JAVA PROGRAMMING

UNIT I

- **OOP concepts-** Data abstraction- encapsulation- inheritance- benefits of inheritance- polymorphism-classes and objects- procedural and object oriented programming paradigm.
- **Java programming** – History of java- comments data types-variables-constants-scope and life time of variables- operators-operator hierarchy-expressions-type conversion and casting- enumerated types- control flow – block scope- conditional statements-loops-break and continue statements- simple java stand alone programs-arrays- console input and output- formatting output-constructors- methods-parameter passing- static fields and methods- access control- this reference- overloading methods and constructors-recursion-garbage collection- building strings- exploring string class

Need for OOP Paradigm

- OOP is an approach to program organization and development, which attempts to eliminate some of the drawbacks of conventional programming methods by incorporating the best of structured programming features with several new concepts.

- OOP allows us to decompose a problem into number of entities called objects and then build data and methods (functions) around these entities.

- The data of an object can be accessed only by the methods associated with the object.

Introduction

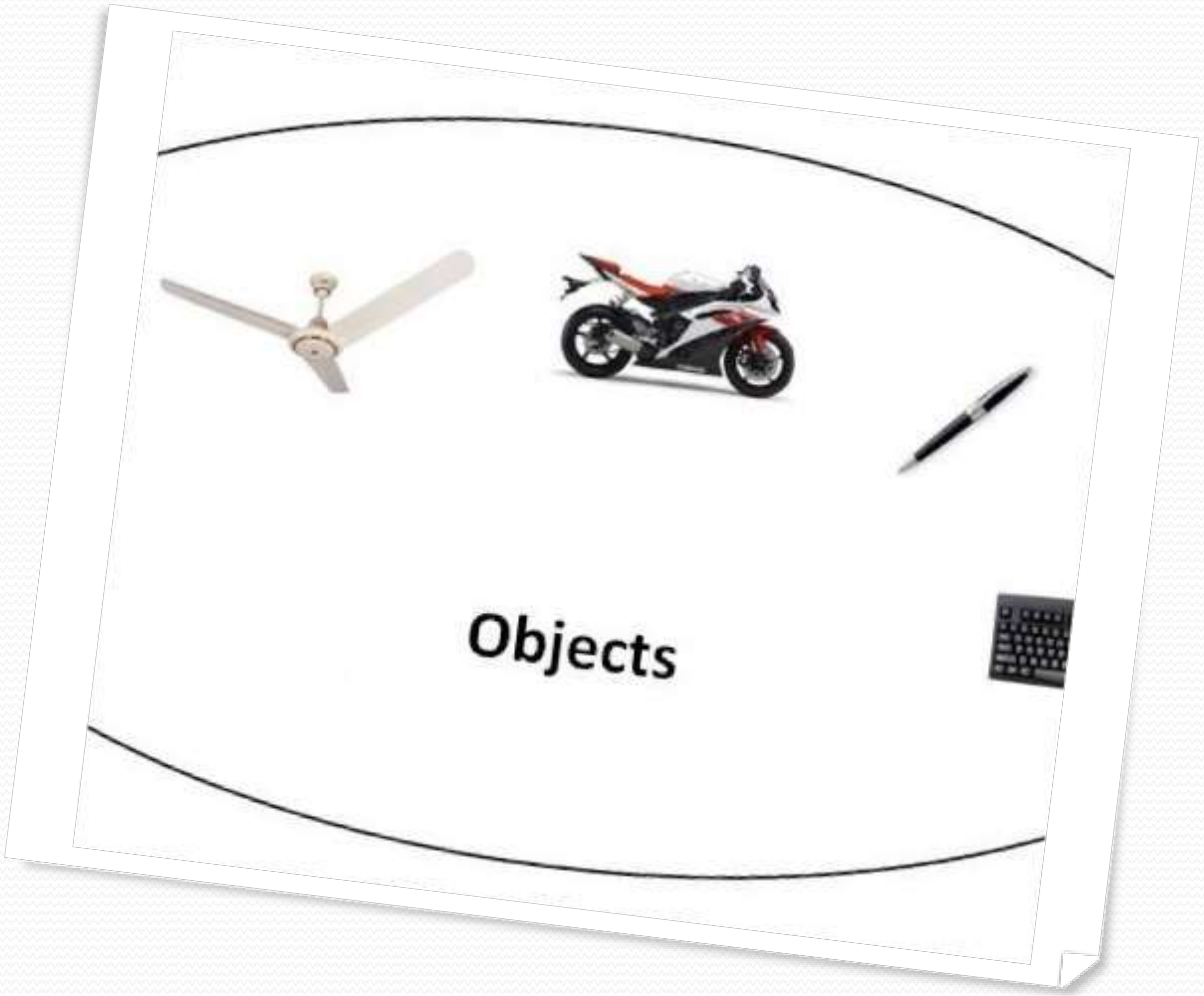
- **Object-oriented programming (OOP)** is a programming paradigm that uses "Objects "and their interactions to design applications.

- It simplifies the software development and maintenance by providing some concepts:

- **Object**
- **Class**
- **Data Abstraction & Encapsulation**
- **Inheritance**
- **Polymorphism**
- **Dynamic Binding**
- **Message Passing**

Object

- 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.


Objects

Class

- The entire set of data and code of an object can be made of a user defined data type with the help of a 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.
- **Classes** are data types based on which objects are created. Objects with similar properties and methods are grouped together to form a Class. Thus a Class represents a set of individual objects.

- Characteristics of an object are represented in a class as Properties. The actions that can be performed by objects become functions of the class and is referred to as Methods.
- A class is thus a collection of objects of similar type . for example: mango, apple, and orange are members of the class fruit . ex: fruit mango; will create an object mango belonging to the class fruit.

Example for class

- **class Human**
- **{**

  private:
- **EyeColor IColor;**
- **NAME personname;**
- **public:**
- **void SetName(NAME anyName);**
- **void SetIColor(EyeColor eyecolor);**
- **};**

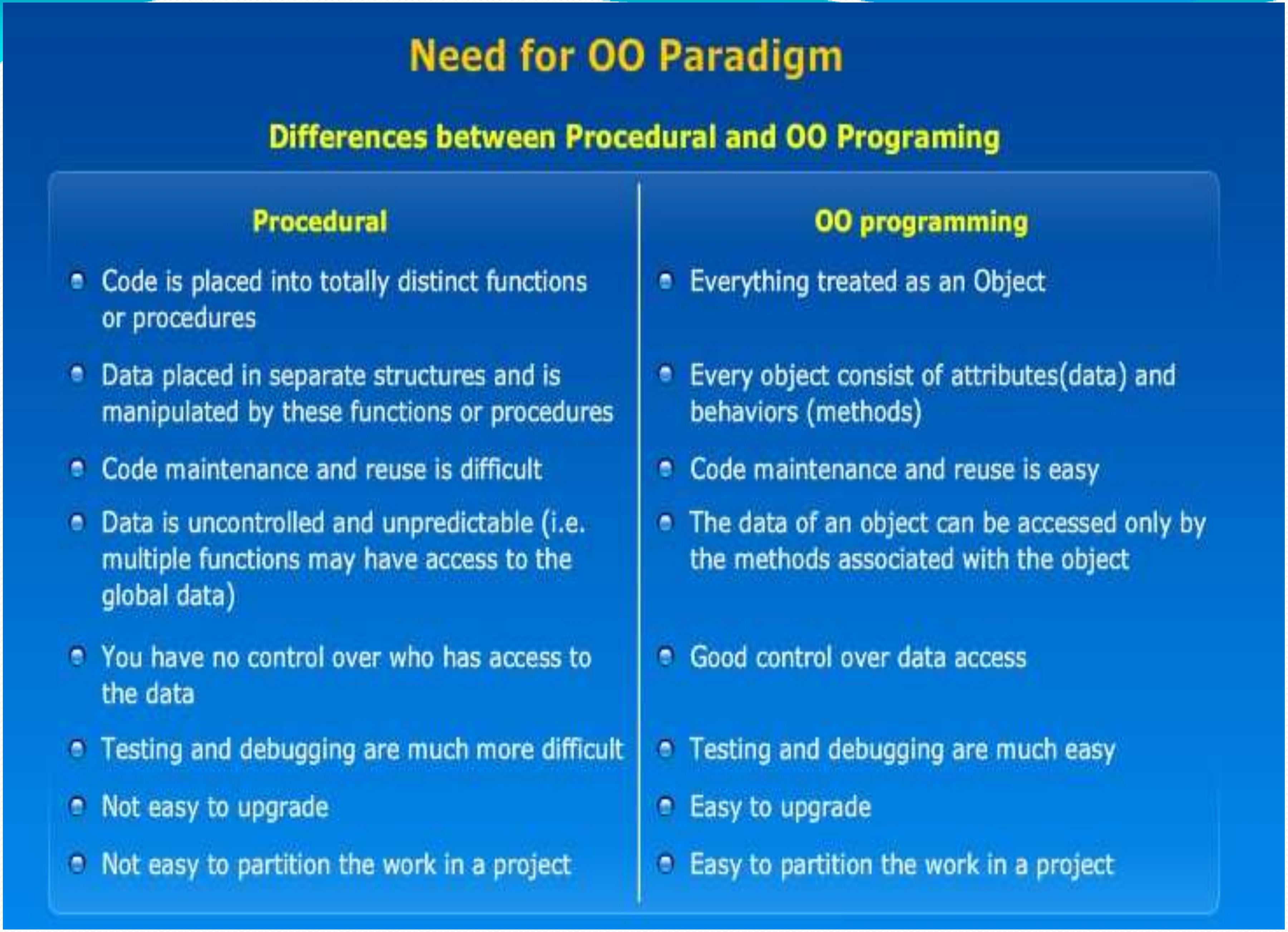
Data abstraction

- Abstraction refers to the act of representing essential features without including the background details or explanations. since the classes use the concept of data abstraction ,they are known as *abstraction data type(ADT).*

For example, a class Car would be made up of an Engine, Gearbox, Steering objects, and many more components. To build the Car class, one does not need to know how the different components work internally, but only how to interface with them, i.e., send messages to them, receive messages from them, and perhaps make the different objects composing the class interact with each other.

An example for abstraction

- Humans manage complexity through abstraction. When you drive your car you do not have to be concerned with the exact internal working of your car(unless you are a mechanic). What you are concerned with is interacting with your car via its interfaces like steering wheel, brake pedal, accelerator pedal etc. Various manufacturers of car has different implementation of car working but its basic interface has not changed (i.e. you still use steering wheel, brake pedal, accelerator pedal etc to interact with your car). Hence the knowledge you have of your car is abstract.



Need for OO Paradigm

Differences between Procedural and OO Programing

Procedural	OO programming
Code is placed into totally distinct functions or procedures	Everything treated as an Object
Data placed in separate structures and is manipulated by these functions or procedures	Every object consist of attributes(data) and behaviors (methods)
Code maintenance and reuse is difficult	Code maintenance and reuse is easy
Data is uncontrolled and unpredictable (i.e. multiple functions may have access to the global data)	The data of an object can be accessed only by the methods associated with the object
You have no control over who has access to the data	Good control over data access
Testing and debugging are much more difficult	Testing and debugging are much easy
Not easy to upgrade	Easy to upgrade
Not easy to partition the work in a project	Easy to partition the work in a project

Some of the Object-Oriented Paradigm are:

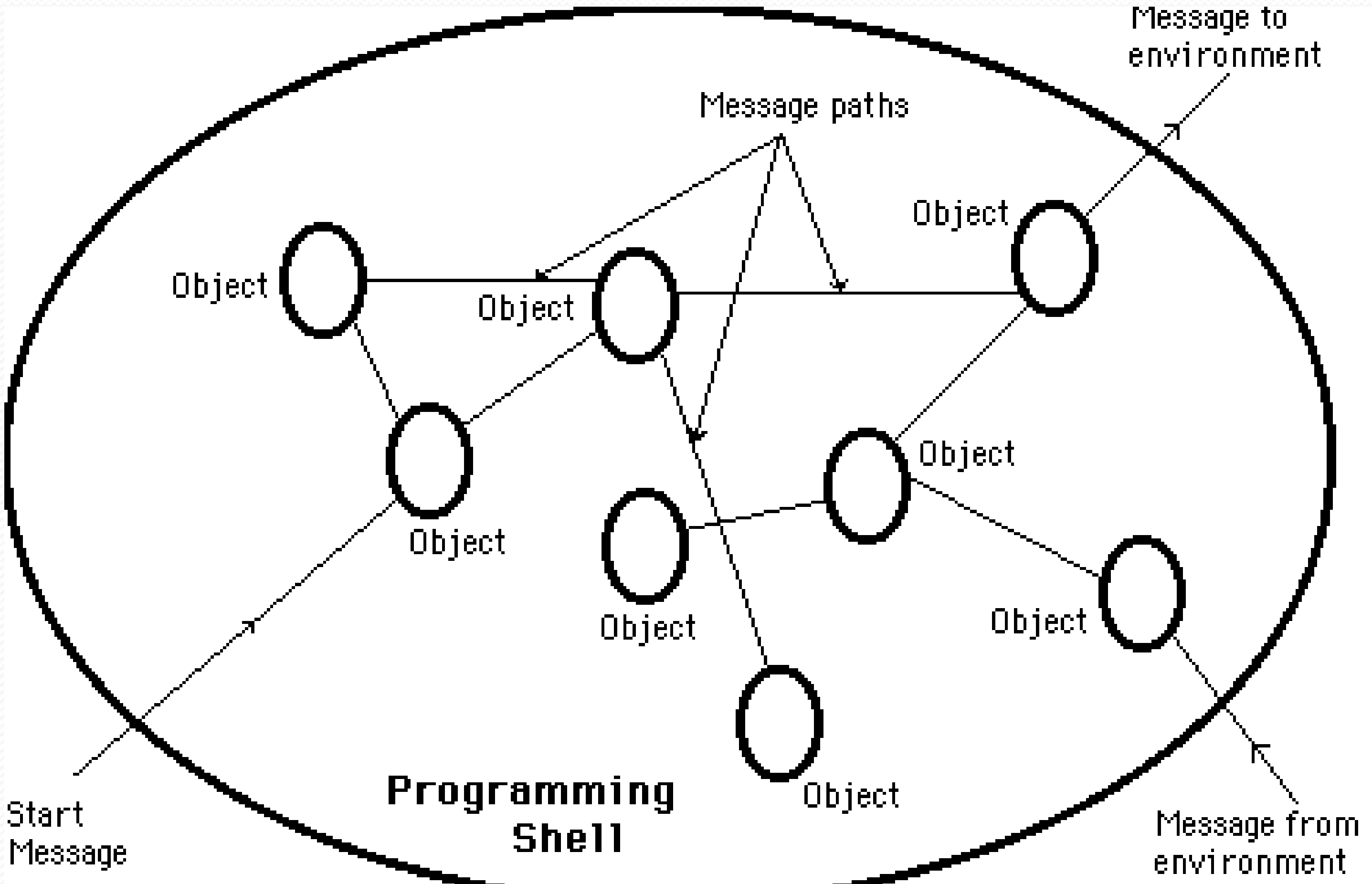
1. Emphasis is on data rather than procedure.
2. Programs are divided into objects.
3. Data Structures are designed such that they Characterize the objects.
4 Methods that operate on the data of an object are tied together in the data structure.
5 Data is hidden and can not be accessed by external functions.
6 Objects may communicate with each other through methods.

A way of viewing world – Agents

- OOP uses an approach of treating a real world agent as an object.

- Object-oriented programming organizes a program around its data (that is, objects) and a set of well-defined interfaces to that data.

- An object-oriented program can be characterized as *data controlling access to code* by switching the controlling entity to data.


Message to environment
Message paths
Object
Object
Object
Object
Object
Object
Object
Object
Object
Start Message
Programming Shell
Message from environment

Responsibility

- primary motivation is the need for a platform-independent (that is, architecture- neutral) language that could be used to create software to be embedded in various consumer electronic devices, such as microwave ovens and remote controls.
- Objects with clear responsibilities.
- Each class should have a clear responsibility.
- If you can't state the purpose of a class in a single, clear sentence, then perhaps your class structure needs some thought.

Messages

- We all like to use programs that let us know what's going on. Programs that keep us informed often do so by displaying status and error messages.

- These messages need to be translated so they can be understood by end users around the world.
-
- The Section discusses translatable text messages. Usually, you're done after you move a message String into a ResourceBundle.

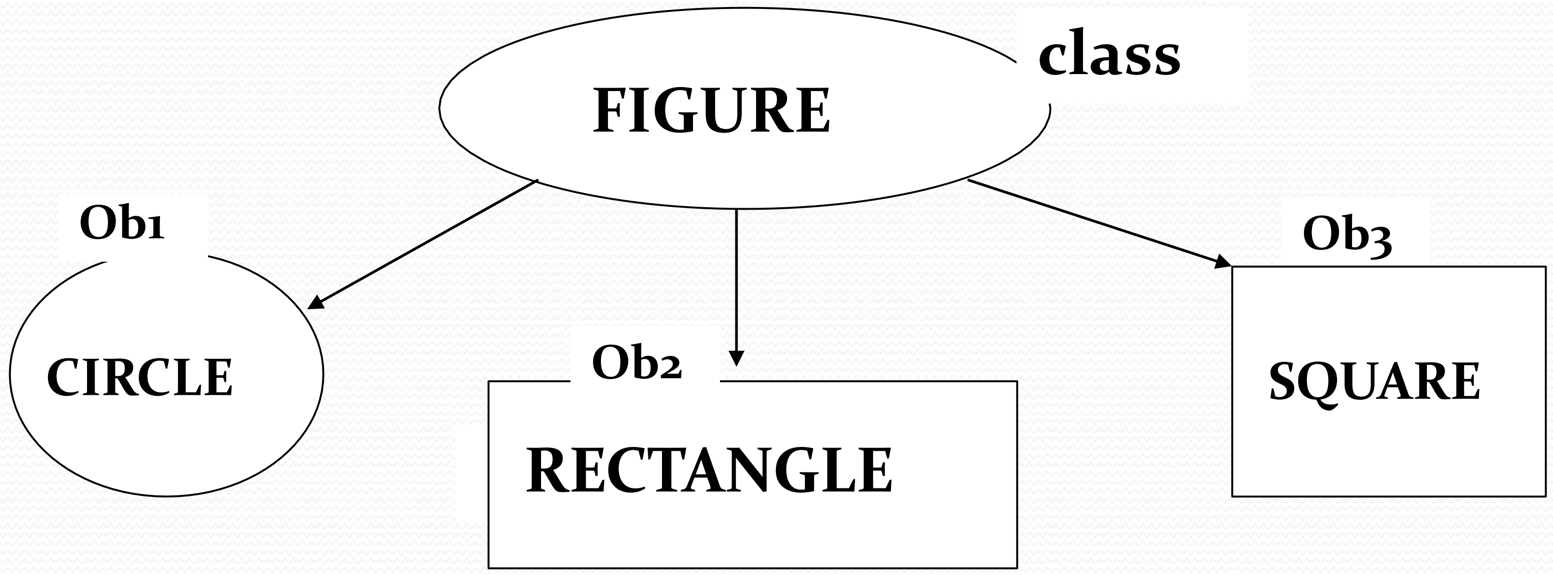
- If you've embedded variable data in a message, you'll have to take some extra steps to prepare it for translation.

Methods

- A method is a group of instructions that is given a name and can be called up at any point in a program simply by quoting that name.
- Drawing a Triangle require draw of three straight lines. This instruction three times to draw a simple triangle.
- We can define a method to call this instruction three times and draw the triangle(i.e. create a method drawLine() to draw lines and this method is called repeatedly to achieve the needed task)
- The idea of methods appears in all programming languages, although sometimes it goes under the name *functions* and sometimes under the name *procedures*.
- The name *methods* is a throw-back to the language C++, from which Java was developed.
- In C++, there is an object called a *class* which can contain methods. However, everything in Java is enclosed within a class .so the functions within it are called methods

CLASSES

- Class is blue print or an idea of an Object
- From One class any number of Instances can be created
- It is an encapsulation of attributes and methods



syntax of CLASS

```
class <ClassName>
{
    attributes/variables;
    Constructors();
    methods();
}
```

INSTANCE

- Instance is an Object of a class which is an entity with its own attribute values and methods.

- Creating an Instance

```
ClassName  refVariable;
refVariable = new Constructor();
                  or
ClassName  refVariable = new  Constructor();
```

Java Class Hierarchy

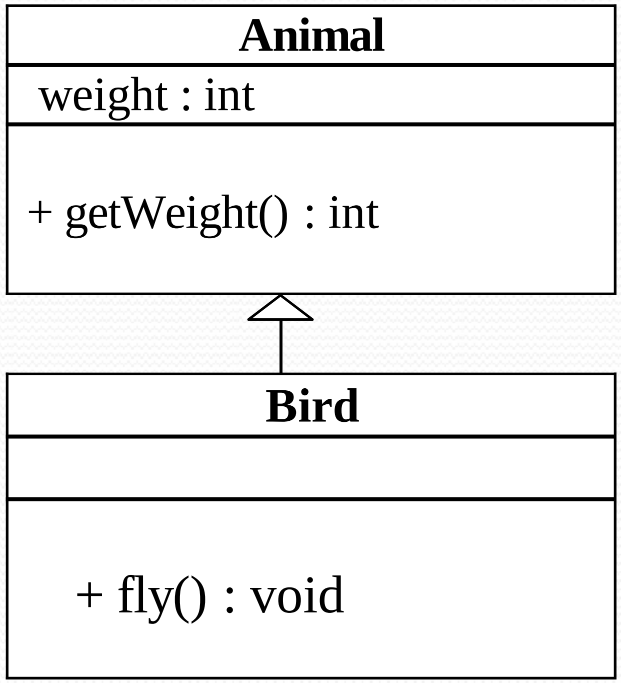
- In Java, class “Object” is the base class to all other classes
  - If we do not explicitly say extends in a new class definition, it implicitly extends Object
  - The tree of classes that extend from Object and all of its subclasses are is called the class hierarchy
  - All classes eventually lead back up to Object
  - This will enable consistent access of objects of different classes.

Inheritance

- Methods allows to reuse a sequence of statements
- *Inheritance* allows to reuse classes by deriving a new class from an existing one
- The existing class is called the *parent class,* or *superclass*, or *base class*
- The derived class is called the *child class* or *subclass*.
- The child class inherits characteristics of the parent class(i.e the child class *inherits* the methods and data defined for the parent class

Inheritance

- Inheritance relationships are often shown graphically in a *class diagram*, with the arrow pointing to the parent class



Method Binding

- Objects are used to call methods.
- **MethodBinding** is an object that can be used to call an arbitrary public method, on an instance that is acquired by evaluating the leading portion of a method binding expression via a value binding.
- It is legal for a class to have two or more methods with the same name.
- Java has to be able to uniquely associate the invocation of a method with its definition relying on the number and types of arguments.
- Therefore the same-named methods must be distinguished:

  1) by the number of arguments, or

  2) by the types of arguments
- Overloading and inheritance are two ways to implement polymorphism.

Method Overriding.

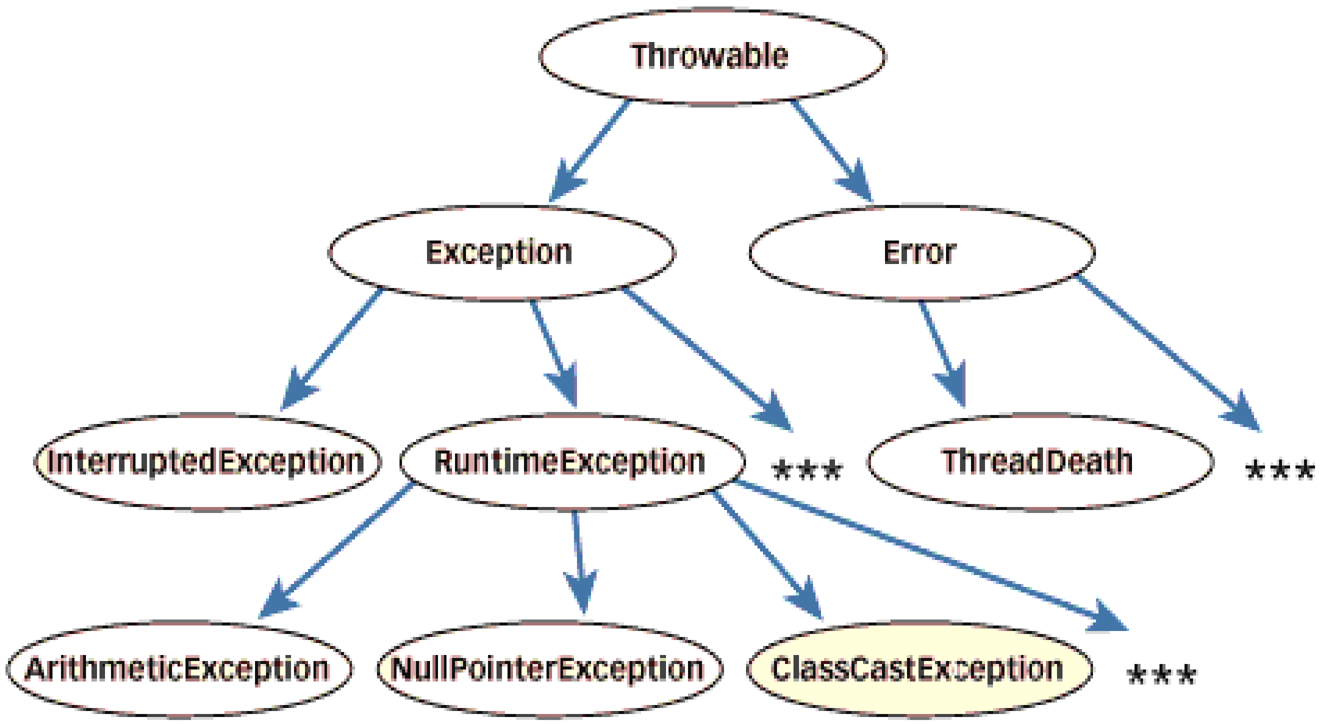
- There may be some occasions when we want an object to respond to the same method but have different behavior when that method is called.
- That means, we should override the method defined in the super class. This is possible by defining a method in a sub class that has the same name, same arguments and same return type as a method in the super class.
- Then when that method is called, the method defined in the sub class is invoked and executed instead of the one in the super class. This is known as overriding.

Exceptions in Java

- Exception is an abnormal condition that arises in the code sequence.
- Exceptions occur during compile time or run time.
- “throwable” is the super class in exception hierarchy.
- Compile time errors occurs due to incorrect syntax.
- Run-time errors happen when
  - User enters incorrect input
  - Resource is not available (ex. file)
  - Logic error (bug) that was not fixed

Exception classes

- In Java, exceptions are objects. When you throw an exception, you throw an object. You can't throw just any object as an exception, however -- only those objects whose classes descend from Throwable.
- Throwable serves as the base class for an entire family of classes, declared in java.lang, that your program can instantiate and throw.
- Throwable has two direct subclasses, Exception and Error.
- Exceptions are thrown to signal abnormal conditions that can often be handled by some catcher, though it's possible they may not be caught and therefore could result in a dead thread.
- Errors are usually thrown for more serious problems, such as OutOfMemoryError, that may not be so easy to handle. In general, code you write should throw only exceptions, not errors.
- Errors are usually thrown by the methods of the Java API, or by the Java virtual machine itself.


Throwable
Exception
Error
InterruptedException
RuntimeException
***
ThreadDeath
***
ArithmeticException
NullPointerException
ClassCastException
***

Summary of OOPS

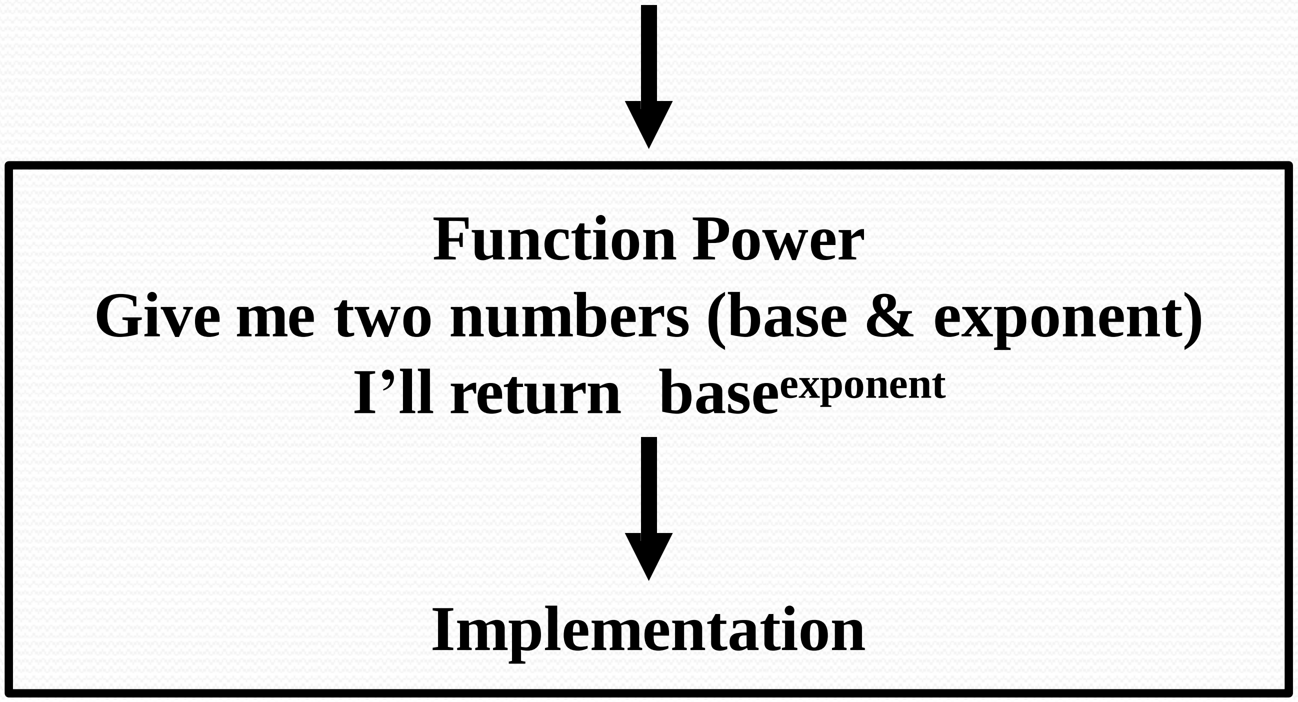
The following are the basic oops concepts: They are as follows:

1. Objects.
2. Classes.
3. Data Abstraction.
4. Data Encapsulation.
5. Inheritance.
6. Polymorphism.
7. Dynamic Binding.
8. Message Passing.

Abstraction in Object-Oriented Programming

Procedural Abstraction

- Procedural Abstractions organize instructions.



Function Power
Give me two numbers (base & exponent)
I'll return $\text{base}^{\text{exponent}}$

Implementation

Data Abstraction

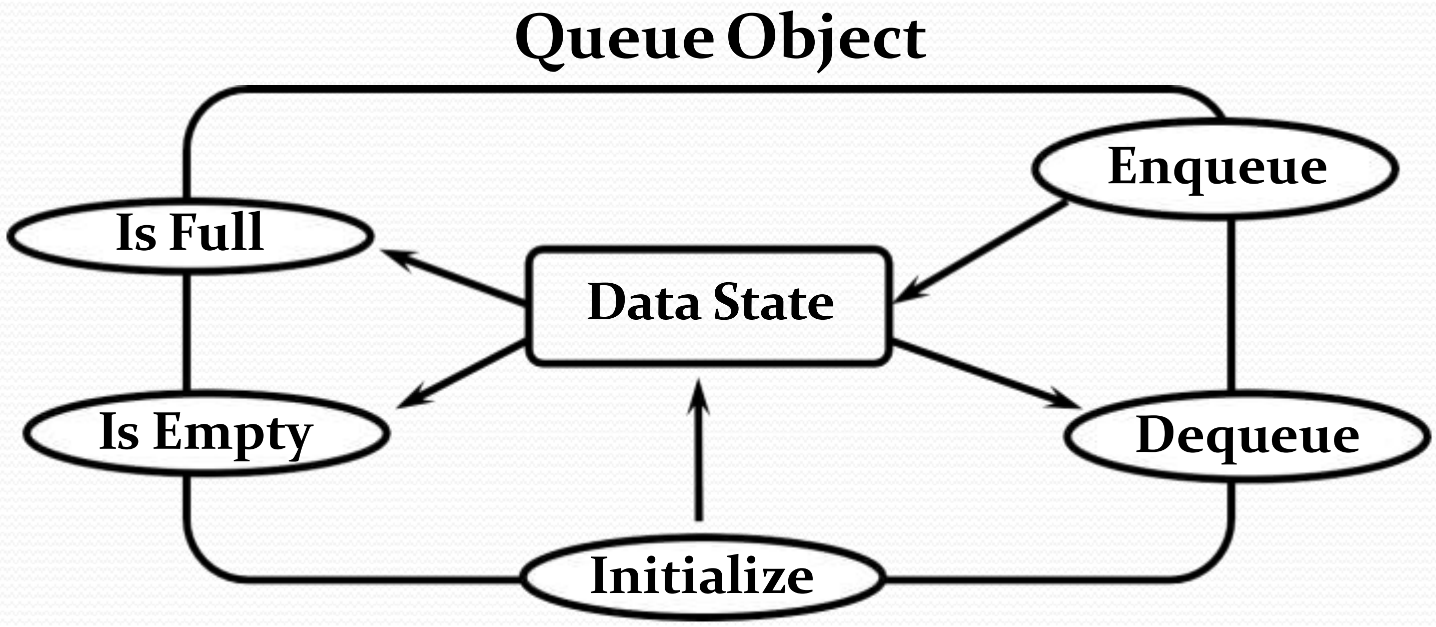
- Data Abstractions organize data.

StudentType

Name (string)
Marks (num)
Grade (char)
Student Number (num)

Behavioral Abstraction

- Behavioral Abstractions combine procedural and data abstractions.



Java History

- Computer language innovation and development occurs for two fundamental reasons:

  1) to adapt to changing environments and uses

  2)to implement improvements in the art of programming
- The development of Java was driven by both in equal measures.
- Many Java features are inherited from the earlier languages:

  B → C → C++ → Java

Before Java: C

- Designed by Dennis Ritchie in 1970s.
- Before C: BASIC, COBOL, FORTRAN, PASCAL
- C- structured, efficient, high-level language that could replace assembly code when creating systems programs.
- Designed, implemented and tested by programmers.

Before Java: C++

- Designed by Bjarne Stroustrup in 1979.
- Response to the increased complexity of programs and respective improvements in the programming paradigms and methods:
  1) assembler languages
  2) high-level languages
  3) structured programming
  4) object-oriented programming (OOP)
- OOP – methodology that helps organize complex programs through the use of inheritance, encapsulation and polymorphism.
- C++ extends C by adding object-oriented features.

Java: History

- In 1990, Sun Microsystems started a project called Green.
- Objective: to develop software for consumer electronics.
- Project was assigned to James Gosling, a veteran of classic network software design. Others included Patrick Naughton, ChrisWarth, Ed Frank, and Mike Sheridan.
- The team started writing programs in C++ for embedding into
  - – toasters
  - – washing machines
  - – VCR's
- Aim was to make these appliances more "intelligent".

Java: History (contd.)

- C++ is powerful, but also dangerous. The power and popularity of C derived from the extensive use of pointers. However, any incorrect use of pointers can cause memory leaks, leading the program to crash.
- In a complex program, such memory leaks are often hard to detect.
- Robustness is essential. Users have come to expect that Windows may crash or that a program running under Windows may crash. ("This program has performed an illegal operation and will be shut down")
- However, users do not expect toasters to crash, or washing machines to crash.
- A design for consumer electronics has to be *robust*.
- Replacing pointers by references, and automating memory management was the proposed solution.

Java: History (contd.)

- Hence, the team built a new programming language called Oak, which avoided potentially dangerous constructs in C++, such as pointers, pointer arithmetic, operator overloading etc.
- Introduced automatic memory management, freeing the programmer to concentrate on other things.
- Architecture neutrality (Platform independence)
- Many different CPU's are used as controllers. Hardware chips are evolving rapidly. As better chips become available, older chips become obsolete and their production is stopped. Manufacturers of toasters and washing machines would like to use the chips available off the shelf, and would not like to reinvest in compiler development every two-three years.
- So, the software and programming language had to be *architecture neutral*.

Java: History (contd)

- It was soon realized that these design goals of consumer electronics perfectly suited an ideal programming language for the Internet and WWW, which should be:
  - object-oriented (& support GUI)
  - – robust
  - – architecture neutral
- Internet programming presented a BIG business opportunity. Much bigger than programming for consumer electronics.
- Java was "re-targeted" for the Internet
- The team was expanded to include Bill Joy (developer of Unix), Arthur van Hoff, Jonathan Payne, Frank Yellin, Tim Lindholm etc.
- In 1994, an early web browser called WebRunner was written in Oak. WebRunner was later renamed HotJava.
- In 1995, Oak was renamed Java.
- A common story is that the name Java relates to the place from where the development team got its coffee. The name Java survived the trade mark search.

Java History

- Designed by James Gosling, Patrick Naughton, Chris Warth, Ed Frank and Mike Sheridan at Sun Microsystems in 1991.
- The original motivation is not Internet: platform-independent software embedded in consumer electronics devices.
- With Internet, the urgent need appeared to break the fortified positions of Intel, Macintosh and Unix programmer communities.
- Java as an “Internet version of C++”? No.
- Java was not designed to replace C++, but to solve a different set of problems.

The Java Buzzwords

- The key considerations were summed up by the Java team in the following list of buzzwords:
  - Simple
  - Secure
  - Portable
  - Object-oriented
  - Robust
  - Multithreaded
  - Architecture-neutral
  - Interpreted
  - High performance
  - Distributed
  - Dynamic

- **simple** – Java is designed to be easy for the professional programmer to learn and use.
- **object-oriented:** a clean, usable, pragmatic approach to objects, not restricted by the need for compatibility with other languages.
- **Robust:** restricts the programmer to find the mistakes early, performs compile-time (strong typing) and run-time (exception-handling) checks, manages memory automatically.
- **Multithreaded:** supports multi-threaded programming for writing program that perform concurrent computations

- **Architecture-neutral:** Java Virtual Machine provides a platform independent environment for the execution of Java byte code
- **Interpreted and high-performance:** Java programs are compiled into an intermediate representation – byte code:

  a) can be later interpreted by any JVM

  b)can be also translated into the native machine code for efficiency.

- **Distributed:** Java handles TCP/IP protocols, accessing a resource through its URL much like accessing a local file.
- **Dynamic:** substantial amounts of run-time type information to verify and resolve access to objects at run-time.
- **Secure:** programs are confined to the Java execution environment and cannot access other parts of the computer.

- **Portability:** Many types of computers and operating systems are in use throughout the world—and many are connected to the Internet.
- For programs to be dynamically downloaded to all the various types of platforms connected to the Internet, some means of generating portable executable code is needed. The same mechanism that helps ensure security also helps create portability.
- Indeed, Java's solution to these two problems is both elegant and efficient.

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

- 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.

- **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.

- **long:** 64-bit integer type.
  Range: -9223372036854775808 to 9223372036854775807.
  Example: long l = 100000000000000000;
  Usage: 1) 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

- **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

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.

- **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

- 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 looses its value when the block is left.
- Initialized in a block, a variable will be re-initialized with every re-entry. Variables lifetime is confined to its scope!

Arrays

- 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 run-time 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 Types

- Java operators are used to build value expressions.
- Java provides a rich set of operators:

    1) assignment

    2) arithmetic

    3) relational

    4) logical

    5) bitwise

Arithmetic assignments

+=	v += expr;	v = v + expr ;
-=	v -=expr;	v = v - expr ;
*=	v *= expr;	v = v * expr ;
/=	v /= expr;	v = v / expr ;
%=	v %= expr;	v = v % expr ;

Basic Arithmetic Operators

+	op1 + op2	ADD
-	op1 - op2	SUBSTRACT
*	op1 * op2	MULTIPLY
/	op1 / op2	DIVISION
%	op1 % op2	REMAINDER

Relational operator

==	Equals to	Apply to any type
!=	Not equals to	Apply to any type
>	Greater than	Apply to numerical type
<	Less than	Apply to numerical type
>=	Greater than or equal	Apply to numerical type
<=	Less than or equal	Apply to numerical type

Logical operators

&	op1 & op2	Logical AND
|	op1 | op2	Logical OR
&&	op1 && op2	Short-circuit AND
||	op1 || op2	Short-circuit OR
!	! op	Logical NOT
^	op1 ^ op2	Logical XOR

Bit wise operators

~	~op	Inverts all bits
&	op1 & op2	Produces 1 bit if both operands are 1
|	op1 |op2	Produces 1 bit if either operand is 1
^	op1 ^ op2	Produces 1 bit if exactly one operand is 1
>>	op1 >> op2	Shifts all bits in op1 right by the value of op2
<<	op1 << op2	Shifts all bits in op1 left by the value of op2

Expressions

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

int **number = 0**;

anArray[0] = 100;

System.out.println (**"Element 1 at index 0: " + anArray[0]**);

int **result = 1 + 2**; // result is now 3 if(**value1 == value2**)

System.out.println(**"value1 == value2"**);

Expressions

- The data type of the value returned by an expression depends on the elements used in the expression.
- The expression number = 0 returns an int because the assignment operator returns a value of the same data type as its left-hand operand; in this case, number is an int.
- As you can see from the other expressions, an expression can return other types of values as well, such as boolean or String. The Java programming language allows you to construct compound expressions from various smaller expressions as long as the data type required by one part of the expression matches the data type of the other.
- Here's an example of a compound expression: 1 * 2 * 3

Control Statements

- Java control statements cause the flow of execution to advance and branch based on the changes to the state of the program.
- Control statements are divided into three groups:
- 1) selection statements allow the program to choose different parts of the execution based on the outcome of an expression
- 2) iteration statements enable program execution to repeat one or more statements
- 3) jump statements enable your program to execute in a non-linear fashion

Selection Statements

- Java selection statements allow to control the flow of program's execution based upon conditions known only during run-time.
- Java provides four selection statements:

  1) if

  2) if-else

  3) if-else-if

  4) switch

Iteration Statements

- Java iteration statements enable repeated execution of part of a program until a certain termination condition becomes true.
- Java provides three iteration statements:

  1) while

  2) do-while

  3) for

Jump Statements

- Java jump statements enable transfer of control to other parts of program.
- Java provides three jump statements:

  1) break

  2) continue

  3) return
- In addition, Java supports exception handling that can also alter the control flow of a program.

Type Conversion

- Size Direction of Data Type
  - Widening Type Conversion (Casting down)
    - Smaller Data Type → Larger Data Type
  - Narrowing Type Conversion (Casting up)
    - Larger Data Type → Smaller Data Type
- Conversion done in two ways
  - Implicit type conversion
    - Carried out by compiler automatically
  - Explicit type conversion
    - Carried out by programmer using casting

Type Conversion

- Widening Type Converstion
  - Implicit conversion by compiler automatically

byte -> short, int, long, float, double
short -> int, long, float, double
char -> int, long, float, double
int -> long, float, double
long -> float, double
float -> double

Type Conversion

- Narrowing Type Conversion
  - Programmer should describe the conversion explicitly

byte -> char
short -> byte, char
char -> byte, short
int -> byte, short, char
long -> byte, short, char, int
float -> byte, short, char, int, long
double -> byte, short, char, int, long, float

Type Conversion

- byte and short are always promoted to int
- if one operand is long, the whole expression is promoted to long
- if one operand is float, the entire expression is promoted to float
- if any operand is double, the result is double

Type Casting

- General form: (targetType) value
- Examples:
- 1) integer value will be reduced module bytes range:

```
        int i;
        byte b = (byte) i;
```

- 2) floating-point value will be truncated to integer value:

```
 float f;
 int i = (int) f;
```

Simple Java Program

- A class to display a simple message:

```
class MyProgram
        {
        public static void main(String[] args)
         {
System.out.println(“First Java program.");
        }
}
```

What is an Object?

- Real world objects are things that have:

  1) state

  2)behavior

  Example: your dog:

- state – name, color, breed, sits?, barks?, wages tail?, runs?
- behavior – sitting, barking, waging tail, running
- A software object is a bundle of variables (state) and methods (operations).

What is a Class?

- A class is a blueprint that defines the variables and methods common to all objects of a certain kind.
- Example: ‘your dog’ is a object of the class Dog.
- An object holds values for the variables defines in the class.
- An object is called an instance of the Class

Object Creation

- A variable is declared to refer to the objects of type/class String:

```
String s;
```

- The value of s is null; it does not yet refer to any object.
- A new String object is created in memory with initial “abc” value:
- String s = new String(“abc”);
- Now s contains the address of this new object.

Object Destruction

- A program accumulates memory through its execution.
- Two mechanism to free memory that is no longer need by the program:

    1) manual – done in C/C++

    2) automatic – done in Java
- In Java, when an object is no longer accessible through any variable, it is eventually removed from the memory by the garbage collector.
- Garbage collector is parts of the Java Run-Time Environment.

Class

- A basis for the Java language.
- Each concept we wish to describe in Java must be included inside a class.
- A class defines a new data type, whose values are objects:
- A class is a template for objects
- An object is an instance of a class

Class Definition

- A class contains a name, several variable declarations (instance variables) and several method declarations. All are called members of the class.
- General form of a class:

```
class classname {
        type instance-variable-1;
        …
        type instance-variable-n;
        type method-name-1(parameter-list) { … }
        type method-name-2(parameter-list) { … }
        …
        type method-name-m(parameter-list) { … }
    }
```

Example: Class Usage

```
class Box {
double width;
double height;
double depth;
}
class BoxDemo {
public static void main(String args[]) {
Box mybox = new Box();
double vol;
mybox.width = 10;
mybox.height = 20;
mybox.depth = 15;
vol = mybox.width * mybox.height * mybox.depth;
System.out.println ("Volume is " + vol);
} }
```

Constructor

- A constructor initializes the instance variables of an object.
- It is called immediately after the object is created but before the new operator completes.

  1) it is syntactically similar to a method:

  2) it has the same name as the name of its class

  3)it is written without return type; the default return type of a class
- constructor is the same classWhen the class has no constructor, the default constructor automatically initializes all its instance variables with zero.

Example: Constructor

```
class Box {
double width;
double height;
double depth;
Box() {
System.out.println("Constructing Box");
width = 10; height = 10; depth = 10;
}
double volume() {
return width * height * depth;
}
}
```

Parameterized Constructor

```
class Box {
double width;
double height;
double depth;
Box(double w, double h, double d) {
width = w; height = h; depth = d;
}
double volume()
 { return width * height * depth;
}
}
```

Methods

- General form of a method definition:
  type name(parameter-list) {

```
                              … return value;
                             …
                           }
```

- Components:

  1)type - type of values returned by the method. If a method does not return any value, its return type must be void.
  2) name is the name of the method
  3) parameter-list is a sequence of type-identifier lists separated by commas
  4) return value indicates what value is returned by the method.

Example: Method

- Classes declare methods to hide their internal data structures, as well as for their own internal use: Within a class, we can refer directly to its member variables:

```
class Box {
double width, height, depth;
void volume() {
System.out.print("Volume is ");
System.out.println(width * height * depth);
}
}
```

Parameterized Method

- Parameters increase generality and applicability of a method:
- 1) method without parameters

```
int square() { return 10*10; }
```

- 2) method with parameters

```
int square(int i) { return i*i; }
```

- Parameter: a variable receiving value at the time the method is invoked.
- Argument: a value passed to the method when it is invoked.

Access Control: Data Hiding and Encapsulation

- Java provides control over the *visibility* of variables and methods.
- *Encapsulation,* safely sealing data within the *capsule* of the class Prevents programmers from relying on details of class implementation, so you can update without worry
- Helps in protecting against accidental or wrong usage.
- Keeps code elegant and clean (easier to maintain)

Access Modifiers: Public, Private, Protected

- *Public:* keyword applied to a class, makes it available/visible everywhere. Applied to a method or variable, completely visible.
- Default(No visibility modifier is specified): it behaves like public in its package and private in other packages.
- *Default Public* keyword applied to a class, makes it available/visible everywhere. Applied to a method or variable, completely visible.

- *Private* fields or methods for a class only visible within that class. Private members are *not* visible within subclasses, and are *not* inherited.
- *Protected* members of a class are visible within the class, subclasses and *also* within all classes that are in the same package as that class.

Visibility

```
public class Circle {
    private double x,y,r;

        // Constructor
     public Circle (double x, double y, double r) {
            this.x = x;
            this.y = y;
            this.r = r;
     }
     //Methods to return circumference and area
    public double circumference() { return 2*3.14*r;}
    public double area() { return 3.14 * r * r; }
}
```

String Handling

- **String** is probably the most commonly used class in Java's class library. The obvious reason for this is that strings are a very important part of programming.
- The first thing to understand about strings is that every string you create is actually an object of type **String**. Even string constants are actually **String** objects.
- For example, in the statement

  System.out.println("This is a String, too");

  the string "This is a String, too" is a **String** constant

- Java defines one operator for **String** objects: +.
- It is used to concatenate two strings. For example, this statement
- String myString = "I" + " like " + "Java.";

  results in **myString** containing

  "I like Java."

- The **String** class contains several methods that you can use. Here are a few. You can
- test two strings for equality by using

  equals( ). You can obtain the length of a string by calling the **length( )** method. You can obtain the character at a specified index within a string by calling **charAt( )**. The general forms of these three methods are shown here:
- // Demonstrating some String methods.

```
class StringDemo2 {
        public static void main(String args[]) {
                String strOb1 = "First String";
                String strOb2 = "Second String";
                String strOb3 = strOb1;
        System.out.println("Length of strOb1: " +
                                        strOb1.length());
System.out.println ("Char at index 3 in strOb1: " +
strOb1.charAt(3));
if(strOb1.equals(strOb2))

System.out.println("strOb1 == strOb2");
else
System.out.println("strOb1 != strOb2");
if(strOb1.equals(strOb3))
System.out.println("strOb1 == strOb3");
else
System.out.println("strOb1 != strOb3");
} }
```

This program generates the following output:

```
Length of strOb1: 12
Char at index 3 in strOb1: s
strOb1 != strOb2
strOb1 == strOb3
```

UNIT-2

TOPICS

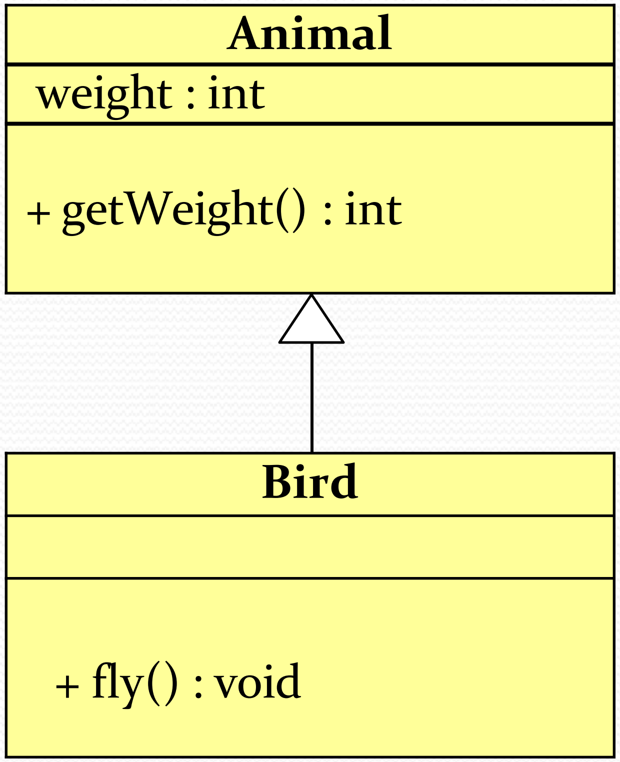
1 Hierarchical abstractions Base class object.
2 subclass, subtype, substitutability.
3 forms of inheritance- specialization,
4 construction, extension, limitation, combination.
5 Benefits of inheritance, costs of inheritance.
6 Member access rules, super uses, using final with inheritance.
7 polymorphism- method overriding, abstract classes.
1 Defining, Creating and Accessing a Package
2 Importing packages
3 Differences between classes and interfaces
4 Defining an interface
5 Implementing interface
6 Applying interfaces
7 variables in interface and extending interfaces

Inheritance

- Methods allows a software developer to reuse a sequence of statements
- *Inheritance* allows a software developer to reuse classes by deriving a new class from an existing one
- The existing class is called the *parent class*, or *superclass*, or *base class*
- The derived class is called the *child class* or *subclass*.
- As the name implies, the child inherits characteristics of the parent
- That is, the child class *inherits* the methods and data defined for the parent class

Inheritance

- Inheritance relationships are often shown graphically in a *class diagram*, with the arrow pointing to the parent class



Inheritance should create an *is-a relationship*, meaning the child *is a* more specific version of the parent

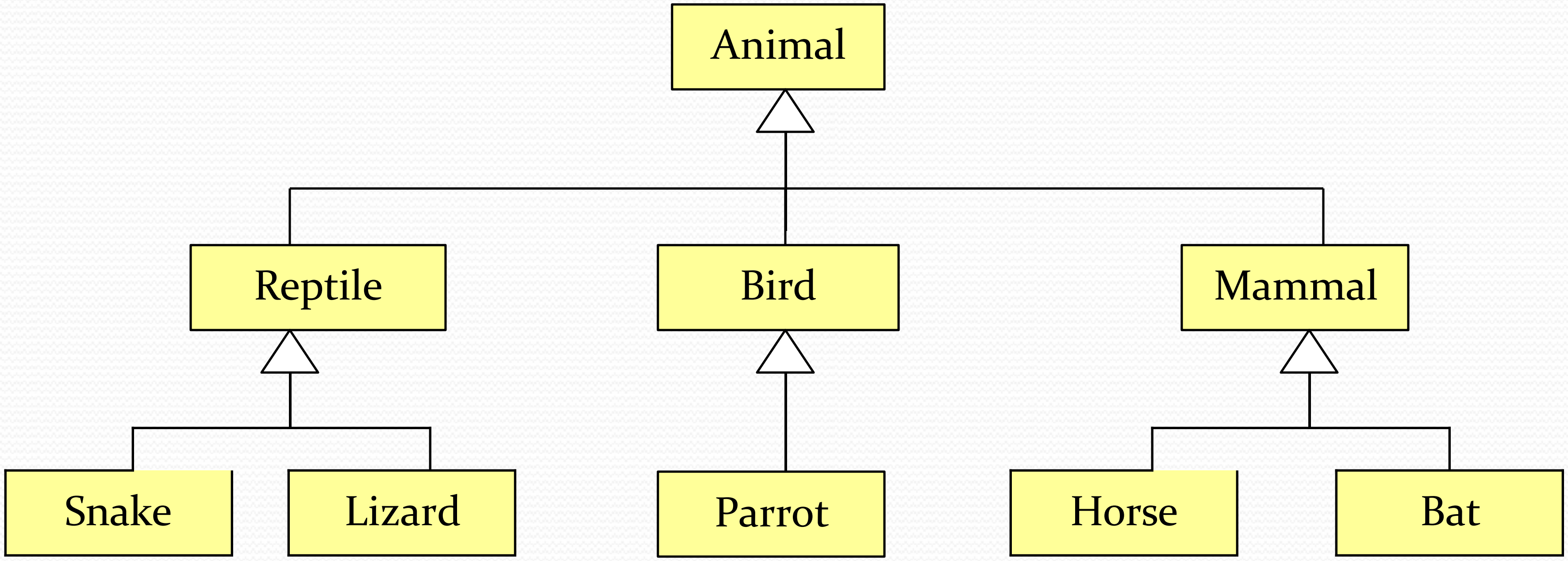
Deriving Subclasses

- In Java, we use the reserved word `extends` to establish an inheritance relationship

```
class Animal
{
  // class contents
   int weight;
   public void int getWeight() {…}
}

class Bird extends Animal
{
   // class contents
   public void fly() {…};
}
```

Class Hierarchy

- A child class of one parent can be the parent of another child, forming *class hierarchies*



- At the top of the hierarchy there's a default class called *Object*.

Class Hierarchy

- Good class design puts all common features as high in the hierarchy as reasonable
- inheritance is transitive
  - An instance of class Parrot is also an instance of Bird, an instance of Animal, …, and an instance of class Object
- The class hierarchy determines how methods are executed:
  - Previously, we took the simplified view that when variable v is an instance of class C, then a procedure call v.proc1() invokes the method proc1() defined in class C
  - However, if C is a child of some superclass C’ (and hence v is both an instance of C *and an instance* of C’), the picture becomes more complex, because methods of class C can *override* the methods of class C’ (next two slides).

Defining Methods in the Child Class: Overriding by Replacement

- A child class can *override* the definition of an inherited method in favor of its own
  - that is, a child can redefine a method that it inherits from its parent
  - the new method must have the same signature as the parent's method, but can have different code in the body

- In java, all methods except of constructors override the methods of their ancestor class by replacement. E.g.:
  - the Animal class has method eat()
  - the Bird class has method eat() and Bird extends Animal
  - variable *b* is of class Bird, i.e. Bird b = ...
  - b.eat() simply invokes the eat() method of the Bird class

- If a method is declared with the `final` modifier, it cannot be overridden

Defining Methods in the Child Class: Overriding by Refinement

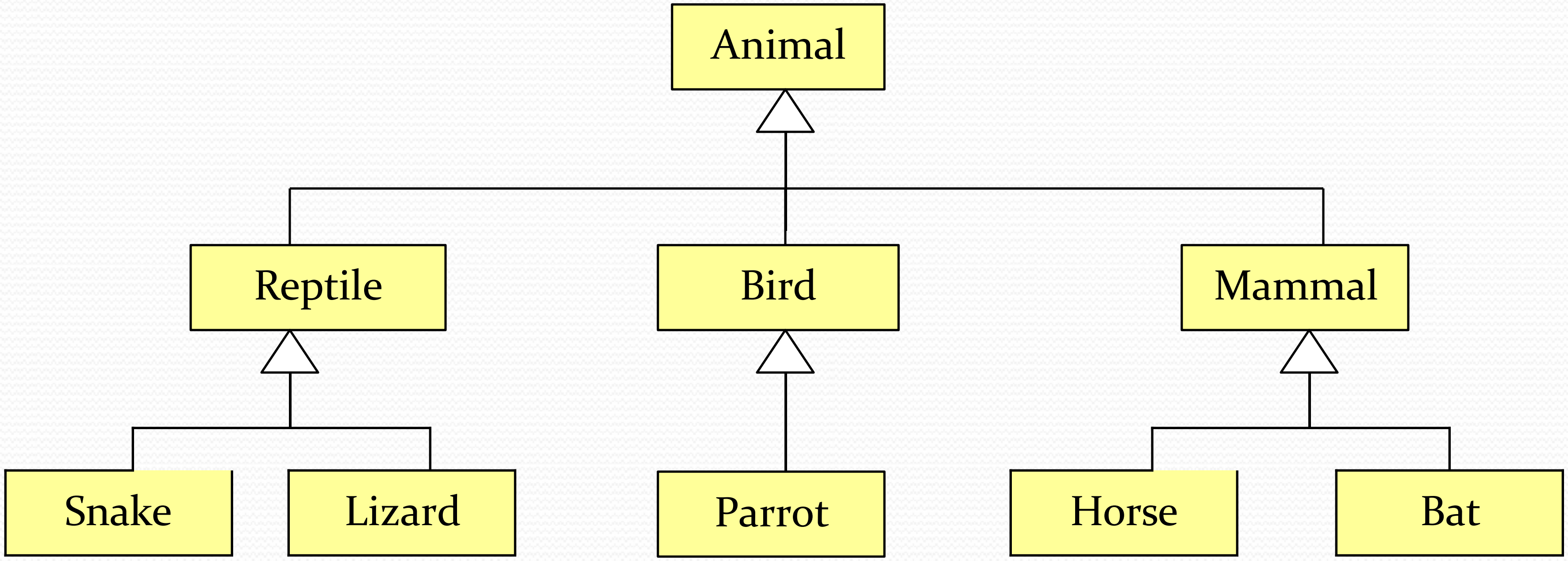
- Constructors in a subclass *override* the definition of an inherited constructor method by *refining* them (instead of replacing them)
  - Assume class Animal has constructors
    Animal(), Animal(int weight), Animal(int weight, int livespan)
  - Assume class Bird which extends Animal has constructors
    Bird(), Bird(int weight), Bird(int weight, int livespan)

  - Let's say we create a Bird object, e.g. Bird b = Bird(5)
  - This will invoke ***first*** the constructor of the Animal (the superclass of Bird) and ***then*** the constructor of the Bird

- This is called *constructor chaining*: If class C0 extends C1 and C1 extends C2 and … Cn-1 extends Cn = Object then when creating an instance of object C0 first constructor of Cn is invoked, then constructors of Cn-1, …, C2, C1, and finally the constructor of C
  - The constructors (in each case) are chosen by their signature, e.g. (), (int), etc…
    - If no constructor with matching signature is found in any of the class Ci for i>0 then the default constructor is executed for that class
    - If no constructor with matching signature is found in the class C0 then this causes a compiler errorFirst the new method must have the same signature as the parent's method, but can have different code in the body

Recap: Class Hierarchy

- In Java, a **class can extend a single other class** (If none is stated then it implicitly extends an Object class)



- Imagine what would happen to method handling rules if every class could extend two others…

  (Answer: It would create multiple problems!)

Hierarchical Abstraction

- An essential element of object-oriented programming is *abstraction.*

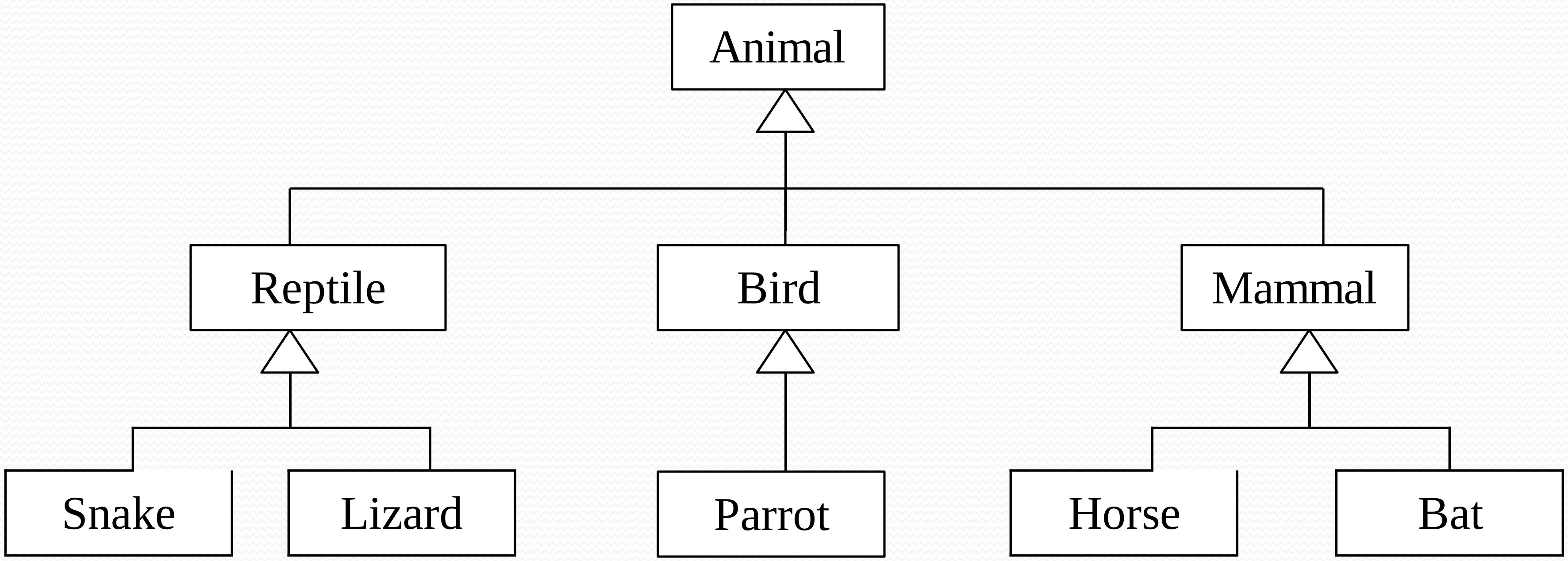
- Humans manage complexity through abstraction. For example, people do not think of a car as a set of tens of thousands of individual parts. They think of it as a well-defined object with its own unique behavior.

- This abstraction allows people to use a car without being overwhelmed by the complexity of the parts that form the car. They can ignore the details of how the engine, transmission, and braking systems work.

- Instead they are free to utilize the object as a whole.

Class Hierarchy

- A child class of one parent can be the parent of another child, forming *class hierarchies*



  - At the top of the hierarchy there's a default class called *Object*.

Class Hierarchy

- Good class design puts all common features as high in the hierarchy as reasonable
- The class hierarchy determines how methods are executed
- inheritance is transitive
  - An instance of class Parrot is also an instance of Bird, an instance of Animal, …, and an instance of class Object

Base Class **Object**

- In Java, all classes use inheritance.
- If no parent class is specified explicitly, the base class **Object** is implicitly inherited.
- All classes defined in Java, is a child of **Object** class, which provides minimal functionality guaranteed to e common to all objects.
- Methods defined in Object class are;

1. equals(Object obj): Determine whether the argument object is the same as the receiver.
2. getClass(): Returns the class of the receiver, an object of type Class.
3. hashCode(): Returns a hash value for this object. Should be overridden when the equals method is changed.
4. toString(): Converts object into a string value. This method is also often overridden.

Base class

1) a class obtains variables and methods from another class
2) the former is called subclass, the latter super-class (Base class)
3) a sub-class provides a specialized behavior with respect to its super-class
4) inheritance facilitates code reuse and avoids duplication of data

Extends

- Is a keyword used to inherit a class from another class
- Allows to extend from only one

```
class One
{          int a=5;
}
```

```
class Two extends One
{
          int b=10;
}
```

Subclass, Subtype and Substitutability

- A subtype is a class that satisfies the principle of substitutability.
- A subclass is something constructed using inheritance, whether or not it satisfies the principle of substitutability.
- The two concepts are independent. Not all subclasses are subtypes, and (at least in some languages) you can construct subtypes that are not subclasses.
- Substitutability is fundamental to many of the powerful software development techniques in OOP.
- The idea is that, declared a variable in one type may hold the value of different type.
- Substitutability can occur through use of inheritance, whether using **extends**, or using **implements** keywords.

Subclass, Subtype, and Substitutability

When new classes are constructed using inheritance, the argument used to justify the validity of substitutability is as follows;

- Instances of the subclass must possess all data fields associated with its parent class.
- Instances of the subclass must implement, through inheritance at least, all functionality defined for parent class. (Defining new methods is not important for the argument.)
- Thus, an instance of a child class can mimic the behavior of the parent class and should be *indistinguishable* from an instance of parent class if substituted in a similar situation.

Subclass, Subtype, and Substitutability

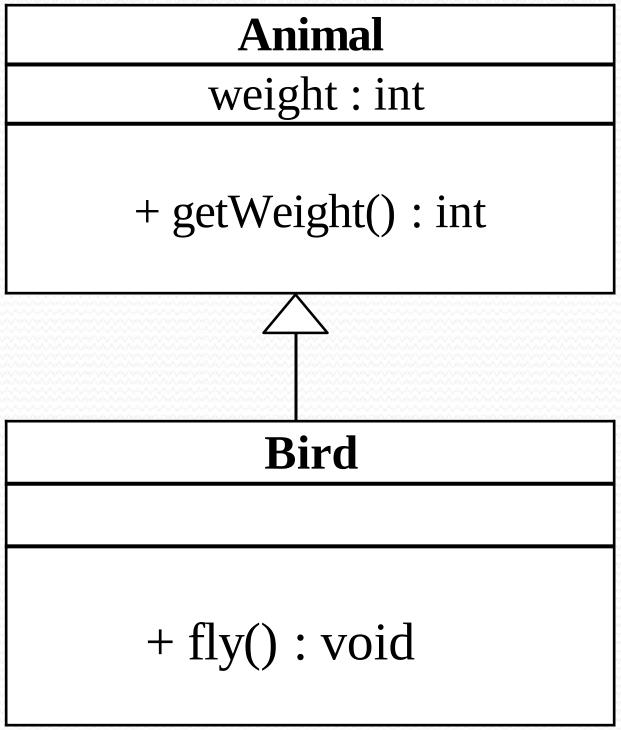
- The term *subtype* is used to describe the relationship between types that explicitly recognizes the principle of *substitution*. A type B is considered to be a subtype of A if an instances of B can legally be assigned to a variable declared as of type A.
- The term *subclass* refers to inheritance mechanism made by extends keyword.
- Not all *subclasses* are *subtypes*. *Subtypes* can also be formed using *interface*, linking types that have no inheritance relationship.

Subclass

- Methods allows to reuse a sequence of statements
- *Inheritance* allows to reuse classes by deriving a new class from an existing one
- The existing class is called the *parent class,* or *superclass*, or *base class*
- The derived class is called the *child class* or *subclass*.
- As the name implies, the child inherits characteristics of the parent(i.e the child class *inherits* the methods and data defined for the parent class

Subtype

- Inheritance relationships are often shown graphically in a *class diagram*, with the arrow pointing to the parent class



Substitutability (Deriving Subclasses)

- In Java, we use the reserved word extends to establish an inheritance relationship

```
   class Animal
        {
          // class contents
           int weight;
 public void int getWeight() {…}
        }

        class Bird extends Animal
        {
           // class contents
 public void fly() {…};
        }
```

Defining Methods in the Child Class: Overriding by Replacement

- A child class can *override* the definition of an inherited method in favor of its own
  - that is, a child can redefine a method that it inherits from its parent
  - the new method must have the same signature as the parent's method, but can have different code in the body
- In java, all methods except of constructors override the methods of their ancestor class by replacement. E.g.:
  - the Animal class has method eat()
  - the Bird class has method eat() and Bird extends Animal
  - variable *b* is of class Bird, i.e. Bird b = …
  - b.eat() simply invokes the eat() method of the Bird class
- If a method is declared with the `final` modifier, it cannot be overridden

Forms of Inheritance

Inheritance is used in a variety of way and for a variety of different purposes .

- Inheritance for Specialization
- Inheritance for Specification
- Inheritance for Construction
- Inheritance for Extension
- Inheritance for Limitation
- Inheritance for Combination

One or many of these forms may occur in a single case.

Forms of Inheritance

(- *Inheritance for Specialization* -)

Most commonly used inheritance and sub classification is for specialization.

Always creates a subtype, and the principles of substitutability is explicitly upheld.

It is the most ideal form of inheritance.

An example of subclassification for specialization is;

```
public class PinBallGame extends Frame {

// body of class

}
```

Specialization

- By far the most common form of inheritance is for specialization.
  - Child class is a specialized form of parent class
  - Principle of substitutability holds
- A good example is the Java hierarchy of Graphical components in the AWT:
  - Component
    - Label
    - Button
    - TextComponent
      - TextArea
      - TextField
    - CheckBox
    - ScrollBar

Forms of Inheritance
(- ***Inheritance for Specification*** -)

This is another most common use of inheritance. Two different mechanisms are provided by Java, ***interface*** and ***abstract***, to make use of *subclassification for specification*. Subtype is formed and substitutability is explicitly upheld.

Mostly, not used for refinement of its parent class, but instead is used for definitions of the properties provided by its parent.

```
class FireButtonListener implements ActionListener {

// body of class

}

class B extends A {

// class A is defined as abstract specification class

}
```

Specification

- The next most common form of inheritance involves specification. The parent class specifies some behavior, but does not implement the behavior
  - Child class implements the behavior
  - Similar to Java interface or abstract class
  - When parent class does not implement actual behavior but merely defines the behavior that will be implemented in child classes
- Example, Java 1.1 Event Listeners:

  ActionListener, MouseListener, and so on specify behavior, but must be subclassed.

Forms of Inheritance

(- ***Inheritance for Construction*** -)

Child class inherits most of its functionality from parent, but may change the name or parameters of methods inherited from parent class to form its interface.

This type of inheritance is also widely used for code reuse purposes. It simplifies the construction of newly formed abstraction but is not a form of subtype, and often violates substitutability.

Example is ***Stack*** class defined in Java libraries.

Construction

- The parent class is used only for its behavior, the child class has no *is-a* relationship to the parent.
  - Child modify the arguments or names of methods
  -
- An example might be subclassing the idea of a *Set* from an existing *List* class.
  - Child class is not a more specialized form of parent class; no substitutability

Forms of Inheritance (- ***Inheritance for Extension*** -)

Subclassification for extension occurs when a child class only adds new behavior to the parent class and does not modify or alter any of the inherited attributes.

Such subclasses are always subtypes, and substitutability can be used.

Example of this type of inheritance is done in the definition of the class Properties which is an extension of the class HashTable.

Generalization or Extension

- The child class generalizes or extends the parent class by providing more functionality
  - In some sense, opposite of subclassing for specialization
- The child doesn't change anything inherited from the parent, it simply adds new features
  - Often used when we cannot modify existing base parent class
- Example, ColoredWindow inheriting from Window
  - Add additional data fields
  - Override window display methods

Forms of Inheritance (- ***Inheritance for Limitation*** -)

Subclassification for limitation occurs when the behavior of the subclass is smaller or more restrictive that the behavior of its parent class.

Like subclassification for extension, this form of inheritance occurs most frequently when a programmer is building on a base of existing classes.

Is not a subtype, and substitutability is not proper.

Limitation

- The child class limits some of the behavior of the parent class.

- Example, you have an existing List data type, and you want a Stack
- Inherit from List, but override the methods that allow access to elements other than top so as to produce errors.

Forms of Inheritance (- *Inheritance for Combination* -)

This types of inheritance is known as *multiple inheritance* in Object Oriented Programming.

Although the Java does not permit a subclass to be formed be inheritance from more than one parent class, several approximations to the concept are possible.

Example of this type is Hole class defined as;

```
class  Hole extends Ball implements PinBallTarget{

// body of class

}
```

Combimnation

- Two or more classes that seem to be related, but its not clear who should be the parent and who should be the child.
- Example: Mouse and TouchPad and JoyStick
- Better solution, abstract out common parts to new parent class, and use subclassing for specialization.

Summary of Forms of Inheritance

- Specialization. The child class is a special case of the parent class; in other words, the child class is a subtype of the parent class.
- Specification. The parent class defines behavior that is implemented in the child class but not in the parent class.
- Construction. The child class makes use of the behavior provided by the parent class, but is not a subtype of the parent class.
- Generalization. The child class modifies or overrides some of the methods of the parent class.
- Extension. The child class adds new functionality to the parent class, but does not change any inherited behavior.
- Limitation. The child class restricts the use of some of the behavior inherited from the parent class.
- Variance. The child class and parent class are variants of each other, and the class-subclass relationship is arbitrary.
- Combination. The child class inherits features from more than one parent class. This is multiple inheritance and will be the subject of a later chapter.

The Benefits of Inheritance

- Software Reusability (among projects)
- Increased Reliability (resulting from reuse and sharing of well-tested code)
- Code Sharing (within a project)
- Consistency of Interface (among related objects)
- Software Components
- Rapid Prototyping (quickly assemble from pre-existing components)
- Polymorphism and Frameworks (high-level reusable components)
- Information Hiding

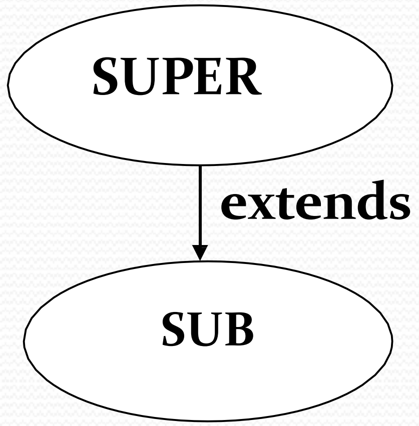
The Costs of Inheritance

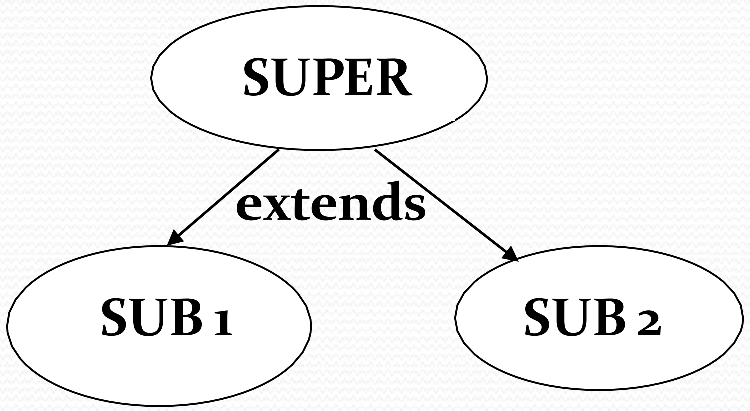
- Execution Speed
- Program Size
- Message-Passing Overhead
- Program Complexity (in overuse of inheritance)

Types of inheritance

- Acquiring the properties of an existing Object into newly creating Object to overcome the re-declaration of properties in deferent classes.
- These are 3 types:

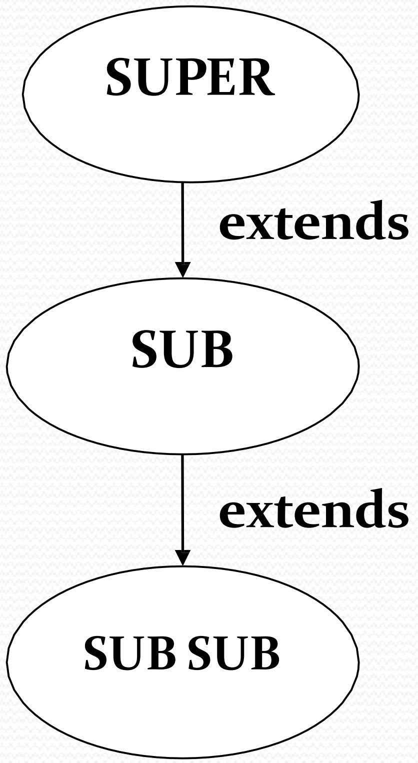
1.Simple Inheritance

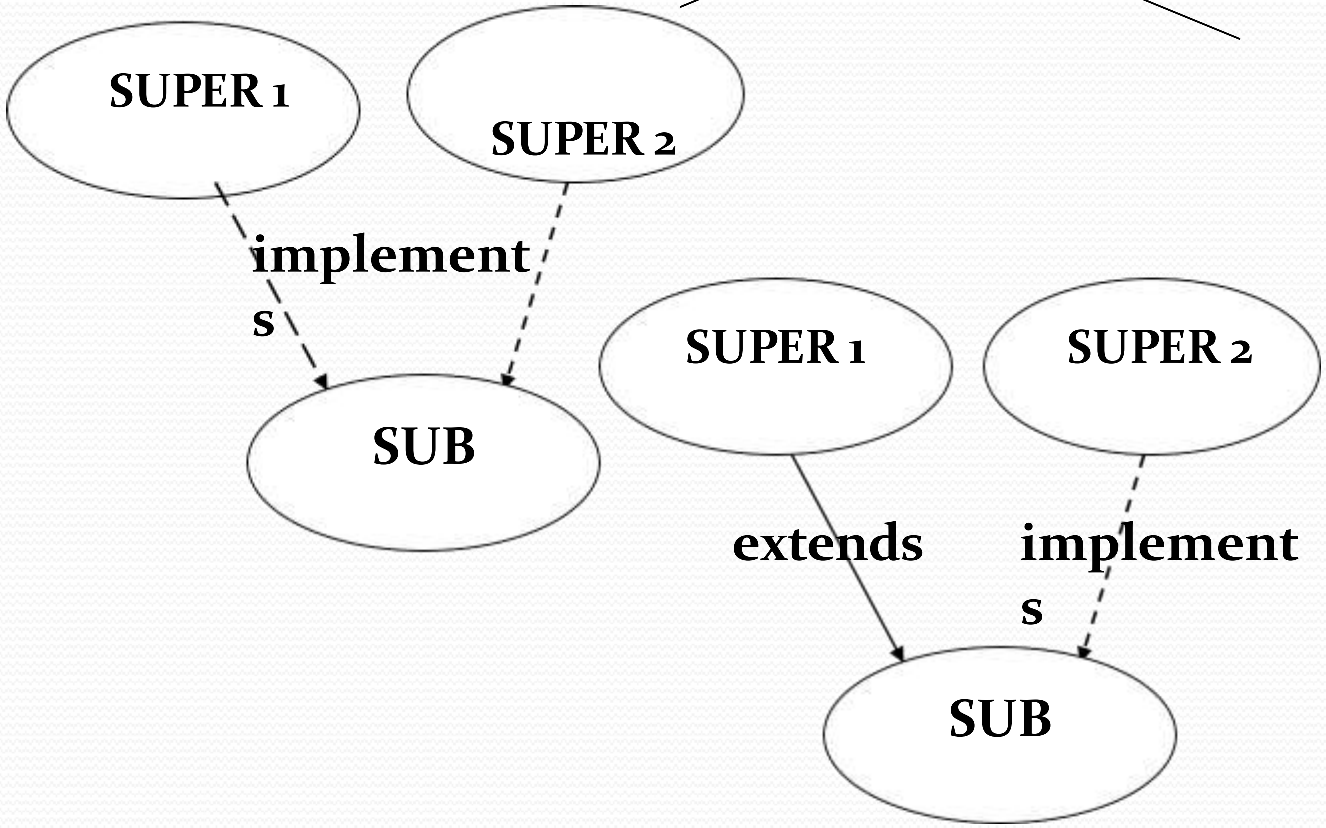




2. Multi Level Inheritance

~~3. Multiple Inheritance~~





Member access rules

- Visibility modifiers determine which class members are accessible and which do not
- Members (variables and methods) declared with public visibility are accessible, and those with private visibility are not
- Problem: How to make class/instance variables visible only to its subclasses?
- Solution: Java provides a third visibility modifier that helps in inheritance situations: protected

Modifiers and Inheritance (cont.)

Visibility Modifiers for class/interface:

`public` : can be accessed from outside the class definition.

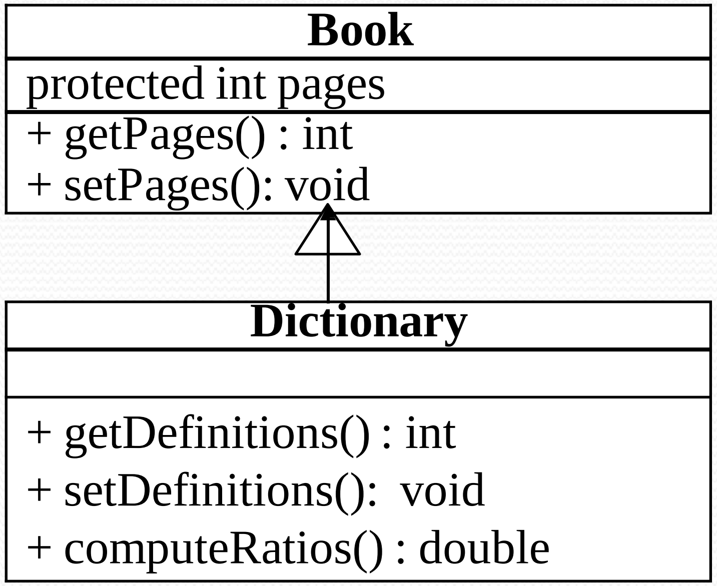
`protected` : can be accessed only within the class definition in which it appears, within other classess in the same package, or within the definition of subclassess.

`private` : can be accessed only within the class definition in which it appears.

default-access (if omitted) features accessible from inside the current Java package

The `protected` Modifier

- The `protected` visibility modifier allows a member of a base class to be accessed in the child
  - `protected` visibility provides more encapsulation than `public` does
  - `protected` visibility is not as tightly encapsulated as `private` visibility



“**super**” uses

- ‘super’ is a keyword used to refer to hidden variables of super class from sub class.
  - super.a=a;

- It is used to call a constructor of super class from constructor of sub class which should be first statement.
  - super(a,b);

- It is used to call a super class method from sub class method to avoid redundancy of code
  - super.addNumbers(a, b);

Super and Hiding

- Why is super needed to access super-class members?
- When a sub-class declares the variables or methods with the same names and types as its super-class:

```
class A {
int i = 1;
}
class B extends A {
int i = 2;
System.out.println(“i is “ + i);
}
```

- The re-declared variables/methods hide those of the super-class.

Example: Super and Hiding

```
class A {
int i;
}
class B extends A {
int i;
B(int a, int b) {
super.i = a; i = b;
}
void show() {
System.out.println("i in superclass: " + super.i);
System.out.println("i in subclass: " + i);
}
}
```

Example: Super and Hiding

- Although the i variable in B hides the i variable in A, super allows access to the hidden variable of the super-class:

```
class UseSuper {
public static void main(String args[]) {
B subOb = new B(1, 2);
subOb.show();
}
}
```

Using final with inheritance

- final keyword is used declare constants which can not change its value of definition.
- final Variables can not change its value.
- final Methods can not be Overridden or Over Loaded
- final Classes can not be extended or inherited

Preventing Overriding with final

- A method declared final cannot be overridden in any sub-class:

```
class A {
final void meth() {
System.out.println("This is a final method.");
}
}
```

This class declaration is illegal:

```
class B extends A {
void meth() {
System.out.println("Illegal!");
}
}
```

Preventing Inheritance with final

- A class declared final cannot be inherited – has no sub-classes.

```
final class A { … }
```

- This class declaration is considered illegal:

```
class B extends A { … }
```

- Declaring a class final implicitly declares all its methods final.
- It is illegal to declare a class as both abstract and final.

Polymorphism

- Polymorphism is one of three pillars of object-orientation.
- Polymorphism: many different (poly) forms of objects that share a common interface respond differently when a method of that interface is invoked:
  1) a super-class defines the common interface
  2) sub-classes have to follow this interface (inheritance), but are also permitted to provide their own implementations (overriding)
- A sub-class provides a specialized behaviors relying on the common elements defined by its super-class.

Polymorphism

- A polymorphic reference can refer to different types of objects at different times
  - In java every reference can be polymorphic except of references to base types and final classes.
- It is the type of the object being referenced, not the reference type, that determines which method is invoked
  - Polymorphic references are therefore resolved at run-time, not during compilation; this is called *dynamic binding*
- Careful use of polymorphic references can lead to elegant, robust software designs

Method Overriding

- When a method of a sub-class has the same name and type as a method of the super-class, we say that this method is overridden.

- When an overridden method is called from within the sub-class:

  1) it will always refer to the sub-class method

  2) super-class method is hidden

Example: Hiding with Overriding 1

```
class A {
int i, j;
A(int a, int b) {
i = a; j = b;
}
void show() {
System.out.println("i and j: " + i + " " + j);
}
}
```

Example: Hiding with Overriding 2

```
class B extends A {
int k;
B(int a, int b, int c) {
super(a, b);
k = c;
}
void show() {
System.out.println("k: " + k);
}
}
```

Example: Hiding with Overriding 3

- When show() is invoked on an object of type B, the version of show() defined in B is used:

```
class Override {
public static void main(String args[]) {
B subOb = new B(1, 2, 3);
subOb.show();
}
}
```

- The version of show() in A is hidden through overriding.

Overloading vs. Overriding

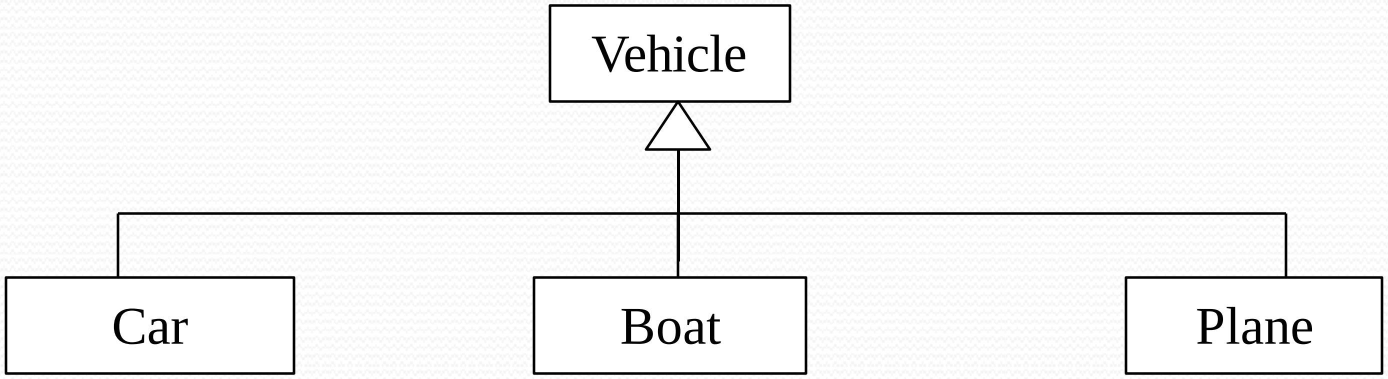
● Overloading deals with multiple methods in the same class with the same name but different signatures	● Overriding deals with two methods, one in a parent class and one in a child class, that have the same signature
● Overloading lets you define a similar operation in different ways for different data	o Overriding lets you define a similar operation in different ways for different object types

Abstract Classes

- Java allows abstract classes
  - use the modifier `abstract` on a class header to declare an abstract class

```
abstract class Vehicle
{ … }
```

- An abstract class is a placeholder in a class hierarchy that represents a generic concept



Abstract Class: Example

- An abstract class often contains *abstract methods*, though it doesn't have to
  - Abstract methods consist of only methods *declarations*, without any method body

```
public abstract class Vehicle
{
    String name;
     public String getName()
           { return name; } \\ method body

     abstract public void move();
                                \\ no body!
}
```

Abstract Classes

- An abstract class often contains *abstract methods*, though it doesn't have to
  - Abstract methods consist of only methods *declarations,* without any method body
- The non-abstract child of an abstract class must override the abstract methods of the parent
- An abstract class cannot be instantiated
- The use of abstract classes is a design decision; it helps us establish common elements in a class that is too general to instantiate

Abstract Method

- Inheritance allows a sub-class to override the methods of its super-class.

- A super-class may altogether leave the implementation details of a method and declare such a method abstract:

- abstract type name(parameter-list);

- Two kinds of methods:

  1) concrete – may be overridden by sub-classes

  2) abstract – must be overridden by sub-classes

- It is illegal to define abstract constructors or static methods.

Defining a Package

- A package is both a naming and a visibility control mechanism:

  1)divides the name space into disjoint subsets It is possible to define classes within a package that are not accessible by code outside the package.

  2)controls the visibility of classes and their members It is possible to define class members that are only exposed to other members of the same package.

- Same-package classes may have an intimate knowledge of each other, but not expose that knowledge to other packages

Creating a Package

- A package statement inserted as the first line of the source file:

```
package myPackage;
class MyClass1 { … }
class MyClass2 { … }
```

- means that all classes in this file belong to the myPackage package.

- The package statement creates a name space where such classes are stored.

- When the package statement is omitted, class names are put into the default package which has no name.

Multiple Source Files

- Other files may include the same package instruction:

1. package myPackage;

```
class MyClass1 { ... }
class MyClass2 { ... }
```

2. package myPackage;

```
class MyClass3{ ... }
```

- A package may be distributed through several source files

Packages and Directories

- Java uses file system directories to store packages.
- Consider the Java source file:

```
package myPackage;
class MyClass1 { … }
class MyClass2 { … }
```

- The byte code files MyClass1.class and MyClass2.class must be stored in a directory myPackage.
- Case is significant! Directory names must match package names exactly.

Package Hierarchy

- To create a package hierarchy, separate each package name with a dot:

  package myPackage1.myPackage2.myPackage3;

- A package hierarchy must be stored accordingly in the file system:
  1) Unix myPackage1/myPackage2/myPackage3
  2) Windows myPackage1\myPackage2\myPackage3
  3) Macintosh myPackage1:myPackage2:myPackage3

- You cannot rename a package without renaming its directory!

Accessing a Package

- As packages are stored in directories, how does the Java run-time system know where to look for packages?

- Two ways:

  1)The current directory is the default start point - if packages are stored in the current directory or sub-directories, they will be found.

  2)Specify a directory path or paths by setting the CLASSPATH environment variable.

CLASSPATH Variable

- CLASSPATH - environment variable that points to the root directory of the system's package hierarchy.
- Several root directories may be specified in CLASSPATH,
- e.g. the current directory and the C:\raju\myJava directory:

  .;C:\raju\myJava
- Java will search for the required packages by looking up subsequent directories described in the CLASSPATH variable.

Finding Packages

- Consider this package statement:
  package myPackage;

In order for a program to find myPackage, one of the following must be true:

1) program is executed from the directory immediately above myPackage (the parent of myPackage directory)

2) CLASSPATH must be set to include the path to myPackage

Example: Package

```
package MyPack;

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

Example: Package

```
class AccountBalance
{
public static void main(String args[])
{
Balance current[] = new Balance[3];
current[0] = new Balance("K. J. Fielding", 123.23);
current[1] = new Balance("Will Tell", 157.02);
current[2] = new Balance("Tom Jackson", -12.33);
for (int i=0; i<3; i++) current[i].show();
}
}
```

Example: Package

- Save, compile and execute:

  1) call the file AccountBalance.java

  2) save the file in the directory MyPack

  3) compile; AccountBalance.class should be also in MyPack

  4)set access to MyPack in CLASSPATH variable, or make the

  parent of MyPack your current directory

  5) run: java MyPack.AccountBalance

- Make sure to use the package-qualified class name.

Importing of Packages

- Since classes within packages must be fully-qualified with their package names, it would be tedious to always type long dot-separated names.

- The import statement allows to use classes or whole packages directly.

- Importing of a concrete class:

  import myPackage1.myPackage2.myClass;

- Importing of all classes within a package:

  import myPackage1.myPackage2.*;

Import Statement

- The import statement occurs immediately after the package

  statement and before the class statement:

  package myPackage;

- import otherPackage1;otherPackage2.otherClass;

  class myClass { … }

- The Java system accepts this import statement by default:

  import java.lang.*;

- This package includes the basic language functions. Without such functions, Java is of no much use.

Example: Packages 1

- A package MyPack with one public class Balance.

  The class has two same-package variables: public constructor and a public show method.

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

Example: Packages 2

The importing code has access to the public class Balance of the
MyPack package and its two public members:

```
import MyPack.*;
class TestBalance {
public static void main(String args[]) {
Balance test = new Balance("J. J. Jaspers", 99.88);
test.show();
}
}
```

Java Source File

Finally, a Java source file consists of:

1) a single package instruction (optional)

2) several import statements (optional)

3) a single public class declaration (required)

4) several classes private to the package (optional)

At the minimum, a file contains a single public class declaration.

Differences between classes and interfaces

- Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body.
- One class can implement any number of interfaces.
- Interfaces are designed to support dynamic method resolution at run time.
- Interface is little bit like a class... but interface is lack in instance variables....that's u can't create object for it.....
- Interfaces are developed to support multiple inheritance...
- The methods present in interfaces r pure abstract..
- The access specifiers public,private,protected are possible with classes, but the interface uses only one spcifier public.....
- interfaces contains only the method declarations.... no definitions.......
- A interface defines, which method a class has to implement. This is way - if you want to call a method defined by an interface - you don't need to know the exact class type of an object, you only need to know that it implements a specific interface.
- Another important point about interfaces is that a class can implement multiple interfaces.

Defining an interface

- Using interface, we specify what a class must do, but not how it does this.
- An interface is syntactically similar to a class, but it lacks instance variables and its methods are declared without any body.
- An interface is defined with an interface keyword.

❖ An interface declaration consists of modifiers, the keyword interface,the interface name, a comma-separated list of parent interfaces (if any), and the interface body.

For example:

```
public interface GroupedInterface extends Interface1, Interface2, Interface3 {
// constant declarations double E = 2.718282;
// base of natural logarithms //
//method signatures
 void doSomething (int i, double x);
 int doSomethingElse(String s);
 }
```

❖ The public access specifier indicates that the interface can be used by any class in any package. If you do not specify that the interface is public, your interface will be accessible only to classes defined in the same package as the interface.

❖ An interface can extend other interfaces, just as a class can extend or subclass another class. However, whereas a class can extend only one other class, an interface can extend any number of interfaces. The interface declaration includes a comma-separated list of all the interfaces that it extends

Implementing interface

General format:

```
access interface name {
type method-name1(parameter-list);
type method-name2(parameter-list);
…
type var-name1 = value1;
type var-nameM = valueM;
…
}
```

- Two types of access:
  1) public – interface may be used anywhere in a program
  2) default – interface may be used in the current package only

- Interface methods have no bodies – they end with the semicolon after the parameter list.

- They are essentially abstract methods.

- An interface may include variables, but they must be final, static and initialized with a constant value.

- In a public interface, all members are implicitly public.

Interface Implementation

- A class implements an interface if it provides a complete set of methods defined by this interface.

  1) any number of classes may implement an interface

  2) one class may implement any number of interfaces

- Each class is free to determine the details of its implementation.

- Implementation relation is written with the implements keyword.

Implementation Format

- General format of a class that includes the implements clause:
- Syntax:

```
 access class name extends super-class implements
interface1, interface2, ..., interfaceN {
                    ...
                                        }
```

- Access is public or default.

Implementation Comments

- If a class implements several interfaces, they are separated with a comma.
- If a class implements two interfaces that declare the same method, the same method will be used by the clients of either interface.
- The methods that implement an interface must be declared public.
- The type signature of the implementing method must match exactly the type signature specified in the interface definition.

Example: Interface

Declaration of the Callback interface:

```
interface Callback
{
void callback(int param);
}
```

Client class implements the Callback interface:

```
class Client implements Callback
{
public void callback(int p)
{
System.out.println("callback called with " + p);
}
}
```

More Methods in Implementation

- An implementing class may also declare its own methods:

```
class Client implements Callback {
public void callback(int p) {
System.out.println("callback called with " + p);
}
void nonIfaceMeth() {
System.out.println("Classes that implement " +
"interfaces may also define " +
"other members, too.");
}
}
```

Applying interfaces

- A Java *interface* declares a set of method signatures i.e., says what behavior exists Does not say how the behavior is implemented
  i.e., does not give code for the methods
- Does not describe any state (but may include "final" constants)
- A concrete class that implements an interface Contains "implements *InterfaceName*" in the class declaration
- Must provide implementations (either directly or inherited from a superclass) of all methods declared in the interface
- An abstract class can also implement an interface
- Can optionally have implementations of some or all interface methods

- Interfaces and Extends both describe an "is- a" relation.
- If B *implements* interface A, then B inherits the (abstract) method signatures in A
- If B *extends* class A, then B inherits everything in A.
- which can include method code and instance variables as well as abstract method signatures.
- Inheritance" is sometimes used to talk about the superclass / subclass "extends" relation only

Variables in interface

- Variables declared in an interface must be constants.
- A technique to import shared constants into multiple classes:

  1)declare an interface with variables initialized to the desired
  values

  2)include that interface in a class through implementation.
- As no methods are included in the interface, the class does not implement.
- anything except importing the variables as constants.

Example: Interface Variables 1

An interface with constant values:

```
import java.util.Random;
interface SharedConstants {
int NO = 0;
int YES = 1;
int MAYBE = 2;
int  LATER  =  3;
int SOON = 4;
int NEVER = 5;
}
```

- Question implements SharedConstants, including all its constants.
- Which constant is returned depends on the generated random number:

```
class Question implements SharedConstants {
Random rand = new Random();
int ask() {
int prob = (int) (100 * rand.nextDouble());
if (prob < 30) return NO;
else if (prob < 60) return YES;
else if (prob < 75) return LATER;
else if (prob < 98) return SOON;
else return NEVER;
}
}
```

- AskMe includes all shared constants in the same way, using them to display the result, depending on the value received:

```
class AskMe implements SharedConstants {
static void answer(int result) {
switch(result) {
case NO: System.out.println("No"); break;
case YES: System.out.println("Yes"); break;
case MAYBE: System.out.println("Maybe"); break;
case LATER: System.out.println("Later"); break;
case SOON: System.out.println("Soon"); break;
case NEVER: System.out.println("Never"); break;
}
}
```

Example: Interface Variables 4

- The testing function relies on the fact that both ask and answer methods.
- defined in different classes, rely on the same constants:

```
public static void main(String args[]) {
Question q = new Question();
answer(q.ask());
answer(q.ask());
answer(q.ask());
answer(q.ask());
}
}
```

Extending interfaces

- One interface may inherit another interface.
- The inheritance syntax is the same for classes and interfaces.

```
interface MyInterface1 {
void myMethod1(...) ;
}
interface MyInterface2 extends MyInterface1 {
void myMethod2(...) ;
}
```

- When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain.

Example: Interface Inheritance 1

- Consider interfaces A and B.

```
interface A {
void meth1();
void meth2();
}

B extends A:
interface B extends A {
void meth3();
}
```

Example: Interface Inheritance 2

- MyClass must implement all of A and B methods:

```
class MyClass implements B {
public void meth1() {
System.out.println("Implement meth1().");
}
public void meth2() {
System.out.println("Implement meth2().");
}
public void meth3() {
System.out.println("Implement meth3().");
} }
```

Example: Interface Inheritance 3

- Create a new MyClass object, then invoke all interface methods on it:

```
class IFExtend {
public static void main(String arg[]) {
MyClass ob = new MyClass();
ob.meth1();
ob.meth2();
ob.meth3();
}
}
```

UNIT-3

Exceptions

- Exception is an abnormal condition that arises when executing a program.
- In the languages that do not support exception handling, errors must be checked and handled manually, usually through the use of error codes.
- In contrast, Java:

  1)provides syntactic mechanisms to signal, detect and handle errors

  2)ensures a clean separation between the code executed in the

  absence of errors and the code to handle various kinds of errors

  3)brings run-time error management into object-oriented programming

Exception Handling

- An exception is an object that describes an exceptional condition (error) that has occurred when executing a program.
- Exception handling involves the following:

1)when an error occurs, an object (exception) representing this error is created and thrown in the method that caused it

2) that method may choose to handle the exception itself or pass it on

3) either way, at some point, the exception is caught and processed

Exception Sources

- Exceptions can be:
  1) generated by the Java run-time system Fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment.
  2) manually generated by programmer’s code Such exceptions are typically used to report some error conditions to the caller of a method.

Exception Constructs

- Five constructs are used in exception handling:
  1) try – a block surrounding program statements to monitor for exceptions
  2) catch – together with try, catches specific kinds of exceptions and handles them in some way
  3) finally – specifies any code that absolutely must be executed whether or not an exception occurs
  4) throw – used to throw a specific exception from the program
  5) throws – specifies which exceptions a given method can throw

Exception-Handling Block

General form:

```
try { … }
catch(Exception1 ex1) { … }
catch(Exception2 ex2) { … }
…
finally { … }
```

where:

1) try { … } is the block of code to monitor for exceptions
2) catch(Exception ex) { … } is exception handler for the exception Exception
3) finally { … } is the block of code to execute before the try block ends

Benefits of exception handling

- Separating Error-Handling code from “regular” business logic code
- Propagating errors up the call stack
- Grouping and differentiating error types

Using Java Exception Handling

```
method1 {
try {
call method2;
} catch (exception e) {
doErrorProcessing;
}
}
method2 throws exception {
call method3;
}
method3 throws exception {
call readFile;
}
```

- Any checked exceptions that can be thrown within a method must be specified in its throws clause.

Grouping and Differentiating Error Types

- Because all exceptions thrown within a program are objects, the grouping or categorizing of exceptions is a natural outcome of the class hierarchy.
- An example of a group of related exception classes in the Java platform are those defined in java.io.IOException and its descendants.
- IOException is the most general and represents any type of error that can occur when performing I/O.
- Its descendants represent more specific errors. For example, FileNotFoundException means that a file could not be located on disk.

- A method can write specific handlers that can handle a very specific exception.

- The FileNotFoundException class has no descendants, so the following handler can handle only one type of exception.

```
catch (FileNotFoundException e) {
...
}
```

- A method can catch an exception based on its group or general type by specifying any of the exception's super classes in the catch statement.

- For example, to catch all I/O exceptions, regardless of their specific type, an exception handler specifies an IOException argument.

```
// Catch all I/O exceptions, including
// FileNotFoundException, EOFException, and so on.
catch (IOException e) {
...
}
```

Termination vs. Resumption

- There are two basic models in exception-handling theory.
- In *termination* the error is so critical there's no way to get back to where the exception occurred. Whoever threw the exception decided that there was no way to salvage the situation, and they don't *want* to come back.
- The alternative is called *resumption*. It means that the exception handler is expected to do something to rectify the situation, and then the faulting method is retried, presuming success the second time. If you want resumption, it means you still hope to continue execution after the exception is handled.

- In resumption a method call that want resumption-like behavior (i.e don’t throw an exception all a method that fixes the problem.)

- Alternatively, place your **try** block inside a **while** loop that keeps reentering the **try** block until the result is satisfactory.

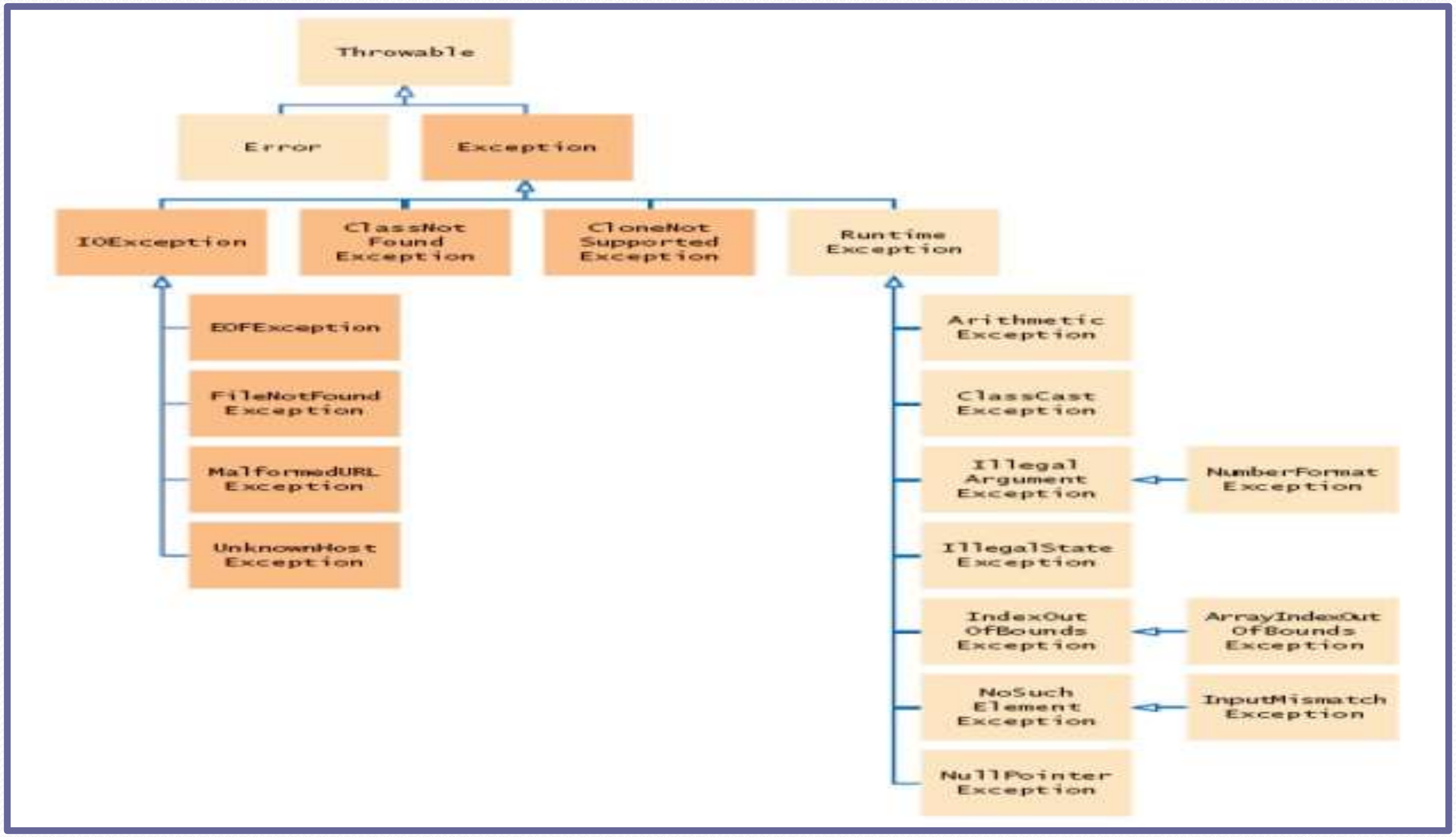
- Operating systems that supported resumptive exception handling eventually ended up using termination-like code and skipping resumption.

Exception Hierarchy

- All exceptions are sub-classes of the build-in class Throwable.
- Throwable contains two immediate sub-classes:

1) Exception – exceptional conditions that programs should catch
   The class includes:
   a) RuntimeException – defined automatically for user programs to include: division by zero, invalid array indexing, etc.
   b) use-defined exception classes

2) Error – exceptions used by Java to indicate errors with the runtime environment; user programs are not supposed to catch them


Throwable
Error
Exception
IOException
ClassNot Found Exception
CloneNot Supported Exception
Runtime Exception
EOFException
FileNotFound Exception
MalformedURL Exception
UnknownHost Exception
Arithmetic Exception
ClassCast Exception
Illegal Argument Exception
NumberFormat Exception
IllegalState Exception
IndexOut OfBounds Exception
ArrayIndexOut OfBounds Exception
NoSuch Element Exception
InputMismatch Exception
NullPointer Exception

Usage of *try-catch* Statements

- Syntax:

```
try {
<code to be monitored for exceptions>
} catch (<ExceptionType1> <ObjName>) {
<handler if ExceptionType1 occurs>
} ...
} catch (<ExceptionTypeN> <ObjName>) {
<handler if ExceptionTypeN occurs>
}
```

Catching Exceptions: The *try-catch* Statements

```
class DivByZero {
public static void main(String args[]) {
try {
System.out.println(3/0);
System.out.println(“Please print me.”);
} catch (ArithmeticException exc) {
//Division by zero is an ArithmeticException
System.out.println(exc);
}
System.out.println(“After exception.”);
}
}
```

Catching Exceptions: Multiple catch

```
class MultipleCatch {
public static void main(String args[]) {
try {
int den = Integer.parseInt(args[0]);
System.out.println(3/den);
} catch (ArithmeticException exc) {
System.out.println(“Divisor was 0.”);
} catch (ArrayIndexOutOfBoundsException exc2) {
System.out.println(“Missing argument.”);
}
System.out.println(“After exception.”);
}
}
```

Catching Exceptions: Nested try's

```
class NestedTryDemo {
public static void main(String args[]){
try {
int a = Integer.parseInt(args[0]);
try {
int b = Integer.parseInt(args[1]);
System.out.println(a/b);
} catch (ArithmeticException e) {
System.out.println("Div by zero error!");
} } catch (ArrayIndexOutOfBoundsException) {
System.out.println("Need 2 parameters!");
} }   }
```

Catching Exceptions: Nested try's with methods

```
class NestedTryDemo2 {
 static void nestedTry(String args[]) {
 try {
 int a = Integer.parseInt(args[0]);
 int b = Integer.parseInt(args[1]);
 System.out.println(a/b);
 } catch (ArithmeticException e) {
System.out.println("Div by zero error!");
 }    }
public static void main(String args[]){
 try {
 nestedTry(args);
 } catch (ArrayIndexOutOfBoundsException e) {
 System.out.println("Need 2 parameters!");
 }  }   }
```

Throwing Exceptions(throw)

- So far, we were only catching the exceptions thrown by the Java system.
- In fact, a user program may throw an exception explicitly:

```
        throw ThrowableInstance;
```

- ThrowableInstance must be an object of type Throwable or its subclass.

Once an exception is thrown by:

```
        throw ThrowableInstance;
```

1) the flow of control stops immediately.

2)the nearest enclosing try statement is inspected if it has a catch statement that matches the type of exception:

3) if one exists, control is transferred to that statement

4) otherwise, the next enclosing try statement is examined

5) if no enclosing try statement has a corresponding catch clause, the default exception handler halts the program and prints the stack

Creating Exceptions

Two ways to obtain a Throwable instance:

1) creating one with the new operator

All Java built-in exceptions have at least two Constructors:

One without parameters and another with one String parameter:

```
throw new NullPointerException("demo");
```

2) using a parameter of the catch clause

```
try { … } catch(Throwable e) { … e … }
```

Example: throw 1

```
class ThrowDemo {
//The method demoproc throws a NullPointerException
  exception which is immediately caught in the try block and
  re-thrown:
static void demoproc() {
try {
throw new NullPointerException("demo");
} catch(NullPointerException e) {
System.out.println("Caught inside demoproc.");
throw e;
}
}
```

Example: throw 2

The main method calls demoproc within the try block which catches and handles the NullPointerException exception:

```
public static void main(String args[]) {
try {
demoproc();
} catch(NullPointerException e) {
System.out.println("Recaught: " + e);
}
}
}
```

throws Declaration

- If a method is capable of causing an exception that it does not handle, it must specify this behavior by the throws clause in its declaration:

```
type name(parameter-list) throws exception-list {
                    …
}
```

- where exception-list is a comma-separated list of all types of exceptions that a method might throw.

- All exceptions must be listed except Error and RuntimeException or any of their subclasses, otherwise a compile-time error occurs.

Example: throws 1

- The throwOne method throws an exception that it does not catch, nor declares it within the throws clause.

```
class ThrowsDemo {
static void throwOne() {
System.out.println("Inside throwOne.");
throw new IllegalAccessException("demo");
}
public static void main(String args[]) {
throwOne();
}
}
```

- Therefore this program does not compile.

Example: throws 2

- Corrected program: throwOne lists exception, main catches it:

```
class ThrowsDemo {
static void throwOne() throws IllegalAccessException {
System.out.println("Inside throwOne.");
throw new IllegalAccessException("demo");
}
public static void main(String args[]) {
try {
throwOne();
} catch (IllegalAccessException e) {
System.out.println("Caught " + e);
}  }  }
```

finally

- When an exception is thrown:

  1) the execution of a method is changed

  2) the method may even return prematurely.

- This may be a problem is many situations.

- For instance, if a method opens a file on entry and closes on exit; exception handling should not bypass the proper closure of the file.

- The finally block is used to address this problem.

finally Clause

- The try/catch statement requires at least one catch or finally clause, although both are optional:

```
try { … }
catch(Exception1 ex1) { … } …
finally { … }
```

- Executed after try/catch whether of not the exception is thrown.

- Any time a method is to return to a caller from inside the try/catch block via:

  1) uncaught exception or

  2) explicit return

  the finally clause is executed just before the method returns.

Example: finally 1

- Three methods to exit in various ways.

```
class FinallyDemo {
//procA prematurely breaks out of the try by throwing an
   exception, the finally clause is executed on the way out:
static void procA() {
try {
System.out.println("inside procA");
throw new RuntimeException("demo");
} finally  {
System.out.println("procA's finally");
}  }
```

Example: finally 2

// procB’s try statement is exited via a return statement, the finally clause is executed before procB returns:

```
static void procB() {
try {
System.out.println("inside procB");
return;
} finally {
System.out.println("procB's finally");
}
}
```

Example: finally 3

- In procC, the try statement executes normally without error, however the finally clause is still executed:

```
static void procC() {
try {
System.out.println("inside procC");
} finally {
System.out.println("procC's finally");
}
}
```

Example: finally 4

- Demonstration of the three methods:

```
public static void main(String args[]) { try {
        procA();
        } catch (Exception e) {
        System.out.println("Exception caught");
        }
        procB();
        procC();
        }
        }
```

Java Built-In Exceptions

- The default java.lang package provides several exception classes, all sub-classing the RuntimeException class.
- Two sets of build-in exception classes:

  1)unchecked exceptions – the compiler does not check if a method handles or throws there exceptions

  2) checked exceptions – must be included in the method's throws clause if the method generates but does not handle them

Unchecked Built-In Exceptions

Methods that generate but do not handle those exceptions need not declare them in the throws clause:

1) ArithmeticException
2) ArrayIndexOutOfBoundsException
3) ArrayStoreException
4) ClassCastException
5) IllegalStateException
6) IllegalMonitorStateException
7) IllegalArgumentException

8. StringIndexOutOfBounds
9. UnsupportedOperationException
10. SecurityException
11. NumberFormatException
12. NullPointerException
13. NegativeArraySizeException
14. IndexOutOfBoundsException
15. IllegalThreadStateException

Checked Built-In Exceptions

Methods that generate but do not handle those exceptions must declare them in the throws clause:

1. NoSuchMethodException NoSuchFieldException
2. InterruptedException
3. InstantiationException
4. IllegalAccessException
5. CloneNotSupportedException
6. ClassNotFoundException

Creating Own Exception Classes

- Build-in exception classes handle some generic errors.
- For application-specific errors define your own exception classes. How? Define a subclass of Exception:

  class MyException extends Exception { … }

- MyException need not implement anything – its mere existence in the type system allows to use its objects as exceptions.

Example: Own Exceptions 1

- A new exception class is defined, with a private detail variable, a one parameter constructor and an overridden toString method:

```
class MyException extends Exception {
private int detail;
MyException(int a) {
detail = a;
}
public String toString() {
return "MyException[" + detail + "]";
}
}
```

Example: Own Exceptions 2

```
class ExceptionDemo {
```

The static compute method throws the MyException exception whenever its a argument is greater than 10:

```
static void compute(int a) throws MyException {
System.out.println("Called compute(" + a + ")");
if (a > 10) throw new MyException(a);
System.out.println("Normal exit");
}
```

Example: Own Exceptions 3

The main method calls compute with two arguments within a try block that catches the MyException exception:

```
public static void main(String args[]) {
try {
compute(1);
compute(20);
} catch (MyException e) {
System.out.println("Caught " + e);
}
}
}
```

Differences between multi threading and multitasking

Multi-Tasking

- Two kinds of multi-tasking:
  1) process-based multi-tasking
  2) thread-based multi-tasking
- Process-based multi-tasking is about allowing several programs to execute concurrently, e.g. Java compiler and a text editor.
- Processes are heavyweight tasks:
  1) that require their own address space
  2) inter-process communication is expensive and limited
  3) context-switching from one process to another is expensive and limited

Thread-Based Multi-Tasking

- Thread-based multi-tasking is about a single program executing concurrently
- several tasks e.g. a text editor printing and spell-checking text.
- Threads are lightweight tasks:

  1) they share the same address space

  2) they cooperatively share the same process

  3) inter-thread communication is inexpensive

  4) context-switching from one thread to another is low-cost
- Java multi-tasking is thread-based.

Reasons for Multi-Threading

- Multi-threading enables to write efficient programs that make the maximum use of the CPU, keeping the idle time to a minimum.
- There is plenty of idle time for interactive, networked applications:

  1) the transmission rate of data over a network is much slower than the rate at which the computer can process it

  2) local file system resources can be read and written at a much slower rate than can be processed by the CPU

  3) of course, user input is much slower than the computer

Thread Lifecycle

- Thread exist in several states:

  1) ready to run

  2) running

  3) a running thread can be suspended

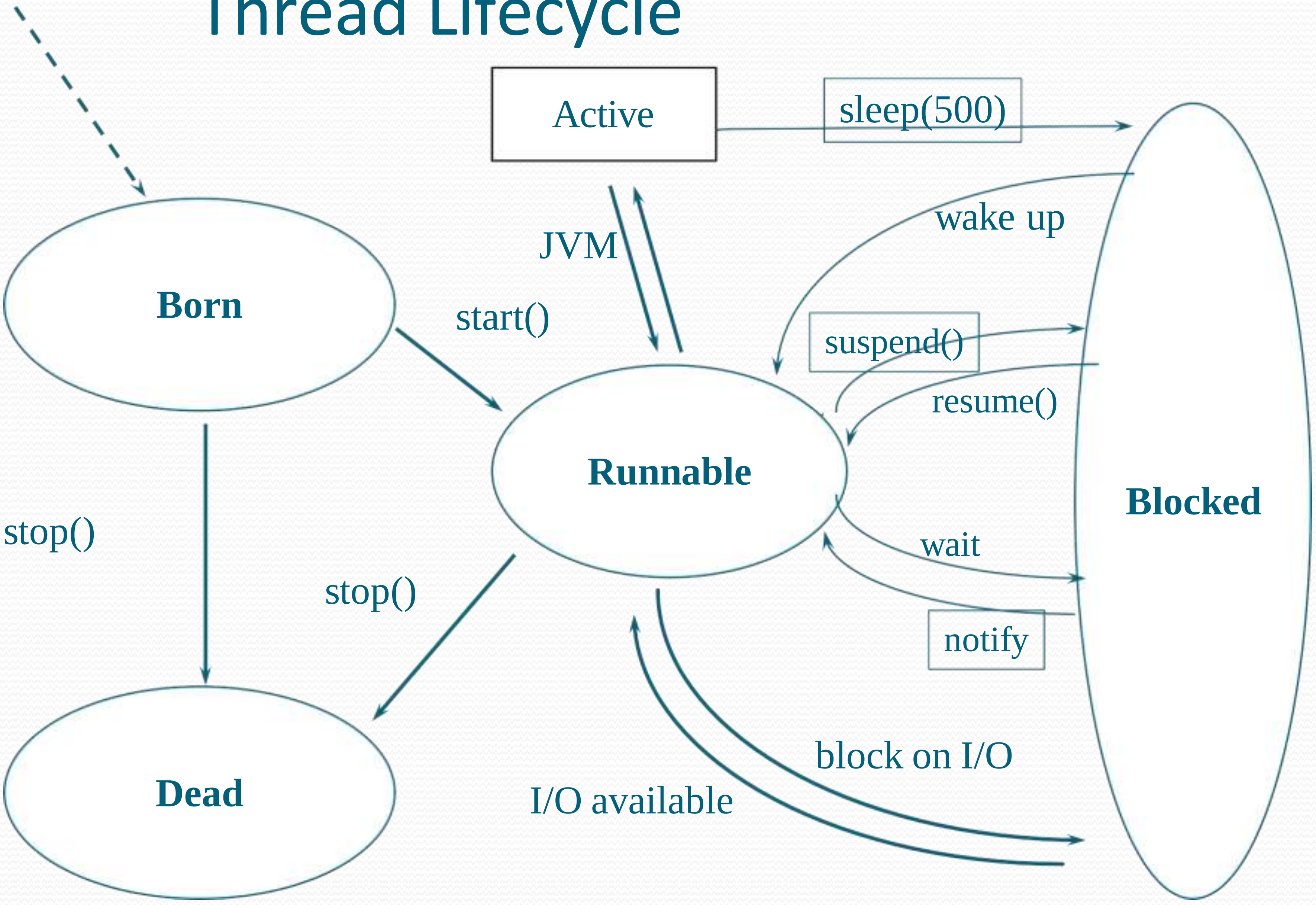
  4) a suspended thread can be resumed

  5) a thread can be blocked when waiting for a resource

  6) a thread can be terminated

- Once terminated, a thread cannot be resumed.

Thread Lifecycle



- **New state –** After the creations of Thread instance the thread is in this state but before the start() method invocation. At this point, the thread is considered not alive.
- **Runnable (Ready-to-run) state –** A thread start its life from Runnable state. A thread first enters runnable state after the invoking of start() method but a thread can return to this state after either running, waiting, sleeping or coming back from blocked state also. On this state a thread is waiting for a turn on the processor.
- **Running state –** A thread is in running state that means the thread is currently executing. There are several ways to enter in Runnable state but there is only one way to enter in Running state: the scheduler select a thread from runnable pool.
- **Dead state –** A thread can be considered dead when its run() method completes. If any thread comes on this state that means it cannot ever run again.
- **Blocked -** A thread can enter in this state because of waiting the resources that are hold by another thread.

Creating Threads

- To create a new thread a program will:
  1) extend the Thread class, or
  2) implement the Runnable interface
- Thread class encapsulates a thread of execution.
- The whole Java multithreading environment is based on the Thread class.

Thread Methods

- Start: a thread by calling start its run method
- Sleep: suspend a thread for a period of time
- Run: entry-point for a thread
- Join: wait for a thread to terminate
- isAlive: determine if a thread is still running
- getPriority: obtain a thread’s priority
- getName: obtain a thread’s name

New Thread: Runnable

- To create a new thread by implementing the Runnable interface:

  1)create a class that implements the run method (inside this method, we define the code that constitutes the new thread):

  ```
  public void run()
  ```

  2) instantiate a Thread object within that class, a possible constructor is:

  ```
  Thread(Runnable threadOb, String threadName)
  ```

3) call the start method on this object (start calls run):

   ```
   void start()
   ```

Example: New Thread 1

- A class NewThread that implements Runnable:

```
class NewThread implements Runnable {
Thread t;
//Creating and starting a new thread. Passing this to the
// Thread constructor – the new thread will call this
// object’s run method:
NewThread() {
t = new Thread(this, "Demo Thread");
System.out.println("Child thread: " + t);
t.start();
}
```

Example: New Thread 2

```
//This is the entry point for the newly created thread – a five-iterations loop
//with a half-second pause between the iterations all within try/catch:
public void run() {
try {
for (int i = 5; i > 0; i--) {
System.out.println("Child Thread: " + i);
Thread.sleep(500);
}
} catch (InterruptedException e) {
System.out.println("Child interrupted.");
}
System.out.println("Exiting child thread.");
}
}
```

Example: New Thread 3

```
class ThreadDemo {
public static void main(String args[]) {
//A new thread is created as an object of
// NewThread:
new NewThread();
//After calling the NewThread start method,
// control returns here.
```

Example: New Thread 4

```
//Both threads (new and main) continue concurrently.
//Here is the loop for the main thread:
try {
for (int i = 5; i > 0; i--) {
System.out.println("Main Thread: " + i);
Thread.sleep(1000);
}
} catch (InterruptedException e) {
System.out.println("Main thread interrupted.");
}
System.out.println("Main thread exiting.");
}
}
```

New Thread: Extend Thread

- The second way to create a new thread:

  1) create a new class that extends Thread

  2) create an instance of that class
- Thread provides both run and start methods:

  1) the extending class must override run

  2) it must also call the start method

Example: New Thread 1

- The new thread class extends Thread:

```
class NewThread extends Thread {
//Create a new thread by calling the Thread’s
// constructor and start method:
NewThread() {
super("Demo Thread");
System.out.println("Child thread: " + this);
start();
}
```

Example: New Thread 2

NewThread overrides the Thread’s run method:

```
public void run() {
try {
for (int i = 5; i > 0; i--) {
System.out.println("Child Thread: " + i);
Thread.sleep(500);
}
} catch (InterruptedException e) {
System.out.println("Child interrupted.");
}
System.out.println("Exiting child thread.");
}
}
```

Example: New Thread 3

```
class ExtendThread {
public static void main(String args[]) {
//After a new thread is created:
new NewThread();
//the new and main threads continue
//concurrently…
```

Example: New Thread 4

```
//This is the loop of the main thread:
try {
for (int i = 5; i > 0; i--) {
System.out.println("Main Thread: " + i);
Thread.sleep(1000);
}
} catch (InterruptedException e) {
System.out.println("Main thread interrupted.");
}
System.out.println("Main thread exiting.");
}
}
```

Threads: Synchronization

- Multi-threading introduces asynchronous behavior to a program.
- How to ensure synchronous behavior when we need it?
- For instance, how to prevent two threads from simultaneously writing and reading the same object?
- Java implementation of monitors:

  1) classes can define so-called synchronized methods
  2) each object has its own implicit monitor that is automatically entered when one of the object’s synchronized methods is called

  3)once a thread is inside a synchronized method, no other thread can call any other synchronized method on the same object

Thread Synchronization

- Language keyword: synchronized
- Takes out a monitor lock on an object
  - Exclusive lock for that thread
- If lock is currently unavailable, thread will block

Thread Synchronization

- Protects access to code, not to data
  - Make data members private
  - Synchronize accessor methods
- Puts a “force field” around the locked object so no other threads can enter
    - Actually, it only blocks access to other synchronizing threads

Daemon Threads

- Any Java thread can be a *daemon* thread.
- Daemon threads are service providers for other threads running in the same process as the daemon thread.
- The run() method for a daemon thread is typically an infinite loop that waits for a service request. When the only remaining threads in a process are daemon threads, the interpreter exits. This makes sense because when only daemon threads remain, there is no other thread for which a daemon thread can provide a service.
- To specify that a thread is a daemon thread, call the setDaemon method with the argument true. To determine if a thread is a daemon thread, use the accessor method isDaemon.

Thread Groups

- Every Java thread is a member of a *thread group*.
- Thread groups provide a mechanism for collecting multiple threads into a single object and manipulating those threads all at once, rather than individually.
- For example, you can start or suspend all the threads within a group with a single method call.
- Java thread groups are implemented by the “ThreadGroup” class in the java.lang package.
- The runtime system puts a thread into a thread group during thread construction.
- When you create a thread, you can either allow the runtime system to put the new thread in some reasonable default group or you can explicitly set the new thread's group.
- The thread is a permanent member of whatever thread group it joins upon its creation--you cannot move a thread to a new group after the thread has been created

The ThreadGroup Class

- The “ThreadGroup” class manages groups of threads for Java applications.
- A ThreadGroup can contain any number of threads.
- The threads in a group are generally related in some way, such as who created them, what function they perform, or when they should be started and stopped.
- ThreadGroups can contain not only threads but also other ThreadGroups.
- The top-most thread group in a Java application is the thread group named main.
- You can create threads and thread groups in the main group.
- You can also create threads and thread groups in subgroups of main.

Creating a Thread Explicitly in a Group

- A thread is a permanent member of whatever thread group it joins when its created--you cannot move a thread to a new group after the thread has been created. Thus, if you wish to put your new thread in a thread group other than the default, you must specify the thread group explicitly when you create the thread.
- The Thread class has three constructors that let you set a new thread's group:

  public Thread(ThreadGroup *group*, Runnable *target*) public Thread(ThreadGroup *group*, String *name*)

  public Thread(ThreadGroup *group*, Runnable *target*, String *name*)
- Each of these constructors creates a new thread, initializes it based on the Runnable and String parameters, and makes the new thread a member of the specified group. For example:

  ThreadGroup myThreadGroup = new ThreadGroup("My Group of Threads");

  Thread myThread = new Thread(myThreadGroup, "a thread for my group");

UNIT-4

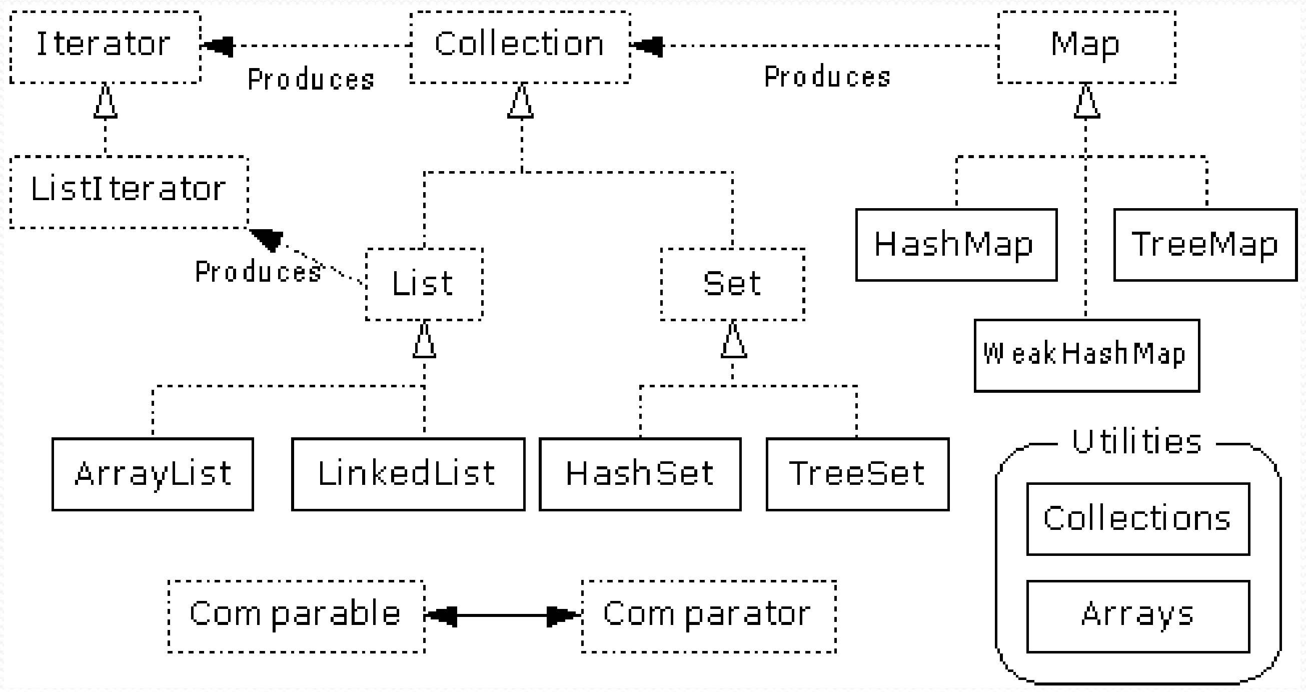
- **Collection Framework in java** – Introduction to java collections- overview of java collection frame work-generics-commonly used collection classes- Array List-vector -hash table-stack-enumeration-iterator-string tokenizer -random -scanner -calendar and properties
- **Files** – streams – byte streams- character stream- text input/output- binary input/output- random access file operations- file management using file class.
- **Connecting to Database** – JDBC Type 1 to 4 drivers-connecting to a database- quering a database and processing the results- updating data with JDBC

Collection Framework in java

collection

- A *collection* — sometimes called a container — is simply an object that groups multiple elements into a single unit.
- Collections are used to store, retrieve, manipulate, and communicate aggregate data.
- Typically, they represent data items that form a natural group, such as a poker hand (a collection of cards), a mail folder (a collection of letters), or a telephone directory (a mapping of names to phone numbers).

Collections Framework Diagram



•Interfaces, Implementations, and Algorithms
•From Thinking in Java, page 462

The collections framework was designed to meet several goals:

- The framework had to be high-performance. The implementations for the fundamental collections (dynamic arrays, linked lists, trees, and hashtables) are highly efficient.
- The framework had to allow different types of collections to work in a similar manner and with a high degree of interoperability.
- Extending and/or adapting a collection had to be easy.

- A *collections framework* is a unified architecture for representing and manipulating collections.

All collections frameworks contain the following:

- **Interfaces:** These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation. In object-oriented languages, interfaces generally form a hierarchy.

- **Implementations**: These are the concrete implementations of the collection interfaces. In essence, they are reusable data structures.

- **Algorithms:** These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. The algorithms are said to be *polymorphic*: that is, the same method can be used on many different implementations of the appropriate collection interface. In essence, algorithms are reusable functionality.

Benefits of the Java Collections Framework

- **Reduces programming effort:** By providing useful data structures and algorithms, the Collections Framework frees you to concentrate on the important parts of your program rather than on the low-level "plumbing" required to make it work. By facilitating interoperability among unrelated APIs, the Java Collections Framework frees you from writing adapter objects or conversion code to connect APIs.
- **Increases program speed and quality:** This Collections Framework provides high-performance, high-quality implementations of useful data structures and algorithms. The various implementations of each interface are interchangeable, so programs can be easily tuned by switching collection implementations. Because you're freed from the drudgery of writing your own data structures, you'll have more time to devote to improving programs' quality and performance.

- **Allows interoperability among unrelated APIs:** The collection interfaces are the vernacular by which APIs pass collections back and forth. If my network administration API furnishes a collection of node names and if your GUI toolkit expects a collection of column headings, our APIs will interoperate seamlessly, even though they were written independently.

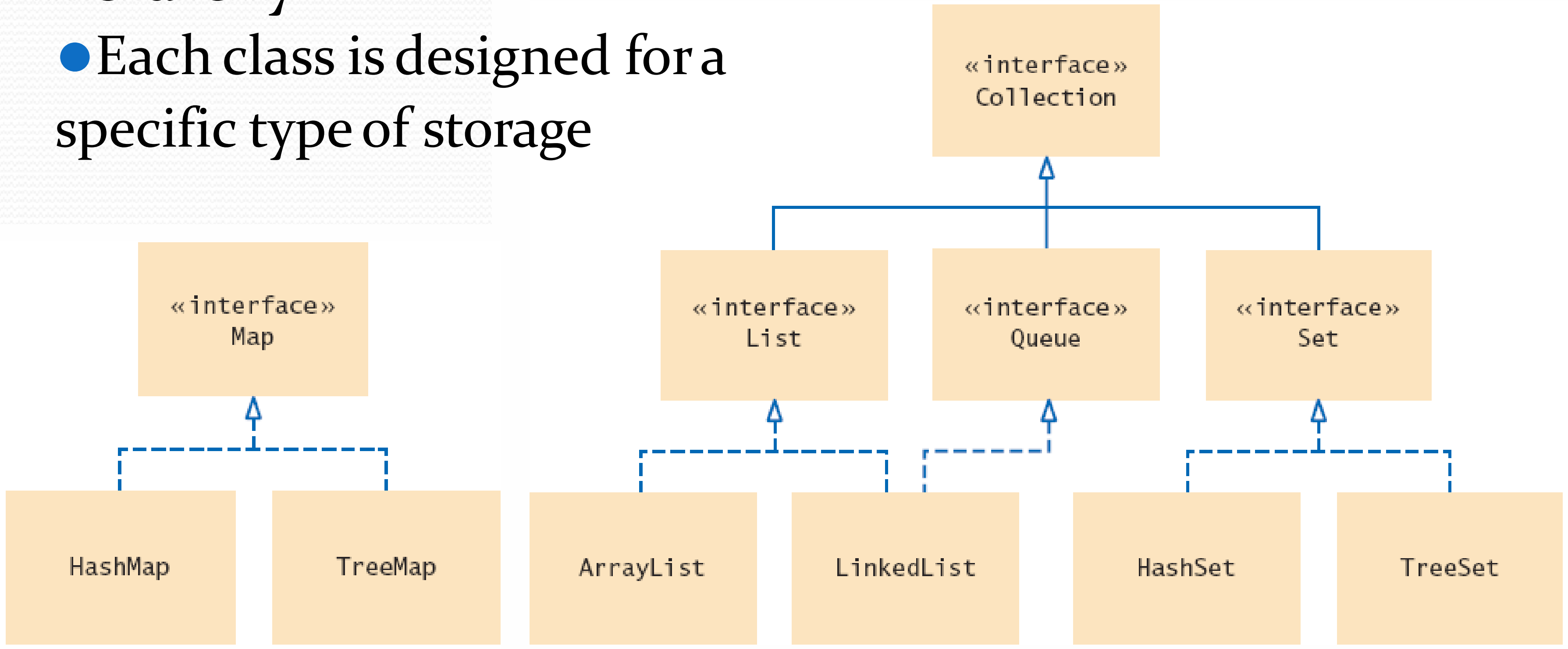
- **Reduces effort to learn and to use new APIs:** Many APIs naturally take collections on input and furnish them as output. In the past, each such API had a small sub-API devoted to manipulating its collections. There was little consistency among these ad hoc collections sub-APIs, so you had to learn each one from scratch, and it was easy to make mistakes when using them. With the advent of standard collection interfaces, the problem went away.

- **Reduces effort to design new APIs:** This is the flip side of the previous advantage. Designers and implementers don't have to reinvent the wheel each time they create an API that relies on collections; instead, they can use standard collection interfaces.

- **Fosters software reuse:** New data structures that conform to the standard collection interfaces are by nature reusable. The same goes for new algorithms that operate on objects that implement these interfaces.

Collections Framework Diagram

- Each collection class implements an interface from a hierarchy
  - Each class is designed for a specific type of storage



Lists and Sets

- Ordered Lists

- `ArrayList`
  - Stores a list of items in a dynamically sized array
- `LinkedList`
  - Allows speedy insertion and removal of items from the list

A **list** is a collection that maintains the order of its elements.

Lists and Sets

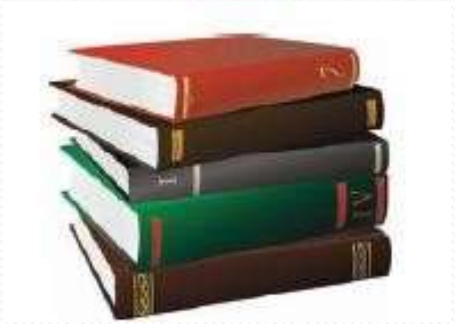
- Unordered Sets

- `HashSet`
  - Uses hash tables to speed up finding, adding, and removing elements
- `TreeSet`
  - Uses a binary tree to speed up finding, adding, and removing elements

A **set** is an unordered collection of unique elements.

Stacks and Queues

- Another way of gaining efficiency in a collection is to reduce the number of operations available
- Two examples are:
  - Stack
    - Remembers the order of its elements, but it does not allow you to insert elements in every position
    - You can only add and remove elements at the top
  - Queue
    - Add items to one end (the tail)
    - Remove them from the other end (the head)
    - Example: A line of people waiting for a bank teller



Maps

- A map stores keys, values, and the associations between them
  - Example:
  - Barcode keys and books

A map keeps associations between key and value objects.

- Keys
  - Provides an easy way to represent an object (such as a numeric bar code)
- Values
  - The actual object that is associated with the key

collection interfaces

The following list describes the core collection interfaces:

- Collection — the root of the collection hierarchy. A collection represents a group of objects known as its *elements*. The Collection interface is the least common denominator that all collections implement and is used to pass collections around and to manipulate them when maximum generality is desired. Some types of collections allow duplicate elements, and others do not. Some are ordered and others are unordered. The Java platform doesn't provide any direct implementations of this interface but provides implementations of more specific sub interfaces, such as Set and List.

- Set — a collection that cannot contain duplicate elements. This interface models the mathematical set abstraction and is used to represent sets, such as the cards comprising a poker hand, the courses making up a student's schedule, or the processes running on a machine..

- List — an ordered collection (sometimes called a *sequence*). Lists can contain duplicate elements. The user of a List generally has precise control over where in the list each element is inserted and can access elements by their integer index (position). If you've used Vector, you're familiar with the general flavor of List.

- Queue — a collection used to hold multiple elements prior to processing. Besides basic Collection operations, a Queue provides additional insertion, extraction, and inspection operations.

- Map — an object that maps keys to values. A Map cannot contain duplicate keys; each key can map to at most one value. If you've used Hashtable.

The last two core collection interfaces are merely sorted versions of Set and Map:

- SortedSet — a Set that maintains its elements in ascending order. Several additional operations are provided to take advantage of the ordering. Sorted sets are used for naturally ordered sets, such as word lists and membership rolls
- .
- SortedMap — a Map that maintains its mappings in ascending key order. This is the Map analog of SortedSet. Sorted maps are used for naturally ordered collections of key/value pairs, such as dictionaries and telephone directories

Summary of interfaces

- The core collection interfaces are the foundation of the Java Collections Framework.

The Java Collections Framework hierarchy consists of two distinct interface trees:

- 1.The first tree starts with the Collection interface, which provides for the basic functionality used by all collections, such as add and remove methods. Its subinterfaces — Set, List, and Queue — provide for more specialized collections.
- The Set interface does not allow duplicate elements. This can be useful for storing collections such as a deck of cards or student records. The Set interface has a subinterface, SortedSet, that provides for ordering of elements in the set.
- The List interface provides for an ordered collection, for situations in which you need precise control over where each element is inserted. You can retrieve elements from a List by their exact position.
- The Queue interface enables additional insertion, extraction, and inspection operations. Elements in a Queue are typically ordered in on a FIFO basis.
- The Deque interface enables insertion, deletion, and inspection operations at both the ends. Elements in a Deque can be used in both LIFO and FIFO.

- 2.The second tree starts with the Map interface, which maps keys and values similar to a Hashtable.
- Map's subinterface, SortedMap, maintains its key-value pairs in ascending order or in an order specified by a Comparator.
- These interfaces allow collections to be manipulated independently of the details of their representation.

The Collection Classes

- Java provides a set of standard collection classes that implement Collection interfaces. Some of the classes provide full implementations that can be used as-is and others are abstract class, providing skeletal implementations that are used as starting points for creating concrete collections.

Classes with Description

1**AbstractCollection** -Implements most of the Collection interface.

2**AbstractList** -Extends AbstractCollection and implements most of the List interface.

3**AbstractSequentialList** Extends AbstractList for use by a collection that uses sequential rather than random access of its elements.

4LinkedList -Implements a linked list by extending AbstractSequentialList.

5ArrayList -Implements a dynamic array by extending AbstractList.

6**AbstractSet** -Extends AbstractCollection and implements most of the Set interface.

7HashSet -Extends AbstractSet for use with a hash table.

8LinkedHashSet -Extends HashSet to allow insertion-order iterations.

9TreeSet -Implements a set stored in a tree. Extends AbstractSet.

10**AbstractMap** -Implements most of the Map interface.

11HashMap -Extends AbstractMap to use a hash table.

12TreeMap -Extends AbstractMap to use a tree.

13WeakHashMap -Extends AbstractMap to use a hash table with weak keys.

14LinkedHashMap-Extends HashMap to allow insertion-order iterations.

15IdentityHashMap -Extends AbstractMap and uses reference equality when comparing documents.

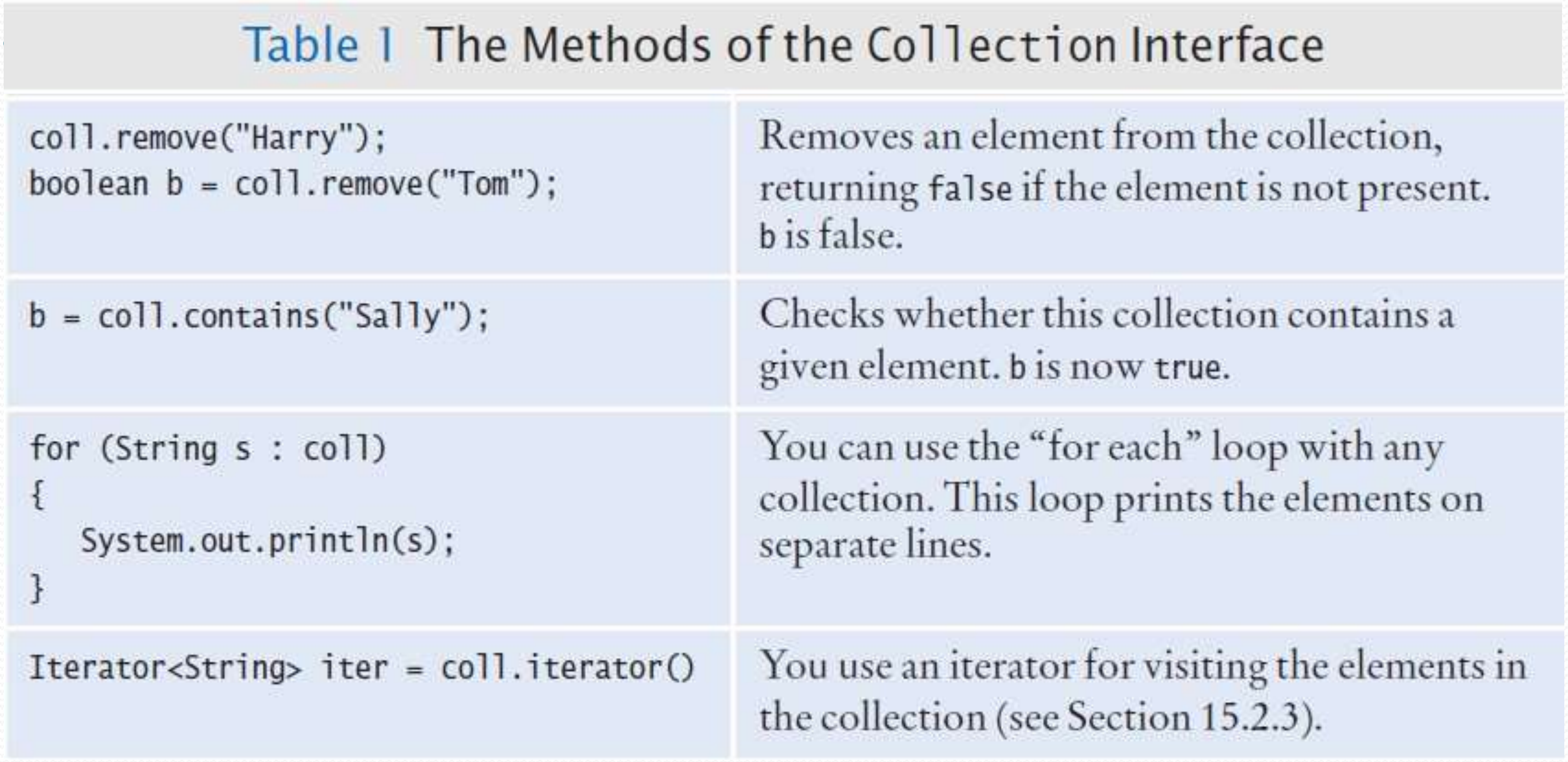


Table 1 The Methods of the Collection Interface

`coll.remove("Harry");`
`boolean b = coll.remove("Tom");`
	Removes an element from the collection, returning `false` if the element is not present. `b` is false.

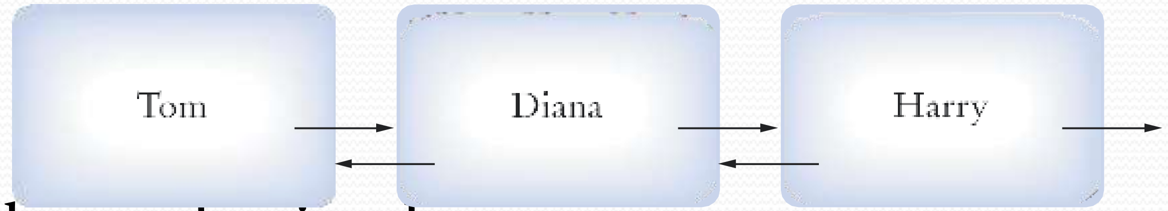
`b = coll.contains("Sally");`	Checks whether this collection contains a given element. `b` is now `true`.

`for (String s : coll)`
`{`
`   System.out.println(s);`
`}`
	You can use the "for each" loop with any collection. This loop prints the elements on separate lines.

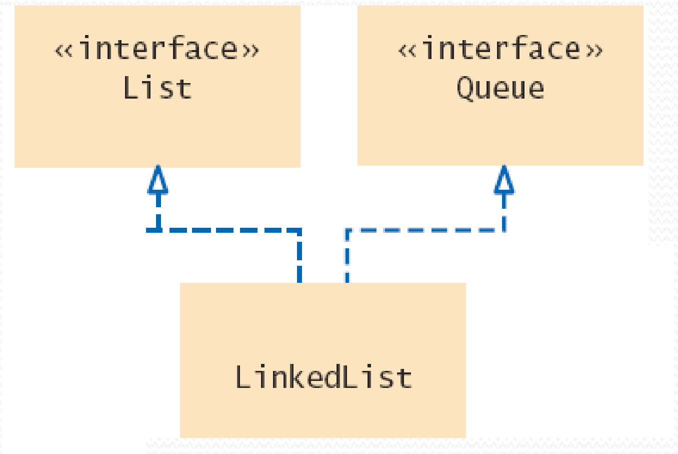
`Iterator<String> iter = coll.iterator()`	You use an iterator for visiting the elements in the collection (see Section 15.2.3).

15.2 Linked Lists

- Linked lists use references to maintain an ordered lists of 'nodes'
  - The 'head' of the list references the first node
  - Each node has a value and a reference to the next node

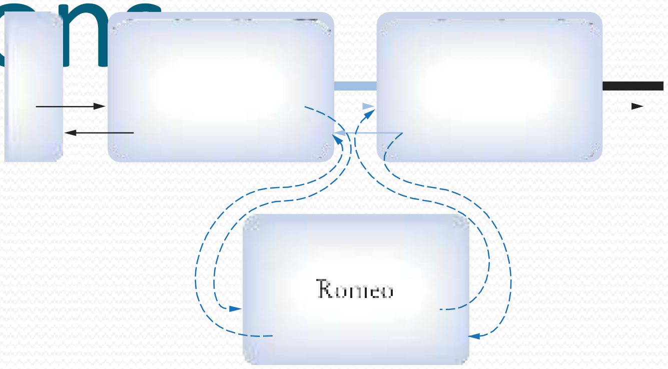


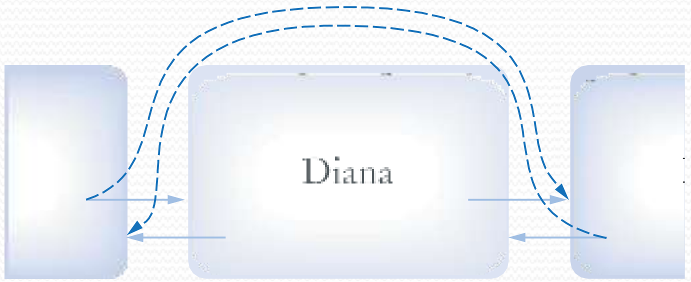
  - They can be used to implement
    - A `List` Interface
    - A `Queue` Interface



Linked Lists Operations

- Efficient Operations
  - Insertion of a node
    - Find the elements it goes between
    - Remap the references
  - Removal of a node
    - Find the element to remove
    - Remap neighbor's references
  - Visiting all elements in order
- Inefficient Operations
  - Random access





Each instance variable is declared just like other variables we have used.

LinkedList: Important Methods

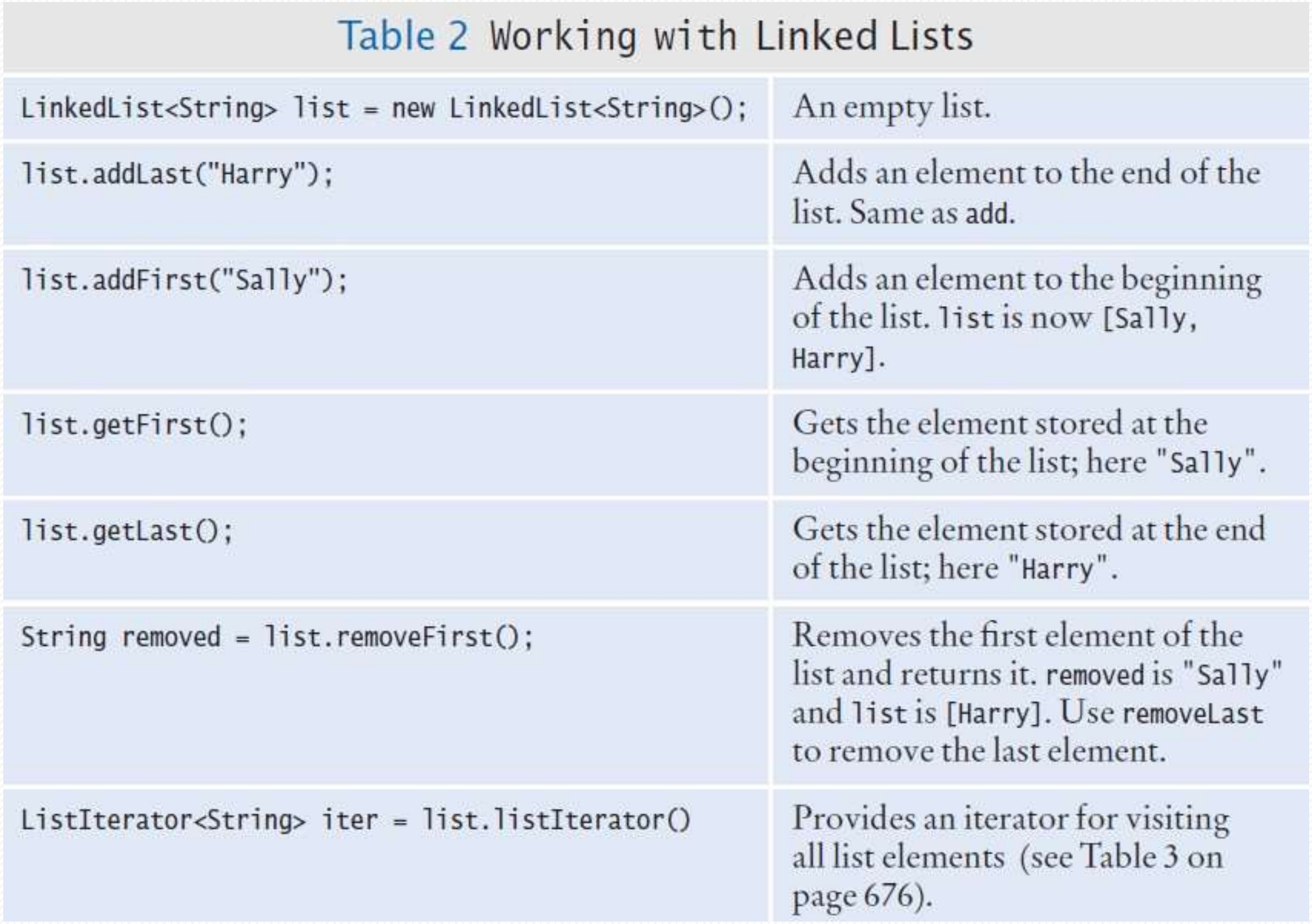


Table 2 Working with Linked Lists

`LinkedList<String> list = new LinkedList<String>();`	An empty list.
`list.addLast("Harry");`	Adds an element to the end of the list. Same as `add`.
`list.addFirst("Sally");`	Adds an element to the beginning of the list. `list` is now `[Sally, Harry]`.
`list.getFirst();`	Gets the element stored at the beginning of the list; here `"Sally"`.
`list.getLast();`	Gets the element stored at the end of the list; here `"Harry"`.
`String removed = list.removeFirst();`	Removes the first element of the list and returns it. `removed` is `"Sally"` and `list` is `[Harry]`. Use `removeLast` to remove the last element.
`ListIterator<String> iter = list.listIterator()`	Provides an iterator for visiting all list elements (see Table 3 on page 676).

Sets

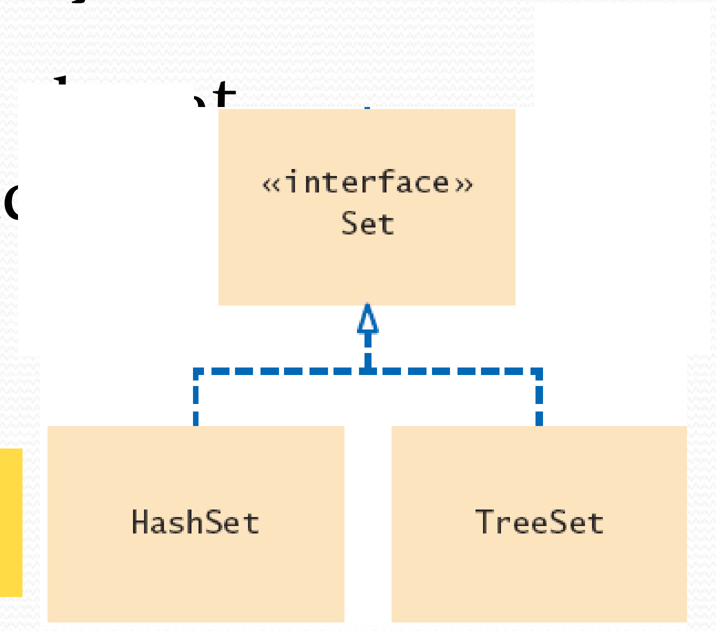
- A set is an unordered collection
  - It does not support duplicate elements
- The collection does not keep track of the order in which elements have been added
  - Therefore, it can carry out its operations more efficiently than an ordered collection

The `HashSet` and `TreeSet` classes both implement the Set interface.

Sets

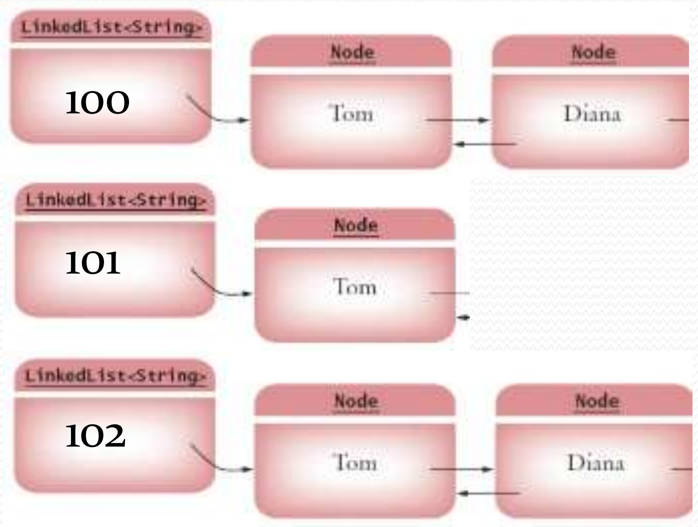
- `HashSet`: Stores data in a Hash Table
- `TreeSet`: Stores data in a Binary Tree
- Both implementations arrange elements so that finding, ac removing elements is efficient

Set implementations arrange the elements so that they can locate them quickly



Hash Table Concept

- Set elements are grouped into smaller collections of elements that share the same characteristic
  - It is usually based on the result of a mathematical calculation on the contents that results in an integer value
  - In order to be stored in a hash table, elements must have a method to compute their integer values



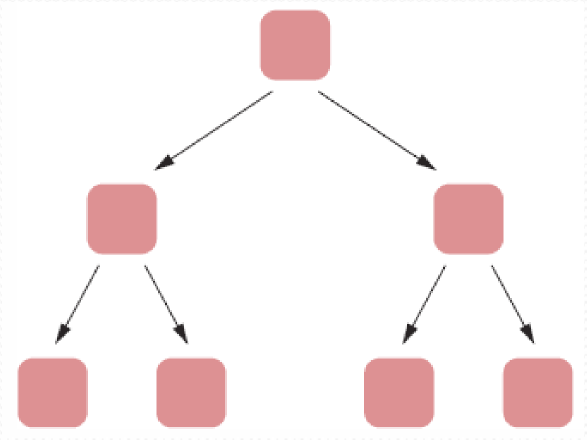
hashCode

- The method is called `hashCode`
  - If multiple elements have the same hash code, they are stored in a Linked list
- The elements must also have an `equals` method for checking whether an element equals another like:
  - String, Integer, Point, Rectangle, Color, and all collection classes

```
Set<String> names = new HashSet<String>();
```

Tree Concept

- Set elements are kept in sorted order
  - Nodes are not arranged in a linear sequence but in a tree shape



  - In order to use a `TreeSet`, it must be possible to compare the elements and determine which one is "larger"

TreeSet

- Use `TreeSet` for classes that implement the `Comparable` interface
  - `String` and `Integer`, for example
  - The nodes are arranged in a 'tree' fashion so that each 'parent' node has up to two child nodes.
    - The node to the left always has a 'smaller' value

```
Set<String> names = new TreeSet<String>();
```

    - The node to the right always has a 'larger' value

Iterators and Sets

- Iterators are also used when processing sets
  - `hasNext` returns true if there is a next element
  - `next` returns a reference to the value of the next element
  - `add` via the iterator is not supported for TreeSet and HashSet

```
Iterator<String> iter = names.iterator();
while (iter.hasNext())
{
   String name = iter.next();
   // Do something with name
}
```

```
for (String name : names)
{
   // Do something with name
}
```

- Note that the elements are not v inserted them.
- They are visited in the order in which the set keeps them:
  - Seemingly random order for a HashSet
  - Sorted order for a TreeSet

Table 4 Working with Sets

`Set<String> names;`	Use the interface type for variable declarations.
`names = new HashSet<String>();`	Use a `TreeSet` if you need to visit the elements in sorted order.
`names.add("Romeo");`	Now `names.size()` is 1.
`names.add("Fred");`	Now `names.size()` is 2.
`names.add("Romeo");`	`names.size()` is still 2. You can't add duplicates.
`if (names.contains("Fred"))`	The `contains` method checks whether a value is contained in the set. In this case, the method returns `true`.

Package java.util

- Contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes (a string tokenizer, a random-number generator, and a bit array).

Interface Summary

- Collection<E>: The root interface in the *collection hierarchy*.
- Comparator<T>: A comparison function, which imposes a *total ordering* on some collection of objects.
- Enumeration<E>: An object that implements the Enumeration interface generates a series of elements, one at a time.
- EventListener: A tagging interface that all event listener interfaces must extend.
- Iterator<E>: An iterator over a collection
- List<E>An ordered collection (also known as a *sequence*).
- ListIterator<E>: An iterator for lists that allows the programmer to traverse the list in either direction, modify the list during iteration, and obtain the iterator's current position in the list.

- Map<K,V>: An object that maps keys to values.
- Observer: A class can implement the Observer interface when it wants to be informed of changes in observable objects.
- Queue<E>: A collection designed for holding elements prior to processing.
- Set<E>: A collection that contains no duplicate elements.
- SortedMap<K,V>: A Map that further provides a *total ordering* on its keys.
- SortedSet<E>: A Set that further provides a *total ordering* on its elements.

Class Summary

- AbstractCollection<E>: This class provides a skeletal implementation of the Collection interface, to minimize the effort required to implement this interface.
- AbstractList<E>: This class provides a skeletal implementation of the List interface to minimize the effort required to implement this interface backed by a "random access" data store (such as an array).
- AbstractMap<K,V>: This class provides a skeletal implementation of the Map interface, to minimize the effort required to implement this interface.
- AbstractQueue<E>: This class provides skeletal implementations of some Queue operations.
- AbstractSequentialList<E>: This class provides a skeletal implementation of the List interface to minimize the effort required to implement this interface backed by a "sequential access" data store (such as a linked list).
- AbstractSet<E>: This class provides a skeletal implementation of the Set interface to minimize the effort required to implement this interface.

- ArrayList<E>: Resizable-array implementation of the List interface
- Arrays: This class contains various methods for manipulating arrays (such as sorting and searching).
- BitSet: This class implements a vector of bits that grows as needed
- Calendar: The Calendar class is an abstract class that provides methods for converting between a specific instant in time and a set of calendar fields: such as YEAR, MONTH, DAY_OF_MONTH, HOUR, and so on, and for manipulating the calendar fields, such as getting the date of the next week

- Collections: This class consists exclusively of static methods that operate on or return collections
- Currency: Represents a currency.
- Date: The class Date represents a specific instant in time, with millisecond precision.
- Dictionary<K,V>: The Dictionary class is the abstract parent of any class, such as Hashtable, which maps keys to values.
- EventObject: The root class from which all event state objects shall be derived.

- GregorianCalendar: GregorianCalendar is a concrete subclass of Calendar and provides the standard calendar system used by most of the world.
- HashMap<K,V>: Hash table based implementation of the Map interface.
- HashSet<E>: This class implements the Set interface, backed by a hash table (actually a HashMap instance)
- .Hashtable<K,V>: This class implements a hashtable, which maps keys to values.
- LinkedList<E>: Linked list implementation of the List interface
- Locale: A Locale object represents a specific geographical, political, or cultural region.
- Observable: This class represents an observable object, or "data" in the model-view paradigm
- Properties: The Properties class represents a persistent set of properties.

- Random: An instance of this class is used to generate a stream of pseudorandom numbers.
- ResourceBundle: Resource bundles contain locale-specific objects.
- SimpleTimeZone: SimpleTimeZone is a concrete subclass of TimeZone that represents a time zone for use with a Gregorian calendar.
- Stack<E>: The Stack class represents a last-in-first-out (LIFO) stack of objects.
- StringTokenizer: The string tokenizer class allows an application to break a string into tokens.
- TimeZone: TimeZone represents a time zone offset, and also figures out daylight savings.
- TreeMap<K,V>: A Red-Black tree based NavigableMap implementation.
- TreeSet<E>: A NavigableSet implementation based on a TreeMap.UUIDA class that represents an immutable universally unique identifier (UUID).
- Vector<E>: The Vector class implements a growable array of objects

Exception Summary

- EmptyStackException: Thrown by methods in the Stack class to indicate that the stack is empty.
- InputMismatchException: Thrown by a Scanner to indicate that the token retrieved does not match the pattern for the expected type, or that the token is out of range for the expected type.
- InvalidPropertiesFormatException: Thrown to indicate that an operation could not complete because the input did not conform to the appropriate XML document type for a collection of properties, as per the Properties specification.
- NoSuchElementException: Thrown by the nextElement method of an Enumeration to indicate that there are no more elements in the enumeration.
- TooManyListenersException: The TooManyListenersException Exception is used as part of the Java Event model to annotate and implement a unicast special case of a multicast Event Source.
- UnknownFormatConversionException: Unchecked exception thrown when an unknown conversion is given.

- Provides for system input and output through data streams, serialization and the file system.

Interface Summary

- DataInput The DataInput interface provides for reading bytes from a binary stream and reconstructing from them data in any of the Java primitive types.

- DataOutputThe DataOutput interface provides for converting data from any of the Java primitive types to a series of bytes and writing these bytes to a binary stream

- .Externalizable Only the identity of the class of an Externalizable instance is written in the serialization stream and it is the responsibility of the class to save and restore the contents of its instances.

- SerializableSerializability of a class is enabled by the class implementing the java.io.Serializable interface.

Class Summary

- BufferedInputStream: A BufferedInputStream adds functionality to another input stream-namely, the ability to buffer the input and to support the mark and reset methods.
- BufferedOutputStream: The class implements a buffered output stream.
- BufferedReader: Reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines.
- BufferedWriter: Writes text to a character-output stream, buffering characters so as to provide for the efficient writing of single characters, arrays, and strings
- ByteArrayInputStream: A ByteArrayInputStream contains an internal buffer that contains bytes that may be read from the stream.
- ByteArrayOutputStream: This class implements an output stream in which the data is written into a byte array.

- CharArrayReader: This class implements a character buffer that can be used as a character-input stream

- .CharArrayWriter: This class implements a character buffer that can be used as an Writer

- Console: Methods to access the character-based console device, if any, associated with the current Java virtual machine.

- DataInputStream: A data input stream lets an application read primitive Java data types from an underlying input stream in a machine-independent way.

- DataOutputStream: A data output stream lets an application write primitive Java data types to an output stream in a portable way.

- File: An abstract representation of file and directory pathnames.
- FileInputStream: A FileInputStream obtains input bytes from a file in a file system.
- FileOutputStream: A file output stream is an output stream for writing data to a File or to a FileDescriptor.
- FileReader: Convenience class for reading character files.
- FileWriter: Convenience class for writing character files.
- FilterInputStream: A FilterInputStream contains some other input stream, which it uses as its basic source of data, possibly transforming the data along the way or providing additional functionality.
- FilterOutputStream: This class is the superclass of all classes that filter output streams
- .FilterReader: Abstract class for reading filtered character streams
- .FilterWriter: Abstract class for writing filtered character streams
- .InputStream: This abstract class is the superclass of all classes representing an input stream of bytes.
- InputStreamReader: An InputStreamReader is a bridge from byte streams to

- ObjectInputStream: An ObjectInputStream deserializes primitive data and objects previously written using an ObjectOutputStream
- ObjectOutputStream: An ObjectOutputStream writes primitive data types and graphs of Java objects to an OutputStream.
- OutputStream: This abstract class is the superclass of all classes representing an output stream of bytes.
- OutputStreamWriter: An OutputStreamWriter is a bridge from character streams to byte streams: Characters written to it are encoded into bytes using a specified charset.
- PrintWriter: Prints formatted representations of objects to a text-output stream.
- RandomAccessFile: Instances of this class support both reading and writing to a random access file.
- StreamTokenizer: The StreamTokenizer class takes an input stream and parses it into "tokens", allowing the tokens to be read one at a time.

Exception Summary

- FileNotFoundException: Signals that an attempt to open the file denoted by a specified pathname has failed.
- InterruptedIOException: Signals that an I/O operation has been interrupted
- InvalidClassException: Thrown when the Serialization runtime detects one of the following problems with a Class.
- InvalidObjectException: Indicates that one or more deserialized objects failed validation tests.
- IOException: Signals that an I/O exception of some sort has occurred.

JDBC connectivity

Introduction

- JDBC stands for **J**ava **D**ata**b**ase **C**onnectivity, which is a standard Java API for database-independent connectivity between the Java programming language and a wide range of databases.

- The JDBC library includes APIs for each of the tasks commonly associated with database usage:

- 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

Required Steps:

- There are following steps required to create a new Database using JDBC application:

- **Import the packages .** Requires that you include the packages containing the JDBC classes needed for database programming. Most often, using *import java.sql.** will suffice.

- **Register the JDBC driver .** Requires that you initialize a driver so you can open a communications channel with the database.

- **Open a connection .** Requires using the *DriverManager.getConnection()* method to create a Connection object, which represents a physical connection with database server.

- To create a new database, you need not to give any database name while preparing database URL as mentioned in the below example.

- **Execute a query .** Requires using an object of type Statement for building and submitting an SQL statement to the database.

- **Clean up the environment .** Requires explicitly closing all database resources versus relying on the JVM's garbage collection.

Creating JDBC Application

- There are six steps involved in building a JDBC application which I'm going to brief in this tutorial:

1.Import the packages:

- To use the standard JDBC package, which allows you to select, insert, update, and delete data in SQL tables, add the following *imports* to your source code:
- //STEP 1. Import required packages
- Syntax :import java.sql.*;

2.Register the JDBC driver:

- This requires that you initialize a driver so you can open a communications channel with the database.
- Registering the driver is the process by which the Oracle driver's class file is loaded into memory so it can be utilized as an implementation of the JDBC interfaces.
- You need to do this registration only once in your program

- //STEP 2: Register JDBC driver
- Syntax:Class.forName("com.mysql.jdbc.Driver");

- **Open a connection**:

- After you've loaded the driver, you can establish a connection using the **DriverManager.getConnection()** method, which represents a physical connection with the database as follows:
- //STEP 3: Open a connection // Database credentials

➢ static final String USER = "username";

➢ static final String PASS = "password";

➢ System.out.println("Connecting to database...");

➢ conn = DriverManager.getConnection(DB_URL,USER,PASS);

- **Execute a query**:
- This requires using an object of type Statement or PreparedStatement for building and submitting an SQL statement to the database as follows:
- //STEP 4: Execute a query

➢ System.out.println("Creating statement...");

➢ stmt = conn.createStatement();

➢ String sql; sql = "SELECT id, first, last, age FROM Employees";

➢ ResultSet rs = stmt.executeQuery(sql);

- Following table lists down popular JDBC driver names and database URL.

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

- All the highlighted part in URL format is static and you need to change only remaining part as per your database setup.

- If there is an SQL UPDATE,INSERT or DELETE statement required, then following code snippet would be required:
- //STEP 4: Execute a query

- System.out.println("Creating statement...");
- stmt = conn.createStatement();
- String sql;
- sql = "DELETE FROM Employees";
- ResultSet rs = stmt.executeUpdate(sql);

- **Extract data from result set**:
- This step is required in case you are fetching data from the database. You can use the appropriate ResultSet.getXXX() method to retrieve the data from the result set as follows:
- //STEP 5: Extract data from result set
- while(rs.next())
- {
- //Retrieve by column name
- int id = rs.getInt("id");
- int age = rs.getInt("age");
- String first = rs.getString("first");
- String last = rs.getString("last");
- //Display values
- System.out.print("ID: " + id);
- System.out.print(", Age: " + age);
- System.out.print(", First: " + first);
- System.out.println(", Last: " + last); }

- **Clean up the environment:**
- You should explicitly close all database resources versus relying on the JVM's garbage collection as follows:

- //STEP 6: Clean-up environment

```
rs.close();
stmt.close();
conn.close();
```

JDBC Driver

- JDBC drivers implement the defined interfaces in the JDBC API for interacting with your database server.
- For example, using JDBC drivers enable you to open database connections and to interact with it by sending SQL or database commands then receiving results with Java.
- The *Java.sql* package that ships with JDK contains various classes with their behaviours defined and their actual implementaions are done in third-party drivers.
- Third party vendors implements the *java.sql.Driver* interface in their database driver.
- **JDBC Drivers Types:**
- JDBC driver implementations vary because of the wide variety of operating systems and hardware platforms in which Java operates. Sun has divided the implementation types into four categories, Types 1, 2, 3, and 4

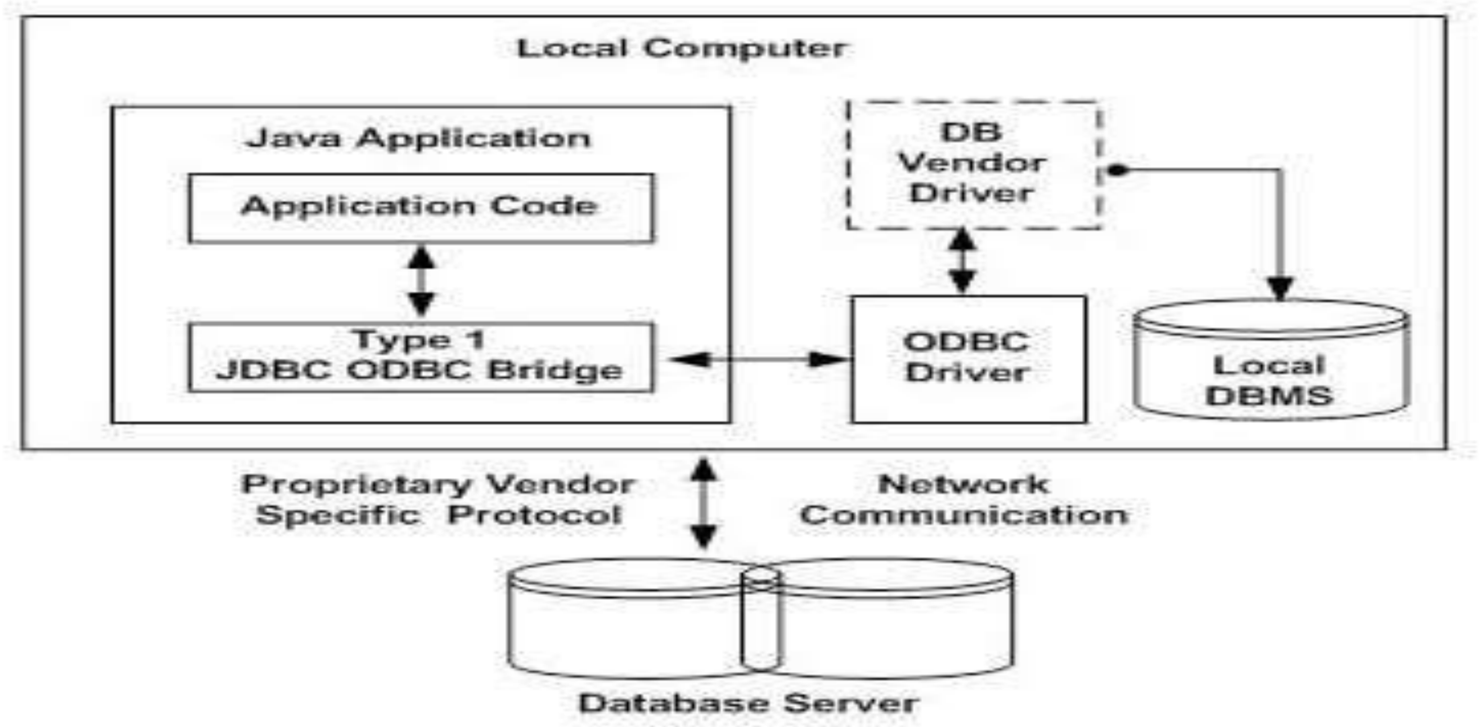
Type 1: JDBC-ODBC Bridge Driver

- In a Type 1 driver, a JDBC bridge is used to access ODBC drivers installed on each client machine.
- Using ODBC requires configuring on your system a Data Source Name (DSN) that represents the target database.

- When Java first came out, this was a useful driver because most databases only supported ODBC access but now this type of driver is recommended only for experimental use or when no other alternative is available.

- The JDBC-ODBC bridge that comes with JDK 1.2 is a good example of this kind of driver.

Type 1: JDBC-ODBC Bridge Driver

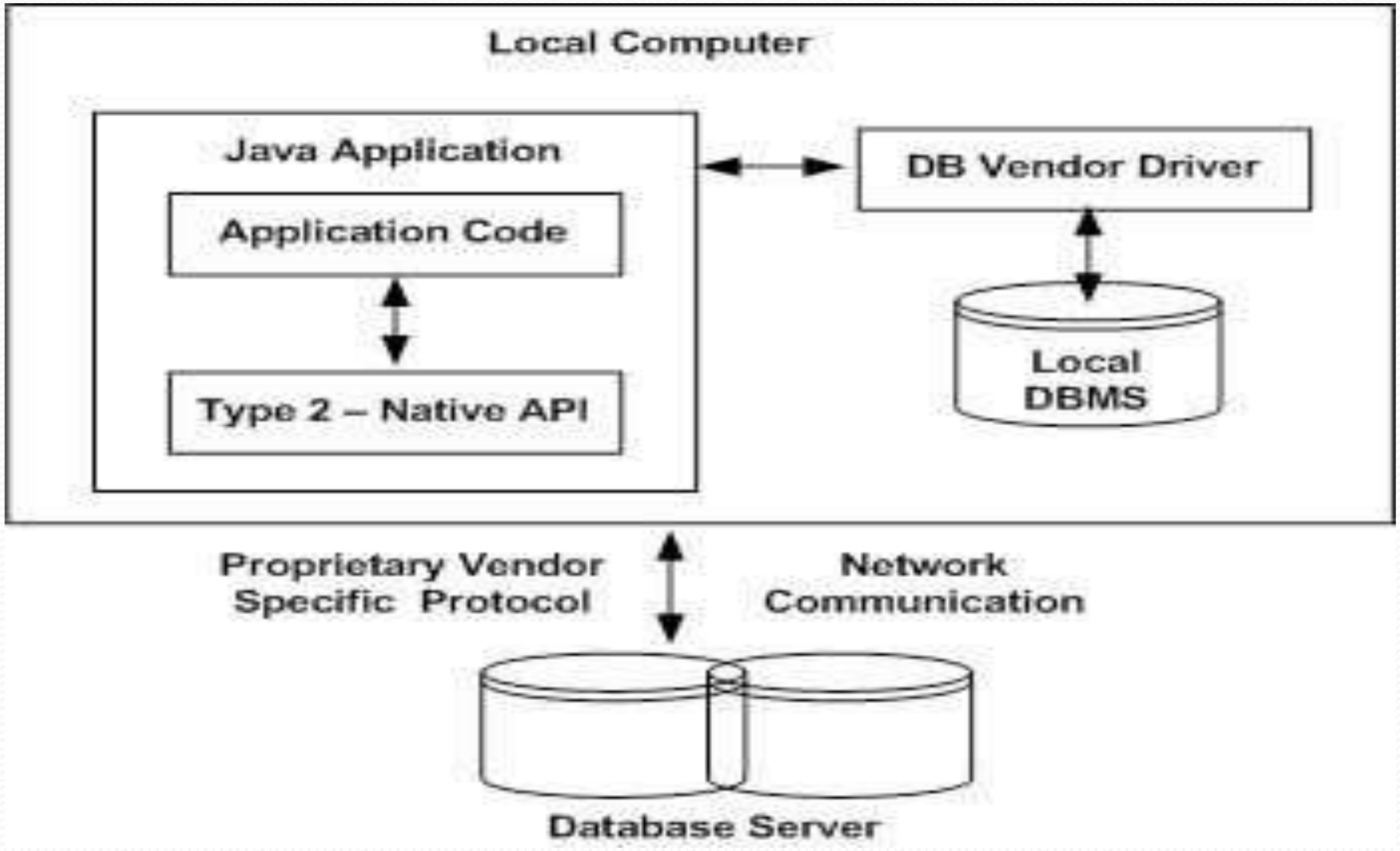


Type 2: JDBC-Native API

- In a Type 2 driver, JDBC API calls are converted into native C/C++ API calls which are unique to the database.
- These drivers typically provided by the database vendors and used in the same manner as the JDBC-ODBC Bridge, the vendor-specific driver must be installed on each client machine.
- If we change the Database we have to change the native API as it is specific to a database and they are mostly obsolete now but you may realize some speed increase with a Type 2 driver, because it eliminates ODBC's overhead.

- The Oracle Call Interface (OCI) driver is an example of a Type 2 driver.

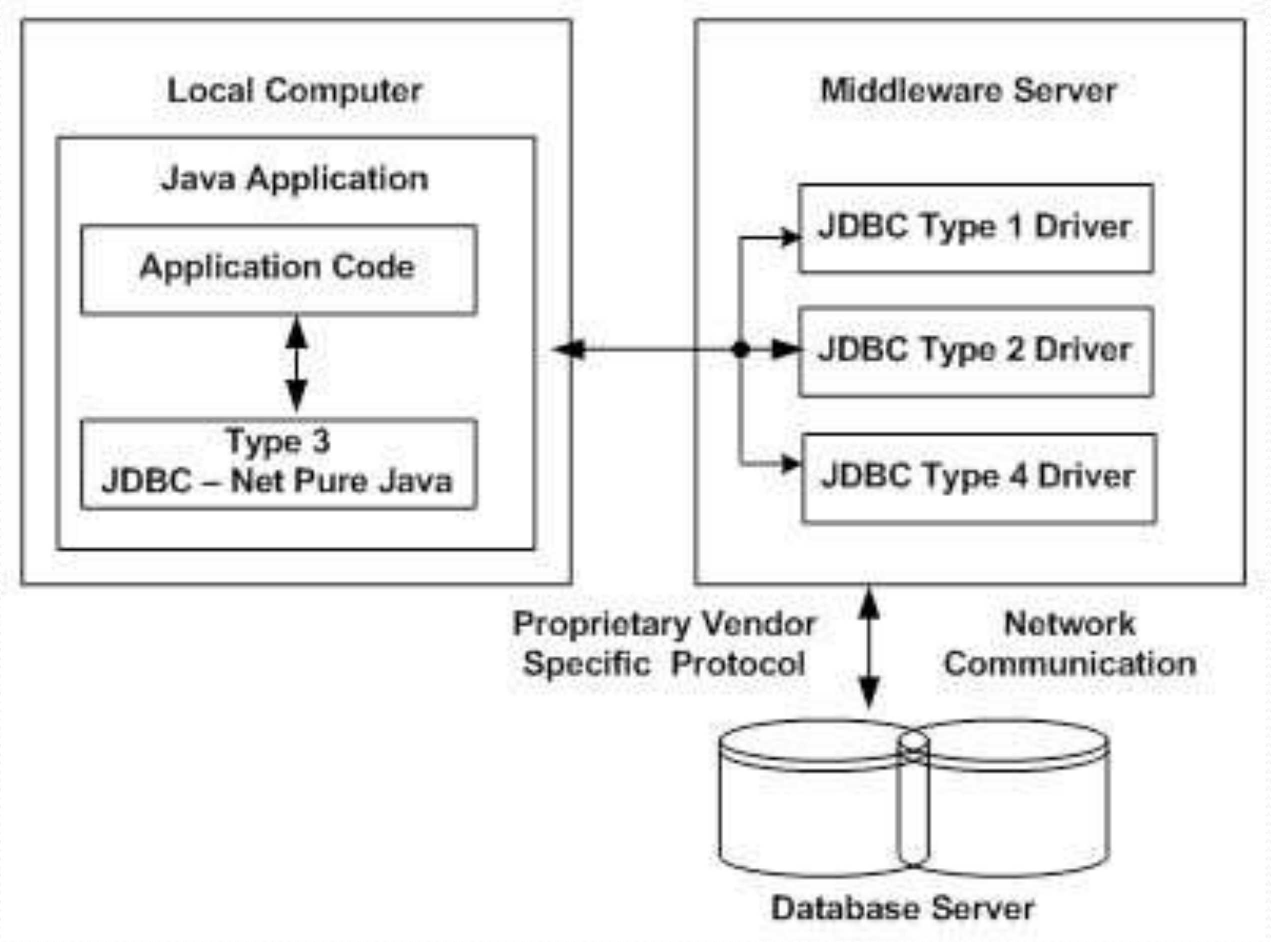
Type 2: JDBC-Native API



Type 3: JDBC-Net pure Java

- In a Type 3 driver, a three-tier approach is used to accessing databases.
- The JDBC clients use standard network sockets to communicate with an middleware application server. The socket information is then translated by the middleware application server into the call format required by the DBMS, and forwarded to the database server.
- This kind of driver is extremely flexible, since it requires no code installed on the client and a single driver can actually provide access to multiple databases

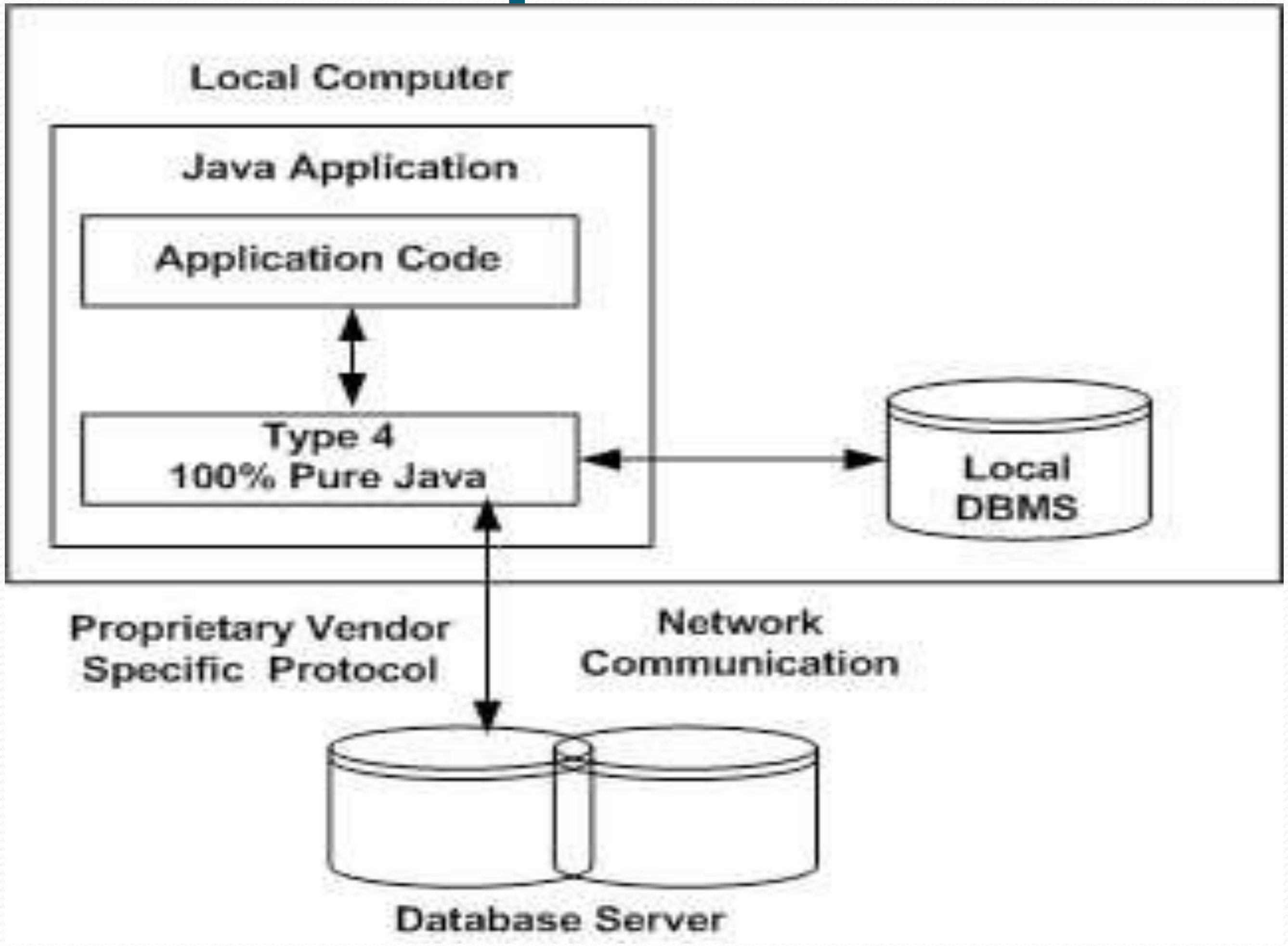
Type 3: JDBC-Net pure Java



Type 4: 100% pure Java

- In a Type 4 driver, a pure Java-based driver that communicates directly with vendor's database through socket connection.
- This is the highest performance driver available for the database and is usually provided by the vendor itself.
- This kind of driver is extremely flexible, you don't need to install special software on the client or server. Further, these drivers can be downloaded dynamically.
- MySQL's Connector/J driver is a Type 4 driver. Because of the proprietary nature of their network protocols, database vendors usually supply type 4 drivers.

Type 4: 100% pure Java



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 yet for your database.
- The type 1 driver is not considered a deployment-level driver and is typically used for development and testing purposes only.

UNIT-5

TOPICS

1 Events, Event sources, Event classes,

2 Event Listeners, Delegation event model

3 Handling mouse and keyboard events, Adapter classes, inner classes.

4 The AWT class hierarchy,

5 user interface components- labels, button, canvas, scrollbars, text

6 components, check box, check box groups, choices

7 lists panels – scrollpane, dialogs

8 menubar, graphics

9 layout manager – layout manager types –boarder, grid, flow, card and grib bag

TOPICS

- Concepts of Applets, differences between applets and applications
- Life cycle of an applet, types of applets
- Creating applets, passing parameters to applets.
- Introduction to swings, limitations of AWT
- components, containers
- Exploring swing- JApplet, JFrame and JComponent
- Icons and Labels, text fields, buttons
- Check boxes, Combo boxes,RadioButton,JButton
- Tabbed Panes, Scroll Panes, Trees, and Tables

Event handling

- For the user to interact with a GUI, the underlying operating system must support event handling.
  1) operating systems constantly monitor events such as keystrokes, mouse clicks, voice command, etc.
  2) operating systems sort out these events and report them to the appropriate application programs
  3)each application program then decides what to do in response to these events

Events

- An *event* is an object that describes a state change in a source.
- It can be generated as a consequence of a person interacting with the elements in a graphical user interface.
- Some of the activities that cause events to be generated are pressing a button, entering a character via the keyboard, selecting an item in a list, and clicking the mouse.
- Events may also occur that are not directly caused by interactions with a user interface.
- For example, an event may be generated when a timer expires, a counter exceeds a value, a software or hardware failure occurs, or an operation is completed.
- Events can be defined as needed and appropriate by application.

Event sources

- A *source* is an object that generates an event.
- This occurs when the internal state of that object changes in some way.
- Sources may generate more than one type of event.
- A source must register listeners in order for the listeners to receive notifications about a specific type of event.
- Each type of event has its own registration method.
- General form is:

  public void add*Type*Listener(*Type*Listener *el*)

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

  1.The method that registers a keyboard event listener is called **addKeyListener()**.

  2.The method that registers a mouse motion listener is called

  addMouseMotionListener( ).

- When an event occurs, all registered listeners are notified and receive a copy of the event object. This is known as *multicasting* the event.

- In all cases, notifications are sent only to listeners that register to receive them.

- Some sources may allow only one listener to register. The general form is:
  public void add*Type*Listener(*Type*Listener *el*) throws java.util.TooManyListenersException

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

- When such an event occurs, the registered listener is notified. This is known as *unicasting* the event.

- A source must also provide a method that allows a listener to unregister an interest in a specific type of event.

- The general form is:
  public void remove*Type*Listener(*Type*Listener *el*)
  Here, *Type* is the name of the event and *el* is a reference to the event listener.

- For example, to remove a keyboard listener, you would call **removeKeyListener**( ).

- The methods that add or remove listeners are provided by the source that generates events.

- For example, the **Component** class provides methods to add and remove keyboard and mouse event listeners.

Event classes

- The Event classes that represent events are at the core of Java's event handling mechanism.
- Super class of the Java event class hierarchy is **EventObject**, which is in **java.util**. for all events.
- Constructor is :

  EventObject(Object *src*)

  Here, *src* is the object that generates this event.
- **EventObject** contains two methods: **getSource( )** and **toString( )**.
- 1. The **getSource( )** method returns the source of the event. General form is : Object getSource( )
- **2.** The **toString( )** returns the string equivalent of the event.

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

Event Classes in java.awt.event

- 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.
- 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.
- 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.

Event Listeners

- A *listener* is an object that is notified when an event occurs.
- Event has two major requirements.
  1. It must have been registered with one or more sources to receive
     notifications about specific types of events.
  2. It must implement methods to receive and process these
     notifications.
- The methods that receive and process events are defined in a set of interfaces found in **java.awt.event**.
- For example, the **MouseMotionListener** interface defines two methods to receive notifications when the mouse is dragged or moved.
- Any object may receive and process one or both of these events if it provides an implementation of this interface.

Delegation event model

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

- In the delegation event model, listeners must register with a source in order to receive an event notification. This provides an important benefit: notifications are sent only to listeners that want to receive them.

- This is a more efficient way to handle events than the design used by the old Java 1.0 approach. Previously, an event was propagated up the containment hierarchy until it was handled by a component.

- This required components to receive events that they did not process, and it wasted valuable time.The delegation event model eliminates this overhead.
  Note

- Java also allows you to process events without using the delegation event model.

- This can be done by extending an AWT component.

Handling mouse events

- mouse events can be handled by implementing the **MouseListener** and the **MouseMotionListener** interfaces.

- **MouseListener Interface** defines five methods. The general forms of these methods are:
  1. void mouseClicked(MouseEvent me)
  2. void mouseEntered(MouseEvent me)
  3. void mouseExited(MouseEvent me)
  4. void mousePressed(MouseEvent me)
  5. void mouseReleased(MouseEvent me)

- **MouseMotionListener Interface.** This interface defines two methods. Their general forms are :
  1. void mouseDragged(MouseEvent me)
  2. void mouseMoved(MouseEvent me)

Handling keyboard events

- Keyboard events, can be handled by implementing the **KeyListener** interface.
- **KeyListner** interface defines three methods. The general forms of these methods are :
    1. void keyPressed(KeyEvent ke)
    2. void keyReleased(KeyEvent ke)
    3. void keyTyped(KeyEvent ke)
- To implement keyboard events implementation to the above methods is needed.

Adapter classes

- Java provides a special feature, called an *adapter class*, that can simplify the creation of event handlers.
- An 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.
- You can define a new class to act as an event listener by extending one of the adapter classes and implementing only those events in which you are interested.

Adapter classes in **java.awt.event are**.

Adapter Class	**Listener Interface**
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
FocusAdapter	FocusListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
WindowAdapter	WindowListener

Inner classes

- Inner classes, which allow one class to be defined within another.

- An inner class is a non-static nested class. It has access to all of the variables and methods of its outer class and may refer to them directly in the same way that other non-static members of the outer class do.

- An inner class is fully within the scope of its enclosing class.
- an inner class has access to all of the members of its enclosing class, but the reverse is not true.

- Members of the inner class are known only within the scope of the inner class and may not be used by the outer class

The AWT class hierarchy

- The AWT classes are contained in the **java.awt** package. It is one of Java's largest packages. some of the AWT classes.
- **<u>AWT Classes</u>**
  1. AWTEvent:Encapsulates AWT events.
  2. AWTEventMulticaster: Dispatches events to multiple listeners.
  3. BorderLayout: The border layout manager. Border layouts use five components: North, South, East, West, and Center.
  4. Button: Creates a push button control.
  5. Canvas: A blank, semantics-free window.
  6. CardLayout: The card layout manager. Card layouts emulate index cards. Only the one on top is showing.

7. Checkbox: Creates a check box control.
8. CheckboxGroup: Creates a group of check box controls.
9. CheckboxMenuItem: Creates an on/off menu item.
10. Choice: Creates a pop-up list.
11. Color: Manages colors in a portable, platform-independent fashion.
12. Component: An abstract super class for various AWT components.
13. Container: A subclass of Component that can hold other components.
14. Cursor: Encapsulates a bitmapped cursor.
15. Dialog: Creates a dialog window.
16. Dimension: Specifies the dimensions of an object. The width is stored in width, and the height is stored in height.
17. Event: Encapsulates events.
18. EventQueue: Queues events.
19. FileDialog: Creates a window from which a file can be selected.
20. FlowLayout: The flow layout manager. Flow layout positions components left to right, top to bottom.

21. Font: Encapsulates a type font.
22. FontMetrics: Encapsulates various information related to a font. This information helps you display text in a window.
23. Frame: Creates a standard window that has a title bar, resize corners, and a menu bar.
24. Graphics: Encapsulates the graphics context. This context is used by various output methods to display output in a window.
25. GraphicsDevice: Describes a graphics device such as a screen or printer.
26. GraphicsEnvironment: Describes the collection of available Font and GraphicsDevice objects.
21. GridBagConstraints: Defines various constraints relating to the GridBagLayout class.
22. GridBagLayout: The grid bag layout manager. Grid bag layout displays components subject to the constraints specified by GridBagConstraints.
23. GridLayout: The grid layout manager. Grid layout displays components in a two-dimensional grid.

30. Scrollbar: Creates a scroll bar control.
31. ScrollPane: A container that provides horizontal and/or vertical scrollbars for another component.
32. SystemColor: Contains the colors of GUI widgets such as windows, scrollbars, text, and others.
33. TextArea: Creates a multiline edit control.
34. TextComponent: A super class for TextArea and TextField.
35. TextField: Creates a single-line edit control.
36. Toolkit: Abstract class implemented by the AWT.
37. Window: Creates a window with no frame, no menu bar, and no title.

user interface components

- Labels: Creates a label that displays a string.
- A *label* is an object of type **Label**, and it contains a string, which it displays.
- Labels are passive controls that do not support any interaction with the user.
- **Label** defines the following constructors:
    1. Label( )
    2. Label(String str)
    3. Label(String str, int how)
- The first version creates a blank label.
- The second version creates a label that contains the string specified by *str*. This string is left-justified.
- The third version creates a label that contains the string specified by *str* using the alignment specified by *how*. The value of *how* must be one of these three constants: **Label.LEFT**, **Label.RIGHT**, or **Label.CENTER**.

- Set or change the text in a label is done by using the **setText( )** method.
- Obtain the current label by calling **getText( )**.
- These methods are shown here:

```
void setText(String str)
String getText( )
```

- For **setText( )**, *str* specifies the new label. For **getText( )**, the current label is returned.
- To set the alignment of the string within the label by calling **setAlignment( )**.
- To obtain the current alignment, call **getAlignment( )**.
- The methods are as follows:

```
void setAlignment(int how)
int getAlignment( )

Label creation:  Label one = new Label("One");
```

Button

- The most widely used control is the push button.
- A *push button* is a component that contains a label and that generates an event when it is pressed.
- Push buttons are objects of type **Button**. **Button** defines these two constructors:

  Button( )

  Button(String str)
- The first version creates an empty button. The second creates a button that contains *str* as a label.
- After a button has been created, you can set its label by calling **setLabel( )**.
- You can retrieve its label by calling **getLabel( )**.
- These methods are as follows:

void setLabel(String str)
String getLabel( )

Here, *str* becomes the new label for the button.

Button creation: Button yes = new Button("Yes");

canvas

- It is not part of the hierarchy for applet or frame windows
- **Canvas** encapsulates a blank window upon which you can draw.
- Canvas creation:

```
        Canvas c = new Canvas();
        Image test = c.createImage(200, 100);
```

- This creates an instance of **Canvas** and then calls the **createImage( )** method to actually make an **Image** object. At this point, the image is blank.

scrollbars

- Scrollbar generates adjustment events when the scroll bar is manipulated.
- Scrollbar creates a scroll bar control.
- *Scroll bars* are used to select continuous values between a specified minimum and maximum.
- Scroll bars may be oriented horizontally or vertically.
- A scroll bar is actually a composite of several individual parts.
- Each end has an arrow that you can click to move the current value of the scroll bar one unit in the direction of the arrow.
- The current value of the scroll bar relative to its minimum and maximum values is indicated by the *slider box* (or *thumb*) for the scroll bar.
- The slider box can be dragged by the user to a new position. The scroll bar will then reflect this value.

- **Scrollbar** defines the following constructors:
  Scrollbar( )
  Scrollbar(int style)
  Scrollbar(int style, int initialValue, int thumbSize, int min, int max)
- The first form creates a vertical scroll bar.
- The second and third forms allow you to specify the orientation of the scroll bar. If *style* is **Scrollbar.VERTICAL**, a vertical scroll bar is created. If *style* is **Scrollbar.HORIZONTAL**, the scroll bar is horizontal.
- In the third form of the constructor, the initial value of the scroll bar is passed in *initialValue*.
- The number of units represented by the height of the thumb is passed in *thumbSize*.
- The minimum and maximum values for the scroll bar are specified by *min* and *max*.
- vertSB = new Scrollbar(Scrollbar.VERTICAL, 0, 1, 0, height);
- horzSB = new Scrollbar(Scrollbar.HORIZONTAL, 0, 1, 0, width);

Text

- **Text is created by Using a TextField class**
- The **TextField** class implements a single-line text-entry area, usually called an *edit*
- *control.*
- Text fields allow the user to enter strings and to edit the text using the arrow
- keys, cut and paste keys, and mouse selections.
- **TextField** is a subclass of **TextComponent**. **TextField** defines the following constructors:

  TextField( )

  TextField(int numChars)

  TextField(String str)

  TextField(String str, int numChars)

- The first version creates a default text field.
- The second form creates a text field that is *numChars* characters wide.
- The third form initializes the text field with the string contained in *str.*
- The fourth form initializes a text field and sets its width.
- **TextField** (and its superclass **TextComponent**) provides several methods that allow you to utilize a text field.
- To obtain the string currently contained in the text field, call **getText()**.
- To set the text, call **setText( )**. These methods are as follows:

```
String getText( )
void setText(String str)
```

Here, *str* is the new string.

Components

- At the top of the AWT hierarchy is the **Component** class.
- **Component** is an abstract class that encapsulates all of the attributes of a visual component.
- All user interface elements that are displayed on the screen and that interact with the user are subclasses of **Component**.
- It defines public methods that are responsible for managing events, such as mouse and keyboard input, positioning and sizing the window, and repainting.
- A **Component** object is responsible for remembering the current foreground and background colors and the currently selected text font.

- To add components

  ```
  Component add(Component compObj)
  ```

  Here, *compObj* is an instance of the control that you want to add. A reference to *compObj* is returned.
  Once a control has been added, it will automatically be visible whenever its parent window is displayed.
- To remove a control from a window when the control is no longer needed call **remove( )**.
- This method is also defined by **Container**. It has this general form:

  ```
  void remove(Component obj)
  ```

  Here, *obj* is a reference to the control you want to remove. You can remove all controls by calling **removeAll( )**.

check box,

- A *check box* is a control that is used to turn an option on or off. It consists of a small box that can either contain a check mark or not.
- There is a label associated with each check box that describes what option the box represents.
- You can change the state of a check box by clicking on it.
- Check boxes can be used individually or as part of a group.
- Checkboxes are objects of the **Checkbox** class.

- **Checkbox** supports these constructors:
  1. Checkbox( )
  2. Checkbox(String str)
  3. Checkbox(String str, boolean on)
  4. Checkbox(String str, boolean on, CheckboxGroup cbGroup)
  5. Checkbox(String str, CheckboxGroup cbGroup, boolean on)
- The first form creates a check box whose label is initially blank. The state of the check box is unchecked.
- The second form creates a check box whose label is specified by *str*. The state of the check box is unchecked.
- The third form allows you to set the initial state of the check box. If *on* is **true**, the check box is initially checked; otherwise, it is cleared.
- The fourth and fifth forms create a check box whose label is specified by *str* and whose group is specified by *cbGroup*. If this check box is not part of a group, then *cbGroup* must be **null**. (Check box groups are described in the next section.) The value of *on* determines the initial state of the check box.

- To retrieve the current state of a check box, call **getState( )**.
- To set its state, call **setState( )**.
- To obtain the current label associated with a check box by calling **getLabel()**.
- To set the label, call **setLabel( )**.
- These methods are as follows:

  boolean getState( )

  void setState(boolean on)

  String getLabel( )

  void setLabel(String str)

  Here, if *on* is **true**, the box is checked. If it is **false**, the box is

  cleared.

  Checkbox creation:

  CheckBox Win98 = new Checkbox("Windows 98", null, true);

check box groups

- It is possible to create a set of mutually exclusive check boxes in which one and only one check box in the group can be checked at any one time.
- These check boxes are oftenccalled *radio buttons*.
- To create a set of mutually exclusive check boxes, you must first define the group to which they will belong and then specify that group when you construct the check boxes.
- Check box groups are objects of type **CheckboxGroup**. Only the default constructor is defined, which creates an empty group.
- To determine which check box in a group is currently selected by calling **getSelectedCheckbox( )**.
- To set a check box by calling **setSelectedCheckbox( )**.
- These methods are as follows:

  Checkbox getSelectedCheckbox( )

  void setSelectedCheckbox(Checkbox which)

  Here, *which* is the check box that you want to be selected. The previously selected checkbox will be turned off.

  - CheckboxGroup cbg = new CheckboxGroup();
  - Win98 = new Checkbox("Windows 98", cbg, true);
  - winNT = new Checkbox("Windows NT", cbg, false);

choices

- The **Choice** class is used to create a *pop-up list* of items from which the user may choose.
- A **Choice** control is a form of menu.
- **Choice** only defines the default constructor, which creates an empty list.
- To add a selection to the list, call **addItem( )** or **add( )**.

```
void addItem(String name)
void add(String name)
```

- Here, *name* is the name of the item being added.
- Items are added to the list in the order to determine which item is currently selected, you may call either **getSelectedItem( )** or **getSelectedIndex( )**.

```
String getSelectedItem( )
int getSelectedIndex( )
```

lists

- The **List** class provides a compact, multiple-choice, scrolling selection list.
- **List** object can be constructed to show any number of choices in the visible window.
- It can also be created to allow multiple selections. **List** provides these constructors:

```
List( )
List(int numRows)
List(int numRows, boolean multipleSelect)
```

- To add a selection to the list, call **add( )**. It has the following two forms:

```
void add(String name)
void add(String name, int index)
```

  - Ex: List os = new List(4, true);

panels

- The **Panel** class is a concrete subclass of **Container**.
- It doesn't add any new methods; it simply implements **Container**.
- A **Panel** may be thought of as a recursively nestable, concrete screen component. **Panel** is the superclass for **Applet**.
- When screen output is directed to an applet, it is drawn on the surface of a **Panel** object.
- **Panel** is a window that does not contain a title bar, menu bar, or border.
- Components can be added to a **Panel** object by its **add( )** method (inherited from **Container**). Once these components have been added, you can position and resize them manually using the **setLocation( )**, **setSize( )**, or **setBounds( )** methods defined by **Component**.
- Ex: Panel osCards = new Panel();

```
        CardLayout cardLO = new CardLayout();
        osCards.setLayout(cardLO);
```

scrollpane

- A *scroll pane* is a component that presents a rectangular area in which a component may be viewed.

- Horizontal and/or vertical scroll bars may be provided if necessary.

- constants are defined by the **ScrollPaneConstants** interface.
  1. HORIZONTAL_SCROLLBAR_ALWAYS
  2. HORIZONTAL_SCROLLBAR_AS_NEEDED
  3. VERTICAL_SCROLLBAR_ALWAYS
  4. VERTICAL_SCROLLBAR_AS_NEEDED

dialogs

- Dialog class creates a dialog window.
- constructors are :

  Dialog(Frame parentWindow, boolean mode)

  Dialog(Frame parentWindow, String title, boolean mode)
- The dialog box allows you to choose a method that should be invoked when the button is clicked.
- Ex: Font   f = new Font("Dialog", Font.PLAIN, 12);

menubar

- MenuBar class creates a menu bar.
- A top-level window can have a menu bar associated with it. A menu bar displays a list of top-level menu choices. Each choice is associated with a drop-down menu.
- To create a menu bar, first create an instance of **MenuBar**.
- This class only defines the default constructor. Next, create instances of **Menu** that will define the selections displayed on the bar.
- Following are the constructors for **Menu**:

  Menu( )
  Menu(String optionName)
  Menu(String optionName, boolean removable)

- Once you have created a menu item, you must add the item to a **Menu** object by using
  MenuItem add(MenuItem item)
- Here, *item* is the item being added. Items are added to a menu in the order in which the calls to **add( )** take place.
- Once you have added all items to a **Menu** object, you can add that object to the menu bar by using this version of **add( )** defined by **MenuBar**:
- Menu add(Menu menu)

Graphics

- The AWT supports a rich assortment of graphics methods.
- All graphics are drawn relative to a window.
- A *graphics context* is encapsulated by the **Graphics** class
- It is passed to an applet when one of its various methods, such as paint( ) or update( ), is called.
- It is returned by the getGraphics( ) method of Component.
- The **Graphics** class defines a number of drawing functions. Each shape can be drawn edge-only or filled.
- Objects are drawn and filled in the currently selected graphics color, which is black by default.
- When a graphics object is drawn that exceeds the dimensions of the window, output is automatically clipped
- Ex:

```
Public void paint(Graphics g)
{
G.drawString("welcome",20,20);
}
```

Layout manager

- A layout manager automatically arranges your controls within a window by using some type of algorithm.
- it is very tedious to manually lay out a large number of components and sometimes the width and height information is not yet available when you need to arrange some control, because the native toolkit components haven't been realized.
- Each **Container** object has a layout manager associated with it.
- A layout manager is an instance of any class that implements the **LayoutManager** interface.
- The layout manager is set by the **setLayout( )** method. If no call to **setLayout( )** is made, then the default layout manager is used.
- Whenever a container is resized (or sized for the first time), the layout manager is used to position each of the components within it.

Layout manager types

Layout manager class defines the following types of layout managers

- Boarder Layout
- Grid Layout
- Flow Layout
- Card Layout
- GridBag Layout

Boarder layout

- The **BorderLayout** class implements a common layout style for top-level windows. It has four narrow, fixed-width components at the edges and one large area in the center.
- The four sides are referred to as north, south, east, and west. The middle area is called the center.
- The constructors defined by **BorderLayout**:

```
BorderLayout( )
BorderLayout(int horz, int vert)
```

- **BorderLayout** defines the following constants that specify the regions:

```
BorderLayout.CENTER
B  orderLayout.SOUTH
BorderLayout.EAST
B    orderLayout.WEST
BorderLayout.NORTH
```

- Components can be added   by

```
void add(Component compObj, Object region);
```

Grid layout

- **GridLayout** lays out components in a two-dimensional grid. When you instantiate a
- **GridLayout**, you define the number of rows and columns. The constructors are

```
GridLayout( )
GridLayout(int numRows, int numColumns )
GridLayout(int numRows, int numColumns, int horz, int vert)
```

- The first form creates a single-column grid layout.
- The second form creates a grid layout
- with the specified number of rows and columns.
- The third form allows you to specify the horizontal and vertical space left between components in *horz* and *vert*, respectively.
- Either *numRows* or *numColumns* can be zero. Specifying *numRows* as zero allows for unlimited-length columns. Specifying *numColumns* as zero allows for unlimited-lengthrows.

Flow layout

- **FlowLayout** is the default layout manager.
- Components are laid out from the upper-left corner, left to right and top to bottom. When no more components fit on a line, the next one appears on the next line. A small space is left between each component, above and below, as well as left and right.
- The constructors are

  FlowLayout( )

  FlowLayout(int how)

  FlowLayout(int how, int horz, int vert)
- The first form creates the default layout, which centers components and leaves five pixels of space between each component.
- The second form allows to specify how each line is aligned. Valid values for are:

  FlowLayout.LEFT

  FlowLayout.CENTER

  FlowLayout.RIGHT

  These values specify left, center, and right alignment, respectively.
- The third form allows to specify the horizontal and vertical space left between components in *horz* and *vert,* respectively

Card layout

- The **CardLayout** class is unique among the other layout managers in that it stores several different layouts.
- Each layout can be thought of as being on a separate index card in a deck that can be shuffled so that any card is on top at a given time.
- **CardLayout** provides these two constructors:

  CardLayout( )

  CardLayout(int horz, int vert)
- The cards are held in an object of type **Panel**. This panel must have **CardLayout** selected as its layout manager.
- Cards are added to panel using

  void add(Component panelObj, Object name);
- methods defined by **CardLayout**:

  void first(Container deck) void

  last(Container deck) void

  next(Container deck)

  void previous(Container deck)

  void show(Container deck, String cardName)

GridBag Layout

- The Grid bag layout displays components subject to the constraints specified by GridBagConstraints.

- **GridLayout** lays out components in a two-dimensional grid.

- The constructors are
  GridLayout( )
  GridLayout(int numRows, int numColumns)
  GridLayout(int numRows, int numColumns, int horz, int vert)

Concepts of Applets

- *Applets* are small applications that are accessed on an Internet server, transported over the Internet, automatically installed, and run as part of a Web document.
- After an applet arrives on the client, it has limited access to resources, so that it can produce an arbitrary multimedia user interface and run complex computations without introducing the risk of viruses or breaching data integrity.
- applets – Java program that runs within a Java-enabled browser, invoked through a "applet" reference on a web page, dynamically downloaded to the client computer

```
import java.awt.*;
import java.applet.*;
public class SimpleApplet extends Applet {
public void paint(Graphics g) {
g.drawString("A Simple Applet", 20, 20);
}
}
```

There are two ways to run an applet:

1. Executing the applet within a Java-compatible Web browser, such as NetscapeNavigator.

2. Using an applet viewer, such as the standard JDK tool, **appletviewer**.

- An appletviewer executes your applet in a window. This is generally the fastest and easiest way to test an applet.

- To execute an applet in a Web browser, you need to write a short HTML text file that contains the appropriate APPLET tag.

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

Differences between applets and applications

- Java can be used to create two types of programs: applications and applets.

- An *application* is a program that runs on your computer, under the operating system of that Computer(i.e an application created by Java is more or less like one created using C or C++).

- When used to create applications, Java is not much different from any other computer language.

- An *applet* is an application designed to be transmitted over the Internet and executed by a Java-compatible Web browser.
-
- An applet is actually a tiny Java program, dynamically downloaded across the network, just like an image, sound file, or video clip.

- The important difference is that an applet is an *intelligent program*, not just an animation or media file(i.e an applet is a program that can react to user input and dynamically change—not just run the same animation or sound over and over.
- Applications require main method to execute.
- Applets do not require main method.
- Java's console input is quite limited
- Applets are graphical and window-based.

Life cycle of an applet

- Applets life cycle includes the following methods
  1. **init( )**
  2. **start( )**
  3. **paint( )**
  4. **stop( )**
  5. **destroy( )**
- When an applet begins, the AWT calls the following methods, in this sequence:

  init( )
  start( )
  paint( )
- When an applet is terminated, the following sequence of method calls takes place:

  stop( )
  destroy( )

- **init( ):** The **init( )** method is the first method to be called. This is where you should initialize variables. This method is called only once during the run time of your applet.
- **start( ):** The **start( )** method is called after **init( )**. It is also called to restart an applet after it has been stopped. Whereas **init( )** is called once—the first time an applet is loaded—**start( )** is called each time an applet's HTML document is displayed onscreen. So, if a user leaves a web page and comes back, the applet resumes execution at **start( )**.
- **paint( ):** The **paint( )** method is called each time applet's output must be redrawn. **paint( )** is also called when the applet begins execution. Whatever the cause, whenever the applet must redraw its output, **paint( )** is called. The **paint( )** method has one parameter of type **Graphics**. This parameter will contain the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required.
- **stop( ):** The **stop( )** method is called when a web browser leaves the HTML document containing the applet—when it goes to another page, for example. When **stop( )** is called, the applet is probably running. Applet uses **stop( )** to suspend threads that don't need to run when the applet is not visible. To restart **start( )** is called if the user returns to the page.
- **destroy( ):** The **destroy( )** method is called when the environment determines that your applet needs to be removed completely from memory. The **stop( )** method is always called before **destroy( )**.

Types of applets

- Applets are two types

  1.Simple applets

  2.JApplets

- Simple applets can be created by extending Applet class
- JApplets can be created by extending JApplet class of javax.swing.JApplet package

Creating applets

- Applets are created by extending the Applet class.

```
import java.awt.*;
 import java.applet.*;
 /*<applet code="AppletSkel" width=300 height=100></applet> */
 public class AppletSkel extends Applet {
 public void init() {
 // initialization
 }
 public void start() {
 // start or resume execution
 }
 public void stop() {
 // suspends execution
 }
 public void destroy() {
 // perform shutdown activities
 }
 public void paint(Graphics g) {
 // redisplay contents of window
 }
}
```

passing parameters to applets

- APPLET tag in HTML allows you to pass parameters to applet.
- To retrieve a parameter, use the **getParameter( )** method. It returns the value of the specified parameter in the form of a **String** object.

```
// Use Parameters
import java.awt.*;
import java.applet.*;
/*
<applet code="ParamDemo" width=300 height=80>
<param name=fontName value=Courier>
<param name=fontSize value=14>
<param name=leading value=2>
<param name=accountEnabled value=true>
</applet>
*/
```

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

Introduction to swings

- Swing is a set of classes that provides more powerful and flexible components than are possible with the AWT.
- In addition to the familiar components, such as buttons, check boxes, and labels, Swing supplies several exciting additions, including tabbed panes, scroll panes, trees, and tables.
- Even familiar components such as buttons have more capabilities in Swing.
- For example, a button may have both an image and a text string associated with it. Also, the image can be changed as the state of the button changes.
- Unlike AWT components, Swing components are not implemented by platform-specific code.
- Instead, they are written entirely in Java and, therefore, are platform-independent.
- The term *lightweight* is used to describe such elements.

- The Swing component are defined in **javax.swing**
  1. AbstractButton: Abstract superclass for Swing buttons.
  2. ButtonGroup: Encapsulates a mutually exclusive set of buttons.
  3. ImageIcon: Encapsulates an icon.
  4. JApplet: The Swing version of Applet.
  5. JButton: The Swing push button class.
  6. JCheckBox: The Swing check box class.
  7. JComboBox : Encapsulates a combo box (an combination of a drop-down list and text field).
  8. JLabel: The Swing version of a label.
  9. JRadioButton: The Swing version of a radio button.
  10. JScrollPane: Encapsulates a scrollable window.
  11. JTabbedPane: Encapsulates a tabbed window.
  12. JTable: Encapsulates a table-based control.
  13. JTextField: The Swing version of a text field.
  14. JTree: Encapsulates a tree-based control.

Limitations of AWT

- AWT supports limited number of GUI components.
- AWT components are heavy weight components.
- AWT components are developed by using platform specific code.
- AWT components behaves differently in different operating systems.
- AWT component is converted by the native   code of the operating system.

- Lowest Common Denominator
  - If not available natively on one Java platform, not available on any Java platform

- Simple Component Set

- Components Peer-Based
  - Platform controls component appearance
  - Inconsistencies in implementations
  - Interfacing to native platform error-prone

components

- Container
  - JComponent
    - AbstractButton
      - JButton
      - JMenuItem
        - JCheckBoxMenuItem
        - JMenu
        - JRadioButtonMenuItem
      - JToggleButton
        - JCheckBox
        - JRadioButton

Components (contd…)

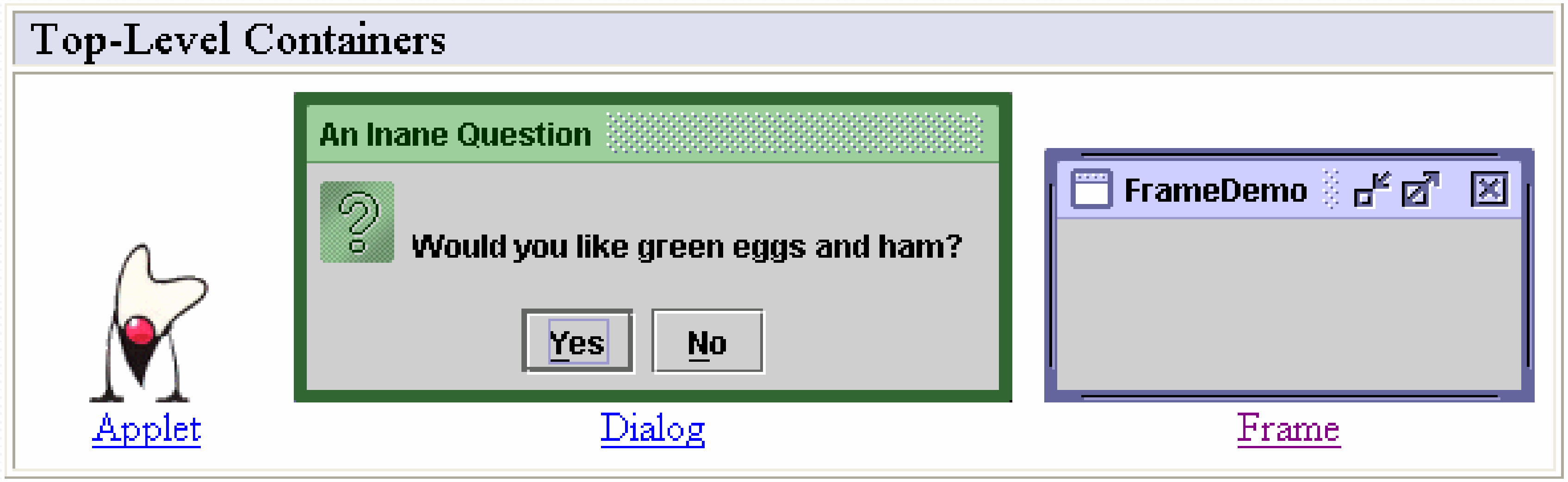
- JComponent
    - JComboBox
    - JLabel
    - JList
    - JMenuBar
    - JPanel
    - JPopupMenu
    - JScrollBar
    - JScrollPane

Components (contd…)

- JComponent
    - JTextComponent
        - JTextArea
        - JTextField
            - JPasswordField
        - JTextPane
            - JHTMLPane

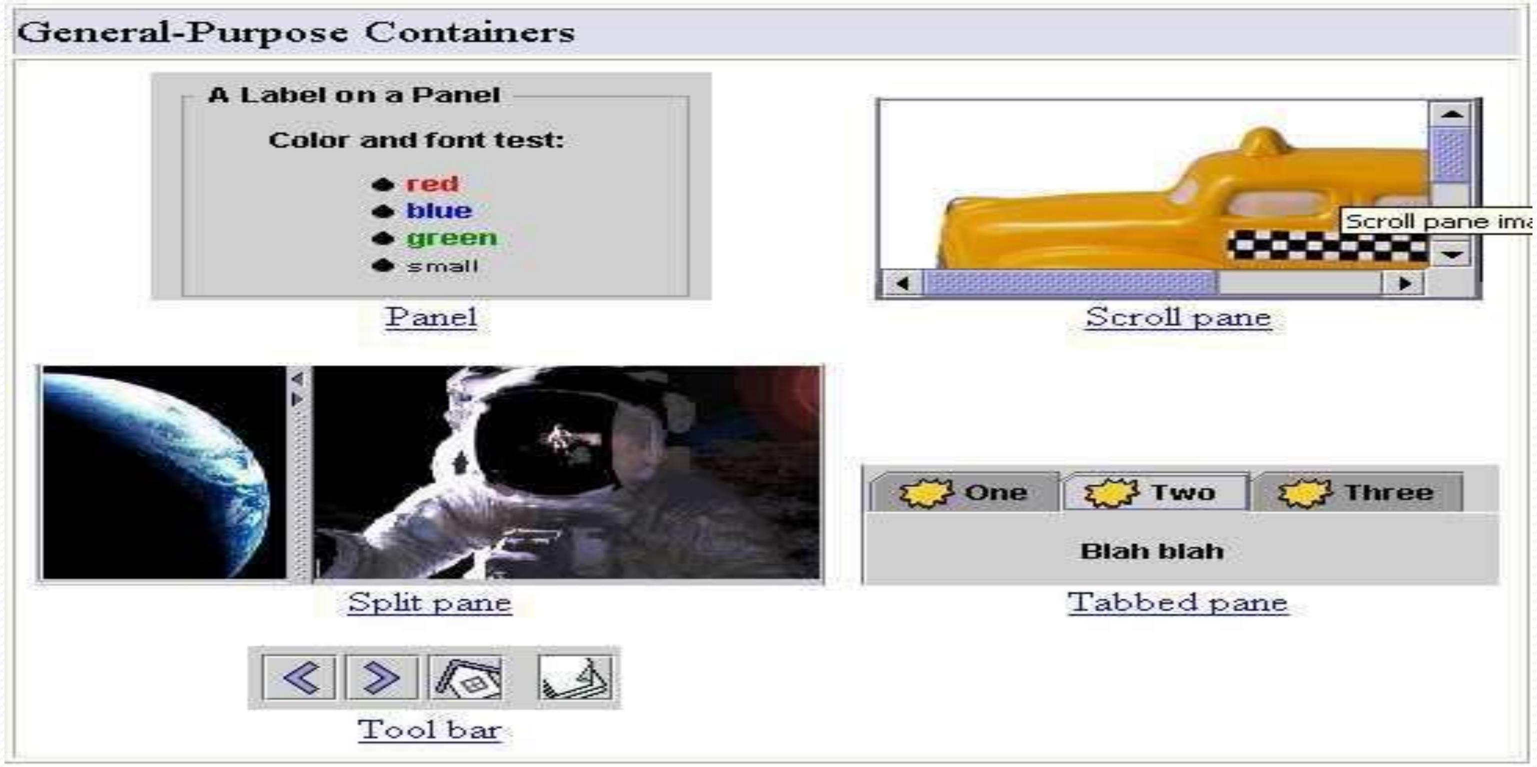
Containers

- Top-Level Containers
- The components at the top of any Swing containment hierarchy



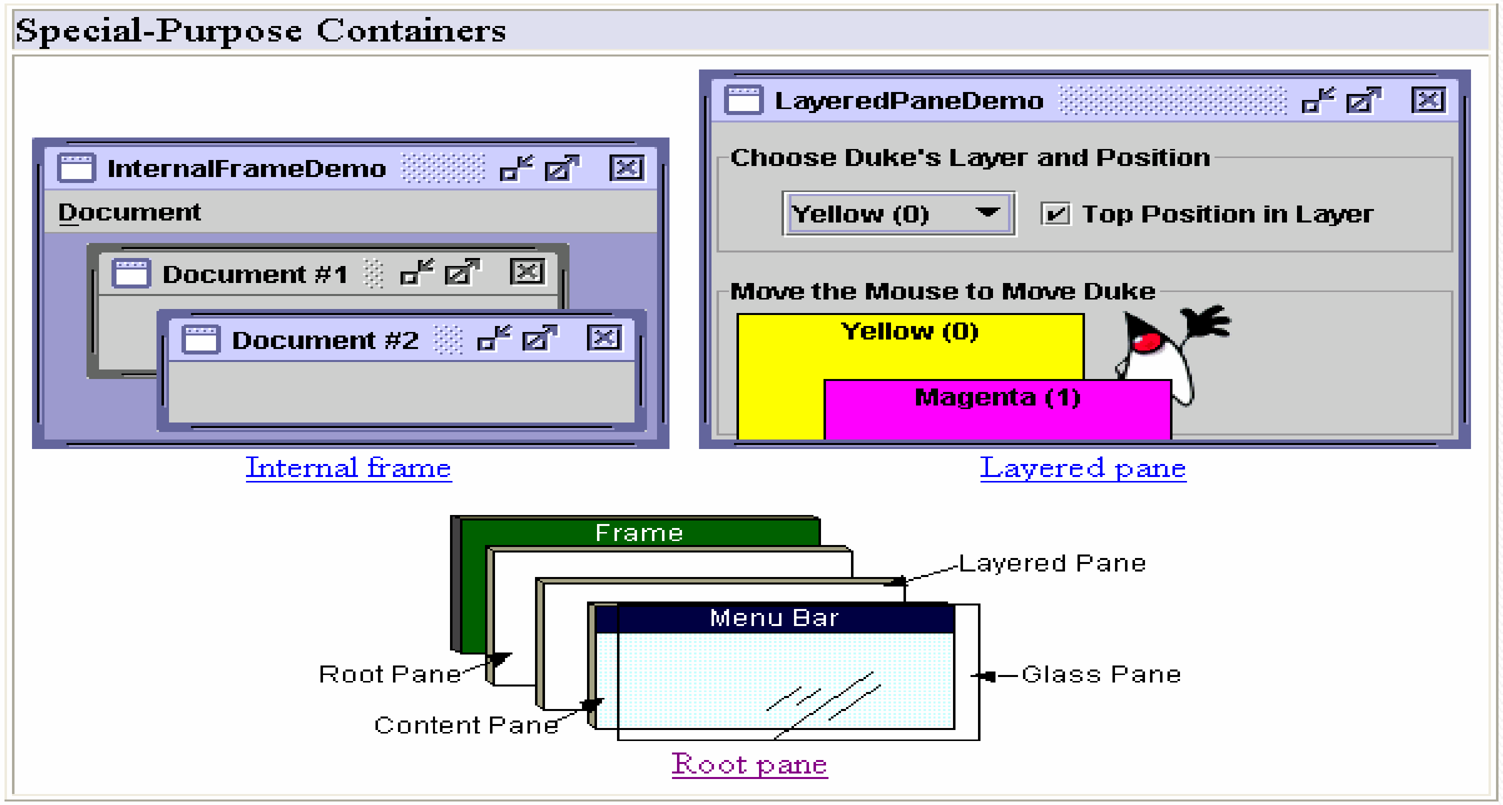
General Purpose Containers

- Intermediate containers that can be used under many different circumstances.



Special Purpose Container

- Intermediate containers that play specific roles in the UI.

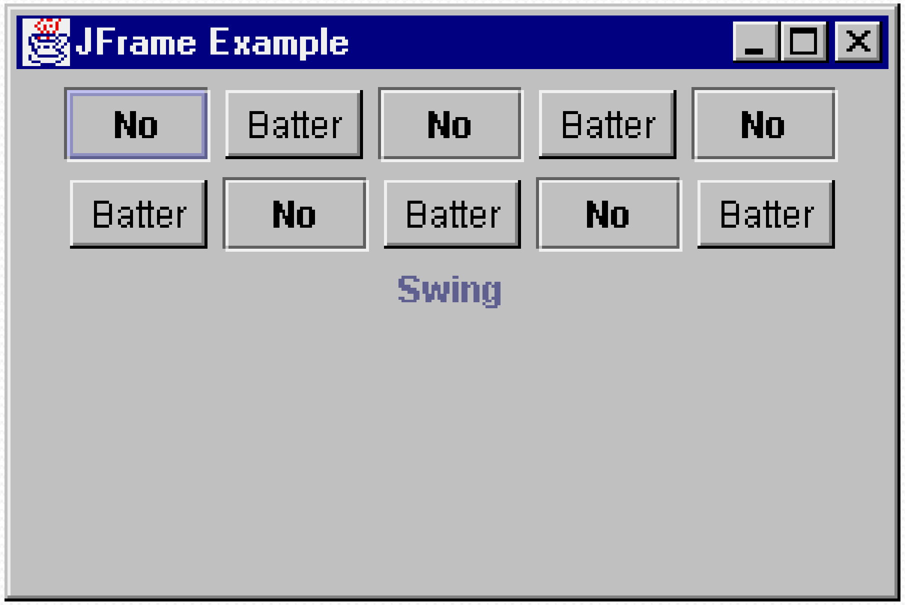


Exploring swing- JApplet

- If using Swing components in an applet, subclass *JApplet*, not *Applet*
  - *JApplet* is a subclass of *Applet*
  - Sets up special internal component event handling, among other things
  - Can have a *JMenuBar*
  - Default *LayoutManager* is *BorderLayout*

JFrame

```
public class FrameTest {
 public static void main (String args[]) {
  JFrame f = new JFrame ("JFrame Example");
  Container c = f.getContentPane();
  c.setLayout (new FlowLayout());
  for (int i = 0; i < 5; i++) {
   c.add (new JButton ("No"));
   c.add (new Button ("Batter"));
  }
  c.add (new JLabel ("Swing"));
  f.setSize (300, 200);
  f.show();
 }
}
```



JComponent

- JComponent supports the following components.
- JComponent
    - JComboBox
    - JLabel
    - JList
    - JMenuBar
    - JPanel
    - JPopupMenu
    - JScrollBar
    - JScrollPane
    - JTextComponent
        - JTextArea
        - JTextField
            - JPasswordField
        - JTextPane
            - JHTMLPane

Icons and Labels

- In Swing, icons are encapsulated by the **ImageIcon** class, which paints an icon from an image.
- constructors are:

  ImageIcon(String filename)

  ImageIcon(URL url)
- The **ImageIcon** class implements the **Icon** interface that declares the methods
  1. int getIconHeight( )
  2. int getIconWidth( )
  3. void paintIcon(Component *comp*,Graphics *g*,int *x*, int *y*)

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

  JLabel(Icon i)

  Label(String s)

  JLabel(String s, Icon i, int align)
- Here, *s* and *i* are the text and icon used for the label. The *align* argument is either **LEFT**, **RIGHT**, or **CENTER**. These constants are defined in the **SwingConstants** interface,
- Methods are:
  1. Icon getIcon( )
  2. String getText( )
  3. void setIcon(Icon i)
  4. void setText(String s)
- Here, *i* and *s* are the icon and text, respectively.

Text fields

- The Swing text field is encapsulated by the **JTextComponent** class, which extends**JComponent**.
- It provides functionality that is common to Swing text components.
- One of its subclasses is **JTextField**, which allows you to edit one line of text.
- Constructors are:

  JTextField( )

  JTextField(int cols)

      JTextField(String s, int cols)

      JTextField(String s)
- Here, *s* is the string to be presented, and *cols* is the number of columns in the text field.

Buttons

- Swing buttons provide features that are not found in the **Button** class defined by the AWT.
- Swing buttons are subclasses of the **AbstractButton** class, which extends **JComponent**.
- **AbstractButton** contains many methods that allow you to control the behavior of buttons, check boxes, and radio buttons.
- Methods are:
    1. void setDisabledIcon(Icon di)
    2. void setPressedIcon(Icon pi)
    3. void setSelectedIcon(Icon si)
    4. void setRolloverIcon(Icon ri)
- Here, *di*, *pi*, *si*, and *ri* are the icons to be used for these different conditions.
- The text associated with a button can be read and written via the following methods:
    1. String getText( )
    2. void setText(String s)
- Here, *s* is the text to be associated with the button.

JButton

- The **JButton** class provides the functionality of a push button.
- **JButton** allows an icon, a string, or both to be associated with the push button.
- Some of its constructors are :

  JButton(Icon i) JButton(String s)

  JButton(String s, Icon i)
- Here, *s* and *i* are the string and icon used for the button.

Check boxes

- The **JCheckBox** class, which provides the functionality of a check box, is a concrete implementation of **AbstractButton.**
- Some of its constructors are shown here:

```
JCheckBox(Icon   i)
JCheckBox(Icon i, boolean state)
JCheckBox(String s)
JCheckBox(String s, boolean state)
JCheckBox(String s, Icon i)
JCheckBox(String s, Icon i, boolean state)
```

- Here, *i* is the icon for the button. The text is specified by *s*. If *state* is **true**, the check box is initially selected. Otherwise, it is not.
- The state of the check box can be changed via the following method:

```
void setSelected(boolean state)
```

- Here, *state* is **true** if the check box should be checked.

Combo boxes

- Swing provides a *combo box* (a combination of a text field and a drop-down list) through the **JComboBox** class, which extends **JComponent**.
- A combo box normally displays one entry. However, it can also display a drop-down list that allows a user to select a different entry. You can also type your selection into the text field.
- Two of **JComboBox**'s constructors are :

```
JComboBox( )
JComboBox(Vector v)
```

- Here, *v* is a vector that initializes the combo box.
- Items are added to the list of choices via the **addItem( )** method, whose signature is:

```
void addItem(Object obj)
```

- Here, *obj* is the object to be added to the combo box.

Radio Buttons

- Radio buttons are supported by the **JRadioButton** class, which is a concrete implementation of **AbstractButton**.
- Some of its constructors are :
  JRadioButton(Icon i) JRadioButton(Icon i,
  boolean state) JRadioButton(String s)
  JRadioButton(String s, boolean state)
  JRadioButton(String s, Icon i)
  JRadioButton(String s, Icon i, boolean state)
- Here, *i* is the icon for the button. The text is specified by *s*. If *state* is **true**, the button is initially selected. Otherwise, it is not.
- Elements are then added to the button group via the following method:
  void add(AbstractButton ab)
- Here, *ab* is a reference to the button to be added to the group.

Tabbed Panes

- A *tabbed pane* is a component that appears as a group of folders in a file cabinet.
- Each folder has a title. When a user selects a folder, its contents become visible. Only one of the folders may be selected at a time.
- Tabbed panes are commonly used for setting configuration options.
- Tabbed panes are encapsulated by the **JTabbedPane** class, which extends **JComponent**. We will use its default constructor. Tabs are defined via the following method:

  ```
  void addTab(String str, Component comp)
  ```
- Here, *str* is the title for the tab, and *comp* is the component that should be added to the tab. Typically, a **JPanel** or a subclass of it is added.
- The general procedure to use a tabbed pane in an applet is outlined here:

  1. Create a **JTabbedPane** object.
  2. Call **addTab( )** to add a tab to the pane. (The arguments to this method define the
     title of the tab and the component it contains.)
  3. Repeat step 2 for each tab.
  4. Add the tabbed pane to the content pane of the applet.

Scroll Panes

- A *scroll pane* is a component that presents a rectangular area in which a component may be viewed. Horizontal and/or vertical scroll bars may be provided if necessary.
- Scroll panes are implemented in Swing by the **JScrollPane** class, which extends **JComponent**. Some of its constructors are :

  JScrollPane(Component comp)

  JScrollPane(int vsb, int hsb)

  JScrollPane(Component comp, int vsb, int hsb)
- Here, *comp* is the component to be added to the scroll pane. *vsb* and *hsb* are **int** constants that define when vertical and horizontal scroll bars for this scroll pane are shown.
- These constants are defined by the **ScrollPaneConstants** interface.
  1. HORIZONTAL_SCROLLBAR_ALWAYS
  2. HORIZONTAL_SCROLLBAR_AS_NEEDED
  3. VERTICAL_SCROLLBAR_ALWAYS
  4. VERTICAL_SCROLLBAR_AS_NEEDED
- Here are the steps to follow to use a scroll pane in an applet:

  1. Create a **JComponent** object.
  2. Create a **JScrollPane** object. (The arguments to the constructor specify the component and the policies for vertical and horizontal scroll bars.)
  3. Add the scroll pane to the content pane of the applet.

Trees

- Data Model - TreeModel
  - default: DefaultTreeModel
  - getChild, getChildCount, getIndexOfChild, getRoot, isLeaf

- Selection Model – TreeSelectionModel

- View - TreeCellRenderer
  - getTreeCellRendererComponent

- Node - DefaultMutableTreeNode

Tables

- A *table* is a component that displays rows and columns of data. You can drag the cursor on column boundaries to resize columns. You can also drag a column to a new position.
- Tables are implemented by the **JTable** class, which extends **JComponent**.
- One of its constructors is :

  JTable(Object data[ ][ ], Object colHeads[ ])
- Here, *data* is a two-dimensional array of the information to be presented, and *colHeads* is a one-dimensional array with the column headings.
- Here are the steps for using a table in an applet:

  1. Create a **JTable** object.
  2. Create a **JScrollPane** object. (The arguments to the constructor specify the table and
  the policies for vertical and horizontal scroll bars.)
  3. Add the table to the scroll pane.
  4. Add the scroll pane to the content pane of the applet.

UNIT-2

- Inheritance –Definition
- Single Inheritance
- Benefits of inheritance
- Member access rules
- super classes
- Polymorphism
- Method overriding
- Using final with inheritance
- abstract classes and
- Base class object.

Definition

- Inheritance is the process of acquiring the properties by the **sub class** (or derived class or child class) from the **super class** (or base class or parent class).
- When a **child** class(newly defined abstraction) inherits(extends) its **parent** class (being inherited abstraction), all the properties and methods of parent class becomes the member of child class.
- In addition, child class can add new data fields(properties) and behaviors(methods), and
- It can override methods that are inherited from its parent class.

Inheritance Basics

The key word **extends** is used to define inheritance in Java.

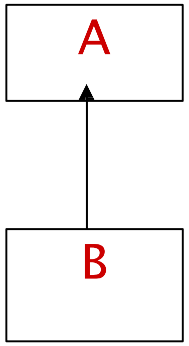
Syntax:-

```
class subclass-name extends superclass-name {
        // body of the class
}
```

Single Inheritance

-Derivation of a class from only one base class is called single inheritance.

```
//base class:
class A{
            //members of A
}
//Derived class syntax:
class  B extends A{
            //members of B
}
```



```
// Create a superclass.
class A {
	int i, j;
	void showij() {
	System.out.println("i and j: " + i + " " + j);
	}
}
// Create a subclass by extending class A.
class B extends A {
	int k;
	void showk() {
		System.out.println("k: " + k);
	}
	void sum() {
		System.out.println("i+j+k: " + (i+j+k));
	}
}
class SimpleInheritance {
	public static void main(String args[]) {
		A superOb = new A();
		B subOb = new B();
		// The superclass may be used by itself.
		superOb.i = 10;
		superOb.j = 20;
		System.out.println("Contents of superOb:");
		superOb.showij();
/* The subclass has access to all public members
of its superclass. */
subOb.i = 7;
subOb.j = 8;
subOb.k = 9;
System.out.println("Contents of   subOb: ");

subOb.showij();
subOb.showk();
System.out.println();

System.out.println("Sum of i, j and k in subOb:");

subOb.sum();
}
}
```

```
Contents of superOb:
i and j: 10 20

Contents of subOb:
i and j: 7 8
k: 9

Sum of i, j and k in subOb:
i+j+k: 24
```

The Benefits of Inheritance

- **Software Reusability** (among projects)
  - Code (class/package) can be reused among the projects.
  - Ex., code to insert a new element into a table can be written once and reused.

- **Code Sharing** (within a project)
  - It occurs when two or more classes inherit from a single parent class.
  - This code needs to be written only once and will contribute only once to the size of the resulting program.

- **Increased Reliability** (resulting from reuse and sharing of code)
  - When the same components are used in two or more applications, the bugs can be discovered more quickly.

- **Information Hiding**
  - The programmer who reuses a software component needs only to understand the nature of the component and its interface.
  - It is not necessary for the programmer to have detailed information such as the techniques used to implement the component.
- **Rapid Prototyping** (quickly assemble from pre-existing components)
  - Software systems can be generated more quickly and easily by assembling preexisting components.
  - This type of development is called Rapid Prototyping.
- **Consistency of Interface**(among related objects)
  - When two or more classes inherit from same superclass, the behavior they inherit will be the same.
  - Thus , it is easier to guarantee that interfaces to similar objects are similar.

- **Software Components**
  - Inheritance enables programmers to construct reusable components.

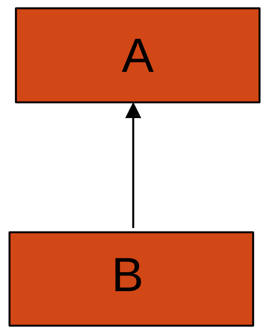
- **Polymorphism and Frameworks** (high-level reusable components)
  - Normally, code reuse decreases as one moves up the levels of abstraction.

  - Lowest-level routines may be used in several different projects, but higher-level routines are tied to a particular application.

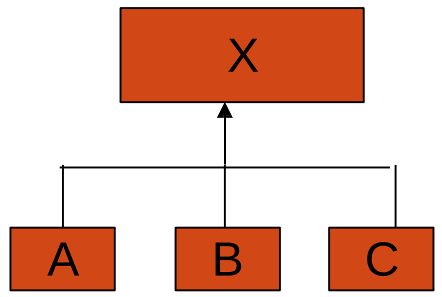
  - Polymorphism in programming languages permits the programmer to generate high-level reusable components that can be tailored to fit different applications by changes in their low-level parts.

Types of Inheritance

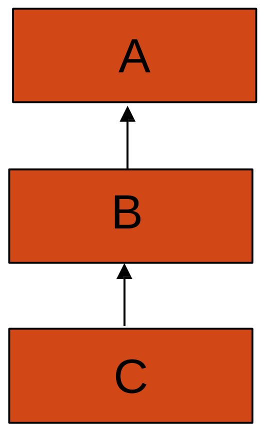
Single Inheritance



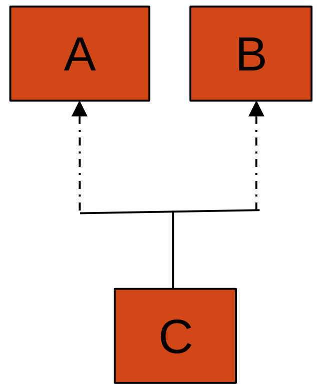
Hierarchical Inheritance



Multilevel Inheritance



Multiple Inheritance



```
//Single Inheritance

class A{
}
class B extends A{
}
```

```
//Multilevel Inheritance

class A{
}
class B extends A{
}
class C extends B{
}
```

```
//Hierarchical Inheritance

class A{
}
class B extends A{
}
class C extends A{
}
```

```
//Multiple Inheritance

interface one{
}
interface two{
}
class A implements one, two{
}
```

- Multiple Inheritance can be implemented by implementing multiple interfaces not by extending multiple classes.

Example :

```
class B extends A implements C , D{

}                                    OK
```

```
class C extends A extends B{                class C extends A ,B{

}                                                   }

                                WRONG
```

A Superclass Variable Can Reference a Subclass Object

- When a reference to a subclass object is assigned to a superclass variable, you will have access only to those parts of the object defined by the superclass.
  Ex:

```
class A{
        int i=10;
}
class B extends A{
        int j=30;
}
class Test{
  public static void main(String args[]){
        A a=new A();
        B b=new B();
        a=b;
        System.out.println(a.i);
        //System.out.println(a.j);
  }
}
```

Super Keyword

- Subclass refers to its immediate superclass by using **super** keyword.

- **super** has two general forms.
  - First it calls the superclass constructor.
  - Second is used to access a member of the superclass that has been hidden by a member of a subclass.

- **Using super to call superclass constructors**

  super (parameter-list);

  - parameter-list specifies any parameters needed by the constructor in the superclass.
  - **super( )** must always be the first statement executed inside a subclass constructor.

```
class Box {
        Box() {
           System.out.println("Box() in super class");
        }
        Box(int a){
          System.out.println("Box(int a) in super class");
        }
}
class BoxWeight extends Box {
        BoxWeight(){
        System.out.println("BoxWeight() in sub class");
        }
}
class DemoBoxWeight{
        public static void main(String args[]) {
            BoxWeight mybox1 = new BoxWeight();
        }
}
```

Output:
Box() in super class
BoxWeight() in sub class

//Using super to call superclass constructors

```
class Box {
Box() {
        System.out.println("Box() in super class");
        }
        Box(int a){
             System.out.println("Box(int a) in super class");
        }
}
class BoxWeight extends Box {
        BoxWeight(){
            super(10);
            System.out.println("BoxWeight() in sub class");
        }
}
class DemoBoxWeight{
        public static void main(String args[]) {
                BoxWeight mybox1 = new BoxWeight();
        }
}
```

Output:
Box(int a) in super class
BoxWeight() in sub class

- The second form of **super** acts somewhat like **this**, except that it always refers to the superclass of the subclass in which it is used.

Syntax: super.member

- Here, member can be either a method or an instance variable.
- This second form of **super** is most applicable to situations in which member names of a subclass hide members by the same name in the superclass.

```
// Using super to overcome name hiding.
class A {
    int i;
}
// Create a subclass by extending class A.
class B extends A {
    int i; // this i hides the i in A
    B(int a, int b) {
        super.i = a; // i in A
        i = b; // i in B
    }
    void show() {
        System.out.println("i in superclass: " + super.i);
        System.out.println("i in subclass: " + i);
    }
}
class UseSuper {
     public static void main(String args[]) {
            B subOb = new B(1, 2);
            subOb.show();
    }
}
```

This program displays the following:

```
i in superclass: 1
i in subclass: 2
```

When Constructors Are Called

- In a class hierarchy, constructors are called in order of derivation, from superclass to subclass.
- super(…) must be the first statement executed in a subclass' constructor.
- If super(…) is not used, the default constructor of each superclass will be executed.
    - Implicitly default form of super (super()) will be invoked in each subclass to call default constructor of superclass.

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

```
Output:
Inside  A’s  constructor
Inside  B’s constructor
Inside C’s constructor
```

Member access rules

A subclass includes all of the members (default, public, protected) of its superclass except private members.

```
class A{
        private int v=10;
        int d=20;
        public int b=30;
        protected int p=40;
}
class B extends A{
        void disp(){
                //System.out.println(“v value : "+v);
                System.out.println(“d value : "+d);
                System.out.println(“b value : "+b);
                System.out.println("p value : "+p);
        }
}
class C extends B{
        void show(){
                System.out.println("p value : "+p);
        }
}
```

```
class Protected{
        public static void main(String args[]){
                B b=new B();
                b.disp();
                C c=new C();
                c.show();
        }
}
```

```
Output:
d value : 20
b value : 30
p value : 40
p value : 40
```

Polymorphism

- Assigning multiple meanings to the same method name

- Implemented using late binding or dynamic binding (run-time binding):

- It means, method to be executed is determined at execution time, not at compile time.
- Polymorphism can be implemented in two ways
    - Overloading
    - Overriding
- When a method in a subclass has the same name, signature and return type as a method in its superclass, then the method in the subclass is said to be overridden the method in the superclass.

- By method overriding, subclass can implement its own behavior.

//Overriding example

```
class A{
    int i,j;
    A(int a,int b){
          i=a;
          i=b;
    }
    void show(){
      System.out.println("i and j :"+i+" "+j);
    }
}
```

```
class B extends A{
    int k;
    B(int a, int b, int c){
          super(a,b);
          k=c;
    }
    void show(){
          System.out.println("k=:"+k);
    }
}
class Override{
          public static void main(String args[]){
                    B subob=new B(3,4,5);
                    subob.show();
          }
}
```

Output:

K: 5

Dynamic Method Dispatch

- Dynamic method dispatch is the mechanism by which a call to an **overridden** method is resolved at run time, rather than compile time.

- When an **overridden** method is called through a superclass reference, the method to execute will be based upon the type of the object being referred to at the time the call occurs. Not the type of the reference variable.

//Dynamic Method Dispatch

```
class A{
      void callme(){
      System.out.println("Inside A's callme method");
      }
}
class B extends A{
      void callme(){
      System.out.println("Inside B's callme method");
      }
}
class C extends A{
      void callme(){
      System.out.println("Inside C's callme method");
      }
}
```

```
 class Dispatch{
public static void main(String args[]){
                A a=new A();
                B b=new B();
                C c=new C();

                A r;

                 r=a;
                 r.callme();

                 r=b;
                 r.callme();

                 r=c;
                 r.callme();
}
}
```

Output:

```
Inside A's callme method
Inside B's callme method
Inside C's callme method
```

```
// Using run-time polymorphism.
class Figure {
      double dim1;
      double dim2;
      Figure(double a, double b) {
            dim1 = a;
            dim2 = b;
      }
      double area() {
   System.out.println("Area for Figure is undefined.");
        return 0;
      }
}
class Rectangle extends Figure {
      Rectangle(double a, double b) {
            super(a,b);
      }
      // override area for rectangle
      double area() {
        System.out.println("Inside Area for Rectangle.");
            return dim1 * dim2;
      }
}
class Triangle extends Figure {
      Triangle(double a, double b) {
                  super(a, b);
      }
      // override area for right triangle
      double area() {
            System.out.println("Inside Area for Triangle.");
                  return dim1 * dim2 / 2;
      }
}

class FindAreas {
      public static void main(String args[]) {
            Figure f = new Figure(10, 10);
            Rectangle r = new Rectangle(9, 5);
            Triangle t = new Triangle(10, 8);
            Figure figref;
            figref = r;
            System.out.println("Area is " + figref.area());
            figref = t;
            System.out.println("Area is " + figref.area());
            figref = f;
            System.out.println("Area is " + figref.area());
      }
}
```

```
Inside Area for Rectangle.
Area is 45
Inside Area for Triangle.
Area is 40
Area for Figure is undefined.
Area is 0
```

Abstract Classes

- A method that has been declared but not defined is an abstract method.
- Any class that contains one or more abstract methods must also be declared abstract.
- You must declare the abstract method with the keyword abstract:

```
abstract type name (parameter-list);
```

- You must declare the class with the keyword abstract:

```
abstract class MyClass{
        ......
}
```

- An abstract class is incomplete, It has “missing” method bodies.
- You cannot instantiate (create a new instance of) an abstract class but you can create reference to an abstract class.
- Also, you cannot declare abstract constructors, or abstract static methods.

- You can declare a class to be abstract even if it does not contain any abstract methods. This prevents the class from being instantiated.

- An abstract class can also have concrete methods.

- You can extend (subclass) an abstract class.
  - If the subclass defines all the inherited abstract methods, it is “complete” and can be instantiated.
  - If the subclass does not define all the inherited abstract methods, it is also an abstract class.

```
// A Simple demonstration of abstract.
abstract class A {
    abstract void callme();
    // concrete methods are still allowed in abstract classes
    void callmetoo() {
            System.out.println("This is a concrete method.");
    }
}
class B extends A {
    void callme() {
    System.out.println("B's implementation of callme.");
    }
}
class AbstractDemo {
    public static void main(String args[]) {
            B b = new B();
            b.callme();
            b.callmetoo();
    }
}
```

Output:

```
B's implementation of callme.
This is a concrete method.
```

Using final with Inheritance

The keyword **final** has three uses:
- To create a constant variable
- To prevent overriding
- To prevent inheritance

To create a constant variable:

– A variable can be declared as **final**. Doing so prevents its contents from being modified. This means that you must initialize a **final** variable when it is declared.

```
class FinalDemo{
        public static void main(String sree[]){
                final int i=20;
                System  em.out.println(i);
                //i=i+1;   can't assign a value to final variable i
        //System.out.println(i);   cannot assign a value to final variable i
        }
}
```

To prevent overriding

To disallow a method from being overridden, specify **final** as a modifier at the start of its declaration. Methods declared as **final** cannot be overridden.

```
class A {
        final void meth() {
                System.out.println("This is a final method.");
        }
}

class B extends A {
        void meth() {                   // ERROR! Can't override.
                        System.out.println("Illegal!");
        }
}
```

To prevent inheritance

→To prevent a class from being inherited precede the class declaration with **final**.

→Declaring a class as **final** implicitly declares all of its methods as **final**, too.

→It is illegal to declare a class as both **abstract** and **final** since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations.

```
final class A {
        // ...
}
// The following class is illegal.
class B extends A {          // ERROR! Can't subclass A
        // ...
}
```

- Normally, Java resolves calls to methods dynamically, at run time. This is called late binding.
- However, since **final** methods cannot be overridden, a call to one can be resolved at compile time. This is called early binding.

The Object Class

- **Object** is a special class, defined by Java.
- **Object** is a superclass of all other classes.
- This means that a reference variable of type **Object** can refer to an object of any other class.
- **Object** defines the following methods:

Method	Purpose
Object clone( )	Creates a new object that is the same as the object being cloned.
boolean equals(Object *object*)	Determines whether one object is equal to another.
void finalize( )	Called before an unused object is recycled.
Class getClass( )	Obtains the class of an object at run time.
int hashCode( )	Returns the hash code associated with the invoking object.
void notify( )	Resumes execution of a thread waiting on the invoking object.
void notifyAll( )	Resumes execution of all threads waiting on the invoking object.
String toString( )	Returns a string that describes the object.
void wait( ) void wait(long *milliseconds*) void wait(long *milliseconds*, int *nanoseconds*)	Waits on another thread of execution.

```
import java.io.*;
import java.util.Scanner;
class CharDemo{
    static char c[]=new char[10];
    public static void main(String sree[])throws Exception{
     //BufferedReader d=new BufferedReader(new InputStreamReader(System.in));
     System.out.println("Enter Characters:");
     for(int i=0;i<10;i++){
          //c[i]=(char)d.read();
          c[i]=(char)System.in.read();
     }
     System.out.println("Entered Characters:");
     for(int i=0;i<10;i++){
          System.out.println(c[i]);
     }
    }
}
```

- Defining an interface
- Implementing an interface
- Differences between classes and interfaces
- Implements and extends keywords
- An application using an interfaces and uses of interfaces

- Defining Package
- Creating and Accessing a Package
- Types of packages
- Understanding CLASSPATH
- importing packages

Interface

- It defines a standard and public way of specifying the behavior of classes.
- It defines a contract of a class.
- Using interface, you can specify what a class must do, but not how it does it.
- All methods of an interface are abstract methods. That is it defines the signatures of a set of methods, without the body.
- A concrete class must implement the interface (all the abstract methods of the Interface).
- Interface allows classes, regardless of their locations in the class hierarchy, to implement common behaviors.

- Once an interface is defined, any number of classes can implement an interface.

- Also, one class can implement any number of interfaces.

- Using the keyword interface, you can fully abstract a class' interface from its implementation.

- Using the keyword implements, you can implement any number of interfaces.

- The methods in interface are abstract by default.

- The variables in interface are final by default.

Defining an Interface

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

```
access interface interfacename {
        return-type method-name1(parameter-list);
        return-type method-name2(parameter-list);
        type final-varname1 = value;
        type final-varname2 = value;
        // ...
        return-type method-nameN(parameter-list);
        type final-varnameN = value;
    }
```

Example:

```
    interface Callback {
            void callback(int param);
    }
```

Here, access is either **public or not used.**

When **no access specifier** is included, then default access results, and the interface is only available to other members of the package in which it is declared. When it is declared as **public**, the interface can be used by any other code.

‘name’ is the name of the interface, and can be any valid identifier.

Notice that the methods which are declared have no bodies. They are, essentially, abstract methods.

Variables can be declared inside of interface declarations. They are implicitly **final** and **static**, meaning they cannot be changed by the implementing class. They **must** also be initialized with a **constant value**.

All methods and variables are implicitly **public if** the interface, itself, is declared as **public.**

Implementing Interfaces

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

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

The general form of **a class that includes the implements clause** looks like this:

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

- Here, access is either public or not used.

- If a class implements more than one Interface, the interfaces are separated with a comma.

- If a class implements two interfaces that declare the same method, then the same method will be **used by clients of** either interface.

- The methods that implement an interface **must be declared** public.

- Also, the **type signature of the implementing method** must match **exactly the type signature specified in the interface definition.**

Here is a small example class that implements the **Callback interface.**

```
class Client implements Callback {
            // Implement Callback's interface
            public void callback(int p) {
                  System.out.println("callback called with " + p);
            }
}
```

Notice that **callback( ) is declared using the public access specifier.**

When you implement an interface method, it must be declared as **public.**

It is both permissible and common for classes that implement interfaces to define **additional members of their own.**

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

//Example for a class which contain both interface and non interface methods

```
class Client implements Callback {
        // Implement Callback's interface
        public void callback(int p) {
            System.out.println("callback called with " + p);
        }

        void nonIfaceMeth() {
            System.out.println(“Non Interface Method…."); 
        }
}
```

Accessing Implementations Through Interface References

- You can declare variables as object references that use an interface rather than a class type.
- Any instance of any class that implements the declared interface can be referred to by such a variable.
- When you call a method through one of these references, the correct version will be called based on the actual instance of the interface being referred to.
- This is one of the key features of interfaces.
- The calling code can dispatch through an interface without having to know anything about the “callee.”

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

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

     Output:
                         callback called with 42
```

- Notice that variable **c is declared to be of the interface type Callback, yet it was** assigned an instance of **Client.**

- **Although c can be used to access the callback( )** method, it cannot access any other members of the **Client class.**

- **An interface reference** variable only has knowledge of the methods declared by its **interface declaration.**

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

- While the preceding example shows, mechanically, how an interface reference variable can access an implementation object, it does not demonstrate the polymorphic power of such a reference.

```
// Another implementation of Callback.
    class AnotherClient implements Callback {
          // Implement Callback's interface
          public void callback(int p) {
                    System.out.println("Another version of callback");
                    System.out.println("p squared is " + (p*p));
          }
    }
```

```
class TestIface2 {
          public static void main(String args[]) {
               Callback c = new Client();
               AnotherClient ob = new AnotherClient();
               c.callback(42);
               c = ob; // c now refers to AnotherClient object
               c.callback(42);
          }
}
```

Output:

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

Partial Implementations

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

```
abstract class Incomplete implements Callback {
        int a, b;
        void show() {
                System.out.println(a + " " + b);
        }
        // ...
}
```

- ➢ If a class includes an interface but does not fully implement the methods defined by that interface, then that class must be declared as **abstract.**
- ➢ Here, the class **Incomplete does not implement callback( )** and must be declared as **abstract**.
- ➢ Any class that inherits **Incomplete must** implement **callback( ) or** be declared **abstract itself.**

Variables in Interfaces

You can define variables in an interface but implicitly they are final variables.

That is you can't modify them.

FinalDemo.java

```
interface FinalDemo{
	int i=100;
	void show();
}
```

FinalTest.java

```
class FinalImpl implements FinalDemo{
	public void show(){
		System.out.println("FinalTest :Show()");
	}
}
class FinalTest{
	public static void main(String sree[]){
		FinalImpl fi=new FinalImpl();
		fi.show();
		//fi.i=200; can't assign a value to variable i
		System.out.println("FinalDemo Varaible i :"+fi.i);

	}
}
```

Output:

```
		FinalTest :Show()
		FinalDemo Varaible i :100
```

Interfaces Can Be Extended

One interface can inherit another by use of the keyword **extends.**

The syntax is the same as for inheriting classes.

When a **class** implements an **interface** that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain.

```
// One interface can extend another.
interface A {
      void meth1();
      void meth2();
}
// B now includes meth1() and meth2() -- it adds meth3().
interface B extends A {
      void meth3();
}
// This class must implement all of A and B
class MyClass implements B {
      public void meth1() {
      System.out.println("Implement meth1().");
}
      public void meth2() {
        System.out.println("Implement meth2().");
      }
      public void meth3() {
        System.out.println("Implement meth3().");
      }
}
class IFExtend {
      public static void main(String arg[]) {
        MyClass ob = new MyClass();
        ob.meth1();
        ob.meth2();
        ob.meth3();
      }
}
```

Output:

```
        Implement meth1().
        Implement meth2().
        Implement meth3().
```

```
interface Callback {
    void callback(int param);
}
```

```
class Client implements Callback {
        // Implement Callback's interface
        public void callback(int p) {
                System.out.println("callback called with " + p);
        }
        void nonIfaceMeth() {
                System.out.println("NonInterface
Method….");
        }
}
```

```
// Another implementation of Callback.
class AnotherClient implements Callback {
        // Implement Callback's interface
        public void callback(int p) {
                System.out.println("Another version of callback");
                System.out.println("p squared is " + (p*p));
        }
}
```

```
class TestIface2 {
        public static void main(String args[]) {
        Callback c = new Client();
        AnotherClient ob = new AnotherClient();
        c.callback(42);
        c = ob; // c now refers to AnotherClient object
        c.callback(42);
}       }
```

Output:

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

Class Vs Interface

- The methods of an Interface are all abstract methods. They cannot have bodies.
- An interface can only define constants.
- You cannot create an instance from an interface.
- An interface can only be **implemented** by classes or extended by other interfaces.
- Interfaces have no direct inherited relationship with any particular class, they are defined **independently.**
- Interfaces themselves have **inheritance** relationship among themselves.
- A class can implement **more than one** interface. By contrast, a class can only inherit a single superclass (abstract or otherwise).

Abstract Class Vs Interface

- An abstract class is written when there are some common features shared by all the objects.
- An interface is written when all the features are implement differently in different objects.

- When an abstract class is written, it is the duty of the programmer to provide sub classes to it.
- An interface is written when the programmer wants to leave the implementation to the third party vendors.

- An abstract class contains some abstract methods and also some concrete methods.
- An interface contains only abstract methods.

- An abstract class can contain instance variables also.
- An interface can not contain instance variables. It contains only constants.

- All the abstract methods of the abstract class should be implemented in its sub classes.
- All the (abstract) methods of the interface should be implemented in its implementation classes.

- Abstract class is declared by using the keyword abstract.
- Interface is declared using the keyword interface.

- An abstract class can only inherit a single super class (abstract or otherwise).
- A class can implement **more than one** interface.

- Interfaces have no direct inherited relationship with any particular class, they are defined **independently.** Interfaces themselves have **inheritance** relationship among themselves.

- An abstract methods of abstract class have abstract modifier.
- A method of interface is an abstract method by default.

Uses of Interface

- To reveal an object's programming interface (functionality of the object) without revealing its implementation.
  - This is the concept of encapsulation.
  - The implementation can change without affecting the caller of the interface.

- To have unrelated classes implement similar methods (behaviors).
  - One class is not a sub-class of another.

- To model multiple inheritance.
  - A class can implement multiple interfaces while it can extend only one class.

Packages

Java provides a mechanism for partitioning the class name space into more manageable chunks. This mechanism is the package.

The package is both a naming and a visibility control mechanism.

A package represents a directory that contains related group of classes and interfaces.

You can define classes inside a package that are not accessible by code outside that package.

You can also define class members that are only exposed to other members of the same package.

Pre-defined packages

1. java.applet
2. java.awt
3. java.beans
4. java.io
5. java.lang
6. java.lang.ref
7. java.math
8. java.net
9. java.nio
10. java.sql
11. java.text
12. java.util
13. java.util.zip
14. javax.sql
15. javax.swing

Defining a Packages

To create a package is quite easy: simply include a **package command as the first** statement in a Java source file.

Any classes declared within that file will belong to the specified package.

The **package statement defines a name space in which classes are** stored.

If you omit the **package statement, the class names are put into the default** package, which has **no name**.

This is the general form of the **package statement:**

Syntax: **package pkg;**

Example: **package MyPackage;**

Java uses file system directories to store packages.

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

You can create a hierarchy of packages.

To do so, simply separate each package name from the one above it by use of a period.

The general form of a multileveled package statement is shown here:

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

A package hierarchy must be reflected in the file system of your Java development system.

For example, a package declared as package java.awt.image; needs to be stored in **java\awt\image on your** Windows.

Finding Packages and CLASSPATH

How does the Java run-time system know where to look for packages that you create?

The answer has two parts:

First, by default, the Java run-time system uses the current working directory as its starting point. Thus, if your package is in the current directory, or a subdirectory of the current directory, it will be found.

Second, you can specify a directory path or paths by setting the **CLASSPATH environmental variable.**

For example, consider the following package specification.

```
package MyPack;
```

In order for a program to find **MyPack, one of two things must be true.**

Either the program is executed from a directory immediately above **MyPack, or CLASSPATH** must be set to include the path to **MyPack.**

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

```
//AccountBalance.java
class AccountBalance {
        public static void main(String args[]) {
                Balance current[] = new Balance[3];
                current[0] = new Balance("K. J. Fielding", 123.23);
                current[1] = new Balance("Will Tell", 157.02);
                current[2] = new Balance("Tom Jackson", -12.33);
                for(int i=0; i<3; i++)
                        current[i].show();
                }
}
```

```
//To compile
javac AccountBalance.java

//To run
java MyPack.AccountBalance

//java AccountBalance        invalid
```

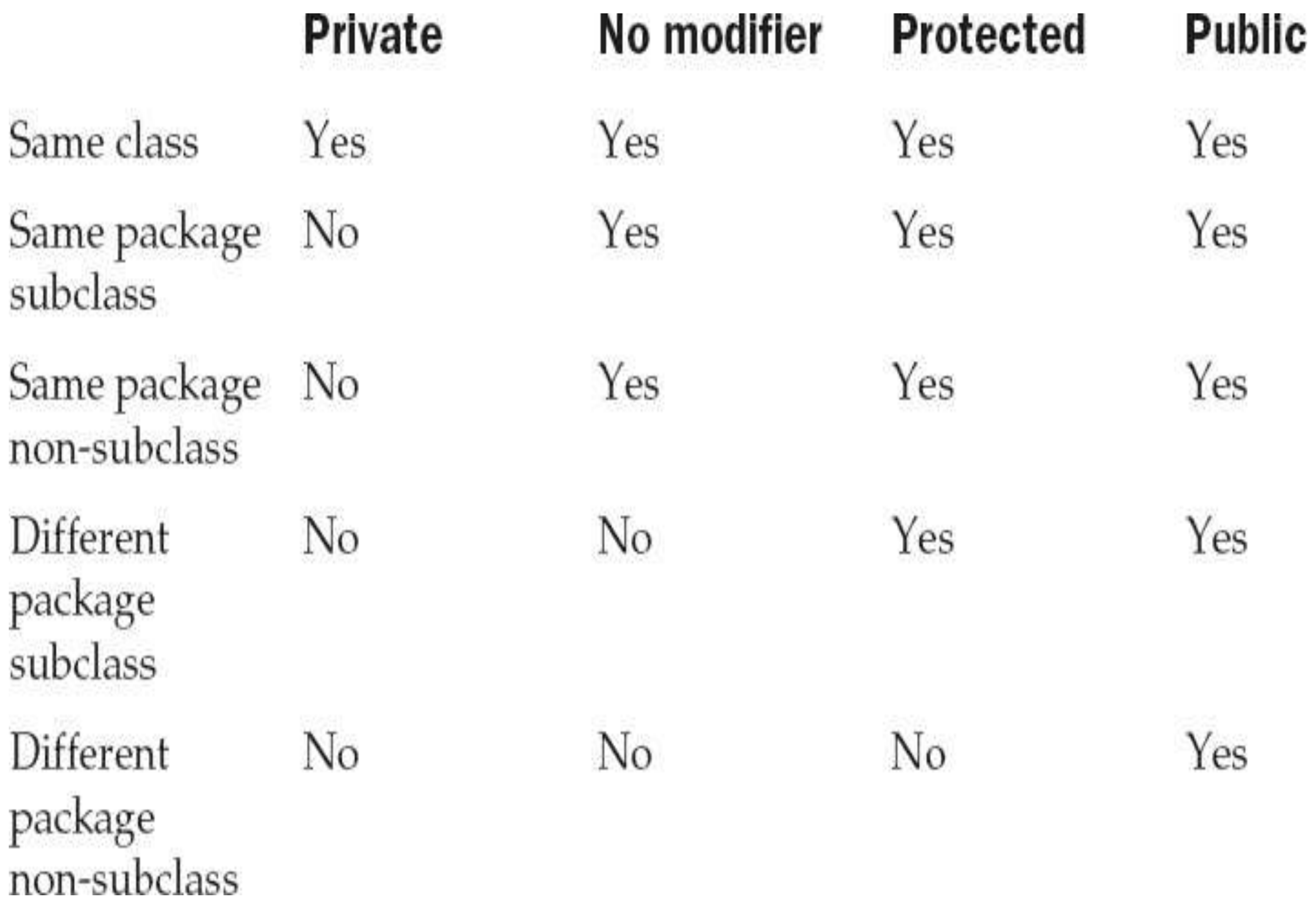
Access Control

Java addresses four categories of visibility for **class members:**

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

A **class** has only two possible access levels: **default** and **public**.

Class Member Access



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

//VarProtection.java

```
package pack1;
public class VarProtection {
 int n = 1;
 private int pri = 2;
 protected int pro = 3;
 public int pub = 4;
 public VarProtection() {
     System.out.println("Individual class constructor");
     System.out.println("default value is: " + n);
     System.out.println("private value is: " + pri);
     System.out.println("protected value is: " + pro);
     System.out.println("public value is: " + pub);
  }
}
```

To Compile:

```
d:\>javac –d . VarProtection.java
```

```
//SameSub .java:
package pack1;
class SameSub extends VarProtection{
    SameSub(){
     System.out.println("subclass constructor");
     System.out.println("default value is: " + n);
     / /  System.out.println("private value is: " + pri);
     System.out.println("protected value is: " + pro);
     System.out.println("public value is: " + pub);
    }
}
```

To Compile:

```
d:\>javac –d . SameSub.java
```

```
// SameDiff.java
package pack1;
class SameDiff{
    SameDiff(){
     VarProtection v1 = new VarProtection();
     System.out.println("Delegationclass constructor");
     System.out.println("default value is: " +v1. n);
     // System.out.println("private value is: " +v1. pri);
     System.out.println("protected value is: " +v1. pro);
     System.out.println("public value is: " + v1.pub);
    }
}
```

To Compile:

```
d:\>javac –d . SameDiff.java
```

//OtherSub.java

```
package pack2;
import pack1.*;
class OtherSub extends VarProtection{
    OtherSub(){
      System.out.println("Different Package subclass constructor");
      //System.out.println("default value is: " + n);
      // System.out.println("private value is: " + pri);
      System.out.println("protected value is: " + pro);
      System.out.println("public value is: " + pub);
    }
}
```

To Compile:

```
d:\>javac –d . OtherSub.java
```

```
// OtherDiff.java
package pack2;
import pack1.*;
class OtherDiff{
    OtherDiff(){
      VarProtection v2=new VarProtection();
      System.out.println("Different Package non-subclass constructor");
      // System.out.println("default value is: " +v2. n);
      // System.out.println("private value is: " + v2.pri);
      // System.out.println("protected value is: " + v2.pro);
      System.out.println("public value is: " + v2.pub);
    }
}
```

To Compile:

```
d:\>javac –d . OtherDiff.java
```

```
// Demo package p1.
package  pack1;
class MainTest{
    public static void main(String args[]){
      VarProtection v=new VarProtection();
      SameDiff s2=new SameDiff();
      SameSub s1=new SameSub();
    }
}
```

To Compile:

```
d:\>javac –d . MainTest.java
```

To Run:

```
d:\>java pack1.MainTest
```

```
package pack2;
import pack1.*;
class OtherMainTest{
    public static void main(String args[]){
      OtherSub os=new OtherSub();
      OtherDiff od=new OtherDiff();
    }
}
```

To Compile:

```
d:\>javac –d . OtherMainTest.java
```

To Run:

```
d:\>java pack2.OtherMainTest
```

Importing Packages

There are no core Java classes in the unnamed default package; all of the standard classes are stored in some named package.

Java includes the **import statement to bring certain classes, or entire** packages, into visibility.

Once imported, a class can be referred to directly, using only its name.

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

This is the general form of the **import statement:**

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

Here, pkg1 is the name of a top-level package, and pkg2 is the name of a subordinate package inside the outer package separated by a dot **(.).**

There is no practical limit on the depth of a package hierarchy, except that imposed by the file system.

Finally, you specify either an explicit classname or a star **(*), which indicates that the Java compiler** should import the entire package.

This code fragment shows both forms in use:

```
import java.util.Date;
import java.io.*;
```

All of the standard Java classes included with Java are stored in a package called **java.**

The basic language functions are stored in a package inside of the java package called **java.lang.**

Normally, you have to import every package or class that you want to use, but **java.lang** is implicitly imported by the compiler for all programs.

This is equivalent to the following line being at the top of all of your programs:

import java.lang.*;

When a package is imported, only those items within the package declared as public will be available to non-subclasses in the importing code. For example, if you want the Balance class of the package MyPack shown earlier to be available as a stand-alone class for general use outside of MyPack, then you will need to declare it as public and put it into its own file, as shown here:

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

```
import MyPack.*;
class TestBalance {
        public static void main(String args[]) {
                /* Because Balance is public, you may use Balance
                        class and call its constructor. */
                Balance test = new Balance("J. J. Jaspers", 99.88);
                test.show(); // you may also call show()
        }
}
```

As an experiment, remove the public specifier from the Balance class and then try compiling TestBalance. As explained, errors will result.

UNIT - 3

Exception Handling in Java

Objectives

- Introduction
- What exceptions are for
- Catching & Throwing exceptions
- Exception Specifications
- Standard Java Exceptions
- Exceptions and Polymorphism
- The **`finally`** clause
- Resource Management
- Uncaught Exceptions

Introduction

- Due to design errors or coding errors, our programs may fail in unexpected ways during execution. An exception is a condition that is caused by run time error in the program. The purpose of the exception handling mechanism is to provide a means to detect and report an "ecxceptional circumstances" .

Error

- An error may produce an incorrect output or may terminate the execution of the program abruptly or even may cause the system to crash. So it is our responsibility to detect and manage the error properly.

Types of error

- Runtime Errors: occur while the program is running if the environment detects an operation that is impossible to carry out.
- Logic Errors: occur when a program doesn't perform the way it was intended
- Syntax Errors: Arise because the rules of the language have not been followed. They are detected by the compiler.

Example of Run Time error

```
Class Error
{
public static void main(String args[])
 {
int a=10;
int b=5;
int c=5;

int x=a/(b+c);
System.out.println("x=" +x);
int y=a/(b-c);   // Errorr division by zero
System.out.println("y=" +y);
}
}
```

Errors and Error Handling

- Some typical causes of errors:
  - Memory errors (i.e. memory incorrectly allocated, memory leaks, “null pointer”)
  - File system errors (i.e. disk is full, disk has been removed)
  - Network errors (i.e. network is down, URL does not exist)
  - Calculation errors (i.e. divide by 0)

Errors and Error Handling

- More typical causes of errors:
  - Array errors (i.e. accessing element –1)
  - Conversion errors (i.e. convert ‘q’ to a number)
  - Can you think of some others?

Errors and Error Handling

- Exceptions – a better error handling
  - Exceptions are a mechanism that provides the best of both worlds.
  - Exceptions act similar to method return flags in that any method may raise and exception should it encounter an error.
  - Exceptions act like global error methods in that the exception mechanism is built into Java; exceptions are handled at many levels in a program, locally and/or globally.

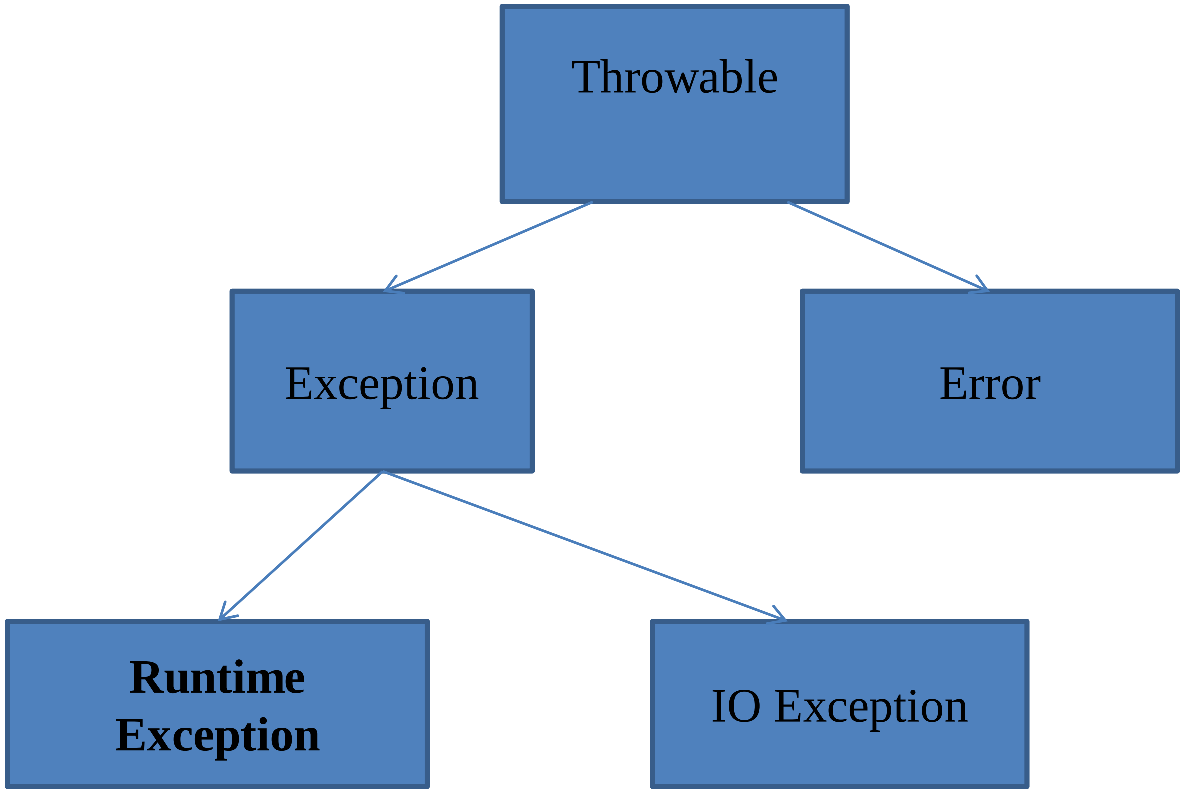
Exceptions

- How do you handle exceptions?
  - To handle the exception, you write a “try-catch” block. To pass the exception “up the chain”, you declare a throws clause in your method or class declaration.
  - If the method contains code that may cause a checked exception, you MUST handle the exception OR pass the exception to the parent class (remember, every class has Object as the ultimate parent)

Coding Exceptions

- Coding Exceptions
- Try-Catch Mechanism
  - Wherever your code may trigger an exception, the normal code logic is placed inside a block of code starting with the “try” keyword:
  - After the try block, the code to handle the exception should it arise is placed in a block of code starting with the “catch” keyword.

Standard Java Exceptions



Catching Exceptions

- Wrap code to be checked in a try-block
  - checking occurs all the way down the execution stack
- try-blocks can be nested
  - control resumes at most enclosed matching handler

Coding Exceptions

- Example
  - try {
    ... normal program code
    }
    catch(Exception e) {
    ... exception handling code
    }

Coding Exceptions

- Types of Exceptions
  - Examples:
    - public void myMethod throws Exception {
    - public void myMethod throws IOException {
    - try { … }
      catch (Exception e) { … }
    - try { … }
      catch (IOException ioe) { … }

Code Examples

- 1. Demonstration of an unchecked exception (NullPointerException)
- 2. Demonstration of checked exceptions:
  - Passing a DivideByZeroException
  - Handling a DivideByZeroException

Example

```
class error2
{
public static void main(String arg[])
{
int a=10;
 int b=5;
 int c=5;
 int x,y;
    try
   {
    x=a/(b-c);
   }
    catch(ArithmeticException e)
   {
   System.out.println(“Division by Zero”);
   }
   Y=a/(b-c);
   System.out.println(“y=“+y);
   }
   }
```

In the previous program we cannot see the value of x just because of the error in the value of y, that is division by zero but when we use the try and catch blocks in exception handling then we can see the value of y which is correct and our program will display an error message shown in the try block.

conclusion

- Exceptions are a powerful error handling mechanism.
- Exceptions in Java are built into the language.
- Exceptions can be handled by the programmer (try-catch), or handled by the Java environment (throws).
- Exception handling can only hide the errors.
- It cannot correct the errors.

UNIT - 4

Multithreaded Programs in Java

Tasks and Threads

- A **task** is an abstraction of a series of steps
  - Might be done in a separate thread
  - Java libraries use the Runnable interface
  - work done by method run()
- **Thread**: a Java class for a thread
  - work done by method run()

- How to associate a task with a thread?
- How to start a thread?

Creating a Task and Thread

- Warning: old way(s), new ways
- First, if you have a thread object, you can call start() on that object
  - Makes it available to be run
  - When it's time to run it, Thread's run() is called
- So, create a thread using "old" (not good) way
  - Write class that extends Thread, e.g. MyThread
  - Define your own run()
  - Create a MyThread object and call start() on it
- We won't do this! Not good design

Runnables and Thread

- Use the “task abstraction” and create a class that implements Runnable interface
  - Define the run() method to do the work you want
- Now, two ways to make your task run in a separate thread
  - Create a Thread object and pass a Runnable to the constructor
  - As before, call start() on the Thread object

Do we need a Thread “manager”?

- If your code is responsible for creating a bunch of tasks, linking them with Threads, and starting them all, then you have things to worry about:
  - What if you start too many threads? Can you manage the number of running threads?
  - Can you shutdown all the threads?
  - If one fails, can you restart it?

Executors

- An Executor is an object that manages running tasks
  - Submit a Runnable to be run with Executor's execute() method
  - So, instead of creating a Thread for your Runnable and calling start() on that, do this:
    - Get an Executor object, say called exec
    - Create a Runnable, say called myTask
    - Submit for running: exec.execute(myTask)

How to Get an Executor

- Use static methods in Executors library.
- Fixed "thread pool": at most N threads running at one time

```
Executor exec =
   Executors.newFixedThreadPool(MAX_THREADS);
```

- Unlimited number of threads

```
Executor exec =
   Executors.newCachedThreadPool();
```

Summary So Far

- Create a class that implements a Runnable to be your “worker”
- Create Runnable objects
- Create an Executor
- Submit each Runnable to the Executor which starts it up in a separate thread

Synchronization

- Understand the issue with concurrent access to shared data?
  - Data could be a counter (int) or a data structure (e.g. a Map or List or Set)
- A <u>critical section</u>: a block of code that can only be safely executed by one thread at a time
- A lock: an object that is “held” by one thread at a time, then “released”

Synchronization in Java (1)

- Any object can serve as a lock
  - Separate object: `Object myLock = new Object();`
  - Current instance: the this object
- Enclose lines of code in a *synchronized* block

```
    synchronized(myLock) {
        // code here
    }
```

- More than one thread could try to execute this code, but one acquires the lock and the others “block” or wait until the first thread releases the lock

Synchronized Methods

- Common situation: all the code in a method is a critical section
  - I.e. only one thread at a time should execute that method
  - E.g. a getter or setter or mutator, or something that changes shared state info (e.g. a Map of important data)
- Java makes it easy: add synchronized keyword to method signature. E.g.

```
  public synchronized void update(…) {
```

Summary So Far

- Concurrent access to shared data
  - Can lead to serious, hard-to-find problems
  - E.g. race conditions
- The concept of a lock
- Synchronized blocks of code or methods
  - One thread at a time
  - While first thread is executing it, others block

More Advanced Synchronization

- A <u>semaphore</u> object
  - Allows simultaneous access by N threads
  - If N==1, then this is known as a <u>mutex</u> (mutual exclusion)
  - Java has a class Semaphore
- Java class CountDownLatch
  - Created with a count (often a number of "worker" threads). Say object is `allWorkersDone`
  - Another thread (a "manager") waits for all the workers to call countDown() on that object
  - So manager blocks with: `allWorkersDone.await()`

Barriers

- Java class CyclicBarrier
  - A rendezvous point or barrier point
  - Worker threads wait at a spot until all get there
  - Then all proceed

Using CountDownLatch

- Here are some common scenarios and demo programs for them
- You’ll use the last of these for the War card-game program!

Scenario #1

- A “manager” thread and N “worker” threads
- Manager starts workers but then must wait for them to finish before doing follow-up work
- Solution:
  - Manager creates a CountDownLatch with value N
  - After workers starts, manager calls await() on that
  - When each worker completes its work, it calls countDown() on the latch
  - After all N call countDown(), manager is un-blocked and does follow-up work
- Example use: parallel divide and conquer like mergesort
- Code example: SyncDemo0.java

Scenario #2

- A “manager” thread and N “worker” threads
- Manager starts workers but wants them to “hold” before doing real work until it says “go”
- Solution:
  - Manager creates a CountDownLatch with value 1
  - After each workers start, it calls await() on that Latch
  - At some point, when ready, the manager calls countDown() on that Latch
  - Now Workers free to continue with their work
- Code example: SyncDemo1.java

Scenario #3

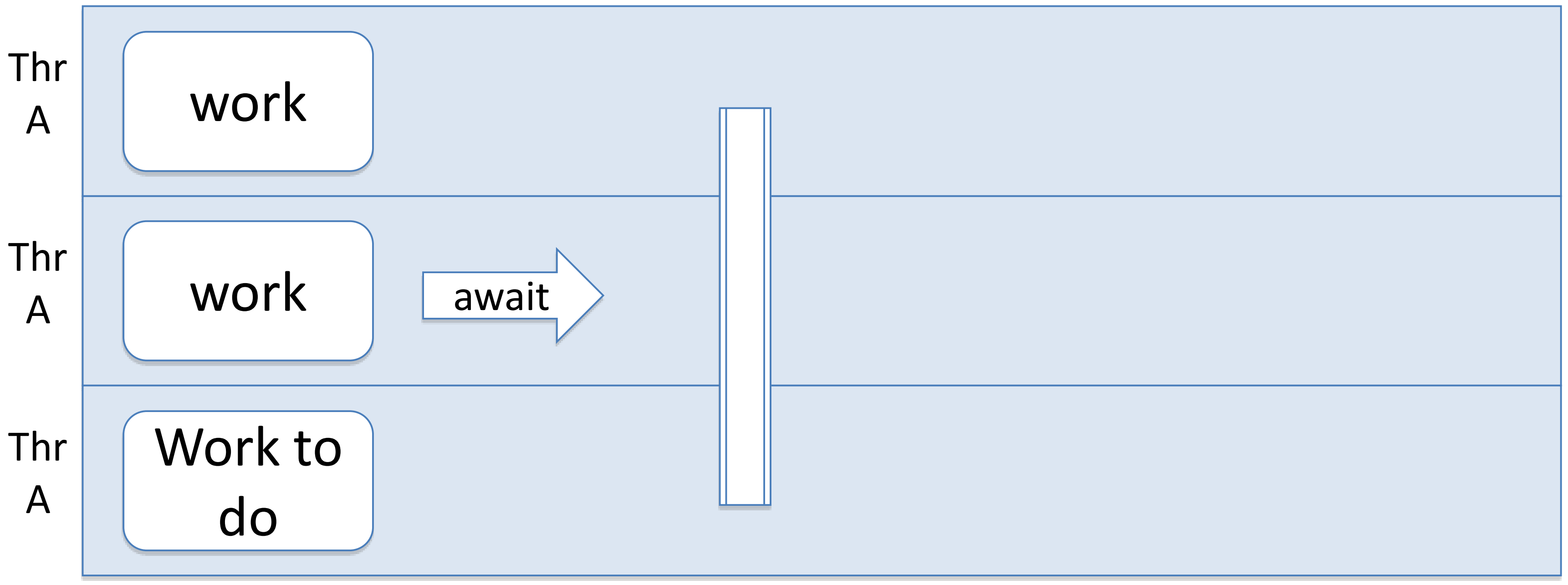
- Work done in “rounds” where:
  - All workers wait for manager to say “go”
  - Each worker does its job and then waits for next round
  - Manager waits for all workers to complete a round, then does some follow-up work
  - When that’s done, manager starts next round by telling workers “go”
- Solution: combine the two previous solutions
  - First Latch: hold workers until manager is ready
  - Second Latch: manager waits until workers finish a round
  - Worker’s run() has loop to repeat
  - Manager must manage Latches, recreating them at end of round
- Example use: a card game or anything that has that kind of structure
- Code example: SyncDemo2.java

Summary of last section

- Multiple threads may need to cooperate
  - Common situation: some workers and a manager
  - One thread may need to wait for one or more thread to complete
  - One or more threads may need to wait to be "released"
  - Or a combination of these situations
- Threads all access a CountDownLatch
  - await() used to wait for enough calls to countDown()

End

- Unused slides follow



dfafaf

Java applets

Introduction

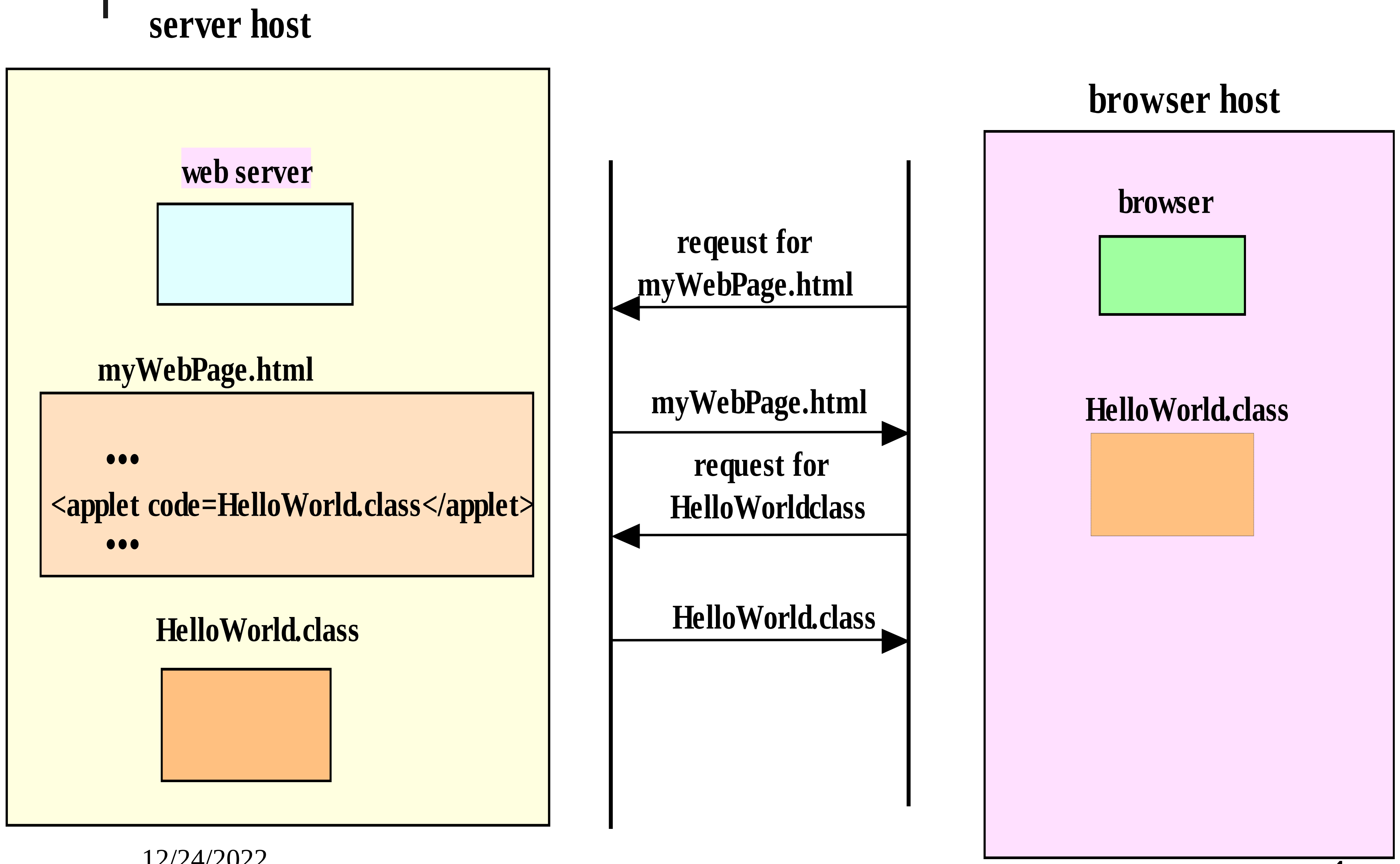
Java applets are one of three kinds of Java programs:

- An ***application*** is a standalone program that can be invoked from the command line.
- An ***applet*** is a program that runs in the context of a browser session.
- A ***servlet*** is a program that is invoked on a server program, and it runs in the context of a web server process.

Applets, web page, client, server

- Applets are programs stored on a web server, similar to web pages.
- When an applet is referred to in a web page that has been fetched and processed by a browser, the browser generates a request to fetch (or download) the applet program, then executes the applet program in the browser's execution context <u>on the client host</u>.

Applets, web page, client, server



Applet Execution - 1

- An applet program is a written as a subclass of the java.Applet class or the javax.swing.Japplet class.
- There is no main() method in an Applet.
- An applet uses AWT for graphics, or JApplet, a subclass of javax.swing.

Applet Execution - 2

- **Life Cycle of an Applet:**
  - **init: This method is intended for whatever initialization is needed for an applet.**
  - **start: This method is automatically called after init method. It is also called whenever user returns to the page containing the applet after visiting other pages.**
  - **stop: This method is automatically called whenever the user moves away from the page containing applets. This method can be used to stop an animation.**
  - **destroy: This method is only called when the browser shuts down normally.**

  - Ref: http://java.sun.com/docs/books/tutorial/deployment/applet/index.html/

Applet Execution - 3

- The applet is running and rendered on the web page.
- Every Applet needs to implement one of more of the init(), the start( ) and the paint( ) methods.
- At the end of the execution, the stop( ) method is invoked, followed by the destroy( ) method to deallocate the applet's resources.

Applet Security

For security reasons, applets that are loaded over the network have several restrictions.

- an applet **cannot** ordinarily **read** or **write** files on the computer that it's executing on.
- an applet **cannot** **make** network connections except to the **host** that it came from.

- Ref: http://java.sun.com/docs/books/tutorial/deployment/applet/index.html/

HTML tags for applets - 1

```
<APPLET
  // the beginning of the HTML applet code
   CODE="demoxx.class"
   // the actual name of the applet (usually a 'class' file)
   CODEBASE="demos/"
   // the location of the applet (relative as here, or a full URL)
    NAME="SWE622"
    // the name of the instance of the applet on this page
    WIDTH="100"
    // the physical width of the applet on the page
    HEIGHT="50"
    // the physical height of the applet on the page
    ALIGN="Top"
    // align the applet within its page space (top, bottom, center)
```

HTML tags for applets - 2

```
<APPLET CODE="SWE622.class" CODEBASE="example/"
                    WIDTH=460 HEIGHT=160
                    NAME="buddy" >
<PARAM NAME="imageSource" VALUE="images/Beans">
<PARAM NAME="backgroundColor" VALUE="0xc0c0c0">
<PARAM NAME="endImage" VALUE=10>
</APPLET>
```

The HelloWorld Applet

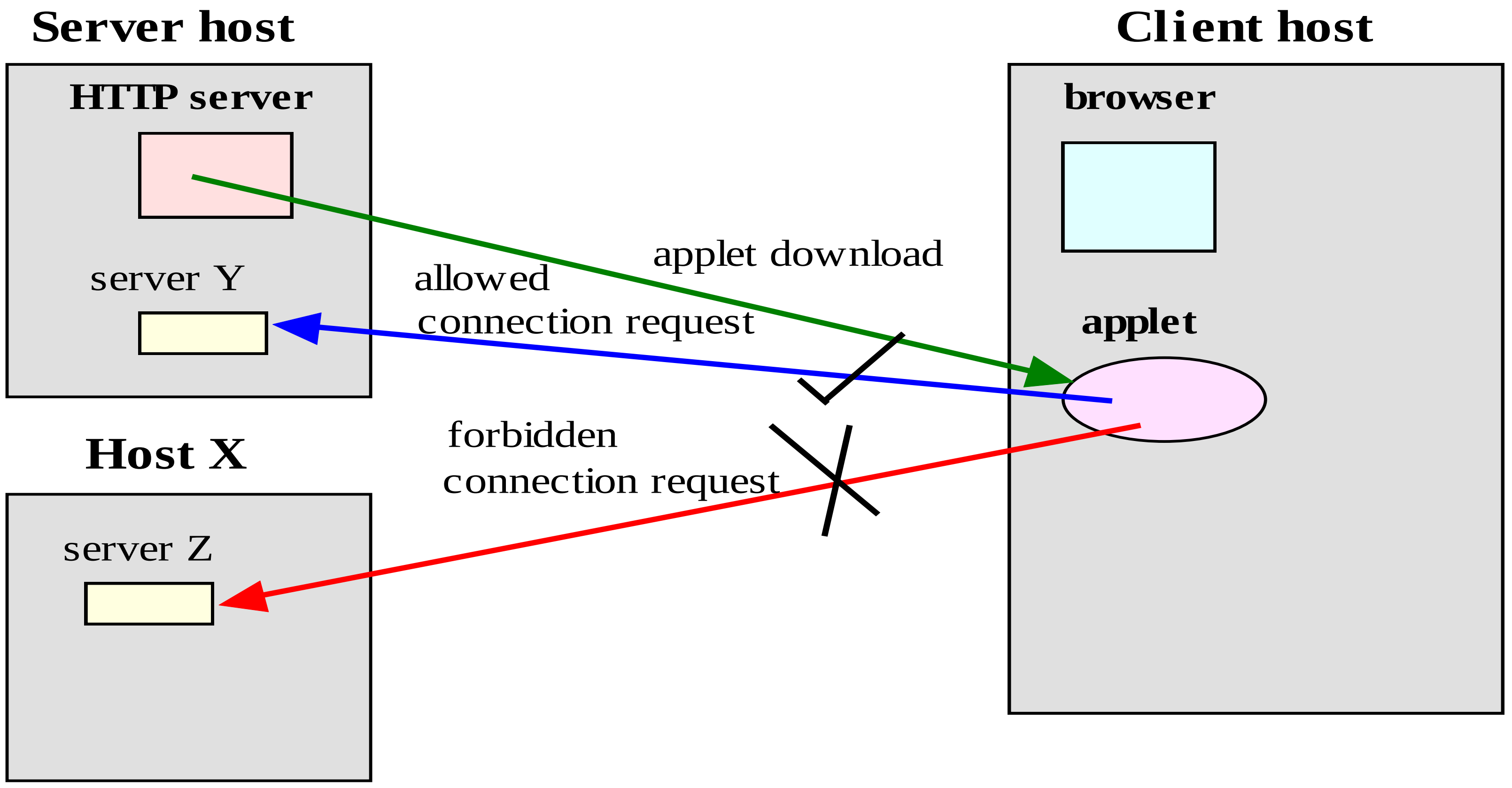
```
<HTML>
<BODY>
<APPLET code=hello.class width=900 height=300>
</APPLET>
</BODY>
</HTML>
```

```
// applet to display a message in a window
import java.awt.*;
import java.applet.*;

public class hello extends Applet {
    public void init( ) {
        setBackground(Color.yellow);
    } // end of init()
```

```
  public void paint(Graphics g) {
      final int FONT_SIZE = 42;
      Font font = new Font("Serif",
         Font.BOLD, FONT_SIZE);
// set font, and color and display message
// on the screen at position 250,150
         g.setFont(font);
         g.setColor(Color.blue);
// The message in the next line is the one
// you will see
         g.drawString("Hello,
                world!",250,150);
   } // end of paint()

} // end of hello
```

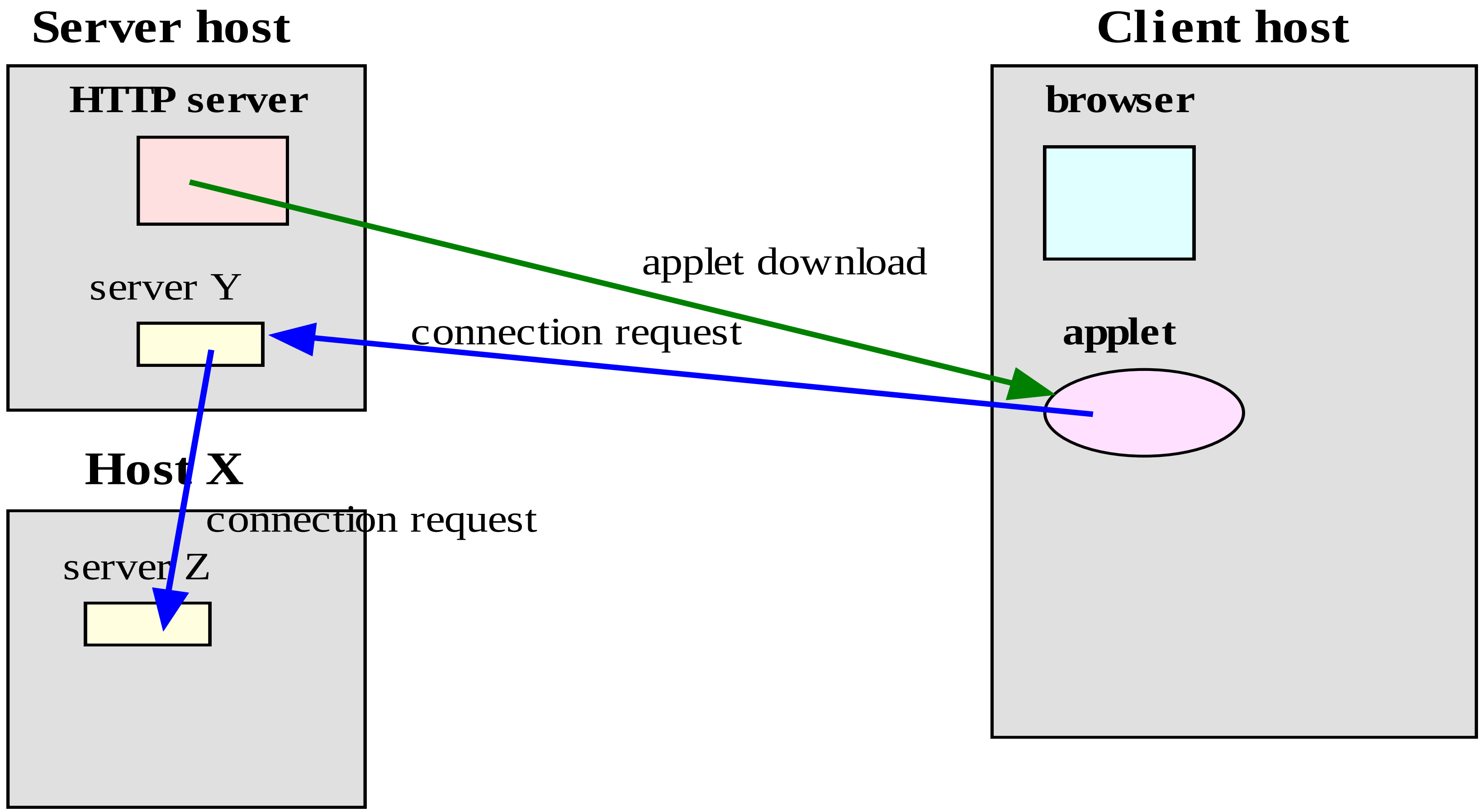
Advanced Applets

- You can use threads in an applet.
- You can make socket/RMI calls (**hw_#3**) in an applet, subject to the security constraints.



Proxy server

A **proxy** server (Y) can be used to circumvent the security constraints.



Summary

- An **applet** is a Java class
- Its code is **downloaded** from a web server
- It runs in the browser's environment on the client host
- It is invoked by a browser when it scans a web page and encounters a class specified with the ***APPLET*** tag
- For security reason, the execution of an applet is normally subject to restrictions:
  - applets cannot access files in the file system on the client host
  - Applets cannot make network connection exception to the server host from which it originated