INDEX

UNIT 1 PPT SLIDES

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.

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.

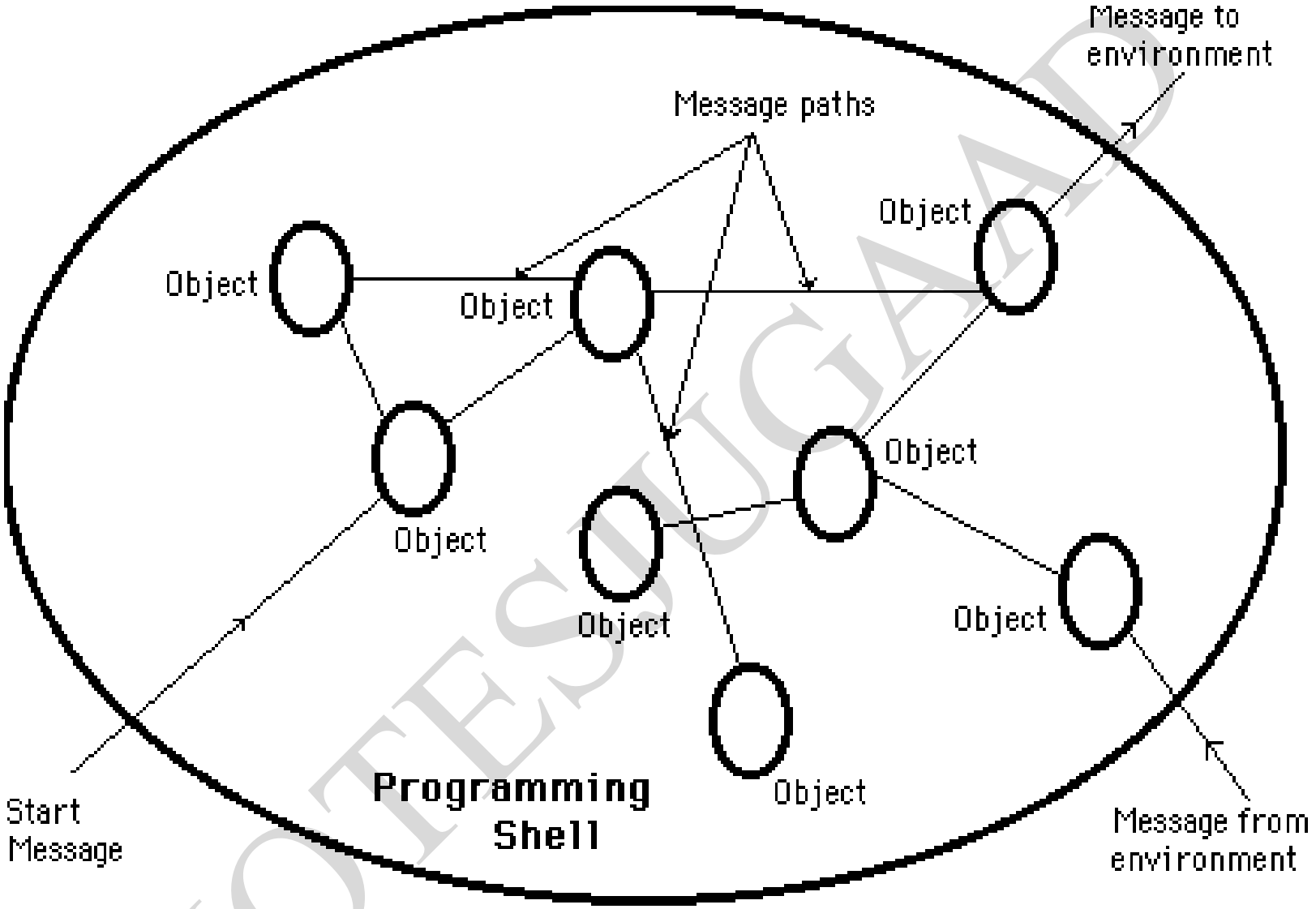
4 Methods that operate on the data of an object are tied together in the data structure.

5 Data is hidden and can not be accessed by external functions.

6 Objects may communicate with each other through methods.

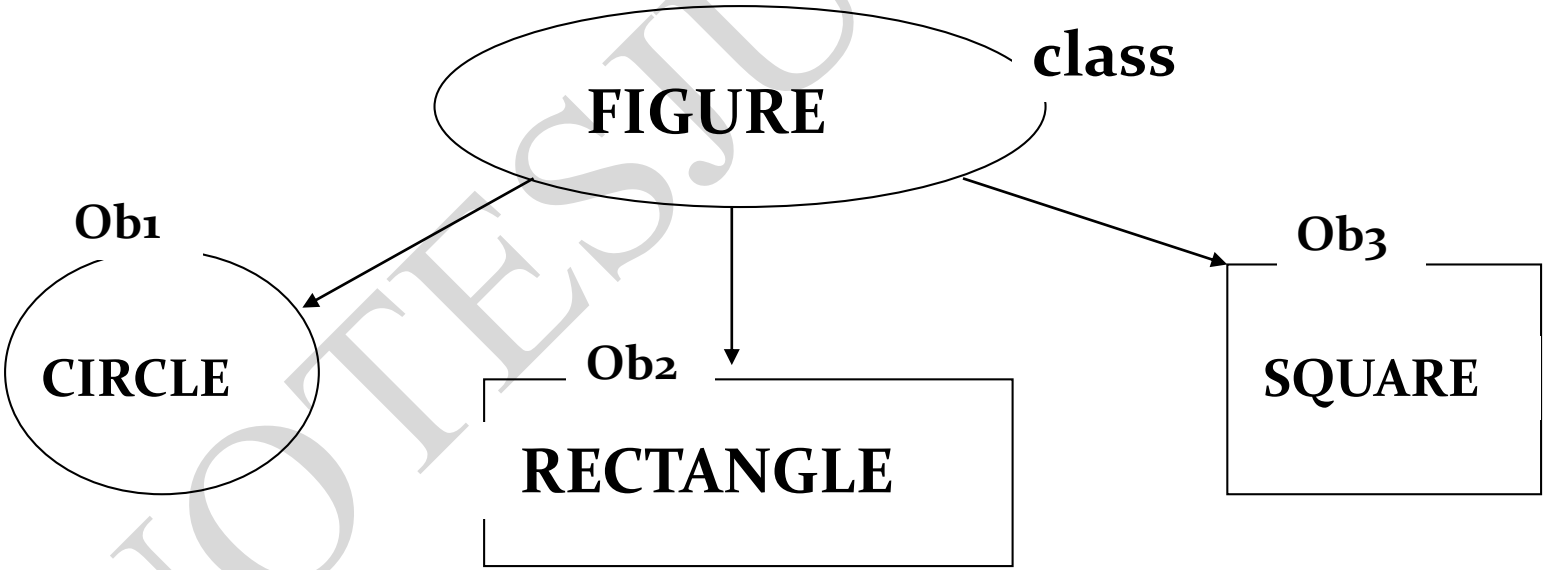
A way of viewing world – Agents

- 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.



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

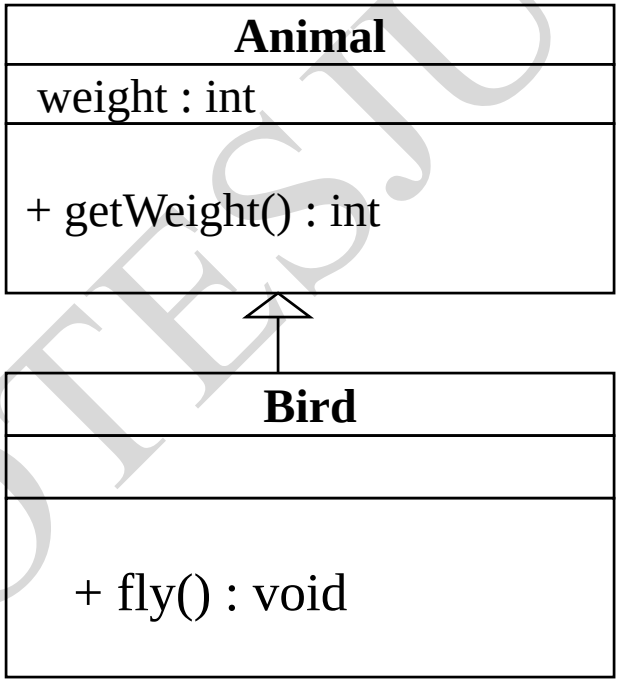
- In Java, class "Object" is the base class to all other classes
  - If we do not explicitly say extends in a new class definition, it implicitly extends Object
  - The tree of classes that extend from Object and all of its subclasses are is called the class hierarchy
  - All classes eventually lead 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 evaluatng 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.

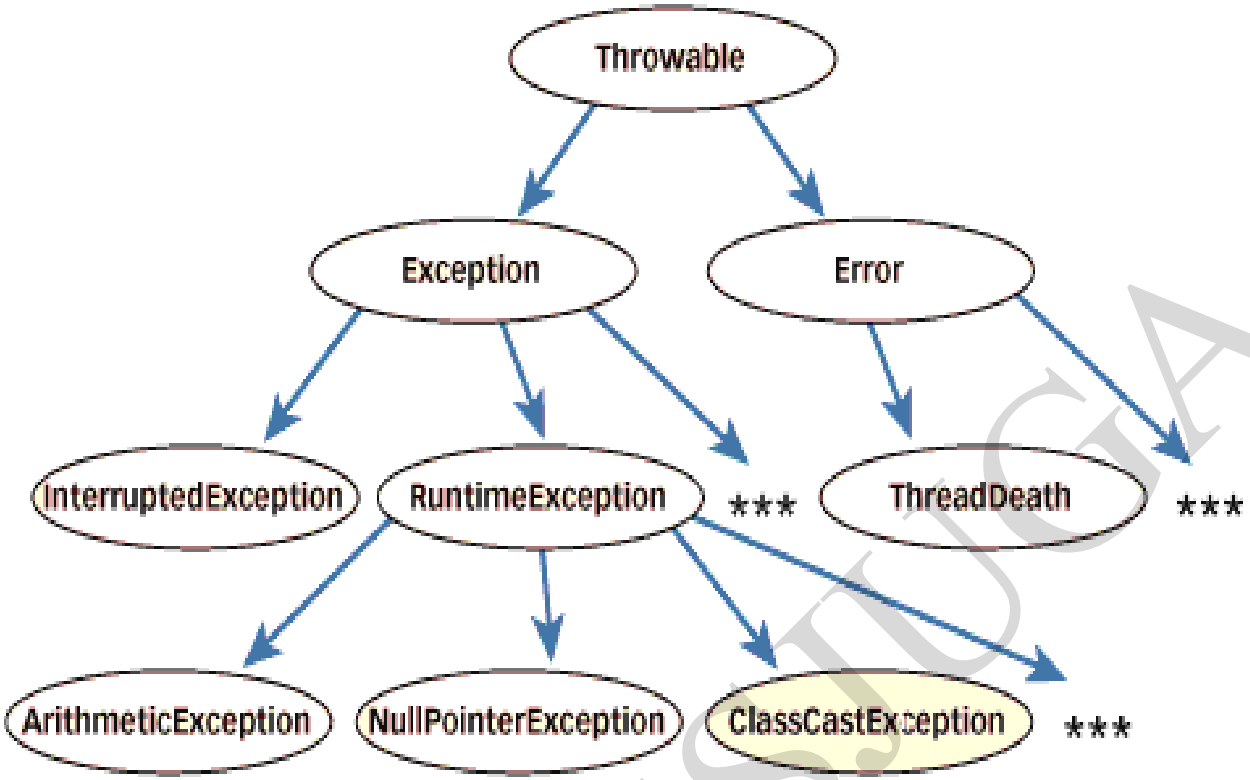
- There may be some occasions when we want an object to respond to the same method but have different behaviour when that method is called.
- That means, we should override the method defined in the superclass. This is possible by defining a method in a sub class that has the same name, same arguments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



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.

Data Abstraction

- Data Abstractions organize data.

StudentType

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

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 = 10000000000000000;
  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
- type promotion – how the type of a variable can be expanded

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 class
- When 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; }
}
```

Keyword this

- Can be used by any object to refer to itself in any class method
- Typically used to
  - Avoid variable name collisions
  - Pass the receiver as an argument
  - Chain constructors

Keyword this

- Keyword this allows a method to refer to the object that invoked it.
- It can be used inside any method to refer to the current object:

```
Box(double width, double height, double depth) {
this.width = width;
this.height = height;
this.depth = depth;
}
```

Garbage Collection

- Garbage collection is a mechanism to remove objects from memory when they are no longer needed.
- Garbage collection is carried out by the garbage collector:
- 1) The garbage collector keeps track of how many references an object has.
- 2) It removes an object from memory when it has no longer any references.
- 3) Thereafter, the memory occupied by the object can be allocated again.
- 4) The garbage collector invokes the finalize method.

finalize() Method

- A constructor helps to initialize an object just after it has been created.
- In contrast, the finalize method is invoked just before the object is destroyed:
- 1) implemented inside a class as:

  protected void finalize() { … }

- 2) implemented when the usual way of removing objects from memory is insufficient, and some special actions has to be carried out

Method Overloading

- It is legal for a class to have two or more methods with the same name.
- However, 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.

Example: Overloading

```
class OverloadDemo {
void test() {
System.out.println("No parameters");
}
void test(int a) {
System.out.println("a: " + a);
}
void test(int a, int b) {
System.out.println("a and b: " + a + " " + b);
}
double test(double a) {
System.out.println("double a: " + a); return a*a;
}
}
```

Constructor Overloading

```
class Box {
double width, height, depth;
Box(double w, double h, double d) {
width = w; height = h; depth = d;
}
Box() {
width = -1; height = -1; depth = -1;
}
Box(double len) {
width = height = depth = len;
}
double volume() { return width * height * depth; }
}
```

Parameter Passing

- Two types of variables:
  1) simple types
  2) class types
- Two corresponding ways of how the arguments are passed to methods:
- 1) by value a method receives a cope of the original value; parameters of simple types
- 2) by reference a method receives the memory address of the original value, not the value itself; parameters of class types

Call by value

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

Call by refference

- **As the parameter hold the same address as the argument, changes to the object inside the method do affect the object used by the argument:**

```
class CallByRef {
public static void main(String args[]) {
Test ob = new Test(15, 20);
System.out.print("ob.a and ob.b before call: “);
System.out.println(ob.a + " " + ob.b);
ob.meth(ob);
System.out.print("ob.a and ob.b after call: ");
System.out.println(ob.a + " " + ob.b);
}
}
```

Recursion

- A recursive method is a method that calls itself:

1) all method parameters and local variables are allocated on the stack
2) arguments are prepared in the corresponding parameter positions
3) the method code is executed for the new arguments
4) upon return, all parameters and variables are removed from the stack
5) the execution continues immediately after the invocation point

Example: Recursion

```
class Factorial {
int fact(int n) {
if (n==1) return 1;
return fact(n-1) * n;
}
}
class Recursion {
public static void main(String args[]) {
Factorial f = new Factorial();
System.out.print("Factorial of 5 is ");
System.out.println(f.fact(5));
} }
```

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: " +
```

```
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:

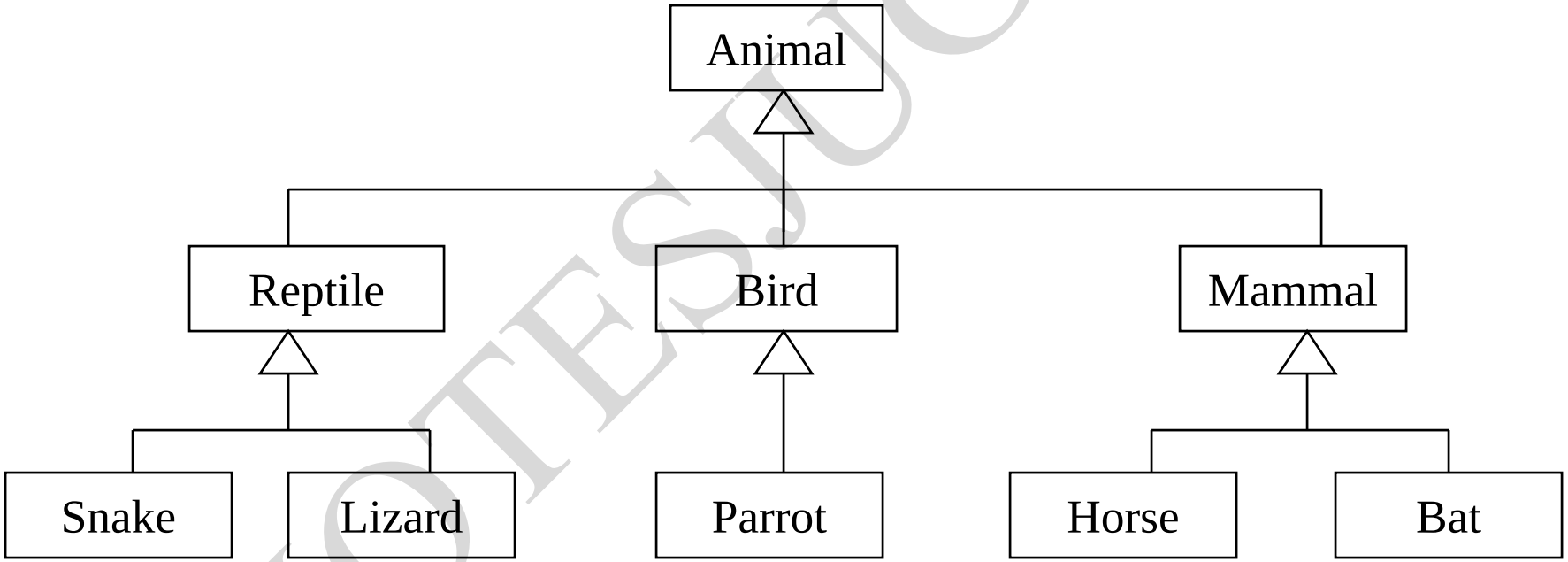
```
Length of strOb1: 12
Char at index 3 in strOb1: s
strOb1 != strOb2
strOb1 == strOb3
```

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.

Base Class **Object** (cont)

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

- **One of the pillars of object-orientation.**
- **A new class is derived from an existing class:**
  - **1) existing class is called super-class**
  - **2) derived class is called sub-class**
- **A sub-class is a specialized version of its super-class:**
  - **1) has all non-private members of its super-class**
  - **2) may provide its own implementation of super-class methods**
- **Objects of a sub-class are a special kind of objects of a super-class.**

extends

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

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

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

- **One baseobj**;// base class object.
- super class object **baseobj** can be used to refer its sub class objects.
- For example, **Two subobj=new Two;**
- **Baseobj=subobj** // now its pointing to sub class

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.

Subclass, Subtype, and Substitutability

- 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

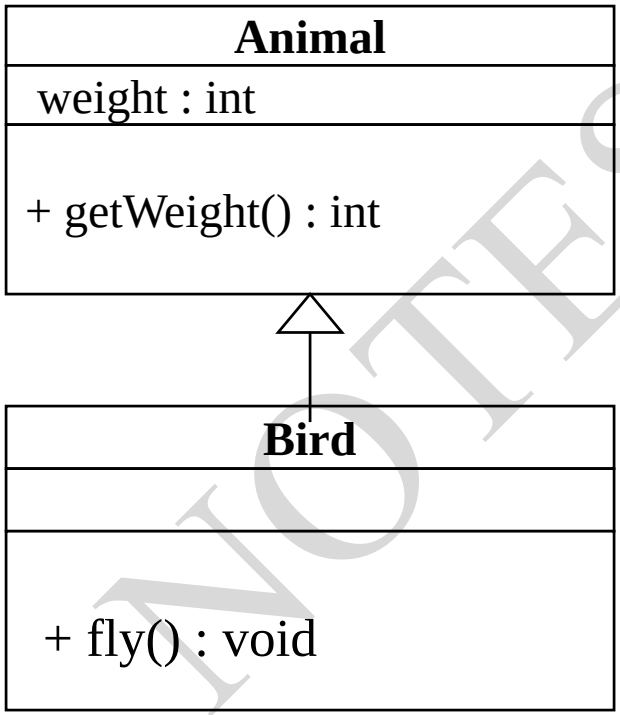
- The term *subtype* is used to describe the relationship between types that explicitly recognizes the principle of *substitution*. A type B is considered to be a subtype of A if an instances of B can legally be assigned to a variable declared as of type A.
- The term *subclass* refers to inheritance 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

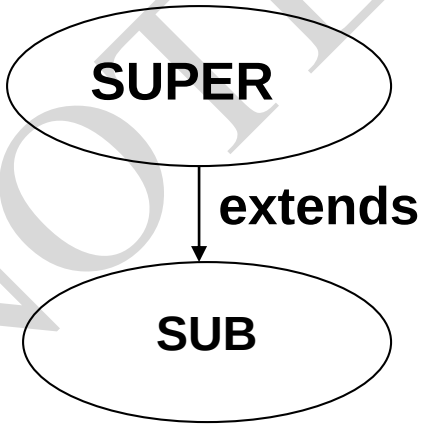
The Costs of Inheritance

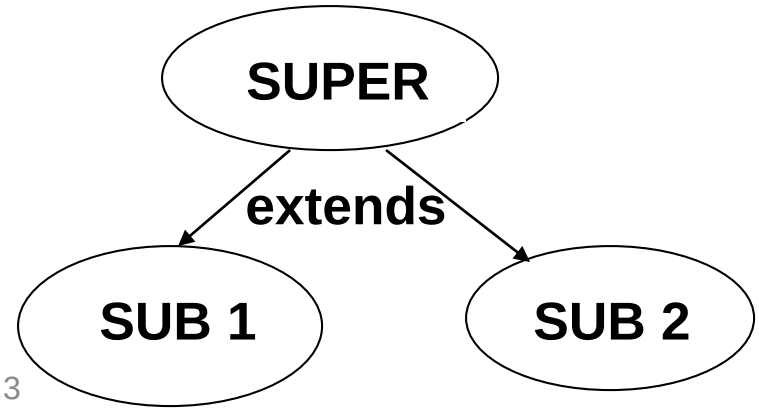
- Execution Speed
- Program Size
- Message-Passing Overhead
- Program Complexity (in overuse of inheritance)

Types of inheritance

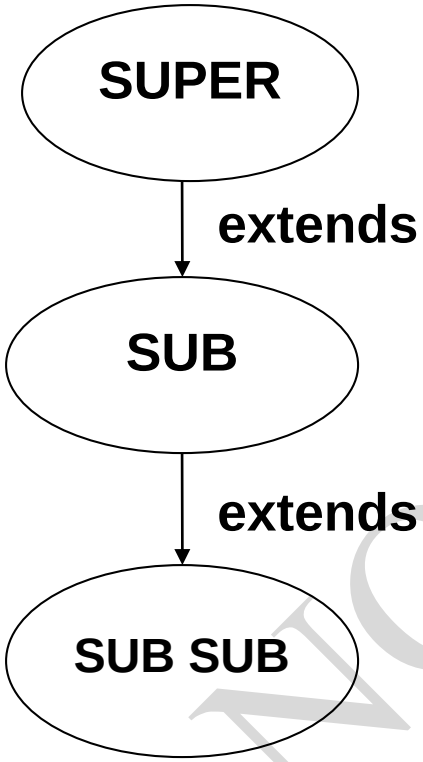
- Acquiring the properties of an existing Object into newly creating Object to overcome the redeclaration of properties in deferent classes.
- These are 3 types:

  1.Simple Inheritance

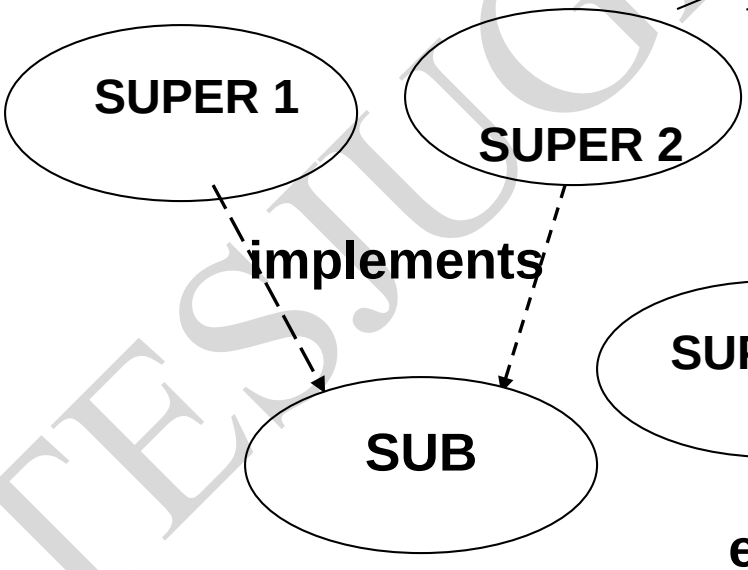


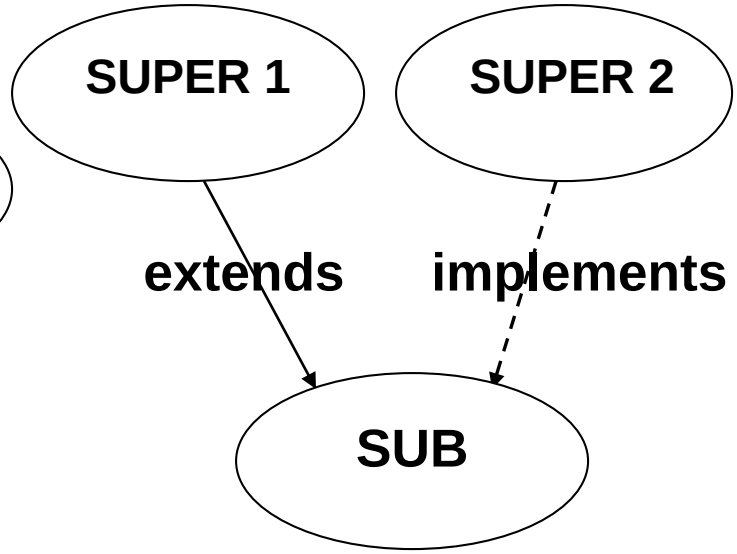


2. Multi Level Inheritance



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.

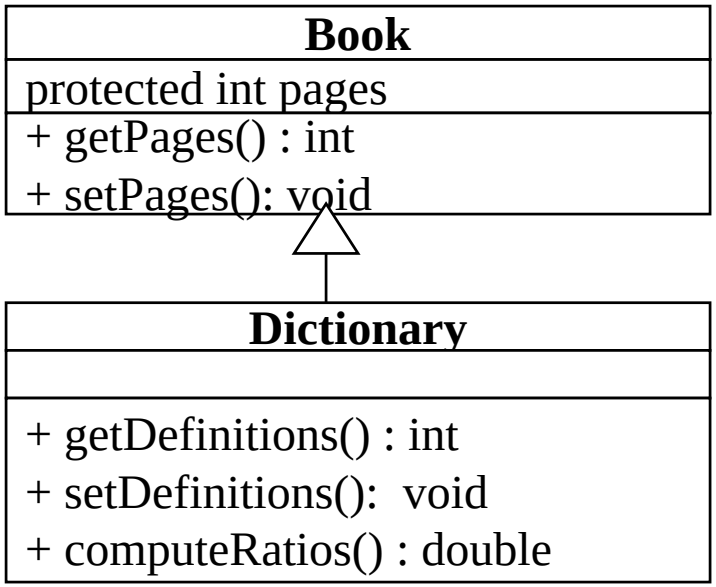
`protected` : can be accessed only within the class definition in which it appears, within other classess in the same package, or within the definition of subclassess.

`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



Example: Super-Class

```
class A {
int i;
void showi() {
System.out.println("i: " + i);
}
}
```

Example: Sub-Class

```
class B extends A {
int j;
void showj() {
System.out.println(“j: " + j);
}
void sum() {
System.out.println("i+j: " + (i+j));
}
}
```

Example: Testing Class

```
class SimpleInheritance {
public static void main(String args[]) {
A a = new A();
B b = new B();
a.i = 10;
System.out.println("Contents of a: ");
a.showi();
b.i = 7; b.j = 8;
System.out.println("Contents of b: ");
subOb.showi(); subOb.showj();
System.out.println("Sum of I and j in b:");
b.sum();}}
```

Multi-Level Class Hierarchy

The basic Box class:

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

Multi-Level Class Hierarchy

Adding the weight variable to the Box class:

```
class BoxWeight extends Box {
double weight;
BoxWeight(BoxWeight ob) {
super(ob); weight = ob.weight;
}
BoxWeight(double w, double h, double d, double m) {
super(w, h, d); weight = m;
}
}
```

Multi-Level Class Hierarchy

Adding the cost variable to the BoxWeight class:

```
class Ship extends BoxWeight {
double cost;
Ship(Ship ob) {
super(ob);
cost = ob.cost;
}
Ship(double w, double h,
double d, double m, double c) {
super(w, h, d, m); cost = c;
}}
```

Multi-Level Class Hierarchy

```
class DemoShip {
public static void main(String args[]) {
Ship ship1 = new Ship(10, 20, 15, 10, 3.41);
Ship ship2 = new Ship(2, 3, 4, 0.76, 1.28);
double vol;
vol = ship1.volume();
System.out.println("Volume of ship1 is " + vol);
System.out.print("Weight of ship1 is”);
System.out.println(ship1.weight);
System.out.print("Shipping cost: $");
System.out.println(ship1.cost);
```

Multi-Level Class Hierarchy

```
vol = ship2.volume();
System.out.println("Volume of ship2 is " + vol);
System.out.print("Weight of ship2 is “);
System.out.println(ship2.weight);
System.out.print("Shipping cost: $“);
System.out.println(ship2.cost);
}
}
```

"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

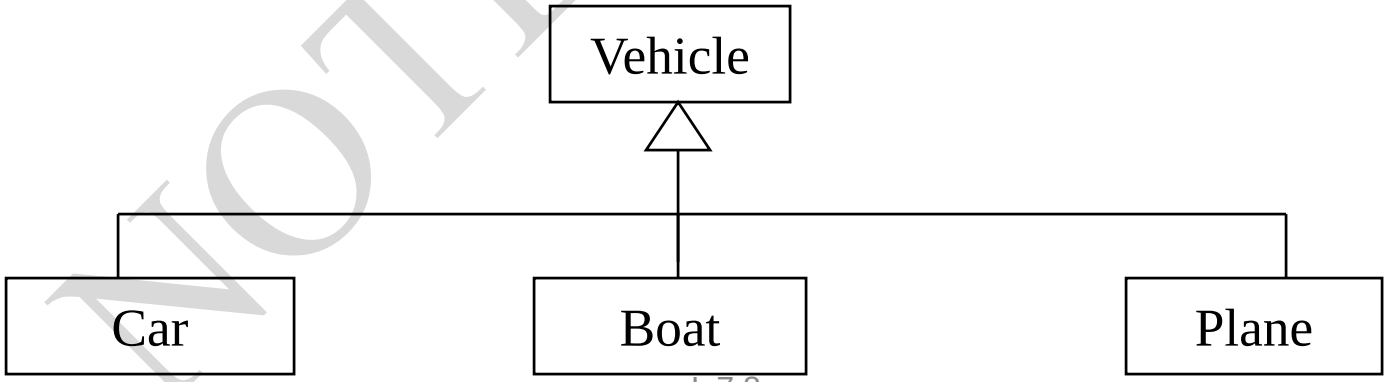
• Overloading deals with multiple methods in the same class with the same name but different signatures • Overloading lets you define a similar operation in different ways for different data	• Overriding deals with two methods, one in a parent class and one in a child class, that have the same signature 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<0) 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.

Defining an Interface

- **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;
}
```

Example: Interface Variables 2

- 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;
}
}
```

Example: Interface Variables 3

- 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();
}
}
```

Package java.io

- 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.
- **DataOutput**The 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.
- **Serializable**Serializability 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 character streams: It reads bytes and decodes them into characters using a specified charset.

- **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.