException**
 - Returns an array of **Method** objects
 - These are the *public member* methods of the class or interface, including inherited methods
 - The methods are returned in no particular order
- **public Method getMethod(String name,
Class... parameterTypes)
throws NoSuchMethodException, SecurityException**



METHOD METHODS, I

- **getDeclaringClass()**
 - Returns the **Class** object representing the class or interface that declares the method represented by this **Method** object
- **getName()**
 - Returns the name of the method represented by this **Method** object, as a **String**
- **getModifiers()**
 - Returns the Java language modifiers for the method represented by this **Method** object, as an integer
- **getParameterTypes()**
 - Returns an array of **Class** objects that represent the formal parameter types, in declaration order, of the method represented by this **Method** object

METHOD METHODS, II

□ `getReturnType()`

- Returns a `Class` object that represents the formal return type of the method represented by this `Method` object

□ `toString()`

- Returns a `String` describing this `Method` (typically pretty long)

□ `public Object invoke(Object obj, Object... args)`

- Invokes the underlying method represented by this `Method` object, on the specified object with the specified parameters
- Individual parameters are automatically unwrapped to match primitive formal parameters

EXAMPLES OF(INVOKE()

- “abcdefg”.length()
➤ 7
- Method lengthMethod =
String.class.getMethod(“length”);
- lengthMethod.invoke(“abcdefg”)
➤ 7
- “abcdefg”.substring(2, 5)
➤ cde
- Method substringMethod = String.class.getMethod (“substring”, int.class, Integer.TYPE);
- substringEMthod.invoke(“abcdefg”, 2, new Integer(5))
➤ cde

ARRAYS I

- To determine whether an object `obj` is an array,
 - Get its class `c` with `Class c = obj.getClass();`
 - Test with `c.isArray()`
- To find the type of components of the array,
 - `c.getComponentType()`
 - Returns `null` if `c` is not the class of an array
- Ex:
 - `int[].class.isArray() == true ;`
 - `int[].class.getComponentType() == int.class`

ARRAYS II

- The **Array** class in `java.lang.reflect` provides *static* methods for working with arrays
- To create an array,
- **Array.newInstance(Class componentType, int size)**
 - This returns, as an **Object**, the newly created array
 - You can cast it to the desired type if you like
 - The **componentType** may itself be an array
 - This would create a multiple-dimensioned array
 - The limit on the number of dimensions is usually 255
- **Array.newInstance(Class componentType, int... sizes)**
 - This returns, as an **Object**, the newly created multidimensional array (with `sizes.length` dimensions)



EXAMPLES

- The following two objects are of the same type:
 - `new String[10]`
 - `Array.newInstance(String.class, 10)`
- The following two objects are of the same type:
 - `new String[10][20]`
 - `Array.newInstance(String.class, 10, 20)`



ARRAYS III

- To get the value of array elements,
 - `Array.get(Object array, int index)` returns an **Object**
 - `Array.getBoolean(Object array, int index)` returns a **boolean**
 - `Array.getByte(Object array, int index)` returns a **byte**
 - etc.
- To store values into an array,
 - `Array.set(Object array, int index, Object value)`
 - `Array.setInt(Object array, int index, int i)`
 - `Array.setFloat(Object array, int index, float f)`
 - etc.



EXAMPLES

- `a = new int[] {1,2,3,4};`
- `Array.getInt(a, 2) // □ 3`
- `Array.setInt(a, 3, 5) // a = {1,2,3, 5 }.`

- `s = new String[] { “ab”, “bc”, “cd” };`
- `Array.get(s, 1) // □ “bc”`
- `Array.set(s, 1, “xxx”) // s[1] = “xxx”`



GETTING NON-PUBLIC MEMBERS OF A CLASS

- All `getXXX()` methods of `Class` mentioned above return only `public` members of the target (as well as ancestor) classes, but they cannot return non-public members.
- There are another set of `getDeclaredXXX()` methods in `Class` that will return all (`even private or static`) members of target class but no inherited members are included.
- `getDeclaredConstructors()`,
`defDeclaredConstrucor(Class...)`
- `getDeclaredFields()`,
`getDeclaredField(String)`
- `getDeclaredmethods()`,
`getDeclaredMethod(String, Class...)`

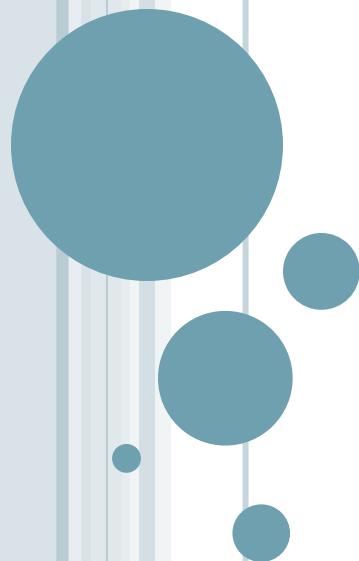


CONCLUDING COMMENTS

- Many of these methods throw exceptions not described here
 - For details, see the Java API
- Reflection isn't used in “normal” programs, but when you need it, it's indispensable
- Studying the java reflection package gives you a chance to review the basics of java class structure.



NESTED CLASSES



NESTED CLASSES

- The Java programming language allows you to define a class within another class. Such a class is called a *nested class* and is illustrated here:
- ```
class OuterClass {
 ...
 class NestedClass {
 ...
 }
}
```
- **Terminology:** Nested classes are divided into two categories: static and non-static. Nested classes that are declared static are simply called *static nested classes*. Non-static nested classes are called *inner classes*.

```
class OuterClass {
 ...
 static class StaticNestedClass {
 ...
 }
 class InnerClass {
 ...
 }
}
```

- A nested class is a member of its enclosing class. Non-static nested classes (inner classes) have access to other members of the enclosing class, even if they are declared private. Static nested classes do not have access to other members of the enclosing class. As a member of the Outer Class, a nested class can be declared private, public, protected, or *package private*. (Recall that outer classes can only be declared public or *default*.)

# WHY USE NESTED CLASSES?

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

## STATIC NESTED CLASSES

- As with class methods and variables, a static nested class is associated with its outer class. And like static class methods, a static nested class cannot refer directly to instance variables or methods defined in its enclosing class — it can use them only through an object reference.
- A static nested class interacts with the instance members of its outer class (and other classes) just like any other top-level class. In effect, a static nested class is behaviorally a top-level class that has been nested in another top-level class for packaging convenience.



## CONT..

- Static nested classes are accessed using the enclosing class name:
- OuterClass.StaticNestedClass
- For example, to create an object for the static nested class, use this syntax:
- OuterClass.StaticNestedClass nestedObject = new OuterClass.StaticNestedClass();



# INNER CLASSES

- As with instance methods and variables, an inner class is associated with an instance of its enclosing class and has direct access to that object's methods and fields. Also, because an inner class is associated with an instance, it cannot define any static members itself.
- Objects that are instances of an inner class exist *within* an instance of the outer class. Consider the following classes:

```
Class OuterClass {
```

```
...
```

```
 class InnerClass {
```

```
 ...
```

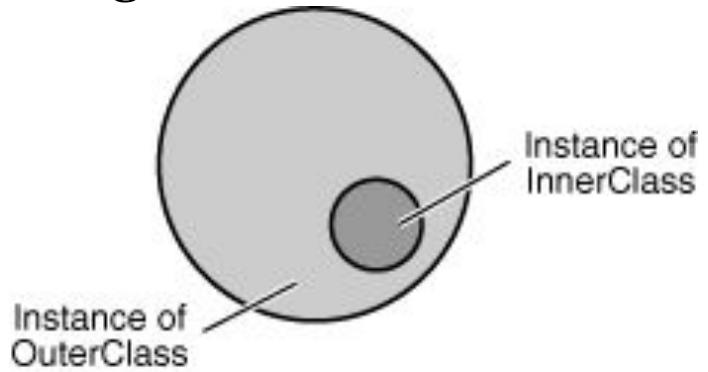
```
 }
```

```
}
```



## CONT..

- An instance of InnerClass can exist only within an instance of OuterClass and has direct access to the methods and fields of its enclosing instance. The next figure illustrates this idea.



- An Instance of InnerClass Exists Within an Instance of OuterClass
- To instantiate an inner class, you must first instantiate the outer class. Then, create the inner object within the outer object with this syntax:

## CONT..

- OuterClass.InnerClass              innerObject              =  
outerObject.new InnerClass();
- Additionally, there are two special kinds of inner classes: local classes and anonymous classes (also called anonymous inner classes).



# LOCAL AND ANONYMOUS INNER CLASSES

- There are two additional types of inner classes. You can declare an inner class within the body of a method. Such a class is known as a *local inner class*. You can also declare an inner class within the body of a method without naming it. These classes are known as *anonymous inner classes*. You will encounter such classes in advanced Java programming.

## □ **Modifiers**

- You can use the same modifiers for inner classes that you use for other members of the outer class. For example, you can use the access specifiers — private, public, and protected — to restrict access to inner classes, just as you do to other class members.

## SUMMARY OF NESTED CLASSES

- A class defined within another class is called a nested class. Like other members of a class, a nested class can be declared static or not. A nonstatic nested class is called an inner class. An instance of an inner class can exist only within an instance of its enclosing class and has access to its enclosing class's members even if they are declared private.



## CONT..

- The following table shows the types of nested classes:

| Types of Nested Classes  |                                    |       |
|--------------------------|------------------------------------|-------|
| Type                     | Scope                              | Inner |
| static nested class      | member                             | no    |
| inner [non-static] class | member                             | yes   |
| local class              | local                              | yes   |
| anonymous class          | only the point where it is defined | yes   |



# Applet

- An applet is a Java program that runs in a Web browser.

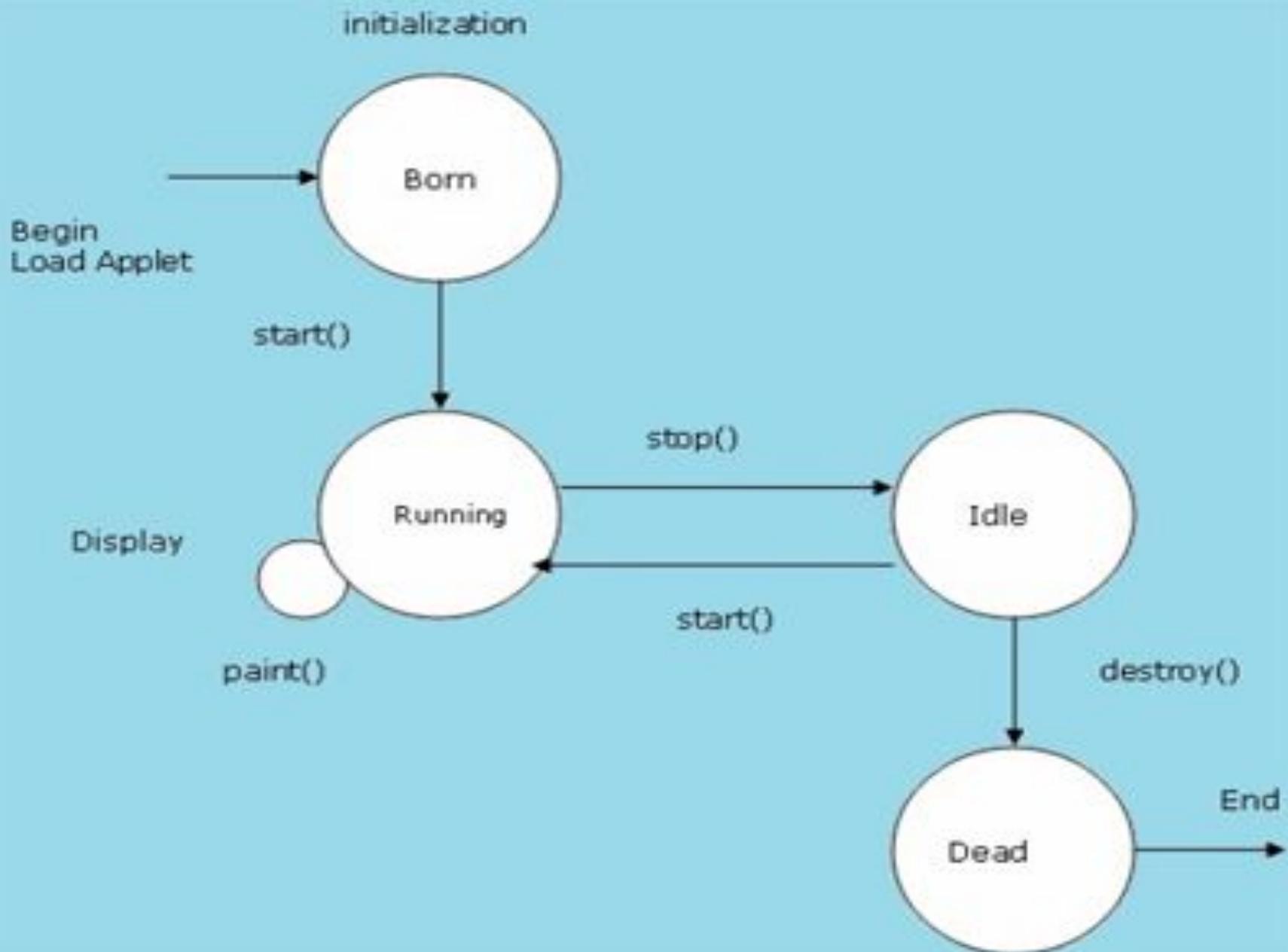
# Difference Between Applet and standalone Java Application

- An applet is a Java class that extends the `java.applet.Applet` class.
- A `main()` method is not invoked on an applet, and an applet class will not define `main()`.
- Applets are designed to be embedded within an HTML page.
- When a user views an HTML page that contains an applet, the code for the applet is downloaded to the user's machine.
- A JVM is required to view an applet. The JVM can be either a plug-in of the Web browser or a separate runtime environment.
- The JVM on the user's machine creates an instance of the applet class and invokes various methods during the applet's lifetime.
- Applets have strict security rules that are enforced by the Web browser. The security of an applet is often referred to as sandbox security, comparing the applet to a child playing in a sandbox with various rules that must be followed.
- Other classes that the applet needs can be downloaded in a single Java Archive (JAR) file.

# Applet LifeCycle

- **init:** This method is intended for whatever initialization is needed for your applet. It is called after the param tags inside the applet tag have been processed.
- **start:** This method is automatically called after the browser calls the init method. It is also called whenever the user returns to the page containing the applet after having gone off to other pages.
- **stop:** This method is automatically called when the user moves off the page on which the applet sits. It can, therefore, be called repeatedly in the same applet.
- **destroy:** This method is only called when the browser shuts down normally. Because applets are meant to live on an HTML page, you should not normally leave resources behind after a user leaves the page that contains the applet.
- **paint:** Invoked immediately after the start() method, and also any time the applet needs to repaint itself in the browser. The paint() method is actually inherited from the java.awt.

# Applet LifeCycle



# A "Hello, World" Applet:

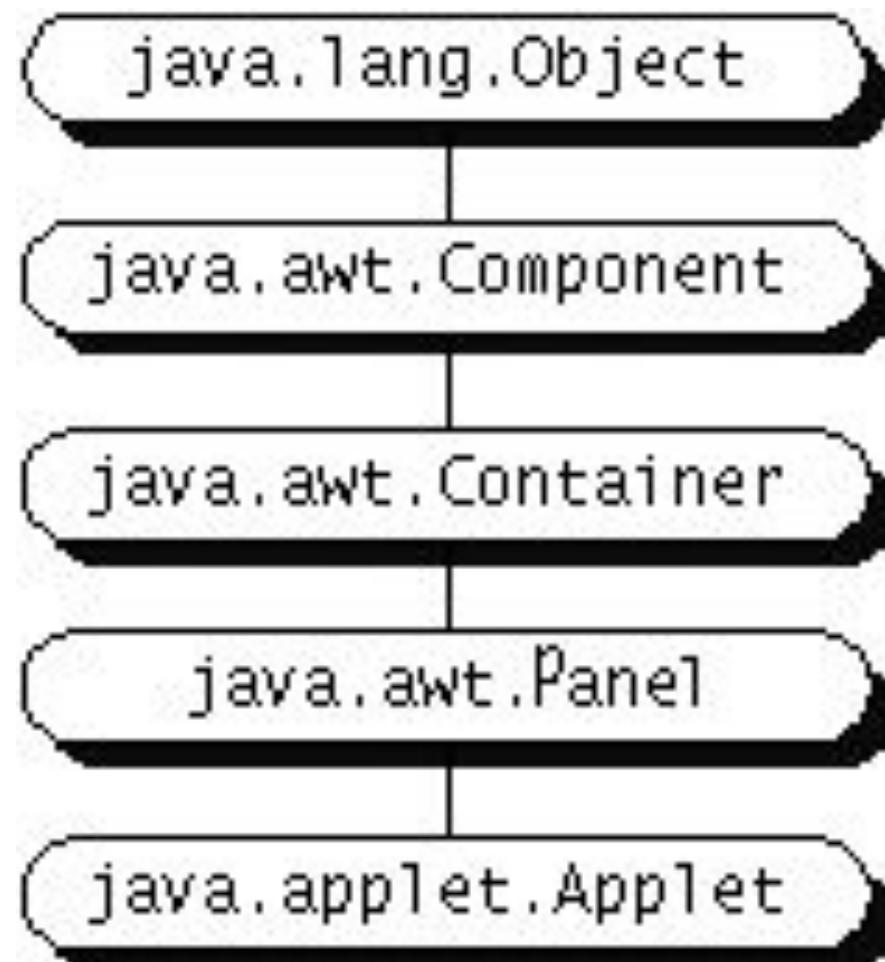
```
import java.applet.*;
import java.awt.*;
public class HelloWorldApplet extends Applet
{
 public void paint (Graphics g)
 {
 g.drawString ("Hello World", 25, 50);
 }
}
```

# Applet Tag

- Syntax <APPLET>...</APPLET> Attribute Specifications **CODE=***CDATA* (class file)
- CODEBASE=*URI* (base URI for class files)
- **WIDTH=***Length* (applet width)
- **HEIGHT=***Length* (applet height)
- ARCHIVE=*CDATA* (archive files)
- OBJECT=*CDATA* (serialized applet)
- NAME=*CDATA* (name for inter-applet communication)
- ALT=*Text* (alternate text)
- ALIGN=[ top | middle | bottom | left | right ] (applet alignment)
- HSPACE=*Pixels* (horizontal gutter)
- VSPACE=*Pixels* (vertical gutter)

# Applet Inheritance Hierarchy

- Applet class derives from the Abstract Window Toolkit (AWT) hierarchy.



# Useful Applet Methods

- **getCodeBase, getDocumentBase**
  - The URL of the:  
**Applet file - getCodeBase**  
**HTML file - getDocumentBase**
- **getParameter**
  - Retrieves the value from the associated HTML PARAM element
- **getSize**
  - Returns the Dimension (width, height) of the applet
- **getGraphics**
  - Retrieves the current Graphics object for the applet
  - The Graphics object does not persist across paint invocations

# Useful Applet Methods, (Continued)

- **showDocument** (AppletContext method)
  - getAppletContext().showDocument(...)
  - Asks the browser to retrieve and display a Web page
  - Can direct page to a named FRAME cell
- **showStatus(String str)**
  - Displays a string in the status line at the bottom of the browser
- **getCursor, setCursor**
  - Defines the Cursor for the mouse, for example,  
CROSSHAIR\_CURSOR, HAND\_CURSOR, WAIT\_CURSOR

# Useful Applet Methods, (Continued)

- `getAudioClip`
- `play`
  - Retrieves an audio file from a remote location and plays it
  - JDK 1.1 supports .au only. Java 2 also supports MIDI, .aiff and .wav
- `getBackground, setBackground`
  - Gets/sets the background color of the applet
  - SystemColor class provides access to desktop colors
- `getForeground, setForeground`
  - Gets/sets foreground color of applet (default color of drawing operations)

# Useful Graphics Methods

- `drawString(string, left, bottom)`
  - Draws a string in the current font and color with the *bottom left* corner of the string at the specified location
  - One of the few methods where the y coordinate refers to the bottom of shape, not the top. But y values are still with respect to the *top left* corner of the applet window
- `drawRect(left, top, width, height)`
  - Draws the outline of a rectangle (1-pixel border) in the current color
- `fillRect(left, top, width, height)`
  - Draws a solid rectangle in the current color
- `drawLine(x1, y1, x2, y2)`
  - Draws a 1-pixel-thick line from (x1, y1) to (x2, y2)

# Useful Graphics Methods, continued

- `drawOval, fillOval`
  - Draws an outlined and solid oval, where the arguments describe a rectangle that bounds the oval
- `drawPolygon, fillPolygon`
  - Draws an outlined and solid polygon whose points are defined by arrays or a `Polygon` (a class that stores a series of points)
  - By default, polygon is closed; to make an open polygon use the `drawPolyline` method
- `drawImage`
  - Draws an image
  - Images can be in JPEG or GIF (including GIF89A) format

# Graphics Color

## □ setColor, getColor

- Specifies the foreground color prior to drawing operation
- By default, the graphics object receives the foreground color of the window
- AWT has 16 predefined colors (Color.red, Color.blue, etc.) or create your own color, new Color(r, g, b)
- Changing the color of the Graphics object affects only the drawing that explicitly uses that Graphics object
  - To make permanent changes, call the *applet's* setForeground method.

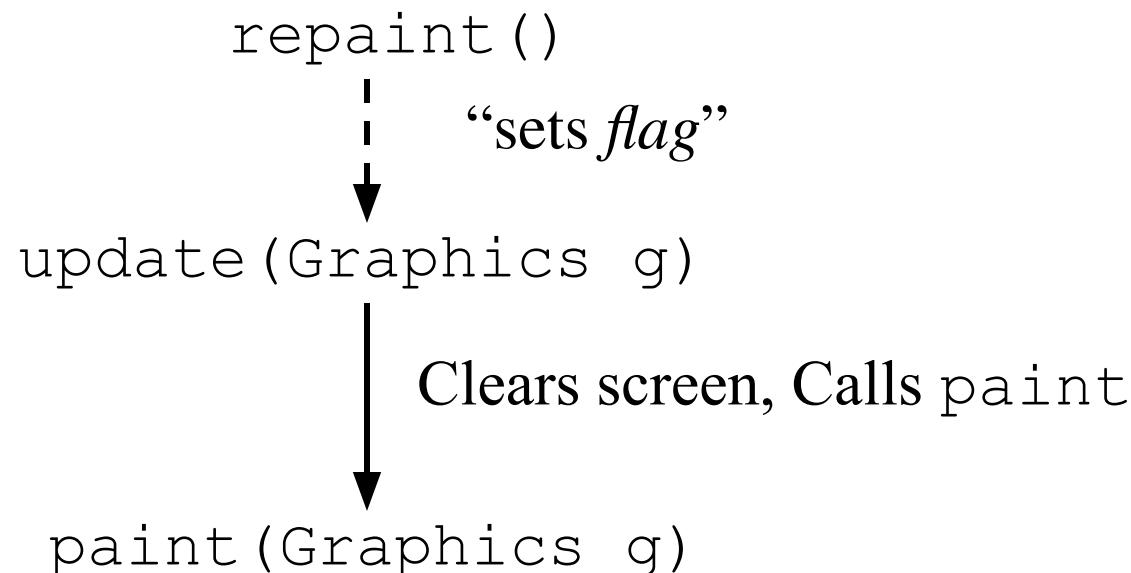
# Graphics Font

## □ **setFont, getFont**

- Specifies the font to be used for drawing text
- Determine the size of a character through FontMetrics (in Java 2 use LineMetrics)
- Setting the font for the Graphics object does not persist to subsequent invocations of paint
- Set the font of the window (I.e., call the *applet's* setFont method) for permanent changes to the Graphics object
- In JDK 1.1, **only 5 fonts** are available: Serif (aka TimesRoman), SansSerif (aka Helvetica), Monospaced (aka Courier), Dialog, and DialogInput

# Graphics Behavior

- Browser calls repaint method to request redrawing of applet
  - Called when applet first drawn or applet is hidden by another window and then re-exposed



# AWT (Abstract Windowing Toolkit)

---

- The AWT is roughly broken into three categories
  - Components
  - Layout Managers
  - Graphics
- Many AWT components have been replaced by Swing components
- It is generally not considered a good idea to mix Swing components and AWT components. Choose to use one or the other.

# AWT Class Hierarchy



# Component

---

- Component is the superclass of most of the displayable classes defined within the AWT. Note: it is abstract.
- MenuComponent is another class which is similar to Component except it is the superclass for all GUI items which can be displayed within a drop-down menu.
- The Component class defines data and methods which are relevant to all Components

setBounds

setSize

setLocation

setFont

setEnabled

setVisible

setForeground -- colour

setBackground -- colour

# Container

---

- Container is a subclass of Component. (ie. All containers are themselves, Components)
- Containers contain components
- For a component to be placed on the screen, it must be placed within a Container
- The Container class defined all the data and methods necessary for managing groups of Components

add

getComponent

getMaximumSize

getMinimumSize

getPreferredSize

remove

removeAll

# Windows and Frames

---

- The Window class defines a top-level Window with no Borders or Menu bar.
  - Usually used for application splash screens
- Frame defines a top-level Window with Borders and a Menu Bar
  - Frames are more commonly used than Windows
- Once defined, a Frame is a Container which can contain Components

```
Frame aFrame = new Frame("Hello World");
aFrame.setSize(100,100);
aFrame.setLocation(10,10);
aFrame.setVisible(true);
```

# Panels

---

- When writing a GUI application, the GUI portion can become quite complex.
- To manage the complexity, GUIs are broken down into groups of components. Each group generally provides a unit of functionality.
- A **Panel** is a rectangular Container whose sole purpose is to hold and manage components within a GUI.

```
Panel aPanel = new Panel();
aPanel.add(new Button("Ok"));
aPanel.add(new Button("Cancel"));
```

```
Frame aFrame = new Frame("Button Test");
aFrame.setSize(100,100);
aFrame.setLocation(10,10);
aFrame.add(aPanel);
```

# Buttons

---

- This class represents a push-button which displays some specified text.
- When a button is pressed, it notifies its Listeners. (More about Listeners in the next chapter).
- To be a Listener for a button, an object must implement the ActionListener Interface.

```
Panel aPanel = new Panel();
Button okButton = new Button("Ok");
Button cancelButton = new Button("Cancel");

aPanel.add(okButton));
aPanel.add(cancelButton));

okButton.addActionListener(controller2);
cancelButton.addActionListener(controller1)
;
```

# Labels

---

- This class is a Component which displays a single line of text.
- Labels are read-only. That is, the user cannot click on a label to edit the text it displays.
- Text can be aligned within the label

```
Label aLabel = new Label("Enter
password");
aLabel.setAlignment(Label.RIGHT);

aPanel.add(aLabel);
```

# List

---

- This class is a Component which displays a list of Strings.
- The list is scrollable, if necessary.
- Sometimes called Listbox in other languages.
- Lists can be set up to allow single or multiple selections.
- The list will return an array indicating which Strings are selected

```
List aList = new List();
aList.add("Calgary");
aList.add("Edmonton");
aList.add("Regina");
aList.add("Vancouver");

aList.setMultipleMode(true);
```

# Checkbox

---

- This class represents a GUI checkbox with a textual label.
- The Checkbox maintains a boolean state indicating whether it is checked or not.
- If a Checkbox is added to a CheckBoxGroup, it will behave like a radio button.

```
Checkbox creamCheckbox = new
CheckBox("Cream");
```

```
Checkbox sugarCheckbox = new
CheckBox("Sugar");
```

```
if (creamCheckbox.getState())
{
 coffee.addCream();
}
```

# Choice

---

- This class represents a dropdown list of Strings.
- Similar to a list in terms of functionality, but displayed differently.
- Only one item from the list can be selected at one time and the currently selected element is displayed.

```
Choice aChoice = new Choice();
aChoice.add("Calgary");
aChoice.add("Edmonton");
aChoice.add("Alert Bay");
String selectedDestination=
aChoice.getSelectedItem();
```

# TextField

---

- This class displays a single line of optionally editable text.
- This class inherits several methods from TextComponent.
- This is one of the most commonly used Components in the AWT

```
TextField emailTextField = new
TextField();

TextField passwordTextField = new
TextField();
passwordTextField.setEchoChar('*');
[...]
```

```
String userEmail =
emailTextField.getText();

String userpassword =
passwordTextField.getText();
```

# TextArea

---

- This class displays multiple lines of optionally editable text.
- This class inherits several methods from TextComponent.
- TextArea also provides the methods: appendText(),  
insertText() and replaceText()

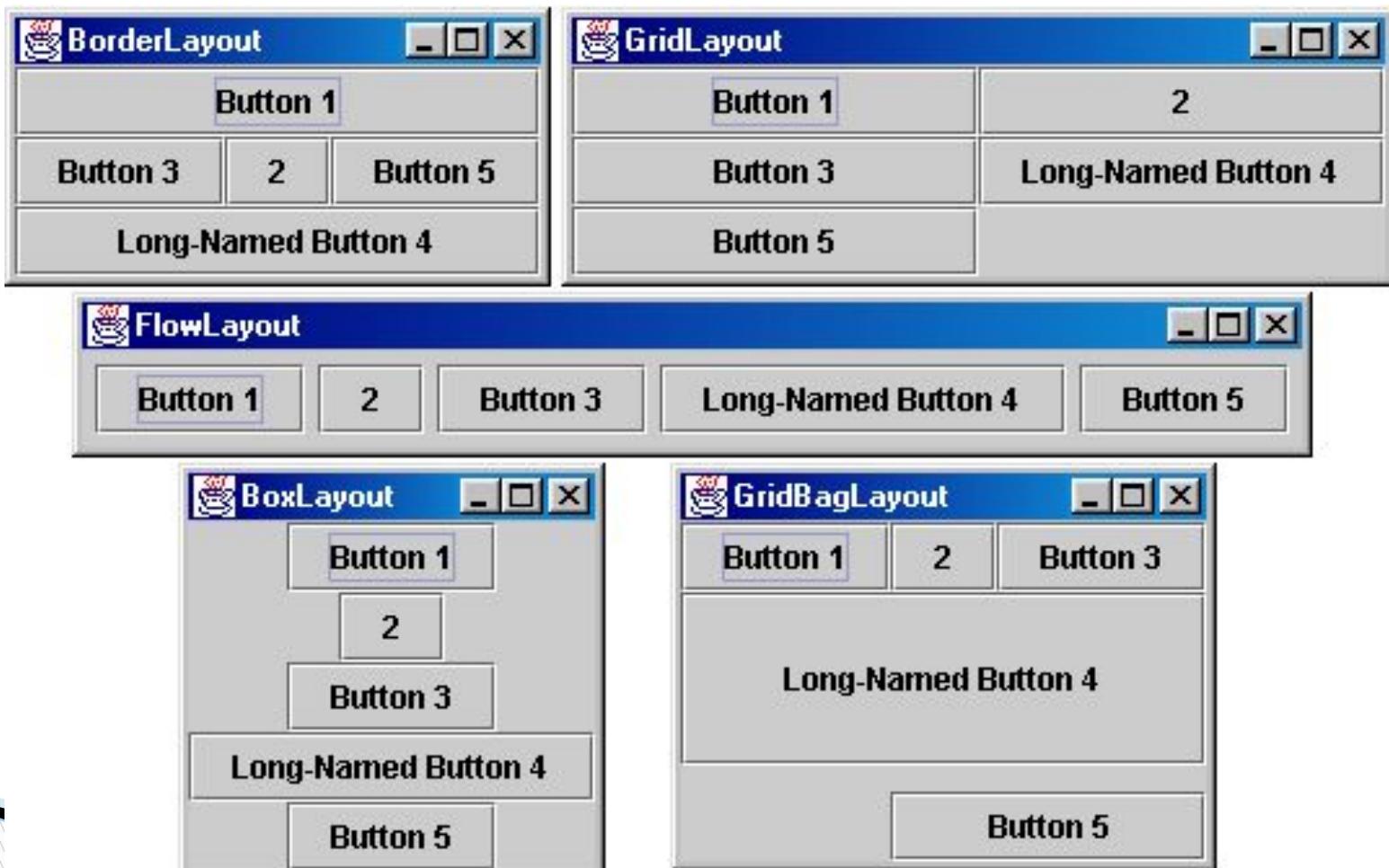
```
// 5 rows, 80 columns
TextArea fullAddressTextArea = new
TextArea(5, 80);
```

```
String userFullAddress=
fullAddressTextArea.getText();
```

# Layout Managers: Outline

- How layout managers simplify interface design?
- Standard layout managers
  - FlowLayout, BorderLayout, CardLayout, GridLayout, GridBagLayout, BoxLayout
- Positioning components manually
- Strategies for using layout managers effectively
- Using invisible components

# Layout Managers



# Layout Managers

- Assigned to each Container
  - Give *sizes* and *positions* to components in the window
  - Helpful for windows whose size changes or that display on multiple operating systems
- Relatively easy for simple layouts
  - But, it is surprisingly hard to get complex layouts with a single layout manager
- Controlling complex layouts
  - Use nested containers (each with its own layout manager)
  - Use invisible components and layout manager options
  - Write your own layout manager
  - Turn layout managers off and arrange things manually

# Array and Collection Classes

# Overview

- **Arrays**
  - Working with arrays
  - Java API support for arrays
- **Collection classes**
  - Types of collection
  - Working with Collections

# Java Arrays – The Basics

- Declaring an array

```
int[] myArray;
int[] myArray = new int[5];
String[] stringArray = new String[10];
String[] strings = new String[] {"one", "two"};
```

- Checking an arrays length

```
int arrayLength = myArray.length;
```

- Looping over an array

```
for(int i=0; i<myArray.length; i++)
{
 String s = myArray[i];
}
```

# Java Arrays – Bounds Checking

- **Bounds checking**
  - Java does this automatically. Impossible to go beyond the end of an array (unlike C/C++)
  - Automatically generates an `ArrayIndexOutOfBoundsException`

# Java Arrays – Copying

- Don't copy arrays “by hand” by looping over the array
- The System class has an `arrayCopy` method to do this efficiently

```
int array1[] = new int[10];
int array2[] = new int[10];
//assume we add items to array1

//copy array1 into array2
System.arraycopy(array1, 0, array2, 0, 10);
//copy last 5 elements in array1 into first 5 of array2
System.arraycopy(array1, 5, array2, 0, 5);
```

# Java Arrays – Sorting

- Again no need to do this “by hand”.
- The `java.util.Arrays` class has methods to sort different kinds of arrays

```
int myArray[] = new int[] {5, 4, 3, 2, 1};
java.util.Arrays.sort(myArray);
//myArray now holds 1, 2, 3, 4, 5
```

- Sorting arrays of *objects* is involves some extra work, as we'll see later...

# Multidimensional Array Declaration

```
int[][] x; //Valid
int [] []x; //Valid
int x[] []; //Valid
int[] []x; //Valid
int[] x[] ; //Valid
int []x[] ; //Valid
int[] x[] [] ; //Valid
int[][] []x; //Valid
int[] [] x[] ; //Valid
int[] []x[] ; //Valid
int[] []x[] ; //Valid
...
...
```

```
int[] a,b; //a->1 b->1

int[] a[],b; //a->2 b->1

int[] a[],b[]; //a->b->2

int[] []a,b; //a->b->2

int[] []a,b[]; //a->2 b->3

int[] []a,[]b; // Error

int[] []a,[]b,[]c; // Error
```

```
int[] x = new int[];
```

```
int[] x = new int[0];
```

```
int[] x = new int[-1];
```

```
int[] x = new int[5];
```

```
int[] x = new int['a']; //byte,short,char,int
```

# Java Arrays

- Advantages
  - Very efficient, quick to access and add to
  - Type-safe, can only add items that match the declared type of the array
- Disadvantages
  - Fixed size, some overhead in copying/resizing
  - Can't tell how many items in the array, just how large it was declared to be
  - Limited functionality, need more general functionality

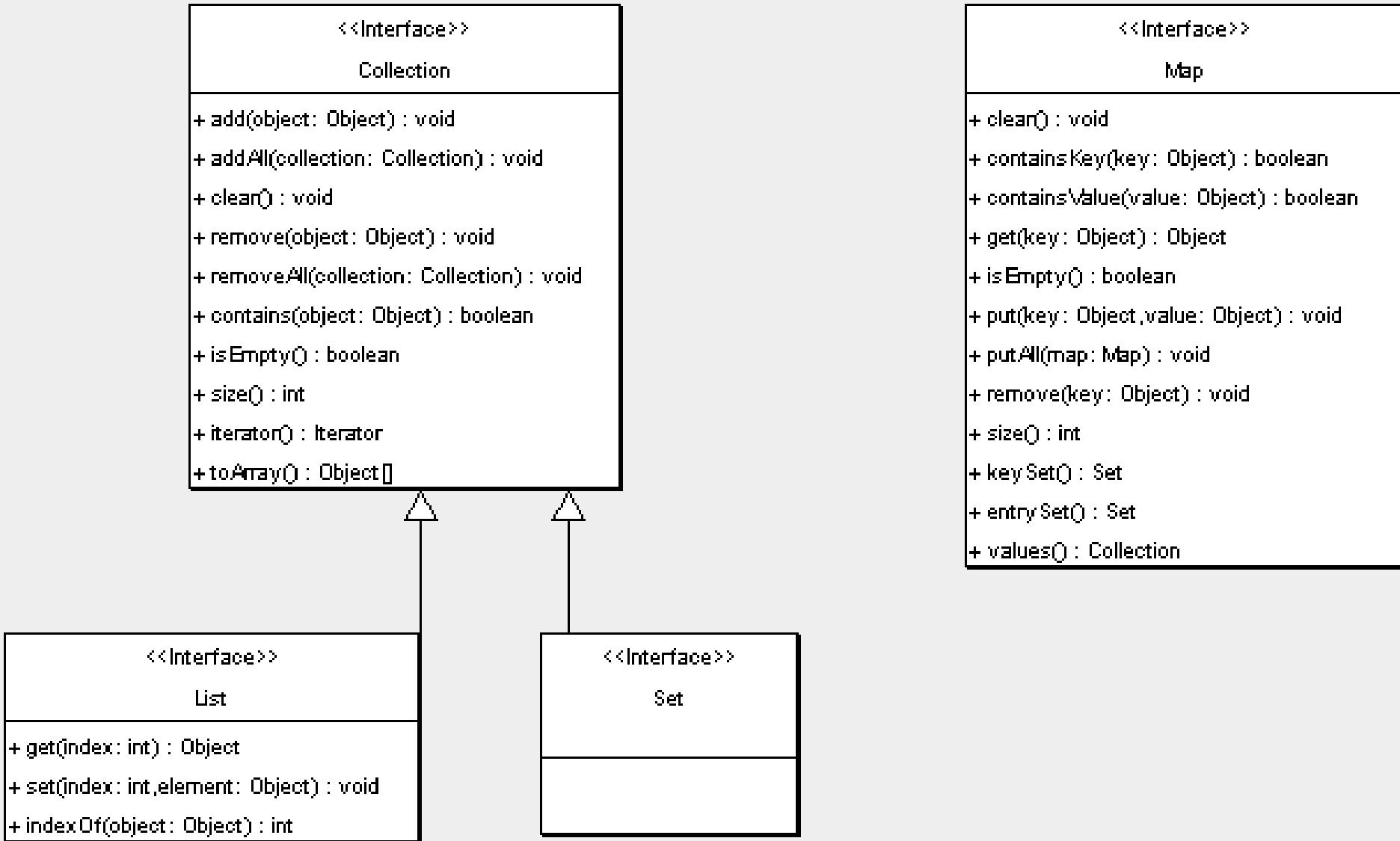
# Java Collections

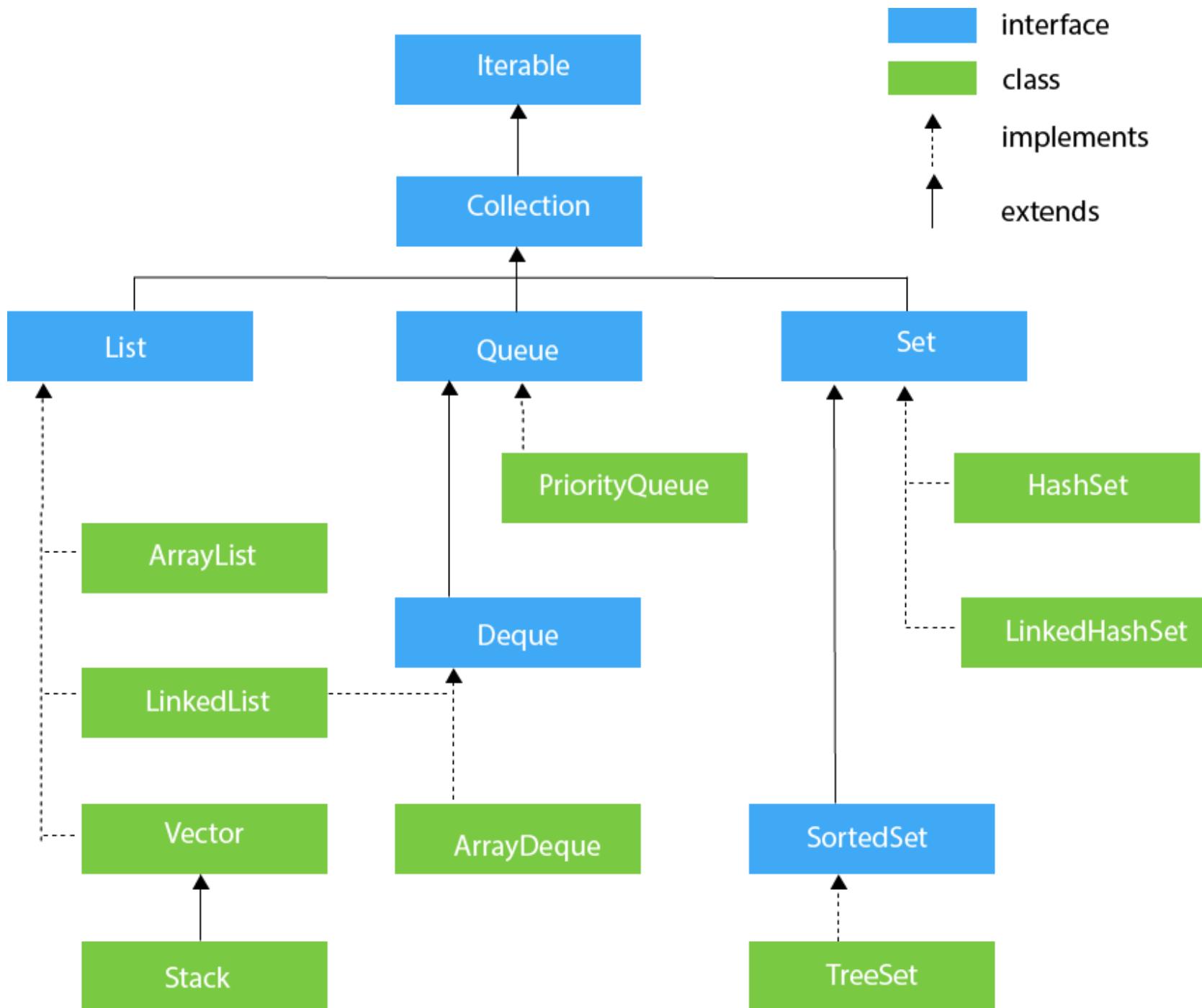
- What are they?
  - A number of pre-packaged implementations of common ‘container’ classes, such as LinkedLists, Sets, etc.
  - Part of the `java.util` package.
- Advantages
  - Very flexible, can hold any kind of object
- Disadvantages
  - Not as efficient as arrays (for some uses)
  - Not type-safe. Store references to Object

# Java Collections

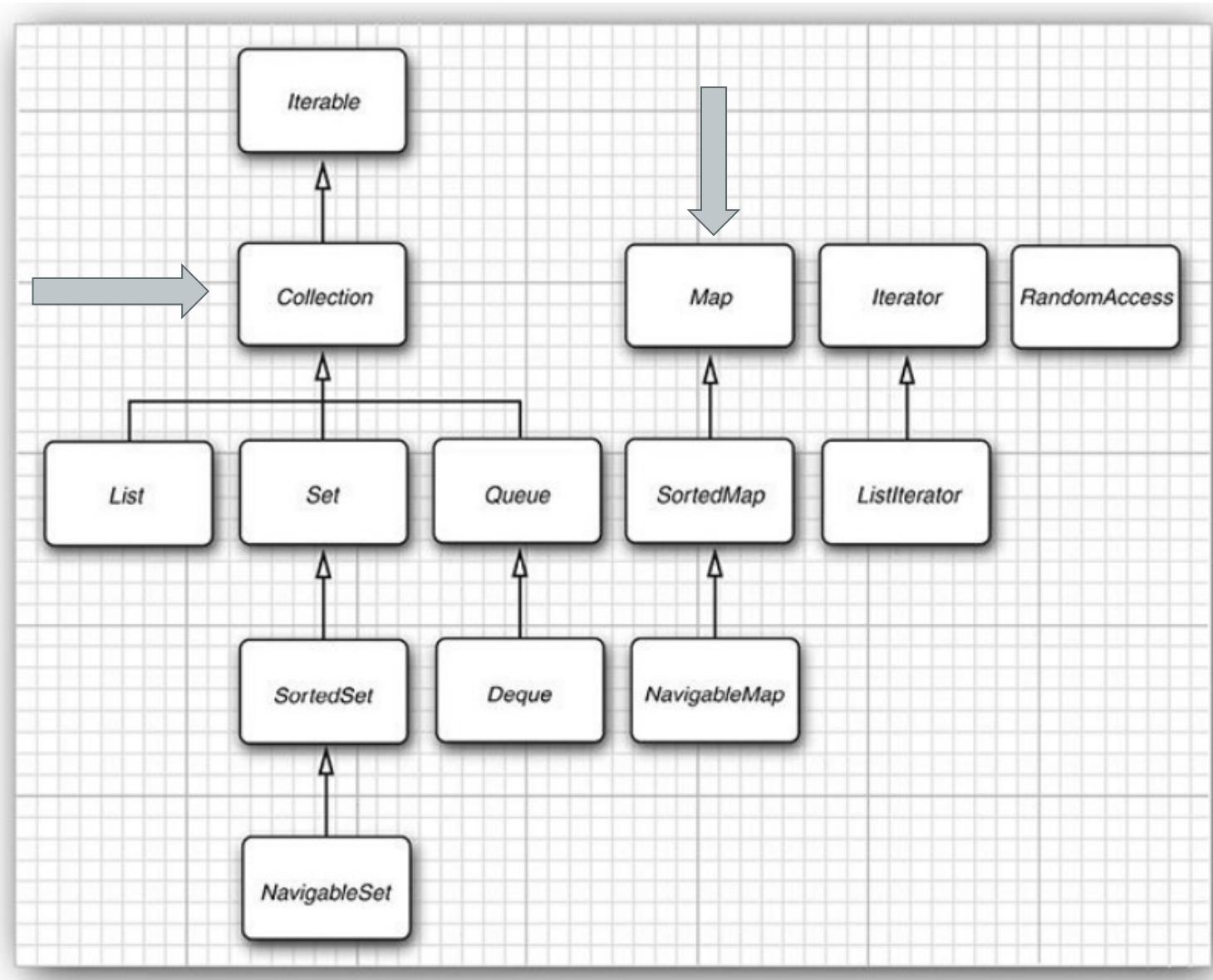
- Two Types of Containers
- Collections
  - Group of objects, which may be restricted or manipulated in some way
  - E.g. an ordered group to make a List or LinkedList
  - E.g. a Set, an unordered group which can only contain one of each item
- Maps
  - Associative array, Dictionary, Lookup Table, Hash
  - A group of name-value pairs

# Java Collections

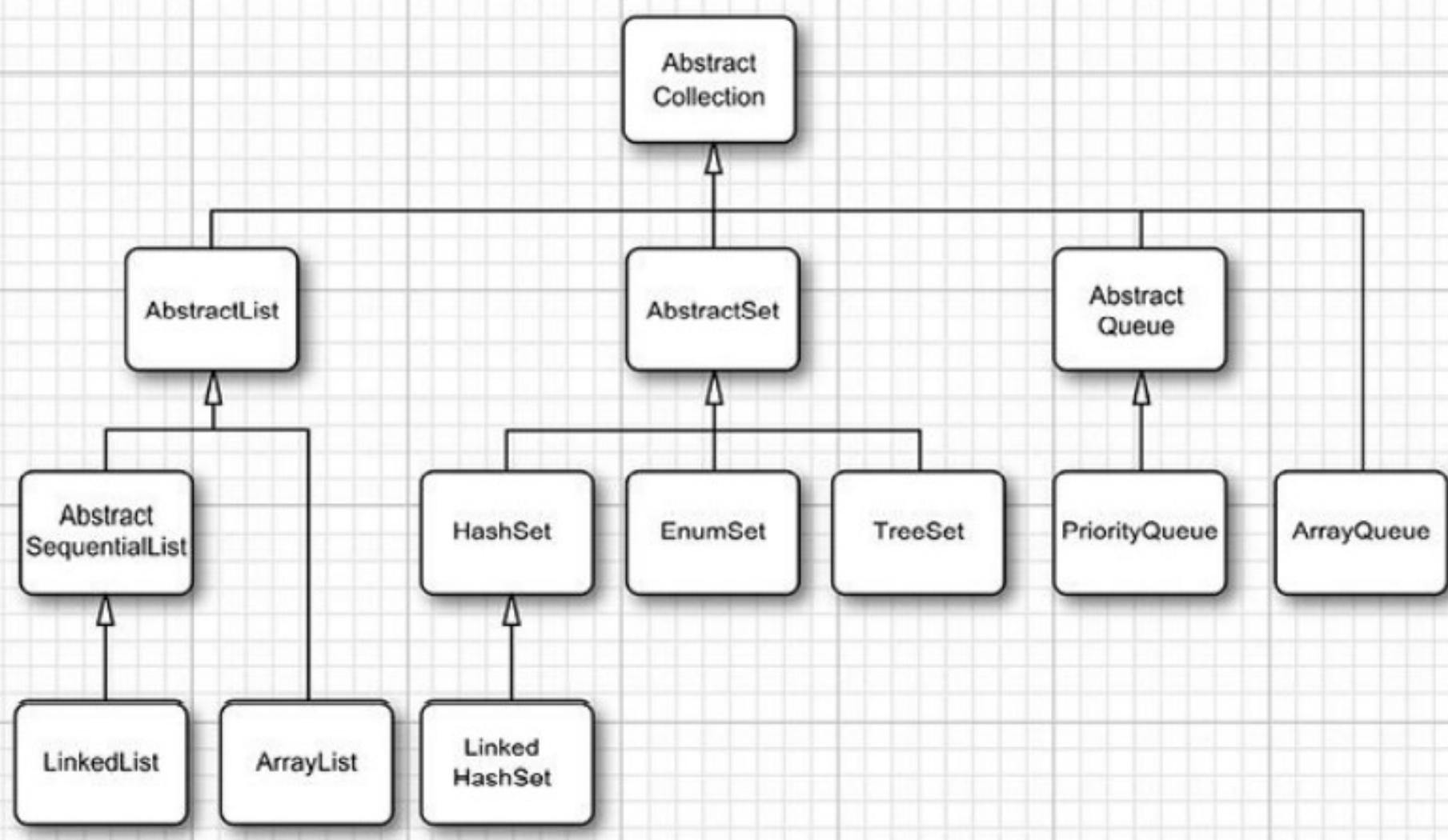




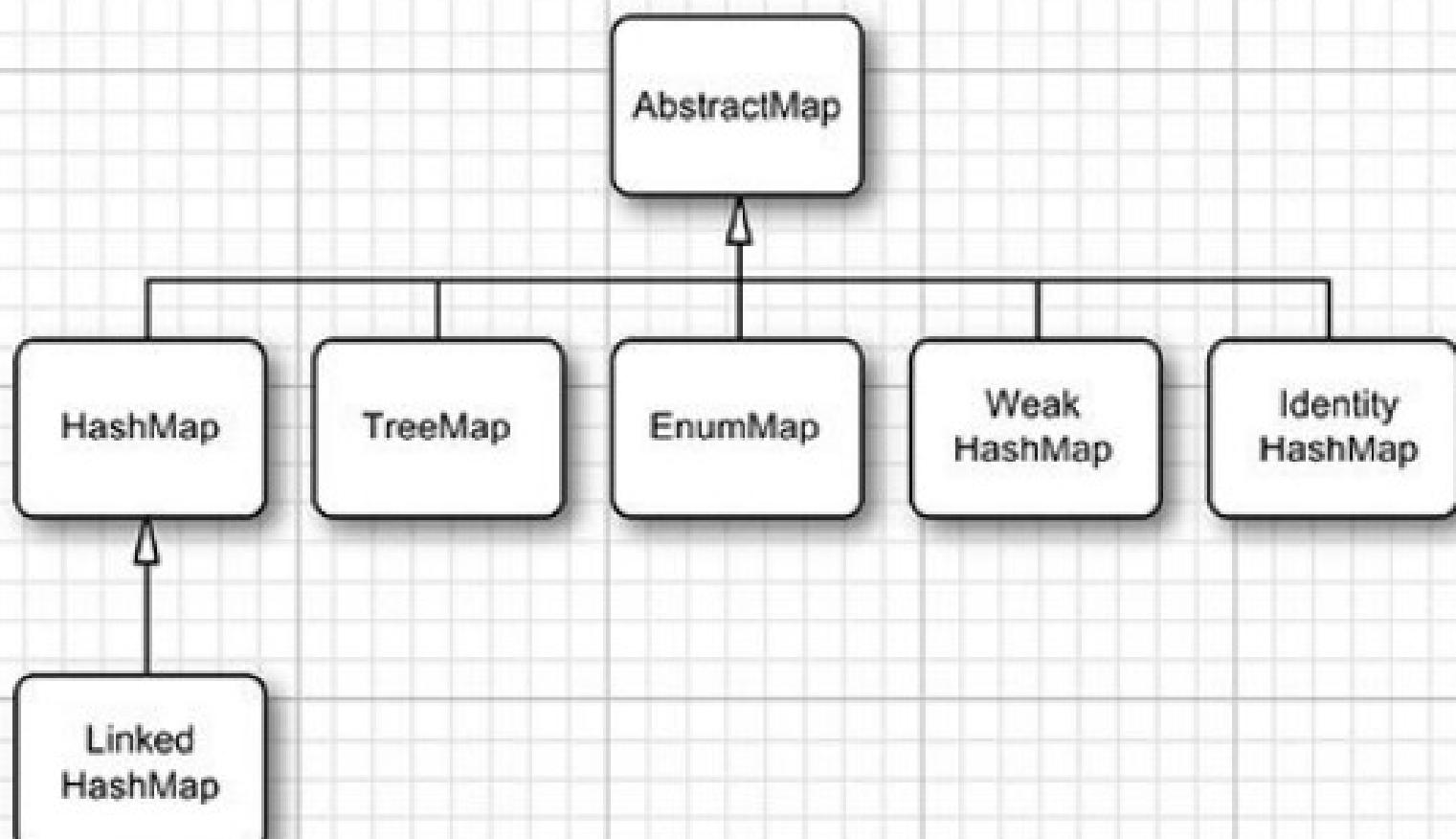
# Interfaces in Collection Framework



# Concrete Collections



# Concrete Collections



# The Collection Interface

```
public interface Collection<E>{
 boolean add(E element);
 Iterator<E> iterator();
 ...
}
```

```
public interface Iterable<E>{
 Iterator<E> iterator();
 ...
}
```

```
public interface Iterator<E>{
 E next();
 boolean hasNext();
 void remove();
 default void forEachRemaining(Consumer<? super E> action);
}
```

# Generic Utility Methods - Collection interface

```
int size()
boolean isEmpty()
boolean contains(Object obj)
boolean containsAll(Collection<?> c)
boolean equals(Object other)
boolean addAll(Collection<? extends E> from)
boolean remove(Object obj)
boolean removeAll(Collection<?> c)
void clear()
boolean retainAll(Collection<?> c)
Object[] toArray()
<T> T[] toArray(T[] arrayToFill)
```

```
public abstract class
AbstractCollection implements
Collection {
 ...
 public abstract Iterator iterator();

 public boolean contains(Object
 obj)
 {
 for (E element : this) // calls
 iterator()
 if (element.equals(obj))
 return true;
 return false;
 }
 ...
}
```

# Java Collections

- Several implementations associated with each of the basic interfaces
- Each has its own advantages/disadvantages
- Maps
  - HashMap, SortedMap
- Lists
  - ArrayList, LinkedList
- Sets
  - HashSet, SortedSet

# Java Collections – The Basics

- HashMap and ArrayList are most commonly encountered
- Usual object creation syntax
- Generally hold references to the interface and not the specific collection
  - Can then process them generically

```
List myList = new ArrayList();
List otherList = new ArrayList(5);
Map database = new HashMap();
Set things = new HashSet();
```

# Java Collections – Adding Items

- For Collections, use add ()

```
List myList = new ArrayList();
myList.add("A String");
myList.add("Other String");
```

- For Maps, use put ()

```
Map myMap = new HashMap();
myMap.put("google", "http://www.google.com");
myMap.put("yahoo", "http://www.yahoo.com");
```

# Java Collections – Copying

- Very easy, just use addAll()

```
List myList = new ArrayList();
//assume we add items to the list
```

```
List otherList = new ArrayList();
myList.addAll(myList);
```

# Collections – Getting Individual Items

- Use `get()`
- Note that we have to *cast* the object to its original type.
- Collections...

```
String s = (String)myList.get(1); //get first element
String s2 = (String)myList.get(10); //get tenth element
```

- Maps...

```
String s = (String)myMap.get("google");
String s2 = (String)mpMap.get("yahoo");
```

# Collections – Getting all items

- For Lists, we could use a `for` loop, and loop through the list to get () each item
- But this doesn't work for Maps.
- To allow generic handling of collections, Java defines an object called an *Iterator*
  - An object whose function is to walk through a Collection of objects and provide access to each object in sequence

# Collections – Getting all items

- Get an iterator using the `iterator()` method
- Iterator objects have three methods:
  - `next()` – gets the next item in the collection
  - `hasNext()` – tests whether it has reached the end
  - `remove()` – removes the item just returned
- Basic iterators only go forwards
  - Lists objects have a `ListIterator` that can go forward and backward

# Collections – Getting all items

- Simple example:

```
List myList = new ArrayList();
//we add items
```

```
Iterator iterator = myList.iterator();
while (iterator.hasNext())
{
 String s = (String) iterator.next();
 //do something with it
}
```

# Collections – Other Functions

- The `java.util.Collections` class has many useful methods for working with collections
  - `min`, `max`, `sort`, `reverse`, `search`, `shuffle`
- Virtually all require your objects to implement an extra interface, called `Comparable`

# The Vector Class

- Vector implements a dynamic array. It is similar to ArrayList, but with two differences –
- Vector is synchronized.
- Vector contains many legacy methods that are not part of the collections framework.
- Vector proves to be very useful if you don't know the size of the array in advance or you just need one that can change sizes over the lifetime of a program.
- Following is the list of constructors provided by the vector class.

# The Vector Class

Following is the list of constructors provided by the vector class.

## **Vector()**

This constructor creates a default vector, which has an initial size of 10.

## **Vector(int size)**

This constructor accepts an argument that equals to the required size, and creates a vector whose initial capacity is specified by size.

## **Vector(int size, int incr)**

This constructor creates a vector whose initial capacity is specified by size and whose increment is specified by incr. The increment specifies the number of elements to allocate each time that a vector is resized upward.

## **Vector(Collection c)**

This constructor creates a vector that contains the elements of collection c.

# The Vector Class

## **void add(int index, Object element)**

Inserts the specified element at the specified position in this Vector.

## **boolean add(Object o) -- Collection(I)**

Appends the specified element to the end of this Vector.

## **boolean addAll(Collection c) -- Collection(I)**

Appends all of the elements in the specified Collection to the end of this Vector, in the order that they are returned by the specified Collection's Iterator.

## **boolean addAll(int index, Collection c)**

Inserts all of the elements in the specified Collection into this Vector at the specified position.

## **void addElement(Object obj)**

Adds the specified component to the end of this vector, increasing its size by one.

## **int capacity()**

Returns the current capacity of this vector.

# The Vector Class

## **boolean contains(Object elem)**

Tests if the specified object is a component in this vector.

## **boolean containsAll(Collection c)**

Returns true if this vector contains all of the elements in the specified Collection.

## **void copyInto(Object[] anArray)**

Copies the components of this vector into the specified array.

## **Object elementAt(int index)**

Returns the component at the specified index.

## **boolean equals(Object o)**

Compares the specified Object with this vector for equality.

## **Object firstElement()**

Returns the first component (the item at index 0) of this vector.

## **Object get(int index)**

Returns the element at the specified position in this vector.

# The Vector Class

## **int indexOf(Object elem)**

Searches for the first occurrence of the given argument, testing for equality using the equals method.

## **int indexOf(Object elem, int index)**

Searches for the first occurrence of the given argument, beginning the search at index, and testing for equality using the equals method.

## **void insertElementAt(Object obj, int index)**

Inserts the specified object as a component in this vector at the specified index.

## **boolean isEmpty() -- Collection(I)**

Tests if this vector has no components.

## **boolean remove(Object o) -- Collection(I)**

Removes the first occurrence of the specified element in this vector, If the vector does not contain the element, it is unchanged.

## **boolean removeAll(Collection c) -- Collection(I)**

Removes from this vector all of its elements that are contained in the specified Collection.

# The Vector Class

## **void removeAllElements()**

Removes all components from this vector and sets its size to zero.

## **boolean removeElement(Object obj)**

Removes the first (lowest-indexed) occurrence of the argument from this vector.

## **void removeElementAt(int index)**

## **int size()**

Returns the number of components in this vector.

## **Object[] toArray()**

Returns an array containing all of the elements in this vector in the correct order.

## **String toString()**

Returns a string representation of this vector, containing the String representation of each element.

## **void trimToSize()**

Trims the capacity of this vector to be the vector's current size.

# Collections – Comparator Example

- Java String comparison is lexicographic not alphabetic, I.e. based on the character set, not alphabetic order

```
public class AlphaComparison implements Comparator
{
 public int compare(Object obj1, Object obj2)
 {
 String s1 = ((String) o1).toLowerCase();
 String s2 = ((String) o2).toLowerCase();
 return s1.compareTo(s2);
 }
}
```

## Collection Method & Description

### **boolean add(Object obj)**

Adds obj to the invoking collection. Returns true if obj was added to the collection. Returns false if obj is already a member of the collection, or if the collection does not allow duplicates.

### **boolean addAll(Collection c)**

Adds all the elements of c to the invoking collection. Returns true if the operation succeeds (i.e., the elements were added). Otherwise, returns false.

### **void clear()**

Removes all elements from the invoking collection.

### **boolean contains(Object obj)**

Returns true if obj is an element of the invoking collection. Otherwise, returns false.

### **boolean containsAll(Collection c)**

Returns true if the invoking collection contains all elements of c. Otherwise, returns false.

## Collection Method & Description

### **boolean equals(Object obj)**

Returns true if the invoking collection and obj are equal. Otherwise, returns false.

### **int hashCode()**

Returns the hash code for the invoking collection.

### **boolean isEmpty()**

Returns true if the invoking collection is empty. Otherwise, returns false.

### **Iterator iterator()**

Returns an iterator for the invoking collection.

### **boolean remove(Object obj)**

Removes one instance of obj from the invoking collection. Returns true if the element was removed. Otherwise, returns false.

### **boolean removeAll(Collection c)**

Removes all elements of c from the invoking collection. Returns true if the collection changed (i.e., elements were removed). Otherwise, returns false.

### **boolean retainAll(Collection c)**

Removes all elements from the invoking collection except those in c. Returns true if the collection changed (i.e., elements were removed). Otherwise, returns false.

## Collection Method & Description

### **int size( )**

Returns the number of elements held in the invoking collection.

### **Object[ ] toArray( )**

Returns an array that contains all the elements stored in the invoking collection. The array elements are copies of the collection elements.

### **Object[ ] toArray(Object array[ ])**

Returns an array containing only those collection elements whose type matches that of array.

## List Interface Method & Description

### **void add(int index, Object obj)**

Inserts obj into the invoking list at the index passed in the index. Any pre-existing elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten.

### **boolean addAll(int index, Collection c)**

Inserts all elements of c into the invoking list at the index passed in the index. Any pre-existing elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten. Returns true if the invoking list changes and returns false otherwise.

### **Object get(int index)**

Returns the object stored at the specified index within the invoking collection.

### **int indexOf(Object obj)**

Returns the index of the first instance of obj in the invoking list. If obj is not an element of the list, -1 is returned.

### **int lastIndexOf(Object obj)**

Returns the index of the last instance of obj in the invoking list. If obj is not an element of the list, -1 is returned.

## List Interface Method & Description

### **ListIterator listIterator()**

Returns an iterator to the start of the invoking list.

### **ListIterator listIterator(int index)**

Returns an iterator to the invoking list that begins at the specified index.

### **Object remove(int index)**

Removes the element at position index from the invoking list and returns the deleted element. The resulting list is compacted. That is, the indexes of subsequent elements are decremented by one.

### **Object set(int index, Object obj)**

Assigns obj to the location specified by index within the invoking list.

### **List subList(int start, int end)**

Returns a list that includes elements from start to end.<sup>1</sup> in the invoking list. Elements in the returned list are also referenced by the invoking object.

# The ArrayList Class

- The ArrayList class extends AbstractList and implements the List interface. ArrayList supports dynamic arrays that can grow as needed.
- Standard Java arrays are of a fixed length. After arrays are created, they cannot grow or shrink, which means that you must know in advance how many elements an array will hold.
- Array lists are created with an initial size. When this size is exceeded, the collection is automatically enlarged. When objects are removed, the array may be shrunk.
- Following is the list of the constructors provided by the ArrayList class.

# The ArrayList Class

## **ArrayList()**

This constructor builds an empty array list.

## **ArrayList(Collection c)**

This constructor builds an array list that is initialized with the elements of the collection **c**.

## **ArrayList(int capacity)**

This constructor builds an array list that has the specified initial capacity. The capacity is the size of the underlying array that is used to store the elements. The capacity grows automatically as elements are added to an array list.

# The ArrayList Class

## **void add(int index, Object element)**

Inserts the specified element at the specified position index in this list. Throws IndexOutOfBoundsException if the specified index is out of range ( $\text{index} < 0 \text{ || } \text{index} > \text{size}()$ ).

## **boolean add(Object o)**

Appends the specified element to the end of this list.

## **boolean addAll(Collection c)**

Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator. Throws NullPointerException, if the specified collection is null.

# The ArrayList Class

## **Object clone()**

Returns a shallow copy of this ArrayList.

## **boolean contains(Object o)**

Returns true if this list contains the specified element. More formally, returns true if and only if this list contains at least one element *e* such that (*o*==*null* ? *e*==*null* : *o.equals(e)*).

## **void ensureCapacity(int minCapacity)**

Increases the capacity of this ArrayList instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument.

## **Object get(int index)**

Returns the element at the specified position in this list. Throws IndexOutOfBoundsException if the specified index is out of range (*index* < 0 || *index* >= *size()*).

# The ArrayList Class

## **int indexOf(Object o)**

Returns the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element.

## **int lastIndexOf(Object o)**

Returns the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element.

## **Object remove(int index)**

Removes the element at the specified position in this list. Throws IndexOutOfBoundsException if the index out is of range ( $\text{index} < 0$  ||  $\text{index} \geq \text{size}()$ ).

## **protected void removeRange(int fromIndex, int toIndex)**

Removes from this List all of the elements whose index is between fromIndex, inclusive and toIndex, exclusive.

# The ArrayList Class

## **Object set(int index, Object element)**

Replaces the element at the specified position in this list with the specified element. Throws IndexOutOfBoundsException if the specified index is out of range ( $\text{index} < 0 \text{ || } \text{index} \geq \text{size}()$ ).

## **int size()**

Returns the number of elements in this list.

## **Object[] toArray()**

Returns an array containing all of the elements in this list in the correct order.  
Throws NullPointerException if the specified array is null.

## **Object[] toArray(Object[] a)**

Returns an array containing all of the elements in this list in the correct order;  
the runtime type of the returned array is that of the specified array.

## **void trimToSize()**

Trims the capacity of this ArrayList instance to be the list's current size.

# The LinkedList Class

- The `LinkedList` class extends `AbstractSequentialList` and implements the `List` interface. It provides a linked-list data structure.
- Following are the constructors supported by the `LinkedList`

**`LinkedList()`**

This constructor builds an empty linked list.

**`LinkedList(Collection c)`**

This constructor builds a linked list that is initialized with the elements of the collection `c`.

# The LinkedList Class

## **void add(int index, Object element)**

Inserts the specified element at the specified position index in this list. Throws IndexOutOfBoundsException if the specified index is out of range ( $\text{index} < 0 \parallel \text{index} > \text{size}()$ ).

## **boolean add(Object o)**

Appends the specified element to the end of this list.

## **boolean addAll(Collection c)**

Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator. Throws NullPointerException if the specified collection is null.

## **boolean addAll(int index, Collection c)**

Inserts all of the elements in the specified collection into this list, starting at the specified position. Throws NullPointerException if the specified collection is null.

# The LinkedList Class

## **void addFirst(Object o)**

Inserts the given element at the beginning of this list.

## **void addLast(Object o)**

Appends the given element to the end of this list.

## **void clear()**

Removes all of the elements from this list.

## **Object clone()**

Returns a shallow copy of this LinkedList.

## **boolean contains(Object o)**

Returns true if this list contains the specified element. More formally, returns true if and only if this list contains at least one element e such that ( $o==null ? e==null : o.equals(e)$ ).

## **Object get(int index)**

Returns the element at the specified position in this list. Throws IndexOutOfBoundsException if the specified index is out of range (index < 0 || index >= size()).

# The LinkedList Class

# The LinkedList Class

# The LinkedList Class

## **Object getFirst()**

Returns the first element in this list. Throws  
NoSuchElementException if this list is empty.

## **Object getLast()**

Returns the last element in this list. Throws  
NoSuchElementException if this list is empty.

## **int indexOf(Object o)**

Returns the index in this list of the first occurrence of the specified element, or -1 if the list does not contain this element.

## **int lastIndexOf(Object o)**

Returns the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element.

# The LinkedList Class

## **ListIterator listIterator(int index)**

Returns a list-iterator of the elements in this list (in proper sequence), starting at the specified position in the list. Throws IndexOutOfBoundsException if the specified index is out of range ( $\text{index} < 0 \parallel \text{index} \geq \text{size}()$ ).

## **Object remove(int index)**

Removes the element at the specified position in this list. Throws NoSuchElementException if this list is empty.

## **boolean remove(Object o)**

Removes the first occurrence of the specified element in this list. Throws NoSuchElementException if this list is empty. Throws IndexOutOfBoundsException if the specified index is out of range ( $\text{index} < 0 \parallel \text{index} \geq \text{size}()$ ).

## **Object removeFirst()**

Removes and returns the first element from this list. Throws NoSuchElementException if this list is empty.

# The LinkedList Class

## **Object removeLast()**

Removes and returns the last element from this list. Throws NoSuchElementException if this list is empty.

## **Object set(int index, Object element)**

Replaces the element at the specified position in this list with the specified element. Throws IndexOutOfBoundsException if the specified index is out of range ( $\text{index} < 0 \text{ || } \text{index} \geq \text{size}()$ ).

## **int size()**

Returns the number of elements in this list.

## **Object[] toArray()**

Returns an array containing all of the elements in this list in the correct order. Throws NullPointerException if the specified array is null.

## **Object[] toArray(Object[] a)**

Returns an array containing all of the elements in this list in the correct order; the runtime type of the returned array is that of the specified array.

## Set Interface Method & Description

### **add( )**

Adds an object to the collection.

### **clear( )**

Removes all objects from the collection.

### **contains( )**

Returns true if a specified object is an element within the collection.

### **isEmpty( )**

Returns true if the collection has no elements.

### **iterator( )**

Returns an Iterator object for the collection, which may be used to retrieve an object.

### **remove( )**

Removes a specified object from the collection.

### **size( )**

Returns the number of elements in the collection.

## SortedSet Method & Description

### **Comparator comparator( )**

Returns the invoking sorted set's comparator. If the natural ordering is used for this set, null is returned.

### **Object first( )**

Returns the first element in the invoking sorted set.

### **SortedSet headSet(Object end)**

Returns a SortedSet containing those elements less than end that are contained in the invoking sorted set. Elements in the returned sorted set are also referenced by the invoking sorted set.

### **Object last( )**

Returns the last element in the invoking sorted set.

### **SortedSet subSet(Object start, Object end)**

Returns a SortedSet that includes those elements between start and end.1. Elements in the returned collection are also referenced by the invoking object.

### **SortedSet tailSet(Object start)**

Returns a SortedSet that contains those elements greater than or equal to start that are contained in the sorted set. Elements in the returned set are also referenced by the invoking object.

# HashSet Class

- HashSet extends AbstractSet and implements the Set interface. It creates a collection that uses a hash table for storage.
- A hash table stores information by using a mechanism called **hashing**. In hashing, the informational content of a key is used to determine a unique value, called its hash code.
- The hash code is then used as the index at which the data associated with the key is stored. The transformation of the key into its hash code is performed automatically.

## HashSet( )

This constructor constructs a default HashSet.

## HashSet(Collection c)

This constructor initializes the hash set by using the elements of the collection c.

# HashSet Class

## **HashSet(int capacity)**

This constructor initializes the capacity of the hash set to the given integer value `capacity`. The capacity grows automatically as elements are added to the HashSet.

## **HashSet(int capacity, float fillRatio)**

This constructor initializes both the capacity and the fill ratio (also called load capacity) of the hash set from its arguments.

Here the fill ratio must be between 0.0 and 1.0, and it determines how full the hash set can be before it is resized upward. Specifically, when the number of elements is greater than the capacity of the hash set multiplied by its fill ratio, the hash set is expanded.

# HashSet Class

## **boolean add(Object o)**

Adds the specified element to this set if it is not already present.

## **void clear()**

Removes all of the elements from this set.

## **Object clone()**

Returns a shallow copy of this HashSet instance: the elements themselves are not cloned.

## **boolean contains(Object o)**

Returns true if this set contains the specified element.

## **boolean isEmpty()**

Returns true if this set contains no elements.

## **Iterator iterator()**

Returns an iterator over the elements in this set.

## **boolean remove(Object o)**

Removes the specified element from this set if it is present.

## **int size()**

Returns the number of elements in this set (its cardinality).

# The Map Interface

- The Map interface maps unique keys to values. A key is an object that you use to retrieve a value at a later date.
- Given a key and a value, you can store the value in a Map object. After the value is stored, you can retrieve it by using its key.
- Several methods throw a NoSuchElementException when no items exist in the invoking map.
- A ClassCastException is thrown when an object is incompatible with the elements in a map.
- A NullPointerException is thrown if an attempt is made to use a null object and null is not allowed in the map.
- An UnsupportedOperationException is thrown when an attempt is made to change an unmodifiable map.

## Map Interface Method & Description

### **void clear()**

Removes all key/value pairs from the invoking map.

### **boolean containsKey(Object k)**

Returns true if the invoking map contains **k** as a key. Otherwise, returns false.

### **boolean containsValue(Object v)**

Returns true if the map contains **v** as a value. Otherwise, returns false.

### **Set entrySet( )**

Returns a Set that contains the entries in the map. The set contains objects of type Map.Entry. This method provides a set-view of the invoking map.

### **boolean equals(Object obj)**

Returns true if **obj** is a Map and contains the same entries. Otherwise, returns false.

### **Object get(Object k)**

Returns the value associated with the key **k**.

### **int hashCode()**

Returns the hash code for the invoking map.

## Map Interface Method & Description

### **boolean isEmpty()**

Returns true if the invoking map is empty. Otherwise, returns false.

### **Set keySet()**

Returns a Set that contains the keys in the invoking map. This method provides a set-view of the keys in the invoking map.

### **Object put(Object k, Object v)**

Puts an entry in the invoking map, overwriting any previous value associated with the key. The key and value are k and v, respectively. Returns null if the key did not already exist. Otherwise, the previous value linked to the key is returned.

### **void putAll(Map m)**

Puts all the entries from m into this map.

### **Object remove(Object k)**

Removes the entry whose key equals k.

### **int size()**

Returns the number of key/value pairs in the map.

### **Collection values()**

Returns a collection containing the values in the map. This method provides a collection-view of the values in the map.

# HashMap Class

- The HashMap class uses a hashtable to implement the Map interface. This allows the execution time of basic operations, such as get( ) and put( ), to remain constant even for large sets.

## HashMap()

This constructor constructs a default HashMap.

## HashMap(Map m)

This constructor initializes the hash map by using the elements of the given Map object m.

## HashMap(int capacity)

This constructor initializes the capacity of the hash map to the given integer value, capacity.

## HashMap(int capacity, float fillRatio)

This constructor initializes both the capacity and fill ratio of the hash map by using its arguments.

# HashMap Class

## **void clear()**

Removes all mappings from this map.

## **Object clone()**

Returns a shallow copy of this HashMap instance: the keys and values themselves are not cloned.

## **boolean containsKey(Object key)**

Returns true if this map contains a mapping for the specified key.

## **boolean containsValue(Object value)**

Returns true if this map maps one or more keys to the specified value.

## **Set entrySet()**

Returns a collection view of the mappings contained in this map.

# HashMap Class

## **Object get(Object key)**

Returns the value to which the specified key is mapped in this identity hash map, or null if the map contains no mapping for this key.

## **boolean isEmpty()**

Returns true if this map contains no key-value mappings.

## **Set keySet()**

Returns a set view of the keys contained in this map.

## **Object put(Object key, Object value)**

Associates the specified value with the specified key in this map.

# HashMap Class

## **putAll(Map m)**

Copies all of the mappings from the specified map to this map. These mappings will replace any mappings that this map had for any of the keys currently in the specified map.

## **Object remove(Object key)**

Removes the mapping for this key from this map if present.

## **int size()**

Returns the number of key-value mappings in this map.

## **Collection values()**

Returns a collection view of the values contained in this map.

# HashMap Class

# The SortedMap Interface

- The SortedMap interface extends Map. It ensures that the entries are maintained in an ascending key order.
- Several methods throw a NoSuchElementException when no items are in the invoking map.
- A ClassCastException is thrown when an object is incompatible with the elements in a map.
- A NullPointerException is thrown if an attempt is made to use a null object when null is not allowed in the map.

## SortedMap Method & Description

### **Comparator comparator()**

Returns the invoking sorted map's comparator. If the natural ordering is used for the invoking map, null is returned.

### **Object firstKey()**

Returns the first key in the invoking map.

### **SortedMap headMap(Object end)**

Returns a sorted map for those map entries with keys that are less than end.

### **Object lastKey()**

Returns the last key in the invoking map.

### **SortedMap subMap(Object start, Object end)**

Returns a map containing those entries with keys that are greater than or equal to start and less than end.

### **SortedMap tailMap(Object start)**

Returns a map containing those entries with keys that are greater than or equal to start.

# The Enumeration Interface

- The Enumeration interface defines the methods by which you can enumerate (obtain one at a time) the elements in a collection of objects.
- This legacy interface has been superceded by Iterator. Although not deprecated, Enumeration is considered obsolete for new code.
- However, it is used by several methods defined by the legacy classes such as Vector and Properties, is used by several other API classes, and is currently in widespread use in application code.

# The Enumeration Interface

- **boolean hasMoreElements( )**

When implemented, it must return true while there are still more elements to extract, and false when all the elements have been enumerated.

- **Object nextElement( )**

This returns the next object in the enumeration as a generic Object reference.

# How to Use Iterator?

- Often, you will want to cycle through the elements in a collection.
- For example, you might want to display each element. The easiest way to do this is to employ an iterator, which is an object that implements either the Iterator or the ListIterator interface.
- Iterator enables you to cycle through a collection, obtaining or removing elements.
- ListIterator extends Iterator to allow bidirectional traversal of a list, and the modification of elements.
- Before you can access a collection through an iterator, you must obtain one.
- Each of the collection classes provides an iterator( ) method that returns an iterator to the start of the collection. By using this iterator object, you can access each element in the collection, one element at a time.

# How to Use Iterator?

- In general, to use an iterator to cycle through the contents of a collection, follow these steps –
  - Obtain an iterator to the start of the collection by calling the collection's iterator( ) method.
  - Set up a loop that makes a call to hasNext( ). Have the loop iterate as long as hasNext( ) returns true.
  - Within the loop, obtain each element by calling next( ).

- **Iterator Methods**

- **boolean hasNext( )**

Returns true if there are more elements. Otherwise, returns false.

- **Object next( )**

Returns the next element. Throws NoSuchElementException if there is not a next element.

- **void remove( )**

Removes the current element. Throws IllegalStateException if an attempt is made to call remove( ) that is not preceded by a call to next( ).

# How to Use Iterator?

- **ListIterator Methods**
- **void add(Object obj)**

Inserts obj into the list in front of the element that will be returned by the next call to next( ).

- **boolean hasNext()**  
Returns true if there is a next element. Otherwise, returns false.
- **boolean hasPrevious()**  
Returns true if there is a previous element. Otherwise, returns false.
- **Object next()**  
Returns the next element. A NoSuchElementException is thrown if there is not a next element.

# How to Use Iterator?

- **int nextIndex( )**  
Returns the index of the next element. If there is not a next element, returns the size of the list.
- **Object previous( )**  
Returns the previous element. A NoSuchElementException is thrown if there is not a previous element.
- **int previousIndex( )**  
Returns the index of the previous element. If there is not a previous element, returns -1.
- **void remove( )**  
Removes the current element from the list. An IllegalStateException is thrown if remove( ) is called before next( ) or previous( ) is invoked.
- **void set(Object obj)**  
Assigns obj to the current element. This is the element last returned by a call to either next( ) or previous( ).

# Collections – Comparable

- The Comparable interface labels objects that can be compared to one another.
  - Allows sorting algorithms to be written to work on any kind of object
  - so long as they support this interface
- Single method to implement

```
public int compareTo(Object o);
```

- Returns
  - A negative number if parameter is less than the object
  - Zero if they're equal
  - A positive number if the parameter is greater than the object

# Comparator

- Both TreeSet and TreeMap store elements in sorted order. However, it is the comparator that defines precisely what *sorted order* means.
- The Comparator interface defines two methods: compare( ) and equals( ). The compare( ) method, shown here, compares two elements for order –
- **The compare Method**
- int compare(Object obj1, Object obj2)  
obj1 and obj2 are the objects to be compared. This method returns zero if the objects are equal. It returns a positive value if obj1 is greater than obj2. Otherwise, a negative value is returned.
- By overriding compare( ), you can alter the way that objects are ordered. For example, to sort in a reverse order, you can create a comparator that reverses the outcome of a comparison.

# Comparator

- **The equals Method**
- The equals( ) method, shown here, tests whether an object equals the invoking comparator –
- boolean equals(Object obj) obj is the object to be tested for equality. The method returns true if obj and the invoking object are both Comparator objects and use the same ordering. Otherwise, it returns false.
- Overriding equals( ) is unnecessary, and most simple comparators will not do so.

# GUI & Event Handling

# GUI History

When Java 1.0 was introduced, it contained a class library, called the Abstract Window Toolkit (AWT), for basic GUI programming.

The basic AWT library deals with user interface elements by delegating their creation and behavior to the native GUI toolkit on each target platform (Windows, Linux, Macintosh, and so on).

For example, if you used the original AWT to put a text box on a Java window. The resulting program could then, in theory, run on any of these platforms, with the “look-and-feel” of the target platform.

# GUI History

In 1996, Netscape created a GUI library they called the IFC (Internet Foundation Classes) that used an entirely different approach. User interface elements, such as buttons, menus, and so on, were painted onto blank windows.

The only functionality required from the underlying windowing system was a way to put up a window and to paint on it. Thus, Netscape's IFC widgets looked and behaved the same no matter which platform the program ran on.

Sun Microsystems worked with Netscape to perfect this approach, creating a user interface library with the code name “Swing.” Swing was available as an extension to Java 1.1 and became a part of the standard library in Java 1.2.

Swing is now the official name for the non-peer-based GUI toolkit.

# Introduction

JAVA provides a rich set of libraries to create Graphical User Interface.

- AWT (Abstract Window Toolkit).
- Swing

Graphical User Interface (GUI) offers user interaction via some graphical components.

- For example our underlying Operating System also offers GUI via window, frame, Panel, Button, Textfield, TextArea, Listbox, Combobox, Label, Checkbox etc. These all are known as components. Using these components we can create an interactive user interface for an application.

GUI provides result to end user in response to raised events. GUI is entirely based events.

- For example clicking over a button, closing a window, opening a window, typing something in a textarea etc. These activities are known as events. GUI makes it easier for the end user to use an application. It also makes them interesting

# Terminologies

**Component** Component is an object having a graphical representation that can be displayed on the screen and that can interact with the user. For examples buttons, checkboxes, list and scrollbars of a graphical user interface.

**Container** Container object is a component that can contain other components. Components added to a container are tracked in a list. The order of the list will define the components' front-to-back stacking order within the container. If no index is specified when adding a component to a container, it will be added to the end of the list.

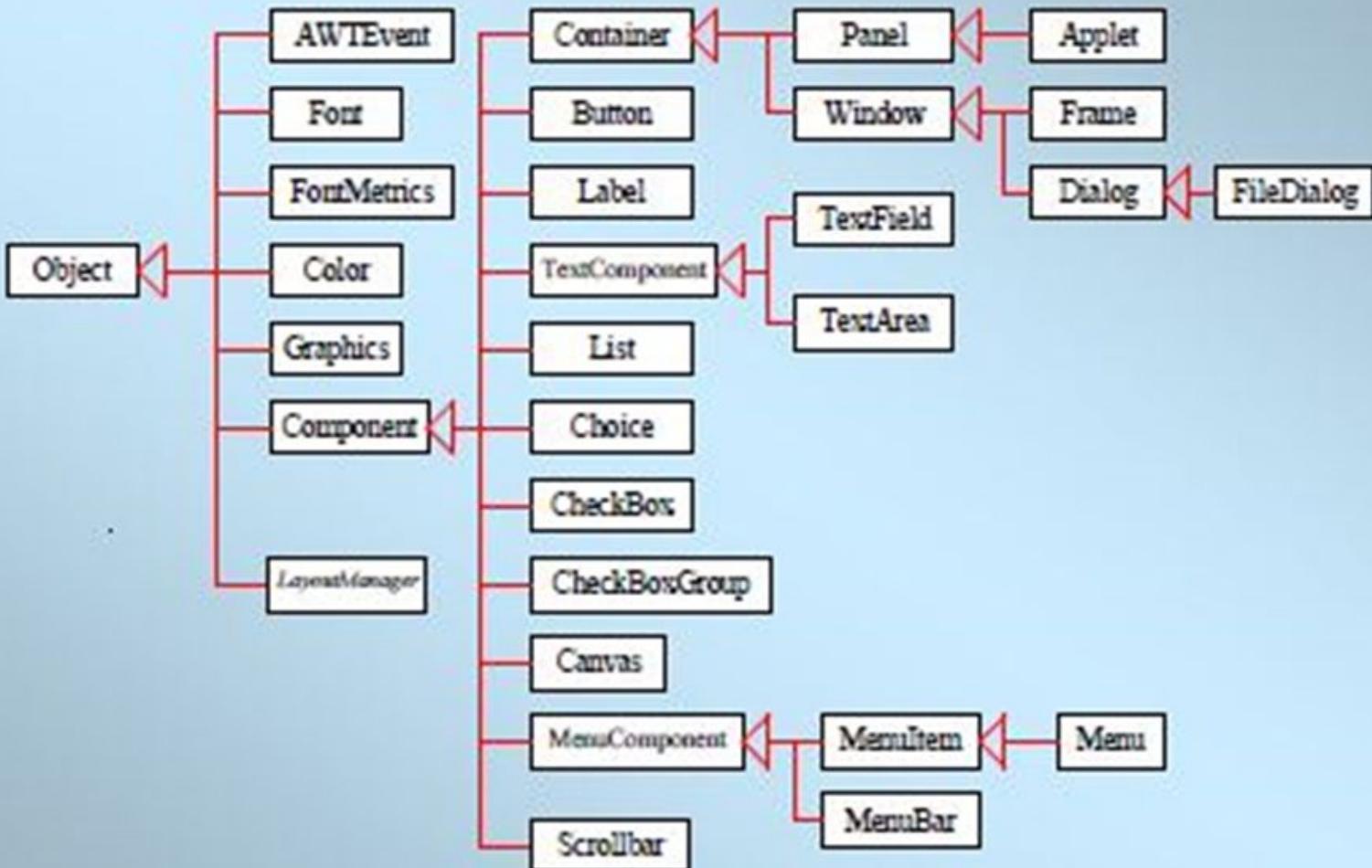
**Window** Window is a rectangular area which is displayed on the screen. In different window we can execute different program and display different data. Window provide us with multitasking environment. A window must have either a frame, dialog, or another window defined as its owner when it's constructed.

# Terminologies

**Panel** Panel provides space in which an application can attach any other components, including other panels.

**Frame** A Frame is a top-level window with a title and a border. The size of the frame includes any area designated for the border. Frame encapsulates window. It has a title bar, menu bar, borders, and resizing corners.

**Canvas** Canvas component represents a blank rectangular area of the screen onto which the application can draw. Application can also trap input events from the user from that blank area of Canvas component.



# Component CLASS

The class Component is the abstract base class for the non menu user-interface controls of AWT. Component represents an object with graphical representation.

- boolean isVisible()
- void setVisible(boolean b)
- void setSize(int width, int height)
- void setLocation(int x, int y)
- void setBounds(int x, int y, int width, int height)
- Dimension getSize()
- void setSize(Dimension d)

# Container Class

The class `Container` is the super class for the containers of AWT. `Container` object can contain other AWT components.

- `Component add(Component comp)`
- `void addContainerListener(ContainerListener l)`
- `float getAlignmentX()` `float getAlignmentY()`
- `LayoutManager getLayout()`
- `Dimension getMaximumSize()` `Dimension getMinimumSize()`
- ...

# Graphics class

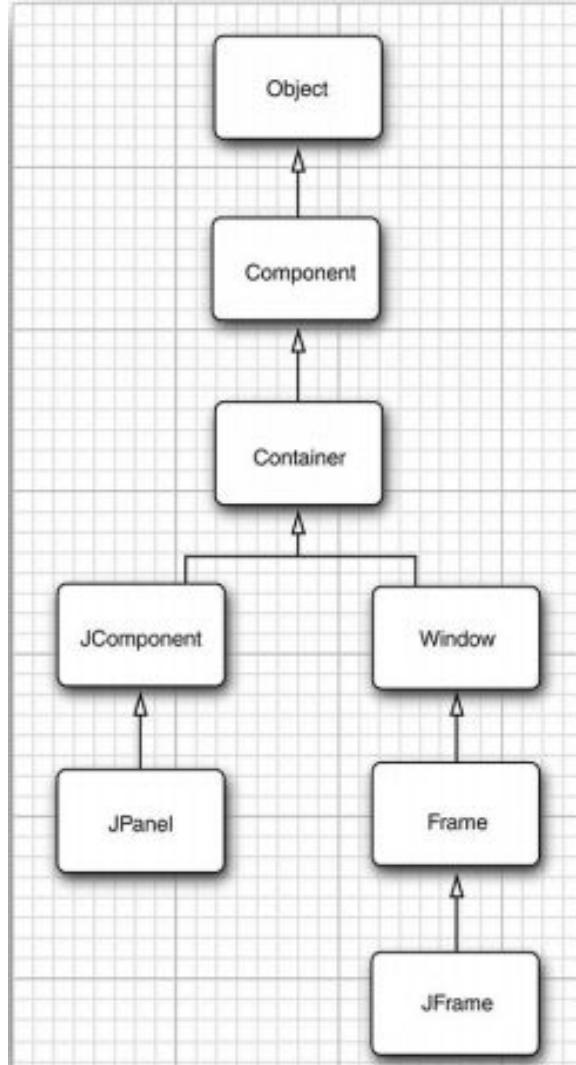
The Graphics class is the abstract super class for all graphics contexts which allow an application to draw onto components that can be realized on various devices, or onto off-screen images as well.

**public abstract class Graphics extends Object**

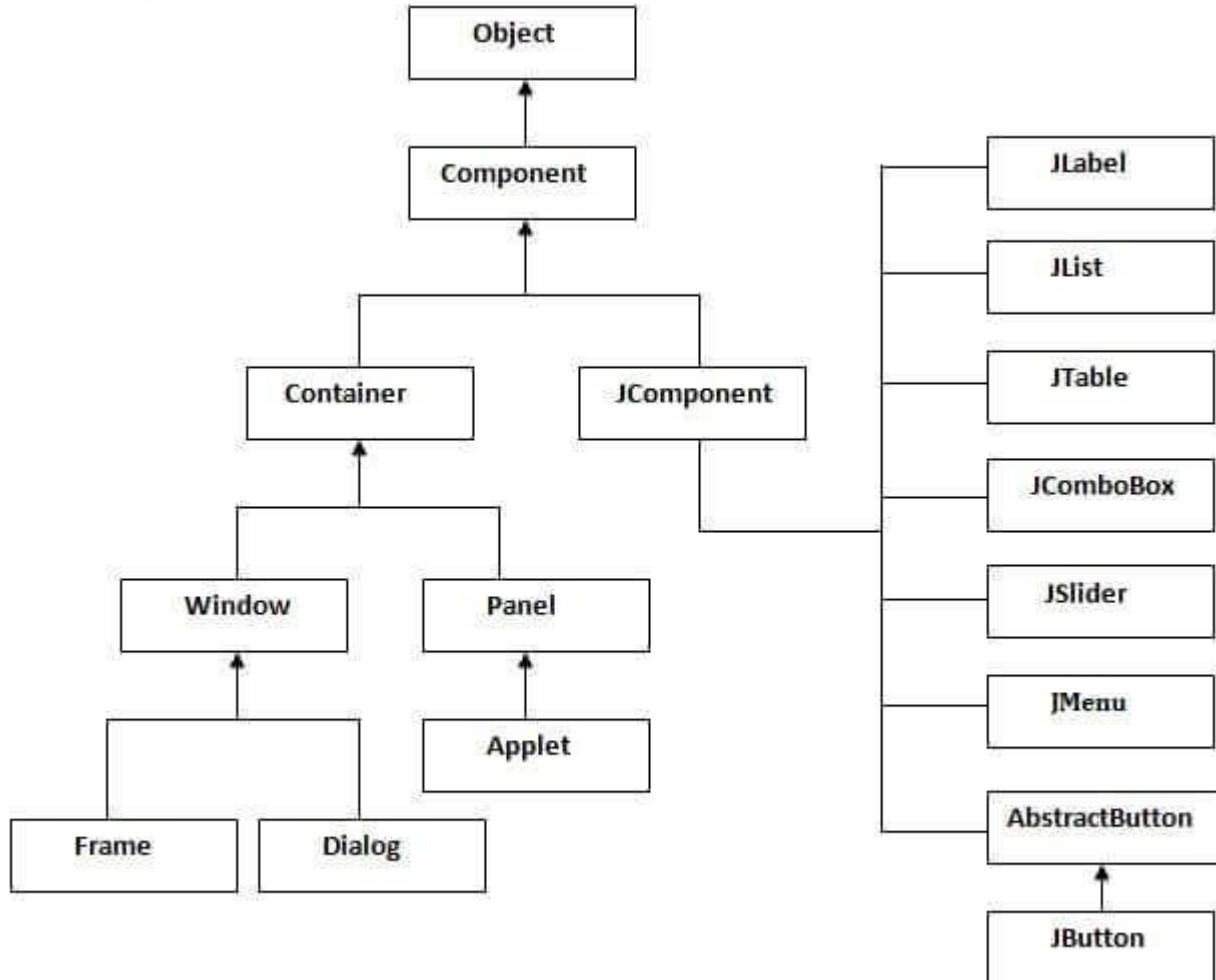
A Graphics object encapsulates all state information required for the basic rendering operations that Java supports.

State information includes the following properties.

- The Component object on which to draw.
- A translation origin for rendering and clipping coordinates.
- The current clip.
- The current color.
- The current font.
- The current logical pixel operation function.
- The current XOR alternation color



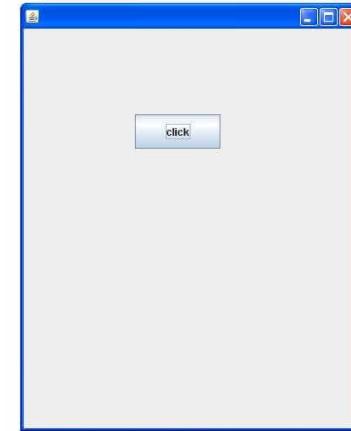
# Swing Classes



# Swing JFrame

```
import javax.swing.*;
public class First SwingExample {
 public static void main(String[] args) {

 JFrame f=new JFrame(); //creating instance of JFrame
 JButton b=new JButton("click"); //creating instance of JButton
 b.setBounds(130,100,100, 40); //x axis, y axis, width, height
 f.add(b); //adding button in JFrame
 f.setSize(400,500); //400 width and 500 height
 f.setLayout(null); //using no layout managers
 f.setVisible(true); //making the frame visible
 }
}
```



# Swing JFrame

```
import javax.swing.*;
public class Simple {
 JFrame f;
 Simple(){
 f=new JFrame();//creating instance of JFrame

 JButton b=new JButton("click");//creating instance of JButton
 b.setBounds(130,100,100, 40);

 f.add(b);//adding button in JFrame

 f.setSize(400,500);//400 width and 500 height
 f.setLayout(null);//using no layout managers
 f.setVisible(true);//making the frame visible
 }

 public static void main(String[] args) {
 new Simple();
 }
}
```

# Swing JFrame

```
import javax.swing.*;
public class Simple2 extends JFrame{//inheriting JFrame
 JFrame f;
 Simple2(){
 JButton b=new JButton("click");//create button
 b.setBounds(130,100,100, 40);

 add(b);//adding button on frame
 setSize(400,500);
 setLayout(null);
 setVisible(true);
 }
 public static void main(String[] args) {
 new Simple2();
 }
}
```

| Modifier and Type   | Method                                                              | Description                                                                                                                                                                                     |
|---------------------|---------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| protected void      | addImpl(Component comp, Object constraints, int index)              | Adds the specified child Component.                                                                                                                                                             |
| protected JRootPane | createRootPane()                                                    | Called by the constructor methods to create the default rootPane.                                                                                                                               |
| protected void      | frameInit()                                                         | Called by the constructors to init the JFrame properly.                                                                                                                                         |
| void                | setContentPane(Contain contentPane)                                 | It sets the contentPane property                                                                                                                                                                |
| static void         | setDefaultLookAndFeelDecorated(boolean defaultLookAndFeelDecorated) | Provides a hint as to whether or not newly created JFrames should have their Window decorations (such as borders, widgets to close the window, title...) provided by the current look and feel. |
| void                | setIconImage(Image image)                                           | It sets the image to be displayed as the icon for this window.                                                                                                                                  |

| Modifier and Type | Method                                   | Description                                    |
|-------------------|------------------------------------------|------------------------------------------------|
| void              | setJMenuBar(JMenuBar menuBar)            | It sets the menubar for this frame.            |
| void              | setLayeredPane(JLayeredPane layeredPane) | It sets the layeredPane property.              |
| JRootPane         | getRootPane()                            | It returns the rootPane object for this frame. |
| TransferHandler   | getTransferHandler()                     | It gets the transferHandler property.          |

```
import java.awt.FlowLayout;
import javax.swing.*;
public class JFrameExample {
 public static void main(String s[]) {
 JFrame frame = new JFrame("JFrame Example");
 JPanel panel = new JPanel();
 panel.setLayout(new FlowLayout());
 JLabel label = new JLabel("JFrame By Example");
 JButton button = new JButton();
 button.setText("Button");
 panel.add(label);
 panel.add(button);
 frame.add(panel);
 frame.setSize(200, 300);
 frame.setLocationRelativeTo(null);
 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
 frame.setVisible(true);
 }
}
```

# Event Handling

# What is an Event?

- Change in the state of an object is known as event.
- Events are generated as result of user interaction with the graphical user interface components.
- E.g.
  - clicking on a button
  - moving the mouse
  - entering a character through keyboard
  - selecting an item from list
  - scrolling the page are the activities that causes an event to happen.
- Applets and java graphics Programming are event-driven.
- Events are supported by `java.awt.event` package.

# Types of Event

## Foreground Events

- Those events which require the direct interaction of user.
- They are generated as consequences of a person interacting with the graphical components in Graphical User Interface.
- For example, clicking on a button, moving the mouse, entering a character through keyboard, selecting an item from list, scrolling the page etc.

## Background Events

- Those events that are not directly caused by the interactions with the user interface are known as background events.
- Operating system interrupts, hardware or software failure, timer expires, an operation completion are the example of background events.

# What is Event Handling?

- Event Handling is the mechanism that controls the event and decides what should happen if an event occurs.
- This mechanism has the code which is known as event handler that is executed when an event occurs.
- For any **event** to occur, the **objects** register themselves as **listeners**.
- No event takes place if there is no listener i.e. nothing happens when an event takes place if there is no listener.
- No matter how many listeners there are, each and every listener is capable of processing an event.
- Java Uses the Delegation Event Model to handle the events.

# The Delegation Event Model

- **Source** - The source is an object on which event occurs.
  - Sources may generate more than one type of event.
  - Source is responsible for providing information of the occurred event to its handler.
  - Java provides us with classes for source object.
- **Listener** - It is also known as event handler.
  - Listener is responsible for generating response to an event.
  - From Java implementation point of view the listener is also an object.
  - Listener waits until it receives an event. Once the event is received, the listener processes the event and then returns.

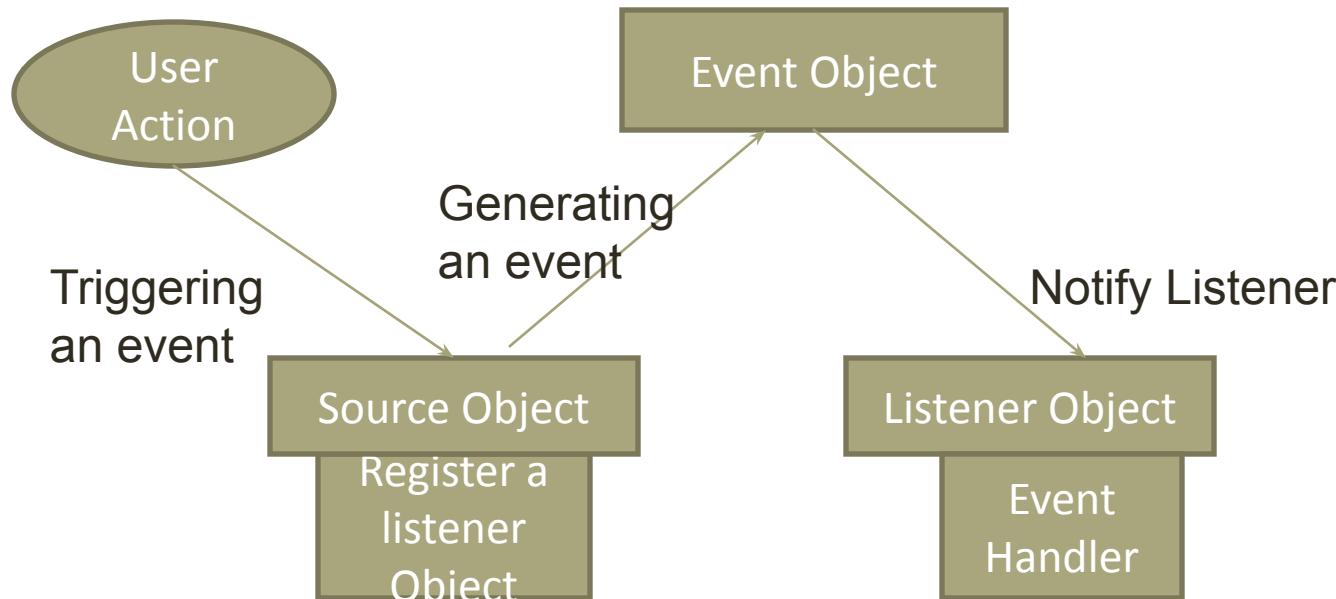
# Con't

- A source must register listeners in order for the listeners to receive notifications about a specific type of event.
  - `public void addTypeListener(TypeListener el)`
  - *Type is the name of the event*
    - **E.g. `addKeyListener()`**
  - When an event occurs, all registered listeners are notified and receive a copy of the event object. This is known as *multicasting the event*.

# Con't

- Some sources may allow only one listener to register.
  - `public void addTypeListener(TypeListener el) throws java.util.TooManyListenersException`
- When such an event occurs, the registered listener is notified. This is known as *unicasting the event*.
- The methods that add or remove listeners are provided by the source that generates events.
  - `public void removeTypeListener(TypeListener el)`

# Delegation based Model



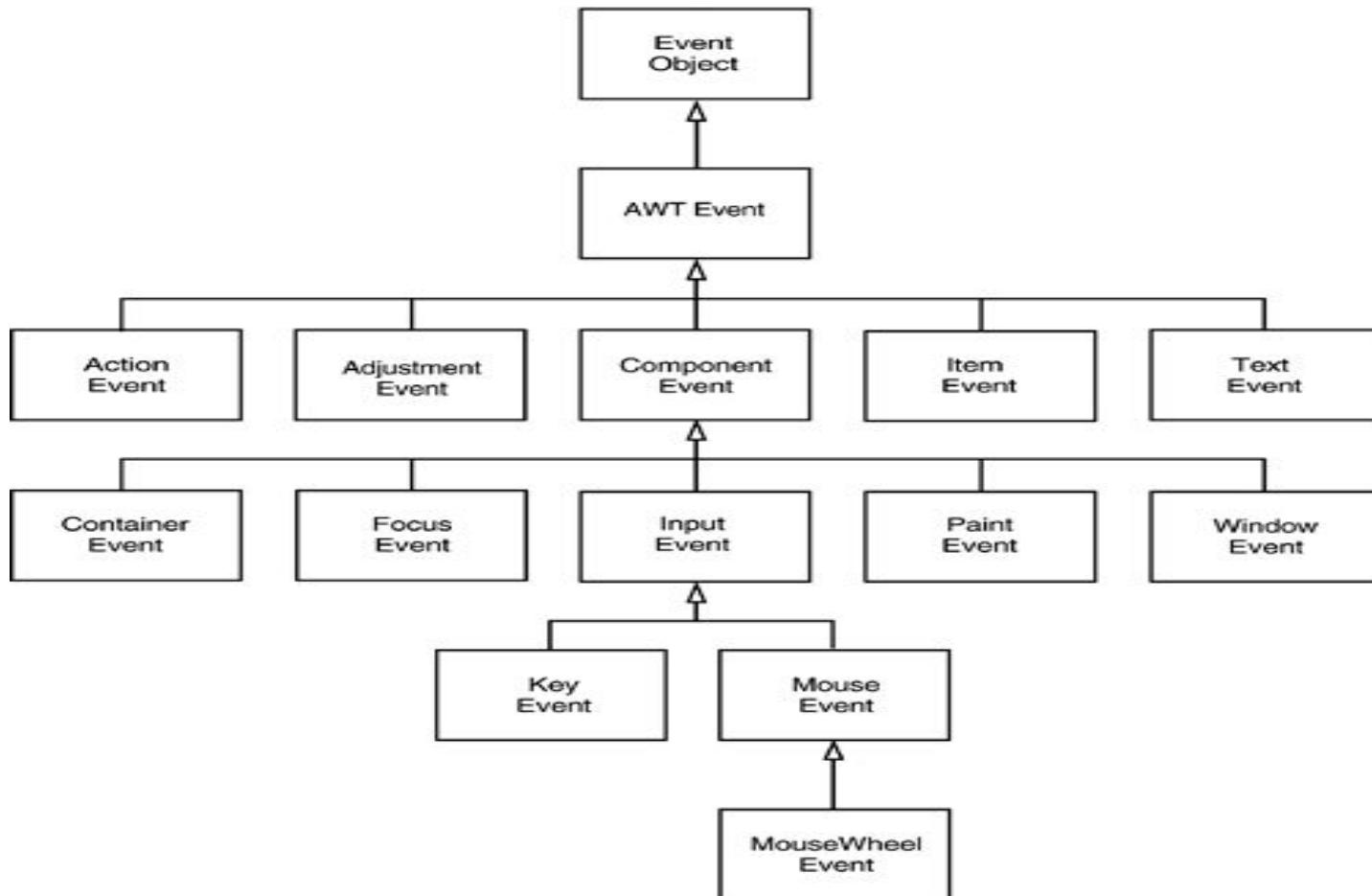
# Steps involved in event handling

- The User clicks the button and the event is generated.
- Now the object of concerned event class is created automatically and information about the source and the event get populated with in same object.
- Event object is forwarded to the method of registered listener class.
- The method is now get executed and returns.

# Con't

- The benefit of this approach is that the user interface logic is completely separated from the logic that generates the event.
- The user interface element is able to delegate the processing of an event to the separate piece of code.
- In this model ,Listener needs to be registered with the source object so that the listener can receive the event notification.
- This is an efficient way of handling the event because the event notifications are sent only to those listener that want to receive them.

# Hierarchy relationship of AWT events



# **java.util.EventObject class**

- Superclass of all events.
- Constructor
  - **EventObject(Object source)**
- Methods
  - **Object getSource()**
    - The object on which the Event initially occurred.
  - **String toString()**
    - Returns a String representation of this EventObject.

# **java.awt.AWTEvent class**

- AWTEvent is a superclass of all AWT events that are handled by the delegation event model.
- This class and its subclasses supercede the original java.awt.Event class.
- **int getID()** - Returns the event type

# Event classes in `java.awt.Event`

| 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 super class 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. |

# Con't

| Event Class     | Description                                                                                                                           |
|-----------------|---------------------------------------------------------------------------------------------------------------------------------------|
| 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. (Added by Java 2, version 1.4)                                                               |
| 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.                                   |

# java.awt.event.ActionEvent class

## Constants

- ACTION\_FIRST ACTION\_LAST ACTION\_PERFORMED
- ALT\_MASK CTRL\_MASK META\_MASK
- SHIFT\_MASK

## Constructor

- ActionEvent(Object *src*, int *type*, String *cmd*)
- ActionEvent(Object *src*, int *type*, String *cmd*, int *modifiers*)
- ActionEvent(Object *src*, int *type*, String *cmd*, long *when*, int *modifiers*)

## Methods

- String getActionCommand( )
- int getModifiers( )
- long getWhen( )

# java.awt.event.InputEvent class

## Constants(old)

|                |              |            |
|----------------|--------------|------------|
| ALT_MASK       | BUTTON2_MASK | META_MASK  |
| ALT_GRAPH_MASK | BUTTON3_MASK | SHIFT_MASK |
| BUTTON1_MASK   | CTRL_MASK    |            |

## Constants(new)

|                   |                     |                   |
|-------------------|---------------------|-------------------|
| ALT_DOWN_MASK     | ALT_GRAPH_DOWN_MASK | BUTTON1_DOWN_MASK |
| BUTTON2_DOWN_MASK | BUTTON3_DOWN_MASK   | CTRL_DOWN_MASK    |
| META_DOWN_MASK    | SHIFT_DOWN_MASK     |                   |

## Methods

- boolean isAltDown( )
- boolean isAltGraphDown( )
- boolean isControlDown( )
- boolean isMetaDown( )
- int getModifiersEx( )
- boolean isShiftDown( )
- int getModifiers( )

# java.awt.event.KeyEvent class

Three types of key events:

- KEY\_PRESSED
- KEY\_RELEASED
- KEY\_TYPED
- VK\_0 through VK\_9 and VK\_A through VK\_Z define the ASCII equivalents of the numbers and letters.

VK\_ENTER      VK\_ESCAPE      VK\_CANCEL      VK\_UP

VK\_DOWN      VK\_LEFT      VK\_RIGHT      VK\_PAGE\_DOWN

VK\_PAGE\_UP      VK\_SHIFT      VK\_ALT      VK\_CONTROL

- The VK constants specify *virtual key codes and are independent of any modifiers, such as control, shift, or alt.*

# Con't

## Constructor

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

## Methods

- char getKeyChar( )
- int getKeyCode( )

# java.awt.event.MouseEvent class

## Fields

|                |                                    |
|----------------|------------------------------------|
| 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          |

## Constructor

- `MouseEvent(Component src, int type, long when, int modifiers, int x, int y, int clicks, boolean triggersPopup)`

## Method

- `int getX( )`
- `int getY( )`
- `Point getPoint( )`

# Con't

- void translatePoint(int *x*, int *y*)
- int getClickCount( )
- boolean isPopupTrigger( )
- int getButton( )

# java.awt.event.TextEvent class

## Constant

- TEXT\_VALUE\_CHANGED

## Constructor

- TextEvent(Object *src*, int *type*)

## Method

- String paramString()

# java.awt.event.WindowEvent class

## Constants

- `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.
- `WINDOW_STATE_CHANGED` The state of the window changed.

# Con't

## Constructors

- `WindowEvent(Window src, int type)`
- `WindowEvent(Window src, int type, Window other)`
- `WindowEvent(Window src, int type, int fromState, int toState)`
- `WindowEvent(Window src, int type, Window other, int fromState, int toState)`

## Methods

- `Window getWindow( )`
- `Window getOppositeWindow()`
- `int getOldState()`
- `int getNewState()`

# java.awt.event.AdjustmentEvent class

## Constants

- **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.

## Constructor

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

## Methods

Adjustable getAdjustable( )

```
int getAdjustmentType()
```

```
int getValue()
```

# java.awt.Component class

## Constants

- COMPONENT\_HIDDEN The component was hidden.
- COMPONENT\_MOVED The component was moved.
- COMPONENT\_RESIZED The component was resized.
- COMPONENT\_SHOWN The component became visible.

## Constructor

- ComponentEvent(Component *src*, int *type*)

## Method

- Component getComponent( )

# ContainerEvent class

- Container Event is generated when a component is added to or removed from a container.

## Constants

- COMPONENT\_ADDED
- COMPONENT\_REMOVED

## Constructor

- ContainerEvent(Component *src*, int *type*, Component *comp*)

## Methods

- Container getContainer( )
- Component getChild( )

# MouseWheelEvent Class

- It is a subclass of MouseEvent.

## Constants

- WHEEL\_BLOCK\_SCROLL     A page-up or page-down scroll event occurred.
- WHEEL\_UNIT\_SCROLLA line-up or line-down scroll event occurred.

## Constructor

- `MouseWheelEvent(Component src, int type, long when, int modifiers, int x, int y, int clicks, boolean triggersPopup, int scrollHow, int amount, int count)`

## Methods

- `int getWheelRotation( )`
- `int getScrollType( )`
- `int getScrollAmount( )`

# FocusEvent Class

## Constants

- FOCUS\_GAINED
- FOCUS\_LOST

## Constructor

- FocusEvent(Component *src*, int *type*)
- FocusEvent(Component *src*, int *type*, boolean *temporaryFlag*)
- Focus Event(Component *src*, int *type*, boolean *temporaryFlag*, Component *other*)

## Methods

- Component getOppositeComponent( )
- boolean isTemporary( )

# ItemEvent Class

## Constants

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

## Constructor

- ItemEvent(*ItemSelectable src, int type, Object entry, int state*)

## Method

- Object getItem( )
- ItemSelectable getItemSelectable( )
- int getStateChange( )

# Sources of Events

| <b>Event Source</b> | <b>Description</b>                                                                                                                |
|---------------------|-----------------------------------------------------------------------------------------------------------------------------------|
| Button              | Generates action events when the button is pressed.                                                                               |
| Checkbox            | 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. |
| Scrollbar           | 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

| 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. (Added by Java 2, version 1.4)                                   |
| 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. (Added by Java 2, version 1.4)                       |
| WindowListener      | Defines seven methods to recognize when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.    |

# The ActionListener Interface

- The actionPerformed( ) method that is invoked when an action event occurs.
  - void actionPerformed(ActionEvent ae)

# The AdjustmentListener Interface

- The adjustmentValueChanged( ) method that is invoked when an adjustment event occurs.
  - void adjustmentValueChanged(AdjustmentEvent ae)

# The ComponentListener Interface

- Methods that are invoked when a component is resized, moved, shown, or hidden.
  - void componentResized(ComponentEvent ce)
  - void componentMoved(ComponentEvent ce)
  - void componentShown(ComponentEvent ce)
  - void componentHidden(ComponentEvent ce)

# The ContainerListener Interface

- When a component is added to a container, componentAdded( ) is invoked.
  - void componentAdded(ContainerEvent *ce*)
- When a component is removed from a container, componentRemoved( ) is invoked.
  - void componentRemoved(ContainerEvent *ce*)

# The FocusListener Interface

- When a component obtains keyboard focus, focusGained() is invoked.
  - void focusGained(FocusEvent *fe*)
- When a component loses keyboard focus, focusLost( ) is called.
  - void focusLost(FocusEvent *fe*)

# The ItemListener Interface

- The `itemStateChanged( )` method that is invoked when the state of an item changes.
  - `void itemStateChanged(ItemEvent ie)`

# The KeyListener Interface

- The **keyPressed()** and **keyReleased()** methods are invoked when a key is pressed and released, respectively.
  - `void keyPressed(KeyEvent ke)`
  - `void keyReleased(KeyEvent ke)`
- The **keyTyped()** method is invoked when a character has been entered.
  - `void keyTyped(KeyEvent ke)`
  - E.g. If a user presses and releases the **A** key, three events are generated in sequence: key pressed, typed, and released.
  - If a user presses and releases the **HOME** key, two key events are generated in sequence: key pressed and released.

# The MouseListener Interface

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

# The MouseMotionListener Interface

- void mouseDragged(MouseEvent *me*)
- void mouseMoved(MouseEvent *me*)

# The MouseWheelListener Interface

- void mouseWheelMoved(MouseWheelEvent *mwe*)

# The TextListener Interface

- The `textChanged( )` method that is invoked when a change occurs in a text area or text field.
  - `void textChanged(TextEvent te)`

# The WindowFocusListener Interface

- `void windowGainedFocus(WindowEvent we)`
- `void windowLostFocus(WindowEvent we)`

# The WindowListener Interface

- `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

- Two steps:
  - 1) Implement the appropriate interface in the listener so that it will receive the type of event desired.
  - 2) Implement code to register and unregister (if necessary) the listener as a recipient for the event notifications.
- A source may generate several types of events.
- Each event must be registered separately.
- An object may register to receive several types of events, must implement all of the interfaces that are required to receive these events.
- `SimpleButtonEvent.java`
- `MouseEvents.java`
- `SimpleKey.java`
- `KeyEvents.java`

# Adapter classes

- Adapter class provides an empty implementation of all methods in an event listener interface.
- Adapter classes are useful when you want to receive and process only some of the events that are handled by a particular event listener interface.
- E.g. MouseMotion Adapter class has two methods, mouseDragged() and mouseMoved().
  - The signatures of these empty methods are exactly as defined in the MouseMotionListener interface.
  - If you were interested in only mouse drag events, then you could simply extend MouseMotionAdapter and implement mouseDragged( ).
  - The empty implementation of mouseMoved( ) would handle the mouse motion events.

# Listener Interfaces implemented by Adapter classes

| <b>Adapter Class</b> | <b>Listener Interface</b> |
|----------------------|---------------------------|
| ComponentAdapter     | ComponentListener         |
| ContainerAdapter     | ContainerListener         |
| FocusAdapter         | FocusListener             |
| KeyAdapter           | KeyListener               |
| MouseAdapter         | MouseListener             |
| MouseMotionAdapter   | MouseMotionListener       |
| WindowAdapter        | WindowListener            |

- AdapterDemo.java
- MousePressedDemo.java

# Inner classes

- InnerClassDemo.java
- MyMouseAdapter is defined within the scope of InnerClassdemo, it has access to all of the variables and methods within the scope of that class.

## Local inner class

- Declare an inner class within the body of a method. Such a class is known as a local inner class.
- To declare an inner class within the body of a method without naming it.
  - These classes are known as anonymous inner classes.

# Anonymous classes

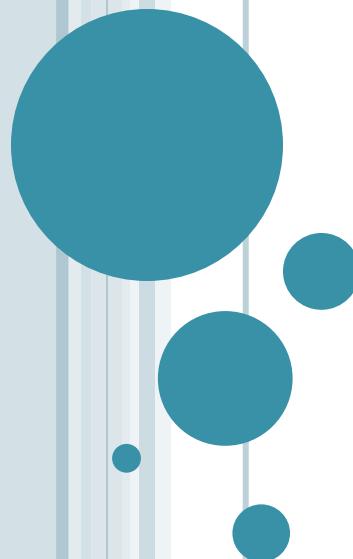
- An anonymous inner class is one that is not assigned a name.
- **AnonymousInnerClassDemo.java**
- new MouseAdapter(){...} – indicates to the compiler that the code between the braces defines an anonymous inner class.
  - It is automatically instantiated when this expression is executed.
- Anonymous inner class is defined within the scope of **AnonymousInnerClassDemo**, it has access to all of the variables and methods within the scope of that class.

# Semantic and Low-Level Events in the AWT

- Low level events are those produced by the keyboard, mouse and those related to windows and are common to all the components: KeyEvent, FocusEvent, MouseEvent, WindowEvent, ComponentEvent and ContainerEvent.
  - Keyboard and mouse are regarded as low level input.
- A semantic event is one that is specific to one particular component. E.g. "clicking a button"; hence, an ActionEvent is a semantic event.
- Low-level events are those events that make this possible. In the case of a button click, a mouse down is low-level event.
- Adjusting a scrollbar is a semantic event, but dragging the mouse is a low-level event.



# EXCEPTION HANDLING



# CHAPTER GOALS

---

- To learn how to throw exceptions
- To be able to design your own exception classes
- To understand the difference between checked and unchecked exceptions
- To learn how to catch exceptions
- To know when and where to catch an exception



# ERROR HANDLING

---

- An Error "indicates serious problems that a reasonable application should not try to catch." while
  - An Exception "indicates conditions that a reasonable application might want to catch."

*Continued...*

# ERROR HANDLING

---

- Instead of programming for success

```
x.doSomething()
```

- you would always be programming for failure:

```
if (!x.doSomething()) return false;
```



# THROWING EXCEPTIONS

- Exceptions:
  - Can't be overlooked
  - Sent directly to an exception handler—not just caller of failed method
- Throw an exception object to signal an exceptional condition
- Example: `IllegalArgumentException`:

```
illegal parameter value
IllegalArgumentException exception
 = new IllegalArgumentException("Amount exceeds balance");
throw exception;
```

# THROWING EXCEPTIONS

---

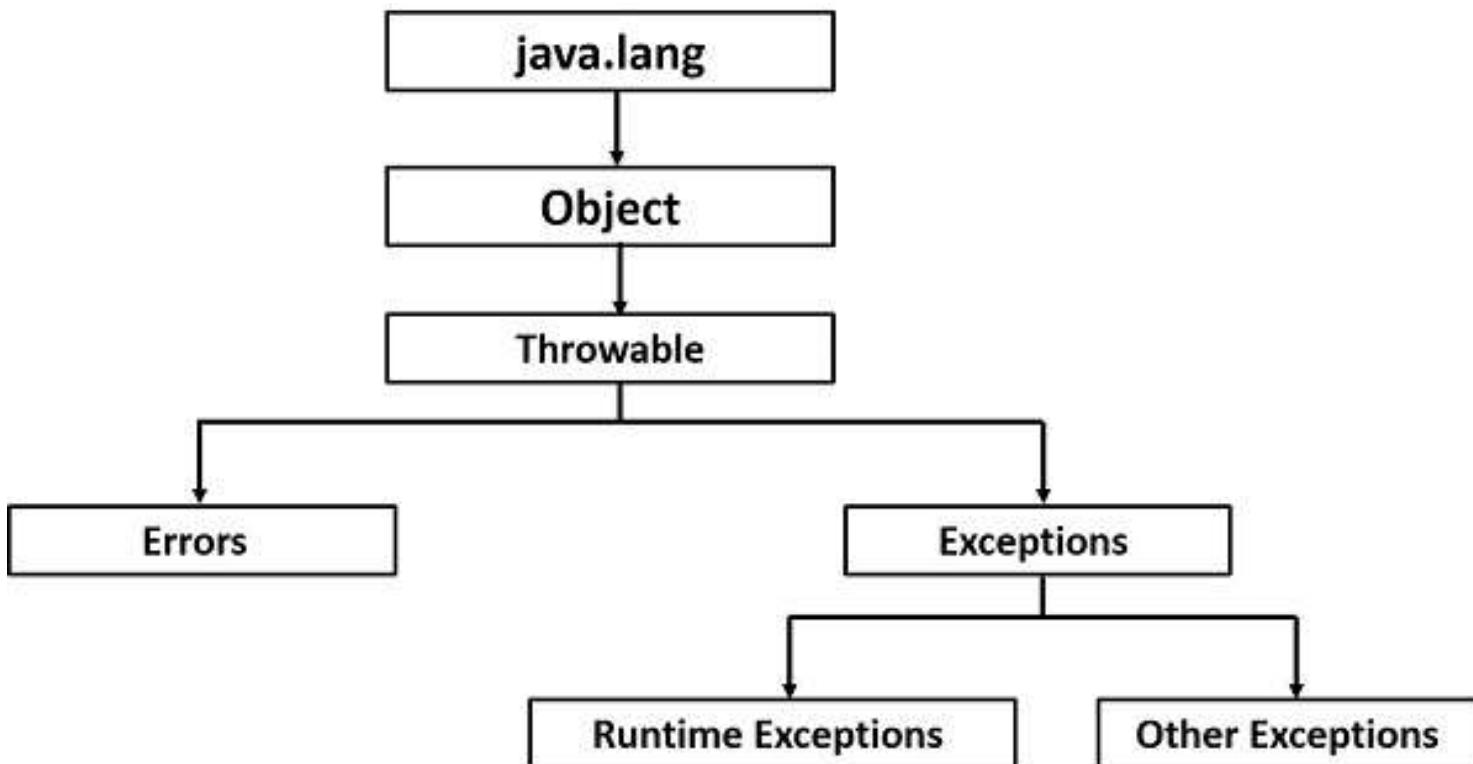
- No need to store exception object in a variable:

```
throw new IllegalArgumentException("Amount exceeds balance");
```

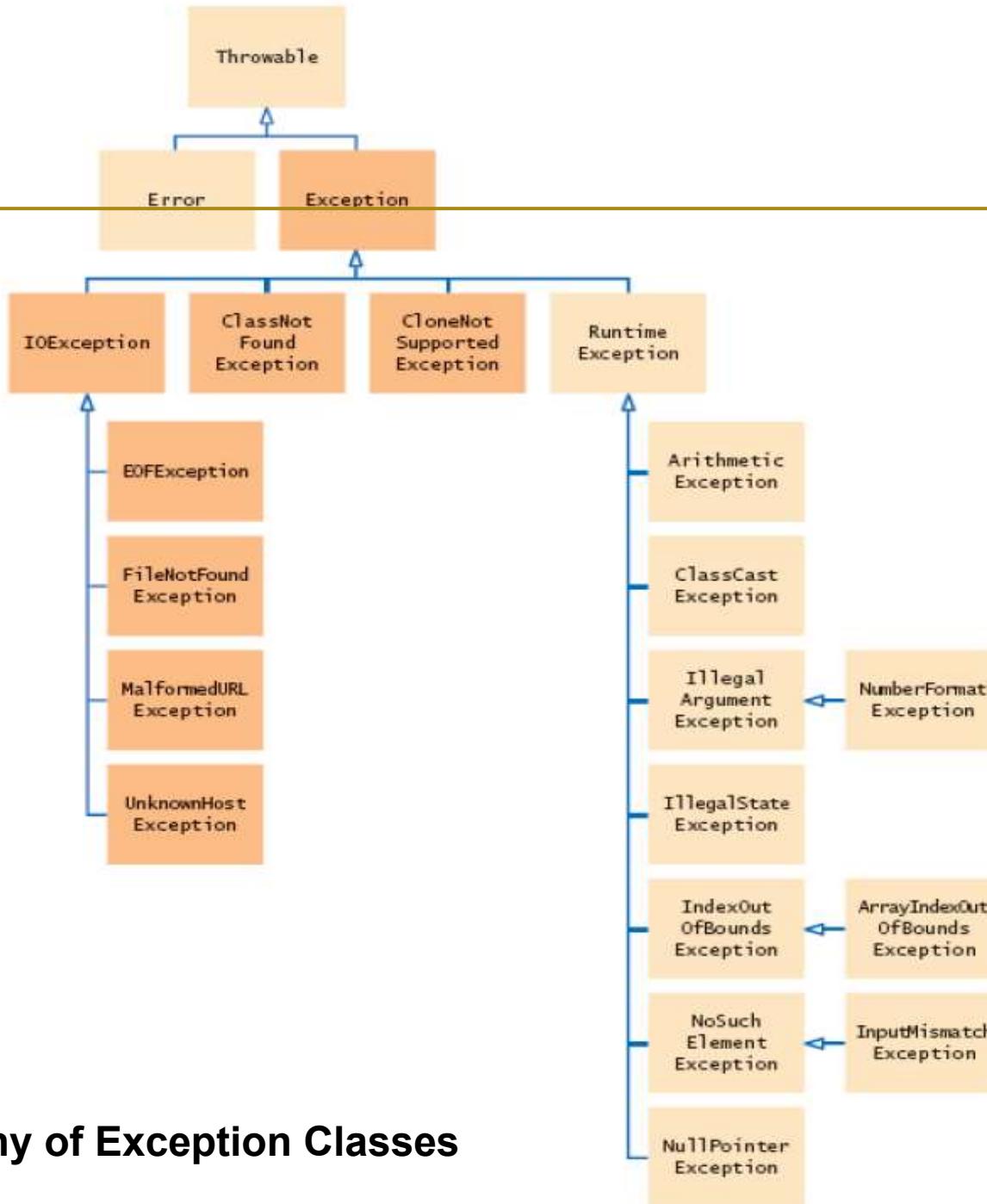
- When an exception is thrown, method terminates immediately
  - Execution continues with an exception handler



# HIERARCHY OF EXCEPTION CLASSES



# HIE



**Figure 1:**  
The Hierarchy of Exception Classes

# CHECKED AND UNCHECKED EXCEPTIONS

---

- Two types of exceptions:
  - Checked
    - The compiler checks that you don't ignore them
    - Due to external circumstances that the programmer cannot prevent
    - Majority occur when dealing with input and output
    - For example, IOException



# CHECKED AND UNCHECKED EXCEPTIONS

---

- Two types of exceptions:

- Unchecked:
  - Extend the class RuntimeException or Error
  - They are the programmer's fault
  - Examples of runtime exceptions:
    - Example of error: OutOfMemoryError

**NumberFormatException**  
**IllegalArgumentException**  
**NullPointerException**



# CHECKED AND UNCHECKED EXCEPTIONS

---

- Categories aren't perfect:
  - `Scanner.nextInt` throws `unchecked InputMismatchException`
  - Programmer cannot prevent users from entering incorrect input
  - This choice makes the class easy to use for beginning programmers
- Deal with checked exceptions principally when programming with files and streams



*Continued...*

# CHECKED AND UNCHECKED EXCEPTIONS

---

- For example, use a Scanner to read a file

```
String filename = . . .;
FileReader reader = new FileReader(filename);
Scanner in = new Scanner(reader);
```

But, FileReader constructor can throw a  
FileNotFoundException



# CHECKED AND UNCHECKED EXCEPTIONS

---

- Two choices:
  - Handle the exception
  - Tell compiler that you want method to be terminated when the exception occurs
    - Use throws specifier so method can throw a checked exception

```
public void read(String filename) throws FileNotFoundException
{
 FileReader reader = new FileReader(filename);
 Scanner in = new Scanner(reader);
 . . .
}
```



*Continued...*

# CHECKED AND UNCHECKED EXCEPTIONS

---

- For multiple exceptions:

```
public void read(String filename)
 throws IOException, ClassNotFoundException
```

- Keep in mind inheritance hierarchy:  
If method can throw an IOException and  
FileNotFoundException, only use IOException
- Better to declare exception than to handle it  
incompetently



# EXCEPTION SPECIFICATION

---

```
accessSpecifier returnType
 methodName(parameterType parameterName, . . .)
 throws ExceptionClass, ExceptionClass, . . .
```

**Example:**

```
public void read(BufferedReader in) throws IOException
```

**Purpose:**

To indicate the checked exceptions that this method can throw



## SELF CHECK

---

3. Suppose a method calls the FileReader constructor and the read method of the FileReader class, which can throw an IOException. Which throws specification should you use?
4. Why is a NullPointerException not a checked exception?



# ANSWER

---

3. The specification throws IOException is sufficient because FileNotFoundException is a subclass of IOException.
4. Because programmers should simply check for null pointers instead of trying to handle a NullPointerException.



## Methods with Description

### **public String getMessage()**

Returns a detailed message about the exception that has occurred. This message is initialized in the Throwable constructor.

### **public Throwable getCause()**

Returns the cause of the exception as represented by a Throwable object.

### **public String toString()**

Returns the name of the class concatenated with the result of getMessage()

### **public void printStackTrace()**

Prints the result of toString() along with the stack trace to System.err, the error output stream.

### **public StackTraceElement [] getStackTrace()**

Returns an array containing each element on the stack trace. The element at index 0 represents the top of the call stack, and the last element in the array represents the method at the bottom of the call stack.

### **public Throwable fillInStackTrace()**

Fills the stack trace of this Throwable object with the current stack trace, adding to any previous information in the stack trace.

# CATCHING EXCEPTIONS

---

- Install an exception handler with try/catch statement
- try block contains statements that may cause an exception
- catch clause contains handler for an exception type



*Continued...*

# CATCHING EXCEPTIONS

- Example:

```
try
{
 String filename = . . .;
 FileReader reader = new FileReader(filename);
 Scanner in = new Scanner(reader);
 String input = in.next();
 int value = Integer.parseInt(input);

 . . .
}
catch (IOException exception)
{
 exception.printStackTrace();
}
catch (NumberFormatException exception)
{
 System.out.println("Input was not a number");
}
```

# CATCHING EXCEPTIONS

---

- Statements in try block are executed
- If no exceptions occur, catch clauses are skipped
- If exception of matching type occurs, execution jumps to catch clause
- If exception of another type occurs, it is thrown until it is caught by another try block

*Continued...*



# CATCHING EXCEPTIONS

---

- catch (IOException exception) block
  - exception contains reference to the exception object that was thrown
  - catch clause can analyze object to find out more details
  - exception.printStackTrace() : printout of chain of method calls that lead to exception



# GENERAL TRY BLOCK

---

```
try
{
 statement
 statement
 . . .
}
catch (ExceptionClass exceptionObject)
{
 statement
 statement
 . . .
}
catch (ExceptionClass exceptionObject)
{
 statement
 statement
 . . .
}
. . .
```

*Continued...*



# GENERAL TRY BLOCK

## Example:

```
try
{
 System.out.println("How old are you?");
 int age = in.nextInt();
 System.out.println("Next year, you'll be " + (age + 1));
}
catch (InputMismatchException exception)
{
 exception.printStackTrace();
}
```

## Purpose:

To execute one or more statements that may generate exceptions. If an exception occurs and it matches one of the catch clauses, execute the first one that matches. If no exception occurs, or an exception is thrown that doesn't match any catch clause, then skip the catch clauses.



# MULTIPLE CATCH BLOCK

```
try {
 file = new FileInputStream(fileName); x = (byte)
 file.read();
}
catch (IOException i)
{ i.printStackTrace(); return -1;
}

catch (FileNotFoundException f) // Not valid!

{ f.printStackTrace(); return -1;
}
```

```
catch (IOException|FileNotFoundException ex)
{
 logger.log(ex);
 throw ex;
```



# THE FINALLY CLAUSE

---

- Exception terminates current method
- Danger: Can skip over essential code
  - Example:

```
reader = new FileReader(filename);
Scanner in = new Scanner(reader);
readData(in);
reader.close();
// May never get here
```



# THE FINALLY CLAUSE

---

- Must execute `reader.close()` even if exception happens
- Use `finally` clause for code that must be executed "no matter what"



# THE FINALLY CLAUSE

---

```
FileReader reader = new FileReader(filename);
try
{
 Scanner in = new Scanner(reader);
 readData(in);
}
finally
{
 reader.close(); // if an exception occurs, finally clause
 // is also executed before exception is
 // passed to its handler
}
```



# THE FINALLY CLAUSE

---

- Executed when `try` block is exited in any of three ways:
  - After last statement of `try` block
  - After last statement of `catch` clause, if this `try` block caught an exception
  - When an exception was thrown in `try` block and not caught
- Recommendation: don't mix `catch` and `finally` clauses in same `try` block



# THE FINALLY CLAUSE

---

```
try
{
 statement
 statement
 . . .
}
finally
{
 statement
 statement
 . . .
}
```

*Continued...*



# THE FINALLY CLAUSE

## Example:

```
FileReader reader = new FileReader(filename) ;
try
{
 readData(reader) ;
}
finally
{
 reader.close() ;
}
```

## Purpose:

To ensure that the statements in the finally clause are executed whether or not the statements in the try block throw an exception.



## SELF CHECK

---

7. Why was the reader variable declared outside the try block?
  - o If it had been declared inside the try block, its scope would only have extended to the end of the try block, and the catch clause could not have closed it.



# DIFF. FINALLY AND FINALIZE()

- **finally** – The finally block *always* executes when the try block exits, except System.exit(0) call. This ensures that the finally block is executed even if an unexpected exception occurs. But finally is useful for more than just exception handling — it allows the programmer to avoid having cleanup code accidentally bypassed by a return, continue, or break. Putting cleanup code in a finally block is always a good practice, even when no exceptions are anticipated.
- **finalize()** – method helps in garbage collection. A method that is invoked before an object is discarded by the garbage collector, allowing it to clean up its state. Should not be used to release non-memory resources like file handles, sockets, database connections etc because Java has only a finite number of these resources and you do not know when the garbage collection is going to kick in to release these non-memory resources through the finalize() method.

# CREATING USER DEFINED EXCEPTIONS

---

- Though Java provides an extensive set of in-built exceptions, there are cases in which we may need to define our own exceptions in order to handle the various application specific errors that we might encounter.
- While defining an user defined exception, we need to take care of the following aspects:
  1. The user defined exception class should extend from Exception class.
  2. The `toString()` method should be overridden in the user defined exception class in order to display meaningful information about the exception.



# CREATING USER DEFINED EXCEPTIONS

- **NegativeAgeException.java**

```
public class NegativeAgeException extends Exception
{
 private int age;
 public NegativeAgeException(int age)
 {
 this.age = age;
 }
 public String toString(){
 return "Age cannot be negative" + " " +age ;
 }
}
```



*Continued...*

# CREATING USER DEFINED EXCEPTIONS

- **CustomExceptionTest.java**

```
public class CustomExceptionTest
{
 public static void main(String[] args) throws Exception{
 int age = getAge();
 if (age < 0){
 throw new NegativeAgeException(age);
 }
 else{
 System.out.println("Age entered is " + age);
 }
 }
 static int getAge(){
 return -10;
 }
}
```

*Continued...*

# CREATING USER DEFINED EXCEPTIONS

## o output

Exception in thread "main" Age cannot be negative -10

at tips.basics.exception.CustomExceptionTest.main(CustomExceptionTest.java)



*Continued...*

# THROW AND THROWS

- The keyword **throw** is used to throw user defined exceptions and it requires a single argument(a throwable class object)

ex: `throw new XYZException("Test");`

- The keyword **throws** is used in method signatures to declare that, this method could possibly throw an exception.

ex: `public void test() throws SQLException {  
}`



# Files and I/O

# Introduction

- The `java.io` package, which provides support for I/O operations.
- A stream can be defined as a sequence of data.
- The `InputStream` is used to read data from a source.
- The `OutputStream` is used for writing data to a destination.

# The Java I/O Classes

|                       |                             |                        |
|-----------------------|-----------------------------|------------------------|
| BufferedInputStream   | FileWriter                  | PipedInputStream       |
| BufferedOutputStream  | FilterInputStream           | PipedOutputStream      |
| BufferedReader        | FilterOutputStream          | PipedReader            |
| BufferedWriter        | FilterReader                | PipedWriter            |
| ByteArrayInputStream  | FilterWriter                | PrintStream            |
| ByteArrayOutputStream | InputStream                 | PrintWriter            |
| CharArrayReader       | InputStreamReader           | PushbackInputStream    |
| CharArrayWriter       | LineNumberReader            | PushbackReader         |
| DataInputStream       | ObjectInputStream           | RandomAccessFile       |
| DataOutputStream      | ObjectInputStream.GetField  | Reader                 |
| File                  | ObjectOutputStream          | SequenceInputStream    |
| FileDescriptor        | ObjectOutputStream.PutField | SerializablePermission |
| FileInputStream       | ObjectStreamClass           | StreamTokenizer        |
| FileOutputStream      | ObjectStreamField           | StringReader           |
| FilePermission        | OutputStream                | StringWriter           |
| FileReader            | OutputStreamWriter          | Writer                 |

# Interfaces

DataInput

DataOutput

Externalizable

FileFilter

FilenameFilter

ObjectInput

ObjectInputValidation

ObjectOutput

ObjectStreamConstants

Serializable

# File

- 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.

## Constructors

- `File(String directoryPath)`
- `File(String directoryPath, String filename)`
- `File(File dirObj, String filename)`
- `File(URI uriObj)`

# Directories

- A directory is a File that contains a list of other files and directories.
  - `isDirectory()`
  - `String[ ] list()`

# Using FilenameFilter

- To limit the number of files returned by the `list( )` method to include only those files that match a certain filename pattern, or *filter*.
  - `String[ ] list(FilenameFilter FFObj)`
- `FilenameFilter` method
  - `boolean accept(File directory, String filename)`

# The listFiles( ) Alternative

Java 2 added listFiles():

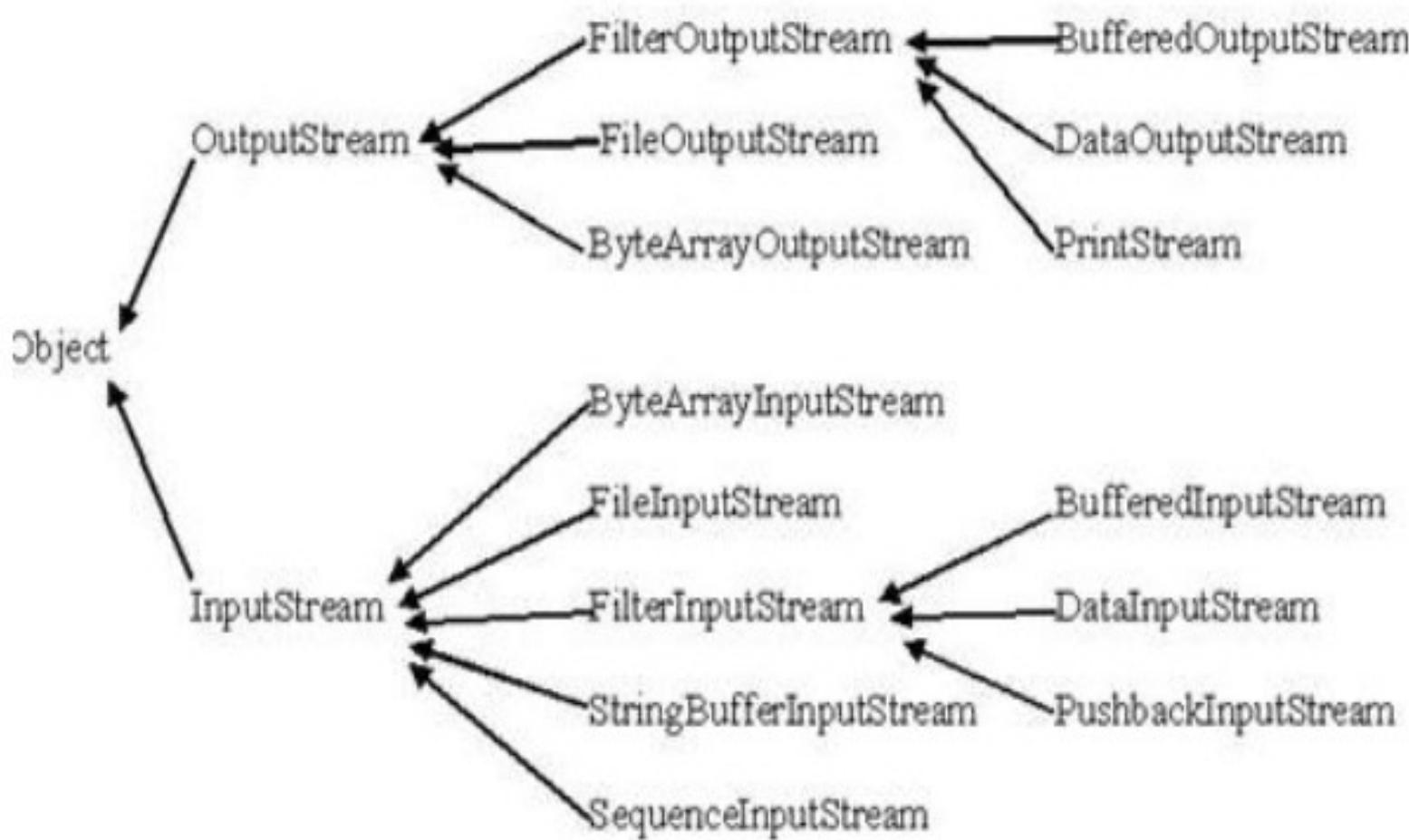
- File[ ] listFiles( )
- File[ ] listFiles(FilenameFilter *FFObj*)
  - Returns File array instead of string array, same as list
- File[ ] listFiles(FileFilter *FObj*)
  - returns those files with path names that satisfy the specified FileFilter.
  - FileFilter method
    - boolean accept(File *path*)

# Creating Directories

# The Stream Classes

- Java's stream-based I/O is built upon four abstract classes: InputStream, OutputStream, Reader, and Writer.
- InputStream and OutputStream are designed for byte streams.
  - Byte streams are used to perform input and output of 8-bit bytes.
  - E.g. works with bytes or other binary objects.
- Reader and Writer are designed for character streams.
  - Character streams are used to perform input and output for 16-bit unicode.
  - Internally FileReader uses FileInputStream and FileWriter uses FileOutputStream.
  - Reads and writes two bytes at a time.
  - E.g. works with characters or strings.

# The Byte Streams



# InputStream

- All of the methods in this class will throw an IOException on error conditions.

| Method                                                                     | Description                                                                                                                                                                                                  |
|----------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 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.                                                                                                                                 |
| void mark(int <i>numBytes</i> )                                            | Places a mark at the current point in the input stream that will remain valid until <i>numBytes</i> bytes are read.                                                                                          |
| boolean markSupported( )                                                   | Returns true if mark( )/reset( ) are supported by the invoking stream.                                                                                                                                       |
| 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 <i>buffer</i> [ ])                                           | Attempts to read up to <i>buffer.length</i> bytes into <i>buffer</i> and returns the actual number of bytes that were successfully read. -1 is returned when the end of the file is encountered.             |
| int read(byte <i>buffer</i> [ ], int <i>offset</i> , int <i>numBytes</i> ) | Attempts to read up to <i>numBytes</i> bytes into <i>buffer</i> starting at <i>buffer[offset]</i> , returning the number of bytes successfully read. -1 is returned when the end of the file is encountered. |

# OutputStream

| Method                                                            | Description                                                                                                                                                                                         |
|-------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>void close()</code>                                         | Closes the output stream. Further write attempts will generate an <code>IOException</code> .                                                                                                        |
| <code>void flush()</code>                                         | Finalizes the output state so that any buffers are cleared. That is, it flushes the output buffers.                                                                                                 |
| <code>void write(int b)</code>                                    | Writes a single byte to an output stream. Note that the parameter is an int, which allows you to call <code>write()</code> with expressions without having to cast them back to <code>byte</code> . |
| <code>void write(byte buffer[ ])</code>                           | Writes a complete array of bytes to an output stream.                                                                                                                                               |
| <code>void write(byte buffer[ ], int offset, int numBytes)</code> | Writes a subrange of <code>numBytes</code> bytes from the array <code>buffer</code> , beginning at <code>buffer[offset]</code> .                                                                    |

# FileInputStream

- The FileInputStream class creates an InputStream that you can use to read bytes from a file.
  - `FileInputStream(String filepath)`
  - `FileInputStream(File fileObj)`
    - throw a FileNotFoundException

# FileOutputStream

- FileOutputStream creates an OutputStream that you can use to write bytes to a file.
  - `FileOutputStream(String filePath)`
  - `FileOutputStream(File fileObj)`
  - `FileOutputStream(String filePath, boolean append)`
  - `FileOutputStream(File fileObj, boolean append)`
    - throw a FileNotFoundException or a SecurityException

# ByteArrayInputStream

- Input stream that uses a byte array as the source.
  - `ByteArrayInputStream(byte array[ ])`
  - `ByteArrayInputStream(byte array[ ], int start, int numBytes)`

# ByteArrayOutputStream

- An output stream that uses a byte array as the destination.
  - `ByteArrayOutputStream( )`
  - `ByteArrayOutputStream(int numBytes)`

# Filtered Byte Streams

- Filter streams are also used to manipulate data reading from an underlying stream.
  - `FilterOutputStream(OutputStream os)`
  - `FilterInputStream(InputStream is)`
- The methods provided in these classes are identical to those in `InputStream` and `OutputStream`.
- It allows the user to make a chain using multiple input stream so that, the operations that are to be applied on this chain, may create a combine effects on several filters.
- By using these streams, there is no need to convert the data from **byte** to **char** while writing to a file.

# Buffered Byte Streams

- A buffered stream extends a filtered stream class by attaching a memory buffer to the I/O streams.
- This buffer allows Java to do I/O operations on more than a byte at a time, hence increasing performance.
- The buffered byte stream classes are **BufferedInputStream** and **BufferedOutputStream**.

# BufferedInputStream

- Allows you to “wrap” any InputStream into a buffered stream and achieve performance improvement.
  - `BufferedInputStream(InputStream inputStream)`
  - `BufferedInputStream(InputStream inputStream, int bufferSize)`

# BufferedOutputStream

- `BufferedOutputStream(OutputStream outputStream)`
  - //buffer of 512 bytes
- `BufferedOutputStream(OutputStream outputStream, int bufferSize)`

# DataInputStream

- The `Java.io.DataInputStream` class lets an application read primitive Java data types from an underlying input stream in a machine-independent way.
  - `DataInputStream(InputStream in)`
  - Similarly; `DataOutputStream(OutputStream out)`

# PrintStream

- The `PrintStream` class provides all of the formatting capabilities we have been using from the `System` file handle, `System.out`.

# The Character Streams

## Reader

- Reader is an abstract class that defines streaming character input.
- Throws IOException

## Writer

- Writer is an abstract class that defines streaming character output.
- Throws IOException

## FileReader

- The FileReader class creates a Reader that read the contents of a file.
- `FileReader(String filePath)`
- `FileReader(File fileObj)`
  - throws FileNotFoundException

# Con't

## FileWriter

- The `FileWriter` class creates a `Writer` that writes the contents to a file.
- `FileWriter(String filePath)`
- `FileWriter(String filePath, boolean append)`
- `FileWriter(File fileObj)`
- `FileWriter(File fileObj, boolean append)`

## BufferedReader

- `BufferedReader` improves performance by buffering input.
  - `BufferedReader(Reader inputStream)`
  - `BufferedReader(Reader inputStream, int bufSize)`

# Con't

## BufferedWriter

- `BufferedWriter(Writer outputStream)`
- `BufferedWriter(Writer outputStream, int bufSize)`

# StreamTokenizer

wc( ) Using a StreamTokenizer

- WordCount.java

# StringTokenizer

# ZipOutputStream

`ZipOutputStream(OutputStream out)`

- `close()`
  - closes the ZIP output stream as well as the stream being filtered.
- `void closeEntry()`
  - closes the current ZIP entry and positions the stream for writing the next entry.
- `void finish()`
  - Finishes writing the contents of the ZIP output stream without closing the underlying stream.
- `void putNextEntry(ZipEntry e)`
  - Begins writing a new ZIP file entry and positions the stream to the start of the entry data.
- `void setComment(String comment)`
  - Sets the ZIP file comment.

# Con't

- void setLevel(int level)
  - Sets the compression level for subsequent entries which are DEFLATED.
- void setMethod(int method)
  - Sets the default compression method for subsequent entries.
- void write(byte[] b, int off, int len)
  - Writes an array of bytes to the current ZIP entry data.

# What is jar files? How to create it?

- JAR : It is a java archive files used to package classes, files etc. as single file.
  - This is similar to zip file in windows.
- To create a jar file with all class files under the current directory.
  - `jar -cvf jarfilename.jar *.class`
  - c - to *create* a JAR file.
  - f - the output goes to a *file* rather than to stdout.
  - v -Produces *verbose* output on stdout while the JAR file is being built.
    - The verbose output tells you the name of each file as it's added to the JAR file.
- Can jar code be retrieved from byte code?
  - Yes, class files can be decompiled using utilities like JAD(JAvA Decompiler). This takes the class files as an input and generates a java file.

# Serialization

- Java provides a mechanism, called object serialization where an object can be represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object.
- After a serialized object has been written into a file, it can be read from the file and deserialized that is, the type information and bytes that represent the object and its data can be used to recreate the object in memory.
- The entire process is JVM independent.
  - An object can be serialized on one platform and deserialized on an entirely different platform.

# Con't

## ObjectOutputStream

- public final void writeObject(Object x) throws IOException

## ObjectInputStream

- public final Object readObject() throws IOException,  
ClassNotFoundException.
- For a class to be serialized successfully two conditions:
  - The class must implement the java.io.Serializable interface.
  - All of the fields in the class must be serializable. If a field is not serializable, it must be marked transient.

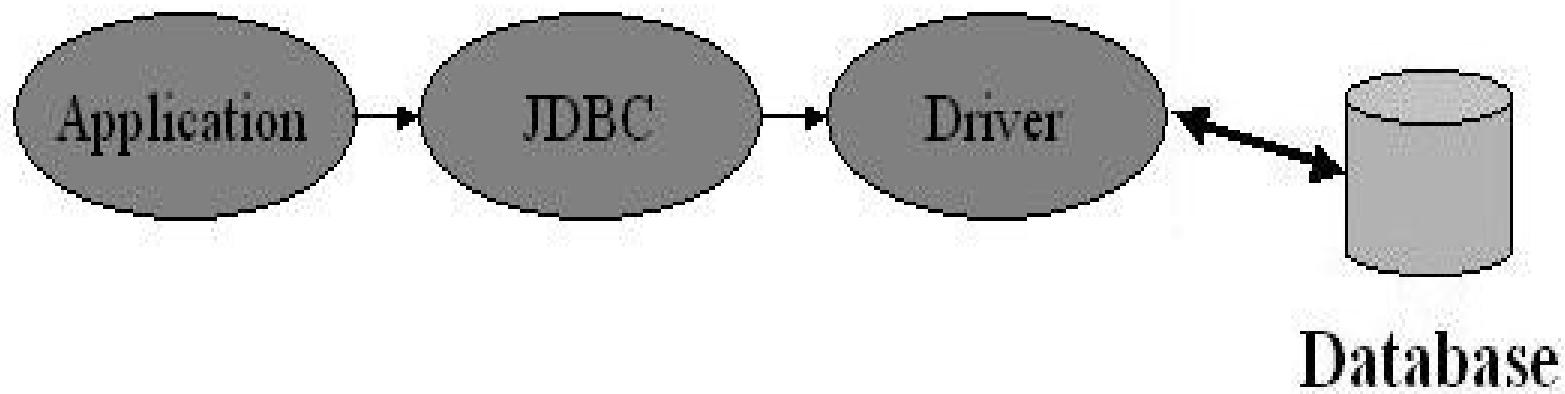
# JDBC

# Introduction

- JDBC stands for **Java Database Connectivity**.
- It is a standard Java API for database-independent connectivity between the Java programming language and a wide range of databases.
- It was developed by JavaSoft, a subsidiary of Sun Microsystems.
- The JDBC library includes APIs for each of the tasks:
  - Making a connection to a database
  - Creating SQL or MySQL statements
  - Executing that SQL or MySQL queries in the database
  - Viewing & Modifying the resulting records.
- JDBC packages **java.sql** and **javax.sql**.

# JDBC Architecture

- General JDBC Architecture consists of two layers:
  - **JDBC API:** This provides the application-to-JDBC Manager connection.
  - **JDBC Driver API:** This supports the JDBC Manager-to-Driver Connection.
- The JDBC API uses a driver manager and database-specific drivers to provide transparent connectivity to heterogeneous databases.



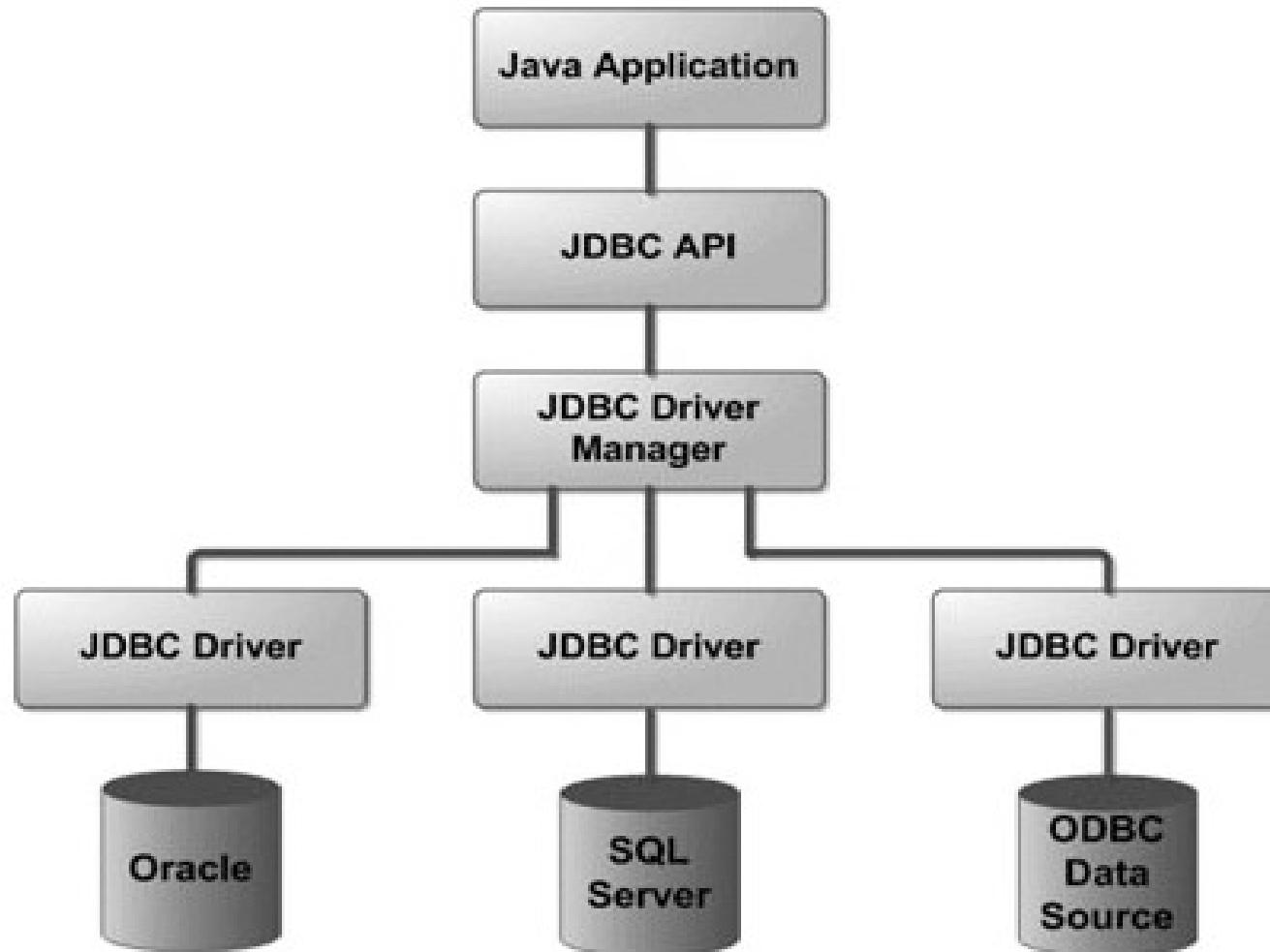
# JDBC Architecture

- JDBC is an API specification developed by *Sun Microsystems* that defines a uniform interface for accessing various relational databases
- JDBC is a core part of the Java platform and is included in the standard JDK distribution
- The primary function of the JDBC API is to provide a means for the developer to issue SQL statements and process the results in a consistent, database-independent manner

# Cont..

- JDBC provides rich, object-oriented access to databases by defining classes and interfaces that represent objects such as:
  1. Database connections
  2. SQL statements
  3. Result Set
  4. Database metadata
  5. Prepared statements
  6. Binary Large Objects (BLOBs)
  7. Character Large Objects (CLOBs)
  8. Callable statements
  9. Database drivers
  10. Driver manager

# Architectural Diagram



# Common JDBC Components

## DriverManager

- This class manages a list of database drivers.
- Matches connection requests from the java application with the proper database.

## Driver

- This interface which will connect java application and database system.
- Database – MySQL, MS-access, Oracle, DB2, Sybase
- Different drivers according to different database.
- Driver manager handles drivers.

# Con't

## Connection

- This interface with all methods for contacting a database.
- All communication with database is through connection object.

## Statement

- This interface is use to submit the SQL statements to the database.

## ResultSet

- These objects hold data retrieved from a database after you execute an SQL query using Statement objects. It acts as an iterator to allow you to move through its data.

## SQLException

- This class handles any errors that occur in a database application.

# JDBC Drivers

- A **JDBC driver** translates standard *JDBC* calls into a network or database protocol or into a database library API call that facilitates communication with the database
- This translation layer provides JDBC applications with database independence
- If the back-end database changes, only the JDBC driver need be replaced with few code modifications required.
- There are four distinct types of JDBC drivers.

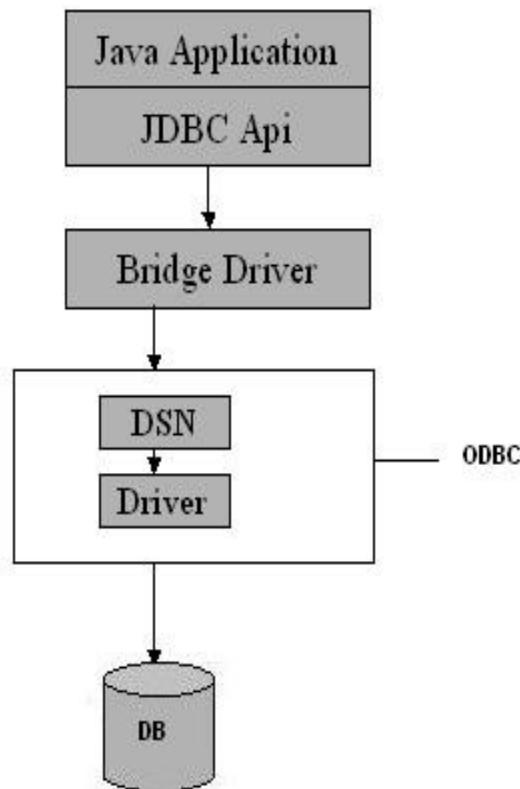
# Types of Drivers

Four Types:

- **Type 1:** JDBC-ODBC Bridge driver (Bridge)
- Type 2:** Native-API/partly Java driver (Native)
- Type 3:** AllJava/Net-protocol driver (Middleware)
- Type 4:** All Java/Native-protocol driver (Pure)

# Type 1 JDBC-ODBC Bridge driver

- The Type 1 driver translates all JDBC calls into ODBC calls and sends them to the ODBC driver.
- The JDBC-ODBC Bridge driver is recommended only for experimental use and is typically used for development and testing purposes only.



# Con't

## Advantage

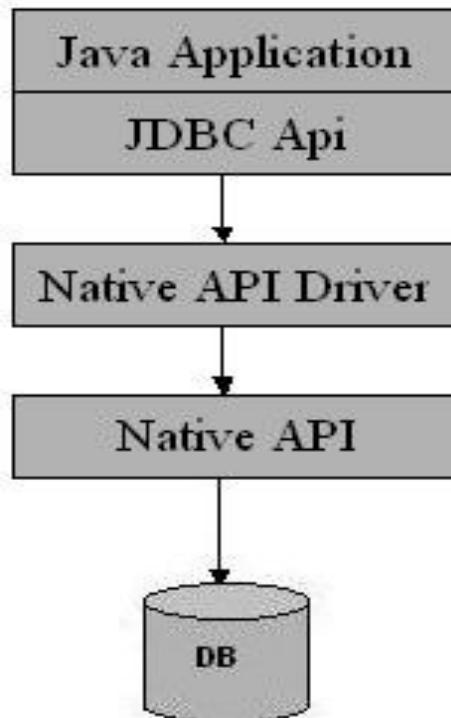
- The JDBC-ODBC Bridge allows access to almost any database, since the database's ODBC drivers are already available.

## Disadvantages

- Since the Bridge driver is not written fully in Java, Type 1 drivers are not portable.
- A performance issue is seen as a JDBC call goes through the bridge to the ODBC driver, then to the database, and this applies even in the reverse process.
- They are the slowest of all driver types.
- The client system requires the ODBC Installation to use the driver.
- Not good for the Web.
-

# Type 2 Native-API/partly Java driver

- Type 2 drivers convert JDBC calls into database-specific calls.
  - i.e. this driver is specific to a particular database.
  - E.g. : Oracle will have oracle native api.



# Con't

## Advantage

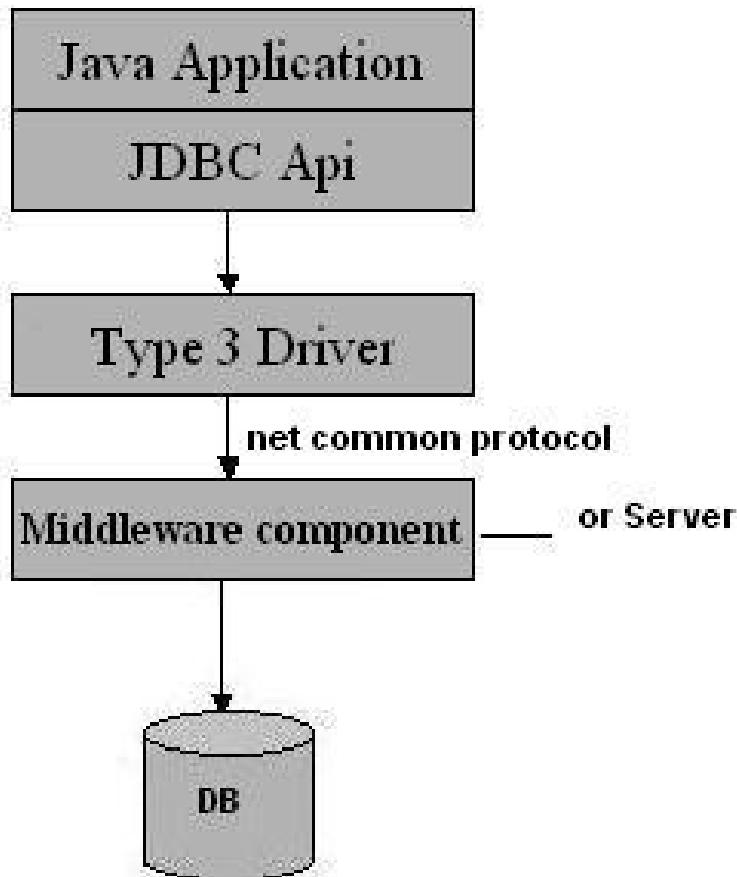
- They typically offer better performance than the JDBC-ODBC Bridge as the layers of communication (tiers) are less than that of Type1 and also it uses Native api which is Database specific.

## Disadvantage

- Native API must be installed in the Client System and hence type 2 drivers cannot be used for the Internet.
- Like Type 1 drivers, it's not written in Java Language which forms a portability issue.
- If we change the Database we have to change the native api as it is specific to a database.
- Mostly obsolete now.
- Usually not thread safe.

# Type 3 All Java/Net-protocol driver

- Type 3 database requests are passed through the network to the middle-tier server.
- The middle-tier then translates the request to the database. If the middle-tier server can in turn use Type1, Type 2 or Type 4 drivers.



# Con't

## Advantage

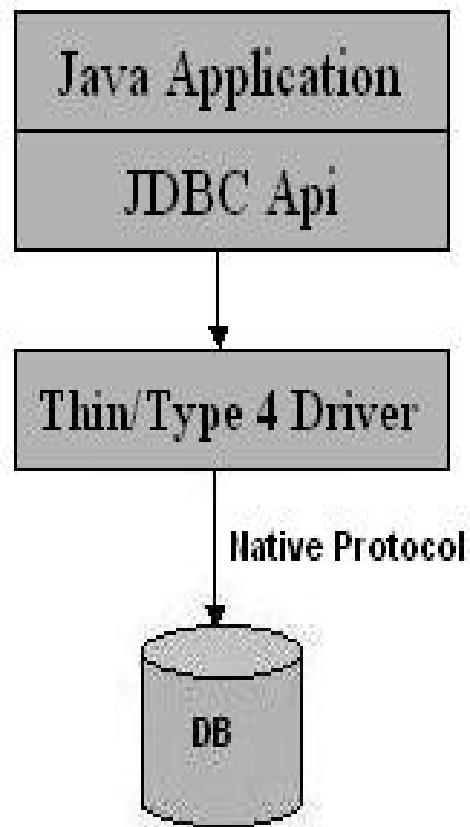
- This driver is server-based, so there is no need for any vendor database library to be present on client machines.
- This driver is fully written in Java and hence Portable. It is suitable for the web.
- There are many opportunities to optimize portability, performance, and scalability.
- The net protocol can be designed to make the client JDBC driver very small and fast to load.
- This driver is very flexible allows access to multiple databases using one driver.
- They are the most efficient amongst all driver types.

## Disadvantage

- It requires another server application to install and maintain.
- Traversing the recordset may take longer, since the data comes through the backend server.

# Type 4 Native-protocol/all-Java driver

- The Type 4 uses java networking libraries to communicate directly with the database server.



# Con't

## Advantage

- They are completely written in Java to achieve platform independence.
- It is most suitable for the web.
- Number of translation layers is very less
- i.e. type 4 JDBC drivers don't have to translate database requests to ODBC or a native connectivity interface or to pass the request on to another server, performance is typically quite good.
- You don't need to install special software on the client or server. Further, these drivers can be downloaded dynamically.

## Disadvantage

- With type 4 drivers, the user needs a different driver for each database.

# Which Driver should be used?

- If you are accessing one type of database, such as Oracle, Sybase, or IBM, the preferred driver type is 4.
- If your Java application is accessing multiple types of databases at the same time, type 3 is the preferred driver.
- Type 2 drivers are useful in situations where a type 3 or type 4 driver is not available.
- The type 1 is typically used for development and testing purposes only.

# 1. Loading a database driver

```
try {
 // Class.forName (String ClassName)
 Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
 //Or any other driver
}catch(Exception x){
 System.out.println ("Unable to load the driver class!");
}
```

## 2. Creating a oracle jdbc Connection

```
try{
 Connection dbConnection =
 DriverManager.getConnection(url,"loginName","Password");
} catch(SQLException x){
 System.out.println ("Couldn't get connection!");
}
```

- DriverManager class manages the JDBC drivers that are installed on the system.
- Its getConnection () method is used to establish a connection to a database.
- It uses a username, password, and a jdbc url to establish a connection to the database and returns a connection object.
- A jdbc Connection represents a session/connection with a specific database.

# Con't

- An application can have one or more connections with a single database, or it can have many connections with different databases.
- A Connection object provides metadata i.e. information about the database, tables, and fields.

| RDBMS  | JDBC driver name                | URL format                                          |
|--------|---------------------------------|-----------------------------------------------------|
| MySQL  | com.mysql.jdbc.Driver           | jdbc:mysql://hostname/ databaseName                 |
| ORACLE | oracle.jdbc.driver.OracleDriver | jdbc:oracle:thin:@hostname:port Number:databaseName |
| DB2    | COM.ibm.db2.jdbc.net.DB2Driver  | jdbc:db2:hostname:port Number/databaseName          |
| Sybase | com.sybase.jdbc.SybDriver       | jdbc:sybase:Tds:hostname: port Number/databaseName  |

### 3. Creating a jdbc Statement object

Three kinds of Statements:

- **Statement** - Execute simple sql queries without parameters.
  - Useful when you are using static SQL statements at runtime.  
`Statement createStatement ()`
  - `Statement statement = dbConnection.createStatement ();`
- **Prepared Statement** - Execute precompiled sql queries with or without parameters.
  - It can accept input parameters at runtime.  
`PreparedStatement prepareStatement (String sql)`
  - `PreparedStatement` objects are precompiled SQL statements.
- **Callable Statement** - Execute a call to a database stored procedure.
  - It can accept runtime input parameters.  
`CallableStatement prepareCall (String sql)`
  - `CallableStatement` objects are SQL stored procedure call statements.

# 4. Executing a SQL statement

Three methods for executing statements:

- All 3 statement can return ResultSet Object.
- ResultSet executeQuery(String SQL) - For a SELECT statement
- int executeUpdate(String SQL) - For statements that create or modify tables(Insert, Update, Delete, Alter, Drop)
- boolean execute(String SQL) - executes an SQL statement that is written as String object.
  - Returns a boolean value of true if a ResultSet object can be retrieved; otherwise, it returns false.
  - Use this method to execute SQL DDL statements or when you need to use truly dynamic SQL.

# ResultSet

- ResultSet provides access to a table of data generated by executing a Statement.
- The table rows are retrieved in sequence.
- A ResultSet maintains a cursor pointing to its current row of data.
- The next () method is used to successively step through the rows of the tabular results.
- By default, only one ResultSet object per Statement object can be open at the same time.

# Methods of ResultSet

- All method throws SQLException

## Navigating a Result Set:

- void beforeFirst()
- void afterLast()
- boolean first()
- void last()
- boolean previous()
- boolean next()
- int getRow()

## Viewing a Result Set:

- int getInt(String columnName)
- int getInt(int columnIndex)

# Con't

## Updating a Result Set

- void updateString(int columnIndex, String s)
- void updateString(String columnName, String s)
- void updateRow()
- void deleteRow()
- void insertRow()

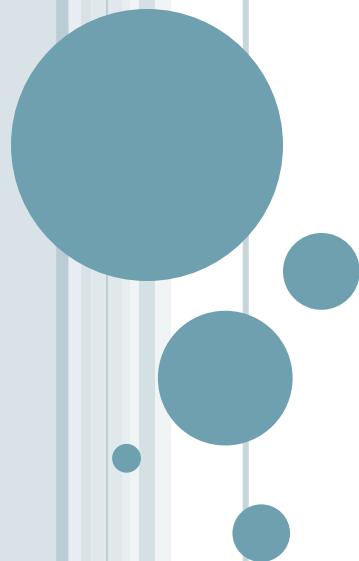
# DataTypes

| SQL         | JDBC/Java            | setXXX        | updateXXX        |
|-------------|----------------------|---------------|------------------|
| VARCHAR     | java.lang.String     | setString     | updateString     |
| CHAR        | java.lang.String     | setString     | updateString     |
| LONGVARCHAR | java.lang.String     | setString     | updateString     |
| BIT         | boolean              | setBoolean    | updateBoolean    |
| NUMERIC     | java.math.BigDecimal | setBigDecimal | updateBigDecimal |
| TINYINT     | byte                 | setByte       | updateByte       |
| SMALLINT    | short                | setShort      | updateShort      |
| INTEGER     | int                  | setInt        | updateInt        |
| BIGINT      | long                 | setLong       | updateLong       |
| REAL        | float                | setFloat      | updateFloat      |
| FLOAT       | float                | setFloat      | updateFloat      |
| DOUBLE      | double               | setDouble     | updateDouble     |
| VARBINARY   | byte[ ]              | getBytes      | updateBytes      |
| BINARY      | byte[ ]              | getBytes      | updateBytes      |
| DATE        | java.sql.Date        | setDate       | updateDate       |
| TIME        | java.sql.Time        | setTime       | updateTime       |
| TIMESTAMP   | java.sql.Timestamp   | setTimestamp  | updateTimestamp  |
| CLOB        | java.sql.Clob        | setClob       | updateClob       |
| BLOB        | java.sql.Blob        | setBlob       | updateBlob       |
| ARRAY       | java.sql.Array       | setARRAY      | updateARRAY      |
| REF         | java.sql.Ref         | SetRef        | updateRef        |
| STRUCT      | java.sql.Struct      | SetStruct     | updateStruct     |

# Con't

- Use to call the setXXX() method of the PreparedStatement or CallableStatement object or the ResultSet.updateXXX() method.
- ResultSet object provides corresponding getXXX() method for each data type to retrieve column value.
  - Each method can be used with column name or by its ordinal position.

# LAMBDA EXPRESSION



# LAMBDA EXPRESSIONS

- Lambda expressions are introduced in Java 8 and are touted to be the biggest feature of Java 8.
- Lambda expression facilitates functional programming, and simplifies the development a lot.
- A lambda expression is characterized by the following syntax.

**parameter -> expression body**



# LAMBDA EXPRESSIONS

- **Important characteristics**
- **Optional type declaration** – No need to declare the type of a parameter. The compiler can inference the same from the value of the parameter.
- **Optional parenthesis around parameter** – No need to declare a single parameter in parenthesis. For multiple parameters, parentheses are required.
- **Optional curly braces** – No need to use curly braces in expression body if the body contains a single statement.
- **Optional return keyword** – The compiler automatically returns the value if the body has a single expression to return the value. Curly braces are required to indicate that expression returns a value.

# JAVA LAMBDA EXPRESSION SYNTAX

(argument-list) -> {body}

Java lambda expression is consisted of three components.

**1) Argument-list:** It can be empty or non-empty as well.

**2) Arrow-token:** It is used to link arguments-list and body of expression.

**3) Body:** It contains expressions and statements for lambda expression.

## No Parameter Syntax

```
() -> {
//Body of no parameter lambda
}
```

## One Parameter Syntax

```
(p1) -> {
//Body of single parameter lambda
}
```

## Two Parameter Syntax

```
(p1,p2) -> {
//Body of multiple parameter lambda
}
```



# LAMBDA EXPRESSIONS

- **LambdaDemo.java**
- Lambda expressions are used primarily to define inline implementation of a functional interface, i.e., an interface with a single method only.
- In the above example, we've used various types of lambda expressions to define the operation method of MathOperation interface.
- Then we have defined the implementation of sayMessage of GreetingService.
- Lambda expression eliminates the need of anonymous class and gives a very simple yet powerful functional programming capability to Java.

# LAMBDA EXPRESSIONS

- **Scope**
- Using lambda expression, you can refer to any final variable or effectively final variable (which is assigned only once).
- Lambda expression throws a compilation error, if a variable is assigned a value the second time.



# LAMBDA EXPRESSIONS

```
public class Java8Tester
{
 final static String salutation = "Hello! ";
 public static void main(String args[])
 {
 GreetingService greetService1 =
message ->System.out.println(salutation+ message);

 greetService1.sayMessage("Mahesh");
 }
 interface GreetingService
 {
 void sayMessage(String message);
 }
}
```

It should produce the following output –

Hello! Mahesh



# METHOD REFERENCES

- Method references help to point to methods by their names.
- A method reference is described using ":" symbol.
- A method reference can be used to point the following types of methods –
  - Static methods
  - Instance methods
  - Constructors using new operator (TreeSet::new)

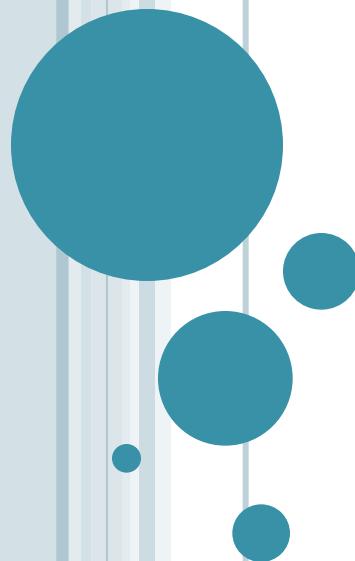


# METHOD REFERENCES

```
public class Java8Tester
{
 public static void main(String args[])
 {
 List names = new ArrayList();
 names.add("Mahesh");
 names.add("Suresh");
 names.add("Ramesh");
 names.add("Naresh");
 names.add("Kalpesh");

 names.forEach(System.out::println);
 }
}
```

# MULTITHREADING



# AGENDA

- Process based and Thread based multitasking : A brief comparison
- Thread basics
- Thread states
- Thread properties
- Thread Priorities
- Thread Synchronization
- Thread & Swing (To be covered later)

# MULTITASKING

- Multitasking is performing two or more tasks at the same time. Nearly all operating systems are capable of multitasking by using one of two multitasking techniques:
  - Process-based multitasking
  - Thread-based multitasking
- A good way to remember the difference between process-based multitasking and thread-based multitasking is to think of process-based as working with multiple programs and thread-based as working with parts of one program

## CONT..

- Process-based multitasking is running two programs concurrently. Programmers refer to a program as a *process*. Therefore, you could say that process-based multitasking is program-based multitasking
- Thread-based multitasking is having a program perform two tasks at the same time. For example, a word processing program can check the spelling of words in a document while you write the document. This is thread-based multitasking

## CONT..

- The objective of multitasking is to utilize the idle time of the CPU. Think of the CPU as the engine of your car
- Your engine keeps running regardless of whether the car is moving. Your objective is to keep your car moving as much as possible so you can get the most miles from a gallon of gas. An idling engine wastes gas
- Programs that run in a networked environment, such as those that process transactions from many computers, need to make a CPU's idle time productive

# OVERHEAD

- The operating system must do extra work to manage multitasking. Programmers call this extra work *overhead* because resources inside your computer are used to manage the multitasking operation rather than being used by programs for processing instructions and data

## CONT..

- Process-based multitasking has a larger overhead than thread-based multitasking. In process-based multitasking, each process requires its own address space in memory
- The operating system requires a significant amount of CPU time to switch from one process to another process. Programmers call this *context switching*, where each process (program) is a context. Additional resources are needed for each process to communicate with each other

## CONT..

- In comparison, the threads in thread-based multitasking share the same address space in memory because they share the same program
- This also has an impact on context switching, because switching from one part of the program to another happens within the same address space in memory
- Likewise, communication among parts of the program happens within the same memory location

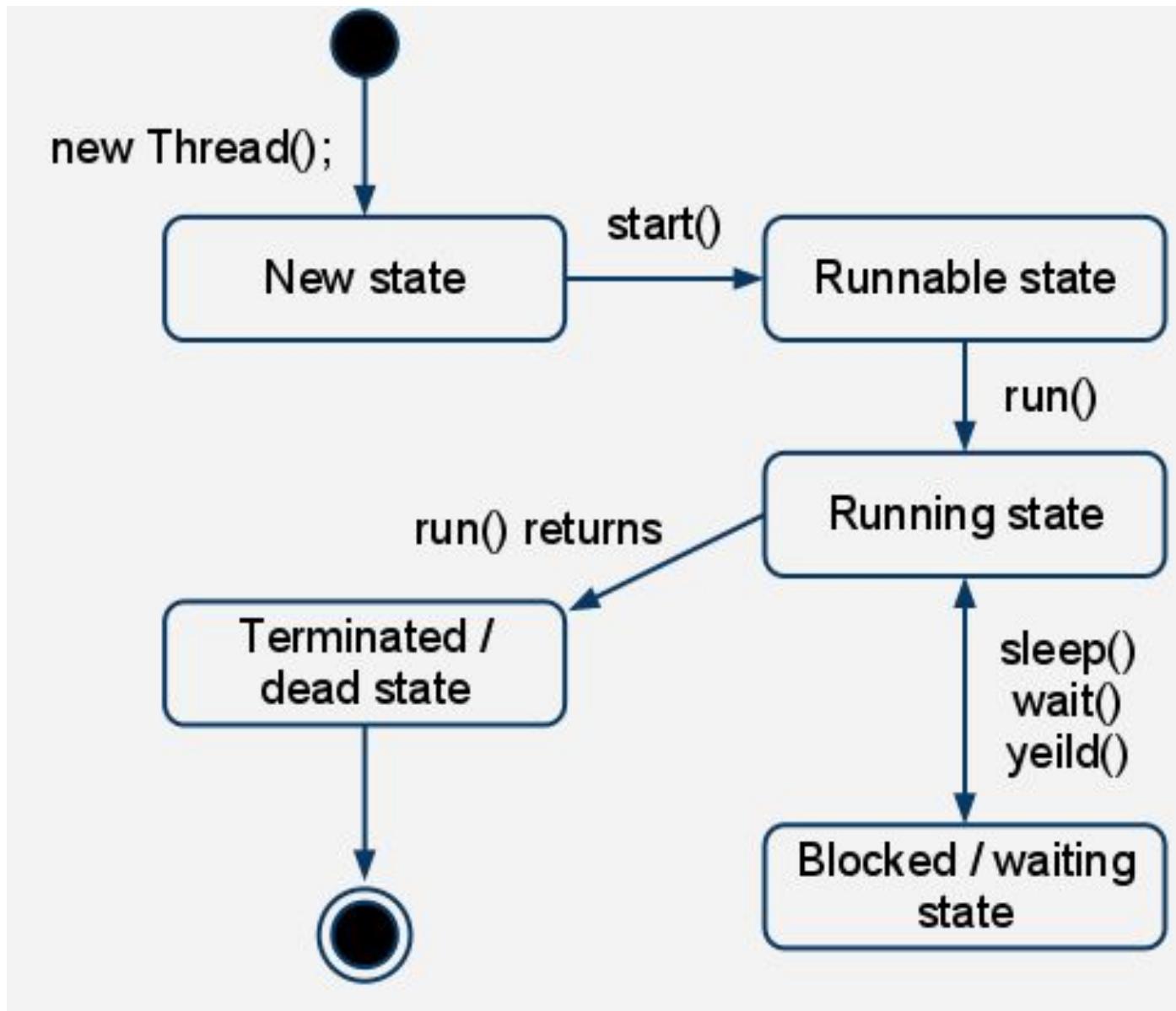
# THREADS

- A thread is part of a program that is running
  - Thread-based multitasking has multiple threads running at the same time (that is, multiple parts of a program running concurrently). Each thread is a different path of execution
- The Java run-time environment manages threads, unlike in process-based multitasking where the operating system manages switching between programs
- Threads are processed asynchronously. This means that one thread can pause while other threads continue to process

# LIFE CYCLE OF A THREAD

- A thread goes through various stages in its life cycle. For example, a thread is born, started, runs, and then dies. Following diagram shows complete life cycle of a thread

## CONT..



## CONT..

- Above mentioned stages are explained here:
- **New:** A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread.
- **Runnable:** After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.
- **Waiting:** Sometimes a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.

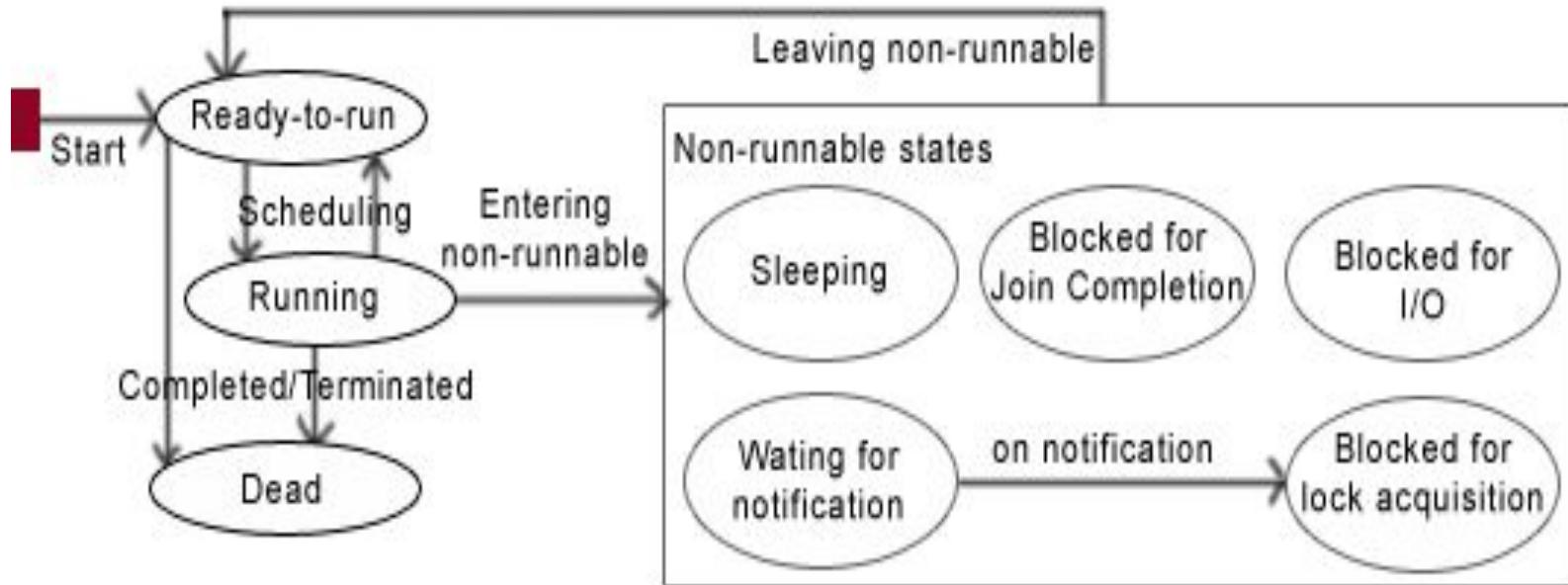
## CONT..

- **Timed waiting:** A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.
- **Terminated:** A runnable thread enters the terminated state when it completes its task or otherwise terminates.

# ADVANTAGES AND DISADVANTAGES

- Reduces the computation time
- Improves performance of an application
- Threads share the same address space so it saves the memory
- Context switching between threads is usually less expensive than between processes
- Cost of communication between threads is relatively low

# DIFFERENT STATES IMPLEMENTING MULTIPLE THREADS ARE:



## CONT..

- As we have seen different states that may be occur with the single thread
- A running thread can enter to any non Runnable state, depending on the circumstances. A thread cannot enter directly to the running state from non Runnable state; firstly it goes to runnable state
- Now let's understand the some non Runnable states which may be occur handling the multithreads

## CONT..

- **Sleeping** – On this state, the thread is still alive but it is not runnable, it might be return to runnable state later, if a particular event occurs. On this state a thread sleeps for a specified amount of time. You can use the method **sleep()** to stop the running state of a thread
  - **static void sleep(long millisecond) throws InterruptedException**

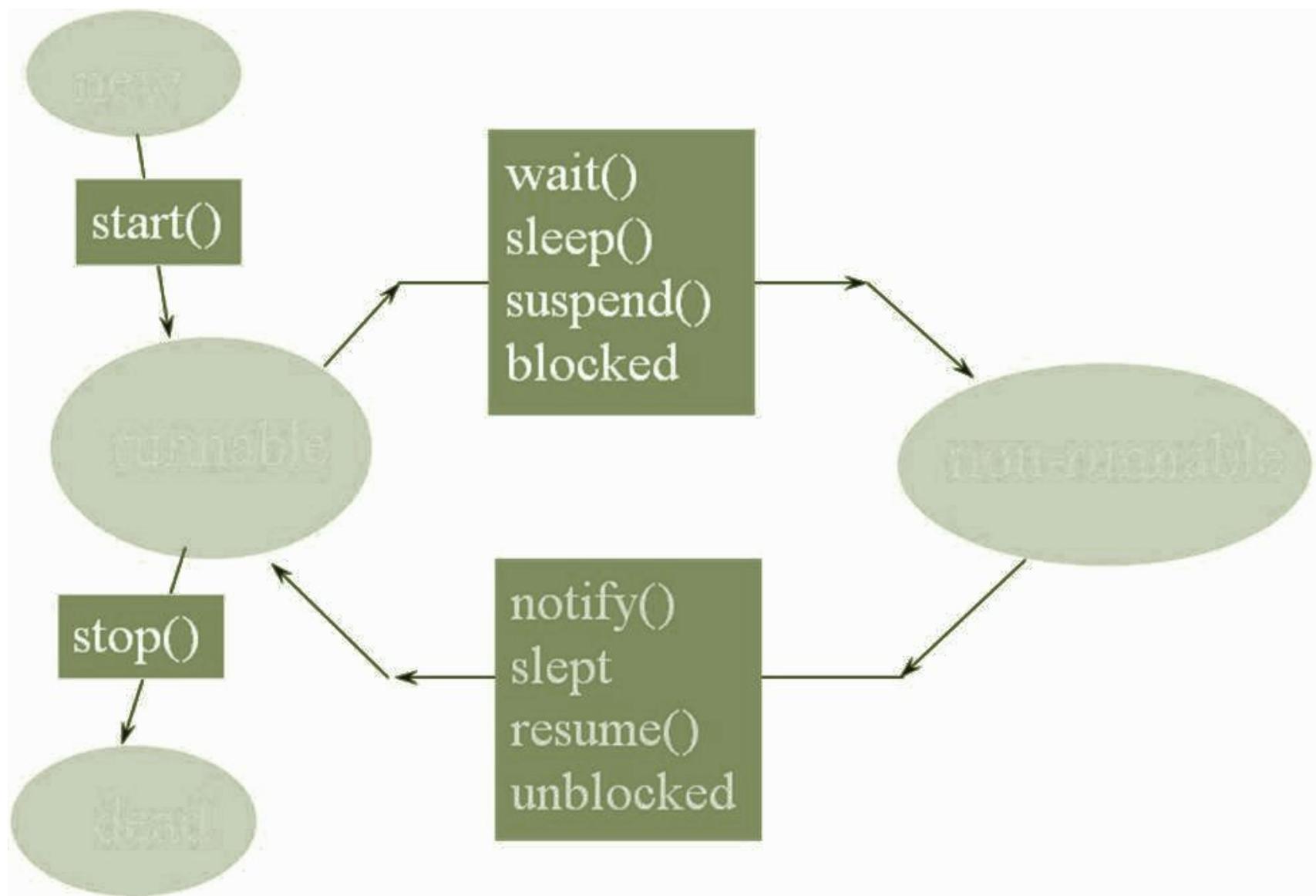
## CONT..

- **Waiting for Notification** – A thread waits for notification from another thread. The thread sends back to runnable state after sending notification from another thread
  - **final void wait() throws InterruptedException**

## CONT..

- **Blocked on I/O** – The thread waits for completion of blocking operation. A thread can enter on this state because of waiting I/O resource. In that case the thread sends back to runnable state after availability of resources
- **Blocked for joint completion** – The thread can come on this state because of waiting the completion of another thread
- **Blocked for lock acquisition** – The thread can come on this state because of waiting to acquire the lock of an object.

# TO BLOCK AND UNBLOCK THREAD



# THREAD PRIORITIES

- 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

# CONT..

## Constant

Thread.MIN\_PRIORITY

## Description

The maximum priority of any thread (an int value of 10)

Thread.MAX\_PRIORITY

The minimum priority of any thread (an int value of 1)

Thread.NORM\_PRIORITY

The normal priority of any thread (an int value of 5)

## Method

## Description

setPriority(int val) This is method is used to set the priority of thread.

getPriority()

This method is used to get the priority of thread.

## CONT..

- When a Java thread is created, it inherits its priority from the thread that created it
- In Java runtime system, **preemptive scheduling** algorithm is applied. If at the execution time a thread with a higher priority and all other threads are runnable then the runtime system chooses the new **higher priority** thread for execution
- On the other hand, if two threads of the same priority are waiting to be executed by the CPU then the **round-robin** algorithm is applied in which the scheduler chooses one of them to run according to their round of **time-slice**

# THREAD SCHEDULER

- In the implementation of threading scheduler usually applies one of the two following strategies:
  - **Preemptive scheduling** – If the new thread has a higher priority then current running thread leaves the runnable state and higher priority thread enter to the runnable state
  - **Time-Sliced (Round-Robin) Scheduling** – A running thread is allowed to be execute for the fixed time, after completion the time, current thread indicates to the another thread to enter it in the runnable state

# DAEMON THREADS

- In Java, any thread can be a Daemon thread. Daemon threads are like a **service providers** for other threads or objects running in the same process as the daemon thread
- Daemon threads are used for background supporting tasks and are only needed while normal threads are executing
- If normal threads are not running and remaining threads are daemon threads then the interpreter exits

## CONT..

- **setDaemon (true/false)** – This method is used to specify that a thread is daemon thread
- **public boolean isDaemon ()** – This method is used to determine the thread is daemon thread or not.

# CREATING A THREAD

- Java defines two ways in to create a thread
  - You can extend the Thread class, itself
  - You can implement the Runnable interface

# CREATING A THREAD BY EXTENDING THREAD CLASS

- 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

| SN | Methods with Description                                                                                                                                                                                                           |
|----|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1  | <b>public void start ()</b><br>Starts the thread in a separate path of execution, then invokes the run() method on this Thread object.                                                                                             |
| 2  | <b>public void run ()</b><br>If this Thread object was instantiated using a separate Runnable target, the run() method is invoked on that Runnable object.                                                                         |
| 3  | <b>public final void setName (String name)</b><br>Changes the name of the Thread object. There is also a getName () method for retrieving the name.                                                                                |
| 4  | <b>public final void setPriority (int priority)</b><br>Sets the priority of this Thread object. The possible values are between 1 and 10.                                                                                          |
| 5  | <b>public final void setDaemon (boolean on)</b><br>A parameter of true denotes this Thread as a daemon thread.                                                                                                                     |
| 6  | <b>public final void join (long millisec)</b><br>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  | <b>public void interrupt ()</b><br>Interrupts this thread, causing it to continue execution if it was blocked for any reason.                                                                                                      |
| 8  | <b>public final boolean isAlive()</b><br>Returns true if the thread is alive, which is any time after the thread has been started but before it runs to completion.                                                                |

## CONT..

- The previous methods are invoked on a particular Thread object
- The following methods in the Thread class are static
- Invoking one of the static methods performs the operation on the currently running thread

# CONT..

| SN | Methods with Description                                                                                                                                       |
|----|----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1  | <b>public static void yield()</b><br>Causes the currently running thread to yield to any other threads of the same priority that are waiting to be scheduled   |
| 2  | <b>public static void sleep(long millisec)</b><br>Causes the currently running thread to block for at least the specified number of milliseconds               |
| 3  | <b>public static boolean holdsLock (Object x)</b><br>Returns true if the current thread holds the lock on the given Object.                                    |
| 4  | <b>public static Thread currentThread ()</b><br>Returns a reference to the currently running thread, which is the thread that invokes this method.             |
| 5  | <b>public static void dumpStack ()</b><br>Prints the stack trace for the currently running thread, which is useful when debugging a multithreaded application. |

# CREATING A 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:

## CONT..

- 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();

# JAVA - THREAD CONTROL

- While suspend ( ), resume ( ), and stop ( ) methods defined by **Thread** class seem to be a perfectly reasonable and convenient approach to managing the execution of threads, they must not be used for new Java programs and obsolete in newer versions of Java

# USING ISALIVE () AND JOIN ()

- Typically, the main thread is the last thread to finish in a program. However, there isn't any guarantee that the main thread won't finish before a child thread finishes
- The main method to sleep until the child threads terminate. However, we estimated the time it takes for the child threads to complete processing
- If our estimate was too short, a child thread could terminate after the main thread terminates
- Therefore, the sleep technique isn't the best one to use to guarantee that the main thread terminates last

## CONT..

- `isAlive ()` method determines whether a thread is still running. If it is, the `isAlive ()` method returns a Boolean true value; otherwise, a Boolean false is returned
- You can use the `isAlive ()` method to examine whether a child thread continues to run. The `join ()` method works differently than the `isAlive ()` method
- The `join ()` method waits until the child thread terminates and “joins” the main thread
- In addition, you can use the `join ()` method to specify the amount of time you want to wait for a child thread to terminate.

# THREAD 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**”

## CONT..

- Threads are synchronized in Java through the use of a monitor. Think of a monitor as an object that enables a thread to access a resource
- Only one thread can use a monitor at any one time period. Programmers say that the thread *owns* the monitor for that period of time. The monitor is also called a *semaphore*
- A thread can own a monitor only if no other thread owns the monitor
- If the monitor is available, a thread can own the monitor and have exclusive access to the resource associated with the monitor

## CONT..

- If the monitor is not available, the thread is suspended until the monitor becomes available. Programmers say that the thread is *waiting* for the monitor
- You have two ways in which you can synchronize threads:
  - You can use the synchronized method or
  - The synchronized statement.

# SYNCHRONIZED STATEMENT

- This is the general form of the synchronized statement:
  - `synchronized(object) { // statements to be synchronized}`
- Here, object is a reference to the object being synchronized
- A synchronized block ensures that a call to a method that is a member of object occurs only after the current thread has successfully entered object's monitor
- Calls to the methods contained in the synchronized block happen only after the thread enters the monitor of the object

## CONT..

- Synchronizing a method is the best way to restrict the use of a method one thread at a time
- However, there will be occasions when you won't be able to synchronize a method, such as when you use a class that is provided to you by a third party
- Although you can call methods within a synchronized block, the method declaration must be made outside a synchronized block

# THE SYNCHRONIZED METHOD

- All objects in Java have a monitor. A thread enters a monitor whenever a method modified by the keyword synchronized is called
- The thread that is first to call the synchronized method is said to be *inside* the method and therefore owns the method and resources used by the method
- Another thread that calls the synchronized method is suspended until the first thread relinquishes the synchronized method

## CONT..

- If a synchronized method is an instance method, the synchronized method activates the lock associated with the instance that called the synchronized method, which is the object known as this during the execution of the body of the method
- If the synchronized method is static, it activates the lock associated with the class object that defines the synchronized method

# REASONS OF WRONG OUTPUT

- Thread priorities do work on most OS's but they often have minimal effect. Priorities help to order the threads that are in the run queue only and will not change the order that the threads are being run in any major way unless you are doing a ton of CPU in each of the threads.
- Your program looks to use a lot of CPU but unless you have fewer threads than there are cores, you may not see any change in output order by setting your thread priorities. If there is a free CPU then even a lower priority thread will be scheduled to run.

# REASONS OF WRONG OUTPUT

- Also, threads are not starved. Even a lower priority thread will given time to run quite often in such a situation as this. You should see higher priority threads be given time sliced to run *more often* but it does not mean lower priority threads will wait for them to finish before running themselves.
- Even if priorities do help to give one thread more CPU than the others, threaded programs are subject to race conditions which help inject a large amount of randomness to their execution. What you should see however, is the max priority thread is more likely to spit out its 0 message sooner than the rest

# JAVA - INTERTHREAD COMMUNICATION

- Consider the classic queuing problem, where one thread is producing some data and another is consuming it
- To make the problem more interesting, suppose that the producer has to wait until the consumer is finished before it generates more data
- In a polling system, the consumer would waste many CPU cycles while it waited for the producer to produce
- Once the producer was finished, it would start polling, wasting more CPU cycles waiting for the consumer to finish, and so on. Clearly, this situation is undesirable

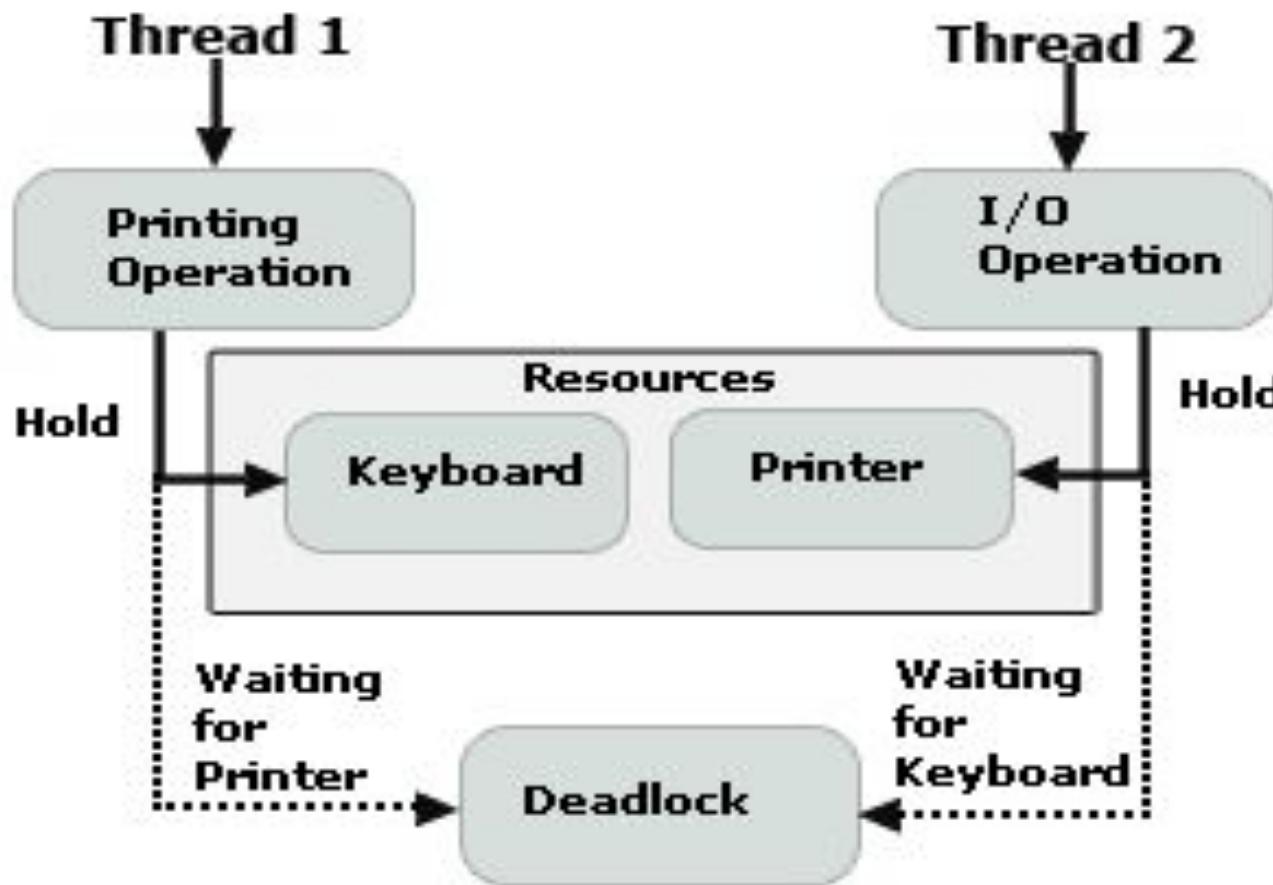
## CONT..

- To avoid polling, Java includes an elegant interprocess communication mechanism via the following methods:
  - **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

## CONT..

- 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
- These methods are declared within Object. Various forms of wait( ) exist that allow you to specify a period of time to wait

# JAVA - THREAD DEADLOCK



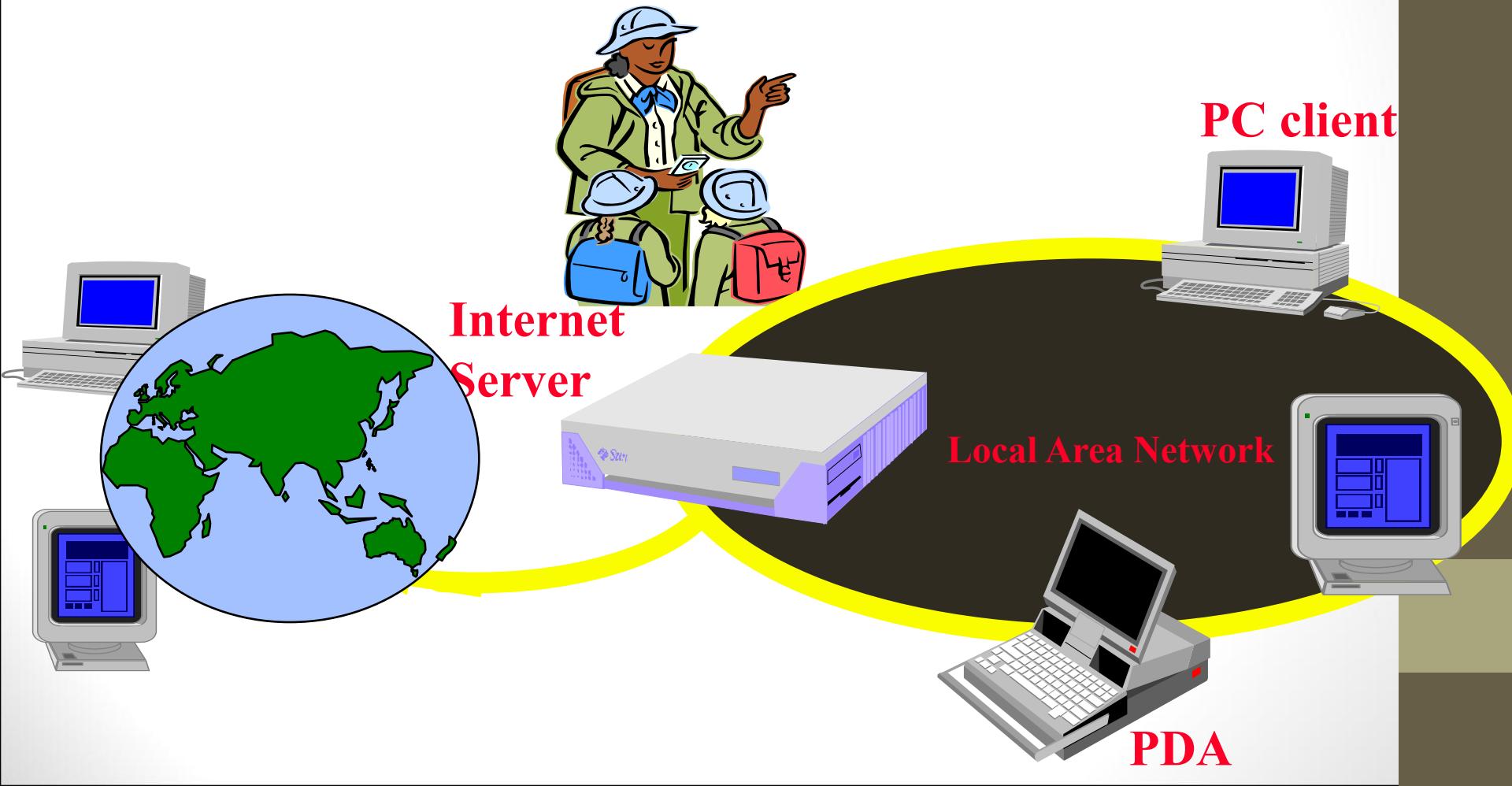
## CONT..

- A special type of error that you need to avoid that relates specifically to multitasking is deadlock, which occurs when two threads have a circular dependency on a pair of synchronized objects
- For example, suppose one thread enters the monitor on object X and another thread enters the monitor on object Y.
  - If the thread in X tries to call any synchronized method on Y, it will block as expected
  - However, if the thread in Y, in turn, tries to call any synchronized method on X, the thread waits forever, because to access X, it would have to release its own lock on Y so that the first thread could complete

# Networking

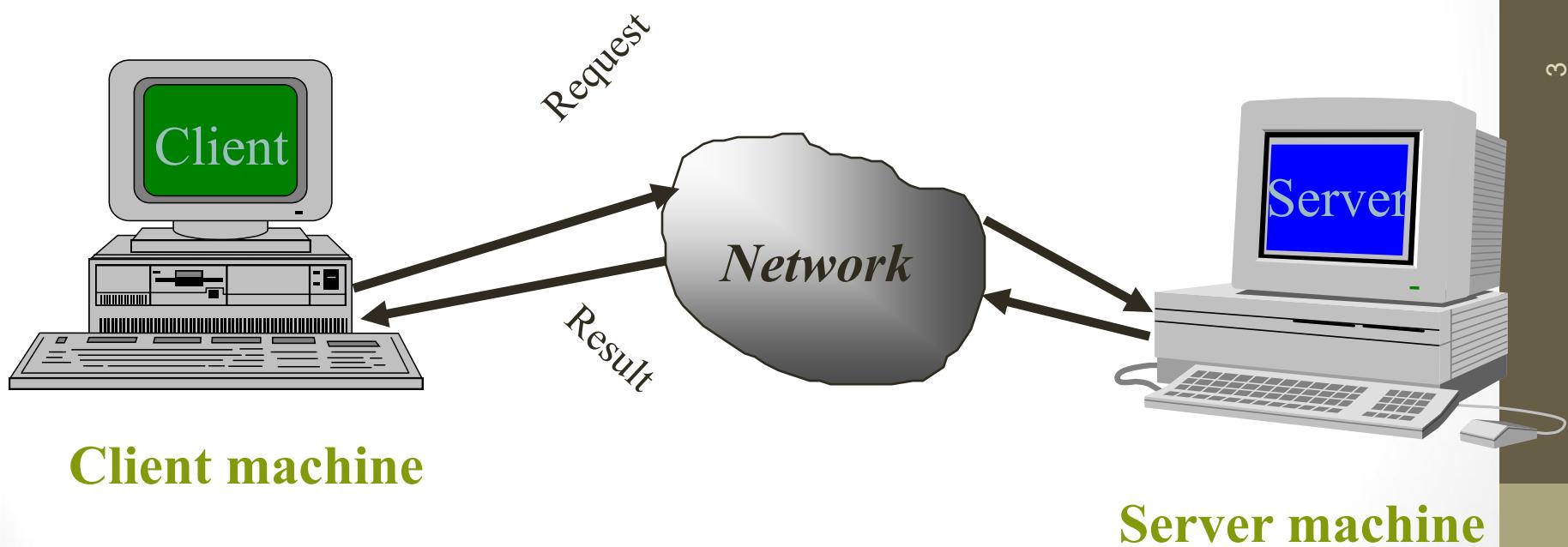
# Java networking

*Network programming* refers to writing programs that execute across multiple devices (computers), in which the devices are all connected to each other using a network.



# Elements of C-S Computing

a client, a server, and network

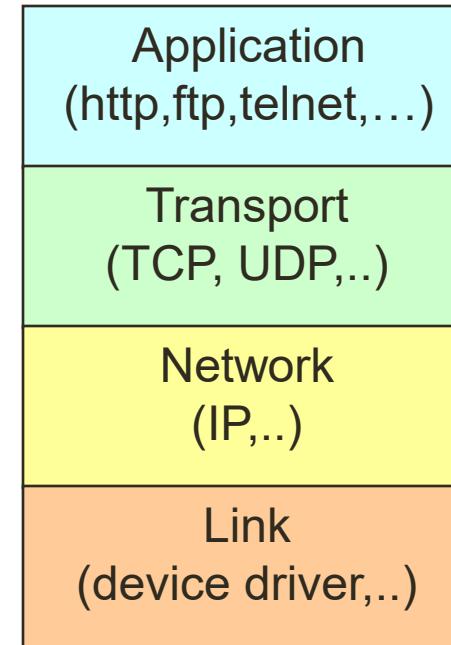


# Client/Server

- A server is anything that has some resource that can be shared.
  - Compute servers, which provide computing power;
  - print servers, which manage a collection of printers;
  - disk servers, which provide networked disk space;
  - web servers, which store web pages.
- A client is simply any other entity that wants to gain access to a particular server.

# Networking Basics

- Applications Layer
  - Standard apps
    - HTTP
    - FTP
    - Telnet
  - User apps
- Transport Layer
  - TCP
  - UDP
  - Programming Interface:
    - Sockets
- Network Layer
  - IP
- Data Link Layer
  - Device drivers



# Java Networking

- The Java platform is very famous in part because of its suitability for writing programs that use and interact with the resources on the Internet and the World Wide Web
- Java-compatible browsers use this ability of the Java platform to the extreme to transport and run applets over the Internet

# Java Networking

- Computers running on the Internet communicate to each other using either the **Transmission Control Protocol (TCP)** or the **User Datagram Protocol (UDP)**.
- When you write Java programs that communicate over the network, you are programming at the application layer.
- To decide which Java classes your programs should use, you do need to understand how TCP and UDP differ.

# TCP(Transmission Control Protocol)

- Internet Protocol (IP) is a low-level routing protocol that breaks data into small packets and sends them to an address across a network, which does not guarantee to deliver said packets to the destination.
- TCP allows a reliable communication between two applications and data received in the same order it was sent.
- Otherwise, an error is reported.
- TCP provides connection-oriented, reliable, bidirectional, persistent, point-to-point, stream-based connections between hosts on the internet.
- Stream communication
- Example applications: HTTP, FTP, Telnet

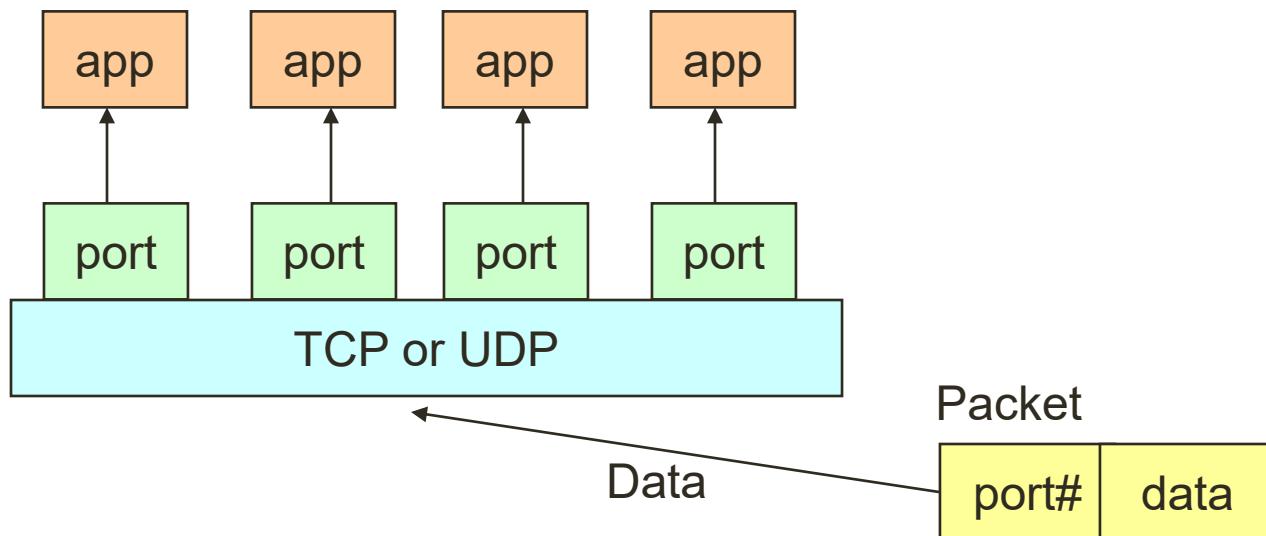
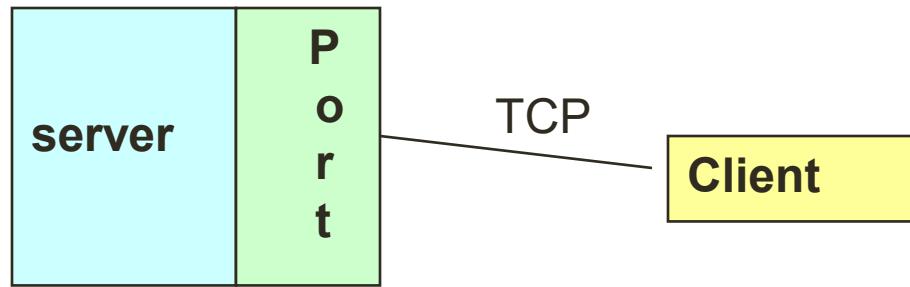
# UDP(User Datagram Protocol)

- A connection-less protocol that allows for packets of data to be transmitted between applications.
- Rather, it sends independent packets of data, called datagrams, from one application to another.
- The order of delivery is not important and is not guaranteed, and each message is independent of any other.
- Fast, connection-less, unreliable transport of packets.
- Datagram communication.
- Example applications: Clock server, Ping

# IP address and Port

- Data transmitted over the Internet is accompanied by addressing information that identifies the computer and the port for which it is destined.
- The computer is identified by its 32-bit IP address, which IP uses to deliver data to the right computer on the network.
- This address type was specified by IPV4.
- IPV6 uses 128 bit value to represent an address.
- Adv: it supports much larger address space.
- Downward compatible with IPV4.
- Ports are identified by a 16-bit number, which TCP and UDP use to deliver the data to the right application on the computer.

# Understanding Ports



# Understanding Ports

- Port numbers range from 0 to 65,535 because ports are represented by 16-bit numbers.
- Some ports have been reserved to support common/well known services:
  - Domain Name System (DNS): 53
  - File Transfer Protocol (FTP): 21
  - Hypertext Transfer Protocol (HTTP): 80
  - Network News Transfer Protocol (NNTP):119
  - Post Office Protocol (POP3): 110
  - Simple Mail Transfer Protocol (SMTP): 25
  - Telnet: 23
  - These ports are called well-known ports.
  - User level process/services generally use port number value  $\geq 1024$ .
- Your applications should not attempt to bind to them.

# Con't

- A server process is said to “listen” to a port until a client is connected to it.
- A server is allowed to accept multiple clients connected to the same port number, although each session is unique.
- To manage multiple client connections, a server process must be multithreaded.

# Socket

- Sockets are just like an end-point of two-way communication, which allow applications to communicate using network hardware and operating systems.
- Basically, socket = IP + ports.
- A client program creates a socket on its end of the communication and attempts to connect that socket to a server.
- When the connection is made, the server creates a socket object on its end of the communication.
- The client and server can now communicate by writing to and reading from the socket.
- Need to include java.net package.
- The `java.net.Socket` class represents a socket, and the `java.net.ServerSocket` class provides a mechanism for the server program to listen for clients and establish connections with them.

# Proxy Servers

- A proxy server speaks the client side of a protocol to another server.
- This is often required when clients have certain restrictions on which servers they can connect to.
- Thus, a client would connect to a proxy server, which did not have such restrictions, and the proxy server would in turn communicate for the client.

# Domain Naming Service (DNS)

- Difficult to remember “`http://192.9.9.1/`”.
- DNS is an Internet service that translates domain names into IP addresses.
- [www.example.com](http://www.example.com)

## InetAddress

- The `InetAddress` class is used to encapsulate both the numerical IP address and the domain name for that address.

# The Networking Classes and Interfaces

|                             |                                  |                         |
|-----------------------------|----------------------------------|-------------------------|
| Authenticator (Java 2)      | InetSocketAddress (Java 2, v1.4) | SocketImpl              |
| ContentHandler              | JarURLConnection (Java 2)        | SocketPermission        |
| DatagramPacket              | MulticastSocket                  | URI (Java 2, v1.4)      |
| DatagramSocket              | NetPermission                    | URL                     |
| DatagramSocketImpl          | NetworkInterface (Java 2, v1.4)  | URLClassLoader (Java 2) |
| HttpURLConnection           | PasswordAuthentication (Java 2)  | URLConnection           |
| InetAddress                 | ServerSocket                     | URLDecoder (Java 2)     |
| Inet4Address (Java 2, v1.4) | Socket                           | URLEncoder              |
| Inet6Address (Java 2, v1.4) | SocketAddress (Java 2, v1.4)     | URLStreamHandler        |

---

|                       |                   |                                                      |
|-----------------------|-------------------|------------------------------------------------------|
| ContentHandlerFactory | SocketImplFactory | URLStreamHandlerFactory                              |
| FileNameMap           | SocketOptions     | DatagramSocketImplFactory<br>(added by Java 2, v1.3) |

# Factory Methods

- The InetAddress class has no visible constructors.
- Factory methods are convention whereby static methods in a class return an instance of that class.
- `static InetAddress getLocalHost( ) throws UnknownHostException`
- `static InetAddress getByName(String hostName) throws UnknownHostException`
- `static InetAddress[ ] getAllByName(String hostName) throws UnknownHostException`

# Methods of InetAddress class

|                                      |                                                                                            |
|--------------------------------------|--------------------------------------------------------------------------------------------|
| boolean equals(Object <i>other</i> ) | Returns true if this object has the same Internet address as <i>other</i> .                |
| byte[ ] getAddress()                 | Returns a byte array that represents the object's Internet address in network byte order.  |
| String getHostAddress()              | Returns a string that represents the host address associated with the InetAddress object.  |
| String getHostName()                 | Returns a string that represents the host name associated with the InetAddress object.     |
| boolean isMulticastAddress()         | Returns true if this Internet address is a multicast address. Otherwise, it returns false. |
| String toString()                    | Returns a string that lists the host name and the IP address for convenience.              |

# Networking Basics

- `java.net` provides the classes for implementing networking applications.
- `java.net` package can be divided in two sections:
- **A Low Level API**, which deals with the following abstractions:
  - Addresses, which are networking identifiers, like IP addresses.
  - Sockets, which are basic bidirectional data communication mechanisms.
  - Interfaces, which describe network interfaces.
- **A High Level API**, which deals with the following abstractions:
  - URIs, which represent Universal Resource Identifiers.
  - URLs, which represent Universal Resource Locators.
  - Connections, which represents connections to the resource pointed to by URLs.

# Sockets

- TCP: Socket, and ServerSocket classes
- Socket – for implementing a client
- ServerSocket – for implementing a server
- UDP: DatagramPacket, DatagramSocket, and MulticastSocket classes.

## TCP/IP Client Sockets

- `Socket(String hostName, int port)`
- Creates a socket connecting the local host to the named host and port; can throw an `UnknownHostException` or an `IOException`.
- `Socket(InetAddress ipAddress, int port)`
- Creates a socket using a preexisting `InetAddress` object and a port; can throw an `IOException`.

# Methods of Sockets class

- InetAddress getInetAddress( )
- Returns the InetAddress associated with the Socket object.
- int getPort( )
- Returns the remote port to which this Socket object is connected.
- int getLocalPort( )
- Returns the local port to which this Socket object is connected.
- Once the Socket object has been created, it gain access to the input and output streams associated with it.
- Each of these methods can throw an IOException.
- InputStream getInputStream( )
- Returns the InputStream associated with the invoking socket.
- OutputStream getOutputStream( )
- Returns the OutputStream associated with the invoking socket.

# High level API

- URL is the class representing a Universal Resource Locator
  - URLConnection is created from a URL and is the communication link used to access the resource pointed by the URL
    - This abstract class will delegate most of the work to the underlying protocol handlers like http or ftp.
  - HttpURLConnection is a subclass of URLConnection and provides some additional functionalities specific to the HTTP protocol

# URL(Uniform Resource Locator)

- Uniquely identify information on the Internet.
- URLs are ubiquitous; every browser uses them to identify information on the Web.
- Format
- <http://www.example.com:80/index.htm>
- http : protocol
- www.example.com : host name or IP address
- 80 : port address
- Index.htm : resource (filename)
- URL class has several constructors, and each can throw a MalformedURLException.

# Con't

- URL(String urlSpecifier)
- URL(String protocolName, String hostName, int port, String path)
- URL(String protocolName, String hostName, String path)
- URL(URL urlObj, String urlSpecifier)

# URL, URN and URI

- URLs always start with a protocol (http) and usually contain information such as the network host name (example.com) and often a document path (/foo/mypage.html).
- URLs may have query parameters and fragment identifiers.
- URN identifies a resource by a unique and persistent name.
  - URNs can identify ideas and concepts. They are not restricted to identifying documents.
  - When a URN does represent a document, it can be translated into a URL by a "resolver".
  - The document can then be downloaded from the URL.

# Cont...

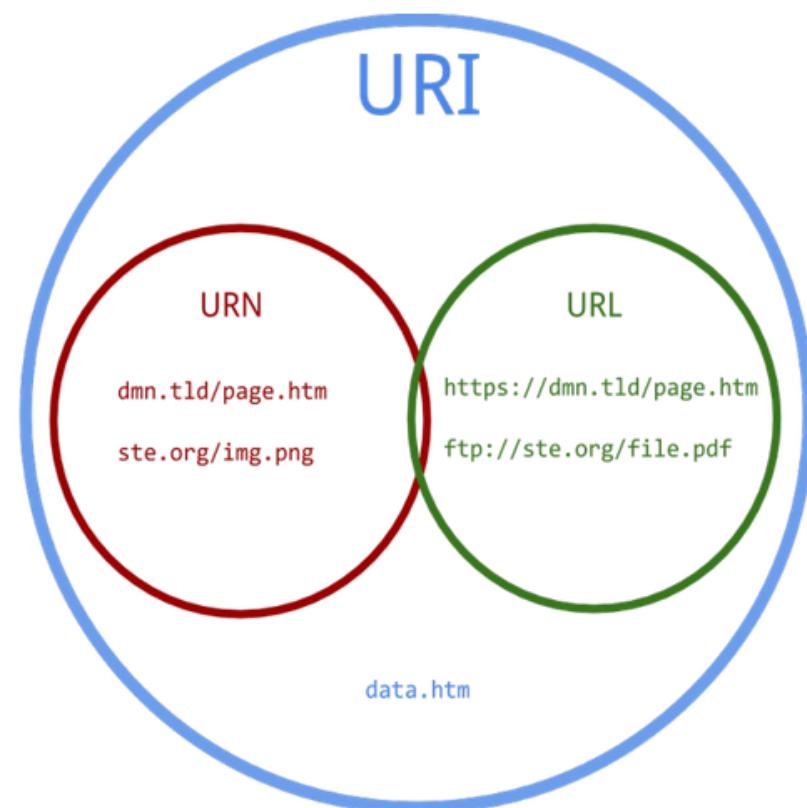
- A Uniform Resource Identifier (URI) is a compact sequence of characters that identifies an abstract or physical resource.
- A URI can be further classified as a locator, a name, or both.
  - The term “Uniform Resource Locator” (URL) refers to the subset of URIs that, in addition to identifying a resource, provide a means of locating the resource by describing its primary access mechanism.
  - (e.g., its network “location”).

# Cont...

- First of all (as we see in the diagram as well) a URL is a **type of URI**.
- But that doesn't mean all URIs are URLs. All butterflies fly, but not everything that flies is a butterfly.
- The part that makes a URI a URL is the inclusion of the "access mechanism", or "network location", e.g. `http://` or `ftp://`.
- The URN is the "globally unique" part of the identification; it's a unique name.

# Cont..

- One can classify URIs as locators (URLs), or as names (URNs), or as both.
- A Uniform Resource Name (URN) functions like a person's name, while a Uniform Resource Locator (URL) resembles that person's street address.
- In other words: the URN defines an item's identity, while the URL provides a method for finding it.
- **URLDemo.java**
- **urlTOuri.java**



# URLConnection

- `URLConnection` is a general-purpose class for accessing the attributes of a remote resource.
- `URLConnection` to inspect the properties of the remote object before actually transporting it locally.

# TCP/IP Server Sockets

- The `ServerSocket` class is used to create servers that listen for either local or remote client programs to connect to them on published ports.
- When you create a `ServerSocket`, it will register itself with the system as having an interest in client connections.
- The constructors for `ServerSocket` reflect the port number that you wish to accept connections on and, optionally, how long you want the queue for said port to be.
- The queue length tells the system how many client connections it can leave pending before it should simply refuse connections.
- The default is 50.

# Con't

- `ServerSocket(int port)`
- Creates server socket on the specified port with a queue length of 50.
- `ServerSocket(int port, int maxQueue)`
- Creates a server socket on the specified port with a maximum queue length of maxQueue.
- `ServerSocket(int port, int maxQueue, InetAddress localAddress)`
- Creates a server socket on the specified port with a maximum queue length of maxQueue. On a multihomed host, localAddress specifies the IP address to which this socket binds.

# Methods of ServerSocket class

- `public Socket accept() throws IOException`  
Waits for an incoming client. This method blocks until either a client connects to the server on the specified port or the socket times out. Otherwise, this method blocks indefinitely.
- `public int getLocalPort()` returns the port that the server socket is listening on.

# Steps occurs when establishing a TCP connection between two computers using sockets

- The server instantiates a ServerSocket object, denoting which port number communication is to occur on.
- The server invokes the accept() method of the ServerSocket class. This method waits until a client connects to the server on the given port.
- After the server is waiting, a client instantiates a Socket object, specifying the server name and port number to connect to.

# Steps occurs when establishing a TCP connection between two computers using sockets

- The constructor of the Socket class attempts to connect the client to the specified server and port number.
- If communication is established, the client now has a Socket object capable of communicating with the server.
- On the server side, the accept() method returns a reference to a new socket on the server that is connected to the client's socket.

# UDP(Datagram Communication)

- Datagrams are bundles of information passed between machines.
- UDP protocol implements datagrams using two classes:
- DatagramPacket object is the data container.
- DatagramSocket is the mechanism used to send or receive the DatagramPackets.
- Constructors:
  - DatagramPacket(byte data[ ], int size)
  - DatagramPacket(byte data[ ], int offset, int size)
  - DatagramPacket(byte data[ ], int size, InetAddress ipAddress, int port)
  - DatagramPacket(byte data[ ], int offset, int size, InetAddress ipAddress, int port)

# DatagramPacket methods

- `InetAddress getAddress( )`
- Returns the destination InetAddress, typically used for sending.
- `int getPort( )`
- Returns the port number.
- `byte[ ] getData( )`
- Returns the byte array of data contained in the datagram. Mostly used to retrieve data from the datagram after it has been received.
- `int getLength( )`
- Returns the length of the valid data contained in the byte array that would be returned from the `getData( )` method. This typically does not equal the length of the whole byte array.

# What is IP Multicasting?

- IP multi-casting is a communication mechanism in which data is communicated from server to a set of clients who are interested in receiving that data.
  - Any client can dynamically enter or leave the communication.
- To understand how multi-casting works, one needs to understand the structure of multicast IP address.
- In terms of classes, it's the Class D IP addresses that are used as multicast IP addresses.
- Below is the structure of Class D IP addresses.
- So, it can be easily said that multicast IPs range from 224.0.0.0 to 239.255.255.255.
- As with the case of ports (where we have well known ports i.e. 0-1024), there are some reserved multicast IP addresses or well known IP addresses.



# What is IP Multicasting?

- In case of multicast communication, the server sends data on a particular multicast IP address and clients who intend to receive that data need to listen on the same multicast address.
- These clients can be various different networks.
- A group of clients listening to same multicast address is known as host group.