ADVANCED OBJECT ORIENTED PROGRAMMING

Engr. Anees Ahmed Soomro

Assistant Professor

CS QUEST Nawabshah

https://anees-soomro.neocities.org

OOP CONCEPTS

1) Encapsulation
2) Inheritance
3) Polymorphism
4) History and Features of Java

Encapsulation

- *Encapsulation* is the mechanism that binds together code and data it manipulates, and keeps both safe from outside interference and misuse.
- One way to think about encapsulation is as a protective wrapper that prevents the code and data from being arbitrarily accessed by other code defined outside the wrapper.
- Access to the code and data inside the wrapper is tightly controlled through a well-defined interface.
- Encapsulation is the process of binding together the methods and data variables as a single entity.
- It keeps both the data and functionality code safe from the outside world.
- It hides the data within the class and makes it available only through the methods.
- Java provides different accessibility scopes (public, protected, private ,default) to hide the data from outside.
- Here which we create a class **"Check"** which has a variable **"amount"** to store the current amount.
- Now to manipulate this variable we create a methods and to set the value of amount we create **setAmount()** method and to get the value of amount we create **getAmount()** method .

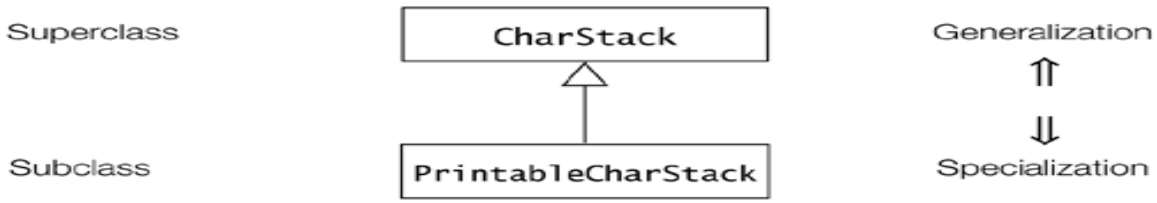
Encapsulation

```
class Check{
      private int amount=0;
      public int getAmount()
{ return amount; }
      public void setAmount(int amt)
{amount=amt;}
    }
public class Mainclass{
      public static void main(String[] args){
         int amt=0;
         Check obj= new Check();
         obj.setAmount(200);
         amt=obj.getAmount();
         System.out.println("Your current amount is :"+amt);
         }
}
```

Inheritance

- There are two fundamental mechanisms for building new classes from existing ones. **inheritance** and **aggregation**.
- It makes sense to inherit from an existing class **Vehicle** to define a class **Car**, since a car is a **vehicle**.
- The class **Vehicle** has several parts; therefore, it makes sense to define a composite object of class Vehicle that has constituent objects of such classes as **Motor**, **Axle**, and **GearBox**, which make up a vehicle.
- Inheritance is illustrated by example that implements a stack of characters that can print its elements on terminal.
- This new stack has all the properties and behaviors of the **CharStack** class, but it also has the additional capability of printing its elements.
- Given that this printable stack is a stack of characters, it can be derived from the **CharStack** class.
- This relationship is shown in Figure 1.6. The class **PrintableCharStack** is called the subclass, and the class **CharStack** is called the superclass.
- The **CharStack** class is a generalization for all stacks of characters, whereas the class **PrintableCharStack** is a specialization of stacks of characters that can also print their elements.

Figure 1.6. Class Diagram Depicting Inheritance Relation



- Deriving a new class from an existing class requires the use of the extends clause in the subclass definition.
- A subclass can extend only one superclass.
- The subclass inherits members of the superclass.
- Inheritance allows a class (subclass) to acquire the properties and behavior of another class (superclass).
- In java, a class can inherit only one class (superclass) at a time but a class can have any number of subclasses.
- It helps to reuse, customize and enhance the existing code.
- So it helps to write a code accurately and reduce the development time.
- Java uses extends keyword to extend a class.

```
class A
{
    public void fun1(int x){
      System.out.println("Int in A is :" + x);
    }
  }
  class B extends A{
    public void fun2(int x,int y){
      fun1(6);  // prints "int in A"
      System.out.println(“int in B is :" + x " and "+y);
    }
  }
  public class Inherit{
    public static void main(String[] args){
      B obj= new B();
      obj.fun2(2,3);
      }
  }
```

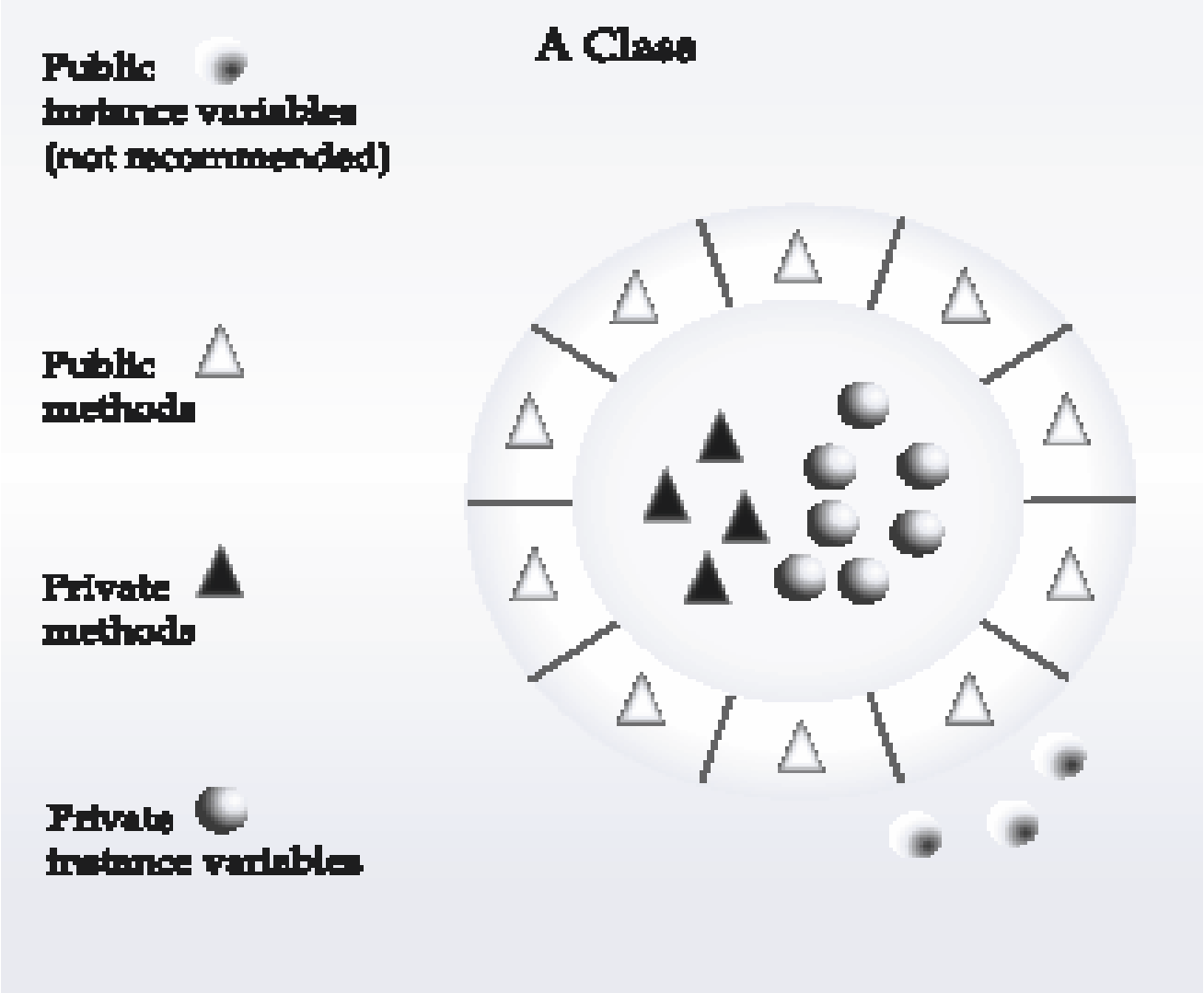


Figure 2-1. *Encapsulation: public methods can be used to protect private data*

Polymorphism

- Polymorphism(from Greek, means “many forms”)is a feature allowing an interface to be used for a general class of actions.
- Specific action is determined by the exact nature of the situation. Consider a stack (which is a last-in, first-out list).
- You might have a program that requires three types of stacks.
- One stack is used for integer values, one for floating-point values, and one for characters.
- The algorithm that implements each stack is the same, even though the data being stored differs.
- In a non–oop language, you would be required to create 3 different sets of stack routines, each set using different names.
- In polymorphism, in Java you can specify a general set of stack routines that all share the same names.
- More generally, the concept of polymorphism is often expressed by the phrase “one interface, multiple methods.”

- This means that it is possible to design a generic interface to a group of related activities.
- This helps reduce complexity by allowing the same interface to be used to specify a *general class of action.*
- It is the compiler's job to select the *specific action* (that is, method) as it applies to each situation.
- The programmer do not need to make this selection manually. Which needs only remember & utilize the general interface.
- Polymorphism allows 1 interface to be used for a set of actions i.e. one name may refer to different functionality.
- Polymorphism allows an object to accept different requests of a client (it then properly interprets the request like choosing appropriate method) and responds according to the current state of the runtime system, all without bothering the user.

- **There are two types of polymorphism**

1. **Compile-time polymorphism** 2. **Runtime Polymorphism**

- In **compile time** Polymorphism, method to be invoked is determined at compile time.
- Compile time polymorphism is supported through the **method overloading** concept in java.
- Method overloading having multiple methods with same name but with different signature (number, type and order of parameters).

```
class A
{
    public  void fun1(int x)
{
   System.out.println("The value of class A is : " + x);
    }
    public void fun1(int x, int y)
{
   System.out.println("The value of class A is : " + x + " and " + y);
    }
   }
   public class polyone{
    public static void main(String[] args)
{
A obj=new A();//Here compiler decides that fun1(int) is to be called and "int" will be printed.
obj.fun1(2);//Here compiler decides that fun1(int, int)is to be called and "int and int" will be printed
        obj.fun1(2,3);
    }
}
```

- In **rumtime** polymorphism, the method to be invoked is determined at the run time.
- The example of run time polymorphism is **method overriding**.
- When a subclass contains a method with the same name and signature as in the super class then it is called as method overriding.

```
class A
{
    public void fun1(int x)
{   System.out.println("int in Class A is : "+ x);
    }
  }
  class B extends A
{
    public void fun1(int x)
{   System.out.println("int in Class B is : "+ x);
    }
  }
  public class polytwo
{
    public static void main(String[] args)
{
      A obj;
      obj= new A(); // line 1
      obj.fun1(2);  // line 2 (prints "int in Class A is : 2")
      obj=new B();  // line 3
      obj.fun1(5);  // line 4 (prints ""int in Class B is : 5")
 }}
```

- **What is Java, History of Java, Features of Java**
- Java was conceived by James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan at Sun Microsystems, Inc. in 1991.
- It took 18 months to develop the first working version.
- This language was initially called “Oak” but was renamed “Java” in 1995.
- Between the initial implementation of Oak in the fall of 1992 and the public announcement of Java in the spring of 1995, many more people contributed to the design and evolution of the language.
- Bill Joy, Arthur van Hoff, Jonathan Payne, Frank Yellin, and Tim Lindholm were key contributors to the maturing of the original prototype.
- Around 1990 James Gosling , Bill Joy and others at Sun Microsystems began developing a language called *Oak*.
- They wanted it primarily to control microprocessors embedded in consumer items such as cable set-top boxes,VCR's, toasters, and also for personal data assistants (PDA).
- To serve these goals, Oak needed to be:
  - **Platform independent (since multiple manufacturers involved).**
  - **Extremely reliable.**
  - **Compact.**

- However, as of 1993, interactive TV and PDA markets had failed to take off.
- Then the Internet and Web explosion began, so Sun shifted the target market to Internet applications and changed the name of the project to Java.
- By 1994 Sun's *HotJava* browser appeared. Written in Java in only a few months, it illustrated the power of *applets*, programs that run within a browser, and also the capabilities of Java for speeding program development.
- Riding along with the explosion of interest and publicity in the Internet, Java quickly received widespread recognition and expectations grew for it to become the dominant software for browser and consumer applications.
- However, the early versions of Java did not possess the breadth and depth of capabilities needed for client (i.e. consumer) applications. For example, the graphics in Java 1.0 seemed crude and clumsy compared to mature software developed with C and other languages.
- Applets became popular and remain common but don't dominate interactive or multimedia displays on web pages. Many other "plug-in" types of programs also run within the browser environment.
- So Java has not succeeded at development of consumer applications. However, Java's capabilities grew with the release of new and expanded versions (see below) and it became a very popular language for development of *enterprise*, or *middleware*, applications such as on line web stores, transactions processing, database interfaces, and so forth.

- Java has also become quite common on small platforms such as cell phones and PDAs. Java is now used in several hundred cell phone models.
- Over 600 million JavaCards, smart cards with additional features provided by Java, have been sold as of the summer of 2004.
- The concept of Write-once-run-anywhere (known as the Platform independent) is one of the important key feature of java language that makes java as the most powerful language.
- **Object Oriented(** Inheritance: Encapsulation: Polymorphism:)
- **Dynamic binding**: Sometimes we don't have the knowledge of objects about their specific types while writing our code.
- It is the way of providing the maximum functionality to a program about the specific type at runtime.
- As the languages like Objective C, C++ fulfills the above four characteristics yet they are not fully object oriented languages because they are structured as well as object oriented languages. But in case of java, it is a fully Object Oriented language because object is at the outer most level of data structure in java.
- No stand alone methods, constants, and variables are there in java. Everything in java is object even the primitive data types can also be converted into object by using the wrapper class.

Robust

- Java has the strong memory allocation and automatic garbage collection mechanism.
- It provides the powerful exception handling and type checking mechanism as compare to other programming languages. Compiler checks the program whether there any error and interpreter checks any run time error and makes the system secure from crash. All of the above features makes the java language robust.

Distributed

- The widely used protocols like HTTP and FTP are developed in java.
- Internet programmers can call functions on these protocols and can get access the files from any remote machine on the internet rather than writing codes on their local system.

Portable

- The feature Write-once-run-anywhere makes the java language portable provided that the system must have interpreter for the JVM.
- Java also has the standard data size irrespective of operating system or the processor.
- These features make the java as a portable language.

Dynamic

- while executing the java program the user can get the required files dynamically from a local drive or from a computer thousands of miles away from the user just by connecting with the Internet.

Secure

- Java does not use memory pointers explicitly.
- All the programs in java are run under an area known as the sand box.
- Security manager determines the accessibility options of a class like reading and writing a file to the local disk.
- Java uses the public key encryption system to allow the java applications to transmit over the internet in the secure encrypted form.
- The byte code Verifier checks the classes after loading.

Multithreaded

- Java is also a Multithreaded programming language.
- Multithreading means a single program having different threads executing independently at the same time. Multiple threads execute instructions according to the program code in a process or a program. Multithreading works the similar way as multiple processes run on one computer.
- Multithreading programming is a very interesting concept in Java.
- In multithreaded programs not even a single thread disturbs the execution of other thread.

- Threads are obtained from the pool of available ready to run threads and they run on the system CPUs. This is how Multithreading works in Java which you will soon come to know in details in later chapters.

Interpreted

We all know that Java is an interpreted language as well. With an interpreted language such as Java, programs run directly from the source code.
The interpreter program reads the source code and translates it on the fly into computations.

- Java as an interpreted language depends on an interpreter program.
  The versatility of being **platform independent** makes Java to outshine from other languages.
- The source code to be written and distributed is platform independent.
  Another advantage of Java as an interpreted language is its error debugging quality.
- Due to this any error occurring in the program gets traced. This is how it is different to work with Java.

Performance

- Java uses native code usage, and lightweight process called threads.
- In the beginning interpretation of byte code resulted the performance slow but the advance version of JVM uses the adaptive and just in time compilation technique that improves the performance.

Architecture Neutral

The term architectural neutral seems to be weird, but yes Java is an architectural neutral language as well. The growing popularity of networks makes developers think distributed. In the world of network it is essential that the applications must be able to migrate easily to different computer systems. Not only to computer systems but to a wide variety of hardware architecture and Operating system architectures as well. The Java compiler does this by generating byte code instructions, to be easily interpreted on any machine and to be easily translated into native machine code on the fly. The compiler generates an architecture-neutral object file format to enable a Java application to execute anywhere on the network and then the compiled code is executed on many processors, given the presence of the Java runtime system. Hence Java was designed to support applications on network.

Comments, White Spaces, Identifiers,

Comments

- A program can be documented by inserting comments at relevant places.
- These comments are for documentation purposes and are ignored by the compiler.
- Java provides three types of comments to document a program:
- A single-line comment: // … to the end of the line
- A multiple-line comment: /* … */
- A documentation (Javadoc) comment: /** … */

Single-line Comment

- All characters after the comment-start sequence
- // through to the end of the line constitute a single-line comment.
- // This comment ends at the end of this line.int age; // From comment-start sequence to the end of the line is a comment.

Multiple-line Comment

- A multiple-line comment, as the name suggests, can span several lines.
- Such a comment starts with /* and ends with */.
- /* A comment on several lines.*/

- The comment-start sequences (//, /*, /**) are not treated differently from other characters when occurring within comments, and are thus ignored. This means trying to nest multiple-line comments will result in compile time error:
- /* Formula for alchemy. gold = wizard.makeGold(stone); /* But it only works on Sundays. */*/ The second occurrence of the comment-start sequence /* is ignored. The last occurrence of the sequence */ in the code is now unmatched, resulting in a syntax error.

Documentation Comment

- A documentation comment is a special-purpose comment that when placed before class or class member declarations can be extracted and used by the javadoc tool to generate HTML documentation for the program.
- Documentation comments are usually placed in front of classes, interfaces, methods and field definitions. Groups of special tags can be used inside a documentation comment to provide more specific information. Such a comment starts with /** and ends with */:
- /** * This class implements a gizmo. * @author K.A.M. * @version 2.0 */ For details on the javadoc tool, see the documentation for the tools in the Java 2 SDK.

Home Work

- class **Computer** with **setOn**(), **getWindow**() methods and **power** of 220 as field.
- class **Motorcycle** with **setkeyOn**(), **getKick**(), **leaveCluch**(), **adjustGear**() and **brake**() methods and **petrol** and **battery** as fields
- class **Traveller** with **getTicket**(),**findBogee**(), **travel**() methods and **money** as integer field.
- class **Patient** with **getAppointment**(), **getMedicine**(), **getTests**() as methods and **amount** as field.

INHERITANCE

- class **Animal** with **Dog**, **Cat**, **Horse**, **Lion**, **Elephant** be sub classes of **Animal** while **BabyDog**, **BabyCat**, **BabyHorse**, **BabyLoin**, **BabyElephant** as subclasses of respective classes and show their behavior by **behave**() method related to each parent and child class with method overriding example.
- class **Max** with **getMax**() method for knowing max value from possible argument values values such as, 2,3 or 3,2 or 2.0, 1 or 1,2.0 or 1.0,2.0 etc to implement method overloading example.