OOM: Concepts & Implementation

Dr.Ranjana Vyas

IIIT-A

OOM: Concepts & Implementation

Procedure Oriented Vs Object Oriented

- **Class & Objects in Java: Inheritance**
  - Inheritance: Method Overloading & Overriding
  - **Abstraction: Intro to Abstract Classes & Methods**
- **OO Design: Simulation S/W Case study**
- **OOM & Java Implementation**
  - **Polymorphism & Abstraction : Java API**
- **GUI in Java & Event Handling**
  - **GUI, Applets & Sandbox Security**
- **Software Project Issues: DBMS & OOM**
  - **Animation, Sound & Connectivity Issues**

Why do you think that Java is really so popular compared with C++ or other languages also following same OO characteristics….??

OOM & Java

- Object-oriented programming developed as the dominant programming methodology during the mid-1980s, dominance was further cemented by the rising popularity of graphical user interfaces, for which object-oriented programming is well-suited.
- In the past decade Java has emerged in wide use partially because of its similarity to C++, but perhaps more importantly because of its implementation using a virtual machine that is intended to *run code unchanged on many different* platforms.
  - Some feel that association with GUIs (real or perceived) was what propelled OOP into the programming mainstream.
  - OOP toolkits also enhanced the popularity of "event-driven programming".

Working of an Object

- Typically, a Java program **creates many objects** from a variety of classes.
- These **objects interact** with one another by sending each other messages.
- Through these object interactions, a Java program can
  - implement a GUI (*using Frame* etc.),
  - run an animation (*using Applet*),
  - or send and receive information over a network (*using Socket Programming*).
- Once an object has completed the work for which it was created, it is garbage-collected and its resources are recycled for use by other objects.

OOPS Characteristics

In OO environment, various Classes exhibit some special `Classes…..

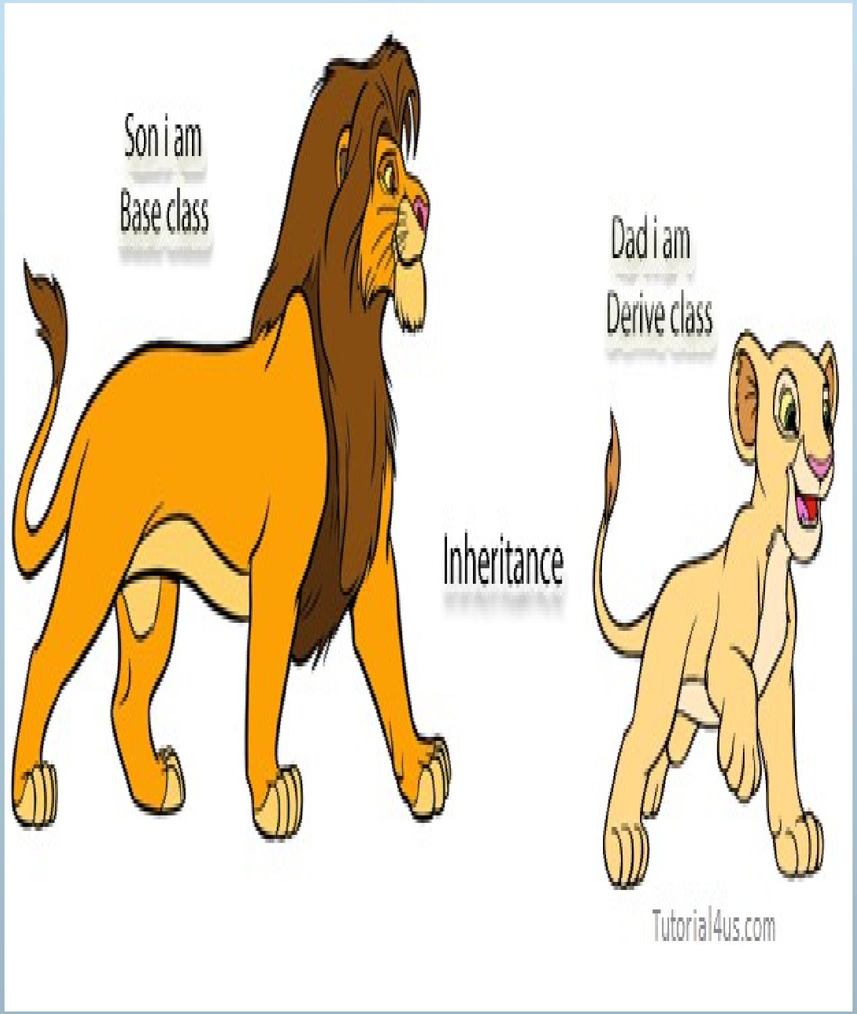
Writing Programs for Classes so identified with relationships known… is rather easier….

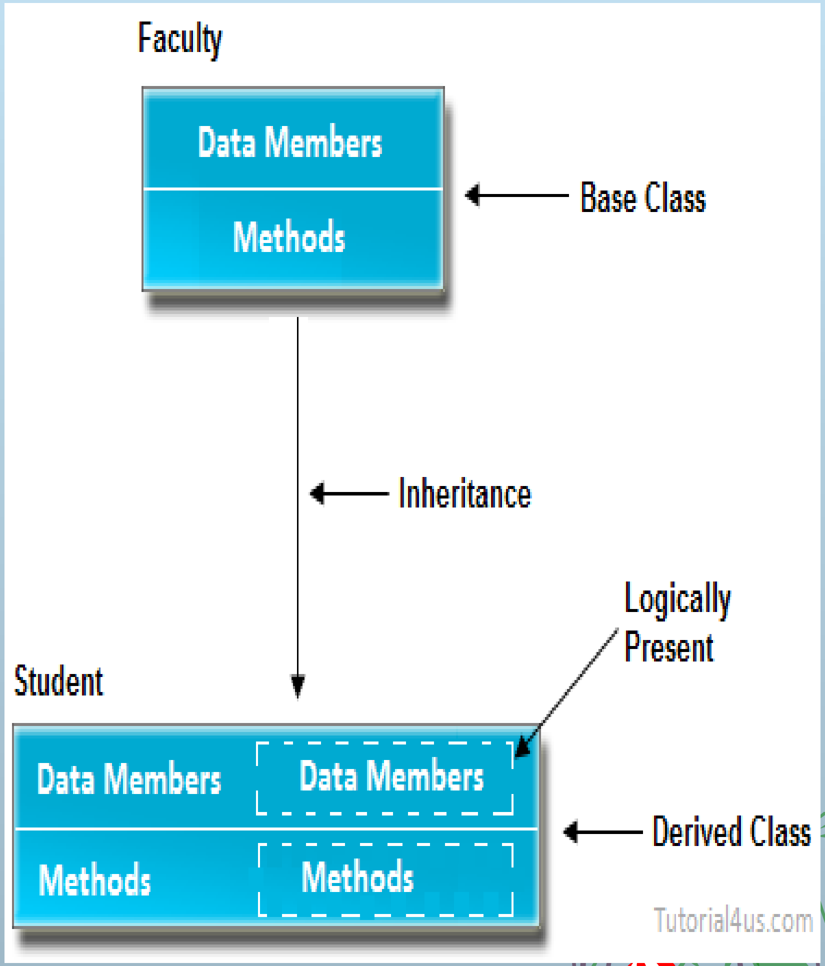
Inheritance is one of the three basic principles of OO Programming..

Inheritance brings many more useful programming constructs in OOPS

Real life example of inheritance

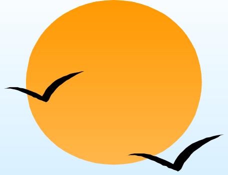
The real life example of **inheritance** is **child** and **parents**, all the properties of father are inherited by his son.





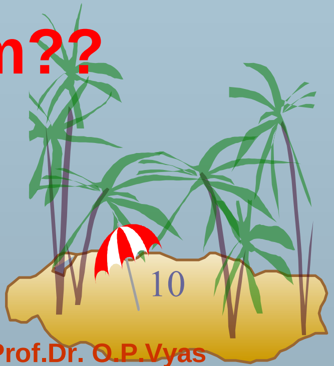
Advantage of inheritance

- If we develop any application using concept of Inheritance than that application have following advantages,
- Application development time is less.
- Application take less memory.
- Application execution time is less.
- Application performance is enhance (improved).
- Redundancy (repetition) of the code is reduced or minimized so that we get consistence results and less storage cost.



OOM: Class Relationships

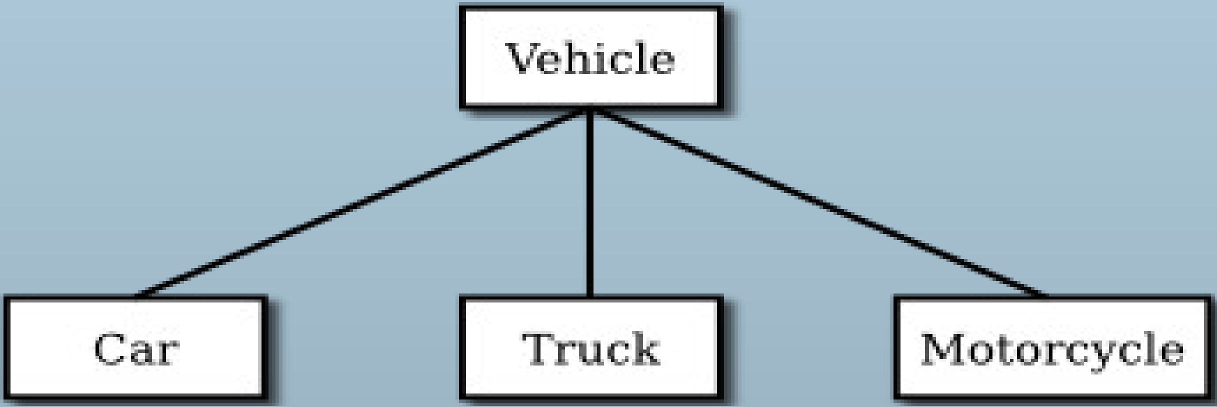
- **Suppose that a program has to deal with motor vehicles, including cars, trucks, and motorcycles.** (This might be a program used by a Department of Motor Vehicles to keep track of registrations**.)**
  - **The program could use a class named *Vehicle* to represent all types of vehicles.**
  - **What could be Attributes and Methods for Class Vehicle**
    - **Do you see any relationship between Cars, Trucks, and Motorcycles ??**
    - **Is there something common between them??**



The ***Vehicle*** class would include
Attributes such
as **registrationNumber** and **owner**
and Methods such
as **transferOfwnership().**

Class Relationships

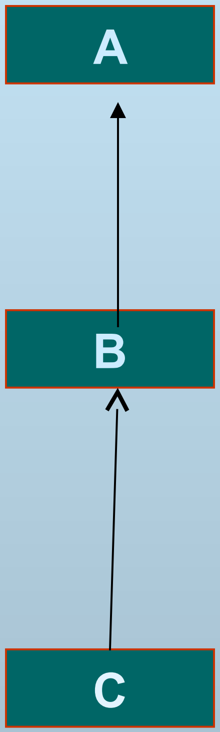
- The Class **Vehicle**, including **cars, trucks, and motorcycles** all will have these Attributes and Methods common in them.
  - Because Every Vehicle will have **Registration Number** and Owner as common attributes and **transferOfwnership()** as common methods.
  - The program could use a class relationship **Inheritance**, so that common Attributes and Methods can be re-used in all these Classes. Since cars, trucks, and motorcycles are types of vehicles.
  - Class ***Vehicle*** can be called **Parent Class** and **Car, Truck Motorcycle** can be Child Classes.



Method Overriding & Super

Consider following :

1) Class A is the **Super class** and B extends A
2) Class C extends B
3) Suppose there is method "**display**" in A which has been **over-ridden** in all the classes.
4) From the method in class C, we wish to call that method of B & A.
5) We are able to call that (method) of B using **super**.



Question: Can we call A's display method from C's display method directly???

OOM

- The Answer is…..
- ✓ No ! Java doesn't allow it.
- ✓ Why would you like to write a Method in Class A , which needs to be over-ridden in all classes B & C !! The whole desire to call a "super's super method" is an indication that this is not a good design.
- ✓ You should rethink your design !!

Directly accessing Grandparent’s member in Java

```
// filename Main.java
ν  class Grandparent {
    public void Print() {
       System.out.println("Grandparent's Print()")
    } }
  class Parent extends Grandparent {
    public void Print() {
       System.out.println("Parent's Print()");
    }}
 class Child extends Parent {
    public void Print() {
       super.super.Print();  // Trying to access
     Grandparent's Print()
       System.out.println("Child's Print()");
    }}
```

```
public class Main
{
public static void
main(String[] args)
{
Child c = new Child();
c.Print();
}
}
```

What will be Output ??

Inheritance

- **The Output is…..**
- ✓ **Compilation Error !**
- ✓ There is error in line "super.super.print();".
- ✓ In Java, a class cannot directly access the grandparent's members. It is allowed in C++ though.
- ✓ In C++, we can use scope resolution operator (::) to access any ancestor's member in inheritance hierarchy **!!**
- ✓ ***In Java, we can access grandparent's members only through the parent class!!***

For example, study the following program

```
//filename Main.java
class Grandparent {
    public void Print() {
        System.out.println("Grandparent's Print()");
    } }
  class Parent extends Grandparent {
    public void Print() {
        super.Print();
        System.out.println("Parent's Print()");
    } }
class Child extends Parent {
    public void Print() {
        super.Print();
        System.out.println("Child's Print()");
    }
}
```

```
public class Main
{
  public static void
main(String[] args) {
   Child c = new Child();
        c.Print();
      }
}
```

What will be Output ??

Output:

```
Grandparent's Print()
Parent's Print()
Child's Print()
```

- **Polymorphism**

Polymorphism & Reference variables

- The reference variable has significant characteristics and thus a role to support polymorphic activities.
- The reference variable can be reassigned to other objects provided that it is not declared final.
- The type of the reference variable would determine the methods that it can invoke on the object.
- A reference variable can refer to any object of its declared type or any subtype of its declared type. A reference variable can be declared as a class or interface type.

Instantiating an Object: Overview

- In Java, the new keyword is used to instantiate an object. The new operator creates the object in memory and returns a reference to the newly created object.
- This object will remain in memory as long as your program retains a reference to the object.
- The following statements declare an Employee reference and use the new keyword to assign the reference to a new Employee object. Employee e = new Employee();
- While instantiating objects it emphasizes the important fact that two entities are being created in memory: the reference and the object.
- The reference e is declared as a reference to an Employee, meaning that e can refer to any Employee object.
- The reference e is *not* an object.
- The object itself does not have a variable name, and the only way you can access and use the object is to use a reference to the object.

References:Overview

- A reference is a 32-bit integer value that contains the memory address of the object it refers to.
- While references are essentially integers, still they need to be declared as a particular data type.
- This is because data types are strictly enforced in Java.
- A reference must be of a particular class data type.
- For example, in the following statements, two Employee references and one String reference are allocated in memory.

Employee e1, e2;

String s;

- Each of these three references consumes the same amount of memory and is essentially an integer data type.
- However, the references e1 and e2 can refer only to Employee objects.
- The reference s can refer only to a String object.

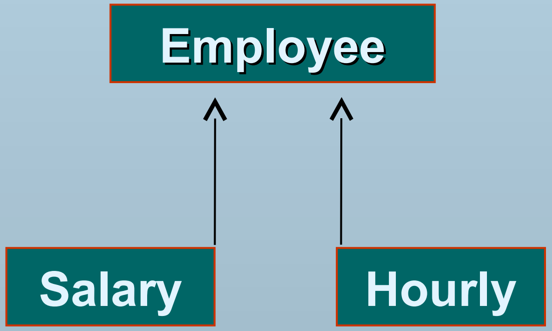
Polymorphism: Parent Class reference

- To demonstrate polymorphism in action, we will use the following **Employee** and **Salary** classes,
- Notice that the Employee class has
  - **three fields: name, address, and number;**
  - **one constructor;**
  - **a mailCheck() method; the toString() method;and various accessor methods.**
  - **There are no fields in the Employee class used to represent the employee's pay because we decided that this data should appear in the child classes.**

Polymorphism in Java

Suppose there is simple Class Inheritance "Employee“ with Child Classes Hourly and Salary , we want to create objects now.,

How will be objects instantiated ??



The Employee Class

Private Members

String **name;**

String **address;**

int **number**;

Constructor

Employee
(name,address,number)

Methods

mailCheck()

toString()

getName()

getAddress( )

setAddress()

getNumber()

The Employee class extends Object implicitly.

```
public class Employee
{
private String name;
private String address;
private int number;
public Employee(String name, String
    address, int number)
{
System.out.println("Constructing an
    Employee");
this.name = name;
this.address = address;
this.number = number;
}
```

The Employee class extends Object implicitly

```
public void mailCheck()
{
System.out.println("Mailing a check to " +
this.name + " " + this.address);
}
public String toString()
{
return name + " " + address + " " + number;
}
public String getName()
{
return name;
}
public String getAddress()
{
return address;
}
public void setAddress(String newAddress)
{
address = newAddress;
}
public int getNumber()
{
return number;
}
}
```

The Salary Class

private members

double salary;

Constructor

Salary(name,address, number,salary)

Methods

getSalary()

setSalary()

computePay()

Indirectly Accessed Members

String name;

String address;

int number;

Super Constructor

Employee

(name,address,number)

mailCheck()

toString()

getName()

getAddress()

setAddress()

getNumber()

ExampleThe Salary class extends the Employee class.

```
public class Salary extends Employee
{
private double salary; //Annual salary
public Salary(String name, String
    address, int number, double salary)
{
super(name, address, number);
SetSalary(salary);
}
public double getSalary()
{
return salary;
}
```

Employee has a salary, so adding a salary field in the Employee class would cause issues later;

```
public void setSalary(double
    newSalary)
{
if(newSalary >= 0.0)
{
salary = newSalary;
}
}
public double computePay()
{
System.out.println("Computing salary
    pay for " + getName());
return salary/52;
}
```

child classes of Employee will contain the fields and methods needed to compute the employee's pay.

Child Class…

- The Salary class, shown in Listing, has one field named salary to represent the annual pay of an employee. The Salary class also has one constructor, a computePay() method, and various accessor and mutator methods.
- The idea behind this design is that not every **Employee has a salary, so adding a salary field in the Employee class would cause issues later**;
- therefore, the child classes of Employee will contain the fields and methods needed to compute the employee's pay.
- As the Hourly class that also extends Employee, and it will have fields for an hourly wage and number of hours worked, as well as a computePay() method.

- Now, let's look at a couple of statements. Suppose that we instantiated an Employee object as follows:

Employee e = new Employee("George W. Bush", "Houston, TX", 43);

- The previous statement creates two entities in memory:
  - **the Employee reference e, and the Employee object to which the e gets assigned.**
  - **The reference is of type Employee, and so is the object it refers to.**
- Similarly, the following statement is valid:

Salary s = new Salary("George Washington", "Valley Forge, DE", 1, 5000.00);

- In this statement, **s is a Salary reference**, and it is assigned to a new Salary object.
  - **Both the reference and the object are of type Salary.**

Polymorphism

- Now, take a look at a statement that at first glance might seem strange

Employee p = new Salary(“ Reuther”, “Rapid City, SD”, 47, 250000.00);

- The left side of the equation creates a reference p of type Employee. The right side of the equation is a new Salary object. **Can you assign an Employee reference to point to a Salary object? Are they compatible data types?**
- The answer to both questions is yes. The *is a relationship carries over to* polymorphism. The right side of the equation is a Salary object. A Salary object is an Employee object.
- Can p refer to an Employee object? Can p refer to Salary object?
- Certainly. In fact, p can refer to an Employee object, and because a Salary object is an Employee object, p can refer to a Salary object as well.

Polymorphism

- From the point of view of polymorphism, a ***Child object is an Parent object.***
- A child object being treated as a parent class type has the following benefits:

1) Using a parent class reference to a child object.
2) Using polymorphic parameters and return values.
3) Creating heterogeneous collections of objects, where not all objects in the collection are of the same type.

- As we know that Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.
- This means a Salary object can be treated as an Employee object.

Using Parent Class References to Child Objects

- A child object can be referenced using a parent class reference.

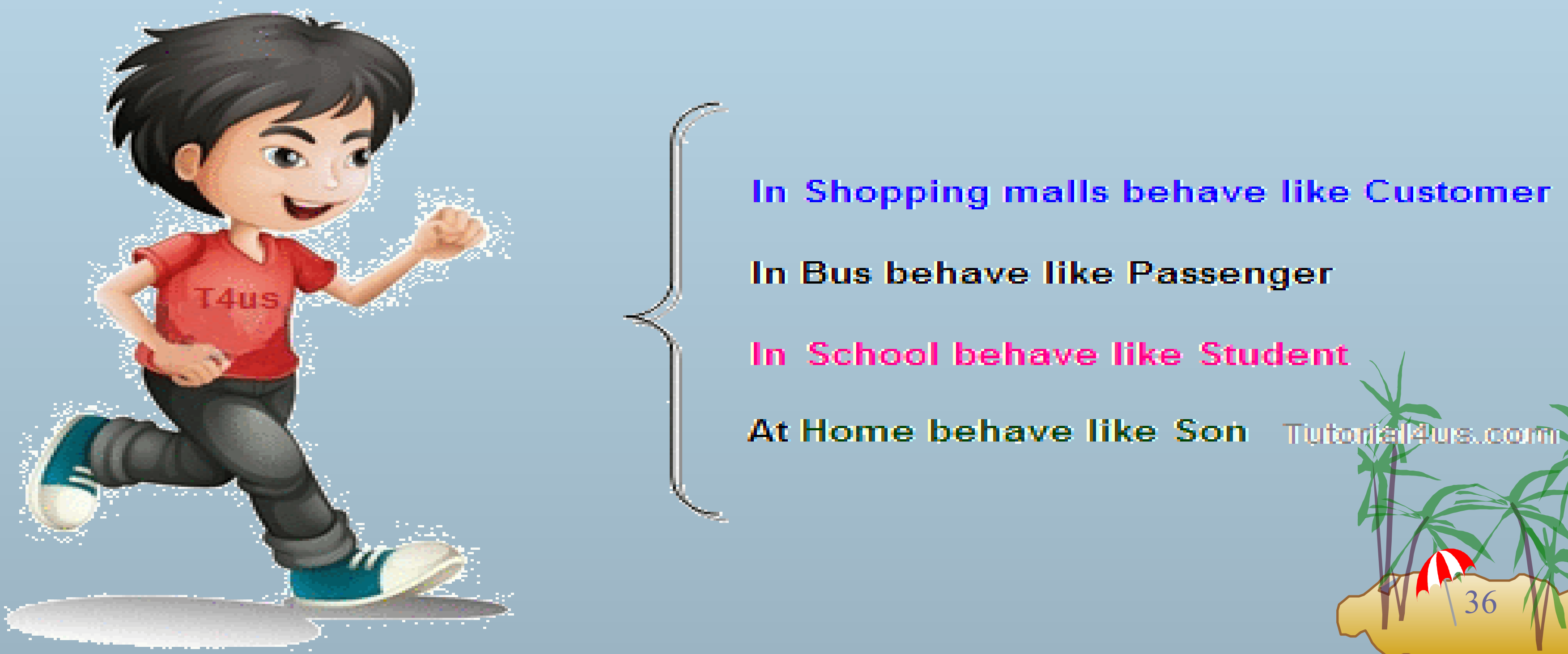
- This is the most fundamental use of polymorphism because using a parent class reference to refer to a child object is what allows you to take advantage of the benefits of polymorphism.

Let us understand the Polymorphism… in detail.

The term *Polymorphism* (*Poly*, *meaning many or* multiple, and *Morph*, *meaning shapes or forms*) *in OOP* refers to the ability of an object to have many forms, which is a direct result of inheritance.

Real life example of polymorphism

- Suppose if you are in **class room** that time you behave like a **student**,
  - when you are in market at that time you behave like a customer,
  - when you at **your home** at that time you behave like a **son or daughter**, **Here one person present in different-different behaviors.**

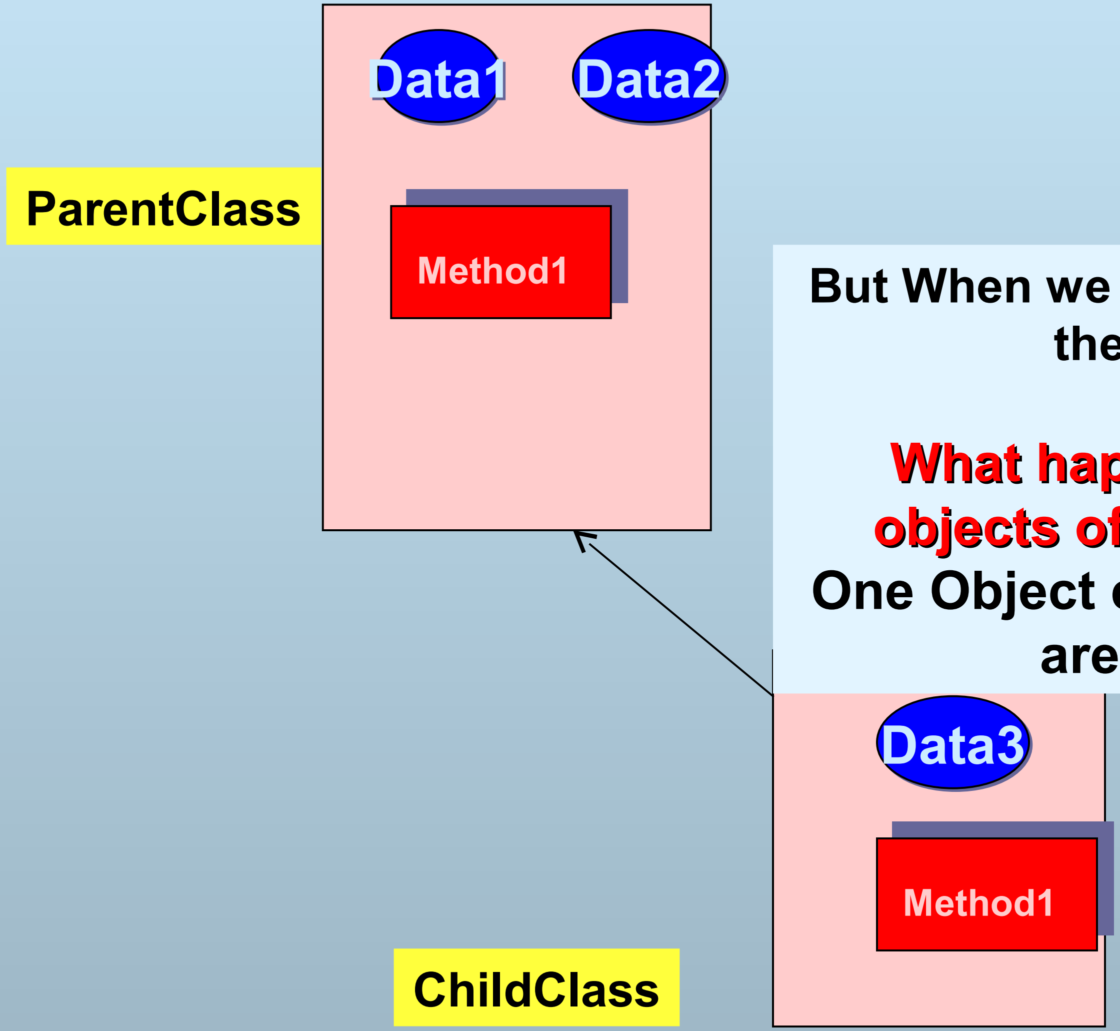

T4us
In Shopping malls behave like Customer
In Bus behave like Passenger
In School behave like Student
At Home behave like Son
Tutorial4us.com

Understanding the Polymorphism

- Polymorphism: Introduction
  - What is Polymorphism & Why ?
  - Method overriding & Method over loading
- Class references to Child Objects ?
  - Instantiation of Objects: Overview
  - Understanding the references
  - Polymorphic parameters
  - Casting references
  - Instanceof Keyword
- Benefits of Polymorphism
  - Extensibilty
  - Heterogeneous Collection

Polymorphism: Inheritance in Java

A Class inheriting Parent Class gets both data and methods (or Functions) of Parent Class and can add new Data3 for Child Class .

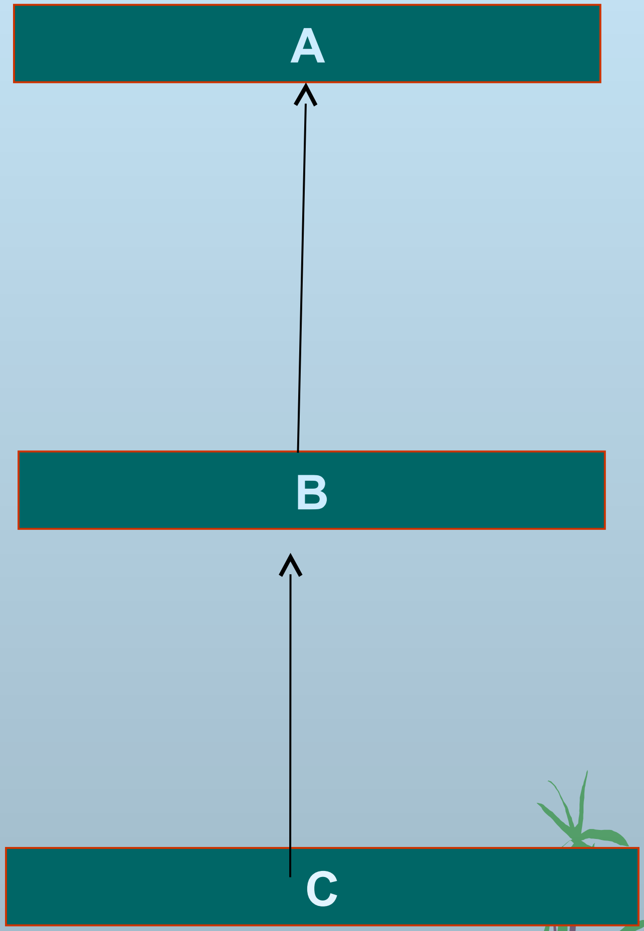


But When we want to instantiate objects from these two classes then

What happens if we instantiate the objects of Child type or Parent type
One Object can have many forms & what are the possibilities??

Objects instantiation in Multi level Inheritance

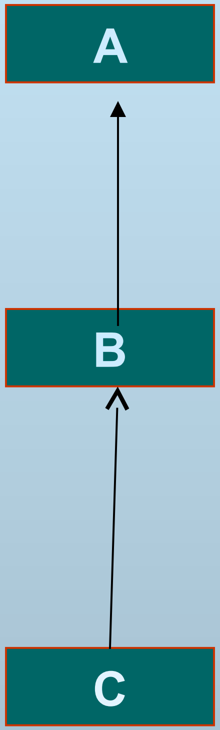
- Consider a situation when we have **Multi level Inheritance** and Object instantiation;
  - **Objects of Class A with reference type A.**
  - **Objects of Class B with reference type B and also of type A.**
  - **Objects of Class C with reference type C and/or of type B , type A.**
- ***In OOP* the ability of an object to have many forms, which is a direct result of inheritance is known as Polymorphism.**



Object Instantiation in Inheritance

Consider following :

1) Can Class A instantiates the object of type B ?
2) Can Class B instantiates the object of type A ? Or of type C ??
3) Can Class C instantiates the object of type A or type B or type C?


$$\begin{tikzcd} \boxed{A} \\ \boxed{B} \arrow[u] \\ \boxed{C} \arrow[u] \end{tikzcd}$$

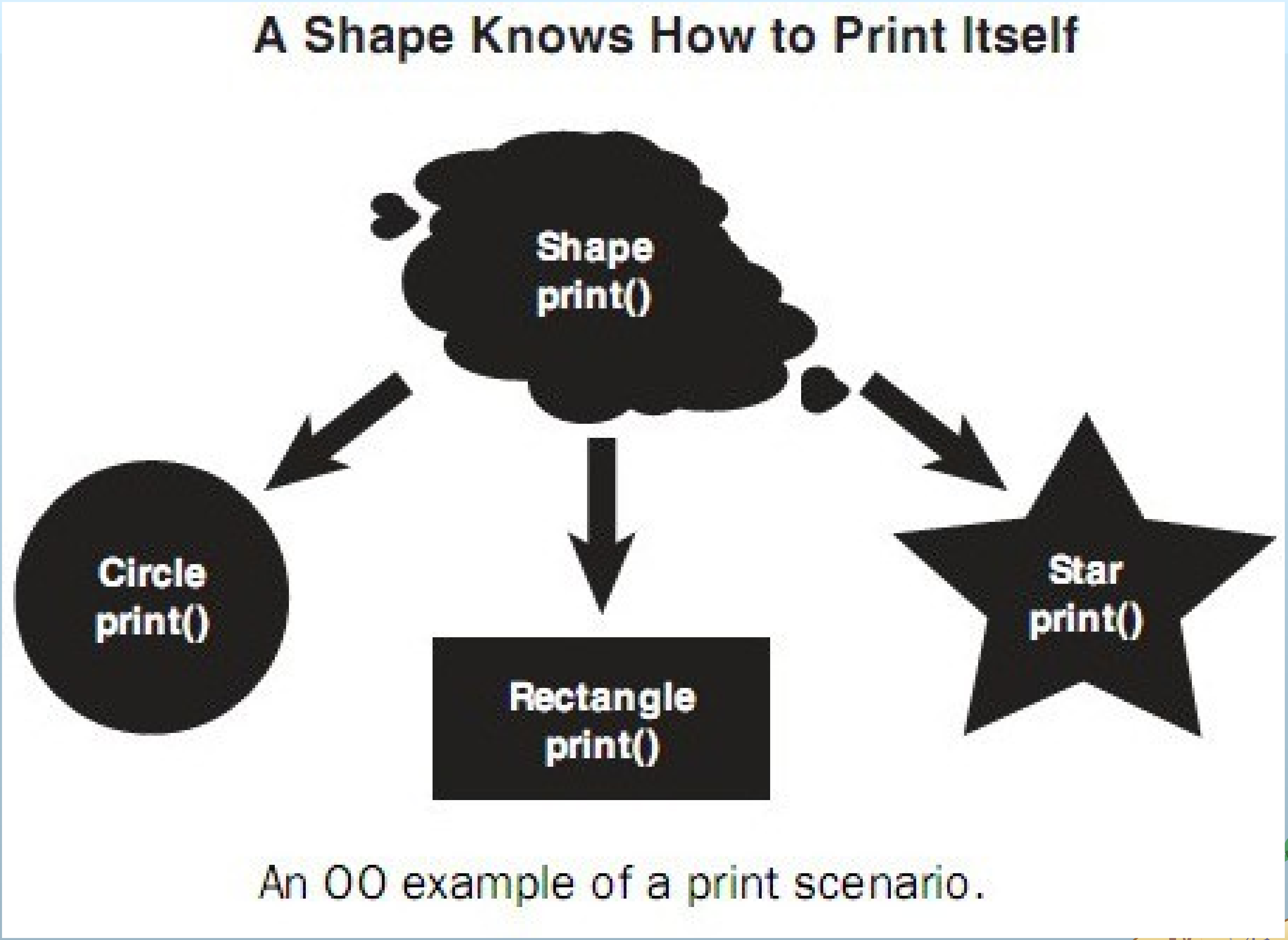
Polymorphism

- An object takes on many forms because an object can be treated as a child type, a parent type, a grandparent type, and so on up the hierarchy tree.
- When we discussed inheritance, we discussed how the *is a relationship is used to* determine if your inheritance is a good design.
  - The *is a relationship is also helpful* when learning polymorphism.
  - With Polymorphism, we can add new classes with little or no modification in general portions of the program, as long as the new classes are part of the inheritance hierarchy that the program processes generically.

Polymorphism: Methods & Objects

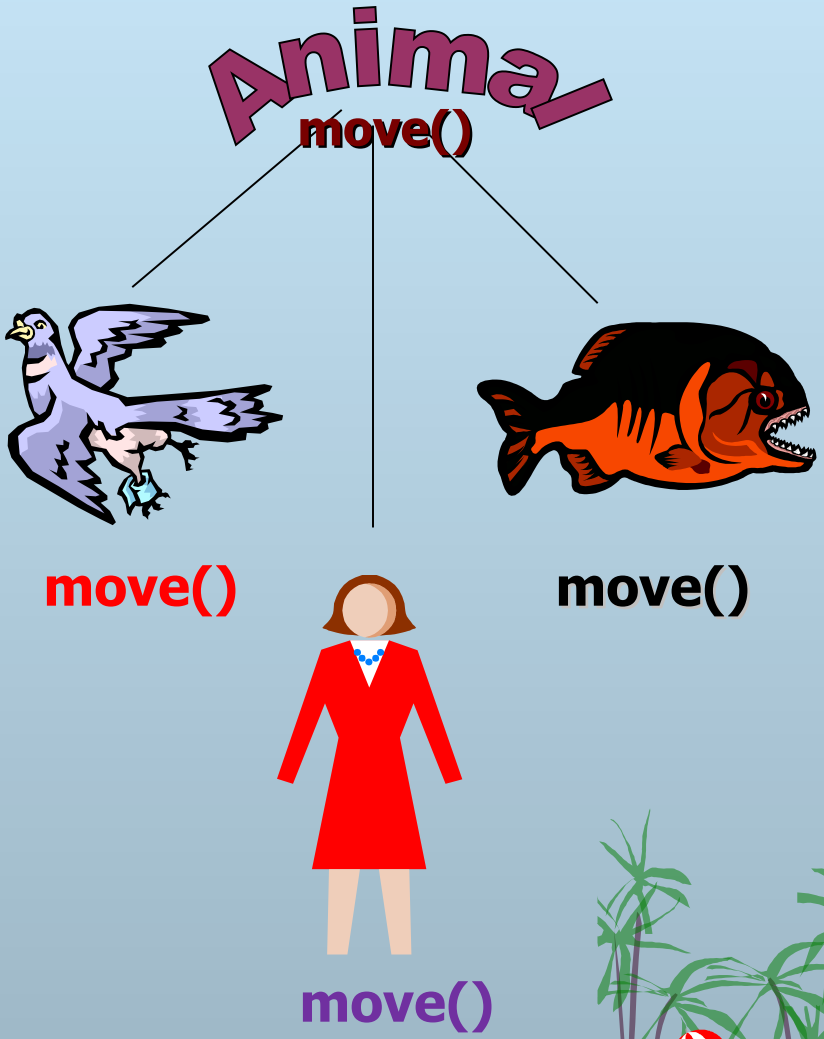
- ***Polymorphism*** **is the capability of an action or *method* to do different things based on the object that it is acting upon. Overloading and overriding are two types of polymorphism .**
- **We say a method is polymorphic if the action performed by the method depends on the actual type of the object to which the method is applied.**
- **Generally, it allows us to mix** methods **and** objects **of** different types **in a consistent way**.

Polymorphism


A Shape Knows How to Print Itself
Shape print()
Circle print()
Rectangle print()
Star print()
An OO example of a print scenario.

Polymorphism

- Ex. Bird, Person and Fish are child classes of Animal.
- Suppose we wish to create a program **that simulates the movement of several types of animals.**
- Each subclass Bird, Fish and Person extends the Superclass Animal, which contain a method move() maintains an animal's current location as x-y coordinates.
- **Each subclass implements method move & overrides** the **move()** method in its own way



Polymorphism for Simulation program

- We design a program which maintains the **array of references to objects** of the various Animal subclass.
  - To simulate the animal's movements, the program sends each object the same message once per second move().
- However each specific type of Animal responds to a move message in a unique way:
  - **A Fish swim three feet,**
  - **A Bird might fly ten feet**
  - **A Person may walk five feet.**
- Each Object know how to modify its x-y coordinates appropriately for its specific types of movements.
- Capability of each object to "do the right thing" in response to the same method call is resulting in to a variety of objects having "many forms" of results thus termed as Polymorphism

Polymorphic behaviour

- Following statements are used to implement Polymorphic behaviour of various Objects

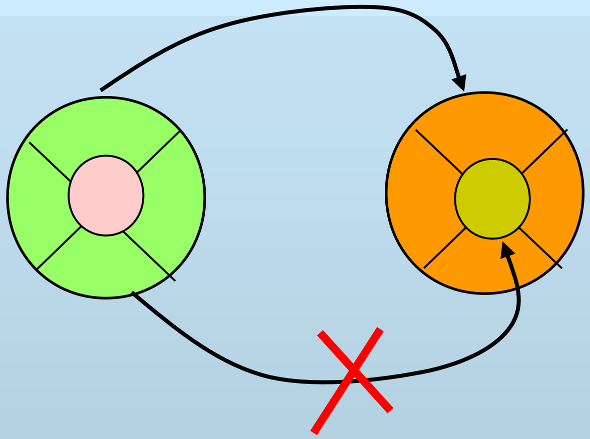
```
Animal [] A = new Animal[3];
A[0] = new Bird();
A[1] = new Person();
A[2] = new Fish();
for (int i = 0; i < A.length; i++)
        A[i].move();
```

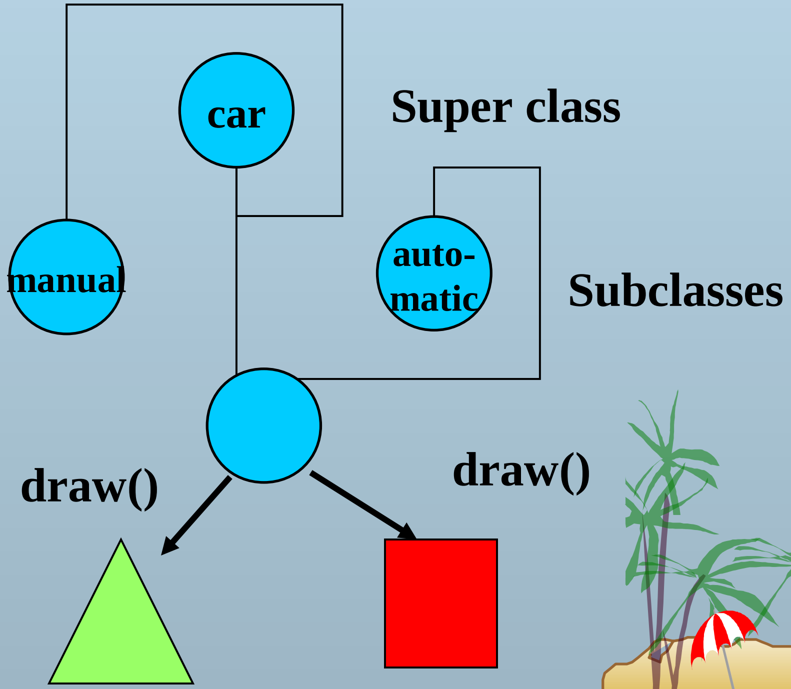
- **References are all the same, but objects are not**
- **Method invoked is that associated with the OBJECT, NOT with the reference**

The 03 principles of OOM works together

- Inheritance , Encapsulation and Polymorphism when properly applied it can produce a programming environment that supports the development of far more robust & scalable programs than does the process-oriented model.

- A well designed hierarchy of classes is the basis for re using the code in which lot of time and expertise was invested in developing & testing.





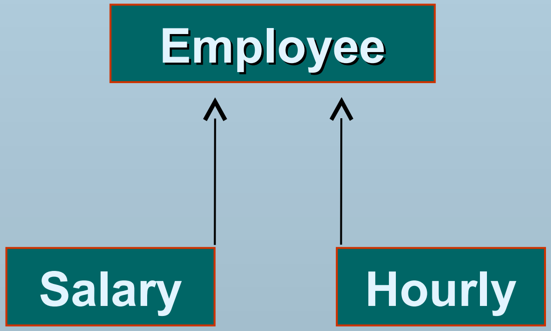
Polymorphism…'IS A' relation

- Any **java object** that can pass more than one **IS-A** test is considered to be polymorphic.
- In Java, all java objects are **polymorphic** since any object will pass the IS-A test for their own type and for the class **Object**.
- It is important to know that the only possible way to access an object is through a reference variable.
- A **reference variable can be of only one type**. Once declared the type of a reference variable cannot be changed.

Polymorphism in Employee class

Let us again study simple Class Inheritance "Employee“ with Child Classes Hourly and Salary , we need to understand the creation of objects with difference references now.,

How will be objects instantiated ??



The Employee Class

Private Members

String name;

String address;

int number;

Constructor

Employee (name,address,number)

Methods

mailCheck()

toString()

getName()

getAddress()

setAddress()

getNumber()

The Salary Class

private members

- double salary;

Constructor

- Salary(name,address, number,salary)

Methods

- getSalary()
- setSalary()
- computePay()

Indirectly Accessed Members

- String name;
- String address;
- int number;

Super Constructor

- Employee (name,address,number)

- mailCheck()
- toString()
- getName()
- getAddress()
- setAddress( )
- getNumber( )

Polymorphism Example

Employee p = new Salary(“ Reuther”, “Rapid City, SD”, 47, 250000.00);

- **In the Object p, the unique features of the Salary Class are absent.**
- **The only attributes that are present, are the ones inherited from the Employee Class.**

Indirectly Accessed Members

String name;

String address;

int number;

SuperConstructor

Employee (name,address,number)

Inherited Methods

mailCheck()

toString()

getName()

getAddress()

setAddress()

getNumber()

Polymorphism Example

Employee p = new Salary(“ Reuther”, “Rapid City, SD”, 47, 250000.00);

- In case methods/members with same name exist in the classes, they will be overridden in this object.
- eg: if the Salary class has a separate mailCheck() method, then the case will be:

Methods

mailCheck()

Indirectly Accessed Members

String name;

String address;

int number;

SuperConstructor

Employee (name,address,number)

Inherited Methods

toString()

getName()

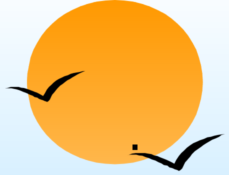
getAddress()

setAddress()

getNumber()

Polymorphism

- Why use an Employee reference to refer to a Salary object? Whynot just use a Salary reference?
- There are situations where using a parent class reference can make your code easier to write and easier to maintain. You can think of an Employee reference as a more generic reference than a Salary reference. Suppose that we have Hourly and Salary classes that both extend Employee. I can then reference each employee's object as either a Salary or Hourly, using a Salary or an Hourly reference. Or I can treat each employee's object as an Employee, and use an Employee reference to refer to any employee, no matter what data type the employee actually is.
- Q: But what do we gain from doing this?
- A: By treating Hourly and Salary objects as type Employee, I can store them in the same data structure (an example of a heterogeneous collection). I can also write a method that has an Employee parameter, which allows both Salary and Hourly objects to be passed in to that method (an example of polymorphic parameters).



Polymorphism: discussions

- **Q: If you treat a Salary object as an Employee, don't you lose the Salary part of the object?**
- **No. The object does not change, just the data type of the reference to it. This is an important point to understand. If I instantiate a Salary object, I get a Salary object, no matter what data type its reference is.**
- **Q: So why not always use a parent class reference if it doesn't change anything?**
- **A: Well, be careful. I said the *object does not change, but how it is*** *viewed does change. If I use an Employee reference to a Salary* object, the object does not lose any data, but I lose the ability to access those fields and methods from the Salary class using the parent class reference.
- **Q: You can never access them? Then you have lost something.**
- **A: No, you can still access them, but you have to cast the Employee** reference to a Salary reference. We need to know the casting process first. It can then be shown with examples where using a parent class reference to a child object is advantageous.

/* File name : Employee.java */

```
public class Employee
{ private String name; private String address; private int
      number;
 public Employee(String name, String address, int number)
{ System.out.println("Constructing an Employee");
this.name = name; this.address = address; this.number =
      number; }
public void mailCheck()
 { System.out.println("Mailing a check to " + this.name + "
" + this.address); }
public String toString()
 { return name + " " + address + " " + number; }
public String getName()
{ return name; }
public String getAddress() { return address; }
public void setAddress(String newAddress)
{ address = newAddress; } public int getNumber() { return
      number; }
```

Now suppose we extend Employee class as follows:

/* File name : Salary.java */

```
public class Salary extends
Employee
{ private double salary; //Annual
salary
public Salary(String name, String
address, int number, double salary)
{ super(name, address, number);
setSalary(salary); } public void
mailCheck()
{ System.out.println("Within
mailCheck of Salary class ");
System.out.println("Mailing check to
" + getName() + " with salary " +
salary); } public double getSalary()
{ return salary; } public void
setSalary(double newSalary)
{ if(newSalary >= 0.0) { salary =
newSalary; } } public double
computePay()
{ System.out.println("Computing
salary pay for " + getName()); return
salary/52; } }
```

- Now, you study the following program carefully and try to determine its output:
- **/* File name : VirtualDemo.java */**

```
public class VirtualDemo
 { public static void main(String [] args)
 { Salary s = new Salary("Mohd
    Mohtashim", "Ambehta, UP", 3,
    3600.00);
Employee e = new Salary("John Adams",
    "Boston, MA", 2, 2400.00);
System.out.println("Call mailCheck using
    Salary reference --");
 s.mailCheck();
System.out.println("\n Call mailCheck
    using Employee reference--");
e.mailCheck(); } }
```

This would produce the result as shown:

Salary reference s,
Employee reference e.

s.mailCheck() **the compiler sees mailCheck() in the Salary class at compile time,**
JVM invokes mailCheck() in the Salary class at run time.

```
Constructing an Employee
Constructing an Employee Call
mailCheck  using Salary
reference –
Within mailCheck of Salary class
 Mailing check to Mohd Mohta
shim with salary 3600.0
```

- Constructing an Employee Constructing an Employee Call mailCheck using Salary reference –
  Within mailCheck of Salary class
   Mailing check to Mohd Mohtashim with salary 3600.0

  Call mailCheck using Employee reference–

  Within mailCheck of Salary class

  Mailing check to John Adams with salary 2400.0

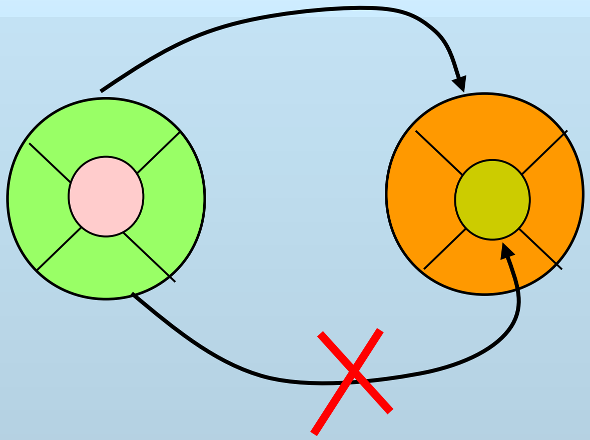
Virtual Methods

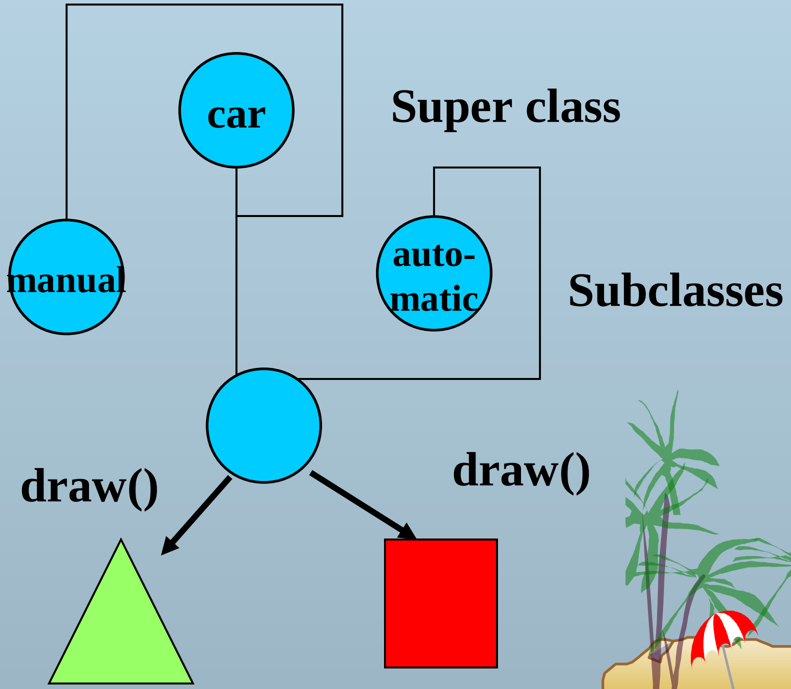
- Here, we instantiate two Salary objects . one using **a Salary reference s**, and the other using an **Employee reference e**.
- While invoking ***s.mailCheck()*** **the** compiler sees **mailCheck(**) in the **Salary class at compile time**, and the **JVM invokes mailCheck()** in the Salary class at run time.
- Invoking **mailCheck()** on e is quite different because **e** is an **Employee reference**. When the compiler sees ***e.mailCheck(),*** the compiler sees the **mailCheck()** method in the **Employee class**.
- Here, at compile time, the compiler used **mailCheck()** in Employee to validate this statement. At run time, however, the JVM invokes **mailCheck() in the Salary class.**
- This behavior is referred to as **virtual method invocation,** and the methods are referred to as virtual methods. All methods in Java behave in this manner, whereby an overridden method is invoked at run time, no matter what data type the reference is that was used in the source code at compile time.

The 03 principles of OOM works together

- Inheritance , Encapsulation and Polymorphism when properly applied it can produce a programming environment that supports the development of far more robust & scalable programs than does the process-oriented model.

- A well designed hierarchy of classes is the basis for re using the code in which lot of time and expertise was invested in developing & testing.





Polymorphism: Methods & Objects

- **We say a method is polymorphic if the action performed by the method depends on the actual type of the object to which the method is applied.**
- Generally, it allows us to mix methods and objects of different types in a consistent way.
- We have seen that with well designed inheritance hierarchy , we can use polymorphic parameters to get some interesting result, while using Parent class references for Child class objects

The Salary Class

private members

double salary;

Constructor

Salary(name,address, number,salary)

Methods

getSalary()

setSalary()

computePay()

Indirectly Accessed Members

String name;

String address;

int number;

Super Constructor

Employee (name,address,number)

mailCheck()

toString()

getName()

getAddress()

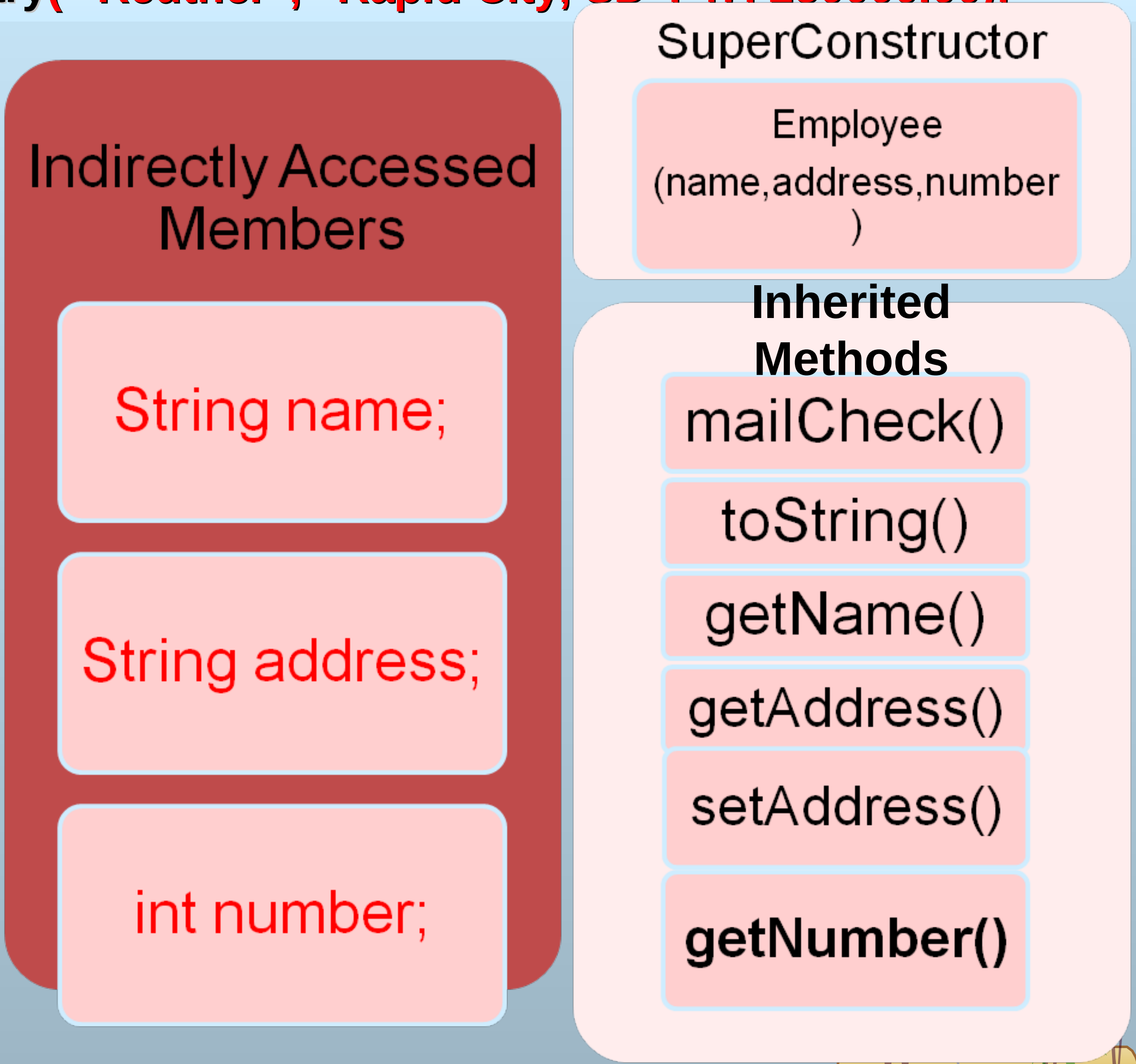
setAddress()

getNumber()

Polymorphism Example

Employee p = new Salary(" Reuther", "Rapid City, SD". 47. 250000.00):

- **In the Object p, the unique features of the Salary Class are absent.**
- **The only attributes that are present, are the ones inherited from the Employee Class.**



Polymorphism Example

Employee p = new Salary(" Reuther", "Rapid City, SD", 47, 250000.00);

- **In case methods/members with same name exist in the classes, they will be overridden in this object.**
-
- **eg: if the Salary class has a separate mailCheck() method, then ;**
  - **what the case will be ?**
  - **How this method of Salary Class gets invoked?**

Methods

mailCheck()

Indirectly Accessed Members

String name;

String address;

int number;

SuperConstructor

Employee (name,address,number)

Inherited Methods

toString()

getName()

getAddress()

setAddress()

getNumber()

Polymorphism

- Though We have treated a Salary object as an Employee, the object does not change, just the data type of the reference to it changes ! This is an important point to understand. If we instantiate a Salary object, we get a Salary object, no matter what data type its reference is.
- *We lose the ability to access those fields and methods from the Salary class using the parent class reference*. How can we access them? Have we lost something ?
- No, we can still access them, but we have to cast the Employee reference to a Salary reference. We need to know the casting process first. It can then be shown with examples where using a parent class reference to a child object is advantageous.
- By treating Hourly and Salary objects as type Employee, we can store them in the same data. I can also write a method that has an Employee parameter, which allows both Salary and Hourly objects to be passed in to that method (an example of polymorphic parameters)

Heterogeneous Collections

- We have seen that by treating Hourly and Salary objects as type Employee, we can store them in the same data structure.

- A common use of polymorphism is to create a collection of data that is not all the same type, but has a common parent. A collection of different objects is referred to as a *heterogeneous collection*.

Heterogeneous Collection

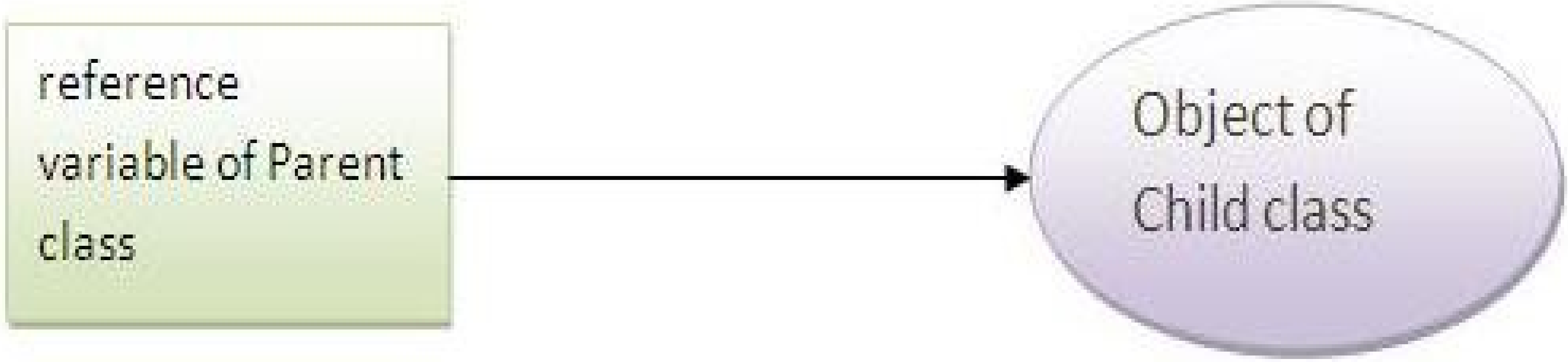
- For example, suppose that we wanted to use an array to keep track of the employees of a company. When creating an array, all the data in the array must be of the same type.
  - **Because there are two types of employees, we could create an array for Salary objects and a second array for Hourly objects.**
- Because Salary and Hourly objects are Employee objects, however, we can create a single array for Employee objects that can contain both Salary and Hourly objects.
- This is an example of a heterogeneous collection because the elements in the array will not all be the same.
- They will all be Employee objects, but some will be Salary objects and some will be Hourly objects. While Salary and Hourly have a common parent, they are different objects.

Upcasting

When reference variable of Parent class refers to the object of Child class, it is known as upcasting

class A{}

class B extends A{}



Example of Java Runtime Polymorphism

- we are creating two classes Bike and Splendar.
  - Splendar class extends Bike class and overrides its run() method. We are calling the run method by the reference variable of Parent class.
  - Since it refers to the subclass object and subclass method overrides the Parent class method, subclass method is invoked at runtime.
- Since method invocation is determined by the JVM not compiler, it is known as runtime polymorphism.

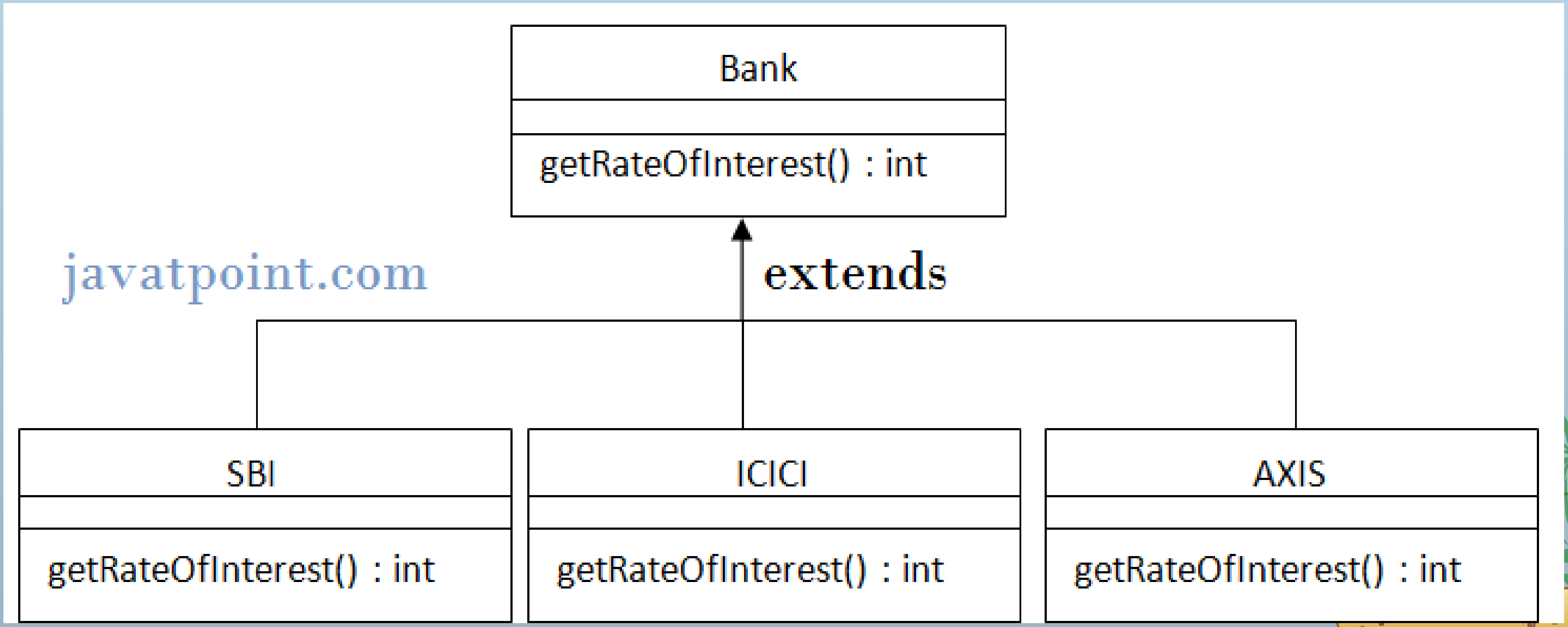
```
class Bike{
void run(){System.out.println("running");}
}
class Splender extends Bike{
void run(){System.out.println("running safely with 60km");}

 public static void main(String args[]){
  Bike b = new Splender();//upcasting
  b.run();
 }
```

Output:running safely with 60km.

Real example of Java Runtime Polymorphism

- Consider a scenario, Bank is a class that provides method to get the rate of interest. But, rate of interest may differ according to banks.
- For example, SBI, ICICI and AXIS banks could provide 8%, 7% and 9% rate of interest.


Bank
getRateOfInterest() : int
javatpoint.com
extends
SBI
getRateOfInterest() : int
ICICI
getRateOfInterest() : int
AXIS
getRateOfInterest() : int

```
class Bank{
int getRateOfInterest(){return 0;}
}

class SBI extends Bank{
int getRateOfInterest(){return 8;}
}

class ICICI extends Bank{
int getRateOfInterest(){return 7;}
}
class AXIS extends Bank{
int getRateOfInterest(){return 9;}
}

class Test3{
public static void main(String args[]){
Bank b1=new SBI();
Bank b2=new ICICI();
Bank b3=new AXIS();
System.out.println("SBI Rate of Interest: "+b1.getRateOfInterest());
System.out.println("ICICI Rate of Interest: "+b2.getRateOfInterest());
System.out.println("AXIS Rate of Interest: "+b3.getRateOfInterest());
}
}
```

Multiple inheritance …

✓ **References can be explored in multiple inheritance, Ex.:**

public interface Vegetarian{}

public class Animal{}

public class Deer extends Animal **implements** Vegetarian{}

Now the Deer class is considered to be polymorphic since this has multiple inheritance. Following are true for the above example:

- **A Deer IS-A Animal**
- **A Deer IS-A Vegetarian**
- **A Deer IS-A Deer**
- **A Deer IS-A Object**

When we apply the reference variable facts to a Deer object reference, the following declarations are legal:

- **Deer d = new Deer();**
- **Animal a = d;**
- **Vegetarian v = d; Object o = d;**

- All the reference variables d,a,v,o refer to the same **Deer object** in the heap.

Casting Process & Polymorphic behavior

- Java is strict about letting you assign variables only to values that match the variable's data type.
- If x is an int, you cannot assign it to other data types unless you use the cast operator.

- Casting process can help us in implementing many desired scenario with Inheritance, polymorphism and encapsulation .

Java Primitive data types

- byte 8 bits
- short 16 bits
- int 32 bits
- long 64 bits
- float 32 bits
- double 64 bits
- char 16 bits

Casting Process

- Java is strict about letting you assign variables only to values that match the variable's data type.
- If x is an int, you cannot assign it to other data types unless you use the cast operator.
- For example, the following statements declare x as an int and then attempt to assign x to a floating-point value.

int x;

double d = 3.5;

x = d; //This does not compile!

x = (int) d; //This does compile since the cast operator is used.

- The cast operator consists of placing the data type that the value is being cast to in parentheses. In this example, f was being cast to an int, so int was placed in parentheses right before f.

Casting References

- **Casting tells the compiler that you are aware of the invalid assignment, but you want to do it anyway. In the ex. above, x was assigned to d by casting, and x will be the value 3.**
- **Casting a floating-point number to an integer data type causes the decimal part of the number to be truncated.**

For ex., suppose that we have a double (**64 bits)** that we want to store in a float (**32 bits)** . Even if the double fits easily in the float, the compiler still requires us to use the cast operator:

```
double pi = 3.14159;
float a = pi; //Does not compile!
float b = (float) pi; //Works fine
```

- We may think the compiler should be smart enough to realize that 3.14159 fits into a float, so no casting is necessary; however, it is important to realize that the compiler only knows data types.
- When we assign a 64-bit double to a 32-bit float, the compiler only sees a larger piece of data being stored in a smaller piece. Because data could be lost, the cast operator tells the compiler you know what you are doing, and any loss of data is acceptable

Casting Process

- Java is strict about letting you assign variables only to values that match the variable's data type.
- If x is an int, you cannot assign it to other data types unless you use the cast operator.
- For example, the following statements declare x as an int and then attempt to assign x to a floating-point value.

int x;

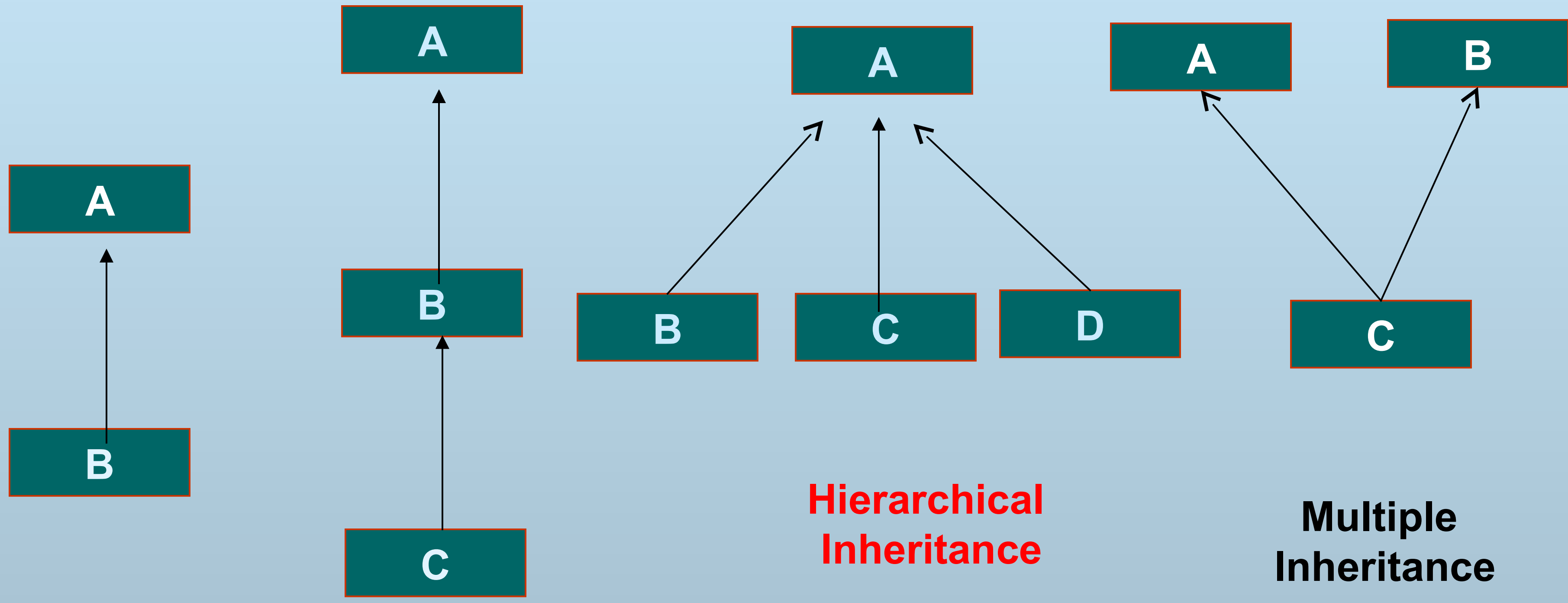
double d = 3.5;

x = d; //This does not compile!

x = (int) d; //This does compile since the cast operator is used.

- The cast operator consists of placing the data type that the value is being cast to in parentheses. In this example, f was being cast to an int, so int was placed in parentheses right before f.

Types of Inheritance & Polymorphic behavior



Single Inheritance

Multi Level Inheritance

Hierarchical Inheritance

Multiple Inheritance

Casting Process & Polymorphic behavior

- Java is strict about letting you assign variables only to values that match the variable’s data type.
- If x is an int, you cannot assign it to other data types unless you use the cast operator.
- Casting process can help us in implementing many desired scenario with Inheritance, polymorphism and encapsulation.
- We can see the benefit of polymorphism known as polymorphic parameters.
- When a method has a parameter that is a reference, any object that is compatible with that reference can be passed in, allowing a method to accept parameters of different data types. Thus polymorphic behaviour

Casting the References

- Suppose we instantiate two Salary objects as follows:
- **Salary s = new Salary("George Washington",**
- **"Valley Forge, DE", 1, 5000.00);**
- **Employee e = new Salary("Rich Raposa", "Rapid City, SD", 47, 250000.00);**
- I want to emphasize that these two statements create *two Salary objects; each* object consumes the same amount of memory and has the same methods and fields allocated in memory.
- The only difference between these two objects is the particular data stored in their respective fields.
- Because s is a Salary reference, we can use s to invoke the accessible fields and methods of both the Salary and Employee class. For example, the following statements are valid:
- **s.setSalary(100000.00); //A Salary method**
- **s.computePay(); //A Salary method**
- **s.mailCheck(); //An Employee method**

Casting in References

- Because e is an Employee reference, we can use e to only invoke the accessible methods of the Employee class.
- For example, we can use e to invoke mailCheck(),but we cannot use e to invoke setSalary() or computePay():
- **e.setSalary(500000.00); //Does not compile!**
- **e.computePay(); //Does not compile!**
- **e.mailCheck(); //An Employee method, so this compiles**
- The Salary object referenced by e has a setSalary() and a mailCheck() method; however, the compiler thinks e refers to an Employee, and attempting to invoke setSalary() or mailCheck() generates a compiler error.
- We need to use the cast operator on e, casting e to a Salary reference, before the Salary methods can be invoked using e. The following statements demonstrate two techniques for casting e to a Salary reference:
- **((Salary) e).computePay();**
- **Salary f = (Salary) e;**
- **f.computePay();**

OOM & Java Implementations

- **OO Design & Developing S/W Package**
- **Abstraction & Interface in Java**
  - Why & How of Abstraction in Java
- **Class, Abstract Class & Interface**
  - Similarities & Differences
- **Polymorphism in Java: Applications**
  - Programming Constructs & Application
- **Java APIs & GUI Implementation**
  - GUI Development & Applets

Inheritance in OOM

- Inheritance is one of the most significant characteristic in OOM, which has many effects.
- Two important effects that stem from inheritance are:
  - **Polymorphism**—where an object can take on **many** forms.
    - Provides useful programming constructs.
  - **Abstraction**—allowing for the creation of **abstract** classes.
    - Provides use of Interfaces in Java.